Bidirectional A^* Search on Time-Dependent Road Networks

Giacomo Nannicini[1,2], Daniel Delling[3], Dominik Schultes[3], Leo Liberti[1]

[1] *LIX, École Polytechnique, F-91128 Palaiseau, France*
Email:{giacomon,liberti}@lix.polytechnique.fr

[2] *Mediamobile, 27 boulevard Hippolyte Marquès, 94200 Ivry sur Seine, France*

[3] *Universität Karlsruhe (TH), 76128 Karlsruhe, Germany*
Email:{delling,schultes}@ira.uka.de

April 2, 2010

Abstract

The computation of point-to-point shortest paths on time-dependent road networks has a large practical interest, but very few works propose efficient algorithms for this problem. We propose a novel approach which tackles one of the main complications of route planning in time-dependent graphs, which is the difficulty of using bidirectional search: since the exact arrival time at the destination is unknown, we start a backward search from the destination node using lower bounds on arc costs in order to restrict the set of nodes that have to be explored by the forward search. Our algorithm is based on A^* with landmarks (ALT); extensive computational results show that it is very effective in practice if we are willing to accept a small approximation factor, resulting in a speed-up of several times with respect to Dijkstra's algorithm while finding only slightly sub-optimal solutions. The main idea presented here can also be generalized to other types of search algorithms.
Keywords. Shortest paths, time-dependent costs, large-scale road networks, goal directed search.

1 Introduction

Route planning in road networks is a practical application that, in recent years, has attracted a lot of attention to the computation of shortest paths on large graphs. In particular, since in several countries there are now road segments covered with traffic sensors, it is possible to generate speed profiles based on historical data. It thus becomes feasible to model the dependence of travelling speed on the time of the day; consequently, situations like rush hour traffic peaks can be taken into account during the calculation, giving much more meaningful results with respect to the static case (where arc costs are always fixed) from the users point of view. In a typical application scenario, e.g., a server machine which provides a route planning web server, one would like to answer several shortest path queries in less than one second of CPU time, on graphs with several millions nodes. This means that we are interested in an algorithm which is able to quickly find good solutions to the Time-Dependent Shortest Path

Problem, which we define as follows.

Time-Dependent Shortest Path Problem (TDSPP): 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 starting time $\tau_0 \in T$ and a time-dependent arc weight function $c : A \times T \to \mathbb{R}_+$ such that for each pair of time instants τ, τ' with $\tau < \tau'$ the property $\forall\ (u, v) \in A\ c(u, v, \tau) + \tau \leq c(u, v, \tau') + \tau'$ holds, 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:

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

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

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

Note that we require the arc weight function to satisfy a certain condition, known as the FIFO property. The FIFO property is also called the *non-overtaking property*, because it basically says that if T_1 leaves u at time τ and T_2 at time $\tau' > \tau$, T_2 cannot arrive at v before T_1 using the arc (u, v). For the TDSPP, the FIFO assumption is usually necessary in order to mantain polynomial complexity: the SPP in time-dependent FIFO networks is polynomially solvable [25], while it is NP-hard in non-FIFO networks [30].

We assume that a function $\lambda : A \to \mathbb{R}_+$ with the following property:

$$\forall (u, v) \in A, \tau \in T \quad \lambda(u, v) \leq c(u, v, \tau),$$

is known. In other words, $\lambda(u, v)$ is a lower bound on the travelling time of arc (u, v) for all time instants in T. In practice, this can easily be computed, given an arc length and the maximum allowed speed on that arc. We naturally extend λ to be defined on paths, i.e., $\lambda(p) = \sum_{(v_i,v_j)\in p} \lambda(v_i, v_j)$. We call G_λ the graph G weighted by the lower bounding function λ.

In this paper, we propose a novel algorithm for the TDSPP on FIFO networks based on a bidirectional A^* algorithm. Since the arrival time is not known in advance (so c cannot be evaluated on the arcs adjacent to the destination node), our backward search occurs on G_λ, and is therefore a time-*independent* search. This is used for bounding the set of nodes that will be explored by the forward search. An extended abstract of this work appeared in [28]. [28] represented the first attempt to tackle the TDSPP in a bidirectional fashion. Since then, our idea has been used for several shortest paths algorithms on time-dependent networks (see e.g., [2, 10]).

1.1 Related Work

Many ideas have been proposed for the computation of point-to-point shortest paths on static graphs (see [11] for a review), and there are algorithms capable of finding the solution in a matter of a few microseconds [1]; adaptations of those ideas for dynamic scenarios, i.e., where arc costs are updated at regular intervals, have been tested as well [12, 27, 32, 34] (see [29] for a survey).

Much less work has been undertaken on the time-dependent variant of the shortest paths problem. The TDSPP was first addressed in [7]: a recursive formula is given to establish the minimum time to travel to a given target starting from a given source at a certain time τ. In [16], Dijkstra's algorithm

[15] is extended to the dynamic case, but the FIFO property, which is necessary to prove that Dijkstra's algorithm terminates with a correct shortest paths tree on time-dependent networks, is not mentioned. Since Dijkstra's algorithm plays an important role in this paper, we review it in Section 2. Given source and destination nodes s and t, the problem of maximizing the departure time from node s with a given arrival time at node t is equivalent to the TDSPP (see [8]). A survey on the TDSPP is given in [13].

Goal-directed search, also called A^* [23], has been adapted to work on all the previously described scenarios; an efficient version for the static case has been presented in [20], and then developed and improved in [21]. 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 [6] and [12].

After the publication of the extended abstract of this work [28], several speed-up techniques have been adapted to the time-dependent scenario. The SHARC-algorithm [3] allows fast *unidirectional* shortest-path calculations in large scale networks. Due to its unidirectional nature, it can easily be used in a time-dependent scenario [9]. Moreover, Contraction Hierarchies [19] have been adapted as well [2], but the memory consumption of this approach is very large. Finally, in [10] the approach introduced in this work is enhanced with an exact bi-level search method (i.e., most of the search is carried out on a smaller network that plays the same role as motorways in real-life road networks). The resulting algorithm, TDCALT (Time-Dependent Core-ALT) [10], is one of the fastest known techniques for route planning in time-dependent road networks.

1.2 Overview

The rest of this paper is organised as follows. First, we review Dijkstra's algorithm in Section 2. In Section 3 we describe A^* search and the ALT algorithm, which are needed for our method. In Section 4 we provide the foundations of our idea in a simple way by employing Dijkstra's algorithm, while we adapt those ideas to the ALT agorithm, giving a specific implementation, in Section 5. We formally prove our method's correctness in Section 6 for both exact and approximated shortest path computations. In Section 7 we propose some modifications that improve the performance of our algorithm, and prove their correctness. Computational experiments in Section 8 show the feasibility of our approach.

2 Dijkstra's Algorithm

Dijkstra's algorithm [15] solves the single source SPP in static directed graphs with non-negative weights in polynomial time. The algorithm can easily be generalized to the time-dependent case [16]. Dijkstra's algorithm is a so-called labeling method.

