Minimum Time-Dependent Travel Times with Contraction Hierarchies

G. VEIT BATZ, ROBERT GEISBERGER, PETER SANDERS,
and CHRISTIAN VETTER, Karlsruhe Institute of Technology (KIT)

Time-dependent road networks are represented as weighted graphs, where the weight of an edge depends on the time one passes through that edge. This way, we can model periodic congestions during rush hour and similar effects. In this work we deal with the special case where edge weights are time-dependent travel times. Namely, we consider two problems in this setting: Earliest arrival queries ask for a minimum travel time route for a start and a destination depending on a given departure time. Travel time profile queries ask for the travel time profile for a start, a destination, and an interval of possible departure times. For an instance representing the German road network, for example, we can answer earliest arrival queries in less than 1.5ms. For travel time profile queries, which are much harder to answer, we need less than 40ms if the interval of possible departure times has a width of 24 hours. For inexact travel time profiles with an allowed error of about 1% this even reduces to 3.2ms. The underlying hierarchical representations of the road network, which are variants of a time-dependent contraction hierarchy (TCH), need less than 1GiB of space and can be generated in about 30 minutes. As far as we know, TCHs are currently the only method being able to answer travel time profile queries efficiently. Altogether, with TCHs, web servers with massive request traffic are able to provide fast time-dependent earliest arrival route planning and computation of travel time profiles.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Earliest arrival problem, road network, server-based route planning, shortest path, speed-up technique, travel time profile

ACM Reference Format:
Batz, G. V., Geisberger, R., Sanders, P., and Vetter, C. 2013. Minimum time-dependent travel times with contraction hierarchies. ACM J. Exp. Algor. 18, 1, Article 1.4 (April 2013), 43 pages.
DOI: http://dx.doi.org/10.1145/2444016.2444020

1. INTRODUCTION

In recent years, there has been considerable work on routing in road networks (see Delling et al. [2009] for an overview). For the special case of constant edge weights (usually highly correlated with travel time), current methods can compute optimal paths orders of magnitude faster than Dijkstra's well-known algorithm. Such techniques are now in widespread use in server-based route-planning systems. It is important to know that Dijkstra's algorithm [Dijkstra 1959] needs several seconds to compute an optimal path in a continental size road network. In practice, this is much

This work was partially supported by DFG grant SA 933/5-1 as well as by the Concept for the Future of KIT within the framework of the German Excellence Initiative. Note that important parts of this work appeared as preliminary versions at the Workshop on Algorithm Engineering and Experiments (ALENEX '09) and the Symposium on Experimental Algorithms (SEA '10).
Author's addresses: Department of Informatics, Karlsruhe Institute of Technology, Box 6980, 76128 Karlsruhe, Germany; email: {batz, sanders}@kit.edu.

DOI: http://dx.doi.org/10.1145/2444016.2444020

to slow. Therefore, server-based route planning would not be possible without today's more advanced algorithms.

Recently, the more realistic time-dependent scenario, which can model periodic congestions during rush hour and similar effects, has gained considerable interest: The edge weights are no longer constant but depend on time. Note that time-dependent edge weights can model different kinds of costs. Here, we only deal with the special case where edge weights are time-dependent travel times. More general time-dependent costs are much harder to handle. Even the seemingly trivial generalization that edges have an additional constant nonnegative cost makes the problem NP-hard [Batz and Sanders 2012]. In this article, we deal with two kinds of time-dependent route planning queries: For a given pair of start and destination as well as a given departure time, we want to compute the earliest possible arrival time and a corresponding optimal route—a so-called *earliest arrival* (EA) *query*. Also, we may not only be interested in the best route for a fixed 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). Computing a travel time profile for a given pair of start and destination with a given departure time interval is called a *travel time profile query*.

In this article, we describe methods to answer both kinds of queries efficiently: For a German road network, we can answer EA queries in less than 1.5ms where the underlying representation of the road network needs less than 1GiB. In the same setting, we need less than 40ms for profile queries if the interval of possible departure times spans 24 hours. For inexact profile queries with an allowed error of about 1% even 3.2ms suffice. To achieve this performance, we use variants of a *time-dependent contraction hierarchy* (TCH), which is a special hierarchical representation of the road network. Such TCHs are generated offline in a computationally intensive preprocessing step. However, the running time of the preprocessing is tolerable and can be parallelized well. For our German road network, the preprocessing requires about 30min sequential running time and about 5min with eight parallel threads. Altogether, with TCHs, Web servers with massive request traffic are able to answer both EA and profile queries virtually at once. Other than these TCHs, we are not aware of any other method to answer profile queries efficiently. For EA queries, there are some other efficient techniques [Delling and Nannicini 2008; Delling 2011; Brunel et al. 2010].

Important parts of this work have been published as preliminary versions [Batz et al. 2009; Batz et al. 2010], and there is also an early technical report [Batz et al. 2008]. However, here, we present several relatively detailed proofs of correctness, which have not been published yet. Also, we describe in detail how the preprocessing works, which we have not done before. To parallelize the preprocessing, we use independent node sets. Although this idea has originally been developed and implemented for the work we describe here, it has also been used in the context of a distributed memory version of TCHs and has thus been mentioned in the corresponding publication [Kieritz et al. 2010].

1.1. Static Contraction Hierarchies and Shortcuts

Originally, *contraction hierarchies* (CHs) have been invented to speed up time-*in*dependent routing (static CHs [Geisberger et al. 2008, 2012]). There, a road network is transformed into a hierarchical representation by contracting all its nodes one after another according to increasing importance for routing. Contracting a node x means that x is conceptually removed from the road network. In doing so, we insert several shortcut edges between the neighbors of x. This way, the shortest distances in the remaining graph are preserved. Having contracted all nodes we get a hierarchical representation (the actual CH structure). It has the following very useful property: For every pair of nodes s and t (where t is reachable from s), there

exists a *shortest up-down-path* from s to t in the CH. Such an up-down-path is a path $\langle s \to \cdots \to y \to \cdots \to t \rangle$ in the CH where $\langle s \to \cdots \to y \rangle$ only goes upward and $\langle y \to \cdots \to t \rangle$ only goes downward in the hierarchy. This enables the following very efficient routing procedure: Simply perform a bidirectional Dijkstra search starting from s and t in the CH where edges are only relaxed if they lead upward in the hierarchy. This runs very fast as well-constructed CHs are flat and sparse, so upward searches reach the top of the CH quite soon while the branching factor is not too large. Note that this method would not work correctly without the shortcuts edges inserted during contraction. They guarantee the existence of the shortest up-down-paths.

1.2. Our Contributions in More Detail

TCHs generalize CHs by allowing time-dependent travel times as edge weights. A reader who is unfamiliar with the details of time-dependent route planning might think that this is just a straightforward extension. This, however, is not the case, as time-dependency makes the whole problem much more complicated. In fact, we use several nontrivial algorithmic ingredients to make TCHs run efficiently. The most important idea in this work is to use approximate computation to find small subgraphs and then to perform the exact computation only on these subgraphs. This also enables exact computations in the presence of space-saving approximated data. Moreover, we can speed up exact profile queries very much this way.

Preprocessing. For TCHs, the preprocessing is much more complicated than for CHs and making it run in reasonable time requires several ideas. This includes cheap approximate shortest path searches to build up small corridors in which the far more expensive exact computations are done. Note that this yields an exact hierarchy, even though approximation and a heuristic are used for intermediate steps. Also, the preprocessing parallelizes quite well for shared memory (Section 4).

EA Queries with TCHs. The guaranteed existence of EA up-down-paths in TCHs ensures that bidirectional search can be used to find exact solutions quickly (Section 5.1). For EA queries on TCHs, however, we have the following problem: We cannot just perform a bidirectional time-dependent Dijkstra search as this would require us to know the earliest arrival time—but this is just the thing we want to compute. To overcome this, we perform an approximate shortest path search as a backward search. Having done that, we perform a time-dependent Dijkstra search that uses only edges touched by the backward search. This yields the sought-after earliest arrival time. Again, the result of this computation is exact (Section 5.2).

Profile Queries with TCHs. TCHs can also be used to answer profile queries in a straightforward and feasible way by simply applying bidirectional profile search (Section 5.3). But we can be much faster: To reduce the search space, we perform a bidirectional approximate search first. This brings a considerable speed-up, while the result of this computation is exact (Section 5.5).

Saving Memory and Speeding Up Profile Queries. TCHs—just like static CHs—make intense use of shortcut edges. But for TCHs this requires much more memory than for static CHs. The reason is that the time-dependent travel time information associated with the shortcuts contains a lot of redundancy. This is what we focus on: Shortcuts store approximated time-dependent edge weights and only original edges store exact ones. Unidirectional and bidirectional searches on such an approximated TCH (ATCH) yield small subgraphs that contain shortcuts (Section 6.1). As exact travel time information is available for nonshortcuts, we can apply exact time-dependent searches after unpacking the shortcuts. This way we reduce memory usage greatly while accepting

only a moderate slowdown for exact EA queries (Section 6.3). For exact profile queries, we even get a further speed-up by contracting whole corridors (Section 6.4).

Inexact Querying. Our techniques provide an accuracy that may be unnecessary in practice. Using solely approximated edge weights yields only a small error but again saves a lot of memory without any slowdown for the EA problem (Section 7.2). Moreover, accepting a small error when computing travel time profiles, we get a great speed-up compared to the exact computation (Section 7.1).

Experimental Evaluation. We have implemented the aforementioned techniques and performed several experiments to support our claims. As inputs, we use road networks of Europe and Germany with time-dependent travel times (Section 8).

1.3. Related Work

Time-dependent route planning started with classical results like a generalization of Dijkstra's algorithm [Dreyfus 1969] and an expensive means of profile search [Orda and Rom 1990]. Dean [1999] provided a basic introduction to the topic. 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 [Delling and Nannicini 2008; Delling 2011; Brunel et al. 2010]. In particular, with arc-flags (TD-SHARC [Delling 2011]) this yields good speed-ups. However, it has problems when time-dependency is strong, in which case either preprocessing becomes prohibitive or query times get fairly high. This can be compensated by combining with landmark A* (ALT [Goldberg and Harrelson 2005]), which yields TD-L-SHARC. For inexact EA queries, TD-SHARC runs very fast (though preprocessing takes a fairly long time) [Delling 2011; Brunel et al. 2010]. Simple contraction combined with ALT without arc-flags also works surprisingly well (TD-CALT [Delling and Nannicini 2008]). We take this as an indication that a combination of TCHs and ALT could further improve the performance. As stated before, TCH generation is computationally intensive but parallelizes well. This also works on distributed memory [Kieritz et al. 2010]. Several concepts discussed here have been applied to time-dependent many-to-many queries [Geisberger and Sanders 2010]. Recently, we described a heuristic version of TCHs that deals with the NP-hard set-up that time-dependent edge weights are travel times with additional constant nonnegative costs [Batz and Sanders 2012]. It shows good running times, and the computed routes are quite near to the optimum.

2. PRELIMINARIES

A road network is usually represented by a directed graph $G = (V, E)$, where nodes represent junctions and edges represent road segments for example.[1] Each edge $u \to v \in E$ has a *travel time function* (TTF) $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ assigned as edge weight—we write $u \to_f v$. The TTF f specifies the *travel time* $f(\tau)$ needed to reach v from u via edge $u \to v$ when starting at *departure time* τ. In road networks, we 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 continuous piecewise linear functions with a period of 24 hours.[2] With $|f|$, we denote the *complexity* (i.e., the number of segments) of f.

[1]This is a common interpretation, but it is not the only interpretation. Alternatively, nodes could represent road segments and edges could represent transitions between road segments. Turn costs can be modeled this way.

[2]Using nonperiodic TTFs would make no real difference for the techniques discussed in this work. In context of real-live applications, an appropriate time window with a sufficient large period should be chosen. This

For TTFs, we need three operations: (i) *Evaluation*: Given a TTF f and a departure time τ we want to compute $f(\tau)$. In order to support average-case constant evaluation time, we store a TTF f in an array of $|f|/4$ buckets, each of which is responsible for an equal-sized interval of departure times. Thus, we can efficiently locate the segment covering time τ by jumping to the right bucket and scanning an expected constant number of bucket entries. (ii) *Linking*: Given two adjacent edges $u \to_f v, v \to_g w$, we want to compute the TTF of the whole path $\langle u \to_f v \to_g w\rangle$, that is, the TTF $g*f : \tau \mapsto g(f(\tau)+\tau)+f(\tau)$ (meaning g "after" f). It can be computed in $O(|g|+|f|)$ time and $|g*f| \in O(|g|+|f|)$ holds. Note that linking is associative, that is, $f*(g*h) = (f*g)*h$ for TTFs f, g, h, but not commutative in general. (iii) *Minimum*: Given two parallel edges $u \to_g v$, $u \to_f v$, we want to merge these edges into one while preserving all shortest paths. The resulting single edge $u \to v$ gets the TTF $\min(g, f)$ defined by $\tau \mapsto \min\{g(\tau), f(\tau)\}$. It can be computed in $O(|g|+|f|)$ time and $|\min(g, f)| \in O(|g|+|f|)$ holds. Note that the set of piecewise linear TTFs with FIFO property and a period of 24 hours is closed under link and minimum operation.

Consider a path $P := \langle v_1 \to_{f_1} v_2 \to_{f_2} \cdots \to_{f_{k-1}} v_k\rangle$ in G. A subpath $\langle v_1 \to \cdots \to v_i\rangle$ of P is called a *prefix* and a subpath $\langle v_i \to \cdots \to v_k\rangle$ is called a *suffix* of P. In a time-dependent road network, the travel time from v_1 to v_k along path P depends on the departure time τ_0 and amounts to $f_P(\tau_0)$, where f_P is defined as $f_{k-1} * \cdots * f_1$. The corresponding arrival time is $f_P(\tau_0)+\tau_0$. Because of the FIFO property, there exists an *earliest arrival time* (EA time) for traveling from a node s to a node t with departure time τ_0 in G, namely, the time

$$\mathrm{EA}_G(s, t, \tau_0) := \min\left\{ f_Q(\tau_0)+\tau_0 \;\middle|\; Q \text{ is path from } s \text{ to } t \text{ in } G \right\} \cup \{\infty\} .$$

If $f_P(\tau_0)+\tau_0 = \mathrm{EA}_G(v_1, v_k, \tau_0)$ holds, then P is called an *earliest arrival (EA) path in G* or, to be more specific, an (v_1, v_k, τ_0)-EA-path in G. If a node t is not reachable from a node s in G, we have $\mathrm{EA}_G(s, t, \tau) = \infty$ for all $\tau \in \mathbb{R}$.

COROLLARY 2.1. *If $\langle v_1 \to \cdots \to v_i \to \cdots \to v_k\rangle$ is a (v_1, v_k, τ_0)-EA-path in G, then every suffix $\langle v_i \to \cdots \to v_k\rangle$ is a $(v_i, v_k, \mathrm{EA}_G(v_1, v_i, \tau_0))$-EA-path in G, and we have $\mathrm{EA}_G(v_1, v_k, \tau_0) = \mathrm{EA}_G(v_i, v_k, \mathrm{EA}_G(v_1, v_i, \tau_0))$.*

The analogous condition for prefixes does not hold in general, but a slightly weakened version of it does.

COROLLARY 2.2. *Let $s, t \in V$ be two nodes such that t is reachable from s in G and let $\tau_0 \in \mathbb{R}$ be a departure time. Then, there always exists at least one (s, t, τ_0)-EA-path $\langle s = v_1 \to v_2 \to \cdots \to v_k = t\rangle$ in G such that for every $i = 1, \ldots, k$, the prefix $\langle v_1 \to \cdots \to v_i\rangle$ is an (s, v_i, τ_0)-EA-path in G.*

Such EA paths are called *prefix-optimal* (s, t, τ_0)-EA-paths. The minimum travel times from s to t for all departure times τ form a TTF, namely

$$\mathrm{TTP}_G(s, t) : \; \tau \mapsto \mathrm{EA}_G(s, t, \tau) - \tau \; .$$

We call this TTF the *travel time profile* (TTP) from s to t. Obviously, $\mathrm{EA}_G(s, t, \tau_0)$ and $\mathrm{TTP}_G(s, t)$ are just what an EA query and a profile query computes, respectively.

Each TTF f implicitly defines an *arrival time function* $\mathrm{arr}\, f : \tau \mapsto f(\tau)+\tau$, which yields the arrival time for a given departure time. Note that $\mathrm{arr}(g * f) = \mathrm{arr}\, g \circ \mathrm{arr}\, f$ holds. 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,

may change the memory usage of course. Regarding the running time, we expect a change by a constant factor.

dep $f(\tau)$ is the set of possible departure times. Dual to the EA time is the interval of *latest departure times* (LD times)

$$\mathrm{LD}_G(s,t,\tau_1) \;:=\; \operatorname{dep}\mathrm{TTP}_G(s,t)(\tau_1) \;\subseteq\; \mathbb{R}$$

for traveling from s to t with arrival time τ_1. This is what an LD query computes.

What we also need is the *k-hop neighborhood* of a node v in a graph G defined as

$$N_G^k(v) := \left\{u \in V \setminus \{v\} \;\middle|\; \exists \text{ path from } u \text{ to } v \text{ or } v \text{ to } u \text{ of } k \text{ or less edges in } G\right\}.$$

A further important notion is the *transpose graph*

$$G^\top := (V, E^\top) := (V, \{v \to_f u \,|\, u \to_f v \in E\})$$

of a graph G. We further define an upper and a lower bound of the minimum travel times from s to t, namely

$$\mathrm{Max}_G(s,t) \;:=\; \min\left\{ \sum_{i=1}^{k-1} \max f_i \;\middle|\; \langle v_1 \to_{f_1} \cdots \to_{f_{k-1}} v_k\rangle \text{ is path from } s \text{ to } t \text{ in } G\right\}$$

and

$$\mathrm{Min}_G(s,t) \;:=\; \min\left\{ \sum_{i=1}^{k-1} \min f_i \;\middle|\; \langle v_1 \to_{f_1} \cdots \to_{f_{k-1}} v_k\rangle \text{ is path from } s \text{ to } t \text{ in } G\right\}.$$

They fulfill $\mathrm{Min}_G(s,t) \le \mathrm{TTP}_G(s,t)(\tau) \le \mathrm{Max}_G(s,t)$ for all $\tau \in \mathbb{R}$.

3. BUILDING BLOCKS

Earliest Arrival Trees. Dijkstra's algorithm not only computes shortest distances from a start node s to all other nodes in a graph but also corresponding shortest paths. Together, these shortest paths form a shortest path tree (e.g., see Mehlhorn and Sanders [2008]). Here, we have similar concepts: A minimal subgraph $T \subseteq G$ that contains exactly one (s, u, τ_0)-EA-path for all $u \in V$ that are reachable from s is called an *earliest arrival* (EA) *tree* or, to be more precise, an (s, τ_0)-EA-tree. Every (s, τ_0)-EA-tree is a directed tree and all (s, u, τ_0)-EA-paths in T are prefix-optimal.

Dijkstra-Like Algorithms. In this work, we utilize several modifications of Dijkstra's well-known algorithm. All these Dijkstra-like algorithms have very similar structure: They grow a search space starting from s while repeatedly updating tentative *node labels* and tentative *predecessor information* by relaxing edges. An edge $u \to v$ is relaxed when its source node u is taken out of the priority queue (PQ). The final node labels and the final predecessor information represent the desired result. Usually, the label of a node u represents some kind of minimum distance from s to u. The predecessor information of all nodes implicitly represents a subgraph that contains an optimal path from s to every reachable node.

In fact, Dijkstra-like algorithms can be distinguished by the following five characteristics: (i) What are the node labels? (ii) What is the structure of the predecessor information? (iii) What is the key (or the priority) of a node label with respect to the priority queue (PQ)? (iv) How are the node labels updated on edge relaxation? (v) What are the initial node labels? Of all Dijkstra-like algorithms used in this work, time-dependent Dijkstra [Dreyfus 1969] and travel time profile search [Orda and Rom 1990] are the two most basic ones. Both of them are well known.

In this work, we use pseudocode to describe algorithms as clearly as possible. It should be noted, however, that we often use nested pseudofunctions to make the structure of the algorithms more clear (e.g., as in Algorithm 1). On the one hand, we hope to increase readability this way. On the other hand, we often use pseudofunction calls to invoke other algorithms or nested pseudofunctions of other algorithms as subprocedures. The goal is both to reduce the size of pseudocode and to emphasize the relationships

between the different algorithms. Note that the syntax of parameter lists of pseudofunctions follows a Pascal-like style. Also, some parameters of the pseudofunctions are references. Their only purpose is to provide context information of the caller to the callee. So, "passing" them does not raise any real additional work.

