Engineering Time-Dependent Many-to-Many Shortest Paths Computation *

Robert Geisberger and Peter Sanders

Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{geisberger,sanders}@kit.edu

Abstract

Computing distance tables is important for many logistics problems like the vehicle routing problem (VRP). While shortest distances from all source nodes in S to all target nodes in T are time-independent, travel times are not. We present the first efficient algorithms to compute time-dependent travel time tables in large time-dependent road networks. Our algorithms are based on time-dependent contraction hierarchies (TCH), currently the fastest time-dependent speed-up technique. The computation of a table is inherently in $\Theta(|S| \cdot |T|)$, and therefore inefficient for large tables. We provide one particular algorithm using only $\Theta(|S| + |T|)$ time and space, being able to answer queries two orders of magnitude faster than the basic TCH implementation. If small errors are acceptable, approximate versions of our algorithms are further orders of magnitude faster.

1998 ACM Subject Classification G.2.2, J.1

Keywords and phrases time-dependent, travel time table, algorithm engineering, vrp

Digital Object Identifier 10.4230/OASIcs.ATMOS.2010.74

1 Introduction

Computing travel times between all locations in a predefined set is a known problem arising in many operations research problems, e.g. vehicle routing. More formally, given a graph $G = (V, E)$, and a set of source nodes $S \subseteq V$ and target nodes $T \subseteq V$, we want to know the travel time between each source and target node (*many-to-many shortest paths problem*). The common approach is to compute a *travel time table* of size $|S| \cdot |T|$, reducing subsequent travel time computations to a simple table look-up. Due to the knowledge of historical traffic data and traffic prediction models, it is possible to forecast travel times in dependence to the departure time. These *time-dependent* travel times allow to compute more realistic routes, especially important for routing within cities with time windows. In this *time-dependent scenario*, each cell in the travel time table corresponds to a travel time function over the departure time.

In contrast to the *static scenario* without time-dependency, such a time-dependent table takes a lot longer to compute and occupies a lot more space. We refine the problem of computing a table to the problem of implementing a *query interface*: Given $s \in S$ and $t \in T$, we want to know the earliest arrival time when we depart at time τ (or the travel time profile for all τ). So any algorithm that previously used a table now just needs to replace its table lookups with calls to this interface. An algorithm behind this interface uses a precomputed data structure with the knowledge of G, S and T to answer these queries fast. Especially in the common case where $|S|, |T| \ll |V|$, such an algorithm is able to answer a query

* Partially supported by DFG grant SA 933/5-1.

10th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'10).
Editors: Thomas Erlebach, Marco Lübbecke; pp. 74–87

OpenAccess Series in Informatics
Schloss Dagstuhl Publishing, Germany

several orders of magnitude faster than a common fast time-dependent query algorithm. We contribute five such algorithms that allow different tradeoffs between precomputation time, space and query time. Furthermore, we provide heuristic versions of these algorithms that are substantially faster and more space-efficient. For these, we are able to guarantee quite tight error bounds for queries.

1.1 Related Work

To the best of our knowledge, there is currently no work on the time-dependent many-to-many shortest paths problem. The related work can be divided into work on the static (time-independent) many-to-many shortest paths problem and work on the time-dependent point-to-point shortest path problem. To compute an $|S| \times |T|$ table on a static road network, the most simple method is to perform $|S|$ single source shortest path computations using Dijkstra's algorithm. But this is more than three orders of magnitude slower than [12]. This algorithm [12] can be adapted to any speed-up technique for shortest paths that is *bidirected*, i.e. using a small forward search from the source node and a small backward search from the target node to find the shortest path, and *non-goaldirected*, i.e. the small forward search does not depend on the target node and vice versa. [12] gains its speedup by computing each forward/backward search space only once and combining them. This combination amounts to cheap operations (add, min on integers) in the static scenario. However, in the time-dependent scenario, they map to expensive operations on travel time functions. Our contributions are more sophisticated algorithms to perform the combination, being several times faster than [12].

We are not able to use the previous simple approach of $|S|$ single source computations in the time-dependent scenario, as even one such computation requires too much main memory. However, many new algorithms for the time-dependent point-to-point shortest path problem have been developed recently. We refer to [5] for an overview. Similar to [12], our algorithms require a time-dependent speed-up technique that is bidirected and non-goaldirected. These requirements stem from from the similar problem of many-to-many shortest paths but our algorithms are significantly more advanced than [12]. We use time-dependent contraction hierarchies (TCH) [3, 4] for our algorithms since it is currently the only one that provides small enough forward and backward search spaces.

The time-dependent vehicle routing problem (TDVRP) is a known problem in operations research and there exist many algorithms to solve it [13, 11, 9, 6, 8]. The goal is to find routes for a fleet of vehicles such that all customers (locations) are satisfied and the total (time-dependent) cost is minimized. Solving this problem is important for logistics, supply chain management and similar industries. Our algorithms provide currently the most efficient way to compute the time-dependent travel times between the distinct locations of the TDVRP. Industrial applications often compute travel times for a discrete set of departure times, e.g. every hour. This approach is very problematic as it is expensive to compute, requires a lot of space (a table for every hour), and provides absolutely no approximation guarantee. Our heuristic variants do not have these disadvantages. We require less precomputation time and space, a very important aspect for companies as they can run the algorithm on smaller and cheaper machines. And, even more important, we provide approximation guarantees potentially resulting in better routes in practice that further reduce the operational costs of their transportation business.

2 Preliminaries

2.1 Time-Dependent Road Networks

Let $G = (V, E)$ be a directed graph representing a road network.[1] Each edge $(u, v) \in E$ has a function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ assigned as edge weight. This function f specifies the time $f(\tau)$ needed to reach v from u via edge (u, v) when starting at *departure time* τ. So the edge weights are called *travel time functions* (TTFs).

In road networks we usually do not arrive earlier when we start later. So all TTFs f fulfill the *FIFO-property*: $\forall \tau' > \tau : \tau' + f(\tau') \geq \tau + f(\tau)$. In this work all TTFs are sequences of *points* representing piecewise linear functions.[2] With $|f|$ we denote the *complexity* (i.e., the number of points) of f. We define $f \sim g :\Leftrightarrow \forall \tau : f(\tau) \sim g(\tau)$ for $\sim \in \{<, >, \leq, \geq\}$.

