Landmark-Based Routing in Dynamic Graphs[⋆]

Daniel Delling and Dorothea Wagner

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{delling,wagner}@ira.uka.de

Abstract. Many speed-up techniques for route planning in static graphs exist, only few of them are proven to work in a dynamic scenario. Most of them use preprocessed information, which has to be updated whenever the graph is changed. However, goal directed search based on landmarks (ALT) still performs correct queries as long as an edge weight does not drop below its initial value. In this work, we evaluate the robustness of ALT with respect to traffic jams. It turns out that—by increasing the efficiency of ALT—we are able to perform fast (down to 20 ms on the Western European network) random queries in a dynamic scenario without updating the preprocessing as long as the changes in the network are moderate. Furthermore, we present how to update the preprocessed data without any additional space consumption and how to adapt the ALT algorithm to a time-dependent scenario. A time-dependent scenario models predictable changes in the network, e.g. traffic jams due to rush hour.

1 Introduction

Computing shortest paths in graphs $G = (V, E)$ is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes. In general, Dijkstra's algorithm [1] finds the shortest path between a given source s and target t. Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques exist [2] yielding faster query times for typical instances, e.g., road or railway networks. Recent research [3,4] even made the calculation of the distance between two points in road networks of the size of Europe a matter of microseconds. Thus, at least for road networks, shortest path computation seems to be solved.

However, most of the existing techniques require a *static* graph, i.e. the graph is known in advance and does not change between two shortest path computations. A more realistic scenario is a dynamic one: new roads are constructed and closed or traffic jams occur. Furthermore, we often know in advance that the motorways are crowded during rush hour. In this work, we adapt the known technique of goal directed search based on landmarks—called ALT [5]—to these dynamic scenarios.

[⋆] Partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

1.1 Related Work

We focus on related work on dynamic speed-up techniques. For static scenarios, see [2] for an overview. An adaption of DIJKSTRA's algorithm to a scenario where travel times depend on the daytime, the *time-dependent* scenario, can be found in [6]. Throughout the paper, we distinguish time-dependent and *time-independent* scenarios. For the latter, edge weights are not dependent on the daytime. A classical speed-up technique is *bidirectional* DIJKSTRA which also starts a search from the target. As bidirectional DIJKSTRA uses no preprocessing, it can be used in a time-independent dynamic scenario without any effort. However, its adaption to a time-dependent scenario is more complicated as the arrival time is unknown in such a scenario.

Goal directed search, also called A^* [7], pushes the search towards a target by adding a potential to the priority of each node. The usage of Euclidean potentials requires no preprocessing. The ALT algorithm, introduced in [5], obtains the potential from the distances to certain landmarks in the graph. Although this approach requires a preprocessing step, it is superior with respect to search space and query times. Goldberg and Harrelson state that ALT may work well in a dynamic scenario. In this work, we persue and advance their ideas. In [8], A^* using Euclidean potentials is adapted to a time-dependent scenario.

Geometric containers [9] attach a label to each edge that represents all nodes to which a shortest path starts with this particular edge. A dynamization has been published in [9] yielding suboptimal containers if edge weights decrease. In [10], ideas from highway hierarchies [11] and overlay graphs [2] are combined yielding very good query times in dynamic road networks.

Closely related to dynamic shortest path computation are the Single-Source and All-Pair-Shortest-Path problems. Both of them have been studied in a dynamic scenario [12,13].

1.2 Overview

In Section 2 we review the ALT algorithm, introduced in [14] and enhanced in [15]. We improve the original algorithm by storing landmark data more efficiently. In Section 3, we briefly discuss how to model traffic in predictable and unexpected cases. The adaption of ALT to our models from Section 3 is located in Section 4. First, we show how to update the preprocessing efficiently *without* any additional requirements of data. The update is based on dynamic shortest path trees that can be reconstructed from the graph with data provided by ALT. However, as already mentioned in [14], it turns out that for the most common type of update, i.e., traffic jams, the update of the preprocessing needs not be done in order to keep queries correct. Finally, we are able to adapt a unidirectional variant of ALT to the time-dependent model. An extensive experimental evaluation can be found in Section 5, proving the feasibility of our approach. There, we focus on the performance of ALT with no preprocessing updates when traffic jams occur. Section 6 concludes this work by a summary and possible future work.

2 Goaldirected Search Based on Landmarks

In this section, we explain the known ALT algorithm [14]. In general, the algorithm is a variant of bidirectional A^* search [7] in combination with landmarks. We follow the implementation presented in [15] enriched by implementation details that increase efficiency.