The *labeling method* for the SPP [17] finds shortest paths from the source to all vertices in the graph; on a static graph with arc weights $w(u, v)$ $\forall(u, v) \in A$, the method works as follows: for every vertex v it maintains its distance label $\ell[v]$, parent node $p[v]$, and status $S[v]$ which may be one of the following: `unreached`, `explored`, `settled`. Initially $\ell[v] = \infty$, $p[v] = NIL$, and $S[v] =$ `unreached` for every vertex v. The method starts by setting $\ell[s] = 0$ and

$S[s]$ = `explored`; while there are labeled (i.e., explored) vertices, the method picks an `explored` vertex u, relaxes all outgoing arcs of u, and sets $S[u]$ = `settled`. To relax an arc (u, v), one checks if $\ell[v] > \ell[u] + w(u, v)$ and, if true, sets $\ell[v] = \ell[u] + w(u, v)$, $p(v) = u$, and $S(v)$ = `explored`. At any iteration of the algorithm, the set of nodes with status `explored` or `settled` is called the *search scope*. If the graph does not contain cycles with negative cost, the labeling method terminates with correct shortest path distances and a shortest path tree. The algorithm can be extended to the time-dependent case on FIFO networks by a simple modification of the arc relaxation procedure: if τ_0 is the departure time from the source node, we check if $\ell[v] > \ell[u] + c(u, v, \tau_0 + \ell[u])$ and, if true, set $\ell[v] = \ell[u] + c(u, v, \tau_0 + \ell[u])$, $p[v] = u$, and $S[v]$ = `explored`. The efficiency of the label-setting method depends on the rule to choose a vertex to scan next. We say that $\ell[v]$ is exact if it is equal to the distance from s to v; it is easy to see that if one always selects a vertex u such that $\ell[u]$ is exact at the selection time, then each vertex is scanned at most once. In this case we only need to relax arcs (u, v) where v is not `settled`, and the algorithm is called *label-setting*. Dijkstra [15] observed that if the cost function c is non-negative and v is an explored vertex with the smallest distance label, then $\ell[v]$ is exact; so, we refer to the labeling method with the minimum label selection rule as Dijkstra's algorithm. If $w(u, v)$ is non-negative $\forall (u, v) \in A$ then Dijkstra's algorithm scans vertices in nondecreasing order of distance from s and scans each vertex at most once; for the point-to-point SPP, we can terminate the labeling method as soon as the target node is `settled`. The algorithm requires $O(|A| + |V| \log |V|)$ amortized time if the queue is implemented as a Fibonacci heap [18]; with a binary heap, the running time is $O((|E| + |V|) \log |V|)$. Note that on road networks, we typically have $|E| = O(|V|)$, therefore the running time of Djkstra's algorithm with binary heaps is $O(|V| \log |V|)$.

One basic variant of Dijkstra's algorithm for the point-to-point SPP is bidirectional search; instead of building only one shortest path tree rooted at the source node s, we also build a shortest path tree rooted at the target node t on the reverse graph (i.e., the graph $\overline{G} = (V, \overline{A})$ where $(u, v) \in \overline{A} \Leftrightarrow (v, u) \in A$). As soon as one node v becomes `settled` in both searches, we are guaranteed that the concatenation of the shortest $s \to v$ path found in the forward search and of the shortest $v \to t$ path found in the backward search is a shortest $s \to t$ path. Since we can think of Dijkstra's algorithm as exploring nodes in circles centered at s with increasing radius until t is reached (see Figure 1), the bidirectional variant is faster because it explores nodes in two circles centered at both s and t, until the two circles meet (see Figure 2); the area within the two circles, which represents the number of explored nodes, will then be smaller than in the unidirectional case, up to a factor of two.

Dijkstra's algorithm applied to time-dependent FIFO networks has been optimized in various ways [4, 5]. We note here that in the time-dependent scenario bidirectional search cannot be applied, since the arrival time at destination node is unknown. We also remark that all speedup techniques based on finding shortest paths in Euclidean graphs [33] cannot be applied either, since the typical arc cost function, the arc travelling time at a certain time of the day, does not yield a Euclidean graph.

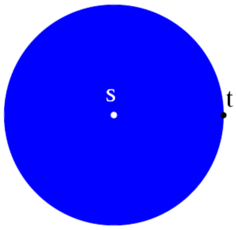

Figure 1: Schematic representation of Dijkstra's algorithm search space

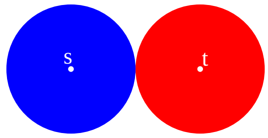

Figure 2: Schematic representation of bidirectional Dijkstra's algorithm search space.

3 A^* with Landmarks

A^* is an algorithm for goal-directed search, similar to Dijkstra's algorithm, but which adds a potential function to the priority key of each node in the queue. The potential function on a node v is an estimate of the distance to reach the target from v; A^* then follows the same procedure as Dijkstra's algorithm, but the use of this potential function, summed to the priority key of each node, has the effect of prioritizing nodes that are likely to be closer to the target node t. If the potential function π 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. A^* is guaranteed to explore no more nodes than Dijkstra's algorithm: if $\pi(v)$ is a good approximation from below of the distance to target, A^* efficiently drives the search towards the destination node, and it explores considerably fewer nodes than Dijkstra's algorithm; if $\pi(v) = 0\,\forall v \in V$, A^* behaves exactly like Dijkstra's algorithm, i.e., it explores the same nodes. In [24] it is shown that A^* is equivalent to Dijkstra's algorithm on a graph with reduced costs $w_\pi(u,v) = w(u,v) - \pi(u) + \pi(v)$; as the length of each path between s and t changes by the same amount $\pi(t) - \pi(s)$, the shortest path is invariant. Note that, since Dijkstra's algorithm requires arc costs to be nonnegative, the potential function should be *consistent*, i.e., $\pi(u) \leq w(u,v) + \pi(v)\ \forall (u,v) \in A$.

One way to compute the potential function, instead of using Euclidean distances, is to use the concept of *landmarks*. Landmarks were first proposed in [20]; they are a preprocessing technique which is based on the triangular inequality. The basic principle is as follows: suppose we have selected a set $L \subset V$ of landmarks, and we have precomputed 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 $\pi_t(u) = \max_{\ell \in L}\{d(u,\ell) - d(t,\ell), d(\ell,t) - d(\ell,u)\}$ is a lower bound for the distance $d(u,t)$, and it can be used as a potential function which preserves optimal paths. On static (i.e., non time-dependent) graphs, landmarks can be used to implement bidirectional search, using some care in modifying the potential function so that it is consistent for both forward and backward search [21]. This translates to ensuring that $w_{\pi_f}(u,v)$ in G is equal to $w_{\pi_b}(v,u)$ in the reverse graph $\overline{G}$, where π_f and π_b are the potential functions for the forward and the backward search, respectively. Bidirectional A^* with the potential function described above is called ALT. It is straightforward to note that, if arc costs can only increase with respect to their original value, i.e., the value used in the precomputation of landmark distances, then the potential function associated to landmarks is still a valid lower bound, even on a time-dependent graph. In [12] this idea is applied to a real road network in order to analyse algorithmic performances, but with a unidirectional search. On road networks, the initial arc cost, which should be a lower bound on the time-dependent cost on that arc, can be easily computed by dividing the arc's length by the maximum allowed speed on that arc's road category.

