Core Routing on Dynamic Time-Dependent Road Networks*

Daniel Delling
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
Email: delling@ira.uka.de

Giacomo Nannicini
LIX, École Polytechnique, F-91128 Palaiseau, France
Email: giacomon@lix.polytechnique.fr

October 28, 2010

Abstract

Route planning in large scale time-dependent road networks is an important practical application of the shortest paths problem that greatly benefits from speed-up techniques. In this paper we extend a two-level hierarchical approach for point-to-point shortest paths computations to the time-dependent case. This method, also known as *core routing* in the literature for static graphs, consists in the selection of a small subnetwork where most of the computations can be carried out, thus reducing the search space. We combine this approach with bidirectional goal-directed search in order to obtain an algorithm capable of finding shortest paths in a matter of milliseconds on continental sized networks. Moreover, we tackle the dynamic scenario where the piecewise linear functions that we use to model time-dependent arc costs are not fixed, but can have their coefficients updated requiring only a small computational effort.

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 [10] for a recent overview. Adaptations of those ideas to dynamic scenarios, i.e., where arc costs are updated at regular intervals, have been successfully tested as well [12, 32].

Much of the focus is now moving to the Time-Dependent Shortest Path Problem (TDSPP), which consists in the SPP applied on a time-dependent network. There are several industrial applications for the TDSPP that require very fast computational times, e.g., a web server which has to answer hundreds of shortest path queries per second. The TDSPP has been first addressed by [4] with a recursion formula; Dijkstra's algorithm

*The first author is currently affiliated with Microsoft Research, Mountain View, CA, USA (dadelling@microsoft.com), while the second author is currently affiliated with Tepper School of Business, Carnegie Mellon University, PA, USA (nannicin@andrew.cmu.edu).

[14] is then extended to the time-dependent case in [15], but the FIFO property, which is necessary to guarantee correctness, is implicitly assumed. The FIFO property, also called the *non-overtaking property*, 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 [24], while it is NP-hard in non-FIFO networks [29]. We focus on the FIFO variant.

Although the TDSPP can be solved by Dijkstra's algorithm on FIFO networks, its application may require several seconds for each shortest path computation on large graphs, thus it may not be suitable for real-time applications. Hierarchical speed-up techniques have been successfully used for the SPP in the static (i.e. non time-dependent) case [10], hence in this paper we generalize these techniques to the time-dependent scenario and analyse the performance of a two-level hierarchical setup (*core routing*) for the TDSPP. The idea behind core routing 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. We combine core routing with the bidirectional goal-directed algorithm that we have recently proposed in [28], improving its query speed by an order of magnitude and reducing preprocessing time and space. Moreover, we tackle the dynamic scenario, where we allow the piecewise linear cost functions that model the arc costs to change their coefficients. We propose an algorithmic framework to efficiently update all data structures needed to restore optimality of the core when the cost functions are updated. An extended abstract of this work appeared in [9].

1.1 Overview

This paper is organized as follows. In Section 2 we describe core routing on static graphs and generalize it to the time-dependent case; we also introduce and discuss the drawbacks of a multi-level hierarchical approach. In Section 3 we give details on the ingredients which are necessary to implement the algorithm in practice, such as the contraction routine. In Section 4 we discuss the dynamic scenario. In Section 5 we provide a detailed experimental evaluation of our method, and analyse the results.

In the rest of this section we introduce our notation, give a formal definition of the problem that we address, and review existing related works.

1.2 Definitions and Notation

Consider an interval $\mathcal{T} = [0, P] \subset \mathbb{R}$ and a function space $\mathbb{F}$ of positive functions $f : \mathbb{R}^+ \to \mathbb{R}^+$ with the property that $\forall \tau > P\ f(\tau) = f(\tau - kP)$, where $k = \max\{k \in \mathbb{N} | \tau - kP \in \mathcal{T}\}$. This implies $f(\tau + P) = f(\tau)\ \forall \tau \in \mathcal{T}$; in other words, f is periodic of period P. We additionally require that $f(x) + x \leq f(y) + y\ \forall f \in \mathbb{F}, x, y \in \mathbb{R}^+, x \leq y$; this ensures that our network respects the FIFO property when the functions are interpreted as travel times [24]. The juxtaposition $f \oplus g$ of two functions $f, g \in \mathbb{F}$ is a function in $\mathbb{F}$ defined as $(f \oplus g)(\tau) = f(\tau) + g(f(\tau) + \tau)\ \forall \tau \in \mathbb{R}^+$. Note that this operation is neither commutative nor associative, and should be evaluated from left to right; that is, $f \oplus g \oplus h = (f \oplus g) \oplus h$. The minimum $\min\{f, g\}$ of two functions $f, g \in \mathbb{F}$ is a function in $\mathbb{F}$ such that $\min\{f, g\}(\tau) = \min\{f(\tau), g(\tau)\}\ \forall \tau \in \mathcal{T}$, i.e., a pointwise minimum. We define the lower bound of f as $\underline{f} = \min_{\tau \in \mathcal{T}} f(\tau)$, and the upper bound as $\overline{f} = \max_{\tau \in \mathcal{T}} f(\tau)$.

Consider a directed graph $G = (V, A)$, where the cost of an arc (u, v) is a time-dependent function given by a function $c : A \to \mathbb{F}$; for simplicity, we will write $c(u, v, \tau)$ instead of $c(u, v)(\tau)$ to denote the cost of the arc (u, v) at time $\tau \in \mathcal{T}$. We define $\lambda, \rho : A \to \mathbb{R}^+$ as $\lambda = \underline{c}$ and $\rho = \overline{c}$, i.e., $\forall (u, v) \in A\ \lambda(u, v) = \underline{c}(u, v)$ and $\rho(u, v) = \overline{c}(u, v)$.

We denote the distance between two nodes $s, t \in V$ with departure from s at time $\tau_0 \in \mathcal{T}$ as $d(s, t, \tau_0)$. The distance function between s and t is defined as $d_*(s, t) : \mathcal{T} \to \mathbb{R}^+, d_*(s, t)(\tau) = d(s, t, \tau)$. We denote by G_λ the graph G weighted by the lower bounding function λ; the distance between two nodes s, t on G_λ is denoted by $d_\lambda(s, t)$.

Given a path $p = (s = v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k = t)$, its *time-dependent cost* is defined as $\gamma(p) = c(v_1, v_2) \oplus c(v_2, v_3) \oplus \cdots \oplus c(v_{k-1}, v_k)$. Its time-dependent cost with departure time at $\tau_0 \in \mathcal{T}$ is denoted as $\gamma(p, \tau_0) = \gamma(p)(\tau_0)$. We denote the subpath of p from v_i to v_j by $p_{|v_i \to v_j}$. The concatenation of two paths p and q is denoted by $p + q$.