The search space of DIJKSTRA's algorithm can be interpreted as a circle around the source. By adding a 'good' potential $\pi : V \to \mathbb{R}$ to the priority of each node, the order in which nodes are removed from the priority queue is altered in such a way that nodes lying on a shortest path to the target yield a low priority. In [7], it is shown that this technique—known as A^*—is equivalent to DIJKSTRA's algorithm on a graph with *reduced costs*, formally $w_\pi(u, v) = w(u, v) - \pi(u) + \pi(v)$. Since DIJKSTRA's algorithm works only on nonnegative edge costs, not all potentials are allowed. We call a potential π *feasible* if $w_\pi(u, v) \geq 0$ for all $(u, v) \in E$. The distance from each node v of G to the target t is the distance from v to t in the graph with reduced edge costs minus the potential of t plus the potential of v. So, if the potential $\pi(t)$ of the target t is zero, $\pi(v)$ provides a *lower bound* for the distance from v to the target t. There exist several techniques [16] to obtain feasible potentials using the layout of a graph. The ALT algorithm uses a small number of nodes—so called *landmarks*—and the triangle inequality to compute feasible potentials. Given a set $S \subseteq V$ of landmarks and distances $d(L, v), d(v, L)$ for all nodes $v \in V$ and landmarks $L \in S$, the following triangle inequations hold: $d(u, v) + d(v, L) \geq d(u, L)$ and $d(L, u) + d(u, v) \geq d(L, v)$. Therefore, $\underline{d}(u, t) := \max_{L \in S} \max\{d(u, L) - d(t, L), d(L, t) - d(L, u)\}$ provides a lower bound for the distance $d(u, t)$ and, thus, can be used as a potential for u.

The quality of the lower bounds highly depends on the quality of the selected landmarks. Thus, several selection strategies exist. To this point, no technique is known for picking landmarks that yield the smallest search space for random queries. Thus, several heuristics exist; the best are *avoid* and *maxCover* [15].

As already mentioned, ALT is a bidirectional variant of A^*. In general, the combination of A^* and bidirectional search is not that easy as it seems. Correctness can only be guaranteed if π_f—the potential for the forward search—and π_r—the potential for the backward search—are *consistent*. This means $w_{\pi_f}(u, v)$ in G is equal to $w_{\pi_r}(v, u)$ in the reverse graph. We use the variant of an average potential function [7] defined as $p_f(v) = (\pi_f(v) - \pi_r(v))/2$ for the forward and $p_r(v) = (\pi_r(v) - \pi_f(v))/2 = -p_f(v)$ for the backward search. Note, that π_f provides better potentials than p_f. Moreover, for a bidirectional variant, the stopping criterion has to be altered: Stop the search if the sum of minimum keys in the forward and the backward queue exceeds $\mu + p_f(s)$, where μ represents the tentative shortest path length.

Improved Efficiency. One downside of ALT seemed to be its low efficiency. In [17], a reduction of factor 44 in search space only leads to a reduction in query times of factor 21. By storing landmark data more efficiently, this gap can be reduced. First, we sort the landmarks by ID in ascending order. The distances from and to a landmark are stored as one 64-bit integer for each node and landmark: The upper 32 bits refer to the 'to' distance and the lower to the 'from' distance. Thus, we initialize a 64-bit vector of size $|S| \cdot |V|$. Both distances of node number $i \in [0, |V| - 1]$ and landmark number $j \in [0, |S| - 1]$ are stored at position $|S| \cdot i + j$. As a consequence, when computing the potential for given node n, we only have one access to the main memory in most times.

3 Modeling Traffic

In the following, we briefly discuss how to model several scenarios of updates due to traffic in road networks. We cover unexpected and predictable updates.

Dynamic Updates. The most common updates of transit times in road networks are those of *traffic jams*. This can just be slight increases for many roads due to rush hour. Nevertheless, increases of higher percentage can happen as well. In the worst case, routes may be closed completely. Currently, traffic reports concentrate on motorways. Nevertheless, with new technologies like car-to-car communication [18] information for all roads will be available. This will lead to scenarios where updates happen quite frequently for all kinds of roads. Analyzing the type of updates one may notice that transit times may increase (high traffic) and may decrease afterwards but will not drop below the transit times of an 'empty' road. Overall, a dynamic speed-up technique for shortest path computation should handle this kind of updates very well and very fast. In the ideal case, the technique should not need any update at all.

Like traffic jams, *construction sites* increase transit times when they are installed and decrease them when the work is done. In most cases, the transit times do not fall below the original value. So, for this case, an updating routine for handling traffic jams can also handle construction sites.

