Time-Dependent Contraction Hierarchies and Approximation[⋆]

Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
`{batz,geisberger,sanders}@kit.edu`

Abstract. Time-dependent Contraction Hierarchies provide fast and exact route planning for time-dependent large scale road networks but need lots of space. We solve this problem by the careful use of approximations of piecewise linear functions. This way we need about an order of magnitude less space while preserving exactness and accepting only a little slow down. Moreover, we use these approximations to compute an exact travel time profile for an entire day very efficiently. In a German road network, e.g., we compute exact time-dependent routes in less than 2 ms. Exact travel time profiles need about 33 ms and about 3 ms suffice for an inexact travel time profile that is just 1 % away from the exact result. In particular, time-dependent routing and travel time profiles are now within easy reach of web servers with massive request traffic.

1 Introduction

In recent years, there has been considerable work on routing in road networks (see [1] for an overview). For the special case of constant edge weights (usually highly correlated with travel time) it is now possible to compute optimal paths orders of magnitude faster than with Dijkstra's algorithm. Such algorithms are now in wide-spread use in server based route-planning systems. There, 10 ms query time are acceptable but decreasing this to 1 ms is still desirable.

Recently, the more realistic *time-dependent* edge weights, which can model congestions during rush-hour and similar effects, have gained considerable interest. For example, time-dependent contraction hierarchies [2, 3] can compute optimal earliest arrival[1] (EA) routes in a German road network with midweek-traffic in about a millisecond. However, it requires a lot of space – too much for current low cost servers. Moreover, in a time-dependent setting, we may not only be interested in the best route for a given departure time, but also in a travel time profile over a long interval of potential departure times, e.g., in order to choose a good departure time. We are not aware of previous solutions that allow such profile queries in time suitable for current systems. In this paper, we address

[⋆] Partially supported by DFG grant SA 933/5-1 and the 'Concept for the Future' of KIT within the framework of the German Excellence Initiative. The original publication is available at www.springerlink.com.

[1] For start, destination, and departure time compute the earliest possible arrival time.

both issues: the *reduction of space requirements* and the *efficient computation of travel time profiles*. It turns out that the key to the solution of *both* problems is to approximate the piecewise linear functions used to describe time-dependent edge weights. Interestingly, this can be done without sacrificing exactness.

Our Contributions in More Detail. Time-dependent contraction hierarchies (TCHs) make intense use of *shortcut* edges. The time-dependent edge weights of the shortcuts contain lots of redundant information. This is where we attack.

We reduce the memory usage of TCHs greatly while accepting only a moderate slowdown of the runtime for the EA problem. Although we (partly) use approximated data, the result of our computation is still exact. The main idea behind this is that shortcuts get *approximated* and non-shortcuts get *exact* time-dependent edge weights. A bidirectional search in such an *approximated TCH* (ATCH) then yields a *corridor* of shortcuts. After unpacking these shortcuts, we can perform a time-dependent search in the *unpacked corridor* (Section 3.1).

TCHs can be used to compute exact travel time profiles in a straightforward but expensive way. However, computing a corridor of shortcuts based on upper and lower bounds first brings better runtimes. Still, the result of our computation remains exact (Section 3.2). But exact computations of this kind are also possible with the space saving ATCHs: We again compute a corridor of shortcuts based on upper and lower bounds. Now, we unpack this corridor completely. However, a profile search in the unpacked corridor is straightforward but slow. Performing a *corridor contraction* instead yields *very good* performance (Section 3.3).

Our techniques provide an accuracy that may not be necessary at all in practice. Using *solely* approximated edge weights yields only a small error but saves lots of memory and provides nearly full runtime performance for the EA problem. Moreover, accepting a small error when computing travel time profiles we get a great speedup compared to the exact computation (Section 3.4).

We have implemented all of the above techniques and performed several experiments to support our claims (see Section 4).

More Related Work. Contraction hierarchies [4] are the basis of TCHs. Time-dependent route planning itself started with classical results (e.g. [5]) showing that a generalization of Dijkstra's unidirectional algorithm works for time-dependent networks and that a small modification yields a (fairly expensive) means of profile search. Some TomTom car navigation systems allow a kind of time-dependent routing. However, the method used is unpublished and probably not able to guarantee optimal routes. A successful approach to fast EA routing is to combine a simpler form of contraction with goal directed techniques [6–8]. In particular, a combination with the arc flag technique (TD-SHARC [7]) yields good speedups, yet has problems when time-dependence is strong – then, either preprocessing becomes prohibitive or loses so much precision that query times get fairly high. However, for *inexact* EA queries it runs very fast (though preprocessing takes fairly long) [7, 8]. A combination with landmark A^* (ALT) works surprisingly well (TD-CALT [6]). We take this as an indication, that a combination with ALT could further improve the performance of TCHs.