For $V' \subset V$, we define $A[V'] = \{(u, v) \in A | u \in V', v \in V'\}$ as the set of arcs with both endpoints in V'. Correspondingly, the subgraph of G induced by V' is $G[V'] = (V', A[V'])$. We define the *union* between two graphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$ as $G_1 \cup G_2 = (V_1 \cup V_2, A_1 \cup A_2)$.

We can now formally state the Time-Dependent Shortest Path Problem:

> Time-Dependent Shortest Path Problem(TDSPP): given a directed graph $G = (V, A)$ with cost function $c : A \to \mathbb{F}$ as defined above, a source node $s \in V$, a destination node $t \in V$ and a departure time $\tau_0 \in \mathcal{T}$, find a path $p = (s = v_1, \ldots, v_k = t)$ in G such that its *time-dependent cost* $\gamma(p, \tau_0)$ is minimum.

We will assume that our problem is to find the fastest path between two nodes with departure at a given time; the "backward" version of this problem, i.e., finding the fastest path between two nodes with *arrival* at a given time, can be solved with the same method (see [6]).

1.3 Label-Correcting Algorithm

The well known Dijkstra's algorithm [14] can be applied in a straightforward manner on a time-dependent FIFO graph to compute $d(s, t, \tau)$ for any two nodes $s, t \in V$ and departure time $\tau \in \mathcal{T}$. Dijkstra's algorithm is a *label-setting* algorithm, because whenever a node is extracted from the priority queue its label is permanently set. The distance function between two nodes $d_*(s, t)$ can be computed with a *label-correcting* algorithm; label-correcting implies that the label of a node is not fixed even after the node is extracted from the priority queue, in that a node may be reinserted multiple times, unlike label-setting algorithms. We refer to [7] for an excellent starting point on the efficient implementation of TDSPP algorithms. We describe here a label-correcting algorithm to compute the cost function associated with the shortest path between two nodes $s, t \in V$. Such an algorithm can be implemented similarly to Dijkstra's algorithm, but using arc *cost functions* instead of arc *lengths*. The label $\ell(v)$ of a node v is a scalar for plain Dijkstra's algorithm, whereas in this case each label is a function of time. In particular, at termination we want $\ell(v) = d_*(s, v)$. We initialize the algorithm assigning constant functions as labels: $\forall \tau \in \mathcal{T}\ \ell(s)(\tau) = 0$ and $\ell(v)(\tau) = \infty\ \forall v \in V$. At each iteration we extract the node u with minimum $\underline{\ell}(u)$ from the priority queue, and relax adjacent edges: for each $(u, v) \in A$, a temporary label $\ell'(v) = \ell(u) \oplus c(u, v)$ is created. Then if $\ell'(v)(\tau) \geq \ell(v)(\tau)$ for all $\tau \in \mathcal{T}$ does not hold, the arc (u, v) yields an improvement for at least one time instant. Hence, in this case we update $\ell(v) = \min\{\ell(v), \ell'(v)\}$, and v is inserted back into the priority queue. The algorithm can be stopped as soon as we extract a node u such that $\underline{\ell}(u) \geq \overline{\ell}(t)$, and $\ell(t)$ yields the solution. An interesting observation from [7] is that the running time of this algorithm depends on the complexity of the cost functions associated with arcs: in the case of piecewise linear functions f with k breakpoints x_i, (i.e. $f(x) =$

$f(x_i) + (f(x_{(i+1) \bmod k}) - f(x_i))(x - x_i)/(x_{(i+1) \bmod k} - x_i)\ \forall x \in [x_i, x_{i+i}], i = 1, \ldots, k)$, the number k has a strong impact on the running time.

[Removed part on SHARC]

1.4 A^* with Landmarks

A^* is an algorithm for goal-directed search which is very similar to Dijkstra's algorithm [22]. 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. Thus, the key of v represents an estimation of the length of the shortest path from s to t passing through v, and nodes are sorted in the priority queue according to this criterion. 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 the target node t. If the potential function is such that $\pi(u) - \pi(v) \leq c(u, v)\ \forall (u, v) \in A$ and $\pi(t) \leq 0$, then A^* computes shortest paths [22] and $\pi(v) \leq d(v, t)\ \forall v \in V$. A^* is equivalent to Dijkstra's algorithm on a graph where arc costs are the reduced costs $w_\pi(u, v) = c(u, v) - \pi(u) + \pi(v)$ [23]. From this, it can be easily seen that if $\pi(v) = 0\ \forall v \in V$ then A^* explores exactly the same nodes as Dijkstra's algorithm, whereas if $\pi(v) = d(v, t)\ \forall v \in V$ only nodes on the shortest path between s and t are settled, as arcs on the shortest path have zero reduced cost. 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* [17]. The main idea is to select a small set of nodes in the graph, sufficiently spread over the whole network, 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 [17]. Bidirectional search can be applied: a forward search is started on G from the source using a potential function π_f which estimates the distance to reach the target, and a backward search is started from the destination using a potential function π_b which estimates the distance from the source. The two potential function must be consistent [18], which means that $w_{\pi_f}(u, v)$ on G is equal to $w_{\pi_b}(v, u)$ on the reverse graph on which the backward search is run. This translates to $\pi_f(v) + \pi_b(v) = \kappa\ \forall v \in V$ for some constant κ.

Bidirectional A^* with the potential function described above is called ALT; an experimental evaluation on static graphs can be found in [18]. Landmark potentials are valid even if arc costs are modified, as long as the new costs are larger than the ones used to compute distances to and from landmarks. This follows immediately from the fact that the landmark potentials must be lower bounds. Therefore, we can use this approach on time-dependent graphs, taking care to use lower bounds to the time-dependent arc costs during the preprocessing phase. 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. The size of the search space greatly depends on how landmarks are positioned over the graph, as it severely affects the quality of the potential func-

tion. Several heuristic selection strategies have been proposed; there is usually a trade off between preprocessing time and quality of the landmark choice. So far no optimal strategy with respect to random queries has been found, i.e., no strategy guarantees to yield the smallest search spaces with respect to shortest path computations where source and destination nodes are chosen at random. Commonly used selection criteria are *avoid* and *maxCover* [20].

Bidirectional search cannot be directly applied on time-dependent graphs, the optimal arrival time at the destination being unknown. In a recent work [28], we proposed a bidirectional ALT algorithm on time-dependent road networks that overcomes this problem. 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 run on G_λ.

Given a graph $G = (V, A)$ and source and destination vertices $s, t \in V$, the algorithm for computing the shortest time-dependent cost path p^* works in three phases.