Predecessor Graphs. In the most basic case, the predecessor information $p[u]$ associated with a node u just stores the predecessor node of u on a tentative optimal path from s to u. After termination, all $p[u]$ with $u \in V$ have reached their final value. In case of Dijkstra's algorithm, for example, the predecessor information implicitly represents a shortest path tree and

$$\langle s = p[\dots p[u]\dots] \to \cdots \to p[p[u]] \to p[u] \to u \rangle$$

is a shortest path from s to u for all $u \in V$. For some Dijkstra-like algorithms, $p[u]$ not only stores a single predecessor node, but it also stores a *set* predecessor nodes. Given a set of predecessor nodes $p[u]$ for all nodes $u \in V$, we often write

$$\mathit{Graph}(p) := \left(\left\{u \in V \;\middle|\; p[u] \neq \emptyset \text{ or } u \in p[v] \text{ with } v \in V\right\},\ \left\{u \to v \in E \;\middle|\; u \in p[v]\right\}\right) \subseteq G$$

to denote the predecessor graph of the underlying Dijkstra-like computation. The *predecessor graph* is simply the graph made up by all edges $u \to v$ with $u \in p[v]$.

Time-Dependent Dijkstra. Given a start node s and a departure time τ_0 time-dependent Dijkstra (see *tdDijkstra*, Algorithm 1) computes $\mathrm{EA}_G(s, u, \tau_0)$ for all $u \in V$ as well as a corresponding (s, τ_0)-EA-tree. The initial node label of the start node s is the departure time τ_0, the label $\tau[u]$ of a node u is the tentative EA time for traveling from s to u. The predecessor information $p[u]$ of u is the predecessor node of u on a corresponding tentative EA path. When we relax an edge $u \to_f v$, we update the label $\tau[v]$ of v by $\tau[v] := \min\{\tau[v], \operatorname{arr} f(\tau[u])\}$. If $\tau[v]$ gets decreased, we also update the predecessor information by $p[v] := u$. As PQ key of a node u, we simply use its current label $\tau[u]$. On road networks, *tdDijkstra* works very similar to the original Dijkstra and has similar running times and memory usage. Also, a node once removed from the PQ is never inserted again. After termination, we have $\tau[u] = \mathrm{EA}_G(s, u, \tau_0)$ for every $u \in V$ and $\mathit{Graph}(p)$ is an (s, τ_0)-EA-tree in G. Note that $p[u]$ is not a set but an element of $V \cup \{\perp\}$ here. Nevertheless, we apply the earlier definition of $\mathit{Graph}(p)$ by identifying $p[u] \in V$ with the singleton set $\{p[u]\}$ and $p[u] = \perp$ with the empty set $\emptyset$.

ALGORITHM 1: The time-dependent version of Dijkstra's algorithm computes final labels $\tau[u] = \mathrm{EA}_G(s, u, \tau_0)$ for all reachable nodes u (and $\tau[u] = \infty$ if u not reachable). The final predecessor graph $\mathit{Graph}(p)$, which is implicitly represented by the array p, is a (s, τ_0)-EA-tree.

```
1  procedure tdDijkstra(s : V, τ0 : ℝ)
2      τ[u] := ∞ for all u ∈ V, τ[s] := τ0                    // initial node labels
3      p[u] := ⊥ for all u ∈ V                                // initial predecessor information
4      procedure tdRelax(u →_f v : Edge, τ, p, Q : Reference)
5          if arr f(τ[u]) ≥ τ[v] then return
6          τ[v] := min{τ[v], arr f(τ[u])}                     // update node label of v
7          p[v] := u                                          // update predecessor information of v
8          if v ∉ Q then Q.insert(v, τ[v])
9          else Q.decreaseKey(v, τ[v])
10     Q := {(s, τ[s])} : PriorityQueue
11     while Q ≠ ∅ do
12         u := Q.deleteMin()
13         for u →_f v ∈ E do tdRelax(u →_f v, τ, p, Q)
```

Profile Search. Given a start node s, the *profile search* (see *profileSearch*, Algorithm 2) computes $\mathrm{TTP}_G(s,u)$ for all $u \in V$. Accordingly, the node label $f[u]$ of a node u is a tentative TTP for traveling from s to u. The tentative predecessor graph $Graph(p)$ contains the corresponding tentative EA paths for all possible departure times $\tau \in \mathbb{R}$. When relaxing an edge $u \to v$ with TTF f_{uv}, we update the label $f[v]$ of v by computing $f[v] := \min(f[v], f_{uv} * f[u])$. The PQ key of a node u is the minimum of its current label, that is, $\min f[u]$. The reader may have noticed that we check whether $f[v]$ lies completely under g_{new} and vice versa (see Lines 5 and 6). Both are essentially the same as computing $\min(g_{\text{new}}, f[v])$ and hence take $O(|g_{\text{new}}| + |f[v]|)$ time. After termination, $Graph(p)$ contains an (s,u,τ)-EA-path for all $u \in V, \tau \in \mathbb{R}$ (if u is reachable from s).

Note that *profileSearch* is very expensive in time and space and not feasible for larger road networks. This is confirmed by our experiments (Section 8). Also, during *profileSearch*, nodes may be reinserted in the PQ after they have been removed. In our experience, however, this is not what makes *profileSearch* so slow on road networks as reinserts happen rarely. Instead, the complexity of *profileSearch* is raised by the increasing complexity of the node labels, that is, of the tentative TTPs that are generated during the search.

ALGORITHM 2: The travel time profile search computes final labels $f[u] = \mathrm{TTP}_G(s,u)$ for all nodes u. The final predecessor graph $Graph(p)$ contains an (s,u,τ)-EA-path for all reachable nodes u and all departure times $\tau \in \mathbb{R}$.

```
1  procedure profileSearch(s : V)
2      f[u] := ∞ for all u ∈ V, f[s] := 0                          // node labels are TTFs
3      p[u] := ∅ for all u ∈ V                                     // predecessor information consists of sets
4      procedure profileRelax(u, v : V, g_new : TTF, f, p, Q : Reference)
5          if g_new(τ) ≥ f[v](τ) for all τ ∈ ℝ then return
6          if g_new(τ) < f[v](τ) for all τ ∈ ℝ then p[v] := ∅       // remove suboptimal predecessors
7          f[v] := min(f[v], g_new)                                // update tentative node label of v
8          p[v] := {u} ∪ p[v]                                      // remember a tentative predecessor of v
9          if v ∉ Q then Q.insert(v, min f[v])
10         else Q.decreaseKey(v, min f[v])                         // minimum of TTF is PQ key
11     Q := {(s, 0)} : PriorityQueue
12     while Q ≠ ∅ do
13         u := Q.deleteMin()
14         for u → v ∈ E with TTF f_uv do profileRelax(u, v, f_uv * f[v], f, p, Q)
```

Profile Interval Search. As *profileSearch* is so expensive, we use *profile interval search* (see *profileIntervalSearch*, Algorithm 3) as a relatively loose approximation of *profileSearch*. Instead of $\mathrm{TTP}_G(s,u)$, as in the case of *profileSearch*, it computes the intervals $[\mathrm{Min}_G(s,u), \mathrm{Max}_G(s,u)]$ for every $u \in V$. Hence, the label of a node u is a tentative interval $[q[u], r[u]]$. When relaxing an edge $u \to_f v$, we update the label of v by $[q[v], r[v]] := \big[\min\{q[v], q[u] + \min f\}, \min\{r[v], r[u] + \max f\}\big]$. As PQ key of u, we use $q[u]$. According to our experience *profileIntervalSearch* runs much faster and needs much less space on road networks than *profileSearch*. In fact, it has similar running times and memory usage as the original Dijkstra. Analogously to *profileSearch*, the predecessor graph of *profileIntervalSearch* contains an EA-path from s to all reachable nodes for all departure times. However, the predecessor graph computed by *profileIntervalSearch* should contain more edges than the one computed by *profileSearch*.

ALGORITHM 3: The travel time profile interval search computes final labels $[q[u], r[u]] = [\mathrm{Min}_G(s,u), \mathrm{Max}_G(s,u)]$ for all nodes u. As in the case of profile search, $Graph(p)$ contains an (s,u,τ)-EA-path for all reachable nodes u and all departure times $\tau \in \mathbb{R}$ after termination.

```
1  procedure profileIntervalSearch(s : V)
2      [q[u], r[u]] := [∞, ∞] for all u ∈ V, [q[s], r[s]] := [0, 0]      // node labels are intervals
3      p[u] := ∅ for all u ∈ V                                           // predecessor information consists of sets
4      procedure intervalRelax(u, v : V, [q_new, r_new] : Interval, q, r, p, Q : Reference)
5          if q_new > r[v] then return
6          if r_new < q[v] then p[v] := ∅                              // remove suboptimal predecessors
7          p[v] := {u} ∪ p[v]                                         // remember a tentative predecessor of v
8          if q_new ≥ q[v] and r_new ≥ r[v] then return
9          [q[v], r[v]] := [min{q[v], q_new}, min{r[v], r_new}]       // update node label of v
10         if v ∉ Q then Q.insert(v, q[v])
11         else Q.decreaseKey(v, q[v])                                // the PQ key is the lower bound of the interval
12     Q := {(s, 0)} : PriorityQueue
13     while Q ≠ ∅ do
14         u := Q.deleteMin()
15         for u →_f v ∈ E do intervalRelax(u, v, [q[u] + min f, r[u] + max f], q, r, p, Q)
```

Monotony of PQ Keys. It is important to know that all the Dijkstra-like algorithms considered thus far show a certain monotonic behavior.

LEMMA 3.1. *Let u_1, u_2 be two nodes such that u_1 is removed from the PQ some time before u_2 during the execution of tdDijkstra (Algorithm 1), profileSearch (Algorithm 2), or profileIntervalSearch (Algorithm 3) respectively. Let τ_1, τ_2 and f_1, f_2 and $[q_1, r_1]$, $[q_2, r_2]$ be the respective node labels at the time when each node is removed. Then, one of the following three conditions holds, respectively: (i) For tdDijkstra, we have $\tau_1 \leq \tau_2$; (ii) for profileSearch, we have* $\min f_1 \leq \min f_2$*; and (iii) for profileIntervalSearch, we have $q_1 \leq q_2$.*

PROOF. All three conditions can be verified easily using induction over the number of removals from the PQ, that is, the number of times *deleteMin* is invoked. □

Time-Dependent One-to-One Search. The Dijkstra-like algorithms described thus far all solve a one-to-all problem. In route planning, however, we are mainly interested in *one-to-one* problems where we only want to route from a node s to a node t. So, we can save running time by stopping the computation as soon as the node label of the destination node t is final. For time-dependent Dijkstra (Algorithm 1), this is the case as soon as $Q.deleteMin()$ yields t. For *profileSearch* and *profileIntervalSearch* (Algorithms 2 and 3), we can stop the execution as soon as $Q.min() > \max f[t]$ or $Q.min() > r[t]$ holds, respectively. This follows directly from Lemma 3.1.

Bidirectional Search. In case of one-to-one search, one can save even more running time if Dijkstra-like algorithms are performed in a *bidirectional* manner. This idea is not new and has been applied to speed up Dijkstra's algorithm in the past [Dantzig 1962]. Basically, it works as follows: To compute a shortest path from a node s to a node t in a graph with time-independent edge weights, we perform two Dijkstra searches "at the same time" until they meet. More precisely, we alternately perform a forward Dijkstra search starting from s and a backward Dijkstra search starting from t. The backward search must use the edges in reverse direction, that is, it runs on the transpose graph. On road networks, the number of processed nodes roughly grows quadratically with the distance from the start node. So, as we have two searches that

both cover an area with about half the radius, instead of one single search with full radius, the number of processed nodes should roughly half in total. Hence, bidirectional search should speed up the one-to-one version of Dijkstra's algorithm by a factor of about two [Bauer et al. 2010]. However, in this work, we do not use bidirectional search to speed up EA or profile queries directly, but rather to enable EA and profile queries on TCH structures (see Sections 5, 6, and 7). This brings much greater speed-ups than just a factor of two (see Section 8).

Backward profileSearch and profileIntervalSearch. Bidirectional search includes backward search. To obtain a backward version of *profileSearch* that computes $\mathrm{TTP}_G(u,t)$ for all $u \in V$, it is not enough just to run it on the transpose road network $G^\top$. We also have to adapt edge relaxation such that the order of the link operation is swapped. More precisely, the invocation of *profileRelax* in Line 14 of Algorithm 2 has to be changed to $\mathit{profileRelax}(u, v, f[v] * f_{uv}, f, p, Q)$. This is necessary because linking of TTFs is not commutative. The final label $f[u]$ of each node $u \in V$ fulfills $f[u] = \mathrm{TTP}_G(u,t)$. The role of the predecessor graph is reverted in case of backward search. Let $S \subseteq G^\top$ be predecessor graph of backward *profileSearch*. Then, $S^\top$ contains a (u,t,τ)-EA-path for all $u \in V$, $\tau \in \mathbb{R}$ (if t is reachable from u).

The behavior of backward *profileIntervalSearch* is mainly inherited from backward *profileSearch*, but there is no need to adapt the edge relaxation. Especially for $S \subseteq G^\top$ being the corresponding predecessor graph, there is also a (u,t,τ)-EA-path in $S^\top$ for all $u \in V, \tau \in \mathbb{R}$ (if t reachable from u). The final label $[q[u], r[u]]$ of each node $u \in V$ fulfills $[q[u], r[u]] = [\mathrm{Min}_G(u,t), \mathrm{Max}_G(u,t)]$. As in the case of forward search, the predecessor graph of backward *profileIntervalSearch* contains more edges than of backward *profileSearch*.

Corridors and Cones. Consider a start node s and a destination node t. A subgraph $C \subseteq G$ containing s and t as well as a path from s to t is a *corridor from s to t*. Now, also consider the nodes $u_1, \ldots, u_k$. A subgraph $C' \subseteq G$ containing s and $u_1, \ldots, u_k$ is called a *cone from s to* $u_1, \ldots, u_k$ if it contains a path from s to each of $u_1, \ldots, u_k$. If C consists exactly of all paths from s to $u_1, \ldots, u_k$ in G, we call it the cone *induced* by start node s and target set $\{u_1, \ldots, u_k\}$ in G. We denote C by $\mathit{Cone}_G(s, \{u_1, \ldots, u_k\})$ in this case. The *induced corridor from s to t in G* defined by $\mathit{Corridor}_G(s,t) := \mathit{Cone}_G(s, \{t\})$ is a special case. Corridors and cones help to reduce running time and memory usage significantly when expensive algorithms have to be used: First, we use a cheap algorithm to build up a corridor or cone that contains the relevant parts of the graph and not too many other nodes and edges. Then, we perform the expensive algorithm only in the corridor or cone to obtain the desired result. This principle is applied very often in this work.

Shortcut Edges. A *shortcut (edge)* is an "artificial" edge in the sense that it does not directly correspond to a single road segment. Instead, shortcuts represent paths in the road network. Note that shortcuts are quite common in route planning. In this work, shortcuts are time dependent and represent paths of length two: For intervals $I_1, \ldots, I_k$ with $\mathbb{R} = I_1 \cup \cdots \cup I_k$, a shortcut $u \to v$ represents a path $\langle u \to x_i \to v \rangle$ for all departure times $\tau \in I_i$ and each $i \in \{1, \ldots, k\}$. The nodes $x_1, \ldots, x_k$ are called *middle nodes*. For some departure intervals I_i, it may happen that $u \to v$ must not be interpreted as a shortcut but as an original edge that corresponds to a real road segment—we set $x_i = \perp$ in this case. The shortcut $u \to v$ is annotated with the corresponding sequence $\langle (I_1, x_1), \ldots, (I_k, x_k) \rangle$. Now, we can unpack $u \to v$ for a given departure time τ_0 to obtain the path $\langle u \to_f x_i \to_g v \rangle$ with $\tau_0 \in I_i$—at least if $x_i \neq \perp$. If one or both of the edges $u \to_f x_i$ and $x_i \to_g v$ are shortcuts themselves, they can be unpacked for the departure times τ_0 and $\mathrm{arr}\, f(\tau_0)$, respectively. In fact, every shortcut can be unpacked completely for a given departure time in a recursive manner. The resulting path consists completely

of original edges and is called the original path represented by the shortcut $u \to v$ for departure time τ_0.[3]

4. PREPROCESSING

In this section, we explain what a TCH structure H is exactly and how it is generated from the original road network G in a preprocessing step. TCHs exploit the inherently hierarchical structure of road networks: Some junctions are more "important" than others, which is, for example, reflected by the fact that more EA paths contain these nodes. The idea is that a node is at a higher level of the hierarchy the more important the corresponding junction is. We express this by a total order[4] $\prec$ on V. There, $u \prec v$ means that u is less important than v.

We construct the hierarchy by contracting the nodes of G in order $\prec$. In principle, contracting a node x means to remove x and all its incident edges from the graph without changing the EA times between the remaining nodes. We achieve this by replacing each path $\langle u \to_f x \to_g v \rangle$ by a shortcut edge $u \to_{g*f} v$ when necessary. A shortcut is necessary if $\langle u \to_f x \to_g v \rangle$ is an EA path for some departure time, that is, if $EA_G(u, v, \tau) = \operatorname{arr}(g * f)(\tau)$ holds for some $\tau \in \mathbb{R}$. When a shortcut is necessary, an edge $u \to_h v$ may already be present. Then, we do not insert another edge but merge the edges: We replace $u \to_h v$ by $u \to_{\min(g*f,h)} v$, avoiding parallel edges this way.

LEMMA 4.1. *Contracting a node $x \in V$ as described earlier transforms the graph $G = (V, E)$ to another graph $G' = (V', E')$ with $V' = V \setminus \{x\}$ and $\mathrm{EA}_{G'}(s, t, \tau) = \mathrm{EA}_G(s, t, \tau)$ for all $s, t \in V', \tau \in \mathbb{R}$. Especially if there is a path $\langle u \to_f x \to_g v \rangle$ in G, which is an EA path for some departure time $\tau_0 \in \mathbb{R}$, then G' contains a shortcut edge $u \to_h v$ with* $\operatorname{arr} h(\tau_0) = \operatorname{arr}(g * f)(\tau_0) = \mathrm{EA}_G(u, v, \tau_0)$.

PROOF. Assume there is an (u, v, τ_0)-EA-path $\langle u \to_f x \to_g v \rangle$ present in G. If there is no edge $u \to_{h'} v$ in G, an appropriate shortcut is inserted. Otherwise, the shortcut $u \to_{g*f} v$ is merged into G', and we have a shortcut edge $u \to_{\min(h', g*f)} v$ in G'. But it is obvious that $\operatorname{arr} \min(h', g * f)(\tau_0) = \operatorname{arr} g * f(\tau_0) = \mathrm{EA}_G(u, v, \tau_0)$ holds as $\langle u \to_f x \to_g v \rangle$ is an (u, v, τ_0)-EA-path in G and there can not be an earlier arrival time, hence. □

In other words, G' has the same nodes as G except for x, and all EA times in G' are the same as in G. Such a graph G' is called an *overlay graph* of G [Holzer et al. 2008]. A TCH is constructed by successively contracting all nodes of G in the order given by $\prec$, that is, in the order $x_1 \prec \cdots \prec x_{|V|}$. So, a TCH is a hierarchy of overlay graphs $G_1 = (V_1, E_1), \ldots, G_{|V|} = (V_{|V|}, E_{|V|})$ with $V_{i+1} = V_i \setminus \{x_i\}$. However, the TCH is never represented this way. This would need too much memory. Instead, we just store the original graph G together with all inserted and merged shortcuts. Also, we remember whether an edge leads upward or downward in hierarchy. In fact, this information is all we need for fast and exact EA and profile queries (see Sections 5 and 6). The resulting structure is the actual time-dependent contraction hierarchy (TCH) H.

COROLLARY 4.2. *Consider a road network G and a corresponding TCH H, that is, H consists of G and all inserted and merged shortcuts. Then, for all departure times as well as for all pairs of start and destination nodes, the corresponding EA times are the same in G and H.*

Constructing the hierarchy requires that the importance relation $\prec$ is already known. If this is not the case, the node order has to be determined first. So, preprocessing

[3]This is a little simplified. Actually, TTFs in this work have a period of 24 hours, and this is also the case for the structure of the middle nodes. More precisely, we have $[0, 24\mathrm{h}) = I_1 \cup \cdots \cup I_k$ and the middle node for every departure time in $\mathbb{R}$ is sufficiently specified by the sequence $\langle (I_1, x_1), \ldots, (I_k, x_k) \rangle$.

[4]A total order $\prec$ is an antisymmetric, transitive relation where always one of $a \prec b$ or $b \prec a$ holds.

consists of two tasks: *node ordering* and *hierarchy construction*. Node ordering as we do it, however, is basically a construction process with a lot of additional work. As a consequence, node ordering not only yields the importance relation $\prec$ but also a corresponding hierarchy. A further construction step is not necessary. Nevertheless, it is natural that we first explain how hierarchy construction works when $\prec$ is already known (Section 4.1). Then, we explain how the node ordering works, as it is a similar process but with additional work (Section 4.2). Both node ordering and construction parallelize quite naturally on shared memory architectures. Once the preprocessing is finished, we can answer EA and profile queries very fast (see Sections 5, 6, and 7).