The choice of landmarks has a great impact on the size of the search space, as it severely affects the quality of the potential function. Several selection strategies exist, although none of them is optimal with respect to random queries, in the sense that none is guaranteed to yield the smallest search space for random source-destination pairs. The best known heuristics are *Avoid* and *Max-Cover* [20, 22].

4 Bidirectional Search on Time-Dependent Graphs

We assume that we are given a graph $G = (V, A)$, source and destination vertices $s, t \in V$, and a departure time $\tau_0 \in T$. In the rest of this paper, we denote by $d(u, v, \tau)$ the length of the shortest path from u to v with departure time τ, and by $d_\lambda(u, v)$ the length of the shortest path from u to v on the graph G_λ. The approach that we propose for bidirectional search on time-dependent graphs is based on a modification of Dijkstra's algorithm.

For any $u, v, \in V, \tau \in T$ let $l(u, v, \tau) \leq d(u, v, \tau)$ be any lower bounding function for the distance between u and v with departure time τ. Assume that we have an upper bound μ on the cost of the optimal solution to TDSPP; e.g., μ is the time-dependent cost of any $s \to t$ path with departure time τ_0. We run Dijkstra's algorithm with the following pruning criterion: eliminate any unsettled node v for which

$$\max\{d(s, u, \tau_0) : u \text{ is } \texttt{settled}\} + l(v, t, \max\{d(s, u, \tau_0) : u \text{ is } \texttt{settled}\}) > \mu. \tag{3}$$

In the following, the term "pruning" stands for "do not insert in the priority queue". Prop. 4.1 estabilishes correctness of our approach.

4.1 Proposition
Nodes satisfying (3) are not necessary to compute the shortest path from s to t with departure time τ_0 using Dijkstra's algorithm.

Proof. Suppose that, at some iteration of Dijkstra's algorithm, the pruning criterion (3) eliminates some nodes which are on the shortest path p^* from s to t with departure time τ_0. Let u be the first of these nodes. Then u is such that $d(s, u, \tau_0) \geq \max\{d(s, u, \tau_0) : u \text{ is } \texttt{settled}\}$. Hence $\gamma_{\tau_0}(p^*) \leq \mu < \max\{d(s, u, \tau_0) : u \text{ is } \texttt{settled}\} + l(u, t, \max\{d(s, u, \tau_0) : u \text{ is } \texttt{settled}\}) \leq d(s, u, \tau_0) + l(u, t, d(s, u, \tau_0)) \leq \gamma_{\tau_0}(p^*)$, which is a contradiction. For the last inequality in the chain, we need the FIFO property. □

This far, the algorithm looks unidirectional, and we did not specify how the lower bounds $l(u, v, \tau)$ can be obtained. We use bidirectional search to this end. Our proposal is as follows: run a backward search from t on G_λ. For each node v settled by the backward search, $d_\lambda(v, t) \leq d(v, t, \tau)\ \forall \tau \in T$, hence we can use $l(v, t, \tau) = d_\lambda(v, t)$. Therefore, the backward search's purpose is to provide bounds for the pruning criterion of the forward search (which is the only search that uses time-dependent costs).

We still need to specify several missing details: how do we obtain μ? How do we choose between performing forward or backward search iterations, and when should the backward search be stopped? Furthermore, we can see intuitively that a straightforward implementation of this algorithm is not likely to be useful in practice, because we would need to perform an extensive backward search (i.e., explore a large portion of the graph around the target, even in the "wrong" direction) before we are able to effectively prune the forward search. In the next section we tackle all this issues, by employing the A^* algorithm instead of Dijkstra's algorithm.

5 Bidirectional Search with A^*

In Section 4 we have described a general framework for bidirectional search on time-dependent graphs. In this section we fill in the missing details, giving a description that can be implemented in practice, and employ the A^* algorithm instead of Dijkstra's algorithm; recall that A^* is a generalization of Dijkstra's algorithm, in that Dijkstra's algorithm is equivalent to A^* with a zero potential function.

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 run on the graph weighted by c with the path cost defined by (1)-(2), 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 to be `explored` by both forward and backward search; let $\mu = \gamma_{\tau_0}(p_v)$, where p_v is the path from s to t passing through v. In the second phase, both searches 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.

The pseudocode for this algorithm is given in Algorithm 1. Note that we use the symbol $\leftrightarrow$ to indicate either the forward search ($\leftrightarrow = \rightarrow$) or the backward search ($\leftrightarrow = \leftarrow$). We denote by $\overrightarrow{A}$ the set of arcs for the forward search, i.e., $\overrightarrow{A} = A$, and by $\overleftarrow{A}$ the set of arcs for the backward search, i.e., $\overleftarrow{A} = \{(u, v) | (v, u) \in A\}$. A typical choice is to alternate between the forward and the backward search at each iteration of the algorithm during the first two phases. Algorithm 1 works with any choice of feasible potential function; but since we use landmark-based potentials (see Section 3), we call this algorithm Time-Dependent ALT (TDALT). A schematic representation of the different phases is given in Fig. 3.

Algorithm 1 TIME-DEPENDENT ALT: Compute the shortest time-dependent path from s to t with departure time τ_0

$\overrightarrow{Q}$.insert$(s, 0)$; $\overleftarrow{Q}$.insert$(t, 0)$; $M := \emptyset$; $\mu := +\infty$; *done* := `false`; *phase* := 1.
while $\neg$*done* **do**
  if $(phase = 1) \vee (phase = 2)$ **then**
    $\leftrightarrow \in \{\rightarrow, \leftarrow\}$
  else
    $\leftrightarrow := \rightarrow$
  $u := \overleftrightarrow{Q}$.extractMin()
  if $(u = t) \wedge (\leftrightarrow = \rightarrow)$ **then**
    done := `true`
    continue
  if $(phase = 1) \wedge (u.\text{dist}^{\rightarrow} + u.\text{dist}^{\leftarrow} < \infty)$ **then**
    $\mu := u.\text{dist}^{\rightarrow} + u.\text{dist}^{\leftarrow}$
    $phase := 2$
  if $(phase = 2) \wedge (\leftrightarrow = \leftarrow) \wedge (\mu < u.\text{key}^{\leftarrow})$ **then**
    $phase := 3$
    continue
  for all arcs $(u, v) \in \overleftrightarrow{A}$ **do**
    if $\leftrightarrow = \leftarrow$ **then**
      M.insert(u)
    else if $(phase = 3) \wedge (v \notin M)$ **then**
      continue;
    if $(v \in \overleftrightarrow{Q})$ **then**
      if $u.\text{dist}^{\leftrightarrow} + c(u, v, u.\text{dist}^{\leftrightarrow}) < v.\text{dist}^{\leftrightarrow}$ **then**
        $\overleftrightarrow{Q}$.decreaseKey$(v, u.\text{dist}^{\leftrightarrow} + c(u, v, u.\text{dist}^{\leftrightarrow}) + \overleftrightarrow{\pi}(v))$
    else
      $\overleftrightarrow{Q}$.insert$(v, u.\text{dist}^{\leftrightarrow} + c(u, v, u.\text{dist}^{\leftrightarrow}) + \overleftrightarrow{\pi}(v))$
return $t.\text{dist}^{\rightarrow}$

