

Bidirectional A^* Search on Time-Dependent Road Networks

GIACOMO NANNICINI^{1,2}, DANIEL DELLING³, DOMINIK SCHULTES³, LEO LIBERTI¹

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

June 24, 2008

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 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 hours traffic peaks can be taken into account during the calculation, giving much more meaningful results with respect to the static case (i.e. 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 $c : A \times T \rightarrow \mathbb{R}_+$, find a path $p = (s = v_1, \dots, v_k = t)$ in G such that its *time-dependent cost* $\gamma_{\tau_0}(p)$, defined recursively as follows:

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

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

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

We assume that a function $\lambda : A \rightarrow \mathbb{R}_+$ 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. $\lambda(p) = \sum_{(v_i, v_j) \in p} \lambda(v_i, v_j)$.

In this paper, we propose a novel algorithm for the TDSPP 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 the graph weighted by the lower bounding function λ . This is used for bounding the set of nodes that will be explored by the forward search.

1.1 Related Work

Many ideas have been proposed for the computation of point-to-point shortest paths on static graphs (see [22, 21] 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 [7, 20, 23, 17].

Much less work has been undertaken on the time-dependent variant of the shortest paths problem; we refer to [17] for a survey. The TDSPP was first addressed in [4]: 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 [9], Dijkstra's algorithm [8] 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. The FIFO property states that for each pair of time instants τ, τ' with $\tau < \tau'$:

$$\forall (u, v) \in A \quad c(u, v, \tau) + \tau \leq c(u, v, \tau') + \tau',$$

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 maintain polynomial complexity: the SPP in time-dependent FIFO networks is polynomially solvable [15], while it is NP-hard in non-FIFO networks [18]. 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 [5]).

Goal-directed search, also called A^* [13], has been adapted to work on all the previously described scenarios; an efficient version for the static case has been presented in [10], and then developed and improved in [11]. Those ideas have been used in [7] on dynamic graphs as well, while the time-dependent case on graphs with the FIFO property has been addressed in [3] and [7].

Moreover, the recently developed SHARC-algorithm [2] 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. While originally the SHARC-algorithm was able to deal only with approximative queries on time-dependent graphs, it has recently been extended [6] to compute optimal solutions. However, as this new version was not available at the time of writing this paper, we compare our results to [2].

1.2 Overview

The rest of this paper is organised as follows. In Section 2 we describe A^* search and the ALT algorithm, which are needed for our method. In Section 3 we describe the foundations of our idea, and present an adaptation of the ALT algorithm based on it. In Section 4 we formally prove our method's correctness. In Section 5 we propose some modifications that improve the performance of our algorithm, and prove their correctness. In Section 6 we discuss computational experiments and provide computational results.

2 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 [14] 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.

One way to compute the potential function, instead of using Euclidean distances, is to use the concept of *landmarks*. Landmarks were first proposed in [10]; 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 func-

tion 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 [11]. The consistency condition requires that $w_{\pi_f}(u, v)$ in G should be 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 [7] 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 *maxCover* [12].

3 Bidirectional Search on Time-Dependent Graphs

Our 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, i.e. the backward search is a reverse search in G_λ , which corresponds to the graph G weighted by the lower bounding function 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 run on the graph weighted by c with the path cost defined by (1)-(2), and the backward search is run on the graph weighted by the lower bounding function λ . All nodes settled by the backward search are included in a set M . Phase 1 terminates as soon as the two search scopes meet.
2. Suppose that $v \in V$ is the first vertex in the intersection of the heaps of the forward and backward search; then the time dependent cost $\mu = \gamma_{\tau_0}(p_v)$ of the path p_v going from s to t passing through v is an upper bound to $\gamma_{\tau_0}(p^*)$. 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.