For TTFs we need the following three operations:

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

In a time-dependent road network, shortest paths depend on the departure time. For given start node s and destination node t there might be different shortest paths for different departure times. The minimal travel times from s to t for all departure times τ are called the *travel time profile* from s to t and are represented by a TTF.

2.2 Algorithmic Ingredients

Profile Search. To compute the travel time profile from a source node s to all other nodes, we use a label correcting modification of Dijkstra's algorithm [14]. The modifications are as follows:

- *Node labels.* Each node v has a tentative TTF from s to v.
- *Priority queue (PQ).* The keys used are the global minima of the labels. Reinserts into the PQ are possible and happen (*label correcting*).
- *Edge Relaxation.* Consider the relaxation of an edge (u, v) with TTF f_{uv}. Let the label of node u be the TTF f_u. The label f_v of the node v is updated by computing the minimum TTF of f_v and $f_{uv} * f_u$.

Min-Max Search. Profile search is a very expensive algorithm. *Min-max search* [3] is a roughly approximating modification of profile search and runs much faster. Essentially it is two searches based on Dijkstra's algorithm, one based on the global minima and one on the

[1] Nodes represent junctions and edges represent road segments.
[2] Here, all TTFs have period $\Pi = 24$h. However, using non-periodic TTFs makes no real difference. Of course, covering more than 24h will increase the memory usage.

maxima of the edge TTFs. The results are a lower and an upper bound on the travel time profile.

Approximations. Give a TTF f. A *lower bound* is a TTF $f^{\downarrow}$ with $f^{\downarrow} \leq f$ and a *lower ε-bound* if further $(1-\varepsilon)f \leq f^{\downarrow}$. An *upper bound* is a TTF $f^{\uparrow}$ with $f \leq f^{\uparrow}$ and an *upper ε-bound* if further $f^{\uparrow} \leq (1+\varepsilon)f$. An *$\varepsilon$-approximation* is a TTF $f^{\updownarrow}$ with $(1-\varepsilon)f \leq f^{\updownarrow} \leq (1+\varepsilon)f$. Approximate TTFs usually have fewer points and are therefore faster to process and require less memory. To compute ε-bounds and ε-approximations from an exact TTF f we use the efficient geometric algorithm described by Imai and Iri [10]. It yields a TTF with minimal number of points for ε in time $O(|f|)$.

2.3 Time-Dependent Contraction Hierarchies

Hierarchies. In a *time-dependent contraction hierarchy* [3] all nodes of G are *ordered* by increasing 'importance' [7]. In order to simplify matters, we identify each node with its importance level, i.e. $V = 1..n$.

Now, the TCH is constructed by *contracting* the nodes in the above order. Contracting a node v means removing v from the graph without changing shortest path distances between the remaining (more important) nodes. This way we construct the next higher level of the hierarchy from the current one. A trivial way to contract a node v is to introduce a shortcut edge (u, w) with TTF $g * f$ for every path $u \to_f v \to_g w$ with $v < u, w$. But in order to keep the graph sparse, we can try to avoid a shortcut (u, w) by finding a *witness* – a travel time profile W from u to v fulfilling $W \leq g * f$. Such a witness proves that the shortcut is never needed. The node ordering and the construction of the TCH are performed offline in a precomputation and are only required once per graph independent of S and T.

Queries. In the point-to-point scenario, we compute the travel time profile between source s and target t by performing a two-phase bidirectional time-dependent profile search in the TCH. The special restriction on a TCH search is that it only goes *upward*, i.e. we only relax edges where the other node is more important. We first perform a bidirectional min-max search. Both search scopes meet at *candidate* nodes u giving lower/upper bounds on the travel time between source and target, allowing us to prune the following profile search. The bidirectional profile search computes forward TTF f_u and backward TTF g_u representing a TTF $g_u * f_u$ from source to target (though not necessarily an optimal one). The travel time profile is $\min\{g_u * f_u \mid u \text{ candidate}\}$.

During the min-max search search we perform *stall-on-demand* (see [7, 3]): A node u is *stalled* when we find that a maximum to u coming from a higher level is better than the minimum. The edges of stalled nodes are not relaxed.

3 Five Algorithms

We engineer five algorithms with different precomputation time, space and query time. They support two types of queries: time and profile queries. A *time query* computes the earliest arrival time at a node t when departing from node s at time τ resulting in a query interface (s, t, τ), and a *profile query* computes the travel time profile between s and t resulting in a query interface (s, t).

Our algorithms have in common that they need to precompute $\forall s \in S$ the target-independent forward search spaces

$$F_s := \{(u, f_u) \mid f_u \text{ is TTF from } s \text{ to } u \text{ in forward upward search}\}$$

and $\forall t \in T$ the symmetric source-independent backward search spaces

$$B_t := \{(u, g_u) \mid g_u \text{ is TTF from } u \text{ to } t \text{ in backward upward search}\} \quad .$$

We compute these search spaces using unidirectional upward profile searches. To reduce the computational effort, we initially perform a min-max search using the *stall-on-demand* technique and use it to prune the following profile search. This technique *stalls* nodes that are reached suboptimally. Note that a node can be reached suboptimally, as our upward search does not relax *downward* edges of a settled node. These stalled nodes will never be a candidate to a shortest path, as they are reached suboptimally. Therefore, we do not store them in the search spaces F_s, B_t. Naturally, we cannot use further pruning techniques from [4] that are applied after an upper bound on the shortest paths distance is obtained.

Algorithm Intersect computes and stores F_s and B_t. The other four algorithms are based on it, we ordered them by decreasing query time. Algorithm 1 shows the implementation of the time query. The main part is to evaluate all paths via the candidate nodes. We order the search space entries by node-ids, so that a single scan of both search spaces finds all candidates. At most 2·#candidates TTF evaluations are required for a query. However, the TTF evaluations are the most expensive part, so we prune them using the (precomputed) minima of f_u, g_u.

