Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks*

Daniel Delling[1] and Giacomo Nannicini[2,3]

[1] Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
delling@ira.uka.de
[2] LIX, École Polytechnique, F-91128 Palaiseau, France
giacomon@lix.polytechnique.fr
[3] Mediamobile, 27 bd Hyppolite Marques, 94200 Ivry Sur Seine, France
giacomo.nannicini@v-trafic.com

Abstract. In a recent work [1], we proposed a point-to-point shortest paths algorithm which applies bidirectional search on time-dependent road networks. The algorithm is based on A^* and runs a backward search in order to bound the set of nodes that have to be explored by the forward search. In this paper we extend the bidirectional time-dependent search algorithm in order to allow core routing, which is a very effective technique introduced for static graphs that consists in carrying out most of the search on a subset of the original node set. Moreover, we tackle the dynamic scenario where the piecewise linear time-dependent arc cost functions are not fixed, but can have their coefficients updated. We provide extensive computational results to show that our approach is a significant speed-up with respect to the original algorithm, and it is able to deal with the dynamic scenario requiring only a small computational effort to update the cost functions and related data structures.

1 Introduction

The Shortest Path Problem (SPP) on static graphs has received a great deal of attention in recent years, because it has interesting practical applications (e.g. route planners for GPS devices, web services) and provides an algorithmic challenge. Several works propose efficient algorithms for the SPP: see [2] for a review, and [3] for an interesting analysis of possible combinations of speed-up techniques.

Much of the focus is now moving to the Time-Dependent Shortest Path Problem (TDSPP), which can be formally stated as follows: given a directed graph $G = (V, A)$, a source node $s \in V$, a destination node $t \in V$, an interval of time instants T, a departure time $\tau_0 \in T$ and a time-dependent arc cost function $c : A \times T \to \mathbb{R}_+$, find a path $p = (s = v_1, \ldots, v_k = t)$ in G such that its *time-dependent cost* $\gamma_{\tau_0}(p)$, defined recursively as follows:

$$\gamma_{\tau_0}(v_1, v_2) = c(v_1, v_2, \tau_0) \tag{1}$$

$$\gamma_{\tau_0}(v_1, \ldots, v_i) = \gamma_{\tau_0}(v_1, \ldots, v_{i-1}) + c(v_{i-1}, v_i, \tau_0 + \gamma_{\tau_0}(v_1, \ldots, v_{i-1})) \tag{2}$$

for all $2 \leq i \leq k$, is minimum.

* Partially supported by the DFG (project WA 654/16-1) and the EU under grant ARRIVAL (contract no. FP6-021235-2).

The TDSPP has been first addressed by [4] with a recursion formula; Dijkstra's algorithm [5] is then extended to the dynamic case in [6], and the FIFO property, which is necessary to guarantee correctness, is implicitly assumed. The FIFO property is also called the *non-overtaking property*, because it states that if T_1 leaves u at time τ_0 and T_2 at time $\tau_1 > \tau_0$, T_2 cannot arrive at v before T_1 using the arc (u, v). The TDSPP in FIFO networks is polynomially solvable [7], while it is NP-hard in non-FIFO networks [8]. We focus on the FIFO variant. The A^* algorithm [9] has been adapted to efficiently compute shortest paths on static road networks in [10,11]. Those ideas have been used in [12] on dynamic graphs as well, while the time-dependent case on graphs with the FIFO property has been addressed in [13,12,1]. The SHARC-algorithm [14], which employs a hierarchical approach combined with goal directed search via arc flags [15], allows fast unidirectional shortest path calculations in large scale networks; it has been recently extended in [16] to compute optimal paths even on time-dependent graphs, and represents the fastest known algorithm so far for time-dependent shortests path computations.

Bidirectional search cannot be directly applied on time-dependent graphs, the optimal arrival time at the destination being unknown. In [1], we tackled this problem running a forward search on the time-dependent graph, and a backward search on a time-independent graph with the purpose of bounding the set of nodes explored by the forward search. To the best of our knowledge, it was the first method allowing practical shortest path computations (i.e., in less than 300 msec) on large scale time-dependent road networks. In this paper we extend those concepts in order to include *core routing* on the time-dependent graph, and we analyze a dynamic scenario as well, in order to take into account updates of the cost function. Core routing is a well known technique for shortest path algorithms on static graphs [3], whose main idea is to shrink the original graph in order to get a new graph (*core*) with a smaller number of vertices. Most of the search is then carried out on the core, yielding a reduced search space.