It is easy to see how TDALT is an instantiation, that employs A^*, of the idea discussed in Section 4. In Phase 1, we use the backward search to find an $s \rightarrow t$ path, in order to compute an upper bound μ to the optimal cost of the solution. In Phase 2, we alternate between the two searches in order to fulfill the pruning criterion (3). Phase 2 ends once (3) is satisfied for all nodes that are not in M. At this point, we only need to explore nodes in M.

6 Correctness

We prove correctness of Algorithm 1. Intuitively, this follows from Prop. 4.1 by applying the necessary modifications to deal with A^* instead of Dijkstra's algorithm. However, we give a slightly different proof, which makes explicit reference to β, because we will use the same technique to prove other claims in the remainder of the paper.

6.1 Theorem
Algorithm 1 computes the shortest time-dependent path from s to t for a given departure time τ_0.

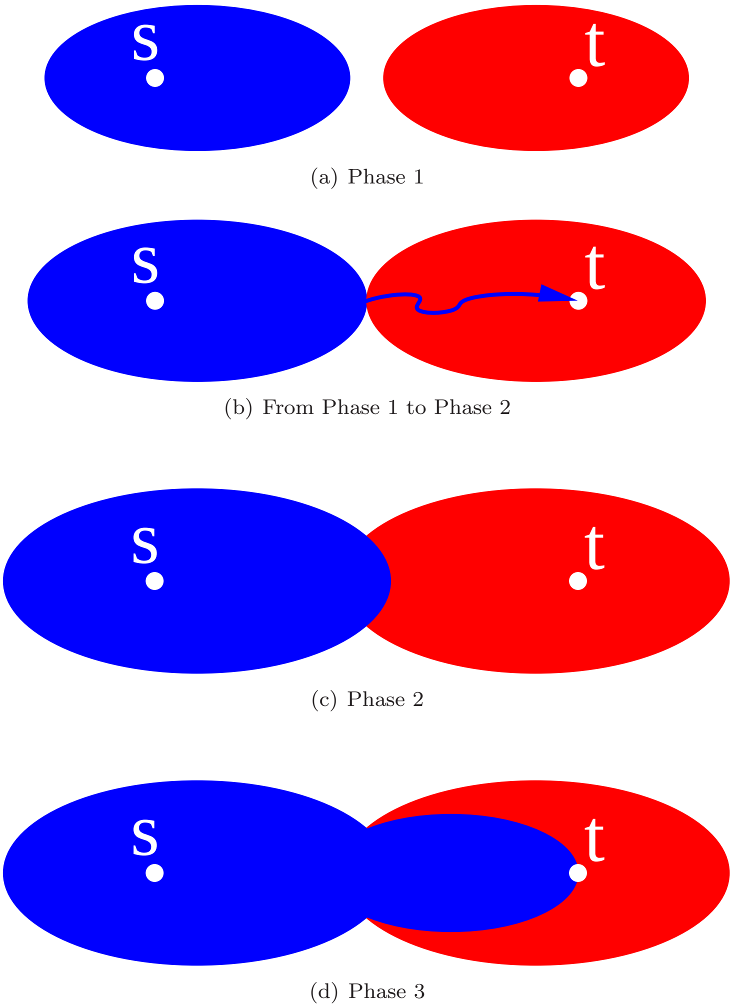

Figure 3: Schematic representation of the TDALT algorithm's search space

Proof. The forward search of Algorithm 1 is exactly the same as the unidirectional version of the A^* algorithm during the first 2 phases, and thus it is correct; just we have to prove that the restriction applied during Phase 3 does not interfere with the correctness of the A^* algorithm, i.e., that we do not prune nodes on the shortest path.

Let μ be an upper bound on the cost of the shortest path; in particular, this can be the cost $\gamma_{\tau_0}(p_v)$ of the $s \to t$ path passing through the first meeting point v of the forward and backward search. Let β be the smallest key of the backward search priority queue at the end of Phase 2. Let p^* be the shortest path from s to t with departure time τ_0, and suppose there are some nodes on p^* which are not settled by the forward search. Let u be the first of these nodes; this implies that $u \notin M$, i.e., u has not been settled by the backward search during the first 2 phases of Algorithm 1. Hence, we have that $\beta \leq \pi_b(u) + d_\lambda(u,t)$; then we have the chain $\gamma_{\tau_0}(p^*) \leq \mu < \beta \leq \pi_b(u) + d_\lambda(u,t) \leq d_\lambda(s,u) + d_\lambda(u,t) \leq d(s,u,\tau_0) + d(u,t,d(s,u,\tau_0)) = \gamma_{\tau_0}(p^*)$, which is a contradiction. □

6.2 Theorem
Let p^* be the shortest path from s to t. If the condition to switch to Phase 3 is $\mu < K\beta$ for a fixed parameter K, then Algorithm 1 computes a path p from s to t such that $\gamma_{\tau_0}(p) \leq K\gamma_{\tau_0}(p^*)$ for a given departure time τ_0.

Proof. Suppose that $\gamma_{\tau_0}(p) > K\gamma_{\tau_0}(p^*)$. Let u be the first node on p^* which is not explored by the forward search; by Phase 3, this implies that $u \notin M$, i.e., u has not been settled by the backward search during the first 2 phases of Algorithm 1. Hence, we have that $\beta \leq \pi_b(u) + d_\lambda(u,t)$; then we have the chain $\gamma_{\tau_0}(p) \leq \mu < K\beta \leq K(\pi_b(u) + d_\lambda(u,t)) \leq K(d_\lambda(s,u) + d_\lambda(u,t)) \leq K(d(s,u,\tau_0) + d(u,t,d(s,u,\tau_0))) = K(\gamma_{\tau_0}(p^*)) < \gamma_{\tau_0}(p)$, which is a contradiction. □

7 Improvements

Performance of the basic version of the algorithm can be improved with the results that we describe in this section.

7.1 Theorem
Let p^* be the shortest path from s to t with departure time τ_0. If all nodes u on p^* settled by the backward search are settled with a key smaller or equal to $d(s,u,\tau_0) + d(u,t,d(s,u,\tau_0))$, then Algorithm 1 is correct.