2 Preliminaries

2.1 Time-Dependent Road Networks

Given a directed graph $G = (V, E)$ representing a road network[2]. Each edge $(u, v) \in E$ has a function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ assigned as edge weight. This function f specifies the time $f(\tau)$ needed to reach v from u via edge (u, v) when starting at *departure time* τ. Such edge weights are called *travel time functions* (TTFs).

In road networks we usually do not arrive earlier when we start later. This is reflected by the fact, that all TTFs f fulfill the *FIFO-property*: $\forall \tau' > \tau$: $\tau' + f(\tau') \geq \tau + f(\tau)$. In this work all TTFs are piecewise linear functions.[3] With $|f|$ we denote the *complexity* (i.e., the number of points) of f.

For TTFs we need the three operations: (1) *Evaluation*: Given a TTF f and a departure time τ we want to compute $f(\tau)$. Using a bucket structure this runs in constant average time. (2) *Linking*: Given two adjacent edges $(u, v), (v, w)$ with TTFs f, g we want to compute the TTF of the whole path $\langle u \to_f v \to_g w\rangle$. This is the TTF $g * f : \tau \mapsto g(f(\tau) + \tau) + f(\tau)$ (meaning g "after" f). It can be computed in $\mathrm{O}(|f| + |g|)$ time and $|g * f| \in \mathrm{O}(|f| + |g|)$ holds. Linking is an associative operation, i.e., $f * (g * h) = (f * g) * h$ for TTFs f, g, h. (3) *Minimum*: Given two parallel edges e, e' from u to v with TTFs f, f', we want to *merge* these edges into one while preserving all shortest paths. The resulting single edge e'' from u to v gets the TTF $\min(f, f')$ defined by $\tau \mapsto \min\{f(\tau), f'(\tau)\}$. It can be computed in $\mathrm{O}(|f| + |f'|)$ time and $|\min(f, f')| \in \mathrm{O}(|f| + |f'|)$ holds.

In a time-dependent road network, shortest paths depend on the departure time. For fixed start and destination nodes s and t and different departure times there might be different shortest paths with different arrival times. The minimal travel times from s to t for all departure times τ also form a TTF which we call the *travel time profile* (TTP) from s to t. Each TTF f implicitly defines an *arrival time function* $\mathrm{arr}f : \tau \mapsto f(\tau) + \tau$ that yields the arrival time for a given departure time. Analogously, the *departure time function* $\mathrm{dep}f := (\mathrm{arr}f)^{-1}$ yields the departure time for a given arrival time – provided that $\mathrm{arr}f$ is a one-to-one mapping. Otherwise, $\mathrm{dep}f(\tau)$ is the set of *possible* departure times.

2.2 Algorithmic Ingredients

In addition to TCHs three known modifications of Dijkstra's algorithm are crucial for this work: *Time-dependent Dijkstra* and and *profile search* are well known. *Interval search* has already been used in the precomputation of TCHs [2].

Time-Dependent Dijkstra. The time-dependent version of Dijkstra's algorithm solves the EA problem. It works exactly like the original except for the relaxation of edges (u, v) with TTFs f_{uv}. Let the label of node u be $d_s(u)$. The old label $d_s(v)$ of the node v is updated by $\min\{d_s(v), \mathrm{arr}f_{uv}(d_s(u))\}$. The initial node label of the start node is the departure time instead of 0.

[2] Nodes represent junctions and edges represent road segments.

[3] Here, all TTFs have period 24 h. Using non-periodic TTFs makes no real difference.

Profile Search. A label correcting modification of Dijkstra's algorithm. It computes the TTPs of all reached nodes for a given start node. Thus, node labels are TTPs. The initial node label of the start node is the TTP which is constant 0. We relax an edge (u, v) with TTF f_{uv} as follows: If f_u is the label of node u, we update the label f_v of node v by computing the minimum TTP $\min(f_v,\, f_{uv} * f_u)$.

Interval Search. Profile search is a very expensive algorithm. *Interval search* runs much faster with a runtime similar to Dijkstra's algorithm. Instead of TTPs it computes intervals containing all possible arrival times. So, the labels are intervals $[a, b] \subset \mathbb{R}_{\geq 0}$. The initial label of the start node is $[0, 0]$. We relax an edge (u, v) with TTF f_{uv} as follows: If $[a_u, b_u]$ is the label of node u, we update the label $[a_v, b_v]$ of node v with $[\min\{a_v, a_u + \min f_{uv}\}, \min\{b_v, b_u + \max f_{uv}\}]$.