Throughout the rest of this paper we will consider a lower bounding function $\lambda : A \rightarrow \mathbb{R}_+$ such that $\forall (u, v) \in A, \tau \in T$ we have $\lambda(u, v) \leq c(u, v, \tau)$. In practice, λ can easily be computed, given an arc length and the maximum allowed speed on that arc. In the experimental evaluation we will consider piecewise linear time-dependent arc cost functions.

The rest of this paper is organized as follows. In Section 2 we briefly review A^* and the bidirectional A^* algorithm applied on a time-dependent graph described in [1]. In Section 3 we describe core routing on static graphs and generalize it to the time-dependent case. In Section 4 we discuss the dynamic scenario. In Section 5 we provide a detailed experimental evaluation of our method, and analyze the results.

2 A^* with Landmarks

A^* is an algorithm for goal-directed search which is very similar to Dijkstra's algorithm. The difference between the two algorithms lies in the priority key.

For A^*, the priority key of a node v is made up of two parts: the length of the tentative shortest path from the source to v (as in Dijkstra's algorithm), and an underestimation of the distance to reach the target from v. The function which estimates the distance between a node and the target is called potential function π; the use of π has the effect of giving priority to nodes that are (supposedly) closer to target node t. If the potential function is such that $\pi(v) \leq d(v,t)\,\forall v \in V$, where $d(v,t)$ is the distance from v to t, then A^* always finds shortest paths [9]; otherwise, it becomes a heuristic. A^* is guaranteed to explore no more nodes than Dijkstra's algorithm.

On a road network, Euclidean distances can be used to compute the potential function, possibly dividing by the maximum allowed speed if arc costs are travelling times instead of distances. A significant improvement over Euclidean potentials can be achieved using *landmarks* [10]. The main idea is to select a small set of nodes in the graph, sufficiently spread over the whole network (several heuristic selection strategies have been proposed — see [17]), and precompute all distances between landmarks and any node of the vertex set. Then, by triangle inequalities, it is possible to derive lower bounds to the distance between any two nodes. Suppose we have selected a set $L \subset V$ of landmarks, and we have stored all distances $d(v,\ell), d(\ell,v)\,\forall v \in V, \ell \in L$; the following triangle inequalities hold: $d(u,t) + d(t,\ell) \geq d(u,\ell)$ and $d(\ell,u) + d(u,t) \geq d(\ell,t)$. Therefore $\pi_f(u) = \max_{\ell \in L}\{d(u,\ell) - d(t,\ell), d(\ell,t) - d(\ell,u)\}$ is a lower bound for the distance $d(u,t)$, and it can be used as a valid potential function for the forward search [10]. Bidirectional search can be applied, but the potential function must be consistent for the forward and backward search [11]. Bidirectional A^* with the potential function described above is called ALT; an experimental evaluation on static graphs can be found in [11]. It is straightforward to observe that, if arc costs can only increase with respect to their original value, the potential function associated with landmarks yields valid lower bound, even on a time-dependent graph; in [12] this idea is applied to a real road network in order to analyse the algorithm's performance both in the case of arc cost updates and of time-dependent cost functions, but in the latter scenario the ALT algorithm is applied in an unidirectional way.

In a recent work [1], a bidirectional ALT algorithm on time-dependent road networks was proposed. The algorithm is based on restricting the scope of a time-dependent A^* search from the source using a set of nodes defined by a time-*independent* A^* search from the destination. The backward search is a reverse search on the graph G weighted by the lower bounding function λ.

Given a graph $G = (V, A)$, source and destination vertices $s, t \in V$, and a departure time $\tau_0 \in T$, let p^* be the shortest path from s to t leaving node s at τ_0. The algorithm for computing p^* works in three phases.

1. A bidirectional A^* search occurs on G, where the forward search is run on the graph weighted by c with the path cost defined by (1)-(2), and the backward search is run on the graph weighted by the lower bounding function λ. All nodes settled by the backward search are included in a set M. Phase 1 terminates as soon as the two search scopes meet.