Proof. Let Q be the backward search queue, let $\text{key}(u)$ be the key for the backward search of node u, let $\beta = \text{key}(v)$ be the smallest key in the backward search queue, which is attained at a node v, and let μ the best upper bound on the cost of the solution currently known. To prove correctness, using the same arguments as in the proof of Thm. 6.1 we must make sure that, when the backward search stops at the end of Phase 2, then all nodes on the shortest path from s to t that have not been explored by the forward search have been added to M. The backward search stops when $\mu < \beta$.

In an A^* search on FIFO networks, the keys of settled nodes are non-decreasing. So every node u which at the current iteration has not been settled by the backward search will be settled with a key $\text{key}(u) \geq \text{key}(v)$, which yields $d(s,u,\tau_0) + d(u,t,d(s,u,\tau_0) \geq \text{key}(v) = \beta > \mu \ \forall u \in Q$. Thus, every node which has not been settled by the backward search cannot be on the shortest path from s to t, and Algorithm 1 is correct. □

This allows the use of larger lower bounds during the backward search: the backward A^* search does not have to compute shortest paths on the graph G_λ, but it should in any case guarantee that, when a node u is settled, its key is an underestimation of the time-dependent cost of the time-dependent shortest path between s and t passing through u. This is similar to what is required by the modified Dijkstra's algorithm of Section 4.

The next proposition is of fundamental practical importance; it states that the backward search can be pruned at nodes already settled by the forward search, because all nodes that are on the shortest path and that have not been settled by the forward search can be reached by the backward search through another path.

7.2 Proposition
During Phase 2 the backward search does not need to explore nodes that have already been settled by the forward search.

Proof. Suppose that the forward search has settled all nodes on the shortest $s \to t$ path p^* up to node u. Then clearly all remaining nodes on p^* are reachable from t in the backward search with a path that does not use any node already settled by the forward search (i.e., the subpath of p^* from u to t). Therefore, the backward search can be pruned at all nodes settled by the forward search. □

By Thm. 7.1, we can take advantage of the fact that the backward search is used only to bound the set of nodes explored by the forward search. This means that we can tighten the bounds used by the backward search: the potential function for the backward search does not have to be feasible. To derive some valid lower bounds we need the following proposition.

7.3 Proposition
At a given iteration, let v be any node settled by the forward search. Then, for each node w which has not been settled by the forward search, $d(s, v, \tau_0) + \pi_f(v) - \pi_f(w) \leq d(s, w, \tau_0)$.

Proof. Since the forward search uses a feasible and consistent potential function π_f, nodes u are settled by increasing value of $d(s, u, \tau_0) + \pi_f(u)$. Hence, for any node unsettled node w we have $d(s, v, \tau_0) + \pi_f(v) \leq d(s, w, \tau_0), +\pi_f(w)$. □

Let v' be any node settled by the forward search, and w a node which has not been settled. Prop. 7.3 suggests that we can use

$$\pi_b^*(w) = \max\{\pi_b(w), d(s, v', \tau_0) + \pi_f(v') - \pi_f(w)\} \tag{4}$$

as a lower bound to $d(s, w, \tau_0)$ during the backward search. To maximize this bound, we should choose v' that maximizes $d(s, v', \tau_0) + \pi_f(v')$, i.e., choose v' as the last node settled by the forward search. We remark that π_b^* is *not* a feasible potential for the backward search: it yields valid lower bounds for the time-dependent graph, but it could overestimate distances on G_λ. The following lemma gives a way to use (4) in a correct way.

7.4 Lemma
If the node v' used to compute the potential function π_b^* defined by (4) is fixed, then we have $\pi_b^*(v) \leq \pi_b^*(u) + \lambda(u, v)$ for each arc $(u, v) \in A$ such that v has not been settled by the forward search.

Proof. By definition we have $\pi_b^*(v) = \max\{\pi_b(v), \alpha - \pi_f(v)\}$, where with α we denoted the key of v', that is, $d(s, v', \tau_0) + \pi_f(v')$, which is fixed by hypothesis. Consider the case $\pi_b^*(v) = \pi_b(v)$; then, since the landmark potential functions π_b and π_f are consistent, we have $\pi_b^*(v) = \pi_b(v) \leq \pi_b(u)+\lambda(u, v) \leq \pi_b^*(u)+\lambda(u, v)$.

Now consider the case $\pi_b^* = \alpha - \pi_f(v)$; then we have $\pi_b^*(v) = \alpha - \pi_f(v) \leq \alpha - \pi_f(u) + \lambda(u,v) \leq \pi_b^*(u) + \lambda(u,v)$, which completes the proof. □

One could think that feasibility and consistency of π_b^* for the backward search on G_λ would follow from Lemma 7.4, by chaining the inequalities $\pi_b^*(v) \leq \pi_b^*(u) + \lambda(u,v)$ for nodes on the shortest path to s. However this is not the case, because Lemma 7.4 is only valid for nodes that have not been settled by the forward search; therefore, the chain of inequalities would fail as soon as we encounter a node that has been settled by the forward search. We can still prove correctness of our algorithm with tightened bounds, thanks to Thm. 7.1.

7.5 Theorem
If we use the potential function π_b^* defined by (4) as potential function for the backward search, for a fixed node v settled by the forward search, then Algorithm 1 is correct.

Proof. Let $d_b(u)$ be the distance from a node u to node t computed by the backward search. We will prove that, when a node u on the shortest path from s to t is settled by the backward search, $d_b(u) \leq d(u, t, d(s, u, \tau_0))\ \forall \tau_0 \in T$. By Prop. 7.3 and Thm. 7.1, this is enough to prove our statement.

Let $q^* = (v_1 = u, \ldots, v_n = t)$ be the shortest path from u to t on G_λ. We proceed by induction on $i : n, \ldots, 1$ to prove that each node v_i is settled with the correct distance on G_λ, i.e. $d_b(v_i) = d_\lambda(v_i, t)$. It is trivial to see that the nodes v_n and v_{n-1} are settled with the correct distance on G_λ. For the induction step, suppose v_i is settled with the correct distance $d_b(v_i) = d_\lambda(v_i, t)$. By Lemma 7.4, we have $d_b(v_i) + \pi_b^*(v_i) \leq d_b(v_i) + \lambda(v_{i-1}, v_i) + \pi_b^*(v_{i-1}) = d_\lambda(v_{i-1}, t) + \pi_b^*(v_{i-1}) \leq d_b(v_{i-1}) + \pi_b^*(v_{i-1})$, hence v_i is extracted from the queue before v_{i-1}. This means that v_{i-1} will be settled with the correct distance $d_b(v_{i-1}) = d_\lambda(v_{i-1}, t)$, and the induction step is proven.