Corridors. Given a start node s and a destination node t. A subgraph C of G containing s and t where t is reachable from s is a *corridor*. Corridors help to speed up profile searches very much: The expensive profile search is performed only in the previously computed corridor (as applied very successfully in the precomputation of TCHs [2]). Corridors can also be used to enable exact computations in the presence of approximated data (see Section 3).

TCHs. In a *time-dependent contraction hierarchy* [2] all nodes of G are *ordered* by increasing *importance*. The TCH (as a structure) 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. The shortest path distances are preserved by introducing *shortcut edges* when necessary. This way we construct the next higher *level* of the hierarchy from the current one. The node ordering and the construction of the TCH are performed in a precomputation.

EA Queries on TCHs. To answer *EA queries* given by users we first perform a bidirectional search in the TCH. The forward search is a time-dependent Dijkstra, the backward search is an interval search. Both searches go *upward* – meaning that only edges leading to more important nodes are used. The meeting points of the searches are called *candidate* nodes. The final step is the *downward search*: a time-dependent Dijkstra that only uses edges touched by the backward search. It starts from the candidate nodes where the arrival times computed by the forward search are used as initial node labels. During the bidirectional search we perform *stall-on-demand* [4, 2]: The search stops at nodes when we already found a better route coming from a higher level.

Unpacking Time-Dependent Shortcuts. A TCH contains two kinds of edges: *shortcut edges* representing paths $\langle u \rightarrow v \rightarrow w\rangle$ and *original edges* stemming from the original road network. Usually a shortest path computed by TCHs contains shortcuts which have to be *unpacked.* As the path represented by a shortcut may again contain shortcuts, we do this in a recursive manner. In time-dependent case a shortcut might represent different paths for different departure times. For some departure times it might even represent an original edge.

3 Applying Approximation

Approximation helps to save memory and to speed up computations. To save memory we use an approximated version of the TCH structure.

Approximated TCHs. An *approximated TCH* (ATCH) with relative error $\varepsilon \in [0,1]$ arises from a given TCH as follows: For all edges that represent an original edge for at least one departure time nothing happens. For all other edges the TTF f is replaced by an *upper bound* $f^{\uparrow}$ with $\forall\tau : f(\tau) \leq f^{\uparrow}(\tau) \leq (1+\varepsilon)f(\tau)$. Implicitly, $f^{\uparrow}$ also represents a *lower bound* $f^{\downarrow} : \tau \mapsto f^{\uparrow}(\tau)/(1+\varepsilon)$. For edges e with *exact* TTF f_e we have $f_e^{\uparrow} = f_e^{\downarrow} = f_e$. Usually $|f^{\uparrow}|$ is considerably smaller than $|f|$. Thus, an ATCH needs considerably less memory than the respective TCH (see Section 4). To compute $f^{\uparrow}$ from an exact TTF f we use an implementation (see Neubauer [9]) of an efficient geometric algorithm described by Imai and Iri [10]. It yields an $f^{\uparrow}$ of minimal $|f^{\uparrow}|$ for ε in time $\mathrm{O}(|f|)$.

Min-Max-TCHs. An extreme case of an approximated TCH is a *Min-Max-TCH.* For a shortcut, that never represents an original edge for any departure time, we only store the pair of numbers $(\min f, \max f)$ instead of the TTF f. Min-Max-TCHs need even less memory than ATCHs (see Section 4).

3.1 Exact Earliest Arrival Queries with Approximated TCHs

Our method for exact EA queries with ATCHs uses the two following new algorithmic ingredients. For their correctness the FIFO-property is required.

Arrival Interval Search. Is similar to interval search and computes an approximated solution of the EA problem, i.e., an interval containing the exact value, which is the best we can do for ATCHs. We relax an edge (u,v) with upper bound $f_{uv}^{\uparrow}$ and lower bound $f_{uv}^{\downarrow}$ as follows: If $[a_u, b_u]$ is the label of u, we update the label $[a_v, b_v]$ of v with $[\min\{a_v, a\}, \min\{b_v, b\}]$ where $[a,b] := [\mathrm{arr} f_{uv}^{\downarrow}(a_u), \mathrm{arr} f_{uv}^{\uparrow}(b_u)]$. The initial label of the start node is $[\tau_0, \tau_0]$ for the departure time τ_0.