Algorithm 1: IntersectTimeQuery(s,t,τ)

```
1  δ := ∞;                                              // tentative arrival time
2  foreach (u, f_u) ∈ F_s, (u, g_u) ∈ B_t do            // loop over all candidate nodes
3      if τ + min f_u + min g_u < δ then                // prune using minima
4          δ' := f_u(τ) + τ;                            // evaluate TTFs
5          δ' := g_u(δ') + δ';
6          δ := min(δ, δ');                             // update tentative arrival time
7  return δ
```

Algorithm 2: IntersectProfileQuery(s,t)

```
1  δ̄ := min_{u candidate} {max f_u + max g_u};          // upper bound based on maxima
2  v := argmin_{u candidate} {min f_u + min g_u};       // minimum candidate
3  δ↑ := g_v↑ * f_v↑;                                   // upper bound based on approximate TTFs
4  δ̄ := min(δ̄, max δ↑);                                 // tighten upper bound
5  foreach (u, ·) ∈ F_s, (u, ·) ∈ B_t do                // loop over all candidate nodes
6      if min f_u + min g_u ≤ δ̄ then                    // prune using minima
7          δ↑ := min(δ↑, g_u↑ * f_u↑);                  // update upper bound
8  δ := g_v * f_v;                                      // tentative travel time profile
9  foreach (u, f_u) ∈ F_s, (u, g_u) ∈ B_t do            // loop over all candidate nodes
10     if ¬(g_u↓ * f_u↓ > δ↑) then                      // prune using lower bounds
11         δ := min(δ, g_u * f_u);                      // update travel time profile
12 return δ
```

The profile query is similar to the time query, but links the two TTFs at the candidate instead of evaluating them. But as the link operation is even more expensive than the evaluation operation, we implement more sophisticated pruning steps, see Algorithm 2. For each $(u, f_u) \in F_s$ we compute and store lower/upper ε-bounds $f_u^{\downarrow}$ / $f_u^{\uparrow}$ and for each $(u, g_u) \in B_t$ we compute and store lower/upper ε-bounds $g_u^{\downarrow}$ / $g_u^{\uparrow}$. Then we pass three times

through the search spaces F_s, B_t:

1. In Line 1 we compute an upper bound $\overline{\delta}$ based on the maxima of the search space TTFs. Also, in Line 2 we compute a candidate with minimum sum of the minima of the search space TTFs. This candidate is usually very important and a good starting point to obtain an tight lower bound.
2. In Lines 3–7 we compute an upper bound $\delta^{\uparrow}$ based on the upper ε-bounds. This bound is tighter than the one based on the maxima.
3. In Lines 8–11 we compute the travel time profile and use the upper bound $\delta^{\uparrow}$ for pruning. So we only execute the very expensive link and minimum operations on f_u and g_u at Line 11.

The profile query is arguably the more important of both query types, as it allows to precompute all earliest arrival times independent of a specific departure time. Also, on the profile query we see the difference to the previous time-independent algorithm [12] that only required one pass. Here, we intentionally perform three passes as this allows to save some expensive TTF operations. Therefore, we store a TTF not at the candidate node u (as [12] did), but at the source or target node. This is necessary to perform the intersection, which allows us to keep only one set of upper bounds $(\overline{\delta}, \delta^{\uparrow})$ in main memory at the same time, and not one set for each pair in $S \times T$.

An important observation is that the computation time and space of a single search space depend only on the graph and the edge weights, and are *independent* of $|S|$ and $|T|$. INTERSECT requires therefore $\Theta(|S| + |T|)$ preprocessing time and space, and $\Theta(1)$ query time, if only $|S|$ and $|T|$ are considered as changing variables.

Algorithm MinCandidate additionally precomputes and stores the minimum candidate (Algorithm 2, Line 2) in a table $c_{\min}(s,t)$.

$$c_{\min}(s,t) := \operatorname*{argmin}_{u \text{ candidate}} \{\min f_u + \min g_u\}$$

By that, we can use it to obtain a good initial upper bound for a time query, by initializing δ in Line 1 of Algorithm 1 with the travel time via $c_{\min}(s,t)$. This allows to prune more candidates and results in faster query times. However, preprocessing time and space are now in $\Theta(|S| \cdot |T|)$, but with a very small constant factor.

Algorithm RelevantCandidates precomputes a set of candidate nodes $c_{\text{rel}}(s,t)$ for each s-t-pair by using lower and upper bounds on the TTFs in F_s and B_t.

$$c_{\text{rel}}(s,t) := \left\{ u \;\middle|\; \neg \left(g_u^{\downarrow} * f_u^{\downarrow} > \min_{v \text{ candidate}} \{ g_v^{\uparrow} * f_v^{\uparrow} \} \right) \right\}$$

This is exactly the set of nodes that are evaluated in Line 11 of Algorithm 2. So it is sufficient to evaluate the candidates in $c_{\text{rel}}(s,t)$ to answer a query correctly. In practice, $c_{\text{rel}}(s,t)$ is stored as an array with $c_{\min}(s,t)$ on the first position. Additionally, we can save space by not storing (u, f_u) in F_s if $\forall t \in T : u \notin c_{\text{rel}}(s,t)$, and symmetrically for F_t. The precomputation time depends on the used lower and upper bounds: Using only min-max-values is fast but results in larger sets, using ε-bounds is slower but reduces the size of the sets.

Algorithm OptCandidate precomputes for every departure time τ an optimal candidate $c_{\text{opt}}(s,t,\tau)$, so a time query only needs to evaluate one candidate.

$$c_{\text{opt}}(s,t,\tau) := \operatorname*{argmin}_{u \text{ candidate}} \{(g_u * f_u)(\tau)\}$$

A candidate is usually optimal for a whole period of time, we store these periods as consecutive, non-overlapping, intervals $[\tau_1, \tau_2)$. In practice, there are only very few intervals per pair (s, t) so that we can find the optimal candidate very fast. The downside of this algorithm is its very high precomputation time since it requires the computation of the travel time profile for any pair $(s, t) \in S \times T$ with the INTERSECT algorithm. Still, storing only the optimal candidates requires usually less space than the travel time profile.