Thus, u will be settled with distance $d_b(u) = d_\lambda(u, t) \leq d(u, t, d(s, u, \tau_0))$, which proves our statement. □

By Thm. 7.5, Algorithm 1 is correct when using π_b^* only if we assume that the node v' used in (4) is fixed at each backward search iteration. Thus, we do the following: we set up 10 checkpoints during the query; when a checkpoint is reached, the node v' used to compute (4) is updated, and the backward search queue is flushed and filled again using the updated π_b^*. This is enough to guarantee correctness. The checkpoints are computed comparing the initial lower bound $\Delta = \pi_f(t)$ and the current distance from the source node, both for the forward search: the initial lower bound is divided by 10 and, whenever the current distance from the source node exceeds $k\Delta/10$ with $k \in \{1, \ldots, 10\}$, π_b^* is updated.

8 Computational Results

In this section, we present an extensive experimental evaluation of the TDALT algorithm. Our implementation is written in C++ using solely the standard template library. As priority queue we use a binary heap. Other types of priority queues were also tested, but it turns out that the impact of the choice of the priority queue has almost no influence on the performance of speed-up techniques. 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 4.2.1, using optimization level 3.

Unless otherwise stated, we use 16 *maxCover* landmarks (see Section 3), computed on the input graph using the lower bounding function λ to weight edges, and we use (4) as potential function for the backward search, with 10 checkpoints (see Section 7). We use dynamic landmark selection, as suggested in [22]. 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. We evaluate the query performance by reporting the average number of settled nodes, i.e., the number of nodes extracted from the priority queues, the number of relaxed edges, and the resulting running times.

Inputs. 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, congesture 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, our European data set does not contain traffic data. We therefore 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; we used piecewise linear cost functions, with one breakpoint for each hour over a day. Arc costs are generated assigning, at each node, several random values that represent peak hour (i.e., hour with maximum traffic increase), duration and speed of traffic increase/decrease for a traffic jam; for each node, two traffic jams are generated, one in the morning and one in the afternoon. Then, for each arc in a node's arc star, a *speed profile* is generated, using the traffic jam's characteristics of the corresponding node, and assigning a random increase factor between 1.5 and 3 to represent that arc's slowdown during peak hours with respect to uncongested hours. We do not assign speed profile to arcs that have both endpoints at nodes with level 0 in a pre-constructed Highway Hierarchy [31], because they represent "unimportant" nodes of the road network (i.e., nodes that only appear in local shortest paths, as opposed to long-distance shortest paths). As a result those arcs will have the same travelling time value throughout the day; for all other arcs, we use the traffic jam values associated with the endpoint with smallest ID.

This method was developed to ensure spatial coherency between traffic increases, i.e., if a certain arc is congested at a given time, then it is likely that adjacent arcs will be congested too. This is a basic principle of traffic analysis [26].

Table 1: Performance of the time-dependent versions of Dijkstra, unidirectional ALT, and our bidirectional approach.

		Error			Query			
			relative		# settled nodes			time
method	K	rate	avg	max	Phase 1	Phase 2	Phase 3	[ms]
Dijkstra	-	0.0%	0.000%	0.00%	-	-	8 877 158	5 757.4
uni-ALT	-	0.0%	0.000%	0.00%	-	-	2 143 160	1 520.8
TDALT	1.00	0.0%	0.000%	0.00%	132 129	2 556 840	3 009 320	2 842.0
	1.05	3.1%	0.012%	3.91%	132 129	1 244 050	1 574 750	1 379.2
	1.07	6.6%	0.034%	6.06%	132 129	849 171	1 098 470	915.4
	1.10	18.1%	0.106%	7.79%	132 129	473 414	622 466	481.9
	1.12	26.1%	0.182%	10.57%	132 129	337 353	444 991	325.0
	1.15	35.4%	0.292%	10.57%	132 129	236 108	311 209	214.2
	1.20	43.0%	0.485%	19.40%	132 129	171 154	225 557	145.3
	1.25	45.4%	0.589%	21.64%	132 129	148 856	196 581	122.3
	1.30	46.4%	0.656%	21.64%	132 129	139 089	184 143	111.6
	1.35	47.0%	0.704%	21.64%	132 129	134 582	178 410	107.4
	1.50	47.1%	0.722%	21.64%	132 129	132 299	175 468	105.4
	1.75	47.2%	0.726%	30.49%	132 129	132 131	175 248	105.4
	2.00	47.2%	0.726%	30.49%	132 129	132 130	175 247	105.4

Random Queries. Table 1 reports the results of our bidirectional ALT variant on time-dependent networks for different approximation values K using the European road network as input. Preprocessing takes approximately 75 minutes and produces 128 *additional* bytes per node (for each node we have to store distances to and from all landmarks).

As the performed TDALT queries compute approximated results instead of optimal solutions, 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. We also report the number of nodes settled at the *end* of each phase of our algorithm, denoting them with the labels Phase 1, Phase 2 and Phase 3.

As expected, we observe a clear trade-off between the quality of the computed solution and query performance. If we are willing to accept an approximation factor of $K = 2.0$, on the European road network queries are on average 55 times faster than Dijkstra's algorithm, but almost 50% of the computed paths will be suboptimal and, although the average relative error is still small, in the worst case the approximated solution has a cost which is 50% larger than the optimal value. The reason for this poor solution quality is that, for such high approximation values, Phase 2 is very short. As a consequence, nodes in the middle of the shortest path are not explored by our approach, and the meeting point of the two search scopes is far from being the optimal one. However, by decreasing the value of the approximation constant K we are able to obtain solutions that are very close to the optimum, and performance is significantly

Table 2: Performance of the time-dependent versions of Dijkstra, unidirectional ALT and our bidirectional approach *without* the tightened potential function π_b^* defined as in (4).

		Error			Query			
			relative		# settled nodes			time
method	K	rate	avg	max	Phase 1	Phase 2	Phase 3	[ms]
Dijkstra	-	0.0%	0.000%	0.00%	-	-	8 877 158	5 757.4
uni-ALT	-	0.0%	0.000%	0.00%	-	-	2 143 160	1 520.8
TDALT	1.00	0.0%	0.000%	0.00%	719 650	3 763 990	3 862 070	3 291.6
	1.05	3.5%	0.023%	4.88%	719 650	2 996 940	3 238 120	2 683.5
	1.07	5.5%	0.046%	6.94%	719 650	2 519 750	2 874 500	2 290.7
	1.10	12.1%	0.123%	9.45%	719 650	1 810 340	2 201 870	1 619.2
	1.12	20.1%	0.237%	10.93%	719 650	1 416 240	1 772 080	1 218.4
	1.15	32.1%	0.474%	14.35%	719 650	1 049 750	1 345 930	842.0
	1.20	44.4%	0.788%	19.42%	719 650	824 331	1 079 290	618.3
	1.25	50.5%	0.994%	24.57%	719 650	755 262	996 631	553.3
	1.30	53.3%	1.104%	24.57%	719 650	735 524	972 294	531.5
	1.35	54.7%	1.166%	24.57%	719 650	727 843	962 950	526.5
	1.50	56.1%	1.248%	28.16%	719 650	720 359	953 704	524.7
	1.75	56.3%	1.261%	39.34%	719 650	719 661	952 947	519.0
	2.00	56.4%	1.262%	39.41%	719 650	719 650	952 933	518.2

better than for unidirectional ALT or Dijkstra. In our experiments, it seems as if the best trade-off between quality and performance is achieved with an approximation value of $K = 1.15$, which yields average query times smaller than 215 ms with a maximum recorded relative error of 10.6%. As in road networks the speed profiles that weight arcs cannot be completely accurate, settling for a slightly suboptimal solution (on average, less than 0.3% over the optimum for $K = 1.15$) is usually not a problem. By decreasing K to values < 1.05 it does not pay off to use the bidirectional variant any more, as the unidirectional variant of ALT is faster and is always correct.