Backward Travel Time Interval Search. Is dual to arrival interval search. Given a destination node t and an interval $[\sigma, \sigma']$ it computes intervals containing the possible times needed for traveling to t if the arrival time lies in $[\sigma, \sigma']$. The algorithm runs *backward* starting from t with $[0,0]$ as initial node label. Consider the (backward) relaxation of an edge (u,v) with upper bound $f_{uv}^{\uparrow}$ and lower bound $f^{\downarrow}_{uv}$. Let the label of node v be $[p_v, q_v]$. The old label $[p_u, q_u]$ of node u is updated with $[\min\{p_u, p\}, \min\{q_u, q\}]$ where $[p, q] := [p_v + \min f^{\downarrow}|_I, q_v + \max f^{\uparrow}|_I]$ and $I := [\max \mathrm{dep} f^{\uparrow}_{uv}(\sigma - q_v), \min \mathrm{dep} f^{\downarrow}_{uv}(\sigma' - p_v)]$.

Queries. Having defined all necessary ingredients, we are able to specify an algorithm for *exact* EA queries on ATCHs and Min-Max-TCHs: Given a start node s, a destination node t, and a departure time τ_0 proceed as follows:

– *Phase 1 (bidirectional upward search).* Perform a bidirectional search using solely *upward* edges with stall-on-demand. The forward search is an arrival interval search from s with initial label $[\tau_0, \tau_0]$ that computes intervals containing *arrival times.* The backward search is an interval search from t with initial label $[0, 0]$ that computes intervals containing *travel times.* The meeting points of the searches are *candidate* nodes. For a candidate node c with forward label $[\tau, \tau']$ and backward label $[\sigma, \sigma']$ the interval $[\tau + \sigma, \tau' + \sigma']$ contains an arrival time for traveling from s to t via c for departure time τ_0.
– *Phase 2 (forward/downward search).* Perform a forward arrival interval search starting from the candidates, that only uses edges touched by the backward search of Phase 1. The initial node labels of the candidates are the arrival time intervals computed by the forward search in Phase 1.
– *Phase 3 (backward/upward search).* Phase 2 yields an interval $[a_t, b_t]$ containing the EA time for t. Now, we perform a backward travel time interval search starting from t with initial label $[a_t, b_t]$. The search runs *backward* and uses only *upward* edges touched by Phase 2. When we reach a node, that has also been reached by the forward search of Phase 1, the node is again a candidate.
– *Phase 4 (unpacking and Dijkstra).* From the candidates provided by Phase 3 perform a forward and a backward BFS on the edges touched by Phases 3 and 1 respectively. Unpacking all shortcuts touched by these BFSs yields a corridor C whose edges have only exact TTFs. A time-dependent Dijkstra in C from s with departure time τ_0 yields the sought-after exact arrival time.

As C contains rather few edges, the time-dependent Dijkstra in Phase 4 does not need much time. As a result, the runtimes are only moderately worse than with exact TCHs. Note, that we could perform Phase 4 directly after Phase 1. But the Phases 2 to 3 help to reduce the candidate set and thus the corridor. An improvement to Phase 4 is to unpack the shortcuts only when they are needed by the time-dependent Dijkstra – this is to say “on demand”.

3.2 Exact Profile Queries with Exact TCHs

Computing TTPs using exact TCHs is straightforward: For a start node s and a destination node t just perform a bidirectional profile search, that only uses upward edges while using stall-on-demand based on global minima and maxima. Again, the meeting points of the bidirectional search are *candidate* nodes, each of them representing a TTP (though not necessary an optimal one). Now, merge all these TTPs using the minimum operation, which yields the sought-after TTP.

As this is quite time consuming, we propose a great improvement: Perform a bidirectional upward interval search first. Again, the meeting points are candidate nodes. Similar to Phase 4 in Section 3.1 perform a forward and a backward BFS starting from the candidates – but do *not* unpack the shortcuts (all edges have exact TTFs this time). Now, perform the bidirectional upward profile search only in the resulting corridor C, which makes the search strongly directed. Again, merge the candidate TTPs, which yields the sought-after TTP.

