Bidirectional A^* Search for Time-Dependent Fast Paths

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

[1] *LIX, École Polytechnique, F-91128 Palaiseau, France*
{giacomon,liberti}@lix.polytechnique.fr
[2] *Mediamobile, 10 rue d'Oradour sur Glane, Paris, France*
giacomo.nannicini@v-trafic.com
[3] *Universität Karlsruhe (TH), 76128 Karlsruhe, Germany*
{delling,schultes}@ira.uka.de

Abstract. The computation of point-to-point shortest paths on time-dependent road networks has many practical applications, but there have been very few works that propose efficient algorithms for large graphs. One of the difficulties of route planning on time-dependent graphs is that we do not know the exact arrival time at the destination, hence applying bidirectional search is not straightforward; we propose a novel approach based on A^* with landmarks (ALT) that starts a search from both the source and the destination node, where the backward search is used to bound the set of nodes that have to be explored by the forward search. Extensive computational results show that this approach 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 suboptimal solutions.

1 Introduction

We consider the 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 departure time $\tau_0 \in T$ and a time-dependent arc weight function $c : A \times T \to \mathbb{R}_+$, find a path $p = (s = v_1, \ldots, v_k = t)$ in G such that its *time-dependent cost* $\gamma_{\tau_0}(p)$, defined recursively as follows:

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

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

for all $2 \leq i \leq k$, is minimum. We also consider a function $\lambda : A \to \mathbb{R}_+$ which has the following property:

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

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.

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 (see [6, 20, 23]).

Much less work has been undertaken on the time-dependent variant of the shortest paths problem; this problem has been first addressed by [4] (a good review of this paper can be found in [10], p. 407): Dijkstra's algorithm [9] is extended to the dynamic case through a recursion formula based on the assumption that the network $G = (V, A)$ has the FIFO property. The FIFO property is also called the *non-overtaking property*, because it basically states that if A leaves u at time τ_0 and B at time $\tau_1 > \tau_0$, B cannot arrive at v before A using the arc (u, v). The FIFO assumption is usually necessary in order to mantain an acceptable level of complexity: the TDSPP in FIFO networks is polynomially solvable [16], 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^* [14], has been adapted to work on all the previously described scenarios; an efficient version for the static case has been presented in [11], and then developed and improved in [12]. Those ideas have been used in [6] on dynamic graphs as well, while the time-dependent case on graphs with the FIFO property has been addressed in [3] and [6].

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. However, in that case SHARC cannot guarantee to find the optimal solution.

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 improvements. 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 A^* algorithm on static graphs can be described as follows. 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 has the effect of giving priority to nodes that are (supposedly) closer to 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. In [15] it is shown that A^* is equivalent to Dijkstra's algorithm on a graph with reduced costs, i.e. $w_\pi(u,v) = w(u,v) - \pi(u) + \pi(v)$.

One way to compute the potential function, instead of using Euclidean distances, is to use the concept of *landmarks*. Landmarks have first been proposed in [11]; 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 a static graph (i.e. non time-dependent), bidirectional search can be implemented, using some care in modifying the potential function so that it is consistent for the forward and backward search (see [12]); the consistency condition states 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, the potential function associated with landmarks is still a valid lower bound, even on a time-dependent graph; in [6] this idea is applied to a real road network in order to analyse the algorithm's performances, but with a unidirectional search.

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, i.e., is guaranteed to yield the smaller search space for random source-destination pairs. The best known heuristics are *avoid* and *maxCover* [13].

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.

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 $s \to t$ 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. 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. □

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.