2. Suppose that $v \in V$ is the first vertex in the intersection of the heaps of the forward and backward search; then the time dependent cost $\mu = \gamma_{\tau_0}(p_v)$ of the path p_v going from s to t passing through v is an upper bound to $\gamma_{\tau_0}(p^*)$. Let β be the key of the minimum element of the backward search queue; phase 2 terminates as soon as $\beta > \mu$. Again, all nodes settled by the backward search are included in M.
3. Only the forward search continues, with the additional constraint that only nodes in M can be explored. The forward search terminates when t is settled.

We call this algorithm Time-Dependent ALT (TDALT). Given a constant $K > 1$, K-approximated solutions can be computed switching from phase 2 to phase 3 as soon as $\beta > K\mu$; as the search stops sooner, the number of explored nodes decreases. We use the backward potential function $\pi^*_b(w) = \max\{\pi_b(w), d(s, v, \tau_0) + \pi_f(v) - \pi_f(w)\}$ described in [1], where π_f and π_b are the landmark potential functions for, respectively, the forward and the backward search, and v is a node already settled by the forward search. To guarantee correctness of this approach (see [1]), we do the following: we set up 10 checkpoints during the query; when a checkpoint is reached, the node v is updated, and the backward search queue is flushed and filled again using the updated π^*_b. We always pick v as the last node settled by the forward search before the checkpoint. The checkpoints are calculated comparing the initial lower bound $\pi_f(t)$ and the current distance from the source node, both for the forward search.

3 Time-Dependent Core-Based Routing

Core-based routing is a powerful approach which has been widely used for shortest paths algorithms on static graphs [3]. The main idea is to use contraction [18]: a routine iteratively removes nodes and adds edges to preserve correct distances between the remaining nodes, so that we have a smaller network where most of the search can be carried out. Note that in principle we can use any contraction routine which removes nodes from the graph and inserts edges to preserve distances. When the contracted graph $G_C = (V_C, A_C)$ has been computed, it is merged with the original graph to obtain $G_F = (V, A \cup A_C)$.

Suppose that we have a contraction routine which works on a time-dependent graph: that is, $\forall u, v \in V_C$, for each departure time $\tau_0 \in T$ there is a shortest path between u and v in G_C with the same cost as the shortest path between u and v in G with the same departure time. We propose the following query algorithm.

1. Initialization phase: start a Dijkstra search from both the source and the destination node on G_F, using the time-dependent costs for the forward search and the time-independent costs λ for the backward search, pruning the search (i.e. not relaxing outgoing arcs) at nodes $\in V_C$. Add each node settled by the forward search to a set S, and each node settled by the backward search to a set T. Iterate between the two searches until: (i) $S \cap T \neq \emptyset$ or (ii) the priority queues are empty.

2. Main phase: (*i*) If $S \cap T \neq \emptyset$, then start an unidirectional Dijkstra search from the source on G_F until the target is settled. (*ii*) If the priority queues are empty and we still have $S \cap T = \emptyset$, then start TDALT on the graph G_C, initializing the forward search queue with all leaves of S and the backward search queue with all leaves of T, using the distance labels computed during the initialization phase. The forward search is also allowed to explore any node $v \in T$, throughout the 3 phases of the algorithm. Stop when t is settled by the foward search.

In other words, the forward search "hops on" the core when it reaches a node $u \in S \cap V_C$, and "hops off" at all nodes $v \in T \cap V_C$. Note that landmark distances need be computed and stored only for vertices in V_C (see [3]). This means that the landmark potential function cannot be used to apply the forward A^* search on the nodes in T. However, we can use the backward distance labels computed with Dijkstra's algorithm during the initialization phase, which are valid distances on G_λ. We call this algorithm Time-Dependent Core-based ALT (TDCALT).