Algorithm Table computes and stores all travel time profiles in a table.

$$\text{table}(s, t) := \min_{u \text{ candidate}} \{g_u * f_u\}$$

It provides the fastest query times, but the space requirements in $\Theta(|S| \cdot |T|)$ have a large constant factor. The table cells are computed with the INTERSECT algorithm.

4 Approximate Travel Time Functions

Approximate TTFs reduce preprocessing time, space and query time of the algorithms in the previous section by several orders of magnitude by sacrificing exactness. While used before for point-to-point queries [4], we are the first to present approximation guarantees for queries.

There are three places to approximate TTFs: on the edges of the TCH, the node label TTFs after the forward/backward searches and finally the TABLE entries. Approximating the latter two can be applied straightforwardly. But performing a TCH profile query on approximate edge TTFs requires a change of the stall-on-demand technique since we must not stall an optimal path. We ensure this by performing the initial min-max query on exact values with stall-on-demand, the latter profile query without.

The query algorithms stay the same, except that INTERSECT profile queries no longer use ε-bounds for pruning, as the overhead does no longer pay off.

Lemmas 1–3 enable us to compute theoretical error bounds. With Lemma 3 we have an error bound when we approximate the edge TTFs with ε_e:

$$\varepsilon_1 := \varepsilon_e(1 + \alpha)/(1 - \alpha\varepsilon_e)$$

The error bound for approximating the search space TTFs with ε_s follows directly from the definition of an ε_s-approximation:

$$\varepsilon_2 := (1 + \varepsilon_s)(1 + \varepsilon_1) - 1$$

Lemma 1 gives an error bound when we link the forward and backward search TTF on a candidate node:

$$\varepsilon_3 := \varepsilon_2(1 + (1 + \varepsilon_2)\alpha)$$

With Lemma 2 we know that ε_3 is an error bound on the approximate travel time profile, the minimum over all candidate TTFs. When we additionally approximate the resulting profile TTF for the table with ε_t, the error bound follows directly from the definition of an ε_t-approximation:

$$\varepsilon_4 := (1 + \varepsilon_t)(1 + \varepsilon_3) - 1$$

In Table 6 we compute the resulting error bounds for our test instance.

▶ **Lemma 1.** *Let $f^{\updownarrow}$ be an ε_f-approximation of TTF f and $g^{\updownarrow}$ be an ε_g-approximation of TTF g. Let α be the maximum slope of g, i.e. $\forall \tau' > \tau : g(\tau') - g(\tau) \leq \alpha\,|\tau' - \tau|$. Then $g^{\updownarrow} * f^{\updownarrow}$ is a $\max\{\varepsilon_g, \varepsilon_f(1 + (1 + \varepsilon_g)\alpha)\}$-approximation of $g * f$.*

Proof. Let τ be a time.

$$\begin{aligned}(g^{\updownarrow} * f^{\updownarrow})(\tau) &= g^{\updownarrow}(f^{\updownarrow}(\tau)+\tau) + f^{\updownarrow}(\tau)\\ &\leq (1+\varepsilon_g)(g(f^{\updownarrow}(\tau)+\tau)) + (1+\varepsilon_f)f(\tau)\\ &\leq (1+\varepsilon_g)(g(f(\tau)+\tau) + \alpha\left|(f^{\updownarrow}(\tau)+\tau) - (f(\tau)+\tau)\right|) + (1+\varepsilon_f)f(\tau)\\ &\leq (1+\varepsilon_g)(g(f(\tau)+\tau) + \alpha\varepsilon_f f(\tau)) + (1+\varepsilon_f)f(\tau)\\ &= (1+\varepsilon_g)g(f(\tau)+\tau) + (1+\varepsilon_f(1+(1+\varepsilon_g)\alpha))f(\tau)\end{aligned}$$

By applying the symmetric transformations, we also obtain $(g^{\updownarrow} * f^{\updownarrow})(\tau) \geq (1-\varepsilon_g)g(f(\tau)+\tau) + (1-\varepsilon_f(1+(1-\varepsilon_g)\alpha))f(\tau)$. ◀

▶ **Lemma 2.** *Let $f^{\updownarrow}$ be an ε_f-approximation of TTF f and $g^{\updownarrow}$ be an ε_g-approximation of TTF g. Then $\min\left\{f^{\updownarrow}, g^{\updownarrow}\right\}$ is a $\max\{\varepsilon_f, \varepsilon_g\}$-approximation of $\min\{f, g\}$.*

Proof. Let τ be a time, WLOG we assume that $\min\left\{f^{\updownarrow}, g^{\updownarrow}\right\}(\tau) = f^{\updownarrow}(\tau)$ and $\min\{f, g\}(\tau) = g(\tau)$. Then $\min\left\{f^{\updownarrow}, g^{\updownarrow}\right\}(\tau) = f^{\updownarrow}(\tau) \leq g^{\updownarrow}(\tau) \leq (1+\varepsilon_g)g(\tau)$ and $\min\left\{f^{\updownarrow}, g^{\updownarrow}\right\}(\tau) = f^{\updownarrow}(\tau) \geq (1-\varepsilon_f)f(\tau) \geq (1-\varepsilon_f)g(\tau)$. ◀

▶ **Lemma 3.** *Let F_s and B_t be the forward/backward search spaces computed on a TCH and $F_s^{\updownarrow}$ and $B_t^{\updownarrow}$ on the same TCH with ε-approximated edge TTFs. In both cases, stall-on-demand was only used with exact min-max values. Let α be the maximum slope of all TTFs in F_s, B_t, and $\alpha\varepsilon < 1$. Then $\{u \mid (u, f_u) \in F_s\} = \{u \mid (u, f_u^{\updownarrow}) \in F_s^{\updownarrow}\}$ and $f_u^{\updownarrow}$ is an $\varepsilon(1+\alpha)/(1-\alpha\varepsilon)$-approximation of f_u, and the same holds for the backward search spaces.*

Proof. $\{u \mid (u, f_u) \in F_s\} = \{u \mid (u, f_u^{\updownarrow}) \in F_s^{\updownarrow}\}$ holds trivially since exact and approximate search both use the same min-max values, the same for the backward search spaces. For the forward search, we will prove via induction over s (starting with the most important node n) that for each $f_u^{\updownarrow}$ in F_s there exists a $k \in \mathbb{N}$ so that f_u is a $((1+\varepsilon)\left(\sum_{i=0}^{k-1}(\alpha\varepsilon)^i\right) - 1)$-approximation of f_u.