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

1: $\overrightarrow{Q}$.insert$(s, 0)$; $\overleftarrow{Q}$.insert$(t, 0)$; $M := \emptyset$; $\mu := +\infty$; *done* := `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: $u := \overleftrightarrow{Q}$.extractMin()
8: **if** $(u = t) \wedge (\leftrightarrow = \rightarrow)$ **then**
9: *done* := `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 $(u, v) \in \overleftrightarrow{A}$ **do**
18: **if** $\leftrightarrow = \leftarrow$ **then**
19: M.insert(u)
20: **else if** $(phase = 3) \wedge (v \notin M)$ **then**
21: **continue**;
22: **if** $(v \in \overleftrightarrow{Q})$ **then**
23: **if** $u.\text{dist}^{\leftrightarrow} + c(u, v, u.\text{dist}^{\leftrightarrow}) < v.\text{dist}^{\leftrightarrow}$ **then**
24: $\overleftrightarrow{Q}$.decreaseKey$(v, u.\text{dist}^{\leftrightarrow} + c(u, v, u.\text{dist}^{\leftrightarrow}) + \overleftrightarrow{\pi}(v))$
25: **else**
26: $\overleftrightarrow{Q}$.insert$(v, u.\text{dist}^{\leftrightarrow} + c(u, v, u.\text{dist}^{\leftrightarrow}) + \overleftrightarrow{\pi}(v))$
27: **return** $t.\text{dist}^{\rightarrow}$

5 Improvements

The basic version of the algorithm can be enhanced by making use of the following results.

5.1 Proposition

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

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, i.e. the backward search does not have to compute shortest paths. This means that we can tighten the bounds used by the backward search, as long as they are still valid lower bounds, even if doing so would result in an A^* backward search that computes suboptimal distances.

5.2 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, $d(s, v, \tau_0) + \pi_f(v) - \pi_f(w) \leq d(s, w, \tau_0)$.

Let v be as in Prop. 5.2, and w a node which has not been settled by the forward search. Prop. 5.2 suggests that we can use

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

as a lower bound to $d(s, w, \tau_0)$ during the backward search. However, to prove the algorithm's correctness when using π_b^* we must 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 $\pi_f(t)$ and the current distance from the source node, both for the forward search.

6 Experiments

In this section, we present an extensive experimental evaluation of our time-dependent ALT algorithm. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.1. 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 [11], 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. Therefore, we have to use artificially generated costs. In order to model the time-dependent costs on each arc, we developed an 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 [17].

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 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.

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).

			Error			Query			
				relative		# settled nodes			time
input	method	K	rate	avg	max	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
	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	2 784 540	3 117 160	3 399.3
		1.02	1.0%	0.003%	1.13%	125 068	2 154 900	2 560 370	2 723.3
		1.05	4.0%	0.029%	4.93%	125 068	1 333 220	1 671 630	1 703.6
		1.10	18.7%	0.203%	8.10%	125 068	549 916	719 769	665.1
		1.13	30.5%	0.366%	12.63%	125 068	340 787	447 681	385.5
		1.15	36.4%	0.467%	13.00%	125 068	265 328	348 325	287.3
		1.20	44.7%	0.652%	18.19%	125 068	183 899	241 241	185.3
		1.30	48.2%	0.804%	23.63%	125 068	141 358	186 267	134.6
		1.50	48.8%	0.844%	25.70%	125 068	130 144	172 157	121.9
		2.00	48.9%	0.886%	48.86%	125 068	125 071	165 650	115.7
USA	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

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 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 neworks with a maximum recorded relative error of 13% (on the European road network, while the corresponding figure is 9.98% for the US instance). 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 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 [6]. 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 [6]), while our time-dependent variant yields query times of 115 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.

Local Queries. For random queries, our bidirectional ALT algorithm (with $K = 1.15$) is roughly 6.7 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 rank[4]. 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

[4] 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.