Another type of change in the structure of a network is the construction or demolition of roads. This can be modeled by insertions or deletions of edges and/or nodes. On the one hand, these type of updates are known in advance and on the hand, they happen not very often. Thus, a dynamic algorithm should be capable of handling these updates but the performance on these updates is not that important like traffic jams.

Normally, transit times of roads are calculated by the average speed on these roads. But profiles of clients differ: some want to be as fast as possible, others want to save fuel. Furthermore, the type of vehicle has an impact on travel times (a truck is slower than a sports car) and even worse, some roads are closed to specific types of vehicles.

Time-Dependency. Most of the changes in transit times are predictable. We know in advance that during rush hour, transit times are higher than at night. And we know that the motorways are crowded at the beginning and end of a holiday. This scenario can be modelled by a *time-dependent* graph [8] which assigns several weights to a specific edge. Each weight represents the travel time at a certain time. As a consequence, the result of an s-t query depends on the time of departure from s. In [19], it is shown that time-dependent routing gets more complicated if the graph is not time consistent. Time consistency means that for each edge (u, v) a delayed departure from u does *not* yield a earlier arrival in v. Throughout this paper, we focus on time consistent graphs. Note that our models from Section 3 can be used in a time-dependent scenario by updating some or all of the weights assigned to an edge.

4 Dynamization

In this section, we discuss how the preprocessing of the ALT algorithm can be updated efficiently. Furthermore, we discuss a variant where the preprocessing has to be updated only very few times. However, this approach may lead to a loss in performance. At the end of this section we introduce a time-dependent variant of the unidirectional ALT algorithm.

4.1 Updating the Preprocessing

The preprocessing of ALT consists of two steps: the landmark selection and calculating the distance labels. As the selection of landmarks are heuristics, we settle for *static* landmarks, i.e., we do not reposition landmarks if the graph is altered. The update of the distance labels can be realized by dynamic shortest path trees. For each landmark, we store two trees: one for the forward edges, one for the backward edges. Whenever an edge is altered we update the tree structure including the distance labels of the nodes. In the following we discuss a memory efficient implementation of dynamic shortest path trees. The construction of a tree can be done by running a complete DIJKSTRA from the root of the tree.

Updating Shortest Path Trees. In [12] the update of a shortest path tree is discussed. The approach is based on a modified DIJKSTRA, trying to identify the regions that have to be updated after a change in edge weight. Therefore, a tree data structure is used in order to retrieve all successors and the parent of a node quickly. As road graphs are sparse we do not need to store any additional information to implement these operations. The successors of n can be determined by checking for each target t of all outgoing edges e whether $d(n) + w(e) = d(t)$ holds. If it holds, t can be interpreted as successor of n. Analogously, we are able to determine the parent of a node n: Iterate all sources s of the incoming edges e of n. If $d(s)+w(e) = d(n)$ holds, s is the parent of n. This implementation allows to iterate all successors of n in $O(\delta)$ where δ is the degree of n. The parent of n can be found in $O(\delta)$ as well. Note that we may obtain a different tree structure than rerunning a complete DIJKSTRA, but as we are only interested in distance labels, this approach is sufficient for the correctness of ALT.

The advantage of this approach is memory consumption. Keeping all distance labels for 16 landmarks on the road network of Western Europe in memory already requires about 2.2 GB of RAM (32 trees with 18 million nodes, 32 bit per node). Every additional pointer would need an additional amount of 2.2 GB. Thus, more advanced tree structures (1, 2, or 4 pointers) lead to an overhead that does not seem worth the effort.

4.2 Two Variants of the Dynamic ALT Algorithm

Eager Dynamic ALT. In the previous section we explained how to update the preprocessed data of ALT. Thus, we could use the update routine whenever the graph is altered. In the following, we call this variant of the dynamic ALT the *eager dynamic* version.

Lazy Dynamic ALT. However, analyzing our dynamic scenarios from Section 3 and the ALT algorithm from Section 2 we observe two important facts. On the one hand, ALT-queries only lose correctness if the potential of an edge results in a negative edge cost in the reduced graph. This can only happen if the cost of the edge drops below the value during preprocessing. On the other hand, for the most common update type—traffic jams—edge weights may increase and decrease but do not drop below the initial value of empty roads. Thus, a potential computed on the graph without any traffic stays feasible for those kinds of updates even when not updating the distances from and to all landmarks. Due to this observation, we may do the preprocessing for empty roads and use the obtained potentials even though an edge is perturbed. In [14], this idea was stated to be semi-dynamic, allowing only increases in edge weights. Nevertheless, as our update routine does not need any additional information, we are able to handle all kinds of updates. Our *lazy dynamic* variant of ALT leaves the preprocessing untouched unless the cost of an edge drops below its initial value.