An interesting observation is that for $K = 2.0$ switching from a static to a time-dependent scenario increases query times only of a factor of ≈ 2: on the European road network, in a static scenario, ALT has query times of 53.6 ms (see [12]), while our time-dependent variant yields query times of 105 ms. We also note that for our bidirectional search there is an additional overhead which increases the time spent per node with respect to unidirectional ALT: on the European road network, using an approximation factor of $K = 1.05$ yields similar query times to unidirectional ALT, but the number of nodes settled by the bidirectional approach is almost 30% smaller. We suppose that this is due to the following facts: in the bidirectional approach, one has to check at each iteration if the current node has been settled in the opposite direction, and during Phase 2 the upper bound has to be updated from time to time. The cost of these operations, added to the phase-switch checks, is probably not negligible.

We also report, for comparison, the results obtained on the European road network using the unmodified ALT potential function π_b for the backward search, instead of the tightened one π_b^* defined as in (4). These can be found in Table 2, which has the same column labels as Table 1. Comparing query times with the same value of the approximation constant K, we see that using the potential function π_b^* yields a significant improvement over π_b. The difference in performance is larger as K increases. For $K = 1$ the difference is very small; for $K = 1.05$ the algorithm with π_b is 95% slower than the one with π_b^*, and the slowdown increases to 236% for $K = 1.10$ and to 293% for $K = 1.15$. With the largest approximation factor that we tested in our experiments, $K = 2$, the algorithm without the tightened potential function is more than 5 times slower. The same behaviour is observed in terms of the number of settled nodes: while for $K = 1$ the number is very similar (only a 28% increase when not using π_b^*), the ratio rapidly grows until it reaches a 444% increase for $K = 2$. Thus, a great deal of the significant improvement that we are able to obtain over Dijkstra's algorithm and unidirectional ALT with our bidirectional variant is due to the use of tightened bounds. If we use the standard ALT potential function π_b for the backward search then we do not manage to obtain a speed-up of more than a factor 3 with respect to unidirectional ALT, but this comes at the price of correctness. Summarizing, in our bidirectional approach one of the great advantages is that we are able to derive better lower bounds for the time-dependent search with respect to the original ALT bounds, and the new potential function accounts for a large computational improvement.

Local Queries. For random queries on the European road network, our bidirectional TDALT algorithm with $K = 1.15$ is roughly 6.5 times faster than unidirectional ALT on average. In order to gain insight whether this speed-up derives from small- or large-range queries, Fig. 4 reports the query times with respect to the Dijkstra rank[1]. These values were gathered on the European road network instance. Note that we use a logarithmic scale due to the fluctuating query times of bidirectional TDALT. Comparing both ALT version, we observe that switching from uni- to bidirectional queries pays off especially for long-distance queries. This is not surprising, because for small distances the overhead for bidirectional routing is not counterbalanced by a significant decrease in the number of explored nodes: unidirectional ALT is faster for local queries. For ranks of 2^{24}, the median of the bidirectional variant is almost 2 orders of magnitude lower than for the unidirectional variant. Another interesting observation is the fact that some outliers of bidirectional TDALT are almost as slow as the unidirectional variant. Comparing different approximation values, we observe that query times differ by roughly the same factor for all ranks less than 2^{23}.

Number of Landmarks. In static scenarios, query times of bidirectional TDALT can be significantly reduced by increasing the number of landmarks to 32 or even 64 (see [12]). In order to check whether this also holds for our time-dependent variant, we recorded our algorithm's performance using different numbers of landmarks. Table 3 reports those results on the European road network. We evaluate 8 maxcover landmarks (yielding a preprocessing effort of 33 minutes and an overhead of 64 bytes per node), 16 maxcover landmarks (75 minutes, 128 bytes per node) and 32 avoid landmarks (29 minutes, 256 bytes per node). Note that we do not report error rates here, as it turned out that the number of landmarks has almost no impact on the quality of

[1]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.

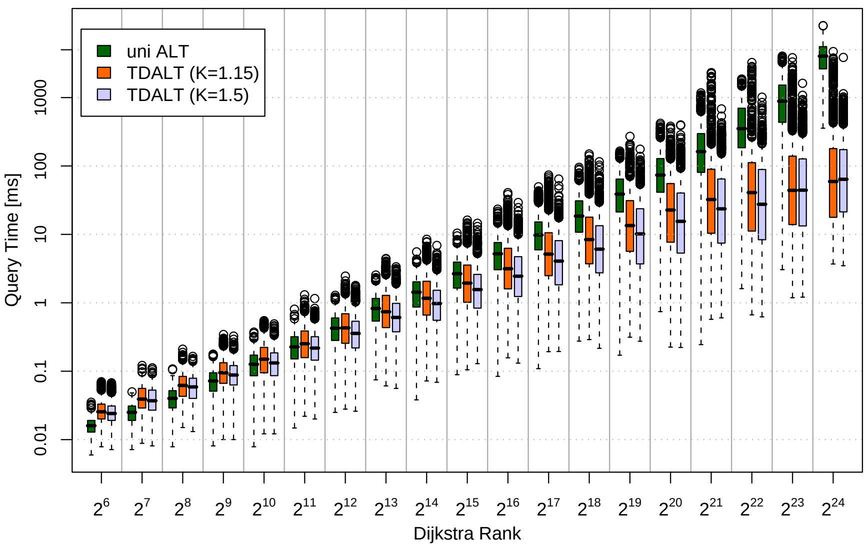

Figure 4: Comparison of unidirectional ALT and bidirectional TDALT using the Dijkstra rank methodology [31]. The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

the computed paths. Surprisingly, the number of landmarks has a very small

Table 3: Performance of unidirectional ALT and bidirectional TDALT with different number of landmarks in a time-dependent scenario.