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 \vec{A} the set of arcs for the forward search, i.e. $\vec{A} = A$, 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 Compute the shortest time-dependent path from s to t with departure time τ_0

```

1:  $\vec{Q}.\text{insert}(s, 0)$ ;  $\overleftarrow{Q}.\text{insert}(t, 0)$ ;  $M := \emptyset$ ;  $\mu := +\infty$ ;  $done := \text{false}$ ;  $phase := 1$ .
2: while  $\neg done$  do
3:   if  $(phase = 1) \vee (phase = 2)$  then
4:      $\leftrightarrow \in \{\rightarrow, \leftarrow\}$ 
5:   else
6:      $\leftrightarrow := \rightarrow$ 
7:    $u := \vec{Q}.\text{extractMin}()$ 
8:   if  $(u = t) \wedge (\leftrightarrow = \rightarrow)$  then
9:      $done := \text{true}$ 
10:   continue
11:   if  $(phase = 1) \wedge (u.\text{dist}^{\rightarrow} + u.\text{dist}^{\leftarrow} < \infty)$  then
12:      $\mu := u.\text{dist}^{\rightarrow} + u.\text{dist}^{\leftarrow}$ 
13:      $phase := 2$ 
14:   if  $(phase = 2) \wedge (\leftrightarrow = \leftarrow) \wedge (\mu < u.\text{key}^{\leftarrow})$  then
15:      $phase := 3$ 
16:   continue
17:   for all arcs  $(u, v) \in \overleftarrow{A}$  do
18:     if  $\leftrightarrow = \leftarrow$  then
19:        $M.\text{insert}(u)$ 
20:     else if  $(phase = 3) \wedge (v \notin M)$  then
21:       continue;
22:     if  $(v \in \vec{Q})$  then
23:       if  $u.\text{dist}^{\leftarrow} + c(u, v, u.\text{dist}^{\leftarrow}) < v.\text{dist}^{\leftarrow}$  then
24:          $\vec{Q}.\text{decreaseKey}(v, u.\text{dist}^{\leftarrow} + c(u, v, u.\text{dist}^{\leftarrow}) + \overleftarrow{\pi}(v))$ 
25:       else
26:          $\vec{Q}.\text{insert}(v, u.\text{dist}^{\leftarrow} + c(u, v, u.\text{dist}^{\leftarrow}) + \overleftarrow{\pi}(v))$ 
27: return  $t.\text{dist}^{\rightarrow}$ 

```

4 Correctness

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 . We have the following theorems.

4.1 Theorem

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

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; we have to prove that the restriction applied during phase 3 does not interfere with the correctness of the A^* algorithm.

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 $s \rightarrow t$ 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. Suppose that Algorithm 1 is not correct, i.e. it computes a sub-optimal path. Let p^* be the shortest path from s to t with departure time τ_0 , and 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 $\gamma_{\tau_0}(p^*) \leq \mu < \beta \leq \pi_b(u) + d_\lambda(u, t) \leq d_\lambda(s, u) + d_\lambda(u, t) \leq d(s, u, \tau_0) + d(u, t, d(s, u, \tau_0)) = \gamma_{\tau_0}(p^*)$, which is a contradiction. \square

4.2 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. \square

5 Improvements

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

5.1 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. 4.1 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, 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. \square

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 then its key is an underestimation of the time-dependent cost of the time-dependent shortest path between s and t passing through u . The next proposition is of fundamental practical importance.

5.2 Proposition

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

Proof. Let $d_b(v)$ be the distance from a node v to node t computed by the backward search if we do not explore any node already explored by the forward search. We will prove that, when a node v on the shortest path from s to t with departure time τ_0 is settled by the backward search, then $d_b(v) \leq d(v, t, d(s, v, \tau_0)) \forall \tau_0 \in T$. By Thm. 5.1, this is enough to prove our statement.

Consider a node v settled by the backward search, but not by the forward search; let q be the shortest path from s to v with departure time τ_0 , let q^* be the shortest path from v to t with departure time $\tau_v = \gamma_{\tau_0}(q)$. Suppose that q^* does not pass through any node already settled by the forward search. Then $d_b(v) \leq \lambda(q^*) \leq d(v, t, d(s, v, \tau_0))$.

Suppose now that q^* passes through a node w already settled by the forward search. Let p be the shortest path from s to w with departure time τ_0 , and let p^* be the shortest path from w to t with departure time $\tau_w = \gamma_{\tau_0}(p)$; clearly v cannot be on p , because otherwise it would have been settled by the forward search. So we have, by the FIFO property and by optimality of p , that $\gamma_{\tau_0}(p) + \gamma_{\tau_w}(p') \leq \gamma_{\tau_0}(q) + \gamma_{\tau_v}(q')$, which means that v does not have to be explored and added to the set M by the backward search, because we already have a better path passing through w . Thus, even if $\text{key}(v) > d(s, v, \tau_0) + d(v, t, d(s, v, \tau_0))$ Algorithm 1 is correct. \square

By Thm. 5.1, 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, even if doing so would result in an A^* backward search that computes suboptimal distances. To derive some valid lower bounds we need the following lemma and propositions.