1. A bidirectional A^* search occurs on G, where the forward search is a time-dependent search with cost function c, and the backward search is run on G_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 $\mu = \gamma(p_v, \tau_0)$ of the path p_v going from s to t passing through v is an upper bound to $\gamma(p^*, \tau_0)$. In the second phase, both search scopes are allowed to proceed until the backward search queue only contains nodes whose associated key exceeds μ. In other words: 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); see [28] for the description of several improvements that have a significant effect on performance. In particular, we use a better (i.e., tighter) potential function π_b^* for the backward search. Note that, for each $K > 1$, if we switch from phase 2 to phase 3 as soon as $\beta > K\mu$, then the algorithm computes a K-approximated solution. A proof is given in [28], where it is also shown that in practice, using $1 < K \leq 1.15$ yields significant speed-ups while deteriorating the solution quality by marginal amounts on average. The upper bound μ is updated during the search in phase 2, whenever the backward search settles a node which is also settled by the forward search. In our computational tests, there is no clear advantage in updating the upper bound in phase 3 (we remark that each update requires some CPU time, because we have to traverse a path to t while keeping track of its time-dependent cost).

2 Time-Dependent Core-Based Routing

Core-based routing is a powerful approach which has been widely and successfully used for shortest paths algorithms on static graphs [3]. The main idea is to use contraction [16]: a routine iteratively removes unimportant 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 $G_F = G_C \cup G = (V, A \cup A_C)$ since $V_C \subset V$.

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 $\tau_0 \in \mathcal{T}$ 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. Alternate 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 that are in G_C and the backward search queue with all leaves of T that are in G_C, 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. The forward search does not relax edges (u, v) such that $u \in T$ and $v \notin T$. 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 $v \in T \cap V_C$. The key observation is that, by the FIFO property, we are allowed to consider only the earliest possible arrival time at each explored node; thus, we can switch to the main phase initializing the priority queue with the labels computed in the first phase (which represent earliest arrival times). Again, since Dijkstra's algorithm is equivalent to A^* with a zero potential function, we could also use Dijkstra's algorithm instead of TDALT in case (ii) during the main phase. Since TDALT is more efficient in practice than Dijkstra's algorithm, we did not test this approach.

2.1 Proposition
The core routing algorithm for time-dependent graphs computes the optimal time-dependent $s - t$ path.

Proof. In case (i), i.e., $S \cap T \neq \emptyset$, the proof is trivial: we switch to unidirectional Dijkstra's algorithm on the original graph (plus added shortcuts, which preserve distances), that terminates with the shortest $s - t$ path on a FIFO network.

In case (ii), the two priority queues are empty and $S \cap T = \emptyset$, thus the shortest path p between s and t with departure time τ_0 passes through at least one node belonging to the core V_C. Let $p = (s, \ldots, u, \ldots, v, \ldots, t)$, where u and v are, respectively, the first and the last node in V_C on the path. If $u = v$, all nodes on $p_{|v \to t}$ can be explored by the forward search during the algorithm, therefore the shortest $s - t$ path is eventually found. Now suppose $u \neq v$. Since the initialization phase explores all non-core nodes reachable from s and t, $u \in S$ and $v \in T$. By definition of v, $p_{|v \to t}$ passes only through

non-core nodes; by the query algorithm, T contains all non-core nodes that can reach t passing only through non-core nodes. It follows that all nodes of $p_{|v \to t}$ are in T. Thus $p_{|u \to t}$ is entirely contained in $G_C \cup G[T] = (V_C \cup T, A_C \cup A[T])$. The subpath $p_{u \to v}$ can be computed using core arcs only by construction of the core, so there is no need to relax edges (w, w') such that $w \in T$ and $w' \notin T$. By correctness of Dijkstra's algorithm, the distance labels for nodes in S represent the earliest possible arrival times, which are the only time instants that have to be considered to find the optimal path on a FIFO network. Initializing the forward search queue with the leaves of S and applying TDALT (that is, A^*) on $G_C \cup G[T]$ then yields the shortest path p by correctness of A^* on FIFO networks. □

2.1 Potential Function for Core Routing

In a typical core routing setting for the ALT algorithm, landmark distances are computed and stored only for vertices in V_C (see [3]), since the initialization phase on non-core nodes uses Dijkstra's algorithm only. This means that the landmark potential function cannot be used to apply the forward A^* search on the nodes in T. However, in order to combine TDALT with a core routing framework we can use the backward distance labels computed with Dijkstra's algorithm during the initialization phase.

2.2 Proposition
The potential function π_f^* that uses the distances on G_λ computed by the backward Dijkstra search for nodes in T, and the landmark potential function for the remaining nodes, is feasible for the forward search.

Proof. We need to show that $\pi_f^*(u) \leq \lambda(u, v) + \pi_f^*(v)$ for each (u, v) in A. For any vertex $w \in T$, let $d(w)$ be the distance label computed by the reverse Dijkstra search from t on G_λ during the initialization phase of the algorithm. We first examine the case $v \notin T$. If $u \notin T$, the proof is trivial since the landmark potential function π_f is feasible. The case $u \in T$ cannot happen because the arc would not be relaxed by the query algorithm (which is equivalent to saying that these arcs do not appear in the graph with reduced costs where the A^* search is carried out). Now examine the case $v \in T$. If $u \notin T$, we have $\pi_f^*(u) \leq d_\lambda(u, t) \leq \lambda(u, v) + d_\lambda(v, t) \leq \lambda(u, v) + d(v) = \lambda(u, v) + \pi_f^*(v)$. Finally, if $u \in T$, arc (u, v) has been relaxed by the backward Dijkstra search, therefore $\pi_f^*(u) = d(u) \leq \lambda(u, v) + d(v) = \lambda(u, v) + \pi_f^*(v)$. □

Thus, we have a way to compute potentials for all nodes in $V_C \cup T$. For the remaining nodes, we can use proxies (which are discussed in Section 3.1). We call the algorithm described in this section Time-Dependent Core-based ALT (TDCALT).

2.2 Multilevel Hierarchy

The two-level query algorithm can be generalized to a multilevel hierarchy as follows. Let $G_C^l = (V_C^l, A_C^l)$ for $l = 0, \ldots, L$ be a hierarchy of graphs such that $V_C^{l+1} \subset V_C^l$ $\forall l = 0, \ldots, L-1$ and $G_C^0 = G$. We assume that $\forall l = 0, \ldots, L, \forall u, v \in V_C^l$, for each departure time $\tau_0 \in \mathcal{T}$ there is a shortest path between u and v in G_C^l with the same cost as the shortest path between u and v in G with the same departure time. A path can be computed with the following algorithm (correctness can be proved almost identically to Proposition 2.1).