This approach may lead to an increase in search space. If an edge e on the shortest path is increased without updating the preprocessing, the weight of e is also increased in G', the graph with reduced costs. Thus, the length of the shortest path increases in G'. So, the search stops later because more nodes are inserted in the priority queue (cf. the stopping criterion in Section 2). However, as long as the edges are not on the shortest path of a requested query the search space does not increase. More precisely, the search space may even decrease because nodes 'behind' the updated edge are inserted later into the priority queue.

4.3 The Time-Dependent ALT Algorithm

In time-independent scenarios, ALT is implemented as bidirectional search. But in time-dependent scenarios, a backward search is prohibited. Thus, we have to use an unidirectional variant of ALT that only performs a forward search. As a consequence, we may use the known stopping criterion of Dijkstra's algorithm: Stop the search when the target node is taken from the priority queue. Furthermore, we may use the potential π_f instead of the average potential p_f yielding better lower bounds (Section 2).

Based on the ideas from 4.2, we can adapt the unidirectional ALT algorithm to the time-dependent scenario. When doing the preprocessing, we use the minimum weight of each edge to compute the distance labels. It is obvious that we obtain a feasible potential. The time-dependent ALT algorithm works analogously to an unidirectional ALT but calculates the estimated departure time from a node in order to obtain the correct edge weight. We alter the priority of each node by adding the potential computed during preprocessing.

Note that the time-dependent ALT algorithm also works in a dynamic time-dependent scenario. Using the same arguments from Section 4.2, the algorithm still performs accurate queries as long as an edge weight does not drop below the value used during the preprocessing. If this happens, the distance labels can be updated using the routine from Section 4.1.

5 Experiments

Our experimental evaluation was done on one CPU of a dual AMD Opteron 252 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.

As inputs, we use the road map of Western Europe, provided by PTV AG for scientific use, and the US network taken from the TIGER/Line Files. The former graph has approximately 18 million nodes and 42.6 million edges, where edge lengths correspond to travel times. The corresponding figures for the USA are 23.9 million and 58.3 million, respectively. Each edge belongs to one of four main categories representing motorways, national roads, local streets, and urban streets. The European network has a fifth category representing rural roads. For updates, we do not consider these rural roads. In general, we measure a *low perturbation* of an edge by an increase of its weight by factor 2. For a *high perturbation* we use an increase by factor 10. In the following, we identify the unidirectional, bidirectional and time-dependent ALT algorithm by *uni*-ALT, ALT and *time*-ALT, respectively. The number of landmarks is indicated by a number after the algorithm, e.g. ALT-16 for 16 landmarks.

The Static ALT Algorithm. In order to compare our ALT implementations in a dynamic scenario, we report the performance of the bi- and unidirectional ALT algorithm in a static scenario. We evaluate different numbers of landmarks with respect to preprocessing, search space and query times performing 10 000 uniformly distributed random s-t queries. Due to memory requirements we used *avoid* for selecting 32 and 64 landmarks. For less landmarks, we used the superior *maxCover* heuristic. Table 1 gives an overview.

We see for bidirectional ALT that doubling the number of landmarks reduces search space and query times by factor 2, which does not hold for the unidirectional variant. This is due to the fact that goal direction works best on motorways as these roads mostly have reduced costs of 0 in the reduced graph. In the unidirectional search, one has to leave to motorway in order to reach the target. This drawback cannot be compensated by more landmarks.

Table 1. Preprocessing, search space, and query times of uni- and bidirectional ALT and Dijkstra based on 10 000 random s-t queries. The column *dist.* refers to the time needed to recompute all distance labels from scratch.

		Preprocessing			Query Unidir.		Query Bidir.	
		time	space	dist.	# settled	time	# settled	time
graph	algorithm	[min]	[MB]	[min]	nodes	[ms]	nodes	[ms]
Europe	Dijkstra	0.0	0	0.0	9 114 385	5591.6	4 764 110	2713.2
	ALT-8	26.1	1 100	2.8	1 019 843	391.6	163 776	127.8
	ALT-16	85.2	2 200	5.5	815 639	327.6	74 669	53.6
	ALT-32	27.1	4 400	11.1	683 566	301.4	40 945	29.4
	ALT-64	68.2	8 800	22.1	604 968	288.5	25 324	19.6
USA	Dijkstra	0.0	0	0.0	11 847 523	6780.7	7 345 846	3751.4
	ALT-8	44.5	1 460	3.4	922 897	329.8	328 140	219.6
	ALT-16	103.2	2 920	6.8	762 390	308.6	180 804	129.3
	ALT-32	35.8	5 840	13.6	628 841	291.6	109 727	79.5
	ALT-64	92.9	11 680	27.2	520 710	268.8	68 861	48.9

Comparing uni- and bidirectional ALT, one may notice that the time spent per node is significantly smaller than for uni-ALT. The reason is the computational overhead for performing a bidirectional search. A reduction in search space of factor 44 (USA, ALT-16) yields a reduction in query time of factor 29. This is an overhead of factor 1.5 instead of 2.1, suggested by the figures in [15], deriving from our more efficient storage of landmark data (cf. Section 2).