5.3 Lemma

Let v be a node, and u its parent node in the shortest path from s to v with departure time τ_0 . Then $d(s, u, \tau_0) + \pi_f(u) \leq d(s, v, \tau_0) + \pi_f(v)$.

Proof. Suppose that ℓ is the active landmark, i.e. the landmark in our landmarks set that currently gives the best bound; we have that either $\pi_f(u) = d_\lambda(u, \ell) - d_\lambda(t, \ell)$ or $\pi_f(u) = d_\lambda(\ell, t) - d_\lambda(\ell, u)$.

First case: $\pi_f(u) = d_\lambda(u, \ell) - d_\lambda(t, \ell)$. We have $d(s, u, \tau_0) + \pi_f(u) = d(s, u, \tau_0) + d_\lambda(u, \ell) - d_\lambda(t, \ell) \leq d(s, u, \tau_0) + d_\lambda(u, v) + d_\lambda(v, \ell) - d_\lambda(t, \ell) \leq d(s, u, \tau_0) + \lambda(u, v) + d_\lambda(v, \ell) - d_\lambda(t, \ell) \leq d(s, v, \tau_0) + \pi_f(v)$.

Second case: $\pi_f(u) = d_\lambda(\ell, t) - d_\lambda(\ell, u)$. We have $d(s, u, \tau_0) + \pi_f(u) = d(s, u, \tau_0) + d_\lambda(\ell, t) - d_\lambda(\ell, u)$; by triangular distance, $d_\lambda(\ell, v) \leq d_\lambda(\ell, u) + d_\lambda(u, v) \leq d_\lambda(\ell, u) + \lambda(u, v)$, which yields $-d_\lambda(\ell, u) \leq -d_\lambda(\ell, v) + \lambda(u, v)$. So $d(s, u, \tau_0) + d_\lambda(\ell, t) - d_\lambda(\ell, u) \leq d(s, u, \tau_0) + d_\lambda(\ell, t) - d_\lambda(\ell, v) + \lambda(u, v) \leq d(s, v, \tau_0) + \pi_f(v)$. \square

5.4 Proposition

At a given iteration, let v be the last 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. There are two possibilities for w : either it has been explored (but not settled) by the forward search, or it has not been explored. Let Q be the set of nodes in the forward search queue. If w has been explored, then $w \in Q$, and clearly $d(s, v, \tau_0) + \pi_f(v) \leq d(s, w, \tau_0) + \pi_f(w)$ because v has been extracted before w , which proves our statement. Otherwise, there is a node $u \in Q$ on the shortest path from s to w with departure time τ_0 which has been explored but not settled. We have that $d(s, v, \tau_0) + \pi_f(v) \leq d(s, u, \tau_0) + \pi_f(u)$ because v has been extracted while u is still in the queue, and by Lemma 5.3, if we examine the nodes $u = u_1, u_2, \dots, u_k = w$ on the shortest path from s to w with departure time τ_0 , we have that $d(s, u_1, \tau_0) + \pi_f(u_1) \leq \dots \leq d(s, u_k, \tau_0) + \pi_f(u_k)$, from which our statement follows. \square

Let v be as in Prop. 5.4, and w a node which has not been settled by the forward search. Prop. 5.4 suggests that we can use

$$\pi_b^*(w) = \max\{\pi_b(w), d(s, v, \tau_0) + \pi_f(v) - \pi_f(w)\} \quad (3)$$

as a lower bound to $d(s, w, \tau_0)$ during the backward search. However, we have to make sure that the bound is valid at each iteration of Algorithm 1.

5.5 Lemma

If the key of the forward search used to compute the potential function π_b^* defined by (3) is fixed, then we have $\pi_b^*(v) \leq \pi_b^*(u) + \lambda(u, v)$ for each arc $(u, v) \in A$.

Proof. By definition we have $\pi_b^*(v) = \max\{\pi_b(v), \alpha - \pi_f(v)\}$, where with α we denoted the key of a node settled by the forward search, 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. \square

This is enough to prove correctness of our algorithm with tightened bounds, as stated in the next theorem.

5.6 Theorem

If we use the potential function π_b^* defined by (3) as potential function for the backward search, with a fixed value of the forward search key, 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. 5.4 and Thm. 5.1, this is enough to prove our statement.