3.3 Exact Profile Queries with ATCHs

Exact profile queries can also be answered using ATCHs. This is possible by adapting the method described in Section 3.2 in a straightforward way: Unpack all shortcuts in the corridor C and perform profile search in the resulting corridor C'. However, this takes some time. The reason is, that C' usually contains many more edges than C. During a profile search the points of the TTFs of these edges are processed again and again and again. Assume, for example, that C' is a path of ℓ edges and that all TTFs have k points. Then, a profile search in C' processes $\Theta(k\ell^2)$ points in the worst case. Here, we propose *corridor contraction* – a much faster algorithm reducing this to $\Theta(k\ell \log \ell)$ points. It exploits the fact that linking is an *associative* operation on TTFs, which enables us to alter the order of link operations without altering the result.

Corridor Contraction. Our algorithm uses a priority queue (PQ) to control the order of performed operations. The elements of the PQ are nodes. As key we use the estimated effort needed to contract each node. First, we insert all nodes in C' except for s, t into the PQ. Then, we *contract* C': While the PQ is not empty, we delete the minimal node v from the PQ and contract it completely. That is, for all paths $\langle u \rightarrow_f v \rightarrow_g w \rangle$ we add an edge (u, w) to C' with TTF $h := g * f$ and remove all edges incident to v from C'. Also we update the keys of u and w. If an edge (u, w) already exists in C', we merge its TTF with h. After termination only an edge (s, t) is left in C'. Its TTF is the sought-after TTP.

As an optimization we thin out C' by a preceding bidirectional approximate upward profile search which computes approximate TTPs that are upper and lower bounds of the exact ones. This makes our method even faster than the one in Section 3.2. However, for Min-Max-TCHs this is not possible of course.

3.4 Inexact Earliest Arrival and Profile Queries

In practice exact results may be not necessary, or the accuracy of the TTFs may be arguable. In such cases, small errors are allowed and all computations can be performed using an *inexact TCH*, which can be obtained from an exact TCH by replacing every TTF f by an inexact TTF $f^{\updownarrow}$ with $(1+\varepsilon)^{-1}f(\tau) \leq f^{\updownarrow}(\tau) \leq (1+\varepsilon)f(\tau)$. Additionally, we store the *conservative bounds* $\min\{\min f, \min f^{\updownarrow}\}$ and

Table 1. Behaviour of EA queries using different methods. ATCH with $\varepsilon = \infty$ denotes Min-Max-TCHs. TCH with $\varepsilon \neq 0$ denotes inexact queries on inexact TCHs, cor.= corridor, UoD= shortcut **U**npacking-**o**n-**D**emand, SPD= speedup of time-dependent Dijkstra, GRO= growth of space usage compared to the original graph, MAX and AVG are maximum and average relative errors.

	ε	space		time		delMin		edges		evals		error [%]	
method	[%]	[B/n]	GRO	[ms]	SPD	#	SPD	#	SPD	#	SPD	MAX	AVG
Germany midweek													
TCH	–	994	10.4	0.72	1 440	520	4 616	5 813	951	1 269	162	0.00	0.00
TCH (cor.)	0.0	994	10.4	0.74	1 401	639	3 756	7 092	780	76	2 704	0.00	0.00
	0.1	286	3.0	0.71	1 460	642	3 739	7 128	770	77	2 669	0.10	0.02
	1.0	214	2.3	0.72	1 440	654	3 670	7 262	762	84	2 446	1.01	0.27
	10.0	113	1.2	1.03	1 006	897	2 676	10 096	548	223	921	9.75	3.84
ATCH (UoD)	0.1	308	3.2	1.10	942	554	4 332	7 734	715	3 080	67	0.00	0.00
	1.0	239	2.5	1.27	816	582	4 124	8 338	664	3 347	61	0.00	0.00
	10.0	163	1.7	2.40	432	824	2 913	21 036	263	7 486	27	0.00	0.00
	∞	118	1.2	1.45	714	698	3 439	20 116	275	3 153	65	0.00	0.00
Europe high traffic													
TCH	–	589	7.8	1.89	1 807	986	9 161	13 003	1 665	2 370	289	0.00	0.00
TCH (cor.)	0.0	589	7.8	3.19	1 071	1 653	5 464	23 031	929	1 412	484	0.00	0.00
	0.1	237	3.1	3.67	931	1 661	5 438	23 142	924	1 427	479	0.14	0.02
	1.0	193	2.5	2.85	1 199	1 716	5 264	24 036	890	1 544	443	1.46	0.20
	10.0	143	1.9	2.68	1 275	1 726	5 233	24 221	883	1 583	432	15.34	2.85
ATCH (UoD)	0.1	256	3.4	2.25	1 518	1 032	8 752	17 894	1 195	5 382	127	0.00	0.00
	1.0	207	2.7	2.47	1 396	1 104	8 152	22 683	943	6 362	108	0.00	0.00
	10.0	164	2.2	7.37	463	1 771	5 100	137 221	156	23 949	29	0.00	0.00
	∞	99	1.3	15.43	221	2 196	4 113	448 360	48	42 939	16	0.00	0.00

$\max\{\max f, \max f^{\updownarrow}\}$ with every edge. Inexact TCHs save lots of memory. But when we compute TTPs they even gain an enormous speedup. This is because the processed TTFs have much less points.

