Engineering Time-dependent One-To-All Computation

Robert Geisberger
Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
geisberger@kit.edu

October 5, 2010

Abstract

Very recently a new algorithm to the nonnegative single-source shortest path problem on road networks has been discovered. It is very cache-efficient, but only on static road networks. We show how to augment it to the time-dependent scenario. The advantage if the new approach is that it settles nodes, even for a profile query, by scanning all downward edges. We improve the scanning of the downward edges with techniques developed for time-dependent many-to-many computations.

1 Introduction

The new algorithm for the nonnegative single-source shortest path problem on road networks [5] uses contraction hierarchies [8]. First, it performs a forward upward search, and then it processes all nodes in descending order of importance. The algorithm is so efficient because it can order the edges very cache-efficient and can be parallelized. We consider the time-dependent scenario, where the travel time depends on the departure time, the edge weights are complex travel time functions (TTFs). Such a TTF maps a departure time to the travel time. We want to solve the problem of computing the travel time profiles from one source node to all other nodes in the graph. A travel time profile is a TTF that maps the departure time at the source to the earliest arrival time at the target node. Significantly the most work on solving our problem is to process the TTFs, the traversal of the graph takes negligible time. So the cache-efficient order of the edges brings is no longer a significant speed-up. But the way the edges are processed allows to prune a lot of expensive TTF operations by using approximate TTFs.

1.1 Related Work

Many new algorithms for the time-dependent point-to-point shortest path problem have been developed recently. We refer to [6] for an overview. Also, a time-dependent version of contraction hierarchies (TCH) [1, 2] exists, and we use it for our new algorithm. It was augmented to compute travel time tables by [7].

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 $\mathrm{O}(|f| + |g|)$ time and $|g * f| \in \mathrm{O}(|f| + |g|)$ holds. Linking is an associative operation, i.e., $f * (g * h) = (f * g) * h$ for TTFs f, g, h.
- *Minimum.* Given two parallel edges e, e' from u to v with TTFs f, f', we want to *merge* these edges into one while preserving all shortest paths. The resulting single edge e'' from u to v gets the TTF $\min(f, f')$ defined by $\tau \mapsto \min\{f(\tau), f'(\tau)\}$. It can be computed in $\mathrm{O}(|f| + |f'|)$ time and $|\min(f, f')| \in \mathrm{O}(|f| + |f'|)$ holds.

In a time-dependent road network, shortest paths depend on the departure time. For 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.

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 [9]. It yields a TTF with minimal number of points for ε in time $\mathrm{O}(|f|)$.

2.2 Time-Dependent Profile Dijkstra

To compute the travel time profile from a source node s to all other nodes, we can use a label correcting modification of Dijkstra's algorithm [10]. The modifications are as follows:

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

[2] Here, all TTFs have period $\Pi = 24\text{h}$. However, using non-periodic TTFs makes no real difference. Of course, covering more than 24h will increase the memory usage.

- *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 $f_{uv} * f_u$.

2.3 Time-Dependent Contraction Hierarchies

Hierarchies. In a *time-dependent contraction hierarchy* [1] all nodes of G are *ordered* by increasing 'importance' [8]. 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 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. This property is reflected in the *upward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow := \{(u, v) \in E \mid u < v\}$ and, the *downward graph* $G_\downarrow := (V, E_\downarrow)$ with $E_\downarrow := \{(u, v) \in E \mid u > v\}$. 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 $\min\{g_u * f_u \mid u \text{ candidate}\}$.

3 Our Algorithm

Given a source node s, we want to compute for each node u in the graph the travel time profile $\delta(u)$ from s to u. We initialize $\delta(s) := (\tau \mapsto 0)$ and all other TTFs $\delta(u) = (\tau \mapsto \infty)$. Then, we perform a forward search from s in $G_\uparrow$ and update the tentative travel time profile $\delta(u)$ for all nodes visited by this search. Now, we process all nodes in the graph by descending importance level and compute

$$\delta(u) := \min\left(\delta(u), \min\{f_v * \delta(v) \mid v \to_{f_v} u \in E_\downarrow\}\right) \tag{1}$$

So essentially, we scan through all incoming downward edges $v \to_f u$ of u, link them to $\delta(v)$ and build the minimum. Doing this naively is very costly, as the travel time functions are complex. As not all of the nodes v contribute to the

minimum in the end, we improve the performance by using pruning techniques developed for the time-dependent travel time table computation. Assume that we have lower/upper ε-bounds $f_v^{\downarrow}$ / $f_v^{\uparrow}$ for f_v and $\delta(v)^{\downarrow}$ / $\delta(v)^{\uparrow}$ for $\delta(v)$. Then we can use Algorithm 1 to accelerate the computation of $\delta(u)$.