Let $q^* = (v_1 = u, \dots, v_n = t)$ be the shortest path from u to t on G_λ . We proceed by induction on $i : n, \dots, 1$ 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 5.5, 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. \square

By Thm. 5.6, Algorithm 1 is correct when using π_b^* only if we assume that the node v used in (3) 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 (3) 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 $k \in \{1, \dots, 10\}$, π_b^* is updated.

6 Computational Results

In this section, we present an extensive experimental evaluation of our time-dependent ALT algorithm. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.1. 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 4.1, using optimization level 3.

Unless otherwise stated, we use 16 *maxCover* landmarks [10], computed on the input graph using the lower bounding function λ to weight edges, and we use (3) as potential function for the backward search, with 10 checkpoints (see Section 5). 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.

Inputs. We tested our algorithm on two different road networks: the road network of Western Europe provided by PTV AG for scientific use, which has

approximately 18 million vertices and 42.6 million arcs, and the road network of the US, taken from the TIGER/Line Files, with 23.9 million vertices and 58.3 million arcs. A travelling time in uncongested traffic situation was assigned to each arc using that arc's category (13 categories for Europe, 4 for US) to determine the travel speed.

Modeling Traffic. Unfortunately, we are not aware of a *large* publicly available real-world road network with time-dependent arc costs. 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 [19], and 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. The breakpoints of these speed profiles are stored in memory as a multiplication factor with respect to the speed in uncongested hours, which allows us to use only 7 bits for each breakpoint. We assume that all roads are uncongested between 11PM and 4AM, so that we do not need to store the corresponding breakpoints; as a result, we store all breakpoints using 16 additional bytes per edge. The travelling time of an arc at time τ is computed via linear interpolation of the two breakpoints that precede and follow τ .

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 [16].

Random Queries. Table 1 reports the results of our bidirectional ALT variant on time-dependent networks for different approximation values K using the European and the US road network as input. For the European road network, 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); for the US road network, the corresponding figures are 92 minutes and 128 bytes per node. For comparison, we also report the results on the same road network for the time-dependent versions of Dijkstra, unidirectional ALT, and the SHARC algorithm [2].

As the performed ALT-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

Table 1: Performance of the time-dependent versions of Dijkstra, unidirectional ALT, SHARC, and our bidirectional approach. For SHARC, we use approximation values of 1.001 and 1.002 (cf. [2] for details).

input	method	K	ERROR			QUERY			
			rate	avg	max	phase 1	phase 2	phase 3	time [ms]
EUR	Dijkstra	-	0.0%	0.000%	0.00%	-	-	8 908 300	6 325.8
	uni-ALT	-	0.0%	0.000%	0.00%	-	-	2 192 010	1 775.8
	1.001-SHARC	-	57.1%	0.686%	34.31%	-	-	140 945	60.3
	1.002-SHARC	-	42.8%	0.583%	34.31%	-	-	930 251	491.4
	ALT	1.00	0.0%	0.000%	0.00%	125 068	2 744 010	3 117 160	3 342.0
		1.02	1.0%	0.003%	1.13%	125 068	2 154 900	2 560 370	2 668.7
		1.05	4.0%	0.029%	4.93%	125 068	1 333 220	1 671 630	1 665.5
		1.10	18.7%	0.203%	8.10%	125 068	549 916	719 769	646.5
		1.13	30.5%	0.366%	12.63%	125 068	340 787	447 681	376.2
		1.15	36.4%	0.467%	13.00%	125 068	265 328	348 325	281.1
		1.20	44.7%	0.652%	18.19%	125 068	183 899	241 241	180.9
		1.30	48.2%	0.804%	23.63%	125 068	141 358	186 267	131.2
		1.40	48.7%	0.835%	25.70%	125 068	133 120	175 754	121.9
		1.50	48.8%	0.844%	25.70%	125 068	130 144	172 157	118.6
USA		1.75	48.9%	0.886%	48.86%	125 068	125 268	165 925	113.0
		2.00	48.9%	0.886%	48.86%	125 068	125 071	165 650	113.0
	Dijkstra	-	0.0%	0.000%	0.00%	-	-	12 435 900	8 020.6
	uni-ALT	-	0.0%	0.000%	0.00%	-	-	2 908 170	2 403.9
	ALT	1.00	0.0%	0.000%	0.00%	272 790	4 091 050	4 564 030	4 534.2
		1.10	21.5%	0.135%	7.02%	272 790	633 758	829 176	656.3
		1.15	54.4%	0.402%	9.98%	272 790	312 575	405 699	289.6
		1.20	62.0%	0.482%	9.98%	272 790	278 345	359 190	251.1
		1.50	64.8%	0.506%	13.63%	272 790	272 790	351 865	247.5
		2.00	64.8%	0.506%	16.00%	272 790	272 791	351 854	246.8