Client profiles. As we do not consider repositioning landmarks, we only have to recompute all distance labels by rerunning a forward and backward Dijkstra from each landmark whenever the client profile changes. With this strategy, we are able to change a profile in 5.5 minutes when using 16 landmarks on the European network.

Updating the Preprocessing. Before testing the lazy variant of dynamic ALT, we evaluate the time needed for updating all distance labels. Note, that even the lazy variant has to update the preprocessing sometimes. With the obtained figures we want to measure the trade-off for which types of perturbations the update of the preprocessing is worth the effort. Figure 1 shows the time needed for updating all 32 trees needed for 16 landmarks if an edge is increased or decreased by factor 2 and 10. We distinguish the different types of edges.

We observe that updating the preprocessing if an important edge is altered is more expensive than the perturbation of other road types. This is due to the fact that motorway edges have many descendants within the shortest path trees. Thus, more nodes are affected by such an update. But the type of update has nearly no impact on the time spent for an update: neither how much an edge is increased nor whether an edge is increased or decreased. For almost all kind of updates we observe high fluctuations in update time. Very low update times are due to the fact that the routine is done if an increased edge is not a tree

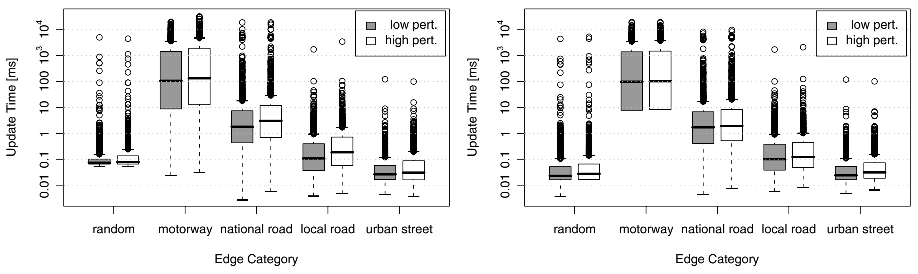

Fig. 1. Required time for updating the 32 shortest path trees needed for 16 landmarks on the European instance. The figure on the left shows the average runtime of increasing one edge by factor 2 (grey) and by 10 (white) while the right reports the corresponding values for decrementing edges. For each category, the figures are based on 10 000 edges.

edge or a decreased edge does not yield a lower distance label. Outliers of high update times are due to the fact that not only the type of the edge has an impact on the importance for updates: altering a urban street being a tree edge near a landmark may lead to a dramatic change in the structure of the tree of this landmark.

Lazy Dynamic ALT. In the following, we evaluate the robustness of the lazy variant of ALT with respect to network changes. Therefore, we alter different types and number of edges by factor 2 and factor 10.

Edge Categories. First, we concentrate on different types of edge categories. Table 2 gives an overview of the performance for both dynamic ALT variants if 1 000 edges are perturbed before running random queries.

We see that altering low-category edges has nearly no impact on the performance of lazy ALT. This is independent of the level of increase. As expected, altering motorway edges yields a loss in performance. We observe a loss of 30–45% for Europe and 15–19% for the US if the level of increase is moderate (factor 2). The situation changes for high perturbation. For Europe, queries are 3.5–5.5 times slower than in the static case (cf. Table 1), depending on the number of landmarks. The corresponding figures for the US are 1.8–2.3. Thus, lazy ALT is more robust on the US network than on the European. The loss in performance is originated from the fact that for most queries, unperturbed motorways on the shortest path have costs of 0 in the reduced graph. Thus, the search stops later if these motorways are perturbed yielding a higher search space (cf. Section 4.2). Nevertheless, comparing the query times to a bidirectional Dijkstra, we still gain a speed-up of above 10. Combining the results from Figure 1 with the ones from Table 2, we conclude that updating the preprocessing has no advantage. For motorways, updating the preprocessing is expensive and altering other types of edges has no impact on the performace of lazy ALT.