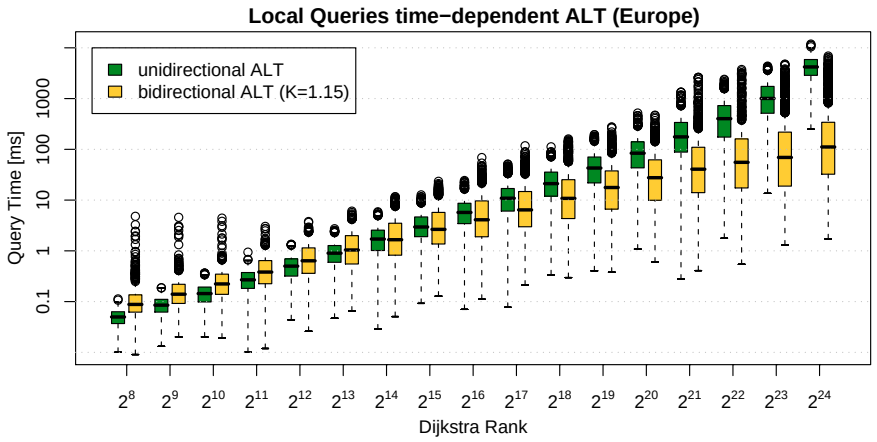

Fig. 1. Comparison of uni- and bidirectional ALT using the Dijkstra rank methodology [19]. The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which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

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 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 [6]). 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. 2 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

Table 2. Performance of uni- and bidirectional ALT 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	-	2 321 760	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	3 399.3	3 043 490	3 465.1
	1.10	863 526	736.5	719 769	665.1	681 836	669.7
	1.15	495 649	382.1	348 325	287.3	312 695	280.0
	1.20	389 096	286.3	241 241	185.3	204 877	170.1
	1.50	320 026	228.4	172 157	121.9	133 547	98.3
	2.00	313 448	222.2	165 650	115.7	126 847	91.1

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 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 algorithm engineering issues such as the balancing of the forward and backward search, and the update of the available upper bound on the optimal solution cost. 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)*. SIAM, 2008. to appear.

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 and D. Wagner. Landmark-based routing in dynamic graphs. In Demetrescu [7].
7. C. Demetrescu, editor. *WEA 2007 — Workshop on Experimental Algorithms*, volume 4525 of *LNCS*, New York, 2007. Springer.
8. C. Demetrescu, R. Sedgewick, and R. Tamassia, editors. *Proceedings of the 7th Workshop on Algorithm Engineering and Experimentation*. SIAM, 2005.
9. E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
10. S. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.
11. A. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. In *SODA 2005*. SIAM, 2005.
12. A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In Demetrescu et al. [8].
13. A. Goldberg and R. Werneck. An efficient external memory shortest path algorithm. In Demetrescu et al. [8], pages 26–40.
14. 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.
15. 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.
16. 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.
17. B. S. Kerner. *The Physics of Traffic*. Springer, Berlin, 2004.
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, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.
20. P. Sanders and D. Schultes. Dynamic highway-node routing. In Demetrescu [7], pages 66–79.
21. P. Sanders and D. Schultes. Engineering fast route planning algorithms. In Demetrescu [7], pages 23–36.
22. D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In W. Thomas and P. Weil, editors, *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 Algorithmics*, 10:1–30, 2005.

A Appendix

Proof of Thm. 4.2

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. □

A.1 Theorem

Algorithm 1 is correct if, when a node u is settled by the backward search, then its key is smaller or equal to $d(s,u,\tau_0) + d(u,t,d(s,u,\tau_0))$.

Proof. Let Q be the backward search queue, let $\text{key}(u)$ be the key for the backward search of node u, let β 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, 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 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. □

Proof of Prop. 5.1

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 is settled by the backward search, $d_b(v) \leq d(v,t,d(s,v,\tau_0)) \ \forall \tau_0 \in T$. By Thm. A.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. □

A.2 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 $d(s, u, \tau_0) + d_\lambda(\ell, t) - d_\lambda(\ell, u) \leq d(s, u, \tau_0) + d_\lambda(\ell, t) - d_\lambda(\ell, v) + \lambda(u, v) \leq d(s, v, \tau_0) + \pi_f(v)$. □

Proof of Prop. 5.2

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. 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 A.2, if we examine the nodes $u_1 = u, u_2, \ldots, u_k = w$ on the shortest path from s to w with departure time τ_0, we have that $d(s, u_1, \tau_0) + \pi_f(u_1) \leq \cdots \leq d(s, u_k, \tau_0) + \pi_f(u_k)$, from which our statement follows. □

A.3 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. □

A.4 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 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. 5.2 and Thm. A.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 A.3, 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.