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 better than for unidirectional ALT or Dijkstra. In our experiments, it seems

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 (3).

input	method	K	ERROR relative			QUERY			
			rate	avg	max	# settled nodes			time
						phase 1	phase 2	phase 3	[ms]
EUR	Dijkstra	-	0.0%	0.000%	0.00%	-	-	8 908 300	6 325.8
	uni-ALT	-	0.0%	0.000%	0.00%	-	-	2 192 010	1 775.8
	ALT	1.00	0.0%	0.000%	0.00%	697 810	3 917 900	3 817 100	3 764.5
		1.02	0.9%	0.002%	1.93%	697 810	3 674 520	3 671 220	3 550.7
		1.05	3.3%	0.022%	4.94%	697 810	3 042 370	3 287 630	3 081.1
		1.10	11.1%	0.110%	9.86%	697 810	1 946 720	2 343 430	1 999.1
		1.13	21.5%	0.262%	12.63%	697 810	1 390 560	1 747 560	1 376.2
		1.15	29.5%	0.420%	14.93%	697 810	1 135 990	1 454 800	1 086.2
		1.20	43.5%	0.800%	17.94%	697 810	852 142	1 111 690	757.8
		1.30	53.6%	1.141%	27.40%	697 810	731 319	965 749	619.7
		1.40	56.1%	1.248%	27.40%	697 810	709 835	939 854	600.2
		1.50	56.9%	1.282%	38.10%	697 810	703 629	932 212	592.3
		1.75	57.4%	1.339%	46.88%	697 810	698 238	925 770	586.1
		2.00	57.5%	1.351%	48.86%	697 810	697 816	925 247	584.2

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 300 ms on both road networks with a maximum recorded relative error of 13% (on the European road network, while the corresponding figure is 9.98% for the US instance). As in road networks the speed profiles that weight arcs cannot be completely accurate, settling for a slightly suboptimal solution (on average, less than 0.5% over the optimum for $K = 1.15$) usually is 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.

Comparing results for $K > 1.15$ for the US with those for Europe, we observe that the number of queries that return suboptimal paths increases, but the average and maximum error rates are smaller than the corresponding values on the European road network with the same values of K . Moreover, the speed-ups of our algorithm with respect to plain Dijkstra are lower on the US instance: the maximum recorded speed-up (for $K = 2.0$) is only of a factor 33. This behaviour has also been observed in the static scenario [7]. However, with $K = 1.15$, which is a good trade-off between quality and speed, query performance is very similar on both networks.

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-16 has query times of 53.6 ms (see [7]), while our time-dependent variant yields query times of 113 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.

Comparing the time-dependent variant of SHARC with our approach, we observe that SHARC with an approximation value of 1.001 settles as many nodes as ALT with $K = 2.0$. However, query performance is better for SHARC due to its small computational overhead. By increasing the approximation value, computational times are slowed by almost one order of magnitude, but the solution quality merely improves. The reason for this poor performance is that SHARC uses a contraction routine which cannot bypass nodes incident to time-dependent edges. As in our scenario about half of the edges are time-dependent, the preprocessing of SHARC takes quite long (≈ 12 hours) and query performance is poor. Summarizing, ALT seems to work much better in a time-dependent scenario.

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 (3). 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 85% slower than the one with π_b^* , and the slowdown increases to 209% for $K = 1.10$ and to 286% 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 22% increase when not using π_b^*), the ratio rapidly grows until it reaches a 460% 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 ALT 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 distance queries, Fig. 1 reports the query times with respect to the Dijkstra rank¹. 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 ALT. 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 bidi-

¹For 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.