1. Initialization: set $l = 0, S = \{s\}, T = \{t\}$.
2. Level selection phase: start a Dijkstra search from both the source and the destination node on G_C^l, initializing the forward search queue with all leaves of S in G_C^l and the backward search queue with all leaves of T in G_C^l, using the time-dependent costs for the forward search and the time-independent costs λ for the backward search. The search must be pruned (i.e., outgoing arcs should not be relaxed) at nodes $\in V_C^{l+1}$. Add each node settled by the forward search to S, and each node settled by the backward search to T. Iterate between the two searches until: (i) $S \cap T \neq \emptyset$ or (ii) the priority queues are empty.
3. Main phase:
   - (i) If $S \cap T \neq \emptyset$, then start an unidirectional Dijkstra search from the source on G until the target is settled.
   - (ii) If the priority queues are empty with $S \cap T = \emptyset$, then if $l < L$, set $l = l + 1$ and return to 2. Otherwise, start TDALT on the graph G_C^L, initializing the forward search queue with all leaves of S in G_C^L and the backward search queue with all leaves of T in G_C^L, 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. The forward search does not relax edges (u, v) such that $u \in T$ and $v \notin T$. Stop when t is settled by the foward search.

In static graphs, multilevel hierarchical methods have shown very good results in practice [3]. However, this does not seem to be true for the time-dependent case. Computational experiments (see Section 5) indicate that the complexity of shortcuts grows rapidly if we apply a strong contraction to the original graph, which yields larger space consumption and slower dynamic updates (see Section 5.4). Besides, it is not clear whether more levels in the hierarchy bring an advantage in terms of reduced query times, since the possibility that the forward and backward search scopes meet before reaching the topmost level increases. [1] describes a multilevel approach, but the resulting space consumption is very high (more than 1 000 bytes per node) making the approach not practical for real-world applications. For all these reasons, it does not seem a good idea in practice to use a multilevel setup, thus we decided to test only a two-level hierarchy in the following.

3 Practical Issues

There are still several missing pieces before a full description of a practical implementation of the algorithm described in Section 2 can be given. Namely, we should describe a way to compute the potential function for nodes in $V \setminus (V_C \cup T)$, discuss the contraction routing, and give an algorithm to retrieve the full shortest path on the original graph when the computations are done on the contracted graph.

3.1 Proxy Nodes

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]. See also [21] for a description. We here report 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 $d(s',s)$.

3.2 Contraction

For the contraction phase, i.e., the routine that selects which nodes have to be bypassed and then adds shortcuts to preserve distances between remaining nodes, we use the algorithm proposed in [8]. 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. Note that, in the case of piecewise linear cost functions, the composition of two functions yields an increase in the number of breakpoints; indeed, if two functions $f, g \in \mathbb{F}$ have, respectively, $B(f)$ and $B(g)$ breakpoints, then the composition $f \oplus g$ may have up to $B(f) + B(g)$ breakpoints in the worst case (see [7] for details). As these points have to be stored in memory, the space consumption may become unpractical if we add too many long shortcuts. Thus, we also enforce a limit I on the maximum number of breakpoints that each shortcut may have: if a shortcut which exceeds this limit would be created, we simply do *not* bypass the corresponding node.

In order to choose which node has to be bypassed at each step of the contraction routine, we maintain a heap of all nodes sorted by a function value (*bypassability score*) which favours nodes whose bypassing creates fewer and shorter shortcuts, and extract the minimum element at each iteration. The bypassability score of a node u is defined as a linear combination of: the expansion of u, the hop-number of the longest shortcut that would be created if u is bypassed, and the largest number of breakpoints of the shortcuts that would be created if u is bypassed. Note that the contraction of u may influence the bypassability score of adjacent nodes, so these scores must be recomputed after a node is chosen. As suggested by [8], we give a larger importance to the expansion of a node when determining its bypassability score, thus using a coefficient of 10 for this factor in the linear combination, whereas the other two factors are added with unitary coefficient. This heuristic is motivated by the good results obtained in [8]; experiments in [16] show that all "reasonable" heuristics to determine the bypassability score perform well in practice. At the end of the contraction routine, we perform an edge-reduction step which removes unnecessary shortcuts from the graph. In particular, for each node of the core $u \in V_C$ we check whether for each arc $(u,v) \in A_C$ and for each $\tau \in \mathcal{T}$ there is a path p from u to v which does not use the arc (u,v) and such that $\gamma_\tau(p) < c(u,v,\tau)$. This step can be performed by computing the cost function $d_*(u,v)$ on the graph $(V_C, A_C \setminus \{(u,v)\})$ with a label-correcting algorithm (Section 1.3) and comparing $d_*(u,v)$ with $c(u,v)$. If $d_*(u,v)(\tau) < c(u,v,\tau)\ \forall \tau \in \mathcal{T}$, then the arc (u,v) is not necessary, as there is a shorter path between u and v for all possible departure times (see also [8]). Whenever a shortcut between two nodes $u, v \in V$ is added, its cost for each time instant of the time interval is computed running a label-correcting algorithm between u and v (Section 1.3).

3.3 Outputting Shortest Paths

Shortcuts are added to the graph in order to accelerate queries. However, as for all shortcut-based speed-up techniques, those shortcuts have to be expanded if we want to retrieve the complete shortest path and not only the distance. Our contraction routine for time-dependent graphs is an augmented version of the one introduced for Highway Hierarchies [31]. In [11], an efficient unpacking routine based on storing all the arcs a shortcut represents is introduced: since arc identifiers may be several bytes long, for each arc (u, v) on the path that the shortcut represents we store the difference between its index and the index of the first outgoing arc of u. As the outgoing arcs of each node are stored contiguously in memory for obvious spatial locality reasons, and the outdegree of nodes is typically small, this difference can be represented in a small number of bits. In our experiments on road networks, 4 bits were always sufficient to store this difference.

However, in the static case a shortcut represents exactly one path because between any two nodes we only need to keep track of the *shortest* arc that connects them, whereas in the time-dependent case the shortest arc between two nodes may be different for each different traversal time. 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. Thus, multiple shortcuts between the same endpoints are not merged. With this modification each shortcut represents exactly one path, so we can directly apply the unpacking routine from [11]. In our experimental evaluation the additional computational time to output a full representation of the shortest path is ≈ 1 millisecond. Section 5.2 reports the total space occupation for the additional data required by this routine. Note that, if we are not interested in obtaining the full representation of the shortest path (i.e. we only want to compute the optimal cost), then multi-edges are not needed.