Proposition 3.1. TDCALT *is correct.*

Since landmark distances are available only for nodes in V_C, the ALT potential function cannot be used "as is" whenever the source or the destination node do not belong to the core. In order to compute valid lower bounds to the distances from s or to t, proxy nodes have been introduced in [19] and used for the CALT algorithm (i.e. core-based ALT on a static graph) in [3]. We briefly report here the main idea: on the graph G weighted by λ, let $t' = \arg\min_{v \in V_C}\{d(t,v)\}$ be the core node closest to t. By triangle inequalities it is easy to derive a valid potential function for the forward search which uses landmark distances for t' as a proxy for t: $\pi_f(u) = \max_{\ell \in L}\{d(u,\ell) - d(t',\ell) - d(t,t'), d(\ell,t') - d(\ell,u) - d(t,t')\}$. The same calculations yield the potential function for the backward search π_b using a proxy node s' for the source s and the distance $d(s',s)$.

Contraction. For the contraction phase, i.e., the routine which selects which nodes have to be bypassed and then adds shortcuts to preserve shortest paths, we use the same algorithm proposed in [16]. We define the *expansion* [19] of a node u as the quotient between the number of added shortcuts and the number of edges removed if u is bypassed, and the *hop-number* of a shortcut as the number of edges that the shortcut represents. We iterate the contraction routine until the expansion of all remaining nodes exceeds a limit c or the hop-number exceeds a limit h. At the end of contraction, we perform an edge-reduction step which removes unnecessary shortcuts from the graph (cf. [16] for details).

Outputting Shortest Paths. TDCALT adds shortcuts to the graph in order to accelerate queries. Hence, if we want to retrieve the complete shortest path (and not only the distance) we must expand those shortcuts. In [20], an efficient unpacking routine based on storing all the edges a shortcut represents is introduced. However, in the static case a shortcut represents exactly one path, whereas in the time-dependent case a shortcut may represent a different path for each different traversal times. We solve this problem by allowing multi-edges: whenever a node is bypassed, a shortcut is inserted to represent each pair of incoming and outgoing edges, even if another edge between the two endpoints already exists. With this modification each shortcut represents exactly one path, so we can directly apply the unpacking routine from [20].

4 Dynamic Time-Dependent Costs

Up to now, time-dependent routing algorithms assumed complete knowledge of the time-dependent cost functions on arcs. However, since the speed profiles on which these functions are based are generated using historical data gathered from sensors (or cams), it is reasonable to assume that also real-time traffic information is available through these sensors. Moreover, other technologies exist to be aware of traffic jams even without having access to real-time speed information (e.g., TMC[1]). In the end, a procedure to update the time-dependent cost functions depending on real-time traffic information would be desirable for practical applications. Since we use piecewise linear functions stored as a list of breakpoints, we will consider modifications in the value of these.

Update procedure. Let (V_C, A_C) be the core of G. Suppose that the cost function of one arc $a \in A$ is modified; the set of core nodes V_C need not change, as long as A_C is updated in order to preserve distances with respect to the uncontracted graph $G = (V, A)$ with the new cost function. There are two possible cases: either the new values of the modified breakpoints are smaller than the previous ones, or they are larger. In the first case, then all arcs on the core A_C must be recomputed by running a label-correcting algorithm between the endpoints of each shortcut, as we do not know which shortcuts the updated arc may contribute to. In the second case, then the cost function for core arcs (i.e. shortcuts) may change for all those arcs $a' \in A_C$ such that a' contains a in its decomposition for at least one time instant τ. In other words, if a contributed to a shortcut a', then the cost of a' has to be recomputed. As the cost of a has increased, then a cannot possibly contribute to other shortcuts, thus we can restrict the update only to the shortcuts that contain the arc. To do so, we store for each $a \in A$ the set $S(a)$ of all shortcuts that a contributes to. Then, if one or more breakpoints of a have their value changed, we do the following.

Let $[\tau_1, \tau_{n-1}]$ be the smallest time interval that contains all modified breakpoints of arc a. If the breakpoints preceding and following $[\tau_1, \tau_{n-1}]$ are, respectively, at times τ_0 and τ_n the cost function of a changes only in the interval $[\tau_0, \tau_n]$. For each shortcut $a' \in S(a)$, let $a'_0, \dots, a'_d$, with $a'_i \in A \, \forall i$, be its decomposition in terms of the original arcs, let $\lambda_j = \sum_{i=0}^{j-1} \lambda(a'_i)$ and $\mu_j = \sum_{i=0}^{j-1} \mu(a'_i)$, where $\forall a \in A$ we define $\mu(a) = \max_{\tau \in T} c(a, \tau)$, i.e., $\mu(a)$ is an upper bound on the cost of arc a. If a is the arc with index j in the decomposition of a', then a' may be affected by the change in the cost function of a only if the departure