Number of Updates. In Table 2, we observed that the perturbation of motorways has the highest impact on the lazy dynamic variant of ALT. Next, we change

Table 2. Search space and query times of lazy dynamic ALT algorithm performing 10 000 random *s*-*t* queries after 1 000 edges of a specific category have been perturbed by factor 2. The figures in parentheses refer to increases by factor 10. The percentage of affected queries (the shortest path contains an updated edge) is given in column number 3.

			lazy ALT-16				lazy ALT-32			
graph	road type	aff.[%]	# settled nodes		increase [%]		# settled nodes		increase [%]	
EUR	All roads	7.5	74 700	(77 759)	0.0	(4.1)	41 044	(43 919)	0.2	(7.3)
	urban	0.8	74 796	(74 859)	0.2	(0.3)	40 996	(41 120)	0.1	(0.4)
	local	1.5	74 659	(74 669)	0.0	(0.0)	40 949	(40 995)	0.0	(0.1)
	national	28.1	74 920	(75 777)	0.3	(1.5)	41 251	(42 279)	0.7	(3.3)
	motorway	95.3	97 249	(265 472)	30.2	(255.5)	59 550	(224 268)	45.4	(447.7)
USA	All roads	3.3	181 335	(181 768)	0.3	(0.5)	110 161	(110 254)	0.4	(0.5)
	urban	0.1	180 900	(180 776)	0.1	(0.0)	109 695	(110 108)	0.0	(0.3)
	local	2.6	180 962	(181 068)	0.1	(0.1)	109 873	(109 902)	0.1	(0.2)
	national	25.5	181 490	(184 375)	0.4	(2.0)	110 553	(112 881)	0.8	(2.9)
	motorway	94.3	207 908	(332 009)	15.0	(83.6)	130 466	(247 454)	18.9	(125.5)

the number of perturbed motorways. Table 3 reports the performance of lazy dynamic ALT when different numbers of motorways are increased by factor 2 and factor 10, respectively, before running random queries on Europe.

For perturbations by factor 2, we observe almost no loss in performance for less than 500 updates, although up to 87% of the queries are affected by the perturbation. Nevertheless, 2 000 or more perturbed edges lead to significant decreases in performance, resulting in query times of about 0.5 seconds for 10 000 updates. Note that the European network contains only about 175 000 motorway edges. As expected, the loss in performance is higher when motorway edges are increased by factor 10. For this case, up to 500 perturbations can be compensated well. Comparing slight and high increases we observe that the lazy variant can

Table 3. Search space and query times of the dynamic ALT algorithm performing 10 000 random *s*-*t* queries after a variable number of motorway edges have been increased by factor 2. The figures in parentheses refer to increases by factor 10. The percentage of affected queries (the shortest path contains an updated edge) is given in column 2.

		lazy ALT-16				lazy ALT-32			
#edges	aff.[%]	# settled nodes		increase [%]		# settled nodes		increase [%]	
100	39.9	75 691	(91 610)	1.4	(22.7)	41 725	(56 349)	1.9	(37.6)
200	64.7	78 533	(107 084)	5.2	(43.4)	44 220	(69 906)	8.0	(70.7)
500	87.1	86 284	(165 022)	15.6	(121.0)	50 007	(124 712)	22.1	(204.6)
1 000	95.3	97 249	(265 472)	30.2	(255.5)	59 550	(224 268)	45.4	(447.7)
2 000	97.8	154 112	(572 961)	106.4	(667.3)	115 111	(531 801)	181.1	(1198.8)
5 000	99.1	320 624	(1 286 317)	329.4	(1622.7)	279 758	(1 247 628)	583.3	(2947.1)
10 000	99.5	595 740	(2 048 455)	697.8	(2643.4)	553 590	(1 991 297)	1252.0	(4763.3)

compensate four times more updates, e.g. 500 increases by factor 10 yield almost the same loss as 2 000 updates by factor 2.

The number of landmarks has almost no impact on the performance if more than 5 000 edges are perturbed. This is due to the fact that for almost all motorways the landmarks do not yield good reduced costs. We conclude that the lazy variant cannot compensate such a high degree of perturbation.

Comparing Lazy and Eager Dynamic ALT. Table 2 shows that lazy ALT-32 yields an increase of 40% in search space for random queries on the European network with 1 000 low perturbed motorway edges. In order to obtain a more detailed insight for which types of queries these differences are originated from, Figure 2 reports the query times of eager and lazy ALT-32 with respect to the Dijkstra rank[1] (of the target node) in this scenario.