The base case holds trivially since for $s = n$, $F_n = \{(n, 0)\} = F_n^{\updownarrow}$.

Inductive step: Let $(u, f_u) \in F_s$, $(u, f_u^{\updownarrow}) \in F_s^{\updownarrow}$. Let $N = \{v \mid (s, v) \in E, s < v\}$, h_{sv} be the exact TTF on the edge (s, v) and $h_{sv}^{\updownarrow}$ its ε-approximation. By definition of N, $f_u = \min\left\{\tilde{f}_u * h_{sv} \mid v \in N, (u, \tilde{f}_u) \in F_v\right\}$ and $f_u^{\updownarrow} = \min\{\tilde{f}_u^{\updownarrow} * h_{sv}^{\updownarrow} \mid v \in N, (u, \tilde{f}_u^{\updownarrow}) \in F_v^{\updownarrow}\}$. By induction hypothesis there exists $k \in \mathbb{N}$ so that $\tilde{f}_u^{\updownarrow}$ is an $((1+\varepsilon)\left(\sum_{i=0}^{k-1}(\alpha\varepsilon)^i\right) - 1)$-approximation of $\tilde{f}_u$. Also $\tilde{f}_u$ has maximum slope α and the edge TTF $h_{sv}^{\updownarrow}$ is an ε-approximation of h_{sv}. So by Lemma 1, $\tilde{f}_u^{\updownarrow} * h_{sv}^{\updownarrow}$ is a $((1+\varepsilon)\left(\sum_{i=0}^{k}(\alpha\varepsilon)^i\right) - 1)$-approximation of $\tilde{f}_u * h_{sv}$ $(k \rightsquigarrow k+1)$. Lemma 2 finally shows that the induction hypothesis holds for $f_u^{\updownarrow}$. So for any TTF in any $F_s^{\updownarrow}$ there exists this $k \in \mathbb{N}$, and with $\alpha\varepsilon < 1$ we follow $\lim_{k\to\infty}((1+\varepsilon)\left(\sum_{i=0}^{k-1}(\alpha\varepsilon)^i\right) - 1) = \varepsilon(1+\alpha)/(1-\alpha\varepsilon)$. This concludes the proof for the forward case. The backward case is similar to the forward case except that we use $N = \{v \mid (u, v) \in E, v < u\}$, $g_u = \min\{g_v * h_{uv} \mid v \in N, (v, g_v) \in B_t\}$ and $g_u^{\updownarrow} = \min\{g_v^{\updownarrow} * h_{uv}^{\updownarrow} \mid v \in N, (v, g_v^{\updownarrow}) \in B_t^{\updownarrow}\}$. ◀

5 On Demand Precomputation

We discussed five algorithms with different precomputation times in Section 3. Only the first algorithm INTERSECT provides precomputation in $\Theta(|S| + |T|)$. All further algorithms are in $\Theta(|S| \cdot |T|)$ as they precompute some data for each pair in $S \times T$. To provide a linear algorithm that benefits from the ideas of the further algorithms, we can compute the additional data ($c_{\min}(s,t)$, $c_{\text{rel}}(s,t)$, $c_{\text{opt}}(s,t)$ or table(s,t)) on demand only for those pairs (s,t) that occur in queries. By that, our algorithm is in $\Theta(|S| + |T| + \#\text{queries})$.

While it takes negligible time for an on demand profile query to compute the additional data at the first occurrence of (s, t), the situation is different for a time query. Depending on the additional precomputation time, we should only compute it after a certain number of queries for that pair occurred. This way, we improve the *competitive ratio*. This ratio is the largest possible ratio between the runtime of an algorithm that knows all queries in advance and our algorithm that does not (*online algorithm*). For just two different algorithms, e.g. INTERSECT and TABLE, this problem is similar to the ski-rental problem. For example, let it 'cost' t_i to answer a query using INTERSECT and t_t using TABLE, and t_c to compute the table cell. Then computing the table cell on the query number $b = \lfloor t_c/(t_i - t_t) \rfloor$ to this cell has a competitive ratio < 2. In practice, we can predict the cost of t_i and t_t from the number of necessary TTF evaluations, and the cost t_c from the sum of the points of the TTFs f_u and g_u of all (relevant) candidates u. When we want to use more than two of the five algorithms online, Azar et al. [1] propose an algorithm with competitive ratio 6.83.

6 Experiments

Input. We use a real-world time-dependent road network of Germany with 4.7 million nodes and 10.8 million edges, provided by PTV AG for scientific use. It reflects the midweek (Tuesday till Thursday) traffic collected from historical data, i.e., a high traffic scenario with about 8 % time dependent edges.

Hardware/Software. The experiments were done on a machine with two Intel Xeon X5550 processors (Quad-Core) clocked at 2.67 GHz with 48 GiB of RAM and 2x8 MiB of Cache running SUSE Linux 11.1. We used the GCC 4.3.2 compiler with optimization level 3.

Basic setup. We use a preprocessed TCH as input file [3]. However, we do not account for its preprocessing time (37 min [3]), as it is independent of S and T. We choose $S, T \subseteq V$ uniformly at random for a *size* $|S| = |T|$. We approximated the TTFs in the graph with ε_e, the TTFs of the search spaces with ε_s and the TTFs in the table with ε_t. We use lower and upper ε_p-bounds for pruning profile queries, or just min-max-values if no ε_p is specified. The precomputation uses all 8 cores of our machine since it can be trivially parallelized and multi-core CPUs are standard these days. We report the *preprocessing* time to compute the forward and backward search spaces as *search* and the time to compute additional data ($c_{\min}$, c_{rel}, c_{opt} or table) as *link*. We also give the used *RAM* reported by the Linux kernel. The time (profile) query performances are averages over 100 000 (1 000) queries selected uniformly at random and performed on a single core. Depending on the algorithm, we also report some more detailed time query statistics. *Scan* is the number of nodes in the search spaces we scanned during a time query. *Meet* is the number of candidate nodes where forward and backward search space met. *Eval* is the number of TTF evaluations. *Succ* is the number of successful reductions of the tentative earliest arrival time due to another evaluated candidate.