4 Dynamic Cost Updates

Modifications in the cost functions can be easily taken into account under weak assumptions if shortcuts have not been added to the graph. However, a two-level hierarchical setup is significantly more difficult to deal with, exactly because of shortcuts: since a shortcut represents the shortest path between its two endpoints for at least one departure time, if some arc costs change then the shortest path which is represented may also be subject to changes. Thus, a procedure to restore optimality of the core is needed. We first analyse the general case for modifications in the breakpoint values; then we focus on the simpler case of increasing breakpoint values, and finally propose an algorithmic framework to deal with general cost changes under some restrictive assumptions which are acceptable in practice. Even though we do not explicitly deal with addition/removal of breakpoints, these are theoretically not much more difficult to handle than modifications in the breakpoint values. What determines the difficulty of handling updates is whether the new cost function is pointwise greater or smaller than the initial cost function, as will be seen in the following. This remains true for addition/removal of breakpoints.

4.1 Analysis of the General Case

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 then 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. This requires a significant computational effort, and should be avoided if we want to perform fast updates for real-time applications. In the second case, then the cost function for core arcs 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 arcs, thus we can restrict the update only to the shortcuts that contain the arc. We now analyse this case in further detail.

4.2 Increases in Breakpoint Values

To perform fast updates in the case that breakpoint values increase, 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 $a'_0, \ldots, a'_d$, 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 $\rho_j = \sum_{i=0}^{j-1} \rho(a'_i)$. 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 time from the starting point of a' is in the interval $[\tau_0 - \rho_j, \tau_n - \lambda_j]$. This is because a can be reached from the starting node of a' no sooner than λ_j, and no later than ρ_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 $[\tau_0 - \rho_j, \tau_n - \lambda_j]$, 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.

4.3 A Realistic Scenario

The procedure described in Section 4.2 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.

4.1 Example
Given $a \in A$, suppose that the cost of its breakpoint at time $\tau \in \mathcal{T}$ increases, and all shortcuts $\in S(a)$ are updated. Suppose that, for a shortcut $a' \in S(a)$, a does not contribute 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.

Note that we could set all breakpoint values on a time-dependent arc (u, v) to be equal to $\lambda(u, v)$ during the preprocessing phase, in which case we could apply our fast update routine for any change in the breakpoint values (as we assumed that the lower bound given by λ is always valid). There are two drawbacks for this extreme scenario: first, the sets $S(a)$ would get larger, therefore each update would require more time; second, query times would be longer, because there would be more temporary shortcuts to explore during the search. In our computational experiments, the shortest path query algorithm is only marginally slowed if more temporary shortcuts are added. There is a tradeoff between better performance of the algorithm (if we set breakpoint values during the preprocessing phase equal to the initial piecewise linear time-dependent arc cost functions) and freedom in the updates of the arc cost functions (if we set all breakpoint values equal to a lower bound on arc costs), such that all intermediate choices between the two extremes are also possible, depending on the application. In the rest of this paper, we assume that the application is such that we are allowed to take breakpoint values equal to the initial piecewise linear time-dependent cost function during the preprocessing phase.

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 the TDCALT algorithm. Our implementation is written in C++ using the Standard Template Library. 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 [17], computed on the core of the input graph using the lower bounding function λ as edge weights, and the improvements discussed in [28]. We do not use the slightly better *maxCover* heuristic for choosing landmarks, because preprocessing times increase while speed of the shortest paths computations is unaffected (in our scenario).

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. Note that the choice of τ_0 has limited impact on query performance since the average length of a random query is several hours. Hence, we encounter traffic jams with almost all 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 (see Section 3.3) 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 two different road networks: the road network of Western Europe, which has approximately 18 million vertices and 42.6 million arcs, and the road network of Germany (4.7 million nodes and 10.8 million edges).

Our German data contains five different realistic traffic scenarios, generated from traffic simulations: Monday, midweek (Tuesday till Thursday), Friday, Saturday, and Sunday. As expected, congestion of roads is higher during the week than on the weekend: $\approx 8\%$ of edges are time-dependent for Monday, midweek, and Friday. The corresponding figures for Saturday and Sunday are $\approx 5\%$ and $\approx 3\%$, respectively. All data has been provided by PTV AG for scientific use.

Unfortunately, we are not aware of a *continental-sized* publicly available real-world road network with time-dependent arc costs. Hence, we decided to use the time-independent road network of Europe and artificially generate time-dependent 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 [28] and ensures spatial coherency for traffic jams; we used piecewise linear cost functions, with one breakpoint for each hour over a day. More breakpoints may be required for shortcuts, since the number of breakpoints can increase. The travelling time of an arc at time τ is computed via linear interpolation of the two breakpoints that precede and follow τ. The breakpoints are stored in an additional array, ordered by the edges they are assigned to. Similarly to an adjacency array graph data structure [5], each arc has a pointer to the first of its assigned breakpoints.

5.1 Contraction Rates

Table 1 shows the performance of TDCALT for different contraction parameters (cf. Section 3), using our European network as input. In this setup, we fix the approximation value K (cf. Section 2) to 1.15, which was found to be a good compromise between speed and quality of computed paths (see [28]). 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. The contraction parameters $C = 0.0$ and $H = 0$ yield a pure TDALT setup: the core of the graph is empty.

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

Table 1: Performance of TDCALT for different contraction rates, using Europe as input. 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 breakpoints of the merged graph compared to the original input.

Core			Preprocessing				Exact Query		Approx. Query ($K = 1.15$)				
param.		core	time	space	increase in		#settled	time	error	relative error		#settled	time
C	H	nodes	[min]	[B/n]	#edges	#points	nodes	[ms]	-rate	avg.	max	nodes	[ms]
0.0	0	100.0%	28	256	0.0%	0.0%	2 931 080	2 939.3	40.1%	0.303%	10.95%	250 248	188.2
0.5	10	35.6%	15	99	9.8%	21.1%	1 165 840	1 224.8	38.7%	0.302%	11.14%	99 622	78.2
1.0	20	6.9%	18	41	12.6%	69.6%	233 788	320.5	34.7%	0.288%	10.52%	19 719	21.7
2.0	30	3.2%	30	45	9.9%	114.1%	108 306	180.0	34.9%	0.287%	10.52%	9 974	13.2
2.5	40	2.5%	39	50	9.1%	138.0%	84 119	149.7	34.1%	0.275%	8.74%	8 093	11.4
3.0	50	2.0%	50	56	8.7%	161.2%	70 348	133.2	32.8%	0.267%	9.58%	7 090	10.3
3.5	60	1.8%	60	61	8.5%	181.1%	60 636	122.3	33.8%	0.280%	8.69%	6 227	9.2
4.0	70	1.5%	88	74	8.5%	223.1%	52 908	115.2	32.8%	0.265%	8.69%	5 896	8.8
5.0	100	1.2%	134	89	8.6%	273.5%	45 020	110.6	32.6%	0.266%	8.69%	5 812	8.4

core shrinks very quickly, hence we store landmark distances only for 6.9% of the nodes. On the other hand, the number of breakpoints 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 if we allow more and longer shortcuts to be built, then the error rate decreases, as well as the maximum and average relative error. We believe that this is due to a combination of factors. First, 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. Second, the core becomes very small for large contraction parameters. This increases the chance that the subpath of the shortest path which passes through the core has a small number of arcs (possibly, only one); as shortcuts represent optimal distances, the chance of computing a suboptimal path decreases. Summarizing, large contraction parameters require more preprocessing time and space, but yield better results in terms of size of the search space and query speed. On the other hand, experiments on the dynamic cost updates (Section 5.4) show that the length of shortcuts should be limited, if we want to perform cost updates in reasonable time.