To preserve correctness we always perform a preceding bidirectional upward interval search that employs stall-on-demand using only the conservative bounds. All further passes of any query avoid stall-on-demand and only relax edges touched by this initial phase. For EA queries we have two further passes: the forward and the downward search as described in Section 2.2 (without backward search this time). In this way we get a corridor-based variant of the original TCH query which also works with exact TCHs. For profile queries the only further pass is a bidirectional upward profile search. So, we actually have the method from Section 3.2, but this time applied to inexact TCHs.

4 Experiments

Inputs and Setup. As inputs we use two road networks of Germany and Western Europe, both provided by PTV AG for scientific use. Germany has 4.7 million nodes, 10.8 million edges, and time-dependent edge weights reflecting

Table 2. Profile queries using different methods. CC= corridor contraction, the rest of the nomenclature is the same as in Table 1.

	ε	space		time	delMin	edges	points	error [%]	
method	[%]	[B/n]	GRO	[ms]	#	#	#	MAX	AVG
Germany midweek									
TCH	–	994	10.4	1 112.02	570	6 796	20 623 155	0.00	0.00
TCH (cor.)	0.0	994	10.4	88.87	646	7 170	1 437 892	0.00	0.00
	0.1	286	3.0	6.13	650	7 208	86 391	0.10	0.02
	1.0	214	2.3	2.94	662	7 348	35 769	1.03	0.27
	10.0	113	1.2	2.48	923	10 361	23 010	9.69	3.84
ATCH (CC)	0.1	308	3.2	36.22	650	29 551	576 099	0.00	0.00
	1.0	239	2.5	32.75	675	32 131	531 795	0.00	0.00
	10.0	163	1.7	105.45	889	92 740	1 731 359	0.00	0.00
	∞	118	1.2	76.58	578	59 368	1 278 095	0.00	0.00
Europe high traffic									
TCH	–	589	7.8	4182.43	1 090	17 234	70 937 950	0.00	0.00
TCH (cor.)	0.0	589	7.8	2 016.86	1 797	25 486	30 734 960	0.00	0.00
	0.1	237	3.1	198.00	1 813	25 655	3 371 555	0.13	0.02
	1.0	193	2.5	105.72	1 882	26 796	1 741 315	1.27	0.20
	10.0	143	1.9	36.75	1 889	26 977	755 646	14.65	2.85
ATCH (CC)	0.1	256	3.4	565.28	1 806	169 378	8 200 162	0.00	0.00
	1.0	207	2.7	382.12	1 887	199 551	5 448 190	0.00	0.00
	10.0	164	2.2	2 306.11	2 429	1 259 891	35 330 837	0.00	0.00

the midweek (Tuesday till Thursday) traffic collected from historical data, i.e., a high traffic scenario with about 8 % time dependent edges. Western Europe has about 18 million nodes and 42.6 million edges. It has been augmented with synthetic time-dependent travel times as in [11] using a high amount of traffic where all edges but local and rural roads have time-dependent edge weights.

The experimental evaluation was done on a machine with four Core i7 Quad-Cores (2.67 Ghz) with 48 GiB of RAM running SUSE Linux 11.1. All programs were compiled by GCC 4.3.2 with optimization level 3. Running times were always measured using one single thread. All figures refer to the scenario that only the EA times and the TTPs have to be determined, without outputting complete path descriptions. However, when reporting memory consumption, we include the space needed to allow fast path reporting. The memory usage is given in terms of the average *total* space usage of a node (not the overhead) in byte per node. We also report the growth factor of the memory usage compared to the *original graph*, i.e., the graph used for time-dependent Dijkstra. For Germany this graph needs 95 byte per node, for Europe 76 byte per node.

We measured the average performance of EA and profile queries for 1 000 randomly selected start and destination pairs. For EA queries the departure time is randomly selected from [0h, 24h) each. To measure the errors we used many more test cases: 1 000 000 EA queries and 10 000 profile queries, where the error of profile queries was measured for 100 random departure times each.