Preprocessing time and search space size of the INTERSECT algorithm are in $\Theta(|S| + |T|)$ as expected, see Table 1. Note that the RAM requirements include the input TCH:

ε_e [%]	-	0.1	1.0	10.0
graph [MiB]	4 497	1 324	1 002	551

The exact time query is two orders of magnitude faster than a standard TCH[3] time query (720 μs [3]). However, the TCH profile query (32.75 ms [3]) is just 22 times slower since most

[3] We compare ourselves to TCH as it is currently the fastest exact speed-up technique.

Table 1 Performance of the INTERSECT algorithm.

size	ε_e [%]	ε_s [%]	ε_p [%]	preprocessing search [s]	preprocessing RAM [MiB]	search spaces [MiB]	search spaces TTF #	search spaces point #	query time [μs]	query scan #	query meet #	query eval #	query succ #	query profile [μs]
100	-	-	0.1	7.5	6 506	1 639	172	2 757	5.17	310	19.6	9.97	3.92	1 329
500	-	-	0.1	33.8	13 115	8 228	172	2 768	7.43	312	20.0	10.38	4.08	1 494
1 000	-	-	0.1	68.0	21 358	16 454	173	2 754	7.97	313	19.9	10.28	4.04	1 412
1 000	-	-	-	53.1	20 830	15 897	173	2 754	7.99	313	19.9	10.28	4.04	7 633
1 000	1.0	-	-	1.5	1 579	349	173	54.4	6.13	313	19.9	10.27	4.05	108.2
1 000	-	1.0	-	64.9	5 302	72	173	6.3	6.46	313	19.9	10.60	4.05	18.4
1 000	0.1	0.1	-	4.7	1 749	189	173	26.7	6.29	313	19.9	10.32	4.04	52.8
1 000	1.0	1.0	-	1.8	1 303	65	173	5.1	5.48	313	19.9	10.36	4.05	15.1
1 000	10.0	10.0	-	0.7	854	47	173	1.9	6.34	313	19.9	15.79	4.05	22.0
10 000	1.0	1.0	-	18.2	2 015	650	174	5.1	6.80	315	19.9	10.34	4.01	16.3

Table 2 Performance of the MINCANDIDATE algorithm.

size	ε_e [%]	ε_s [%]	preprocessing search [s]	preprocessing link [s]	preprocessing RAM [MiB]	$c_{\min}$ [MiB]	search space [MiB]	query time [μs]	query scan #	query meet #	query eval #	query succ #	query profile [μs]
100	-	-	6.0	0.0	6 481	1	1 583	3.11	310	19.6	3.65	1.04	6 941
1 000	-	-	53.1	0.4	20 849	7	15 897	4.97	313	19.9	3.72	1.05	7 087
1 000	1.0	1.0	1.8	0.4	1 310	7	65	4.09	313	19.9	3.77	1.07	13.8
10 000	1.0	1.0	18.2	49.0	2 777	649	650	4.94	315	19.9	3.81	1.07	14.4

time is spent on computing the large resulting TTFs. Approximating the edge TTFs ($\varepsilon_e > 0$) reduces preprocessing time and RAM, approximating search spaces ($\varepsilon_s > 0$) reduces search space sizes. When we combine both to $\varepsilon_e = \varepsilon_s = 1\%$, we reduce preprocessing time by a factor of 30 and search space size by 240. We can only compare with TCH for approximated edge TTFs, as TCH computes the search spaces at query time and does not approximate any intermediate result. For $\varepsilon_e = 1\%$, we are 27 times faster than TCH (2.94 ms [3]). But it pays off to approximate the search space TTFs, for $\varepsilon_e = \varepsilon_s = 0.1\%$, we are 56 times faster than TCH and even have smaller error (Table 6). Usually we would expect that the query time is independent of the table size, however, due to cache effects, we see an increase with increasing table size and a decrease with increasing ε's. Still the number of TTF evaluations is around 10 and thus 5 times large than the optimal (just 2).

By storing the minimal candidate (MINCANDIDATE, Table 2), we can reduce the number of evaluations to 3.7, which also reduces the query time. Although the precomputation is in $\Theta(|S| \cdot |T|)$, this only becomes significant for size 10 000 (or larger). The time query is only about one third faster, as we still scan on average about 310 nodes in the forward/backward search spaces. For exact profile queries, there is no advantage to INTERSECT as we can afford ε_p-bound pruning at query time there.

Algorithm RELEVANTCANDIDATE (Table 3) makes scanning obsolete. It stores 1.2–3.4 candidates per source/target-pair, depending on used approximations. This is significantly smaller than the 20 meeting nodes we had before, accelerating the time query by a factor of 2–4. But being in $\Theta(|S| \cdot |T|)$ becomes already noticeable for size 1 000. Again, the exact profile query does not benefit. Due to the knowledge of *all* relevant candidates, we only

Table 3 Performance of the RelevantCandidate algorithm.

size	ε_e [%]	ε_s [%]	ε_p [%]	preprocessing search [s]	link [s]	RAM [MiB]	c_{rel} [MiB]	c_{rel} #	search spaces [MiB]	TTF #	TTF [%]	point #	query time [μs]	eval #	profile [μs]
100	-	-	0.1	7.5	0.3	6 517	1	1.2	246	21	12	3 453	0.72	2.26	1 202
1 000	-	-	0.1	68.0	59.7	35 550	32	1.2	3 565	40	23	2 690	1.29	2.27	1 412
1 000	-	-	1.0	67.4	14.2	23 931	34	1.4	4 195	46	27	2 737	1.36	2.55	2 032
1 000	-	-	10.0	65.9	8.8	22 369	49	3.3	8 965	82	47	3 277	2.00	3.71	7 484
1 000	-	-	-	53.1	6.1	20 879	49	3.4	9 343	86	50	3 264	2.00	3.72	7 651
1 000	1.0	1.0	-	1.8	5.0	1 400	46	3.0	31	82	47	5.5	1.02	3.76	10.5
10 000	1.0	1.0	-	18.2	651.3	9 310	4 605	3.0	415	110	63	5.6	1.71	3.80	11.7