4.1. Constructing the Hierarchy

Earlier in the text, we explained that the TCH is never represented explicitly as hierarchy of overlay graphs. This is also the case during preprocessing. Instead, we successively construct a TCH H starting from G by inserting and merging more and more shortcuts. Initially, we set $H := G$. All shortcuts $u \to v$ that turn out to be necessary when a node is contracted are added to H (or merged into H if an edge $u \to v$ is already present, respectively). Every time a node x is contracted, it is not really removed but marked as contracted. By $R = (V_R, E_R)$, we denote the subgraph of H, which is induced by all nodes that are not contracted yet.[5] Conceptually, a node x and its incident edges are removed from R as soon as x is contracted. Hence, we call R the *remaining graph*. Note that the EA times in G and H are always the same and that R is an overlay graph of both G and H.

The construction phase is divided in iterations. In every iteration, we contract a number of nodes. At the beginning of every iteration, we build the set

$$I := \left\{ x \in V_R \;\middle|\; \forall\, u \in N_R^1(x) : x \prec u \right\} \tag{1}$$

of nodes, which are less important than all their neighbors in R. Note that I is an independent node set[6] in R. Having built the set I, we contract all the nodes in I. This includes that all necessary shortcuts are added to H (or merged into H, respectively). The remaining graph has changed now. On the one hand, all nodes in I and their adjacent edges have been removed from R. On the other hand, some new shortcuts may be present in R and some edges in R may have a modified TTF now (due to merging).

The next iteration works exactly like the one before: We build another independent node set I, which is a subset of the now smaller set of remaining nodes V_R. Then, we contract all the nodes in the new set I while inserting some shortcuts into H (or merging some shortcuts, respectively). Then, we perform another iteration of this kind and so on. We repeat this process until R is the empty graph. When this process ends, all nodes are contracted, and H is completely transformed into a TCH.

Note that the contraction of a node $x \in I$ has no influence on the contraction of any other node in I. So, all nodes in I can be contracted in an arbitrary order or even in parallel without altering the result of this process. This is the case because the contraction of a node does not alter the EA times in a graph. Moreover, no two nodes in I are adjacent in R, which means that no node in I loses or gains any edges when another node in I is contracted. If some nodes in I have common neighbors, then we simply merge all parallel inserted shortcuts.

[5]The subgraph of a graph $G = (V, E)$, which is induced by a subset of nodes $X \subseteq V$, is defined as the subgraph $\big(X, \{u \to v \in E \mid \{u, v\} \subseteq X\}\big) \subseteq G$.

[6]An *independent node set* is a set of nodes in a graph such that no two nodes in the set are adjacent.

Avoiding Shortcuts. When we contract a node x, we have to insert or merge a shortcut $u \to_{g*f} v$ for each path $\langle u \to_f x \to_g v \rangle$ in R, which is an (u, v, τ)-EA-path for at least one $\tau \in \mathbb{R}$. To find out whether this is the case or not, we perform the one-to-one version of *profileSearch* in the remaining graph R with start node u and destination node v; we call this the *witness search*. The witness search yields $f[v] = \mathrm{TTP}_R(u, v) = \mathrm{TTP}_G(u, v)$ as final label of v. If $\forall \tau : (g * f)(\tau) > f[v](\tau)$ holds, which can be checked in $\mathrm{O}(|g * f| + |f[v]|)$ time, then the shortcut is not necessary. We call $f[v]$ a *witness profile* in this case. However, *profileSearch* is a very expensive algorithm regarding time and space. Moreover, the complexity of the TTFs present in R increases during the construction process (see Section 8). As a consequence *profileSearch* gets more and more expensive making the construction process infeasible. Alternatively, we could just insert *all* possible shortcuts. This is also not good, because the resulting TCH would probably need far too much space and queries would probably be much slower.

Faster Witness Search Using Corridors. However, the witness search can be accelerated: If we first perform the one-to-one version of *profileIntervalSearch* with start node u and destination node v in R, we get the interval $[q[v], r[v]] = [\mathrm{Min}_R(u, v), \mathrm{Max}_R(u, v)]$. So, if $r[v] < \min(g * f)$ holds, we know that no shortcut is needed—without doing *profileSearch*. We call $[q[v], r[v]]$ a *witness interval* in this case. Otherwise, if $q[v] > \max(g * f)$, we know that the shortcut is required, also without *profileSearch*. But if $[q[v], r[v]]$ and $[\min g * f, \max g * f]$ overlap, we know nothing and perform *profileSearch*. However, several edges can be ignored during *profileSearch* here: Let S be the predecessor graph (see Section 3) of the *profileIntervalSearch*. Then, we perform *profileSearch* only in the corridor $C := \mathit{Corridor}_S(u, v)$. To obtain the corridor C, we only have to perform a breadth-first search (BFS) in $S^\top$ starting from v while adding a transposed version of all touched edges to C.

Thinning out the Corridor Heuristically. We can further decrease the time spent on *profileSearch* by using a corridor that is even thinner than C. To do so, we do not store the whole the predecessor information during *profileIntervalSearch*. Instead, we only remember two predecessors of a node w with label $[q[w], r[w]]$ at a time: the one that lastly improved $q[w]$ and the one that lastly improved $r[w]$ (i.e., $|p[w]| \leq 2$). Of course, the resulting thinner corridor C' might not contain all EA paths from u to v or even no EA path. This may lead to unnecessary shortcuts, as we may fail to prove that a shortcut is not needed, but it turned out that this happens quite rarely. Instead, the running time spent on *profileSearch* is reduced greatly. So, the generation of the corridor is a heuristic that works very well. Note that this heuristic never prevents any necessary shortcuts. The result is always correct as the heuristic is conservative.

Sample Search. It is also possible to further reduce the cases where *profileSearch* is performed: First, we perform a *sample search*, a one-to-one *tdDijkstra* on R from u to v with departure time $\Pi/2$. There, Π is the period of the TTFs. Even if $[q[v], r[v]]$, as computed by the *profileIntervalSearch*, and $[\min(g * f), \max(g * f)]$ overlap, sample search might still yield that $\mathrm{EA}_R(u, v, \Pi/2) = \mathrm{arr}(g * f)(\Pi/2)$. Then, we know that the shortcut is necessary without *profileSearch*. However, we perform a sample search only occasionally: We maintain a value β which we set to zero at the very beginning of the construction process. Every time a shortcut is inserted, we increase β by some value λ^+. If a potential shortcut is not inserted, we decrease β by another value λ^-. If β gets larger than some threshold ξ, we switch to the sample search mode, meaning that we always perform a sample search before interval search. If, in contrast, β gets smaller than $-\xi$, we switch back to the no sample search mode. In our implementation, we chose $\lambda^+ := 4$, $\lambda^- := 1$, and $\xi := 1{,}000$. As an intuition, we perform a sample search

when shortcuts are more probable, and we omit a sample search when shortcuts are less probable.

Hop Limit. To avoid that sample search, *profileIntervalSearch*, and *profileSearch* take too much time, we limit the search radius to 16 hops (this has been adopted from the preprocessing of static CHs). The idea is that the edges of a node w are not relaxed if the EA path from u to w has more than 16 edges. Here, it must be noted that the predecessor graph of *profileIntervalSearch* and *profileSearch* does not contain a unique path from u to w. So, in these cases, we just use the number of hops that emerge from the edge $w' \to w \in E_R$ relaxed last. Of course, the hop limit might sometimes prevent us from finding witnesses, which means that unnecessary shortcuts are inserted. However, the resulting TCH structures are sparse enough to allow small running times of EA and profile queries (Section 8).

4.2. Ordering the Nodes

We already suggested that the node ordering is actually a hierarchy construction plus a lot of extra work. In every iteration, the node ordering chooses an independent node set J in the remaining graph R. Then, it contracts all nodes in J while adding some shortcuts to H or merging them, respectively. Having finished an iteration, we choose another independent node set J in the new R and so on. This is the same procedure described in Section 4.1, and all the techniques described there are also applied.

However, as $\prec$ is not yet fully established during this process, we need another way to decide which nodes are contracted next, that is, which nodes are put into J. To do so, we assign a tentative cost value $u.cost$ to every node $u \in V$. This way, we estimate how attractive a node is to be contracted. The nodes with smaller costs should be contracted earlier. Actually, the tentative cost estimates how the remaining graph R would change when a node were contracted. Accordingly, we perform a *simulated contraction* of a node u in R whenever u gains or loses an edge, but this happens exactly when a neighbor x of u in R is really contracted. After the simulated contraction of u, we update the tentative cost $u.cost$ of u.

Note that the simulated contraction does not alter H or R. Instead, we just look at which shortcuts are necessary without actually inserting or merging them and without marking u as contracted. The only purpose of the simulated contraction is to find out how a real contraction would change R.

Basic Node-Ordering Procedure. In more detail, the node ordering works as follows: In an initial step before the first iteration, we determine the initial tentative cost for all nodes $u \in V$ by performing a simulated contraction for every node $u \in V$. Then, we perform a sequence of iterations similar to the construction process described in Section 4.1. The independent node set, however, is chosen differently this time. Instead of a set I, as described in Equation (1), we choose a set $J \subseteq V_R$ with the property

$$x \in J \quad \Longrightarrow \quad \forall\, u \in N_R^2(x) : \big(u \notin J \ \text{ and } \ x.cost \leq u.cost\big), \tag{2}$$

which means that every node in J has minimal cost in its 2-hop neighborhood and none of its 2-hop neighbors is also added to J. Then, we contract all nodes in J while inserting or merging the necessary shortcuts. But before we begin with the next iteration by choosing the next independent node set J in the changed remaining graph R, we update the tentative costs of the neighbors of the just contracted nodes. To do so, we have to simulate the contraction of these neighbors of course.

Note that an independent node set J as characterized by Property (2) is different from an independent node set I, as defined by Equation (1). And this is not only the case because we use the tentative cost instead of $\prec$. The gap between the nodes in J is

at least three hops instead of two hops, as in the case of I. The reason for this difference is explained at the end of this section in the paragraph about parallelization.

Determining the Node Order. During the node ordering, we successively set up the relation $\prec$ in the following way: Let $J_1, \ldots, J_k$ be the sequence of the repeatedly chosen independent node sets ordered by time of creation. For $u \in J_i$ and $v \in J_j$ with $i < j$, we set $u \prec v$. For $u, v \in J_i$, we can choose freely between $u \prec v$ and $v \prec u$.

Maintaining the Tentative Costs. The contraction of a node x changes the edges of its adjacent nodes in R, and (as said earlier) we have to update the tentative costs of these adjacent nodes. More precisely, the contraction of x means that every node $u \in N_R^1(x)$ loses at least an edge $u \to x$ or $x \to u$ and possibly gains one or more shortcuts. So, we have to update the tentative cost $u.cost$ of all nodes $u \in N_R^1(x)$ because $u.cost$ estimates how the contraction of u would change R (and this depends on the edges that u has in R). But updating this estimate includes a simulated contraction of u. Having finished the simulated contraction of u, we compute the new tentative cost of u as a linear combination of four *cost terms*.

Mainly, the four cost terms are chosen in a way that we obtain a hierarchy, which is *flat* and *sparse*. As a consequence, the paths from the bottom of the hierarchy to its top are not too long and the hierarchy does not contain too many edges. Such hierarchies hasten our query algorithms, as these algorithms only go upward from start and destination node. However, to achieve a lower memory usage, a low complexity of TTFs is also important. The four cost terms *edge quotient*, *hierarchy depth*, *unpacked edge quotient*, and *complexity quotient*—here, sorted by importance—try to ensure these properties and are defined as follows.

The *edge quotient* helps to keep the hierarchy sparse. When a node u was contracted, all its incident edges would be removed from R, and then some shortcuts would be inserted. Accordingly, the edge quotient

$$Edges(u) := \frac{\#\,\text{inserted shortcuts}}{\max\{1,\ \#\,\text{edges removed from } R\}}$$

expresses how the number of edges would be changed locally by the contraction of u. Note that the edge quotient works better than the more intuitive term *edge difference*

$$\#\,\text{inserted shortcuts} - \#\,\text{edges removed from } R$$

would. This is because the values of the difference could get so large that other terms would not have enough influence any more.

Using only the edge quotient, one can get quite slow queries, as the resulting hierarchy might be sparse but not flat. So, we preferably contract nodes everywhere in the graph in a uniform way rather than keeping to a small region. To ensure this, we maintain an attribute $u.depth$ of every node u. At the very beginning of the node ordering, we set $u.depth := 1$ for all nodes u. Whenever a node x is really contracted, we update $u.depth$ of all its neighbors $u \in N_R(x)$ by setting

$$u.depth := \max\{x.depth + 1,\ u.depth\}$$

before we mark x as contracted. The term *hierarchy depth* is simply the current value $Depth(u) := u.depth$. Obviously, $u.depth$ can be maintained without simulated contraction. Note that the way we repeatedly choose the independent set J during node ordering also helps to uniformly distribute the node contractions.

When a node u is contracted, all the inserted (and maybe some of the removed) edges are shortcuts. The *unpacked edges quotient* is the same as the edge quotient, but all shortcuts count for the number of edges of the original path (see Section 3) they

represent, that is, for the number of edges that we get if we unpack them completely:

$$Unpack(u) := \frac{\sum_{\text{inserted shortcuts}} \#\,\text{original edges when unpacked}}{\max\{1,\ \sum_{\text{edges removed from } R} \#\,\text{original edges when unpacked}\}}.$$

The purpose of this term is to balance the length of the shortcuts incident to a node. Similar to the previous term, this should support a uniform distribution which supports a flat hierarchy.

The *complexity quotient*, which is the last of the four terms, helps to keep the complexity of the TTFs low. This saves memory, as the hierarchy needs less space the fewer segments the present TTFs have. Also, this should speed up the profile searches being performed during preprocessing, and also the answering of profile queries after preprocessing. This is because profile search gets slower the more segments the processed TTFs have. However, the use of this term may slow down EA queries a little. We define

$$Complex(u) := \frac{\sum_{\text{inserted shortcuts}} \#\,\text{segments of TTF}}{\max\{1,\ \sum_{\text{edges removed from } R} \#\,\text{segments of TTF}\}}.$$

As said before, we use a linear combination of the four cost terms to compute the tentative cost of a node. Of course, different configurations are possible for assigning values to the coefficients. We use the configuration

$$2Edges(u) + Depth(u) + Unpack(u) + 2Complex(u),$$

which we found by trial and error. We have not performed a systematic exploration.

Caching Simulated Contractions. Every time we simulate the contraction of a node u, we cache the results of the simulation: For each path $\langle v \to_f u \to_g w\rangle$, we remember whether we found a witness or not, and we remember the complexity of $g * f$. So, if the node u is contracted again, we can save a lot of work by looking up every path $\langle v \to u \to w\rangle \subseteq G$ in the cache—regardless of whether it is a simulated or a real contraction this time. However, if v or w are contracted before u, we delete the corresponding piece of information from the cache to save memory. The cached complexity $|g * f|$ is needed to calculate the complexity quotient.

Parallelization. Like the contraction of the nodes in a set I as defined by Equation (1), the contraction of the nodes in a set J, as characterized by Property (2), can be performed in parallel quite naturally for shared memory architectures. Also, the simulated contractions of the adjacent nodes can be performed in parallel. But, if simulations are performed in parallel, the following question arises: Is it possible that two threads both simulate the contraction of the same node? This would be redundant work. But this can never happen because of the 3-hop gap between the nodes in an independent node set J as characterized by Property (2).

Note that we have not yet explained how such sets J, as characterized by Property (2), are chosen. However, this can be done quite easily by choosing the nodes $x \in V_R$, that have minimal tentative cost in $N_R^2(x)$—if a tie-braking rule for nodes with equal tentative cost is available. This also works when the set J is computed in parallel.

Reusing a Node Order. We already said that the node ordering not only yields the node order relation $\prec$ but also a complete TCH and that it is not necessary to perform a separate construction process afterward. However, if we have different sets of TTFs for the road network G, we can compute a node order for only one of these sets. Then, we can reuse this order for all the other sets of TTFs and perform the construction process without further node ordering each. This may increase the running times needed for construction and also for querying, as the resulting TCH may have a somewhat worse

quality, but it might work well enough, and our experiments confirm this (Section 8). This, way we can save a considerable amount of time during preprocessing.

5. BASIC QUERYING WITH A TCH

In this section, we explain how TCH structures can be used to perform very fast EA and profile queries—with exact query results. We achieve this by using *bidirectional upward search*, that is, a bidirectional search (see Section 3) where we only relax edges that lead upward in the hierarchy. As said before, this is fast if the TCH is flat and sparse. This works because of the guaranteed existence of prefix-optimal EA up-down-paths in TCHs (see Section 5.1). EA up-down-paths are analogous to shortest up-down-paths mentioned in the context of static CHs (see Section 1.1).

For EA queries, bidirectional search has a problem, namely, that exact time-dependent backward search would require us to know the arrival time, which is part of what we want to compute. To overcome this, we apply approximation. More precisely, we use *tdDijkstra* as forward search but *profileIntervalSearch* as backward search because it does not require a known arrival time. Even though this does not yield an EA path, we know that the predecessor graphs of forward and backward search together contain an EA up-down-path for the given departure time. All that is left to do is to perform a further *tdDijkstra* on the edges touched by the backward search (see Section 5.2). Bidirectional profile queries, in contrast, are simpler. This is because *profileSearch* does not require a fixed arrival time when used as backward search. As a result, profile queries on TCHs are straightforward (see Section 5.3).

Both EA and profile queries can be made faster by applying a pruning technique called *stall-on-demand* (see Section 5.4). However, in the case of profile queries, we can even do better: Before we perform the bidirectional *profileSearch*, we perform a bidirectional *profileIntervalSearch* first. This yields two cones in which we we perform the bidirectional *profileSearch*. This needs much less time because much less edges have to be processed by the *profileSearch* (see Section 5.5).

5.1. EA Up-Down-Paths and Representing a TCH

All query algorithms presented in this work include bidirectional search that goes only upward in the TCH. The correctness of these algorithms relies on the aforementioned existence of prefix-optimal EA up-down-path in TCHs. Given a TCH H, we call a path

$$\langle u_1 \to \cdots \to u_k \to x \to v_\ell \to \cdots \to v_1 \rangle \subseteq H$$

an *up-down-path* if $u_1 \prec \cdots \prec u_k \prec x$ and $v_1 \prec \cdots \prec v_\ell \prec x$ hold, that is, $\langle u_1 \to \cdots \to u_k \to x \rangle$ only goes upward and $\langle x \to v_\ell \to \cdots \to v_1 \rangle$ only goes downward. We call x the *top node* of the up-down-path because x is the highest up with respect to the hierarchy. Theorem 5.1 guarantees the existence of prefix-optimal EA up-down-paths in TCHs—for all departure times.

THEOREM 5.1. *Let H be a TCH, $s, t \in V$ such that t is reachable from s, and $\tau_0 \in \mathbb{R}$. Then, there is an up-down-path in H, which is also a prefix-optimal (s, t, τ_0)-EA-path.*

PROOF. According to Corollary 2.2, there exists a prefix-optimal (s, t, τ_0)-EA-path $P := \langle v_1 \to_{f_1} \cdots \to_{f_{k-1}} v_k \rangle$ in G. According to Corollary 4.2, P is also a prefix-optimal (s, t, τ_0)-EA-path in H. If P is not an up-down-path, we choose a local minimum of P excluding s and t, that is, we choose $j \in \{2, \ldots, k-1\}$ such that $v_j \prec v_{j-1}, v_{j+1}$. By the prefix-optimality of P, we know that $\langle v_{j-1} \to_{f_{j-1}} v_j \to_{f_j} v_{j+1} \rangle$ is an $(v_{j-1}, v_{j+1}, \mathrm{EA}_G(s, v_{j-1}, \tau_0))$-EA-path in H. So, during the preprocessing, when contracting v_j, a shortcut edge $v_{j-1} \to_h v_{j+1}$ with

$$\operatorname{arr} h\big(\mathrm{EA}_G(s, v_{j-1}, \tau_0)\big) = \operatorname{arr}(f_j * f_{j-1})\big(\mathrm{EA}_G(s, v_{j-1}, \tau_0)\big) = \mathrm{EA}_G(s, v_{j+1}, \tau_0) \,.$$

is inserted (see Lemma 4.1). As a consequence, $Q := \langle v_1 \to_{f_1} \cdots \to_{f_{j-2}} v_{j-1} \to_h v_{j+1} \to_{f_{j+1}} \cdots \to_{f_{k-1}} v_k \rangle$ is a prefix-optimal (s, t, τ_0)-EA-path in H. By setting $P := Q$, the previous argument can be applied again and again until the resulting path P^* is an up-down-path from s to t. Note that this process terminates because, in every step, Q has one edge less than P and a single edge surely is an up-down-path. So, by construction, P^* is an up-down-path as well as a prefix-optimal (s, t, τ_0)-EA-path in H. □

The given proof is similar to the proof of Geisberger et al. [2012], which states that static CH query is correct.