[1] http://www.tmcforum.com/

time from the starting point of a' is in the interval $[\tau_0 - \mu_j, \tau_n - \lambda_j]$. This is because a can be reached from the starting node of a' no sooner than λ_j, and no later than μ_j. Thus, in order to update the shortcut a', we need to run a label-correcting algorithm between its two endpoints only in the time interval $[\tau_0 - \mu_j, \tau_n - \lambda_j]$, as the rest of the cost function is not affected by the change. In practice, if the length of the time interval $[\tau_0, \tau_n]$ is larger than a given threshold we run a label-correcting algorithm between the shortcut's endpoints over the whole time period, as the gain obtained by running the algorithm over a smaller time interval does not offset the overhead due to updating only a part of the profile with respect to computing from scratch.

The procedure described above is valid only when the value of breakpoints increases. In a typical realistic scenario, this is often the case: the initial cost profiles are used to model normal traffic conditions, and cost updates occur only to add temporary slowdowns due to unexpected traffic jams. When the temporary slowdowns are no longer valid we would like to restore the initial cost profiles, i.e. lower breakpoints to their initial values, without recomputing the whole core. If we want to allow fast updates as long as the new breakpoint values are larger than the ones used for the initial core construction, without requiring that the values can only increase, then we have to manage the sets $S(a)\,\forall a \in A$ accordingly. We provide an example that shows how problems could arise.

Example 4.1. Given $a \in A$, suppose that the cost of its breakpoint at time $\tau \in T$ increases, and all shortcuts $\in S(a)$ are updated. Suppose that, for a shortcut $a' \in S(a)$, a does not contibute to a' anymore due to the increased breakpoint value. If a' is removed from $S(a)$ and at a later time the value of the breakpoint at τ is restored to the original value, then a' would not be updated because $a' \notin S(a)$, thus a' would not be optimal.

Our approach to tackle this problem is the following: for each arc $a \in A$, we update the sets $S(a)$ whenever a breakpoint value changes, with the additional constraint that elements of $S(a)$ after the initial core construction phase cannot be removed from the set. Thus, $S(a)$ contains all shortcuts that a contributes to with the current cost function, plus all shortcuts that a contributed to during the initial core construction. As a consequence we may update a shortcut $a' \in S(a)$ unnecessarily, if a contributed to a' during the initial core construction but ceased contributing after an update step; however, this guarantees correctness for all changes in the breakpoint values, as long as the new values are not strictly smaller than the values used during the initial graph contraction. From a practical point of view, this is a reasonable assumption.

Since the sets $S(a)\,\forall a \in A$ are stored in memory, the computational time required by the core update is largely dominated by the time required to run the label-correcting algorithm between the endpoints of shortcuts. Thus, we have a trade-off between query speed and update speed: if we allow the contraction routine to build long shortcuts (in terms of number of bypassed nodes, i.e. "hops", as well as travelling time) then we obtain a faster query algorithm, because we are able to skip more nodes during the shortest path computations. On the other hand, if we allow only limited-length shortcuts, then the query search space is larger, but the core update is significantly faster as the label-correcting algorithm takes less time. In Section 5 we provide an experimental evaluation for different scenarios.

5 Experiments

In this section, we present an extensive experimental evaluation of our time-dependent ALT algorithm. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 3.

We use 32 *avoid* landmarks [10], computed on the core of the input graph using the lower bounding function λ to weight edges, and we use the tightened potential function π_b^* described in Section 2 as potential function for the backward search, with 10 checkpoints. When performing random s-t queries, the source s, target t, and the starting time τ_0 are picked uniformly at random and results are based on 10 000 queries. In the following, we restrict ourselves to the scenario where only distances — not the complete paths — are required. However, our shortcut expansion routine for TDCALT needs less than 1 ms to output the whole path; the additional space overhead is ≈ 4 bytes per node.