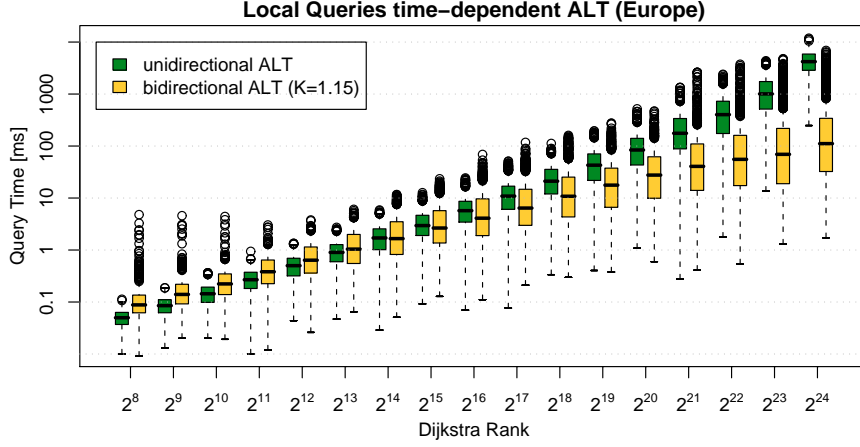


Figure 1: Comparison of uni- and bidirectional ALT using the Dijkstra rank methodology [19]. The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

rectional 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 ALT are almost as slow as the unidirectional variant.

Number of Landmarks. In static scenarios, query times of bidirectional ALT can be significantly reduced by increasing the number of landmarks to 32 or even 64 (see [7]). In order to check whether this also holds for our time-dependent variant, we recorded our algorithm’s performance using different numbers of landmarks. Tab. 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 the computed paths. Surprisingly, the number of landmarks has a very small influence on the performance of time-dependent ALT. Even worse, increasing the number of landmarks even yields larger average query times for unidirectional ALT and for bidirectional ALT 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. 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

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

	K	8 landmarks		16 landmarks		32 landmarks	
		#	settled time [ms]	#	settled time [ms]	#	settled time [ms]
uni-ALT	-	2 321 760	1 739.8	2 192 010	1 775.8	2 111 090	1 868.5
ALT	1.00	3 240 210	3 270.6	3 117 160	3 342.0	3 043 490	3 465.1
	1.10	863 526	736.5	719 769	646.5	681 836	669.7
	1.15	495 649	382.1	348 325	281.1	312 695	280.0
	1.20	389 096	286.3	241 241	180.9	204 877	170.1
	1.50	320 026	228.4	172 157	118.6	133 547	98.3
	2.00	313 448	222.2	165 650	113.0	126 847	91.1

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.

7 Conclusion and Future Work

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. 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. We have compared our algorithm to existing methods, showing that this approach for bidirectional search is able to significantly decrease computational times.

Future research will include the possibility of an initial contraction phase for a time-dependent graph, which would be useful for several purposes, and dealing with dynamic time-dependent cost functions, in order to be able to take into account unexpected changes such as those that are caused by unpredictable events. The idea of bidirectional routing on time-dependent graphs, using a time-dependent forward search and a time-independent backward search, may be applied to other static routing algorithms, in order to generalize them in a time-dependent scenario.

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] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 08)*, pages 13–26. SIAM, 2008.
- [3] I. Chabini and L. Shan. 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.
- [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] C. Daganzo. Reversibility of time-dependent shortest path problem. Technical report, Institute of Transportation Studies, University of California, Berkeley, 1998.
- [6] D. Delling. Time-Dependent SHARC-Routing. submitted for publication, 2008.
- [7] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA 2007)*, volume 4525 of *LNCS*, pages 52–65, New York, 2007. Springer.
- [8] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [9] S. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.
- [10] 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.
- [11] 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)*, volume 129 of *Lecture Notes in Computer Science*. Springer, 2006.
- [12] 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.
- [13] 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.
- [14] 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.

- [15] 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.
- [16] B. S. Kerner. *The Physics of Traffic*. Springer, Berlin, 2004.
- [17] G. Nannicini and L. Liberti. Shortest paths on dynamic graphs. *International Transactions in Operational Research*, 15:1–13, 2008.
- [18] 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.
- [19] 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.
- [20] P. Sanders and D. Schultes. Dynamic highway-node routing. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA 2007)*, volume 4525 of *LNCS*, pages 66–79, New York, 2007. Springer.
- [21] P. Sanders and D. Schultes. Engineering fast route planning algorithms. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA 2007)*, volume 4525 of *LNCS*, pages 23–36, New York, 2007. Springer.
- [22] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In W. Thomas and P. Weil, editors, *24th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, volume 4393 of *LNCS*, pages 23–36, New York, 2007. Springer.
- [23] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM Journal of Experimental Algorithms*, 10:1–30, 2005.