We already said that we do not store TCHs in form of hierarchies of overlay graphs but as single graphs containing all original edges and all shortcuts together with the information whether an edge goes upward or downward in the hierarchy (see Section 4). A TCH $H = (V, E_H)$ can be decomposed into two DAGs $H_\uparrow := (V, E_\uparrow) := (V, \{u \to v \in H \mid u \prec v\} =$ and $H_\downarrow := (V, E_\downarrow) := (V, \{u \to v \in H \mid v \prec u\})$, where $H = H_\uparrow \cup H_\downarrow$ and $E_\uparrow \cap E_\downarrow = \emptyset$ hold. The up-down-paths in H are exactly the paths $\langle s \to \cdots \to x \to \cdots \to t \rangle \subseteq H$ with the property

$$\langle s \to \cdots \to x \rangle \subseteq H_\uparrow \quad \text{and} \quad \langle x \to \cdots \to t \rangle \subseteq H_\downarrow \,. \tag{3}$$

All query algorithms in this work perform bidirectional upward searches to find all relevant EA up-down-paths, but up-down-paths are fully characterized by Property (3). So, $H_\uparrow$ and $H_\downarrow$ are enough to perform the bidirectional upward searches: Forward searches run in the graph $H_\uparrow$ and backward searches in the graph $H_\downarrow$. So, what we really store are the DAGs $H_\uparrow$ and $H_\downarrow$, but the latter is stored with reverted edges.

5.2. EA Queries

Given a start node s, a destination node t, and a departure time τ_0, we want to compute $\mathrm{EA}_G(s, t, \tau_0)$ as well as a corresponding EA path in G. Using TCHs, we can do this very fast: First, we run a *tchEaQuery* (Algorithm 4) on H to compute (s, t, τ_0)-EA up-down-path. Second, we unpack this EA up-down-path recursively for departure time τ_0 to obtain an (s, t, τ_0)-EA path in G (see Section 3). Both steps take very little time.

The *tchEaQuery* (Algorithm 4) runs in two phases. The first phase (Lines 16 to 25) is a time-dependent bidirectional upward search. The forward search is a *tdDijkstra* starting from s running in $H_\uparrow$, the backward search is a *profileIntervalSearch* starting from t running in $H_\downarrow$. Both searches only relax edges that lead upward in the TCH. In case of the backward search, all edges are relaxed in reverse direction. Note that forward and backward search are performed alternately. This is controlled by the variable d, which determines the current direction. In contrast to the forward search, the backward search yields only approximate results. This is because we do not know the EA time, which is just what we want to compute.

The nodes where forward and backward search meet are called *candidate nodes*. More precisely, a candidate node u is a node with $\tau[u] + r[u] < \infty$. We store the candidate nodes in the *candidate set X*. During the bidirectional search, we maintain an upper bound $B \geq \mathrm{EA}_G(s, t, \tau_0)$ with initial value ∞. Every time the two searches meet in a node u, we set $B := \min\{B, \tau[u] + r[u]\}$. The bidirectional search can be stopped as soon as the minima of forward and backward PQ both exceed B (Line 17). The upper bound B can be used to rule out candidate nodes that cannot lie on an (s, t, τ_0)-EA up-down-path (Line 20).

The second phase or *downwardSearch* (Lines 7 to 15) is actually a *tdDijkstra* on the transpose predecessor graph of the backward search, that is $Graph(p_t)^\top \subseteq H_\downarrow$. Note that $Graph(p_t)^\top$ does not need to be build up before we run the *downwardSearch*. Instead, the predecessor information p_t can be used directly. This is reflected in Line 15.

The *downwardSearch* has multiple start nodes, namely the candidate nodes, which are inserted in the PQ Q at Lines 10 and 11. Note that we again use the upper bound B to rule out candidate nodes there.

With the guaranteed existence of an (s, t, τ_0)-EA up-down-path in H (see Theorem 5.1), we can prove that the *tchEaQuery* really computes such an up-down-path.

THEOREM 5.2. *Let H be a TCH, $s, t \in V$ such that t is reachable from s, and $\tau_0 \in \mathbb{R}$. Then, the tchEaQuery (Algorithm 4) returns an (s, t, τ_0)-EA-path in H.*

PROOF. According to Theorem 5.1, there exists an up-down-path with top node x_0 in H, which is also a prefix-optimal (s, t, τ_0)-EA-path. Surely, x_0 is reached both by forward and backward search. Thus, we have $x_0 \in X$, and there is an $(x_0, t, \mathrm{EA}_H(s, x_0, \tau_0))$-EA-path $Q_0 \subseteq \mathit{Graph}(p_t)^\top \subseteq H_\downarrow$. Now, consider the graph G_0 formed by $\mathit{Graph}(p_t)^\top$, the start node s, and by an edge $s \to_{f_x} x$ with $f_x :\equiv \mathrm{EA}_H(s, x, \tau_0) - \tau_0$ for each $x \in X$. Consider the concatenated path $P_0 := \langle s \to_{f_{x_0}} x_0 \rangle Q_0$. As Q_0 is suffix of an (s, t, τ_0)-EA-path in H, we have

$$\mathrm{arr}\, f_{P_0}(\tau_0) = \mathrm{arr}(f_{Q_0} * f_{x_0})(\tau_0) = \mathrm{EA}_H\big(x_0, t, \mathrm{EA}_H(s, x_0, \tau_0)\big) = \mathrm{EA}_H(s, t, \tau_0).$$

Thus, it is $\mathrm{EA}_{G_0}(s, t, \tau_0) = \mathrm{EA}_H(s, t, \tau_0)$, as by construction, there are no better EA times in G_0 than in H for departure time τ_0. But it is easy to see that *downwardSearch* is essentially a *tdDijkstra* in G_0 starting from s. Thus, *tchEaQuery* yields the desired

ALGORITHM 4: EA query using a TCH structure $H = H_\uparrow \cup H_\downarrow$. Given the nodes s and t as well as a departure time τ_0, this algorithm computes an (s, t, τ_0)-EA up-down-path. As subroutines, the subprocedures *tdRelax* and *intervalRelax* from Algorithms 1 and 3 are invoked, respectively.

```
1  function tchEaQuery(s, t : V, τ0 : ℝ) : Path
2      τ[u] := ∞ for all u ∈ V, τ[s] := τ0,                    // initial node labels of forward...
3      [q[u], r[u]] := [∞, ∞] for all u ∈ V, [q[t], r[t]] := [0, 0]   // ...and backward search
4      p_s[u] := ⊥, p_t[u] := ∅ for all u ∈ V                  // initial predecessor information
5      B := ∞, d := t                          // upper bound of EA time and current search direction
6      X := ∅ : Set                                                    // set of candidate nodes
7      function downwardSearch() : Path
8          τ_down[u] := ∞ for all u ∈ V
9          Q := ∅ : PriorityQueue                                      // PQ for downward search
10         foreach u ∈ X do
11             if B < ∞ and τ[u] + q[u] ≤ B then τ_down[u] := τ[u], Q.insert(u, τ_down[u])
12         while Q ≠ ∅ do
13             u := Q.deleteMin()
14             if u = t then return ⟨s = p_s[... p_s[t] ...] → ··· → p_s[t] → t⟩
15             for v ∈ p_t[u] with u →_f v in H_↓^⊤ do tdRelax(u →_f v, τ_down, p_s, Q)
16     Q_s := {(s, τ0)}, Q_t := {(t, 0)} : PriorityQueue          // PQs of forw. and backw. search
17     while (Q_s ≠ ∅ or Q_t ≠ ∅) and min{Q_s.min(), Q_t.min()} ≤ B do
18         if Q_¬d ≠ ∅ then d := ¬d                    // change of direction: ¬s := t and ¬t := s
19         u := Q_d.deleteMin()
20         if B < ∞ and τ[u] + q[u] ≤ B then X := X ∪ {u}
21         B := min{B, τ[u] + r[u]}
22         for u →_f v ∈ E_d do                                    // with E_s := E_↑ and E_t := E_↓^⊤
23             if d = s then tdRelax(u →_f v, τ, p_s, Q_s)
24             else intervalRelax(u, v, [q[u] + min f, r[u] + max f], q, r, p_t, Q_t)
25     return downwardSearch()
```

ALGORITHM 5: A profile query using a TCH structure $H = H_\uparrow \cup H_\downarrow$. For a given start node s and a given destination node t, this algorithm computes $\mathrm{TTP}_G(s,t)$. As, subroutine, the subprocedure *profileRelax* from Algorithm 2 is invoked.

```
1  function tchProfileQuery(s, t : V) : TTF
2      f_s[u] := ∞ for all u ∈ V \ {s}, f_s[s] := ≡ 0
3      f_t[u] := ∞ for all u ∈ V \ {t}, f_t[t] := ≡ 0
4      X := ∅ : Set                                              // candidate set
5      Q_s := {(s, 0)}, Q_t := {(t, 0)} : PriorityQueue          // forw. and backw. PQ
6      B := ∞, d := t                                            // upper bound of travel time, search direction
7      while (Q_s ≠ ∅ or Q_t ≠ ∅) and min{Q_s.min(), Q_t.min()} ≤ B do
8          if Q_¬d ≠ ∅ then d := ¬d                              // with ¬s := t and ¬t := s
9          u := Q_d.deleteMin()
10         if B < ∞ and min f_s[u] + min f_t[u] ≤ B then X := X ∪ {u}
11         B := min{B, max f_s[u] + max f_t[u]}
12         for u → v ∈ E_d with TTF f_uv do                      // with E_s := E_↑ and E_t := E_↓^⊤
13             if d = s then profileRelax(u, v, f_uv ∗ f_s[u], f_s, ⊥, Q_s)
14             else profileRelax(u, v, f_t[u] ∗ f_uv, f_t, ⊥, Q_t)   // ignore predecessor information
15     return min(f_1, min(f_2, ... min(f_{k−1}, f_k) ...)) with {x_1, ..., x_k} := X and f_i := f_t[x_i] ∗ f_s[x_i]
```

result. Note that the stopping condition of the while loop (Line 17) does not affect the correctness. This follows from Lemma 3.1. Also, the ruling out of candidate nodes in Lines 11 and 20 only applies to nodes that do not lie on an (s,t,τ_0)-EA-path. □

5.3. Profile Queries

Given a start node s and a destination node t, we want to know $\mathrm{TTP}_G(s,t)$. In theory, such profile queries can be solved with a one-to-one *profileSearch*. However, for larger road networks, this is not feasible. Instead, we use *tchProfileQuery* (Algorithm 5), which runs much faster as our experiments show (see Section 8). Similar to *tchEaQuery*, it performs a bidirectional search on H, which only goes upward in the hierarchy. Both forward and backward searches are a *profileSearch* and run in $H_\uparrow$ and $H_\downarrow$, respectively. By $f_s[u]$, we denote the label of a node u with respect to forward search. Analogously, we use $f_t[u]$ for the backward search. Of course, the backward search relaxes all edges in reverse direction; we will no longer mention this explicitly in the rest of this document.

Note that an additional downward search is not necessary in this case, as *profileSearch* does not need a fixed departure time. As linking of TTFs is, in general, not commutative, the backward version of *profileSearch* links TTFs the other way round than the forward version (Line 14, also see Section 3). In contrast, the merging of TTFs is commutative. We exploit this to speed up Line 15: By using a further PQ, we control which two TTFs we merge next. First, we insert all f_i in the PQ using $|f_i|$ as key. Second, we repeatedly remove two minimal TTFs f, g from the PQ and insert the result $\min(f,g)$ with key $|\min(f,g)|$ until only one TTF is left. This is the sought-after result.

THEOREM 5.3. *Let H be a TCH and $s,t \in V$. Then, the tchProfileQuery (Algorithm 5) returns* $\mathrm{TTP}_G(s,t)$.

PROOF. From Theorem 5.1, we know that, for all departure times $\tau \in \mathbb{R}$, there is a prefix-optimal (s,t,τ)-EA up-down-path in H with top node x_τ. Surely, x_τ is reached by forward and backward search with the final labels $f_s[x_\tau]$ and $f_t[x_\tau]$, respectively, such that $\mathrm{arr}\, f_s[x_\tau](\tau) = \mathrm{EA}_G(s,x_\tau,\tau)$ and $\mathrm{arr}\, f_t[x_\tau](\mathrm{EA}_G(s,x_\tau,\tau)) = \mathrm{EA}_G(x_\tau,t,\mathrm{EA}_G(s,x_\tau,\tau))$ holds. Together, we have $\mathrm{arr}(f_t[x_\tau] * f_s[x_\tau])(\tau) = \mathrm{EA}_G(s,t,\tau)$, and of course we have $x_\tau \in X =: \{x_1,\ldots,x_k\}$. So, for $x_\tau = x_i$, we know that

ALGORITHM 6: Stall-on-Demand as used by the forward search during the bidirectional phase of *tchEaQuery*. Stalls the forward search at the node u if we manage to prove that the tentative path to u is not an optimal one. If the stalling of u is successful, then the stalling is propagated to further nodes in the manner of a BFS in $H_\uparrow$. The return value is *true* in this case.

```
1  procedure tdPropagateStalling(x : V)
2      F := {x} : FifoQueue                                   // propagate in manner of a BFS
3      while F ≠ ∅ do
4          u := F.popFront()
5          for u →_f v ∈ E↑ do
6              stall_new := arr f(stalled_s[u]) // prune if unreached, already stalled, or stalling fails
               if τ[v] = ∞ or stalled_s[v] < ∞ or τ[v] ≤ stall_new then continue
7              stalled_s[v] := stall_new
8              F.pushBack(v)
9  function tdCanBeStalled(u : V) : {true, false}
10     if stalled_s[u] < τ[u] then return true
11     for w →_f u ∈ E↓ do
12         if τ[w] < ∞ and arr f(τ[w]) < τ[u] then
13             stalled_s[u] := arr f(τ[w])
14             tdPropagateStalling(u)
15             return true
16     return false
```

arr $f_i(\tau) = \text{arr}(f_t[x_\tau] * f_s[x_\tau])(\tau) = \text{EA}_G(s, t, \tau)$. If f is the TTF returned by *tchProfileQuery*, we have $f(\tau) = \min(f_1, \ldots \min(f_i, \ldots \min(f_{k-1}, f_k) \ldots) \ldots)(\tau) = \text{EA}_G(s, t, \tau) - \tau$. Thus, $f = \text{TTP}_G(s, t)$ holds. □

5.4. Stall-on-Demand

The running time of *tchEaQuery* and *tchProfileQuery* can be further improved using a technique called *stall-on-demand*. It was originally developed in the context of highway node routing [Schultes and Sanders 2007] and is also used with static CHs [Geisberger et al. 2008]. The idea is to stop the forward or backward search at nodes where we can easily prove that H contains better paths than $H_\uparrow$ or $H_\downarrow$ alone, respectively. If this happens, we say that a node u is *stalled* regarding the forward or backward search, respectively. In case of the forward search in the *tchEaQuery*, we do this by adding the following pseudocode right after Line 19 of Algorithm 4.

if $d = s$ **and** *tdCanBeStalled*(u) **then continue**

Also, we have to add the initializer

$\text{stalled}_s[u] := \infty$ for all $u \in V$

to Algorithm 4 somewhere between the Lines 2 and 6. The operation *tdCanBeStalled* is defined in Algorithm 6.

When invoked, *tdCanBeStalled* examines the incoming edges of the node u in $H_\downarrow$ trying to find an arrival time which is better than $\tau[u]$. In case of success, the node u is stalled regarding the forward search and the edges of u in $H_\uparrow$ are not relaxed hence. The better arrival time found by *tdCanBeStalled* is stored in $\text{stalled}_s[u]$ (Line 13). If a node u is successfully stalled, we can propagate the stalling to further nodes as a better path to u may be part of a better path to some other node (Lines 1 to 8). The propagation works in the manner of a BFS in $H_\uparrow$ and stops at nodes we fail to stall or

that are already stalled. Note that stall-on-demand can be applied analogously to the backward search in *tchEaQuery*. The only differences in this case are

—that we do not have exact travel times, so we prune in terms of upper and lower bounds instead, and
—that the role of $H_\uparrow$ and $H_\downarrow$ must be exchanged because we search backward.

Stall-on-demand can also be applied to *tchProfileQuery*. There, we use upper and lower bounds both for forward and backward search. Stall-on-demand does not affect the correctness of *tchEaQuery*, *tchProfileQuery*, or any other TCH-based query algorithm in this work. This is because the correctness of our algorithm relies on the existence of *prefix-optimal* up-down paths. Such paths are *never* eliminated by stall-on-demand.

5.5. Using Profile Interval Search and Cones to Accelerate Profile Queries

According to our experiments, *tchProfileQuery* is feasible, but it is still not fast enough (Section 8). Instead, we perform *tchConeProfileQuery* (Algorithm 7), which is essentially a *tchProfileQuery* with a preceding phase. The preceding phase uses bidirectional *profileIntervalSearch* and BFS to extract a relatively small subgraph $C_\uparrow \cup C_\downarrow \subseteq H$ that contains enough nodes and edges for *tchProfileQuery* to compute $\mathrm{TTP}_G(s,t)$ correctly, but not too many other nodes and edges. During the bidirectional *profileIntervalSearch*, we apply stall-on-demand (Lines 8 and 12): $stalled_d[u]$ stores the better upper bound of travel time for a node u with respect to forward or backward search, a function $intervalCanBeStalled(V, \{s,t\})$ performs stalling and propagation in terms of upper and lower bounds, as hinted in Section 5.4 (details are omitted).

Note that $C_\uparrow$ and $C_\downarrow$ can be obtained quite easily: In case of $C_\uparrow$, we just perform a BFS on the transpose predecessor graph $Graph(p_s)^\top$ of the forward search starting from the candidate nodes stored in X. Neither S nor $Graph(p_s)^\top$ need to be built up, but the predecessor information p_s can be used directly. While doing so, we add every touched edge of $Graph(p_s)$ to $C_\uparrow$. For $C_\downarrow$, all this works analogously using $Graph(p_t)$. Note that we do not apply stall-on-demand when we perform the *tchProfileQuery* on $C_\uparrow \cup C_\downarrow$, as we do not expect a significant further speed-up from a stall-on-demand, which also works in terms of upper and lower bounds.

THEOREM 5.4. *Let H be a TCH and $s,t \in V$. Then, the tchConeProfileQuery (Algorithm 7) returns* $\mathrm{TTP}_G(s,t)$.