Input. We tested our algorithm on the road network of Western Europe provided by PTV AG for scientific use, which has approximately 18 million vertices and 42.6 million arcs. A travelling time in uncongested traffic situation was assigned to each arc using that arc's category (13 different categories) to determine the travel speed. Since we are not aware of a *large* publicly available real-world road network with time-dependent arc costs we used artificially generated costs. In order to model the time-dependent costs on each arc, we developed a heuristic algorithm, based on statistics gathered using real-world data on a limited-size road network, which is described in [1] and ensures spatial coherency for traffic jams.

Contraction Rates. Table 1 shows the performance of TDCALT for different contraction parameters (cf. Section 3). In this setup, we fix the approximation constant K to 1.15, which was found to be a good compromise between speed and quality of computed paths (see [1]). As the performed TDCALT queries may compute approximated results instead of optimal solutions when $K > 1$, we record three different statistics to characterize the solution quality: error rate, average relative error, maximum relative error. By *error rate* we denote the percentage of computed suboptimal paths over the total number of queries. By *relative error* on a particular query we denote the relative percentage increase of the approximated solution over the optimum, computed as $\omega/\omega^* - 1$, where ω is the cost of the approximated solution computed by our algorithm and ω^* is the cost of the optimum computed by Dijkstra's algorithm. We report *average* and *maximum* values of this quantity over the set of all queries. Note that contraction parameters of $c = 0.0$ and $h = 0$ yield a pure TDALT setup.

Table 1. Performance of TDCALT for different contraction rates. c denotes the maximum expansion of a bypassed node, h the hop-limit of added shortcuts. The third column records how many nodes have *not* been bypassed applying the corresponding contraction parameters. Preprocessing effort is given in time and *additional* space in bytes per node. Moreover, we report the increase in number of edges and interpolation points of the merged graph compared to the original input.

Core			Preprocessing				Error			Query	
param.		core	time	space	increase in			relative		#settled	time
c	h	nodes	[min]	[B/n]	#edges	#points	rate	avg.	max	nodes	[ms]
0.0	0	100.0%	28	256	0.0%	0.0%	40.1%	0.303%	10.95%	250 248	188.2
0.5	10	35.6%	15	99	9.8%	21.1%	38.7%	0.302%	11.14%	99 622	78.2
1.0	20	6.9%	18	41	12.6%	69.6%	34.7%	0.288%	10.52%	19 719	21.7
2.0	30	3.2%	30	45	9.9%	114.1%	34.9%	0.287%	10.52%	9 974	13.2
2.5	40	2.5%	39	50	9.1%	138.0%	34.1%	0.275%	8.74%	8 093	11.4
3.0	50	2.0%	50	56	8.7%	161.2%	32.8%	0.267%	9.58%	7 090	10.3
3.5	60	1.8%	60	61	8.5%	181.1%	33.8%	0.280%	8.69%	6 227	9.2
4.0	70	1.5%	88	74	8.5%	223.1%	32.8%	0.265%	8.69%	5 896	8.8
5.0	90	1.2%	134	89	8.6%	273.5%	32.6%	0.266%	8.69%	5 812	8.4

As expected, increasing the contraction parameters has a positive effect on query performance. Interestingly, the space overhead first decreases from 256 bytes per node to 41 ($c = 1.0$, $h = 20$), and then increases again. The reason for this is that the core shrinks very quickly, hence we store landmark distances only for 6.9% of the nodes. On the other hand, the number of interpolation points for shortcuts increases by up to a factor ≈ 4 with respect to the original graph. Storing these additional points is expensive and explains the increase in space consumption.

It is also interesting to note that the maximum error rate decreases when we allow more and longer shortcuts to be built. We believe that this is due to the fact that long shortcuts decrease the number of settled nodes and have large costs, so at each iteration of TDCALT the key of the backward search priority queue β increases by a large amount. As the algorithm switches from phase 2 to phase 3 when $\mu/\beta < K$, and β increases by large steps, phase 3 starts with a smaller maximum approximation value for the current query μ/β. This is especially true for short distance queries, where the value of μ is small.

Query speed. Table 2 reports the results of TDCALT for different approximation values K using the European road network as input. In this experiment we used contraction parameters $c = 3.5$ and $h = 60$, i.e. we allow long shortcuts to be built to favour query speed. For comparison, we also report the results on the same road network for the time-dependent versions of Dijkstra, unidirectional ALT, TDALT and the time-dependent SHARC algorithm [16].