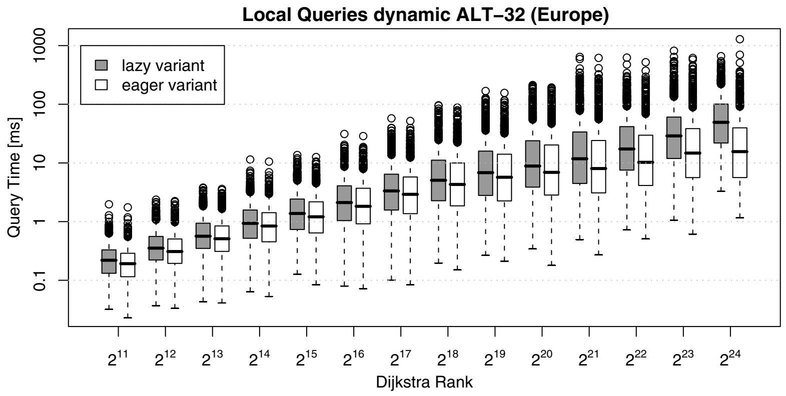

Fig. 2. Comparison of query times of the lazy and eager dynamic variant of ALT using the Dijkstra rank methodology. The queries were run after 1 000 motorways were increased by factor 2. The results are represented as box-and-whisker plot. Outliers are plotted individually.

Query performance varies so heavily that we use a logarithmic scale. For each rank, we observe queries performing 20 times worse than the median. This is originated from the fact that for some queries no landmark provides very good lower bounds resulting in significantly higher search spaces. Comparing the eager and lazy dynamic version, we observe that query times differ only by a small factor for Dijkstra ranks below 2^{20}. However, for 2^{24}, the eager version is about factor 5 faster than the lazy one. This is due to the fact that those types of queries contain a lot of motorways and most of the jammed edges are used. The eager version yields a good potential for these edge while the lazy does

[1] For an s-t query, the Dijkstra rank of node v is the number of nodes inserted in the priority queue before v is reached. Thus, it is a kind of distance measure.

not. We conclude that lazy ALT is more robust for small range queries than for long distance requests. Note that in a real-world scenario, you probably do not want to use traffic information that is more than one hour away from your current position. As the ALT algorithm provides lower bounds to *all* positions within the network, it is possible to ignore traffic jams sufficiently without any additional information.

Time-Dependent ALT. Our final experiments cover the time-dependent scenario, in which bidirectional search is prohibited. Thus, we compare time-ALT with a time-dependent variant of Dijkstra's algorithm [6]. This variant of Dijkstra works like the normal one but calculates the departure time from a node in order to use the correct edge weight. Our current implementation of time-dependency assigns 24 different transit times to each edge, representing the travel time at each hour of a day. Again, we interpret the initial values as empty roads and add transit times according to rush hours. Table 4 gives an overview of the performance on the European network for different scenarios of traffic during the day. We study three models differing in how much transit time is added to *all* edges during the rush hours. The first (high traffic) increases the transit time on all roads by factor 3 during peak hours. The low traffic scenario uses increases of factor 1.5. For comparison, the no traffic scenario uses the same (initial) edge weight for all times of the day. Our models are inspired by [8].

Table 4. Search space and query times of time-dependent ALT and Dijkstra performing 10 000 random *s*-*t* queries for different types of expected traffic in a time-dependent scenario. As input, the European network is used.

	no traffic		low traffic		high traffic	
algorithm	# settled	time [ms]	# settled	time [ms]	# settled	time[ms]
Dijkstra	9 029 972	8 383.2	9 034 915	8 390.6	9 033 100	8 396.1
time-ALT-16	794 622	443.7	1 969 194	1543.3	3 130 688	2 928.3

We observe a speed-up of approximately factor $3 - 5$ towards Dijkstra's algorithm, depending on the scenario. This relatively low speed-up in contrast to a speed-up of factor 50 for time-independent bidirectional ALT-16 towards bidirectional Dijkstra is due to two facts. On the one hand, the unidirectional ALT algorithm performs much worse than the bidirectional variant (see Table 1). On the other hand, lower bounds are much worse than in a time-independent scenario because an edge increased by factor 2 during rush hour counts like a perturbed edge in the time-independent scenario. As lazy ALT cannot compensate a very high degree of perturbation and we apply our traffic model to all edges including motorways, these figures are not counterintuitive.

Comparing Table 1 and 4, our time-dependent variant is 30% slower than the time-independent unidirectional ALT. This is due to overhead in computing the estimated departure time from each node.