PROOF. For every $\tau \in \mathbb{R}$, there is an up-down-path with top node x_τ in H, which is a prefix-optimal (s,t,τ)-EA-path in H as Theorem 5.1 tells us. Surely, x_τ is reached both by forward and backward search of *bidirTchProfileIntervalSearch* and thus contained in X. Correspondingly, we have a prefix-optimal (s,x_τ,τ)-EA-path $P_\tau \subseteq Graph(p_s) \subseteq H_\uparrow$ and a prefix-optimal $(x_\tau,t,\mathrm{EA}_G(s,x_\tau,\tau))$-EA-path $Q_\tau \subseteq Graph(p_t)^\top \subseteq H_\downarrow$. As $C_\uparrow$ and $C_\downarrow^\top$ are induced cones, we have $P_\tau \subseteq C_\uparrow$ and $Q_\tau \subseteq C_\downarrow$, respectively. As x_τ lies on a prefix-optimal (s,t,τ)-EA up-down-path, P_τ and Q_τ together form a prefix-optimal (s,t,τ)-EA up-down-path $R_\tau := P_\tau Q_\tau$. But because of $R_\tau \subseteq C_\uparrow \cup C_\downarrow$, we apply Theorem 5.3 with $H_\uparrow := C_\uparrow$ and $H_\downarrow := C_\downarrow$ and are finished. □

6. EXACT SPACE-EFFICIENT QUERYING USING APPROXIMATION

With the techniques described in Section 5, TCHs allow very fast answering of EA queries as well as fast profile queries. Unfortunately, TCHs need a lot of memory compared to the data structure needed to perform *tdDijkstra*. We overcome this problem by the careful use of approximation. More precisely, we generate approximate versions of the hierarchy that need much less space (Section 6.1). For EA queries, we get a

ALGORITHM 7: An improved version of *tchProfileQuery*. It uses bidirectional *profileIntervalSearch* to obtain the cone $C_\uparrow \subseteq H_\uparrow$ and the transpose cone $C_\downarrow \subseteq H_\downarrow$. The expensive *tchProfileQuery* is only performed on $C_\uparrow \cup C_\downarrow \subseteq H$, which contains only few parts of H not needed to compute $\mathrm{TTP}_G(s,t)$. Invokes *intervalRelax* (see Algorithm 3) and *tchProfileQuery* (Algorithm 5). The bidirectional *profileIntervalSearch* applies stall-on-demand based on upper and lower bounds. To do so, it invokes a subprocedure *intervalCanBeStalled*, which works as hinted in Section 5.4.

```
1  function tchConeProfileQuery(s, t : V) : TTF
2      function bidirTchProfileIntervalSearch() : (Graph, Graph)
3          [q_s[u], r_s[u]] := [q_t[u], r_t[u]] := [∞, ∞], p_s[u] := p_t[u] := ∅ for all u ∈ V
4          [q_s[s], r_s[s]] := [q_t[t], r_t[t]] := [0, 0]
5          B := ∞, d := t
6          X := ∅ : Set
7          Q_s := {(s, 0)}, Q_t := {(t, 0)} : PriorityQueue
8          stalled_d[u] := ∞ for all u ∈ V, d ∈ {s, d}
9          while (Q_s ≠ ∅ or Q_t ≠ ∅) and min{Q_s.min(), Q_t.min()} ≤ B do
10             if Q_¬d ≠ ∅ then d := ¬d                    // with ¬s := t and ¬t := s
11             u := Q_d.deleteMin()
12             if intervalCanBeStalled(u, d) then continue
13             if B < ∞ and q_s[u] + q_t[u] ≤ B then X := X ∪ {u}
14             B := min{B, r_s[u] + r_t[u]}
15             for u →_f v ∈ E_d do                        // with E_s := E_↑ and E_t := E_↓^⊤
16                 intervalRelax(u, v, [q_d[u] + min f, r_d[u] + max f], q_d, r_d, p_d, Q_d)
17         Let S := Graph(p_s) ⊆ H_↑ and T := Graph(p_t) ⊆ H_↓^⊤
18         return (Cone_S(s, X), Cone_T(t, X)^⊤)
19     (C_↑, C_↓) := bidirTchProfileIntervalSearch()
20     return tchProfileQuery(s, t) with C_↑ as H_↑ and C_↓ as H_↓, do not apply stall-on-demand
```

moderate slowdown this way, but the results of the computations are still exact (Section 6.3). For profile queries, we also get exact results, but we additionally obtain a further speed-up (Section 6.4).

6.1. Approximated TCHs

To save memory, we use *approximated TCHs* (ATCHs) and *Min-Max-TCHs*. Although these variants of TCHs contain partly approximated data, they can be used to compute exact results. An ATCH with relative error $\varepsilon > 0$ is generated from a given TCH H as follows: For all edges $u \to_f v$ in $G \subseteq H$, nothing happens. All other edges $u \to_f v$ in $H \backslash G$ are shortcut edges and their TTF f is replaced by a TTF $\overline{f}$ with $\forall \tau : f(\tau) \le \overline{f}(\tau) \le (1+\varepsilon) f(\tau)$. We call $\overline{f}$ an upper bound. Implicitly, $\overline{f}$ also represents a lower bound $\underline{f} : \tau \mapsto \overline{f}(\tau)/(1+\varepsilon)$. For all edges $u \to_f v$ in G, we set $\overline{f} := \underline{f} := f$.

Usually, $|\overline{f}|$ is considerably smaller than $|f|$. Thus, an ATCH needs considerably less memory than the corresponding TCH (Section 8). To compute $\overline{f}$ from an exact TTF f, we use an implementation [Neubauer 2009] of an efficient geometric algorithm [Imai and Iri 1987] that yields an $\overline{f}$ of minimal $|\overline{f}|$ for a given ε in time $\mathrm{O}(|f|)$. The computed $\overline{f}$ may violate the FIFO property, but this can be repaired in $\mathrm{O}(|\overline{f}|)$ time. If $\overline{f}$ fulfills the FIFO property, then $\underline{f} = (1+\varepsilon)^{-1}\overline{f}$ also does.

A *Min-Max-TCH* is essentially an extreme case of an ATCH. For edges $u \to_f v$ in $H \backslash G$, we set $\underline{f} :\equiv \min f$ and $\overline{f} :\equiv \max f$, which means we only store a pair of numbers in this case. Min-Max-TCHs need even less memory than ATCHs (Section 8).

Note that we do not apply approximation during preprocessing. In this article, approximation is only applied after preprocessing. Thus, ATCHs and Min-Max-TCHs are generated from complete exact TCHs only.

6.2. Three Basic Algorithms

In order to deal with ATCHs and Min-Max-TCHs we need three further Dijkstra-like algorithms: earliest arrival (EA) interval search, latest departure (LD) interval search, and approximate profile search.

Earliest Arrival Interval Search. The *EA interval search* (*eaIntervalSearch*, Algorithm 8) is similar to *profileIntervalSearch*. For all $u \in V$, it computes intervals that contain $\mathrm{EA}_G(s, u, \tau_0)$ for given $s \in V$, $\tau_0 \in \mathbb{R}$. Hence, the label of a node u is a tentative arrival interval $[q[u], r[u]]$ and, when relaxing an edge $u \to v$ with lower bound $\underline{f}$ and upper bound $\overline{f}$, we update the label of v by computing $[q[v], r[v]] := [\min\{q[v], \operatorname{arr} \underline{f}(q[u])\}, \min\{r[v], \operatorname{arr} \overline{f}(r[u])\}]$. As PQ key of u, we use $q[u]$. In fact, *eaIntervalSearch* is an approximate version of *tdDijkstra* (Algorithm 1). After termination, the predecessor graph graph contains a prefix-optimal (s, u, τ_0)-EA path for all reachable nodes u.

Latest Departure Interval Search. The *latest departure* (LD) interval search is dual to *eaIntervalSearch* (Algorithm 8). It runs backward starting from the target node. So, given a target node t and an arrival interval $[\tau_1, \tau_1']$, it computes intervals $[q[u], r[u]] \supseteq \mathrm{LD}_G(u, t, [\tau_1, \tau_1'])$ for all $u \in V$. Accordingly, the label of a node u is a tentative departure interval $[q[u], r[u]]$. The set of predecessors of u is denoted as $p[u]$. For sake of brevity, we only show the pseudocode of the relaxation procedure (*ldIntervalRelax*, Algorithm 9). Note that late departure times are better than early ones. So, we use a maximum PQ Q here. When relaxing an edge $u \to v$ with lower bound $\underline{f}$ and upper bound $\overline{f}$ in *backward* direction, we update the label and the predecessor information of the node u by invoking $\mathit{ldIntervalRelax}(v, u, [q_{\mathrm{new}}, r_{\mathrm{new}}], q, r, p, Q)$, where $[q_{\mathrm{new}}, r_{\mathrm{new}}] := \left[\min \operatorname{dep} \overline{f}(q[v]), \max \operatorname{dep} \underline{f}(r[v])\right]$ is the departure interval for traveling the edge $u \to v$ with arrival interval $[q[v], r[v]]$. The initial label of the node t is $[\tau_1, \tau_1']$. After termination, the predecessor graph contains a (u, t, τ)-EA-path for all $u \in V$ and all $\tau \in \mathrm{LD}_G(u, t, [\tau_1, \tau_1'])$ (if t is reachable from u).

Approximate Profile Search. The *approximate profile search* is an approximate version of *profileSearch* (Algorithm 2), which is more accurate than *profileIntervalSearch* (Algorithm 3) but also slower. However, it still runs much faster than exact *profileSearch*. Again, for sake of brevity, we only show the pseudocode of the relaxation

ALGORITHM 8: An approximate version of *tdDijkstra*. For a given start node s and a given departure interval $[\tau_0, \tau_0']$, this algorithm computes labels $[q[u], r[u]] \ni \mathrm{EA}_G(s, u, \tau)$ for all reachable nodes $u \in V$ and all $\tau \in [\tau_0, \tau_0']$. As a subroutine, *intervalRelax* from Algorithm 3 is invoked.

```
1 procedure eaIntervalSearch(s : V, [τ0, τ0'] : Interval)
2     [q[u], r[u]] := [∞, ∞], p[u] := ∅ for all u ∈ V
3     [q[s], r[s]] := [τ0, τ0']
4     Q := {(s, τ0)} : PriorityQueue
5     while Q ≠ ∅ do
6         u := Q.deleteMin()
7         for u → v ∈ E with the lower bound f and upper bound f̄ do
8             intervalRelax(u, v, [arr f(q[u]), arr f̄(r[u])], q, r, p, Q)
```

ALGORITHM 9: Edge relaxation procedure of *latest departure interval search*, a Dijkstra-like algorithm computing intervals containing $\mathrm{LD}_G(u, t, [\tau_1, \tau_1'])$ for a given destination node $t \in V$ and a given arrival interval $[\tau_1, \tau_1'] \subseteq \mathbb{R}$. After termination, $Graph(p)^\top \subseteq G$ contains a (u, t, τ)-EA-path for all $\tau \in \mathrm{LD}_G(u, t, [\tau_1, \tau_1'])$ and all nodes u such that t can be reached from u in G.

```
1 procedure ldIntervalRelax(u, v : V, [q_new, r_new] : Interval, q, r, p, Q : Reference)
2     if r_new < q[v] then return
3     if q_new > r[v] then p[v] := ∅
4     p[v] := {u} ∪ p[v]
5     if q_new ≤ q[v] and r_new ≤ r[v] then return
6     [q[v], r[v]] := [ max{q[v], q_new}, max{r[v], r_new} ]
7     if v ∉ Q then Q.insert(v, q[v])
8     else Q.increaseKey(v, q[v])        // for maximum PQs keys are increased, not decreased
```

ALGORITHM 10: Edge relaxation procedure of *approximate profile search*. Given an $s \in V$, this Dijkstra-like algorithm computes pairs of bounding functions $(\underline{f}[u], \overline{f}[u])$, i.e., $\underline{f}[u](\tau) \leq \mathrm{TTP}_G(s, u)(\tau) \leq \overline{f}[u](\tau)$ holds for all $\tau \in \mathbb{R}$. After termination, $Graph(p) \subseteq G$ contains an (s, u, τ)-EA-path for all reachable nodes u and all $\tau \in \mathbb{R}$.

```
1 procedure approxProfileRelax(u, v : V, g_new, ḡ_new : TTF, f, f̄, p, Q : Reference)
2     if g_new(τ) ≥ f̄[v](τ) for all τ ∈ ℝ then return
3     if ḡ_new(τ) < f[v](τ) for all τ ∈ ℝ then p[v] := ∅
4     (f[v], f̄[v]) := ( min(f[v], g_new), min(f̄[v], ḡ_new) )
5     p[v] := {u} ∪ p[v]
6     if v ∉ Q then Q.insert(v, min f[v])
7     else Q.decreaseKey(v, min f[v])
```

procedure here (*approxProfileRelax*, Algorithm 10). For all $u \in V$, the approximate profile search computes pairs of TTFs $(\underline{f}[u], \overline{f}[u])$ that fulfill $\underline{f}[u](\tau) \leq \mathrm{TTP}_G(s, u)(\tau) \leq \overline{f}[u](\tau)$ for all $\tau \in \mathbb{R}$. Consider the relaxation of an edge $u \to v$ with lower bound $\underline{f}$ and upper bound $\overline{f}$. Let the pairs of tentative bounding functions $(\underline{f}[u], \overline{f}[u])$ and $(\underline{f}[v], \overline{f}[v])$ be the labels of the nodes u and v, respectively. Then, we update the label of v by invoking $approxProfileRelax(u, v, \ \underline{f} * \underline{f}[v], \overline{f} * \overline{f}[v]), \underline{f}, \overline{f}, p, Q)$ for Q being the PQ. The final predecessor graph $Graph(p)$ of the approximate profile search contains a prefix-optimal (s, u, τ)-EA path for all reachable nodes u and all $\tau \in \mathbb{R}$ after termination.

6.3. Exact EA Queries with ATCHs

Given an ATCH H with relative error ε, we want to compute the *exact* value of $\mathrm{EA}_G(s, t, \tau_0)$ and a corresponding EA path. To do so, we use the *atchEaQuery* (Algorithm 11), which works in three phases. Actually, we have to perform a *tdDijkstra* on G, as we have exact TTFs only for the edges in $G \subseteq H$. But this would be slow, so the idea is to restrict the *tdDijkstra* to a very small corridor $C \subseteq G$. For this reason, we first use the ATCH H to perform several approximate Dijkstra-like searches one after another to select and successively thin out a corridor in G (Phases 1 and 2). Then, we perform a *tdDijkstra* only on this corridor (Phase 3). Note that the pseudocode of *atchEaQuery* invokes *intervalRelax* (see Algorithm 3), *ldIntervalRelax* (Algorithm 9), and *tdDijkstra* (Algorithm 1) as subroutines.

ALGORITHM 11: EA query using an ATCH H with relative error $\varepsilon > 0$. After some preparatory work, a *tdDijkstra* on a small corridor $C \subseteq G$ is performed where exact TTFs are available. As C contains an (s, t, τ_0)-EA-path, the result is a sought-after EA path.

```
1  function atchEaQuery(s, t : V, τ0 : ℝ) : Path
2      [q_d[u], r_d[u]] := [∞, ∞], p_d[u] := ∅ for all u ∈ V, d ∈ {s, t, down, up}
3      [q_s[s], r_s[s]] := [τ0, τ0], [q_t[t], r_t[t]] := [0, 0]
4      X := Y := ∅ : Set
5      procedure bidirectionalSearch()
6          Q_s := {(s, τ0)}, Q_t := {(t, 0)} : PriorityQueue
7          B := ∞, d := t
8          while (Q_s ≠ ∅ or Q_t ≠ ∅) and min{Q_s.min(), Q_t.min()} ≤ B do
9              if Q_¬d ≠ ∅ then d := ¬d                          // with ¬s := t, ¬t := s
10             u := Q_d.deleteMin()
11             if B < ∞ and q_s[u] + q_t[u] ≤ B then X := X ∪ {u}
12             B := min{B, r_s[u] + r_t[u]}
13             for u → v ∈ E_d with bounds f and f̄ do          // with E_s := E_↑, E_t := E_↓^⊤
14                 if d = s then intervalRelax(u, v, [arr f(q_s[u]), arr f̄(r_s[u])], q_s, r_s, p_s, Q_s)
15                 else intervalRelax(u, v, [q_t[u] + min f, r_t[u] + max f̄], q_t, r_t, p_t, Q_t)
16     procedure eaIntervalDownwardSearch()
17         Q := ∅ : PriorityQueue
18         foreach u ∈ X do
19             if r_s[u] + r_t[u] ≤ B then
20                 [q_down[u], r_down[u]] := [q_s[u], r_s[u]], Q.insert(u, q_down[u])
21         while Q ≠ ∅ do
22             u := Q.deleteMin()
23             if u = t then break
24             for v ∈ p_t[u] with f and f̄ being the bounds of u → v in H_↓ do
25                 intervalRelax(u, v, [arr f(q_down[u]), arr f̄(r_down[u])], q_down, r_down, p_down, Q_down)
26     procedure upwardSearch()
27         [q_up[t], r_up[t]] := [q_down[t], r_down[t]]
28         Q := {(t, q_up[t])} : MaxPriorityQueue
29         while Q ≠ ∅ do
30             u := Q.deleteMax()          // from a maximum PQ we take out a maximal element
31             if r_up[u] < ∞ then Y := Y ∪ {u}
32             for v ∈ p_down[u] with f and f̄ being the bounds of u → v in H_↓^⊤ do
33                 [q_new, r_new] := [min dep f̄(q_up[u]), max dep f(r_up[u])]
34                 if r_new < q_down[v] or q_new > r_down[v] then continue
35                 ldIntervalRelax(u, v, [q_new, r_new], q_up, r_up, p_up, Q)
36     function corridorDijkstra() : Path
37         let S := Graph(p_s) ⊆ H_↑ and T := Graph(p_up) ⊆ H_↓^⊤
38         let C := Cone_S(s, Y) ∪ Cone_T(t, Y)^⊤ ⊆ H
39         run tdDijkstra(s, t, τ0) on graph C unpacking each edge u → v for departure time τ[u]
40         return ⟨s = p[... p[t] ...] → ··· → p[p[t]] → p[t] → t⟩ with p taken from tdDijkstra
41     bidirectionalSearch()
42     eaIntervalDownwardSearch()
43     upwardSearch()
44     return corridorDijkstra()
```

Phase 1: Bidirectional Search. At first, we perform a bidirectional search, where the forward search is an *eaIntervalSearch*, starting from s, and the backward search is a *profileIntervalSearch*, starting from t. All candidate nodes (i.e., the meeting points) are stored in the candidate set X. With $S := Graph(p_s) \subseteq H_\uparrow$ (as in Line 37), we know that $Cone_S(s, X) \cup Cone_{Graph(p_t)}(t, X)^\top$ contains an (s, t, τ_0)-EA up-down-path. So, we could skip Phase 2 and continue directly with Phase 3. However, the backward search is a relatively rough approximation. So, we expect that $Graph(p_t) \subseteq H_\downarrow^\top$ and X—and thus $Cone_S(s, X) \cup Cone_{Graph(p_t)}(t, X)^\top$—are larger than necessary, which means that Phase 3 needs more time than necessary. To remedy this, we perform Phase 2.

Phase 2: Thinning. Starting from the candidate nodes in X, we perform the *eaIntervalDownwardSearch*, which is an *eaIntervalSearch* on $Graph(p_t)^\top \subseteq H_\downarrow$. This yields an arrival interval $[q_{\text{down}}[t], r_{\text{down}}[t]] \ni \text{EA}_G(s, t, \tau_0)$. Having done that, we use this arrival interval to perform the *upwardSearch*, an LD interval search starting from t that runs on $Graph(p_{\text{down}})^\top \subseteq H_\downarrow$. The *upwardSearch* reaches nodes that have already been reached by the forward search during the first phase. These meeting points are again candidate nodes that we store in the set $Y \subseteq X$. Note that *eaIntervalDownwardSearch* and *upwardSearch* have much more accurate information at hand than the backward search had during the first phase. Also, the conditions in Line 34 should rule out several nodes. The predecessor graph $Graph(p_{\text{up}})$ should contain less edges than $Graph(p_t)$ and the set Y should be smaller than X. Thus, $C = Cone_S(s, Y) \cup Cone_T(t, Y)$ should be significantly thinner than $Cone_S(s, X) \cup Cone_{Graph(p_t)}(t, X)$.

Phase 3: Corridor Dijkstra. To construct the corridor C (Line 38), we do the same as in case of the *tchConeProfileQuery* (see Algorithm 7): Starting from the nodes in the candidate set Y, we perform a BFS on $Graph(p_s)^\top$ and $Graph(p_{\text{up}})^\top$, respectively, and copy all touched edges to $C \subseteq H$. Now, we could recursively unpack all shortcuts in C to obtain a subgraph $C' \subseteq G$ that does not contain any shortcuts. Of course, all TTFs in C' would be exact as all edges edges of C' would lie in G and only edges of $H \setminus G$ have inexact TTFs. So, we could perform a *tdDijkstra* in C' to compute $\text{EA}_G(s, t, \tau_0)$ and a corresponding EA path. However, although this works fast, we can still be a little faster: We perform the *tdDijkstra* directly on C and unpack the shortcuts only on demand. More precisely, whenever we relax a shortcut edge $u \to v$ in $H \setminus G$ (for which we do not have exact TTFs), we unpack only this shortcut and only for the departure time $\tau[u]$ (where $\tau[u]$ is the label of u with respect to *tdDijkstra*). The resulting original path $\langle u = w_1 \to_{f_1} \cdots \to_{f_{k-1}} w_k = v \rangle$ lies completely in G, so all TTFs are exact and we can update the label of v by computing $\tau[v] := \min\{\tau[v], \text{arr } f_{k-1}(\ldots \text{arr } f_2(\text{arr } f_1(\tau_u)) \ldots)\}$.

THEOREM 6.1. *Let H be an ATCH with relative error $\varepsilon > 0$. Then, with $s, t \in V$, $\tau_0 \in \mathbb{R}$, the atchEaQuery (Algorithm 11) returns an (s, t, τ_0)-EA-path in G.*

PROOF. Theorem 5.1 ensures the existence of a prefix-optimal (s, t, τ_0)-EA up-down-path with top node $x_0 \in X$ in H. So, there is an (s, x_0, τ_0)-EA-path P_0 in H with $P_0 \subseteq S = Graph(p_s) \subseteq H_\uparrow$. We argue similarly as in the case of Theorem 5.2 and consider the graph G_0 consisting of $Graph(p_t)^\top \subseteq H_\downarrow$, the node s, and the edges $s \to x$ with $x \in X$. As TTF of $s \to x$, we define $f_{sx} :\equiv \text{EA}_G(s, x, \tau_0)$ with $\underline{f}_{sx} :\equiv q_s[x] - \tau_0$ and $\overline{f}_{sx} :\equiv r_s[x] - \tau_0$ as lower and upper bound, respectively. We find that

$$\text{EA}_G(s, t, \tau_0) = \text{EA}_{G_0}(s, t, \tau_0) = \text{EA}_{Graph(p_t)^\top}\big(x_0, t, \text{EA}_G(s, x_0, \tau_0)\big)$$

holds. As the *eaIntervalDownwardSearch* is essentially the same as an *eaIntervalSearch* on G_0 we have $\text{EA}_G(s, t, \tau_0) \in [q_{\text{down}}[t], r_{\text{down}}[t]]$. So, as the *upwardSearch* runs on $Graph(p_{\text{down}})^\top$, it also reaches x_0, and its transpose predecessor graph $T^\top = Graph(p_{\text{up}})^\top$ contains an (x_0, t, τ)-EA-path for all $\tau \in \text{LD}_G(x_0, t, [q_{\text{down}}[t], r_{\text{down}}[t]])$.