5.2 Random Queries

In this section we analyse the performance of TDCALT for different values of the approximation constant 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 so to favour query speed. We did not use larger values for the contraction parameters because the reduction in terms of CPU time is small, and the dynamic cost updates become unpractical (see Section 5.4). Results are recorded in Table 2, and are gathered over 10 000 queries with source and destination nodes picked at random. For comparison, we also report the results on the same road network for the time-dependent versions of Dijkstra, unidirectional ALT, TDALT and the SHARC algorithm. SHARC, introduced

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

		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	-	392	118	0.0%	0.000%	0.00%	66 908	78.1
TDALT	1.00	28	256	0.0%	0.000%	0.00%	2 931 080	2 953.3
	1.05	28	256	3.4%	0.013%	4.16%	1 516 710	1 409.5
	1.10	28	256	19.6%	0.108%	7.88%	561 253	464.2
	1.15	28	256	40.1%	0.303%	10.95%	250 248	184.4
	1.25	28	256	51.0%	0.603%	21.64%	134 911	86.1
	1.35	28	256	52.6%	0.712%	21.64%	116 090	70.3
	1.50	28	256	52.8%	0.734%	21.64%	113 040	68.1
	2.00	28	256	52.9%	0.737%	30.49%	112 826	68.0
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
	1.25	60	61	42.0%	0.549%	15.52%	4 160	5.4
	1.35	60	61	43.4%	0.649%	18.78%	3 843	4.9
	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

in [2] and augmented to time-dependent scenarios in [8], is a unidirectional technique based on arc-flags [26] and shortcuts (for details, see [2, 8]). In particular, Dijkstra's algorithm is used as a baseline to measure speedup factors, while SHARC is the fastest known algorithm for time-dependent shortest paths, although it is not able to deal with dynamic scenarios.

We report the amount of preprocessing time (in minutes) and space (in additional bytes per node) required by each algorithm. Besides, the performed queries may compute approximated results instead of optimal solutions, depending on the value of K; thus, as in Section 5.1 we record three different statistics to characterize the solution quality: error rate, average relative error, maximum relative error. We also record the average number of nodes settled at the end of the computation by each different algorithm, as well as the average CPU time in milliseconds.

In terms of preprocessing space, TDCALT with contraction parameters $C = 3.5, H = 60$ is the algorithm requiring less memory: only 61 additional bytes per node, while SHARC requires 118. Both TDALT and unidirectional ALT store landmark distances for all nodes in the graph, thus occupying 256 additional bytes per node. Algorithms which do not employ a hierarchical structure require a shorter preprocessing time: 28 minutes for TDALT and unidirectional ALT, which is the time to select 32 landmarks with the *avoid* heuristic and compute landmark distances. The contraction phase takes longer: 60 minutes for TDCALT, which only has to compute landmark distances for the core after the graph contraction, while SHARC takes 392 minutes because of the computation of arc-flags.

We now analyse the size of the search spaces and query times for the different algorithms. We restrict our attention to exact algorithms, i.e., $K = 1$. In our comparison, the algorithm with the largest average search space is Dijkstra's algorithm, with ≈ 8.8

million nodes, and represents our baseline. The TDALT algorithm yields a reduction of a factor 3 with respect to the baseline. Interestingly, the reduction is of a factor 4.31 with unidirectional ALT, thus the search space is smaller than with the bidirectional TDALT algorithm. This is easily explained if we consider that the bidirectional algorithm described in Section 1.4 may explore twice all nodes in the search space of the backward search. The SHARC algorithm yields a reduction of a factor 132.69 with respect to Dijkstra's algorithm, and TDCALT improves the factor even further, with a search space which is 145.76 times smaller than the baseline. Thus, hierarchical methods are considerably more efficient in terms of number of settled nodes, as confirmed by many studies on static road networks (e.g. [3]). However, the reduction in the number of nodes does not translate into an equal reduction of query times, because hierarchical methods need to do substantially more work per node: more edges are relaxed (due to added shortcuts), and the evaluation of pruning criteria takes time as well.

[Removed part]

If we only observe average query times for the different exact algorithms, we see that the fastest method is SHARC, which is 73.8 times faster than Dijkstra's algorithm. Second best is TDCALT, with a speedup of 47.6. Note that these speedup factors are significantly smaller than the search space reduction that the algorithms achieve, for the reasons stated above. Unidirectional ALT is faster than TDALT: the speedups with respect to the baseline are, respectively, 3.1 and 1.95.

Next, we analyse the performance of TDALT and TDCALT when increasing the value of the approximation constant K. We notice that the quality of the computed paths improves when using TDCALT with respect to TDALT for fixed K. As observed in Section 5.1, we believe that this is due to the presence of long shortcuts. The errors decrease in all respects: error rate, average relative error and maximum relative error. Search space sizes and query times greatly benefit from a value of K strictly larger than 1. The best tradeoffs between path quality and speed are obtained for $K \in [1, 1.15]$.

Finally, we observe that TDCALT is at least one order of magnitude faster than TDALT on average. If we can accept a maximum approximation factor $K \geq 1.05$ then TDCALT is also faster than (exact) SHARC, by one order of magnitude for $K \geq 1.20$.

5.3 Local Queries

For random queries, TDCALT is one order of magnitude faster than TDALT on average. TDCALT is significantly faster than unidirectional ALT, while TDALT is faster than the unidirectional ALT only for $K \geq 1.05$. In order to gain insight whether these speedups derive from small or large distance queries, Fig. 1 reports the query times with respect to the Dijkstra rank. For an s-t query, the Dijkstra rank of node t is the number of nodes settled before t is settled: thus, it is some kind of distance measure. These values were gathered on the European road network instance, using contraction parameters as in Table 2, i.e., $C = 3.5$ and $H = 60$.