Algorithm 1: BuildMinimum(u)

```
1  δ̄ := min_{v→_{f_v} u∈E↓} {max f_v + max δ(v)};        // upper bound based on
   maxima
2  (v → ·) := argmin_{v→_{f_v} u∈E↓} {min f + min δ(v)};          // minimum node
3  δ↑ := f_v↑ ∗ δ(v)↑;              // upper bound based on approximate TTFs
4  δ̄ := min(δ̄, max δ↑);                                  // tighten upper bound
5  foreach v →_{f_v} u ∈ E↓ do                 // loop over all downward edges
6  |  if min f_v + min δ(v)↓ ≤ δ̄ then                    // prune using minima
7  |  |  δ↑ := min(δ↑, f_v↑ ∗ δ(v)↑);                    // update upper bound
8  δ(u) := min(δ(u), f_v ∗ δ(v));                // tentative travel time profile
9  foreach v →_{f_v} u ∈ E↓ do                 // loop over all downward edges
10 |  if ¬(f_v↓ ∗ δ(u)↓ > δ↑) then                  // prune using lower bounds
11 |  |  δ(u) := min(δ(u), f_v ∗ δ(v));          // update travel time profile
```

We pass three times through the incoming downward edges $v \to_{f_v} u \in E_{\downarrow}$.

1. In Line 1 we compute an upper bound $\overline{\delta}$ based on the maxima of f_v and $\delta(v)$. Also, in Line 2 we compute the edge with minimum sum of the minima of f_v and $\delta(v)$. This edge 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_v and $\delta(v)$ at Line 11.

In comparison, a Profile Dijkstra would update $\delta(u)$ gradually when he processes node v, and thus cannot pass through all the downward edges several times to compute the intermediate upper bounds $\overline{\delta}$ and $\delta^{\uparrow}$.

4 Improvements

Some of the improvements from [5] can be applied, especially the reordering of nodes, and the parallel processing. However, we did not adopt SIMD instructions or GPU, as we now operate on complex TTFs.

5 Core-based Computation

An interesting observation is that [5] cannot be used like Dijkstra's algorithm to compute the distances to close-by nodes, as the nodes are no longer processed in order of increasing distance. However, we can compute the distances only for a core of the contraction hierarchy, that is a number of k most important nodes.

algorithm	threads	prune ε [%]	query time [s]
Dijkstra	-	-	116
TCH	1	-	105
TCH	1	10	108
TCH	1	1	48.2
TCH	1	0.1	32.6
TCH	1	0.01	35.6
TCH	2	0.1	16.0
TCH	4	0.1	8.8
TCH	6	0.1	6.5
TCH	8	0.1	5.7

Table 1: Performance.

Computing only the distances to all core nodes is faster, especially in the time-dependent scenario. Also, it takes much less space, as the TTFs usually contain thousands of points.

An application of the core-based computation is the computation of arc flags [5, §7.2], but now only for a core. In [3] it is observed that this brings significant speed-ups for the static scenario. However, in the time-dependent scenario, upper/lower bounds used to compute exact arc flags, as using exact TTFs is too time-consuming [4]. These arc flags are very weak, and simple heuristic computation of the arc flags using time-sampling significantly accelerate the query, but provide no approximation guarantee [4]. So we expect that when we are able to compute exact arc flags with exact TTFs, this provides both a fast and exact query.

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[3] with two Intel Xeon E5345 processors (Quad-Core) clocked at 2.33 GHz with 16 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 [1]. We use a core-size of 10 000, and select 100 core nodes uniformly at random as source of our query.

The experimental results are presented in Table 1. The number of threads corresponds to the number used to answer a single query.

[3]The machine used in [2, 7] is currently repaired, we plan to publish results for this machine later.

7 Conclusion

We presented an efficient algorithm for time-dependent one-to-all computation. By applying it to a core, it can be used to accelerate precomputation of speed-up techniques.

References

[1] 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.

[2] 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*, pages 166–177. Springer, May 2010.

[3] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[4] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, July 2009. Special Issue: European Symposium on Algorithms 2008.

[5] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. Technical Report MSR-TR-2010-125, Microsoft Research, 2010.

[6] 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.

[7] Robert Geisberger and Peter Sanders. 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)*, OpenAccess Series in Informatics (OASIcs), 2010.

[8] 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.

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

[10] 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.