Especially with

$$\mathrm{LD}_G\big(x_0, t, \mathrm{EA}_G(s, t, \tau_0)\big) = \mathrm{LD}_G\big(x_0, t, \mathrm{EA}_G\big(x_0, t, [q_{\mathrm{down}}[t], r_{\mathrm{down}}[t]]\big)\big) \ni \mathrm{EA}_G(s, x_0, \tau_0),$$

we know that there is an $(x_0, t, \mathrm{EA}_G(s, x_0, \tau_0))$-EA-path $Q_0 \subseteq T^\top \subseteq H_\downarrow$. So, because of $x_0 \in Y$, we have $Q_0 \subseteq \mathit{Cone}_T(t, Y)^\top$. Together with $P_0 \subseteq \mathit{Cone}_S(s, Y)$, this shows that the concatenation $P_0 Q_0 \subseteq H$ is an (s, t, τ_0)-EA-path, which is contained in C. Thus, *corridorDijkstra* computes the desired result. □

To make *atchEaQuery* run even a little faster, we apply stall-on-demand to its first phase analogously to the description in Section 5.4, though we omit all details here.

Note that *atchEaQuery* also works with Min-Max-TCHs. But in this case, the *upwardSearch* does not have more information than the backward search at hand. So, the extra work needed to perform *eaIntervalDownwardSearch* and *upwardSearch* does not pay off on this case. For this reason, we omit the second phase and set $C := \mathit{Cone}_S(s, X) \cup \mathit{Cone}_{\mathit{Graph}(p_t)}(t, X)^\top$ for Min-Max-TCHs.

6.4. Exact Profile Queries with ATCHs

An ATCH H with relative error ε can also be used to compute $\mathrm{TTP}_G(s, t)$ exactly. This is done by the *atchProfileQuery* (Algorithm 12). It applies similar ideas as the *atchEaQuery* (Algorithm 11) and also runs in three phases. In the first phase (Line 23), we perform a bidirectional *profileIntervalSearch* to obtain a cone $C_\uparrow \subseteq H_\uparrow$ and a transpose cone $C_\downarrow \subseteq H_\downarrow$. Then, in the second phase (Lines 24 to 32), we thin out these cones by a bidirectional approximate profile search. This is similar to the second phase of *atchEaQuery*. A difference is that we thin out not only the backward cone but also the forward cone here.

Finally, in the third phase, we unpack all the shortcuts in $\mathit{Cone}_S(s, X) \cup \mathit{Cone}_T(t, X)^\top$ completely for all departure times (Line 33). This yields the relatively thin corridor $C \subseteq G$ from s to t. To extract $\mathit{Cone}_S(s, X)$ and $\mathit{Cone}_T(t, X)$, we perform BFS starting from X on S and T respectively. This is analogously to the third phase of *atchEaQuery* (Section 6.3).

It would be possible to perform a *profileSearch* on C, as all edges in C have exact TTFs. However, this would still take its time, even on a very thin corridor. Instead, we contract the whole corridor C (Line 34), which is much faster. Contracting a corridor means to contract one node of the corridor after another. More precisely, we remove one node of C after another while inserting shortcuts. This is very similar to the preprocessing where we contract the whole road network G (see Section 4). However, the contraction of a corridor is performed online and not offline as in case of preprocessing. For this reason, we do not have the time to check whether a shortcut is necessary or not. Instead, we simply insert all, shortcuts.

During corridor contraction, we control the order of the nodes by a PQ. As PQ key of a node x, we use

$$c_x := \sum_{\langle u \to_f x \to_g v \rangle \subseteq C} |f| + |g| \,.$$

As we expect that $|g| + |f|$ and $|g * f|$ are quite correlated, we think c_x is a good estimate of the total complexity of the TTFs created by contracting x. In this way, we try to process TTFs with few segments earlier. This speeds up the computation, as every segment probably produces more segments during repeated link and merge operations. But this is exactly why *profileSearch* is slow: There, nodes are processed in an order that does not pay attention to the number of newly created segments which increases the total number of processed segments a lot.

ALGORITHM 12: Profile query using an ATCH with $\varepsilon > 0$. After building up the cones $C_\uparrow$ and $C_\downarrow$ and then thinning them, we obtain a corridor $C \subseteq G$ that contains an (s,t,τ)-EA-path in G for all $\tau \in \mathbb{R}$. Contracting C completely yields $\mathrm{TTP}_G(s,t)$. As subroutines, *bidirTchProfileIntervalSearch* (see Algorithm 7) and *approxProfileRelax* (Algorithm 10) are invoked.

```
1  function atchProfileQuery(s, t : V, τ0 : ℝ) : TTF
2      function corridorContraction(s, t : V, C : Graph) : TTF
3          Q := ∅ : PriorityQueue
4          foreach node x of C with x ≠ s, t do
5              c_x := Σ_{⟨u →f x →g v⟩ ⊆ C} |f| + |g|   // estimate the total complexity of the new TTFs...
6              Q.insert(x, c_x)                          // ... and use the result as PQ key
7          while Q ≠ ∅ do
8              x := Q.deleteMin()                        // contract the presumably easiest node next
9              foreach path ⟨u →f x →g v⟩ ⊆ C do
10                 remove u → x and x → v from C
11                 if u → v not in C then insert an edge u →g∗f v in C    // insert an edge or...
12                 else replace u →h v by u →min(h,g∗f) v in C            // ... merge with an existing one
13             remove x from C
14         for all former neighbors y of x with y ≠ s, t do
15             c_y := Σ_{⟨u →f y →g v⟩ ⊆ C} |f| + |g|     // recompute the PQ key of all neighbors of x
16             Q.updateKey(y, c_y)
17         return f for s →f t in C      // at this point C consists exactly of one edge from s to t
18     f_s[u] := f̄_s[u] := f_t[u] := f̄_t[u] := ∞, p_s[u] := p_t[u] := ∅ for all u ∈ V
19     f_s[s] := f̄_s[s] := f_t[t] := f̄_t[t] ≡ 0
20     Q_s := {(s, 0)}, Q_t := {(t, 0)} : PriorityQueue
21     X := ∅ : Set
22     B := ∞, d := t
23     (C↑, C↓) := bidirTchProfileIntervalSearch()
24     while (Q_s ≠ ∅ or Q_t ≠ ∅) and min{Q_s.min(), Q_t.min()} ≤ B do
25         if Q_¬d ≠ ∅ then d := ¬d                      // with s := ¬t, t := ¬s
26         u := Q_d.deleteMin()
27         if B < ∞ and min f_s[u] + min f_t[u] ≤ B then X := X ∪ {u}
28         B := min{B, max f̄_s[u] + max f̄_t[u]}
29         for edge u → v in C_d with bounds f_uv and f̄_uv do   // with C_s := C↑, C_t := C↓^⊤
30             if d = s then approxProfileRelax(u, v, f_uv ∗ f_s[u], f̄_uv ∗ f̄_s[u], f_s, f̄_s, p_s, Q_s)
31             else approxProfileRelax(u, v, f_t[u] ∗ f_uv, f̄_t[u] ∗ f̄_uv, f_t, f̄_t, p_t, Q_t)
32     let S := Graph(p_s) ⊆ C↑ ⊆ H↑ and T := Graph(p_t) ⊆ C↓^⊤ ⊆ H↓^⊤
33     Unpack Cone_S(s, X) ∪ Cone_T(t, X)^⊤ completely for all departure times yielding C ⊆ G.
34     return corridorContraction(s, t, C)
```

As an example, consider a corridor from s to t that only consists of a single path from s to t that has ℓ edges. Assume every TTF of this path has k segments. If we perform a *profileSearch* starting from s on this path, the number of processed segments is in $\mathrm{O}(\sum_{i=1}^{\ell} ik) = \mathrm{O}(k\ell^2)$. If we contract the path using *corridorContraction* instead, we rather expect that $\mathrm{O}(k\ell \log \ell)$ segments are processed.

THEOREM 6.2. *Let H be an ATCH with relative error $\varepsilon > 0$. Then, for all $s,t \in V$, the atchProfileQuery (Algorithm 12) returns* $\mathrm{TTP}_G(s,t)$.

PROOF. We argue as in the correctness proof of the *tchConeProfileQuery* (see Theorem 5.4) and obtain that $C_\uparrow \cup C_\downarrow$ contains a prefix-optimal (s, t, τ)-EA up-down-path R_τ with top node x_τ for all $\tau \in \mathbb{R}$. The upward and the downward part of R_τ, respectively, lie completely in $C_\uparrow$ and $C_\downarrow$. So, both forward and backward search of the bidirectional approximate profile search on $C_\uparrow \cup C_\downarrow$ reach x_τ. This implies $x_\tau \in X$. Hence, we find that $S \cup T^\top$ also contains a prefix-optimal (s, t, τ)-EA up-down-path R'_τ. Clearly, R'_τ is contained in $Cone_S(s, X) \cup Cone_T(t, X)^\top$. Thus, C contains an (s, t, τ)-EA-path for all $\tau \in \mathbb{R}$. So, *corridorContraction* returns the sought-after TTP. □

In case of Min-Max-TCHs, we omit the thinning of $C_\uparrow$ and $C_\downarrow$. Instead, we unpack $C_\uparrow \cup C_\downarrow$ and perform *corridorContraction* directly after *bidirTchProfileIntervalSearch*. This is because there is no information present in a Min-Max-TCHs to further reduce the size of $C_\uparrow$ and $C_\downarrow$.

7. INEXACT QUERYING

In practice, the accuracy of the input data may be arguable, or results with some error may simply be good enough. In such cases, we can use an inexact TCH with relative error $\varepsilon > 0$. It is generated from an exact TCH H by replacing all TTFs f by an inexact TTF $f_\approx$ with $(1+\varepsilon)^{-1} f(\tau) \leq f_\approx(\tau) \leq (1+\varepsilon) f$ for all $\tau \in \mathbb{R}$. To compute $f_\approx$ from f, we use the Imai-Iri algorithm and restore the FIFO property if necessary (as in Section 6.1). Note that we not only replace the TTFs of the shortcut edges—as we do in case of ATCHs—but also the TTFs of original edges. Also, we annotate every edge $u \to_{f_\approx} v$ in an inexact TCH with the conservative bounds $\min\{\min f, \min f_\approx\}$ and $\max\{\max f, \max f_\approx\}$. Inexact TCHs have similar memory usage as ATCHs.

With inexact TCHs, we can perform both inexact EA and inexact profile queries. However, inexact profile queries are the kind of inexact queries we are actually interested in, as with *atchEaQuery*, we already have a space-efficient algorithm that is fast enough. For profile queries, in contrast, we get an enormous further speed-up with inexact TCHs. Yet, the benefit of our inexact EA query algorithm is that we can use the same hierarchical representation of the road network for both EA and profile queries. Otherwise, we had to keep an ATCH and an inexact TCH in memory at the same time.

Note that the relative error of the computed inexact results is not limited by ε. In theory, the relative error can even get quite large. As a consequence, our correctness proofs for inexact querying (Theorems 7.1 and 7.2) only show that the algorithms return some path (or some TTF, respectively) if the destination is reachable from the start. However, in our experiments, we observe only small errors (Section 8). Note that Geisberger and Sanders [2010] give an upper bound for the relative error depending on the maximum slopes of TTFs. However, we do not exploit this so-called *approximation guaranty* here.

7.1. Inexact Profile Queries

With inexact TCHs, we only get an approximation of $\mathrm{TTP}_G(s, t)$. But compared to *atchProfileQuery* (Algorithm 12), which is our fastest algorithm for exact profile queries, we get an enormous speed-up (see Section 8). To achieve this performance, we simply apply *tchConeProfileQuery* (Algorithm 7) to the inexact TCH. The only modification is, that the stall-on-demand applied by the subprocedure *bidirTchProfileIntervalSearch* (Line 19) must be used with the conservative bounds instead of the bounds $\min f_\approx$ and $\max f_\approx$. Otherwise, it can happen that stall-on-demand stalls so many nodes that forward and backward search do not meet—even if t *is* reachable from s. Then, the query would return ∞, which means that the algorithm did not find any path from s to t.

THEOREM 7.1. *Let H be an inexact TCH with relative error $\varepsilon > 0$. Given a start node s and a destination node t, the inexact profile query just described returns a TTF $f \neq \infty$, if and only if t is reachable from s in G.*

PROOF. If t is not reachable from s, then H does not contain a path from s to t. Otherwise, H contains an up-down-path P_τ from s to t with top node x_τ for every $\tau \in \mathbb{R}$ such that P_τ is a prefix-optimal (s, t, τ)-EA-path in the corresponding exact TCH H. This and the fact that *bidirTchProfileIntervalSearch* works in terms of the conservative bounds implies that $C_\uparrow \cup C_\downarrow$ also contains an up-down-path with top node x_τ, which is a prefix-optimal (s, t, τ)-EA-path in H for all $\tau \in \mathbb{R}$. But this means that the forward and the backward search of *tchProfileQuery* surely meet in one or more candidate nodes and that the returned TTF is different from ∞. □

7.2. Inexact EA Queries

For inexact EA queries, we perform *tchConeEaQuery* (Algorithm 13) on an inexact TCH. It is actually a variant of *tchEaQuery* (Algorithm 4) where not only the backward search but also the forward search is a *profileIntervalSearch* (see *bidirTchProfileIntervalSearch*, Algorithm 7). However, as the *downwardSearch* requires arrival times for the candidate nodes instead of travel time intervals, we have to perform an additional *tdDijkstra* on $C_\uparrow$ as an upward search before invoking *downwardSearch*.

The *tchConeEaQuery* yields exact EA times when applied to an exact TCH. For inexact TCHs, it returns an up-down-path from the start s to the destination t if and only if t is reachable from s. But this requires that the stall-on-demand performed by the *bidirTchProfileIntervalSearch* works in terms of conservative bounds—just like in the case of inexact profile queries (Section 7.1). Otherwise, we may again stall too many nodes and no path from s to t may be found even if there is one. In fact, the only reason for using *bidirTchProfileIntervalSearch* as the first phase is to make stall-on-demand applicable to EA queries in the presence of inexact data.

Let $T := \mathit{Graph}(p_t) \subseteq H_\downarrow^\top$ be the predecessor graph of the backward search of *bidirTchProfileIntervalSearch*. The *tchConeEaQuery* does not need the extraction of the transpose backward cone $C_\downarrow := \mathit{Cone}_T(t, X)^\top \subseteq H_\downarrow$ performed in Line 18 of Algorithm 7.

ALGORITHM 13: A variant of *tchEaQuery* (Algorithm 4) using similar methods as *tchConeProfileQuery* (Algorithm 7). For exact TCHs, it returns the exact EA time as well as a corresponding EA up-down-path. For inexact TCHs, it yields an inexact EA time as well as a not necessary optimal up-down-path. As subroutines, it invokes *bidirTchProfileIntervalSearch* (see Algorithm 7), *tdRelax* (see Algorithm 1), and *downwardSearch* (see Algorithm 4).

```
1  function tchConeEaQuery(s, t : V, τ0 : ℝ) : Path
2      (C↑, ·) := bidirTchProfileIntervalSearch()          // bidirectional search
3      Let B be the final value as used in bidirTchProfileIntervalSearch
4      Let q_t[u], p_t[u] be the final values as used in bidirTchProfileIntervalSearch for all u ∈ V
5      p[u] := ⊥, τ[u] := τ_down[u] := ∞ for all u ∈ V, τ[s] := τ0
6      Q := {(s, τ0)} : PriorityQueue                      // upward search
7      Y := ∅ : Set                                        // candidate set
8      while Q ≠ ∅ do
9          u := Q.deleteMin()
10         if τ[u] > B + τ0 then break      // B is upper bound of travel time not of arrival time
11         if τ[u] + q_t[u] ≤ B + τ0 then Y := Y ∪ {u}
12         for u →_f v in C↑ do tdRelax(u →_f v, τ, p, Q)
13     return downwardSearch() with q_t as q and Y as candidate set   // invoked from
       Algorithm 4
```

Table I. Some Properties of Our Input Graphs and the Available TTF Sets
The percentage of time-dependent (i.e., nonconstant) TTFs, the average relative (avg. rel.) delay of the TTFs including (incl.) and excluding (excl.) the constant (const.) ones, as well as the space usage in byte per node (B/n).

	Germany					Europe	
	Monday	midweek	Friday	Saturday	Sunday	medium	high
time-dependency	7.0%	7.2%	6.3%	3.8%	2.4%	1.0%	6.2%
avg. rel. delay (incl. const.)	2.5%	2.6%	2.2%	1.2%	0.7%	1.2%	7.7%
avg. rel. delay (excl. const.)	36.1%	36.0%	34.9%	31.7%	29.1%	123.5%	124.0%
space [B/n]	96	95	88	65	55	47	76

This is because *downwardSearch* uses the predecessor information of the backward search directly. As a consequence, we can omit the BFS on T when performing *bidirTch-ProfileIntervalSearch*.

THEOREM 7.2. *Let H be an inexact TCH with relative error $\varepsilon > 0$ and $s, t \in V$. Let $\tau_0 \in \mathbb{R}$ be the departure time. Then, tchConeEaQuery (Algorithm 13) computes a (not necessary optimal) up-down-path in H, if and only if t is reachable from s in G. For $\varepsilon = 0$ (i.e., the TCH is exact), it computes an (s, t, τ_0)-EA up-down-path.*

PROOF. If t is not reachable from s, then there is no path from s to t in H. Otherwise, we argue as in the proof of Theorem 7.1 and find that $C_\uparrow \cup C_\downarrow$ contains an up-down-path with top node y_0 which is also a prefix-optimal (s, t, τ_0)-EA-path in H. So, the candidate set Y is surely not empty after the upward search, and hence the *downwardSearch* reaches t. Thus, an arrival time other than ∞ and an up-down-path from s to t is returned.

For $\varepsilon = 0$, we argue that y_0 is reached by the upward search with final label $\mathrm{EA}_G(s, y_0, \tau_0)$, and as in the correctness proof of *tchEaQuery* (see Theorem 5.2), we argue that the *downwardSearch* reaches t with final label $\mathrm{EA}_G(s, t, \tau_0)$. □

8. EXPERIMENTS

8.1. Input Road Networks

As inputs, we use two road networks, of Germany and Western Europe, both provided by PTV AG for scientific use. For Germany, which has about 4.7 million nodes and about 10.8 million edges, we have five sets of time-dependent edge weights collected from historical data: They reflect the traffic of Monday, midweek (Tuesday through Thursday), Friday, Saturday, and Sunday each and have different percentage of time-dependent (i.e., nonconstant) TTFs (Table I). For Saturday and Sunday, for example, there are fewer nonconstant TTFs. This seems to be natural, as we expect less traffic at the weekend than during the week. We use the abbreviations Mon, mid, Fri, Sat, and Sun.

For Western Europe, which has about 18 million nodes and about 42.6 million edges, we have two sets of edge weights. In both sets, all nonconstant TTFs are synthetically generated, as described by Nannicini et al. [2008]. The first set reflects a medium (med) amount of traffic where motorways and national roads can be congested, that is, only motorways and national roads have non-constant TTFs. Urban streets, local streets, and rural roads are time-independent, that is, have constant TTFs. The second set reflects a high amount of traffic. There, not only motorways and national roads but also urban roads have nonconstant TTFs (see [Delling 2009]).

Table I also reports the average relative delay of all sets of time-dependent edge weights (i.e., of all sets of TTFs). The *relative delay* of a TTF f is defined as $(\max f - \min f)/\min f$. Table I reports the average relative delay over all TTFs, both including and excluding the constant ones. Note that Delling [2011] also reports an

average relative delay, but over EA paths, not over edges. We think the relative delay is a good indicator of how well pruning techniques work in the context of *profileIntervalSearch*. Remember that *profileIntervalSearch* uses travel time intervals as node labels computed on the basis of $\min f$ and $\max f$ for TTFs f. So, larger delays result in wider intervals, which means that intervals are more likely to overlap. As a consequence, predecessor sets are less often emptied (see Algorithm 3, Line 6). Also, stall-on-demand (Section 5.4) is more likely to fail for wider intervals. As all our query algorithms make use of *profileIntervalSearch*, their running time should increase with the relative delay of the TTF sets. Our experiments confirm this. This also applies to the preprocessing, which utilizes *profileIntervalSearch* as well. Note that the percentage of time-dependent TTFs also influences the running times—especially when *profileSearch* is involved.