Table 5. Comparison of lazy ALT and DynHNR [10]. 'Space' indicates the additional overhead per node. We report the performance of static and dynamic random queries. For the latter the average search space is reported after 10 and 1000 edges have been increased by factor 2 and—in parentheses—factor 10. Note that the tests for DynHNR were performed on a slightly different maschine.

	preprocessing		static queries		dynamic queries			
	time	space	time	#settled	#settled nodes			
method	[min]	[B/node]	[ms]	nodes	10 updates		1000 updates	
lazy ALT-16	85	128	53.6	74 441	74 971	(75 501)	97 123	(255 754)
lazy ALT-32	27	256	29.4	40 945	41 060	(43 865)	59 550	(224 268)
lazy ALT-64	62	512	19.6	25 324	26 247	(26 901)	42 930	(201 797)
DynHNR	18	32	1.2	1 414	2 385	(8 294)	204 103	(200 465)

Comparison to Dynamic Highway-Node Routing. Analyzing the figures from Table 5, it turns out that an approach based solely on landmarks cannot compete with Dynamic Highway-Node Routing [10] as long as the number of perturbed edges stay little. However, the situation changes if more than 1000 edges are updated. For factor-2 perturbations, ALT yields lower search spaces than DynHNR and for factor-10 perturbations both techniques are very close to each other. Nevertheless, with respect to space requirements, DynHNR is superior to ALT, and the preprocessing of DynHNR can be updated more efficiently than the preprocessing of ALT.

6 Discussion

We evaluated adaptions of ALT to dynamic scenarios covering predictable and unexpected changes. In a time-independent scenario, a variant not updating the preprocessing loses almost no performance as long as the number of perturbed roads stays moderate. When using 64 landmarks, random queries are done in 20 ms on the European network and in 50 ms on the US network. However, for some types of updates the preprocessing can be updated in moderate time without storing any additional data.

Analyzing the dynamic scenarios, the time-dependent model seems to be superior to the time-independent model. Especially for long range queries, updates may occur during the traversal of the shortest path. While this can be compensated by rerunning a query from the current position, one cannot take into account jams that are on the route but probably will have disappeared as soon as you reach the critical section.

Summarizing, landmark based routing yields good query times in dynamic scenarios. Furthermore, landmarks harmonize well with other techniques like reach [15], highway hierarchies [11], or even transit nodes [4]. As the dynamization of ALT comes for free, adding landmarks to other techniques in dynamic scenarios may be worth focusing on.

Acknowledgments. We would like to thank Marco Gaertler, Robert Görke, Martin Holzer, and Bastian Katz for valueable input and proof-reading.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
2. Wagner, D., Willhalm, T.: Speed-Up Techniques for Shortest-Path Computations. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 23–36. Springer, Heidelberg (2007)
3. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-Performance Multi-Level Graphs. In: 9th DIMACS Challenge on Shortest Paths (2006)
4. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Time Shortest-Path Queries in Road Networks. In: Algorithm Engineering and Experiments (ALENEX). pp. 46–59 (2007)
5. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. pp. 156–165 (2005)
6. Cooke, K., Halsey, E.: The shortest route through a network with time-dependent intemodal transit times. Journal of Mathematical Analysis and Applications 14, 493–498 (1966)
7. Ikeda, T., Hsu, M., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K., Mitoh, K.: A fast algorithm for finding better routes by AI search techniques. In: Vehicle Navigation and Information Systems Conference (1994)
8. Flinsenberg, I.C.M.: Route planning algorithms for car navigation. PhD thesis, Technische Universiteit Eindhoven (2004)
9. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric containers for efficient shortest-path computation. ACM Journal of Experimental Algorithmics 10, 1–30 (2005)
10. Sanders, P., Schultes, D.: Dynamic Highway-Node Routing. In: 6th Workshop on Experimental Algorithms (WEA) to appear (2007)
11. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: 9th DIMACS Challenge on Shortest Paths (2006)
12. Narvaez, P., Siu, K.Y., Tzeng, H.Y.: New dynamic algorithms for shortest path tree computation. IEEE/ACM Trans. Netw. 8, 734–746 (2000)
13. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. J. ACM 51, 968–992 (2004)
14. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research (2004)
15. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Algorithm Engineering and Experimentation (ALENEX). pp. 26–40 (2005)
16. Sedgewick, R., Vitter, J.S.: Shortest paths in Euclidean space. Algorithmica 1, 31–48 (1986)
17. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In: Algorithm Engineering and Experiments (ALENEX). pp. 129–143 (2006)
18. Dashtinezhad, S., Nadeem, T., Dorohonceanu, B., Borcea, C., Kang, P., Iftode, L.: TrafficView: a driver assistant device for traffic monitoring based on car-to-car communication. In: Vehicular Technology Conference, pp. 2946–2950. IEEE, New York (2004)
19. Kaufman, D.E., Smith, R.L.: Fastest paths in time-dependent networks for intelligent-vehicle-highway systems application. IVHS Journal 1, 1–11 (1993)