We also measured the machine-independent behaviour of our algorithms: In all cases we count the number of deleteMin-Operations and touched edges (which is identical to the number of relaxed edges for time-dependent Dijkstra). For EA queries we also count how often TTFs are evaluated (including similar operations like, e.g., computing $\max \operatorname{dep} f_{uv}^{\uparrow}(\sigma - q_v)$ in Section 3.1). For profile queries we count the points of the TTFs processed by link and minimum operations.

Results. Table 1 evaluates EA queries for different approaches. Inexact TCHs (Section 3.4) save space and are at most 2 times slower than exact TCHs. For Germany there is no slow down for small ε. The maximum errors are small for small ε, the average errors are even smaller. However, in theory one can easily construct inputs where errors could get much larger than ε. ATCHs (Section 3.1) run slower by a factor of 1.5–3.3 for Germany and 1.2–8.2 for Europe. However, ATCHs save memory: 3.2–8.4 times less space than exact TCHs for Germany and 2.3–6.0 for Europe. Note, that the speed difference would further decrease when shortest paths were to be computed since this is done anyway for ATCHs.

For profile queries look at Table 2. Exact TCHs, using the straightforward method (Section 3.2), take about 1.1 s for Germany and 4.2 s for Europe – far too slow for a server scenario. Restricting profile search to a corridor (also Section 3.2) helps, but ATCHs with corridor contraction (Section 3.3) work better. For Germany Min-Max-TCHs also work well. For Europe they do not show acceptable running times. However, for really fast profile queries we need inexact TCHs. This works especially well for Germany.

Figure 1 shows the distribution of running times of profile queries on Germany: For $i = 5..22$ we look at 100 queries with the property that Dijkstra's Algorithm settles 2^i nodes (2^i is called the *Dijkstra rank*). A profile query in a middle-sized German town (rank 2^{12}) needs less than 1 ms on ATCHs with corridor contraction. Plain profile search (Section 2.2) is much slower. We stopped when the average running time exceeded 10 s. In Section 3.3 we claim that corridor contraction brings a considerable additional speedup. Indeed, when we

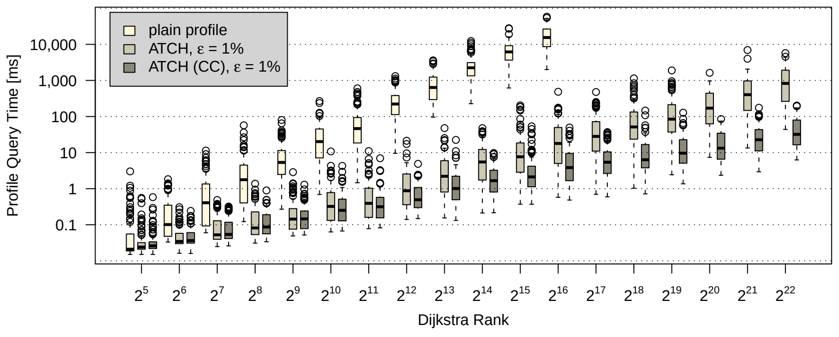

Fig. 1. Profile query times over the Dijkstra rank for different methods. CC means corridor contraction is used, otherwise a profile search in the corridor is performed.

Table 3. Comparison of different algorithms for exact and inexact time-dependent EA and profile (TTP) queries. Memory usage is given as overhead, errors are max. rel. errors. REL= relative speed of profile queries compared to time-dependent Dijkstra, SH= SHARC, inex= inexact, app= approximate, heu= heuristic, spc eff= space efficient.

method	ε [%]	prepro. [h:m]	ovh. [B/n]	EA SPD	TTP REL	err. [%]	prepro. [h:m]	ovh. [B/n]	EA SPD	TTP REL	err. [%]
		Germany midweek					Europe high traffic				
TCH	–	0:28+0:09	899	1 440	11.66	0.00	3:45+0:59	513	1 807	1.69	0.00
ATCH	1	0:28+0:09	144	816	31.65	0.00	3:45+0:59	131	1 396	9.94	0.00
ATCH	∞	0:28+0:09	23	714	13.49	0.00	3:45+0:59	23	221	–	0.00
CALT	–	0:09	50	280	–	0.00	1:00	61	47	–	0.00
SH	–	1:16	155	60	0.02	0.00	6:44	134	70	–	0.00
L-SH	–	1:18	219	238	–	0.00	6:49	198	150	–	0.00
inex TCH	1	0:28+0:09	119	1 440	352.59	1.03	3:45+0:59	117	1 199	32.31	1.46
inex TCH	10	0:28+0:09	18	1 006	417.99	9.75	3:45+0:59	67	1 275	92.95	15.34
app CALT	–	0:09	50	804	–	13.84	1:00	61	624	–	8.69
heu SH	–	3:26	137	2 164	1.40	0.61	22:12	127	1 958	–	1.60
heu L-SH	–	3:28	201	3 915	–	0.61	22:17	191	2 703	–	1.60
spc eff SH	–	3:48	68	1 177	–	0.61	–	–	–	–	–
spc eff SH	–	3:48	14	491	–	0.61	–	–	–	–	–

replaced corridor contraction by a profile search in the precomputed corridor it ran considerably slower (also Figure 1).