8.2. Experimental Setup

The experimental evaluation is done on a machine with two Core i7 Quad-Cores (2.67GHz) with 48GiB of RAM running SUSE Linux 11.1. All programs are compiled by GCC 4.3.2 with optimization level 3. Running times are always measured using one single thread if not stated otherwise. All figures refer to the scenario that only the EA times and the TTPs have to be determined, without outputting complete path descriptions. But when reporting memory consumption, we include the space needed to allow fast path reporting. Memory usage is always given in terms of the total space usage in average byte per node. Table I reports the memory usage of all input road networks as original graphs (i.e., the graphs used for *tdDijkstra*). For TCHs, ATCHs, Min-Max-TCHs, and inexact TCHs, we also report the memory overhead compared to the original graph, as a growth factor and partly in byte per node.

We measure the average running time of the different EA query algorithms by performing a bulk of 1,000 queries each. Therein, all the triples (s, t, τ_0) of start s, destination t, and departure time τ_0 are selected randomly from $V \times V \times [0\text{h}, 24\text{h})$. For profile queries, a departure time is not necessary of course. Note that we always execute such a bulk of queries three times and report the median of the three average running times as a result. We do this to prevent accidental outliers, which we observed in the past. Occasional activities triggered by the operating system during our experiments may be a possible source.

To measure the errors, we use many more test cases: 1,000,000 random EA queries and 10,000 random profile queries, where the error of profile queries is measured for 100 random departure times each. We always report the average and the maximum relative error. We also measure the machine-independent behavior of our algorithms: In all cases, we count the number of deleteMin operations and of touched edges. For *tdDijkstra*, the number of touched edges is identical to the number of relaxed edges. For other query algorithms, this also includes the number of edges copied by BFSes, of unpacked shortcuts, and of edges processed during corridor contraction (as, e.g., in Algorithm 12). For EA queries, we additionally count how often nonconstant TTFs are evaluated (including similar operations like, e.g., computing $\max \operatorname{dep} \underline{f}(r_{\text{up}}[u])$ as in Algorithm 11). For profile queries, in contrast, we count the number of segments of the TTFs processed by link and minimum operations.

8.3. Results

Preprocessing. Table II shows the running times of node ordering and construction for all input graphs. We not only measure the sequential running times (1 thread), but also the parallel running times for shared memory (2, 4, and 8 threads). The node ordering takes considerably longer than the construction. For both node ordering and

Table II. Running Times of Node Ordering and Construction for All Our Input Graphs with 1, 2, 4, and 8 Threads. For 2, 4, and 8 Threads, We also Report the Speed-Ups Achieved by Parallel Execution (PS). constr.= construct.

	# threads													
	1		2				4				8			
	order	constr.	order		constr.		order		constr.		order		constr.	
	[h:m:s]	[h:m:s]	[h:m:s]	PS	[h:m:s]	PS	[h:m:s]	PS	[h:m:s]	PS	[h:m:s]	PS	[h:m:s]	PS
Germany														
Mon	0:28:23	0:07:41	0:14:42	1.9	0:03:54	2.0	0:07:53	3.6	0:02:06	3.7	0:04:50	5.9	0:01:16	6.1
mid	0:29:21	0:07:33	0:14:57	2.0	0:03:57	1.9	0:08:06	3.6	0:02:09	3.5	0:05:04	5.8	0:01:14	6.1
Fri	0:24:37	0:06:19	0:12:21	2.0	0:03:16	1.9	0:06:39	3.7	0:01:47	3.5	0:04:03	6.1	0:01:02	6.1
Sat	0:15:09	0:03:45	0:07:23	2.1	0:02:00	1.9	0:04:01	3.8	0:01:08	3.3	0:02:33	5.9	0:00:41	5.5
Sun	0:12:30	0:03:10	0:06:08	2.0	0:01:41	1.9	0:03:14	3.9	0:00:56	3.4	0:02:06	5.9	0:00:33	5.7
Europe														
high	3:52:49	0:51:58	1:56:44	2.0	0:26:33	2.0	1:02:35	3.7	0:14:04	3.7	0:37:42	6.2	0:08:02	6.5
med	1:31:37	0:21:42	0:48:29	1.9	0:11:51	1.8	0:27:42	3.3	0:06:14	3.5	0:17:49	5.1	0:03:40	5.9

Table III. Running Time of Node Ordering for Germany Midweek When the Different Techniques Used to Speed up the Node Ordering Are Incrementally Deactivated. += activated, -= deactivated, heur.= heuristic. We Report the Running Times for Both Sequential (1 Thread) and Parallel (8 Threads) Execution.

		order time [h:m:s]			
		+ sample search	- sample search	- sample search	- sample search
threads	caching of	+ heur. thinning	+ heur. thinning	- heur. thinning	- heur. thinning
#	witnesses	+ interval search	+ interval search	+ interval search	- interval search
Germany midweek					
1	+	0:29:21	0:35:25	4:36:34	$\geq$ 12h
	-	1:37:57	1:52:32	$\geq$12h	$\geq$ 12h
8	+	0:05:04	0:05:45	1:16:19	$\geq$ 12h
	-	0:15:53	0:17:17	4:00:17	$\geq$ 12h

construction, the parallelization scales pretty well. The preprocessing takes longer the more nonconstants TTFs are present in a graph and the larger their average delay is.

The reader may remember that preprocessing mainly consists of many node contractions where we try to omit as many shortcuts as possible (Section 4). There, we use some techniques to speed up the expensive *profileSearch* or to even prevent its application. These techniques are: (i) a preceding sample search, (ii) a preceding *profileIntervalSearch*, (iii) a heuristic to thin out the corridor, and (iv) the caching of simulated contractions; (i)–(iii) are described in Section 4.1, (iv) is described in Section 4.2. We demonstrate the impact of (i)–(iii) on the running time of the node ordering by incrementally deactivating them. We do this both with (iv) activated and deactivated, both for sequential (1 thread) and parallel (8 threads) execution. Table III shows the resulting running times with a timeout of 12h. The impact of (i), that is, sample search, seems to be rather small, though it can be noticed. The impact of (ii)–(iv), that is, heuristic thinning, *profileIntervalSearch*, and caching, is much larger. The results show that all the techniques except sample search are vital for feasible preprocessing.

We also tested how preprocessing works without the hop limit described in Section 4.1. It turned out that the hop limit has little effect on the node ordering for Germany midweek, but for Europe high traffic, the node ordering gets unfeasible without hop limit. So, a hop limit of 16 is always active. In the past, we also used aggressive edge reduction [Batz et al. 2009]: During preprocessing, we periodically check for unnecessary shortcuts $u \rightarrow_f v$. In principle, we do this by a *profileSearch* from u to v. Of

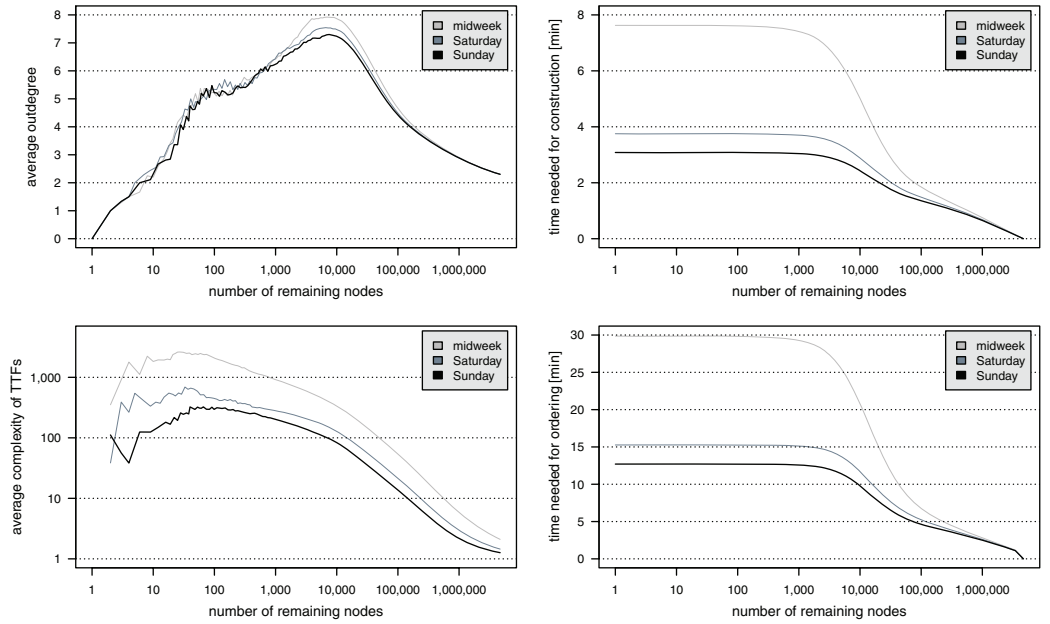

Fig. 1. Behavior during the construction of the TCHs for Germany midweek, Saturday, and Sunday. As the x-axis shows the number of nodes not yet contracted, time "flows" from right to left in the charts. It is shown how the average out degree (top left) and the average complexity of the TTFs (bottom left) of the remaining graph evolve during construction. Also, it is shown how the running time proceeds during construction (top right). We also show how the running time proceeds during node ordering (bottom right).

course, this can be accelerated in a similar way as node contraction. However, in the current setting, the edge reduction did not have much impact, so it was omitted.

Figure 1 gives some insight into the TCH construction for the German road network with different sets of TTFs. It turns out that for graphs with more nonconstant TTFs, the number of inserted shortcuts does not increase too much. Instead, we have a significantly higher complexity of the TTFs of these shortcuts. Besides the higher relative delay, which is likely to raise the number of necessary profile searches, the increased complexity of TTFs is also a probable source of longer running time. Figure 1 also makes clear that the majority of the preprocessing time is needed for the last 100,000 nodes. But from around the last 1,000 nodes, the construction and the node ordering get faster again.

Queries. Table IV shows the running times of exact EA and profile queries for selected query algorithms using TCHs and ATCHs. For ATCHs, we chose $\varepsilon = 2.5\%$ because it seems to be a good compromise between space usage and running time. As expected, the *tchEaQuery* (Algorithm 4) is faster than *atchEaQuery* (Algorithm 11) but the necessary TCH needs much more space than the ATCH of course. The running times for EA queries on Europe, which has much more nodes and edges than Germany, suggest that these algorithms scale well with the size of the road network. The *atchProfileQuery* (Algorithm 12) performs quite well, as it shows running times considerably less than 0.1s on the German road networks. This means profile queries can be answered instantaneously there. For Europe, we cannot provide instantaneous profile queries, but the running times are still not bad. Altogether, it is not surprising that profile queries run faster the less time-dependency a graph has.

Table IV. Behavior of Different Query Algorithms for All Our Input Graphs
Running time, speed-up, and memory usage of *tchEaQuery* (TCH/EA, see Algorithm 4), *tchEaQuery* with BFS as backward search and separate forward search (TCH/backw. BFS), *atchEaQuery* (ATCH/EA, see Algorithm 11), and *atchProfileQuery* (ATCH/profile, see Algorithm 12), SPD = speed-up over *tdDijkstra*, GRO = growth factor of memory usage compared to the original graph.

	TCH							ATCH, $\varepsilon = 2.5\%$					
	space			EA		backw. BFS		space			EA		profile
	total	overhead		time		time		total	overhead		time		time
	[B/n]	[B/n]	GRO	[ms]	SPD	[ms]	SPD	[B/n]	[B/n]	GRO	[ms]	SPD	[ms]
Germany													
Mon	1,001	906	10.5	0.84	1,269	2.58	411	206	111	2.2	1.39	761	34.24
mid	995	899	10.4	0.76	1,401	2.57	414	208	112	2.2	1.37	775	38.57
Fri	833	745	9.5	0.68	1,560	2.25	470	188	100	2.1	1.27	831	30.30
Sat	401	337	6.2	0.52	1,992	1.88	551	127	62	2.0	0.83	1,242	6.72
Sun	265	210	4.8	0.48	2,149	1.71	599	100	45	1.8	0.74	1,388	4.40
Europe													
high	599	523	7.9	1.98	1,916	4.16	913	192	116	2.5	4.04	940	550.21
med	187	140	4.0	1.25	2,947	4.25	865	84	37	1.8	2.61	1,409	290.38

As explained in Section 5, we apply time-dependent bidirectional search to make hierarchical routing run in the presence of time-dependent edge weights. However, would it be not good enough to replace the backward search by a BFS? Of course, we can apply stall-on-demand only to the forward search then. Also, we cannot stop forward and backward search based on an upper bound B as in Line 17 of Algorithm 4, but then it is pointless to perform forward and backward search in an alternating manner. So, we perform forward and backward search separately. The resulting simplified query algorithm takes at least three times longer than *tchEaQuery* for Germany (Table IV). For Europe high traffic, the *tchEaQuery* is only a little more than two times faster. Supposedly, this is because of the larger relative delay, which negatively affects the backward search of *atchEaQuery*. More precisely, the number of overlapping travel time intervals during the backward *profileIntervalSearch* may be so large that comparatively few predecessors are ruled out (see Lines 5 and 6 of Algorithm 3). Also, stall-on-demand may fail comparatively often.

Tables V and VI take a closer look at the behavior of EA and profile queries on TCHs, ATCHs, Min-Max-TCHs, and inexact TCHs. For *atchEaQuery*, the parameter ε provides a trade-off between running time and space usage. For *atchProfileQuery* we observe a minimum running time for ε near 1 %. Our interpretation is that there is a tradeoff between the running time of thinning phase and of the rest of *atchProfileQuery*, that is, of bidirectional *profileIntervalSearch*, corridor unpacking, and corridor contraction. For greater ε, the thinning phase needs less time as the processed approximate TTFs have lower complexity, but the effect of thinning, pruning, and stall-on-demand is smaller then. For smaller ε, the thinning phase needs more time, but thinning, pruning,and stall-on-demand have greater effect. The observed numbers of touched edges and segments confirm this interpretation.

The running times of *atchEaQuery* and *atchProfileQuery* on the Min-Max-TCH of Germany are smaller than for the ATCH with $\varepsilon = 10\%$. At first glance, this seems surprising, but note that Min-Max-TCHs use the upper and lower bounds $\max f$ and $\min f$, respectively, of *exact* TTFs f. ATCHs, in contrast, use the upper and lower bounds $\max \overline{f}$ and $\min \underline{f}$. Also, with the relatively large $\varepsilon = 10\%$ it is likely that the impact of thinning and stall-on-demand becomes poor. So, storing the exact bounds $\max f$ and $\min f$ in the ATCH may lower the running times of *atchEaQuery*, especially for larger values of ε. However, we have not tried this. In contrast to Germany midweek,

Table V. Behavior of EA Queries Using Different Methods
ATCH with $\varepsilon = \infty$ denotes Min-Max-TCHs; struct.= used hierarchical data structure, inex.= inexact, algr.= number of algorithm with respect to this article, SPD= speed-up compared to *tdDijkstra*, GRO= growth factor of space usage compared to the original graph, MAX and AVG are maximum and average relative errors.

	ε		space		time		delMin		edges		evals		error [%]	
struct.	[%]	algr.	[B/n]	GRO	[ms]	SPD	#	SPD	#	SPD	#	SPD	MAX	AVG
Germany midweek														
TCH	–	4	995	10.4	0.75	1,428	520	4,612	5,820	950	1,271	162	0.00	0.00
	–	13	995	10.4	0.74	1,440	639	3,755	7,101	779	76	2,686	0.00	0.00
inex. TCH	0.1	13	286	3.0	0.70	1,516	642	3,737	7,138	775	78	2,651	0.10	0.02
	1.0		214	2.2	0.69	1,553	654	3,669	7,271	761	85	2,432	1.01	0.27
	2.5		172	1.8	0.72	1,470	668	3,594	7,429	745	92	2,234	2.44	0.79
	10.0		113	1.2	1.06	1,006	898	2,674	10,109	547	223	920	9.75	3.84
ATCH	0.1	11	309	3.2	1.15	930	558	4,299	7,281	760	3,182	65	0.00	0.00
	1.0		239	2.5	1.24	857	588	4,080	7,993	692	3,553	58	0.00	0.00
	2.5		208	2.2	1.38	769	625	3,841	9,168	603	4,172	49	0.00	0.00
	10.0		163	1.7	2.56	416	841	2,854	18,947	292	8,871	23	0.00	0.00
	∞		118	1.2	1.55	685	638	3,761	16,212	341	4,043	51	0.00	0.00
Europe high traffic														
TCH	–	4	599	7.9	2.11	1,798	1,021	8,847	13,681	1,563	2,482	276	0.00	0.00
	–	13	599	7.9	3.25	1,168	1,715	5,266	24,274	881	1,485	460	0.00	0.00
inex. TCH	0.1	13	239	3.1	2.70	1,408	1,722	5,245	24,389	877	1,498	456	0.15	0.02
	1.0		195	2.6	2.76	1,376	1,782	5,069	25,361	843	1,624	421	1.50	0.20
	2.5		175	2.3	2.94	1,292	1,875	4,817	26,948	794	1,836	372	3.37	0.48
	10.0		144	1.9	2.92	1,302	1,801	5,016	25,692	832	1,681	407	16.21	2.88
ATCH	0.1	11	258	3.4	2.55	1,489	1,142	7,907	16,693	1,281	6,420	107	0.00	0.00
	1.0		208	2.7	2.89	1,312	1,223	7,384	20,336	1,052	7,819	87	0.00	0.00
	2.5		192	2.5	4.17	910	1,351	6,684	27,583	775	10,500	65	0.00	0.00
	10.0		165	2.2	8.02	473	1,850	4,882	74,315	288	26,911	25	0.00	0.00
	∞		100	1.3	17.04	223	1,690	5,344	207,099	103	51,023	13	0.00	0.00

Min-Max-TCHs do not work quite as well for Europe high traffic: The EA query times are significantly worse (though still not bad). But for profile queries, this variant was so slow that we got a timeout and thus had to omit that line.

For inexact profile queries (see *tchConeProfileQuery*, Algorithm 7) on Germany midweek, we observe the smallest running time for ε around 2.5%. A trade-off between the quality of the conservative bounds $\min\{\min f, \min f_{\approx}\}$, $\max\{\max f, \max f_{\approx}\}$ and the complexity of the inexact TTFs is a possible explanation: For smaller ε, the conservative bounds are nearer to the exact bounds $\min f$ and $\max f$ than for larger ε. As a consequence, stall-on-demand and pruning have more effect during the bidirectional *profileIntervalSearch*. But for smaller ε, the complexity of the inexact TTFs increases. The number of edges and segments touched during the computation support this interpretation.

For inexact profile queries on Europe high traffic, we see a clear correspondence between space and running time. This seems to be natural, as the relative delay is much larger there than for Germany midweek, and this makes the impact of pruning based on conservative bounds less effective. As a result, the running time is governed rather by the complexity of the inexact TTFs than by the quality of the conservative bounds. This interpretation is again supported by the number of processed edges and segments.

Obviously, decreased memory usage directly corresponds to decreased accuracy for inexact profile queries. So, as space and running time are related, we get a trade-off

Table VI. Behavior of Profile Queries Using Different Methods
Note: The nomenclature is as in Table V.

struct.	ε [%]	algr.	space [B/n]	space GRO	time [ms]	delMin #	edges #	segments #	error [%] MAX	error [%] AVG
Germany midweek										
TCH	–	5	995	10.4	1,216.29	570	6,805	16,717,022	0.00	0.00
	–	7	995	10.4	95.46	647	7,179	840,551	0.00	0.00
inex. TCH	0.1	7	286	3.0	6.55	650	7,218	50,812	0.10	0.02
	1.0		214	2.2	3.13	662	7,358	20,928	1.03	0.27
	2.5		172	1.8	2.11	677	7,524	12,580	2.44	0.79
	10.0		113	1.2	2.55	924	10,374	12,759	9.69	3.84
ATCH	0.1	12	309	3.2	38.69	651	27,563	466,251	0.00	0.00
	1.0		239	2.5	35.23	675	29,950	449,012	0.00	0.00
	2.5		208	2.2	38.74	701	34,008	498,837	0.00	0.00
	10.0		163	1.7	118.78	889	86,127	1,434,780	0.00	0.00
	∞		118	1.2	81.81	579	55,214	1,050,447	0.00	0.00
Europe high traffic										
TCH	–	5	599	7.9	4,907.39	1,132	18,176	54,617,858	0.00	0.00
	–	7	599	7.9	2,339.37	1,866	26,879	17,704,460	0.00	0.00
inex. TCH	0.1	7	239	3.1	228.48	1,882	27,060	1,918,590	0.15	0.02
	1.0		195	2.6	121.69	1,953	28,267	978,794	1.37	0.20
	2.5		175	2.3	102.51	2,067	30,276	834,996	3.28	0.48
	10.0		144	1.9	41.77	1,970	28,627	426,327	14.69	2.88
ATCH	0.1	12	258	3.4	646.29	1,875	160,496	5,195,293	0.00	0.00
	1.0		208	2.7	431.89	1,960	189,115	3,481,885	0.00	0.00
	2.5		192	2.5	513.66	2,107	256,981	4,178,386	0.00	0.00
	10.0		165	2.2	2,773.90	2,536	1,340,823	24,100,717	0.00	0.00

between accuracy and running time for these types of queries. For inexact EA queries (Algorithm 13), the running time changes only slightly with the space usage. This means that we mainly have a trade-off between space and accuracy in this case. Note that the observed maximum errors are small for small ε, and the average errors are even smaller. In theory, however, one can easily construct inputs where errors could get much larger than ε.