Note that we use a logarithmic scale due to some outliers that require large computation time. The figure clearly indicates that both speedup techniques pay off only for long distance queries. If the source and destination node are close to each other, then unidirectional ALT is faster than TDCALT by an order of magnitude in some cases. This is expected, since for small distances TDCALT may result in a simple application of Dijkstra's algorithm, with no speedup techniques. For sufficiently long distances, however, the median of TDCALT is almost two orders of magnitude faster than unidirectional ALT. TDALT is typically positioned between unidirectional ALT and TDCALT.

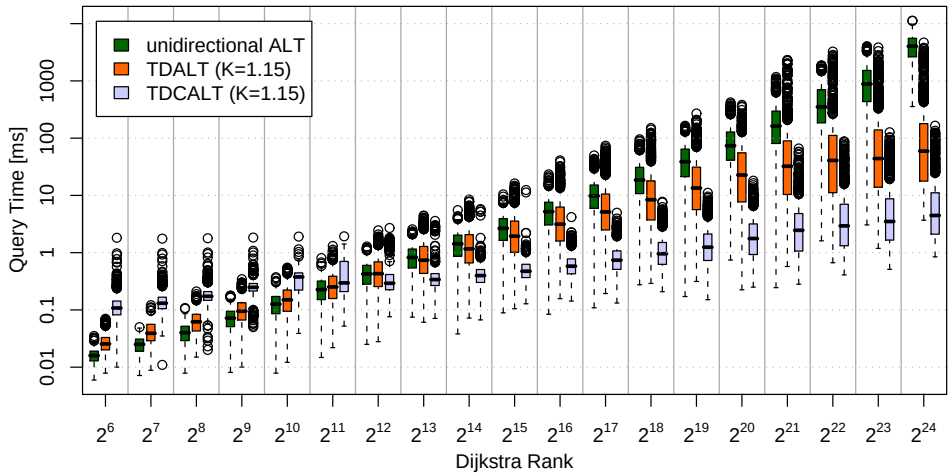

Figure 1: Comparison of unidirectional ALT, TDALT and TDCALT (on Europe) using the Dijkstra rank methodology [31]. The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

Summarizing, the proposed speedup technique is particularly effective for long distance queries, which are the most difficult cases to deal with in practice, hence the most interesting.

5.4 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 9PM, 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 are the most frequently used during the shortest path computations, thus they contribute to a large number of shortcuts.

In Table 3 we report average and maximum required time over 1000 runs to update the core in case of a single traffic jam, applying different contraction parameters. The input again is Europe. We also report the corresponding figures for a batch update of 1000 traffic jams (computed over 100 runs), in order to reduce the fluctuations and give a clearer indication of required CPU time when performing multiple updates. Besides, we measured the average and maximum time required to update the core when modifying a single breakpoint on a motorway arc selected uniformly at random; we also record the corresponding values when modifying 1000 single breakpoints on random motorway arcs (computed over 100 runs). As there is no spatial locality when updating a single breakpoint over random arcs, this represents a difficult scenario. Note that in this experiment we limit the length of shortcuts in terms of uncongested travel time (as reported in the third column). 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,

cont.		limit	space	traffic jam single[ms]		traffic jam batch[ms]		single breakpoint single[ms]		single breakpoint batch[ms]		query
C	H	[min]	[B/n]	av.	max	av.	max	av.	max	av.	max	[ms]
0.0	0	–	256	0.0	0	0	0	0.0	0	0	0	188.2
0.5	10	5	123	0.4	28	372	488	0.1	5	97	166	81.5
		10	121	0.7	49	619	799	0.1	12	183	383	85.2
		15	119	0.7	49	707	1 083	0.1	11	202	407	74.2
		20	119	0.7	49	820	1 200	0.2	59	291	459	73.8
1.0	20	5	82	7.8	229	7 144	8 090	1.8	78	1 853	2 041	34.5
		10	72	21.2	778	20 329	22 734	5.8	371	5 957	9 266	27.1
		15	68	32.1	2 226	27 327	33 313	7.2	427	7 291	11 522	25.4
		20	66	37.0	2 231	30 787	39 470	8.8	1 197	8 476	11 426	22.8
2.0	30	5	88	17.4	290	16 293	17 493	5.7	283	5 019	6 017	33.7
		10	82	90.5	3 868	79 092	85 259	27.6	1 894	24 943	27 501	22.8
		15	79	171.0	4 604	120 018	142 455	49.4	2 451	46 237	58 936	19.7
		20	77	219.7	5 073	187 595	206 569	63.3	5 510	60 940	65 954	16.4

Table 3: CPU time required to update the core for different contraction parameters and limits for the length of shortcuts.

and if we allow the shortcuts length to grow indefinitely we may have unpractical update times. Hence, we also report preprocessing space in terms of additional bytes per node, and query times with $K = 1.15$. We remark that Table 3 only considers the CPU time required to update the core, and does not take into account the computational effort to modify the cost functions for arcs at level 0 in the hierarchy, i.e., not belonging to the core. However, this effort is negligible in practice, because the modification of a breakpoint of an arc outside the core has an influence only on the arc itself. Therefore, the update is carried out by simply modifying the corresponding breakpoint value, whereas the core update is considerably more time-consuming (see Section 4).

As expected, the effort to update the core becomes more expensive with increasing contraction parameters. First, we consider the scenario where we generate 1000 traffic jams over motorway arcs, and modify the cost functions accordingly. For $C = 0.5, H = 10$ the updates are very fast, even if we allow long shortcuts (i.e. 20 minutes of uncongested travel time). The average CPU time for an update of 1000 traffic jams is always smaller than 1 second, therefore we are able to deal with a large number of breakpoint modifications in a short time. This is confirmed by the very small average time required to update the core after modifying a random breakpoint on a random motorway arc, which is smaller than 0.2 milliseconds. As we increase the contraction parameters, dynamic updates take longer to deal with. A larger number of long shortcuts is created, therefore update times grow rapidly, requiring several seconds. The average time to update the core after adding 1000 traffic jams with contraction parameters $C = 1, H = 20$ is at least one order of magnitude larger than the respective values with parameters $C = 0.5, H = 10$. Very large updates are feasible in practice only if we limit the length of shortcuts to 5 minutes of uncongested travel time; for most practical applications, however, updates are not very frequent, therefore adding 1000 traffic jams in ≈ 30 seconds is reasonably fast. If we consider contraction parameters $C = 2, H = 30$, then the updates for this scenario may require several minutes; however, limiting the length of shortcuts helps.