Table 2 shows that TDCALT yields a significant improvement over TDALT with respect to error rates, preprocessing space, size of the search space and query times. The latter two figures are improved by one order of magnitude. For exact queries, TDCALT is faster than unidirectional ALT by one order of

Table 2. Performance of time-dependent Dijkstra, unidirectional ALT, SHARC, TDALT and TDCALT with different approximation values K

		Preproc.		Error			Query	
		time	space		relative		# settled	time
technique	K	[min]	[B/n]	rate	av.	max	nodes	[ms]
Dijkstra	-	0	0	0.0%	0.000%	0.00%	8 877 158	5 757.4
uni-ALT	-	28	256	0.0%	0.000%	0.00%	2 056 190	1 865.4
SHARC	-	511	112	0.0%	0.000%	0.00%	84 234	75.3
TDALT	1.00	28	256	0.0%	0.000%	0.00%	2 931 080	2 939.3
	1.15	28	256	40.1%	0.303%	10.95%	250 248	188.2
	1.50	28	256	52.8%	0.734%	21.64%	113 040	71.2
TDCALT	1.00	60	61	0.0%	0.000%	0.00%	60 961	121.4
	1.05	60	61	2.7%	0.010%	3.94%	32 405	62.5
	1.10	60	61	16.6%	0.093%	7.88%	12 777	21.9
	1.15	60	61	33.0%	0.259%	8.69%	6 365	9.2
	1.20	60	61	39.8%	0.435%	12.37%	4 707	6.4
	1.30	60	61	43.0%	0.611%	16.97%	3 943	5.0
	1.50	60	61	43.7%	0.679%	20.73%	3 786	4.8
	2.00	60	61	43.7%	0.682%	27.61%	3 781	4.8

Table 3. CPU time required to update the core in case of traffic jams for different contraction parameters. The length of shortcuts is limited to 20 minutes of travel time (10 minutes for the values in parentheses).

cont.		space		single traffic jam				batch update (1 000 jams)				query	
c	h	[B/n]		av.[ms]		max[ms]		av.[ms]		max[ms]		time [ms]	
0.0	0	256	(256)	0	(0)	0	(0)	0	(0)	0	(0)	188.2	(188.2)
0.5	10	100	(103)	1	(1)	49	(49)	820	(619)	1200	(799)	76.8	(85.2)
1.0	20	45	(50)	37	(21)	2231	(778)	30787	(20329)	39470	(22734)	22.8	(27.1)
2.0	30	51	(56)	220	(90)	5073	(3868)	187595	(79092)	206569	(85259)	16.4	(22.8)

magnitude, and the improvement over Dijkstra's algorithm is of a factor ≈ 50. Comparing TDCALT to SHARC, we see that for exact queries SHARC yields better query times by a factor ≈ 1.6, although preprocessing time and space for SHARC are larger. However, SHARC cannot efficiently deal with dynamic scenarios. If we can accept a maximum approximation factor $K \geq 1.05$ then TDCALT is faster than SHARC, by one order of magnitude for $K \geq 1.20$. The size of the search space decreases by even larger factors, but in terms of time spent per node SHARC is faster than TDCALT, as we observed in [1].

Dynamic Updates. In order to evaluate the performance of the core update procedure (see Section 4) we generated several traffic jams as follows: for each traffic jam, we select a path in the network covering 4 minutes of uncongested travel time on motorways. Then we randomly select a breakpoint between 6AM and 9 PM, and for all edges on the path we multiply the corresponding breakpoint value by a factor 5. As also observed in [12], updates on motorway edges are the most difficult to deal with, since those edges contribute to a large number of shortcuts. In Table 3 we report average and maximum required time over 1 000 runs to update the core in case of a single traffic jam, applying different contraction parameters. Moreover, we report the corresponding figures for a batch-update of 1000 traffic jams (100 runs), in order to reduce the fluctuations and give a clearer indication of required CPU time when performing multiple updates. Note that for this experiment we limit the length of shortcuts to 20 minutes (10 for the values in parentheses) of uncongested travel time. This is because in the dynamic scenario the length of shortcuts plays the most important role when determining the required CPU effort for an update operation, and if we allow the shortcuts length to grow indefinitely we may have unpractical update times. Hence, we also report query times with $K = 1.15$.