For the comparison with (the best) other techniques look at Table 3. For EA queries we only compare speedups of time-dependent Dijkstra – absolute query times would be unreliable as different machines are used. As plain profile search takes too long, we are not able to report speedups for profile queries. Instead, we also compare the running time of profile queries with time-dependent Dijkstra. This way we get a *relative speed*. As our preprocessing works in two phases (*node ordering* and *contraction*) there are preprocessing times like 0:28+0:09 for TCH based techniques (28 min node ordering and 9 min contraction). Node orders can be reused for different traffic scenarios, i.e., they need not to be recomputed. However, this might slow down the query time a bit [2].

For exact queries ATCHs dominate TD-SHARC [7] in all respects. However, TD-CALT [6, 7] has much better preprocessing time. For Europe the advantage of TCH based techniques over TD-CALT with respect to query time becomes much larger. This is an indication that TCH combined with ALT will not scale well with the input size. For inexact EA queries approximate TD-CALT works much better. But again, it is outperformed by inexact TCHs except for preprocessing. Heuristic TD-SHARC has better speedups for EA queries but worse memory usage and preprocessing times than inexact TCHs. For *space efficient* TD-SHARC [8] the memory usage is very good but the speedups are worse than for inexact TCHs. Regarding profile search, TCH based techniques come off as a clear winner as they run up to three orders of magnitude faster in the exact and about 250–300 times faster in the in the inexact setting.

5 Conclusions and Future Work

We have demonstrated that using approximations of travel time functions greatly reduces the space consumption of time-dependent contraction hierarchies. By using these approximation only for obtaining a corridor of possibly useful roads, we can still obtain exact results. We have also explained how travel time profiles can be computed very efficiently – fast enough for current server systems.

The achieved space reduction by up to an order of magnitude may not be the end of the story because we can possibly come up with much more compact representations of the approximations than the piecewise linear functions currently used. It might be possible to represent the TTFs with some tabulated patterns and then just store references and scaling factors. This way (near exact) time-dependent route planning may even be possible on mobile devices.

Future work will have to allow even more realistic modelling, in particular, incorporating traffic jams, and allowing additional objective functions.

Acknowledgements. We thank Christian Vetter for important contributions to the TCH code. There were many fruitful discussions with Daniel Delling.

References

1. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In Lerner, J., Wagner, D., Zweig, K.A., eds.: Algorithmics of Large and Complex Networks. Volume 5515 of LNCS. Springer (2009) 117–139
2. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-Dependent Contraction Hierarchies. In: Finocchi, I., Hershberger, J., eds.: ALENEX'09. SIAM (2009)
3. Vetter, C.: Parallel Time-Dependent Contraction Hierarchies (2009) Student Research Project. `http://algo2.iti.kit.edu/download/vetter_sa.pdf`
4. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. [12] 319–333
5. Orda, A., Rom, R.: Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. Journal of the ACM **37**(3) (1990) 607–625
6. Delling, D., Nannicini, G.: Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In Hong, S.H., Nagamochi, H., Fukunaga, T., eds.: ISAAC'08. Volume 5369 of LNCS., Springer (2008) 813–824
7. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica (2009) Special Issue: European Symposium on Algorithms 2008.
8. Brunel, E., Delling, D., Gemsa, A., Wagner, D.: Space-Efficient SHARC-Routing. In: SEA'10. Volume 6049 of LNCS., Springer (2010).
9. Neubauer, S.: Space Efficient Approximation of Piecewise Linear Functions (2009) Student Research Project. `http://algo2.iti.kit.edu/download/neuba_sa.pdf`
10. Imai, H., Iri, M.: An optimal algorithm for approximating a piecewise linear function. Journal of Information Processing **9**(3) (1987) 159–162
11. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A* Search for Time-Dependent Fast Paths. [12] 334–346
12. McGeoch, C.C., ed.: WEA'08. Volume 5038 of LNCS., Springer (2008)