Table 4 Performance of the OptCandidate algorithm.

size	ε_e [%]	ε_s [%]	ε_p [%]	preprocessing search [s]	link [s]	RAM [MiB]	c_{opt} [MiB]	c_{opt} #	search spaces [MiB]	TTF #	TTF [%]	point #	query time [μs]	profile [μs]
100	-	-	0.1	7.5	2.1	6 494	1	1.5	241	21	12	3 456	0.49	1 168
500	-	-	0.1	33.8	53.7	13 101	10	1.5	1 608	33	19	2 946	0.73	1 391
1 000	-	-	0.1	68.0	213.8	21 443	39	1.5	3 489	39	22	2 704	0.81	1 332
1 000	-	-	-	53.1	1 032.2	20 908	39	1.5	3 489	39	22	2 704	0.84	1 339
1 000	1.0	-	-	1.5	19.4	1 666	37	1.4	72	39	23	49.5	0.56	21.6
1 000	-	1.0	-	64.9	7.1	5 318	39	1.5	17	41	23	6.0	0.51	3.5
1 000	0.1	0.1	-	4.7	11.5	1 822	39	1.5	42	39	22	25.9	0.54	9.8
1 000	1.0	1.0	-	1.8	6.3	1 385	38	1.5	15	41	24	4.9	0.48	3.0
1 000	10.0	10.0	-	0.7	7.6	963	62	3.1	19	68	39	2.0	0.49	5.7
10 000	1.0	1.0	-	18.2	788.3	7 741	3 775	1.5	226	63	36	5.0	0.90	3.5

need to store 12% of all computed search space TTFs for size 100. This percentage increases naturally when the table size increases, or when we use worse bounds for pruning (larger ε_p), allowing a flexible tradeoff between preprocessing time and space. Note that this algorithm can be used to make the Intersect algorithm more space-efficient by storing only the required TTFs and dropping c_{rel}.

We reduce the time query below 1 μs for any tested configuration with the OptCandidate algorithm (Table 4),[4] as we always just need two TTF evaluations. Exact precomputation time becomes very expensive due to the high number of TTF points, pruning with ε_p-bounds has a big impact by almost a factor 4. However, in the heuristic scenario, precomputation is just around 25% slower than RelevantCandidate (ε_p-pruning brings no speed-up), but provides more than 80% smaller search spaces and 3 times faster profile queries.

Naturally the best query times are achieved with the Table algorithm (Table 5). They are around a factor two smaller than OptCandidate, and up to 3 000 times faster than a TCH time query, and 4 000 000 times faster than a time-dependent Dijkstra [3]. Note that we do not report profile query timings as they are a simple table look-up. The larger precomputation time compared to OptCandidate comes from the additional overhead to store the table. We cannot compute exact tables larger than size 500. But practical cases of

[4] $\#c_{\text{opt}} > \#c_{\text{rel}}$ is possible when candidates are optimal for several periods of time.

Table 5 Performance of the TABLE algorithm.

size	ε_e [%]	ε_s [%]	ε_p [%]	ε_t [%]	preprocessing search [s]	preprocessing link [s]	preprocessing RAM [MiB]	table [MiB]	table points #	query time [μs]
100	-	-	0.1	-	7.5	1.9	7 638	1 086	7 672	0.25
500	-	-	0.1	-	33.8	58.5	45 659	27 697	7 829	0.42
500	-	-	-	-	26.6	266.7	45 532	27 697	7 829	0.42
500	1.0	-	-	-	0.8	4.8	1 924	427	117.6	0.26
1 000	1.0	-	-	-	1.5	19.0	3 625	1 689	116.3	0.32
1 000	1.0	1.0	-	-	1.8	6.3	1 577	180	9.6	0.25
1 000	-	-	0.1	1.0	68.0	298.2	21 489	110	4.6	0.25
1 000	0.1	0.1	-	0.1	4.7	12.3	2 112	270	16.0	0.26
1 000	1.0	1.0	-	1.0	1.8	6.7	1 484	94	3.4	0.23
1 000	10.0	10.0	-	10.0	0.7	7.1	1 017	76	2.1	0.22
10 000	1.0	1.0	-	-	18.2	772.1	27 118	18 109	9.7	0.39
10 000	1.0	1.0	-	1.0	18.2	815.2	17 788	9 342	3.4	0.38

Table 6 Observed errors from 100 000 queries together with the theoretical error bounds.

graph ε_e [%]	1.0	-	0.1	1.0	10	-	0.1	1.0	10
search space ε_s [%]	-	1.0	0.1	1.0	10	-	0.1	1.0	10
table ε_t [%]	-	-	-	-	-	1.0	0.1	1.0	10
avg. error [%]	0.08	0.12	0.014	0.18	2.1	0.17	0.023	0.30	3.1
max. error [%]	0.89	0.98	0.169	1.75	16.9	1.00	0.266	2.66	24.9
theo. bound [%]	2.07	1.44	0.350	3.55	41.0	1.00	0.450	4.58	55.1

size 1 000 can be computed with less than 2 GiB of RAM when we use approximations (table TTFs are ε_t-approximations).

Compared to $|S| \cdot |T| = 500 \cdot 500$ exact TCH profile queries, taking 1023 s on 8 threads, our algorithm achieves a speed-up of 11. This speed-up increases for $\varepsilon_e = 1\%$ to 16 since there the duplicate work of the TCH queries has a larger share. And it increases further for table size 1 000, our speed-up is then 18, as our search increases linearly and only linking is quadratic in size. We were not able to compare ourselves to a plain single source Dijkstra profile query, as our implementation of it runs out of RAM (48 GiB) after around 30 minutes.

A visual comparison of all five algorithms is Figure 1. We see that the decrease for time and profile query are almost independent of the size. However, the quadratic part (upper part of bar) for preprocessing time and space is very dominant for size = 10 000. Also, the OPTCANDIDATE algorithm requires less space than the RELEVANTCANDIDATE algorithm, as we need to store less candidates and can drop more entries from the stored search spaces.