Next, we analyse update times for modifications of a single breakpoint over random motorway arcs. We observe that they confirm the analysis for the previous scenario (adding 1000 traffic jams). For small contraction parameters (or if we limit shortcuts to

Table 4: Performance of TDCALT on our German road network instance. *Scenario* depicts the traffic day.

		Preproc.		Error			Query		
		time	space		relative		#settled	#relaxed	time
scenario	K	[min]	[B/n]	rate	av.	max	nodes	edges	[ms]
Monday	1.00	9	50.3	0.0%	0.000%	0.00%	2 984	11 316	4.84
	1.15	9	50.3	8.3%	0.051%	11.00%	1 588	5 303	1.84
	1.50	9	50.3	8.3%	0.052%	17.25%	1 587	5 301	1.84
midweek	1.00	9	50.3	0.0%	0.000%	0.00%	3 190	12 255	5.36
	1.15	9	50.3	8.2%	0.051%	13.84%	1 593	5 339	1.87
	1.50	9	50.3	8.2%	0.052%	13.84%	1 592	5 337	1.86
Friday	1.00	8	44.9	0.0%	0.000%	0.00%	3 097	12 162	5.21
	1.15	8	44.9	7.8%	0.052%	11.29%	1 579	5 376	1.82
	1.50	8	44.9	7.8%	0.054%	21.19%	1 579	5 374	1.82
Saturday	1.00	6	27.8	0.0%	0.000%	0.00%	1 856	7 188	2.42
	1.15	6	27.8	4.4%	0.031%	11.50%	1 539	5 542	1.71
	1.50	6	27.8	4.4%	0.031%	24.17%	1 539	5 541	1.71
Sunday	1.00	5	19.1	0.0%	0.000%	0.00%	1 773	6 712	2.13
	1.15	5	19.1	4.0%	0.029%	12.72%	1 551	5 541	1.68
	1.50	5	19.1	4.1%	0.029%	17.84%	1 550	5 540	1.68

a small length in terms of uncongested travelling time), updating the core after modifying one breakpoint requires on average less than 10 milliseconds, whereas if we modify 1000 breakpoints we need less than 10 seconds. For $C = 0.5, H = 10$ we can carry out the updates in less than 0.5 seconds. If we allow shortcuts to grow, then updates may require several seconds.

If we compare the time required to update the core after adding 1000 traffic jams with respect to modifying 1000 breakpoints, we see that our update routine greatly benefits from spatial locality of the modified arcs: the first scenario is only ≈3-4 times slower than the second, but the number of modified arcs is larger, because each traffic jam extends over several motorway arcs. However, this is expected: as each shortcut is updated only once, modifications on contiguous arcs may require no additional effort, if all modified arcs belong to the same shortcut. In real-world applications, traffic jams typically occur on contiguous arcs [25], therefore our update routine should behave better in practice than in worst-case scenarios.

Summarizing, 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 arcs have their cost changed we can rerun the core arcs computation, i.e., recomputing all arcs on the core from scratch, which only takes a few minutes.

5.5 Traffic Days

Next, we focus on the impact of arc cost perturbation on TDCALT, where by perturbation we mean the difference between the static lower bounds used to compute landmark distances, and the time-dependent costs. Table 4 reports the performance of TDCALT using our German road network with our different traffic scenarios as input. Dijkstra

settles 2.2 million nodes in ≈ 1.5 seconds in this setup, independent of the traffic day.

We observe that approximation values of $K > 1.15$ do not pay off in terms of query performance and switching from exact to approximate queries yields less improvement for Germany than for Europe. Moreover, it does not pay off to drop correctness in low traffic scenarios. Still, query performance of TDCALT is excellent. Exact queries are between 280 and 704 times faster—depending on the traffic situation—than plain Dijkstra. The traffic scenario has almost no influence on approximate TDCALT: less than 10% of the queries return suboptimal paths even if $K > 1$ is used. Such paths can be computed 900 times faster than with Dijkstra. We also observe that we obtain better speed-ups with this input data than on the European road network, which may suggest that our synthetic traffic data for the European network is overly pessimistic.

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] G. V. Batz, D. Delling, P. Sanders, and C. 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] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14:2.4, August 2009. Special Section on Selected Papers from ALENEX 2008.

[3] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3, January 2010. Special Section devoted to WEA'08.

[4] K. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14:493–498, 1966.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[6] C. Daganzo. Reversibility of time-dependent shortest path problem. Technical report, Institute of Transportation Studies, University of California, Berkeley, 1998.

[7] B. C. Dean. Continuous-time dynamic shortest path algorithms. Master's thesis, Massachussets Institute of Technology, 1999.

[8] D. Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, Sept. 2008.

[9] D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, 2008.

[10] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[11] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 141–174. American Mathematical Society, 2009.

[12] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In Demetrescu [13], pages 52–65.

[13] C. Demetrescu, editor. *6th Workshop on Experimental Algorithms*, volume 4525 of *LNCS*, New York, 2007. Springer.

[14] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] S. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

[16] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In McGeoch [27], pages 319–333.

[17] A. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 156–165, Philadelphia, 2005. SIAM.

[18] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX 06)*, Lecture Notes in Computer Science, pages 129–143. Springer, 2006.

[19] A. Goldberg, H. Kaplan, and R. Werneck. Better landmarks within reach. In Demetrescu [13], pages 38–51.

[20] A. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *Proceedings of the 7th Workshop on Algorithm Engineering and Experimentation (ALENEX 05)*, pages 26–40, Philadelphia, 2005. SIAM.

[21] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 93–139. American Mathematical Society, 2009.

[22] E. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science and Cybernetics*, SSC-4(2):100–107, 1968.

[23] T. Ikeda, M. Tsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. In *Proceedings for the IEEE Vehicle Navigation and Information Systems Conference*, pages 291–296, 2004.

[24] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[25] B. S. Kerner. *The Physics of Traffic*. Springer, Berlin, 2004.

[26] U. Lauther. An extremely fast exact algorithm for finding shortest paths in static networks with geographical background. In M. Raubal, A. Sliwinsky, and W. Kuhn, editors, *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22 of *IfGI prints*, pages 219–230. Institut für Geoinformatik, Westfalische Wilhelms-Universität, 2004.

[27] C. McGeoch, editor. *Proceedings of the 8th Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *Lecture Notes in Computer Science*, New York, 2008. Springer.

[28] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A^* search for time-dependent fast paths. In McGeoch [27], pages 334–346.

[29] A. Orda and R. Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[30] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.

[31] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In G. Stølting Brodal and S. Leonardi, editors, *13th Annual European Symposium on Algorithms (ESA 2005)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[32] P. Sanders and D. Schultes. Dynamic highway-node routing. In Demetrescu [13], pages 66–79.