		8 landmarks		16 landmarks		32 landmarks	
	K	# settled	time [ms]	# settled	time [ms]	# settled	time [ms]
uni-ALT	-	2 280 420	1 446.4	2 143 160	1 520.8	2 056 190	1 623.3
TDALT	1.00	3 147 440	2 745.5	3 009 320	2 842.0	2 931 080	2 953.3
	1.05	1 714 210	1 373.8	1 574 750	1 379.2	1 516 710	1 409.5
	1.10	768 368	540.2	622 466	481.9	561253	464.2
	1.15	461 259	293.5	311 209	214.2	250 248	184.4
	1.20	375 900	230.6	225 557	145.3	164 419	111.1
	1.50	326 076	195.8	175 468	105.3	113 040	68.1
	2.00	325 801	195.8	175 247	105.4	112 826	68.0

influence on the performance of TDALT. Even worse, increasing the number of landmarks even yields larger average query times for unidirectional ALT and for bidirectional TDALT with low K-values. This is due the fact that the search space decreases only slightly, but the additional overhead for accessing landmarks increases when there are more landmarks to take into account (even though we use dynamic landmark selection, because of increased cache misses). However, when increasing K, a larger number of landmarks yields faster query times: with $K = 2.0$ and 32 landmarks we are able to perform time-dependent queries 70 times faster than plain Dijkstra, but the solution quality in this case is as poor as in the 16 landmarks case. Summarizing, for $K > 1.10$ increasing the number of landmarks has a positive effect on computational times, although switching from 16 to 32 landmarks does not yield the same benefits as from 8 to 16, and thus in our experiments is not worth the extra memory. On the other hand, for $K \leq 1.10$ and for unidirectional ALT increasing the number of landmarks has a negative effect on computational times, and thus is never a good choice in our experiments.

Traffic Days. Next, we focus on the impact of arc cost perturbation on TDALT, where by perturbation we mean the difference between the static lower bounds used to compute landmark distances, and the time-dependent costs. Therefore, Table 4 reports the performance of uni- and bidirectional time-dependent ALT for different traffic days on the German road network. Dijkstra settles 2.2 million nodes in ≈ 1.5 seconds in this setup, independent of the traffic day.

Table 4: Performance of TDALT on our German road network instance. *Scenario* depicts the degree of perturbation.

			Error			Query		
				relative		#settled	#relaxed	time
scenario	algorithm	K	rate	av.	max	nodes	edges	[ms]
Monday	uni-ALT	–	0.0%	0.000%	0.00%	193 087	230 665	140.38
	TDALT	1.00	0.0%	0.000%	0.00%	106 743	127 190	88.53
		1.15	12.5%	0.094%	13.02%	51 137	60 838	37.23
		1.50	12.5%	0.096%	24.27%	51 119	60 816	37.12
midweek	uni-ALT	–	0.0%	0.000%	0.00%	200 236	239 112	147.20
	TDALT	1.00	0.0%	0.000%	0.00%	116 476	138 696	98.27
		1.15	12.4%	0.094%	14.32%	50 764	60 398	36.91
		1.50	12.5%	0.097%	27.59%	50 742	60 371	36.86
Friday	uni-ALT	–	0.0%	0.000%	0.00%	196 551	235 083	143.52
	TDALT	1.00	0.0%	0.000%	0.00%	116 857	139 175	98.28
		1.15	12.0%	0.096%	14.03%	50 891	60 550	36.92
		1.50	12.1%	0.098%	30.77%	50 874	60 531	36.82
Saturday	uni-ALT	–	0.0%	0.000%	0.00%	148 331	177 568	100.07
	TDALT	1.00	0.0%	0.000%	0.00%	63 717	76 001	47.41
		1.15	10.5%	0.088%	13.97%	50 042	59 607	36.00
		1.50	10.6%	0.089%	26.17%	50 036	59 600	35.63
Sunday	uni-ALT	–	0.0%	0.000%	0.00%	142 631	170 670	92.79
	TDALT	1.00	0.0%	0.000%	0.00%	58 956	70 333	42.96
		1.15	10.4%	0.088%	14.28%	50 349	59 994	36.04
		1.50	10.5%	0.089%	32.08%	50 345	59 988	35.74

We observe that the degree of perturbation has only a mild impact on unidirectional ALT and bidirectional TDALT. In a low traffic scenario, unidirectional ALT queries are up 16 times faster than plain Dijkstra, while this values drops to 10 if more edges are perturbed. Switching from exact to approximate queries does not pay off in low traffic scenarios: the gain in performance is only around 20% which seems rather low compared to the loss in quality of paths. However, this value increases to a factor of up to 3 in high traffic scenarios. Still, comparing Tables 1 and 4, the gain in performance for dropping correctness is much lower for Germany than for Europe. We assume that this derives from the size of the graph. With increasing graph size, lower bounds get worse as the gap between lower bound distance and time-dependent distance increases. This would also explain why speed-ups for unidirectional ALT are higher for Germany than for Europe.

9 Conclusion

We have presented an algorithm which applies bidirectional search on a time-dependent road network, where the backward search is used to bound the set of nodes that have to be explored by the forward search; this algorithm is based on the ALT variant of the A^* algorithm. We have discussed related theoretical issues, and we proved the algorithm's correctness. Our idea can be adapted and applied to several shortest path algorithms, and lays the foundations for future work. Extensive computational experiments show that this algorithm is very effective in practice if we are willing to accept a small approximation factor: the exact version of our algorithm is slower than unidirectional ALT, but if we can accept a decrease of the solution quality of a few percentage points with respect to the optimum then our algorithm is several times faster. For practical applications, this is usually a good compromise.

References

[1] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[2] 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.

[3] 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.

[4] L. Buriol, M. Resende, and M. Thorup. Speeding up dynamic shortest path algorithms. *INFORMS Journal on Computing*, accepted for publication.

[5] I. Chabini. Discrete dynamic shortest path problems in transportation applications: complexity and algorithms with optimal run time. *Transportation Research Records*, 1645:170–175, 1998.

[6] I. Chabini and S. Lan. 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In Demetrescu [14], pages 52–65.

[13] D. Delling and D. Wagner. Time-Dependent Route Planning. In R. K. Ahuja, R. H. Möhring, and C. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

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

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

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

[17] L. R. Ford and D. R. Fulkerson. *Modern Heuristic Techniques for Combinatorial Problems.* Princeton University Press, Princeton, NJ, 1962.

[18] M. Fredman and R. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] G. Nannicini, P. Baptiste, G. Barbier, D. Krob, and L. Liberti. Fast paths in large-scale dynamic road networks. *Computational Optimization and Applications*, 45(1):143–158, 2010.

[28] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A^* search for time-dependent fast paths. In C. McGeoch, editor, *Proceedings of the 8th Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *Lecture Notes in Computer Science*, pages 334–346, New York, 2008. Springer.

[29] G. Nannicini and L. Liberti. Shortest paths on dynamic graphs. *International Transactions in Operational Research*, 15:551–563, 2008.

[30] 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.

[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 [14], pages 66–79.

[33] R. Sedgewick and J. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1(1):31–48, 1986.

[34] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM Journal of Experimental Algorithmics*, 10:1–30, 2005.