In Section 6.3, we claim that *atchEaQuery* runs faster if we unpack the shortcuts only on demand instead of unpacking the whole corridor before we perform *tdDijkstra*. Moreover, in Section 6.4, we claim that *corridorContraction* greatly accelerates profile queries on ATCHs. Figure 2 shows that both are really the case. It displays the distribution of running times of EA and profile queries on Germany midweek using a methodology by Sanders and Schultes [2005]: For $i = 5..22$, we look at a bulk of 100 queries with the property that the one-to-one version of Dijkstra's algorithm settles the destination node as the 2^i-th node (2^i is called the *Dijkstra rank*). In the case of EA queries, this refers to *tdDijkstra*, and in the case of profile queries, to a time-independent Dijkstra. Note that we again execute such a bulk of queries three times to prevent accidental outliers. Then, for every query in the bulk, we report the median of the three available measured running times. For profile queries on ATCHs, we display the results of *atchProfileQuery* (with *corridorContraction*) as well as the results of a modified version of *atchProfileQuery* where *corridorContraction* is replaced by a *profileSearch* on the unpacked corridor (see Section 6.4). This demonstrates that we are really faster with *corridorContraction*. We also display the distribution of running times for a plain *profileSearch*. As this runs very slow, we stop after the average running time exceeds 10 s. Running a plain *profileSearch* with 1,000 randomly selected

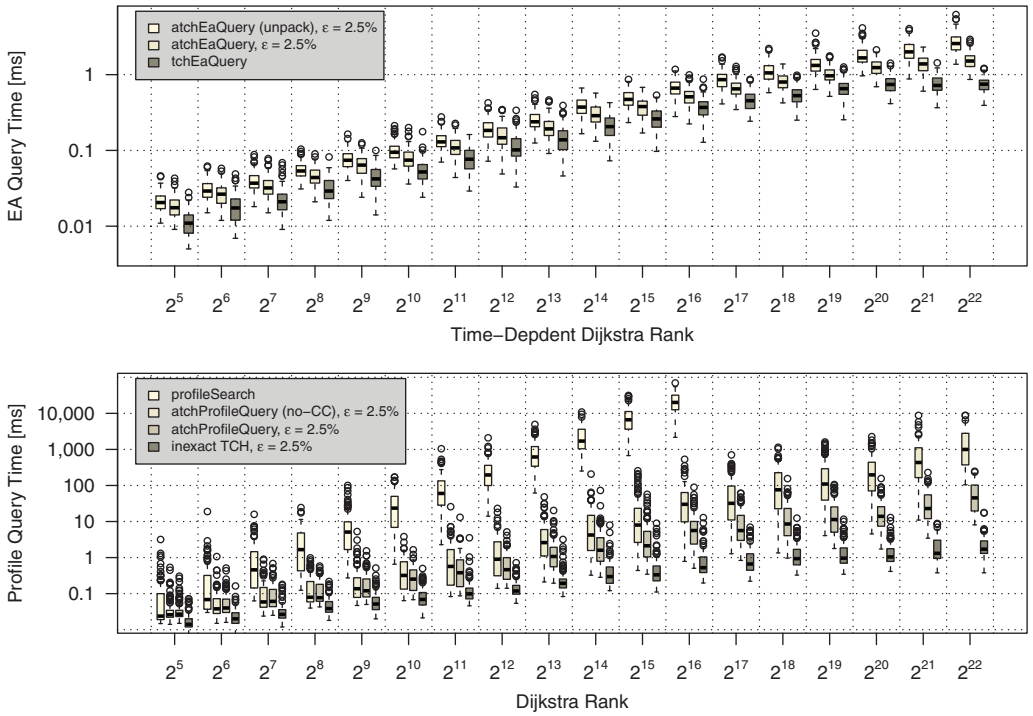

Fig. 2. Running times of EA (top) and profile queries (bottom) over Dijkstra rank for Germany midweek. Top: unpack = shortcuts not only unpacked on demand, but corridor completely unpacked first. Bottom: no-CC = *profileSearch* in the corridor instead of *corridorContraction*, inexact TCH = *tchConeProfileQuery* on an inexact TCH.

pairs of start and destination node would be a matter of weeks. This is why we do not report speed-ups of profile queries. Figure 2 also shows the distribution for *tchEaQuery* and inexact profile queries using *tchConeProfileQuery*.

Reusing Node Orders. In Section 4.2, we claim that a node order, once computed, can be reused to govern the construction of a TCH for the same graph but with a different set of TTFs. Table VII shows the resulting behavior of TCH construction and several query algorithms. It turns out, that this "recycling" works surprisingly well for Germany. For Europe, we have a clear increase of construction time, space usage, and query times, but it still works. As an extreme case, we perform node ordering only with constant edge weights. More precisely, we replace all TTFs of Germany midweek and of Europe high traffic by their minimum and perform node ordering for the resulting graphs. The ordering took 6min 59sec for Germany and 27min 37sec for Europe. As a result, we observe further increased memory usage and query times. Again, this effect seems to be stronger for Europe. In the past [Batz et al. 2009], we even used static CHs [Geisberger et al. 2008] for node ordering. For a higher percentage of nonconstant or stronger varying TTFs, recycling of node orders may not work well enough.

So, depending on the underlying road network and the available sets of TTFs, we could save a considerable amount of time by doing node ordering only for the "easy" instances. For the hard instances, we would recycle one of the easily obtained orders to govern the TCH construction. Whether all this works well in a specific application context must be found out experimentally by the user.

Table VII. Behavior of TCHs, ATCHs, and Inexact TCHs When Node Orders Are Reused to Govern the TCH Construction for a Different Set of TTFs

We report running times and memory usage of *tchEaQuery* (TCH/EA, Algorithm 4), *atchEaQuery* (ATCH/EA, Algorithm 11), *atchProfileQuery* (ATCH/profile, Algorithm 12), and inexact profile queries using *tchConeProfileQuery* (inexact TCH/profile, Algorithm 7). We also report the time needed to construct the TCHs for the different orders. constr. = construct, overh. = overhead, err. = relative error, GRO = growth factor of memory usage compared to the original graph, MAX = maximum relative error, const = node order arising from constant edge weights.

			TCH			ATCH, $\varepsilon = 2.5\%$				inexact TCH, $\varepsilon = 2.5\%$			
		constr.	space		EA	space		EA	profile	space		profile	
		time	total	overh.	time	total	overh.	time	time	total	overh.	time	err. [%]
graph	order	[h:m:s]	[B/n]	GRO	[ms]	[B/n]	GRO	[ms]	[ms]	[B/n]	[B/n]	[ms]	MAX
Germany													
mid	Mon	0:07:44	1,004	10.5	0.85	208	2.2	1.47	40.23	173	1.8	2.12	2.43
	mid	0:07:33	995	10.4	0.76	208	2.2	1.37	38.57	172	1.8	2.11	2.44
	Fri	0:07:51	1,002	10.5	0.82	209	2.2	1.47	38.57	173	1.8	2.12	2.45
	Sat	0:08:39	1,041	10.9	0.88	211	2.2	1.52	40.70	175	1.8	2.28	2.45
	Sun	0:09:18	1,066	11.2	0.91	213	2.2	1.56	40.88	176	1.8	2.31	2.41
	const	0:10:09	1,147	12.0	1.05	219	2.3	1.60	38.24	180	1.9	2.30	2.45
Sat	Mon	0:03:53	422	6.5	0.60	129	2.0	0.90	5.87				
	mid	0:03:49	422	6.5	0.58	129	2.0	0.90	6.07				
	Fri	0:03:50	418	6.5	0.61	128	2.0	0.88	6.90				
	Sat	0:03:45	401	6.2	0.52	127	2.0	0.83	6.72				
	Sun	0:03:60	416	6.5	0.61	128	2.0	0.93	6.17				
	const	0:04:22	458	7.1	0.58	133	2.1	0.87	5.82				
Sun	Mon	0:03:10	282	5.1	0.52	102	1.9	0.77	4.19				
	mid	0:03:08	283	5.1	0.54	102	1.9	0.81	4.30				
	Fri	0:03:08	279	5.1	0.57	102	1.9	0.80	4.38				
	Sat	0:03:10	273	4.9	0.53	101	1.8	0.81	4.21				
	Sun	0:03:10	265	4.8	0.48	100	1.8	0.74	4.40				
	const	0:03:18	299	5.4	0.64	105	1.9	0.78	4.10				
Europe													
high	high	0:51:58	599	7.9	1.98	192	2.5	4.04	550.21	175	2.3	102.51	3.28
	med	1:28:22	723	9.5	2.66	208	2.7	4.53	680.03	189	2.5	143.01	3.19
	const	2:06:06	842	11.1	3.33	214	2.8	5.80	781.82	194	2.5	206.84	2.82
med	high	0:32:35	222	4.7	1.92	88	1.9	3.62	324.15				
	med	0:21:42	187	4.0	1.25	84	1.8	2.61	290.38				
	const	0:55:10	267	5.7	1.96	91	1.9	3.78	301.43				

Comparison with Goal-Directed Techniques. To compare the TCH-based methods with some goal-directed route planning algorithms, look at Table VIII. For EA queries, we only compare speed-ups of *tdDijkstra*—absolute query times would be unreliable because different machines are used. As plain *profileSearch* takes too long, we are not able to report speed-ups for profile queries. Instead, we also compare the running times of profile queries with *tdDijkstra*. The resulting "speed-ups" enable us to compare the running times of different profile query algorithms in a machine-independent way. Note that larger values of these "speed-ups" mean smaller running times. As our preprocessing works in two phases (node ordering and construction), we always report two preprocessing times for TCH-based techniques, for example, 0:29 / 0:08 (29min node ordering and 8min construction). Remember that the node ordering already yields a complete TCH structure, so a separate construction phase is not necessary after the node ordering.

Table VIII. Comparison of Different TCH-Based and Goal-Directed Methods for Exact and Inexact Time-Dependent EA and Profile Queries.
Memory usage is given as overhead (ovh.), errors are maximal relative errors. prepro. = preprocessing, SPD = speed-up achieved by EA queries, "SPD" = "speed-up" achieved by profile queries, i.e., speed of profile queries compared to *tdDijkstra*, err. = relative error, inex = inexact, L = combination with ALT, apprx = approximate, hr = heuristic, se = space efficient.

		Germany midweek						Europe high traffic					
	ε	prepro.	ovh.	EA	profile	err. [%]		prepro.	ovh.	EA	profile	err. [%]	
method	[%]	[h:m]	[B/n]	SPD	"SPD"	MAX	AVG	[h:m]	[B/n]	SPD	"SPD"	MAX	AVG
exact queries													
TCH	–	0:29 / 0:08	899	1,428	11.16	0.00	0.00	3:53 / 0:52	523	1,798	1.62	0.00	0.00
ATCH	0.1	0:29 / 0:08	213	930	27.53	0.00	0.00	3:53 / 0:52	182	1,489	5.88	0.00	0.00
ATCH	1.0	0:29 / 0:08	144	857	30.23	0.00	0.00	3:53 / 0:52	132	1,312	8.79	0.00	0.00
ATCH	2.5	0:29 / 0:08	112	769	27.49	0.00	0.00	3:53 / 0:52	116	910	7.39	0.00	0.00
ATCH	10.0	0:29 / 0:08	68	416	8.97	0.00	0.00	3:53 / 0:52	89	473	1.37	0.00	0.00
ATCH	∞	0:29 / 0:08	23	685	13.02	0.00	0.00	3:53 / 0:52	24	223	–	0.00	0.00
TD-CALT	–	0:09	50	280	–	0.00	0.00	1:00	61	47	–	0.00	0.00
TD-SHARC	–	1:16	155	60	0.02	0.00	0.00	6:44	134	70	–	0.00	0.00
TD-L-SHARC	–	1:18	219	238	–	0.00	0.00	6:49	198	150	–	0.00	0.00
inexact queries													
inex TCH	0.1	0:29 / 0:08	191	1,516	162.50	0.10	0.02	3:53 / 0:52	163	1,408	16.62	0.15	0.02
inex TCH	1.0	0:29 / 0:08	119	1,553	340.74	1.03	0.27	3:53 / 0:52	119	1,376	31.20	1.50	0.20
inex TCH	2.5	0:29 / 0:08	77	1,470	505.03	2.44	0.79	3:53 / 0:52	98	1,292	37.04	3.37	0.48
inex TCH	10.0	0:29 / 0:08	18	1,006	416.90	9.75	3.84	3:53 / 0:52	68	1,302	90.92	16.21	2.88
apprx TD-CALT	–	0:09	50	804	–	13.84	0.05	1:00	61	624	–	8.69	0.28
hr TD-SHARC	–	3:26	137	2,164	1.40	0.61		22:12	127	1,958	–	1.60	
hr TD-L-SHARC	–	3:28	201	3,915	–	0.61		22:17	191	2,703	–	1.60	
se TD-SHARC	–	3:48	68	1,177	–	0.61		–	–	–	–	–	
se TD-SHARC	–	3:48	14	491	–	0.61		–	–	–	–	–	

For exact queries, ATCHs dominate TD-SHARC [Delling 2011] in all respects. TD-CALT [Delling and Nannicini 2008; Delling 2009] is also dominated except for the preprocessing time where TD-CALT is much better. 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 graph size. For inexact EA queries, approximate TD-CALT has much better speed-ups than in the exact case. Compared to that, inexact TCHs have even better speed-up but worse preprocessing time and memory usage. However, the maximum error of approximate TD-CALT is very large. Regarding that, inexact TCHs are much better, at least if ε is not too large. But, to be fair, it must be noted that the average error of approximate TD-CALT is really small. Heuristic TD-SHARC [Delling 2011] has better speed-ups for EA queries but worse memory usage than inexact TCHs. For heuristic space-efficient TD-SHARC [Brunel et al. 2010] the memory usage is very good, but the speed-ups are worse than for inexact TCHs. For all inexact variants of TD-SHARC, the maximum error is smaller than for inexact TCHs—except for small values of ε where inexact TCHs have smaller maximum error but higher memory usage. With respect to the preprocessing time, inexact TCHs are much better than the inexact variants of TD-SHARC.

Regarding profile search, TCH-based techniques are far better than the goal-directed techniques, as they are three orders of magnitude faster for exact queries, and about 300 times faster for inexact ones—if the goal-directed techniques are able to answer profile queries in reasonable time at all.

9. CONCLUSIONS AND FUTURE WORK

TCHs are able to answer earliest arrival and travel time profile queries time- and space-efficiently. Corridors and cones accelerate profile queries and enable exact computations, although we use space-efficient approximate travel time functions on shortcuts. Additionally, corridor contraction speeds up profile queries even more. For the midweek scenario on the German road network parallel preprocessing using 8 threads takes less than 6.5 minutes. With TCH-based representations of the road network, which need less than 1GiB space, we are able to answer exact EA queries in less than 1.5ms and exact profile queries in less than 40ms. If an error of about 1% is allowed, we can answer profile queries in even less than 3.2ms. We are not aware of any other efficient technique for answering profile queries on road networks.

Future work will have to allow even more realistic modeling, in particular, incorporating traffic jams in real time, and allowing more general time-dependent cost functions beyond travel times. Note that more general cost functions make time-dependent route planning much more difficult—it gets NP-hard even when restricted to the relatively simple special case that costs are travel times with additional time-invariant constants. A heuristic version of TCHs for this setup has already been presented [Batz and Sanders 2012], but exact TCHs are still missing there. Time-dependent profile queries with generalized costs are also not considered yet. Another interesting problem is mobile time-dependent route planning. The ideas of this work may help to develop a suitable algorithm for both mobile time-dependent route planning and generalized time-dependent profile queries.

ACKNOWLEDGMENTS

We thank Sabine Neubauer for her implementation [Neubauer 2009] of the Imai-Iri algorithm for approximation of polygons [Imai and Iri 1987] that we use in this work to generate ATCHs and inexact TCHs from exact TCHs. Also, we thank Daniel Delling and the people at our research group for fruitful discussions. We also thank Time Bingmann for proofreading and the anonymous reviewers for their very useful suggestions.

REFERENCES

Batz, G. V., Delling, D., Sanders, P., and Vetter, C. 2009. Time-dependent contraction hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 97–105.

Batz, G. V., Geisberger, R., Neubauer, S., and Sanders, P. 2010. Time-dependent contraction hierarchies and approximation. See Festa [2010], 166–177. http://www.springerlink.com/content/u787292691813526/.

Batz, G. V., Geisberger, R., and Sanders, P. 2008. Time dependent contraction hierarchies—basic algorithmic ideas. Tech. rep. ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH).

Batz, G. V. and Sanders, P. 2012. Time-dependent route planning with generalized objective functions. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Springer.

Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., and Wagner, D. 2010. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM J. Exp. Algor. 15,* 2.3 (January 2010), 1–31.

Brunel, E., Delling, D., Gemsa, A., and Wagner, D. 2010. Space-efficient SHARC-routing. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Springer, 47–58.

Dantzig, G. B. 1962. *Linear Programming and Extensions*. Princeton University Press.

Dean, B. C. 2004. Shortest paths in FIFO time-dependent networks: theory and algorithms. Tech. rep., Massachusetts Institute of Technology.

Delling, D. 2009. Engineering and augmenting route planning algorithms. Ph.d. Dissertation, Universität Karlsruhe, Fakultät für Informatik. http://i11www.ira.uka.de/extra/publications/d-earpa-09. pdf.

Delling, D. 2011. Time-dependent SHARC-routing. *Algorithmica 60,* 1, 60–94.

Delling, D. and Nannicini, G. 2008. Bidirectional core-based routing in dynamic time-dependent road networks. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, Springer, 813–824.

DELLING, D., SANDERS, P., SCHULTES, D., AND WAGNER, D. 2009. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, Eds., Springer, 117–139.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numer. Math. 1*, 269–271.

DREYFUS, S. E. 1969. An appraisal of some shortest-path algorithms. *Oper. Res. 17,* 3, 395–412.

FESTA, P. (Ed.). 2010. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Springer.

GEISBERGER, R. AND SANDERS, P. 2010. Engineering time-dependent many-to-many shortest paths computation. In *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*.

GEISBERGER, R., SANDERS, P., SCHULTES, D., AND DELLING, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Springer, 319–333.

GEISBERGER, R., SANDERS, P., SCHULTES, D., AND VETTER, C. 2012. Exact routing in large road networks using contraction hierarchies. *Trans. Sci.* 46, 3, 388–404.

GOLDBERG, A. V. AND HARRELSON, C. 2005. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*. SIAM, 156–165.

HOLZER, M., SCHULZ, F., AND WAGNER, D. 2008. Engineering multilevel overlay graphs for shortest-path queries. *ACM J. Exp. Algor. 13,* 2.5, 1–26.

IMAI, H. AND IRI, M. 1987. An optimal algorithm for approximating a piecewise linear function. *J. Info. Proces. 9,* 3, 159–162.

KIERITZ, T., LUXEN, D., SANDERS, P., AND VETTER, C. 2010. Distributed time-dependent contraction hierarchies. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10).* Springer, 83–93.

MEHLHORN, K. AND SANDERS, P. 2008. *Algorithms and Data Structures: The Basic Toolbox*. Springer.

NANNICINI, G., DELLING, D., LIBERTI, L., AND SCHULTES, D. 2008. Bidirectional A* search for time-dependent fast paths. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Springer, 334–346.

NEUBAUER, S. 2009. Space efficient approximation of piecewise linear functions. Studienarbeit, Universität Karlsruhe, Fakultät für Informatik. http://algo2.iti.kit.edu/download/neubauer_sa.pdf.

ORDA, A. AND ROM, R. 1990. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *J. ACM 37,* 3, 607–625.

SANDERS, P. AND SCHULTES, D. 2005. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*. Springer, 568–579.

SCHULTES, D. AND SANDERS, P. 2007. Dynamic highway-node routing. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*. Springer, 66–79.

Received November 2010; revised September 2012; accepted September 2012