We analyze the observed errors and theoretical error bounds[5] for size 1 000 in Table 6. Note that these error bounds are independent of the used algorithm. The maximum slope[6] is $\alpha = 0.433872$. The average observed error is always below the used ε's, however, we still see the stacking effect. Our bound is about a factor of two larger than the maximum

[5] Note that $\varepsilon_p > 0$ used for pruning does not cause errors.
[6] To compute the maximum slope, we compute the forward and backward search space for every node in the graph. But this is *only* required for the theoretical error bounds, and not used in our algorithms.

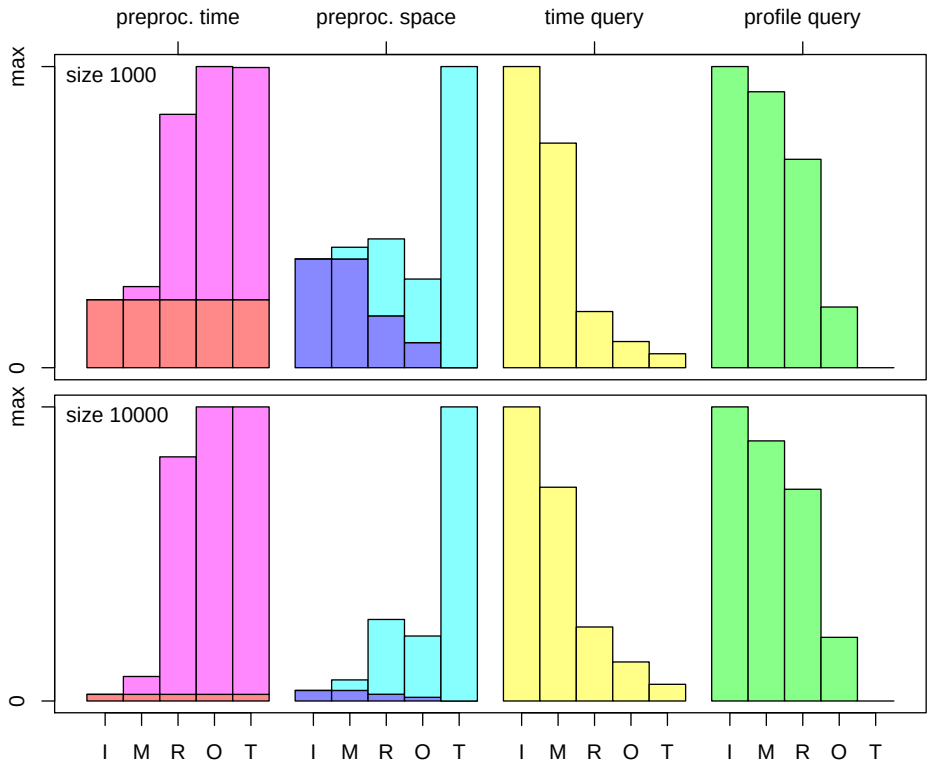

Figure 1 Comparison of the INTERSECT, MINCANDIATE, RELEVANTCANDIDATE, OPTCANDIDATE and TABLE algorithm with $\varepsilon_e = \varepsilon_s = 1\%$, $\varepsilon_p = \varepsilon_t = 0\%$. Preprocessing time is split into search (lower) and link (upper) and space is split into TTFs (lower) and additional data (upper). The vertical axis is relative to the maximum compared value in each group. Exact values are in Tables 1–5.

observed error for the edge approximations (ε_e). This is because our bound assumes that any error stacks during the linking of the TTFs, however, in practice TTFs do not often significantly change. The same explanation holds for the search space approximations (ε_s), although our bound is better since we only need to link two TTFs. When we combine edge and search space approximations, the errors roughly add up, this is because approximating an already approximated TTF introduces new errors. Approximating the TTFs in the table (ε_t) gives the straight ε_t-approximation unless it is based on already approximated TTFs. The provided theoretical bounds are pretty tight for the tested TCH instance, just around a factor of two larger than the maximum observed bounds.

7 Conclusions and Future Work

The computation of forward and backward search spaces in the TCH only once for each source and target node speeds up travel time table computations and subsequent queries by intersecting these search spaces. For exact profile queries, only a table can significantly improve the runtime. For exact time queries, and all approximate queries, further algorithms precompute additional data to speed up the queries with different tradeoffs in precomputation time and space. A large impact on the precomputation time and space has the use of approximate TTFs. We are able to reduce time by more than one and space by more than two orders of magnitude with an average error of less than 1%.

Our algorithms are also an important step to a time-dependent transit node routing algorithm [2]. Transit node routing is currently the fastest speedup technique for time-independent road networks and essentially reduces the shortest path search to a few table lookups. Our algorithms can either compute or completely replace such tables.

Acknowledgments. We thank G. Veit Batz for his great implementation of TCH that made developing our extensions very comfortable.

References

1 Yossi Azar, Y. Bartal, E. Feuerstein, Amos Fiat, Stefano Leonardi, and A. Rosen. On Capital Investment. *Algorithmica*, 25(1):22–36, 1999.

2 Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

3 Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.

4 Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*. Springer, May 2010.

5 Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

6 Alberto V. Donati, Roberto Montemanni, Norman Casagrande, Andrea E. Rizzoli, and Luca M. Gambardella. Time dependent vehicle routing problem with a multi ant colony system. *European Journal of Operational Research*, 185:1174–1191, 2008.

7 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

8 Hideki Hashimoto, Mutsunori Yagiura, and Toshihide Ibaraki. An Iterated Local Search Algorithm for the Time-Dependent Vehicle Routing Problem with Time Windows. *Discrete Optimization*, 5:434–456, 2008.

9 Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Vehicle Dispatching with Time-Dependent Travel Times. *European Journal of Operational Research*, 144:379–396, 2003.

10 H. Imai and Masao Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of Information Processing*, 9(3):159–162, 1987.

11 Soojung Jung and Ali Haghani. Genetic Algorithm for the Time-Dependent Vehicle Routing Problem . *Journal of the Transportation Research Board*, 1771:164–171, 2001.

12 Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

13 Chryssi Malandraki and Mark S. Daskin. Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms. *Transportation Science*, 26(3):185–200, 1992.

14 Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.