As expected, the effort to update the core becomes more expensive with increasing contraction parameters. However, for $c = 1.0$, $h = 20$ with maximum shortcut length of 20 minutes, we have reasonable update times together with query times of 22.8 ms: an update of 1 000 traffic jams can be done in less than 40 seconds, which should be sufficient in most applications. In most cases, the required time to update the core for a single traffic jam is of a few milliseconds, and query times are fast even with limited length shortcuts. We observe a clear trade off between query times and update times depending on the contraction parameters, so that for those applications which require frequent updates we can minimize update costs while keeping query times < 100 ms, and for applications which require very few or no updates we can minimize query times. If most of the graph's edges have their cost changed we can rerun the core edges computation, which takes less than 15 minutes.

6 Conclusion

We have proposed a bidirectional ALT algorithm for time-dependent graphs which uses a hierarchical approach: the bidirectional search starts on the full graph, but is soon restricted to a smaller network in order to reduce the number of explored nodes. This algorithm is flexible and allows us to deal with the dynamic scenario, where the piecewise linear time-dependent cost functions on arcs are not fixed, but can have their coefficients updated. Extensive computational experiments show a significant improvement over existing time-dependent algorithms, with query times reduced by at least an order of magnitude in almost all scenarios, and a faster and less space consuming preprocessing phase. Updates in the cost functions are dealt with in a practically efficient way, so that traffic jams can be added in a few milliseconds, and we can parameterize the preprocessing phase in order to balance the trade off between query speed and update speed.

References

1. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A* search for time-dependent fast paths. In: [22], pp. 334–346
2. Wagner, D., Willhalm, T.: Speed-up techniques for shortest-path computations. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 23–36. Springer, Heidelberg (2007)

3. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. In: [22], pp. 303–318
4. Cooke, K., Halsey, E.: The shortest route through a network with time-dependent internodal transit times. J. of Math. Analysis and Applications 14, 493–498 (1966)
5. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
6. Dreyfus, S.: An appraisal of some shortest-path algorithms. Operations Research 17(3), 395–412 (1969)
7. Kaufman, D.E., Smith, R.L.: Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. Journal of Intelligent Transportation Systems 1(1), 1–11 (1993)
8. Orda, A., Rom, R.: Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. Journal of the ACM 37(3), 607–625 (1990)
9. Hart, E., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems, Science and Cybernetics SSC 4(2), 100–107 (1968)
10. Goldberg, A., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: Proceedings of SODA 2005, pp. 156–165. SIAM, Philadelphia (2005)
11. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Proceedings of ALENEX 2006. LNCS, pp. 129–143. Springer, Heidelberg (2006)
12. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: [21], pp. 52–65
13. Chabini, I., Lan, S.: Adaptations of the A^* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks. IEEE Transactions on Intelligent Transportation Systems 3(1), 60–74 (2002)
14. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In: Proceedings of the ALENEX 2008, pp. 13–26. SIAM, Philadelphia (2008)
15. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up dijkstra's algorithm. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 189–202. Springer, Heidelberg (2005)
16. Delling, D.: Time-Dependent SHARC-Routing. In: Halperin, D., Mehlhorn, K. (eds.) Esa 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
17. Goldberg, A., Werneck, R.: Computing point-to-point shortest paths from external memory. In: Demetrescu, C., Sedgewick, R., Tamassia, R. (eds.) Proceedings of ALENEX 2005, pp. 26–40. SIAM, Philadelphia (2005)
18. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: [22], pp. 319–333
19. Goldberg, A., Kaplan, H., Werneck, R.: Better landmarks within reach. In: [21], pp. 38–51
20. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: Shortest Paths: Ninth DIMACS Implementation Challenge. DIMACS Book. American Mathematical Society (to appear, 2008)
21. Demetrescu, C. (ed.): WEA 2007. LNCS, vol. 4525. Springer, Heidelberg (2007)
22. McGeoch, C.C. (ed.): WEA 2008. LNCS, vol. 5038. Springer, Heidelberg (2008)