Highway Hierarchies Star*

Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner

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

Abstract. We study two speedup techniques for route planning in road networks: highway hierarchies (HH) and goal directed search using landmarks (ALT). It turns out that there are several interesting synergies. Highway hierarchies yield a way to implement landmark selection more efficiently and to store landmark information more space efficiently than before. ALT gives queries in highway hierarchies an excellent sense of direction and allows some pruning of the search space. For computing shortest distances and approximately shortest travel times, this combination yields a significant speedup over HH alone. We also explain how to compute actual shortest paths very efficiently.

1 Introduction

Computing fastest routes in a road networks $G = (V, E)$ from a source s to a target t is one of the showpieces of real-world applications of algorithmics. In principle, we could use DIJKSTRA's algorithm [1]. But for large road networks this would be far too slow. Therefore, there is considerable interest in speedup techniques for route planning.

A classical technique that gives a speedup of around two for road networks is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. Most speedup techniques use bidirectional search as an (optional) ingredient.

Another classical approach is goal direction via **A*** *search* [2]: lower bounds define a vertex potential that directs search towards the target. This approach was recently shown to be very effective if lower bounds are computed using precomputed shortest path distances to a carefully selected set of about 20 **L***andmark* nodes [3, 4] using the **T**riangle inequality (*ALT*). Speedups up to a factor 30 over bidirectional DIJKSTRA can be observed.

A property of road networks worth exploiting is their inherent hierarchy. Commercial systems use information on road categories to speed up search. 'Sufficiently far away' from source and target, only 'important' roads are used. This requires manual tuning of the data and a delicate tradeoff between computation speed and suboptimality of the computed routes. In a previous paper [5] we introduced the idea to *automatically* compute *highway hierarchies* that yield *optimal* routes *uncompromisingly quickly*. This was the first speedup technique that was able to preprocess the road network of a continent in realistic time and obtain large speedups (several thousands) over DIJKSTRA's algorithm. In [6] the basic method was considerably accelerated using many small measures and using *distance tables*: shortest path distances in the highest level of the hierarchy are precomputed. This way, it suffices to search locally around source and target node until the shortest path distance can be found by accessing the distance table.

A different hierarchy based method—reach based routing [7]—profits considerably from a combination with ALT [8]. The present state of affairs is that the combined method from

* Partially supported by DFG grant SA 933/1-3. and by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

[8] shows performance somewhat inferior to highway hierarchies with distance tables but without goal direction. Both methods turn out to be closely related. In particular, [8] uses methods originally developed for highway hierarchies to achieve fast preprocessing. Here, we explore the natural question how highway hierarchies can be combined with goal directed search in general and with ALT in particular.

1.1 Overview and Contributions

In the following sections we first review highway hierarchies in Section 2 (Algorithm HH) [6]. A new result presented there is a very fast algorithm for explicitly computing the shortest paths by precomputing unpacked versions of shortcut edges. Section 3 reviews Algorithm ALT [3, 4] and introduces refined algorithms for selecting landmarks. The main innovation there is restricting landmark selection to nodes on higher levels of the highway hierarchy.

The actual integration of highway hierarchies with ALT (Algorithm HH*) is introduced in Section 4. This is nontrivial in several respects. For example, we need incremental access to the distance tables for finding upper bounds and a different way to control the progress of forward and backward search. We also have to overcome the problem that search cannot be stopped when search frontiers meet. On the other hand, there are several simplifications compared to ALT. Abandoning the reliance on a stopping criterion allows us to use simpler, faster, and stronger lower bounds. Using distance tables obviates the need for dynamic landmark selection. Another interesting approach is to stop the search when a certain guaranteed solution quality has been obtained. There are several interesting further optimisations. In particular, we can be more space efficient than ALT by storing no landmark information on the lowest level of the hierarchy. We describe how the missing information can be reconstructed efficiently at query time. As a side effect, we introduce a way to limit the length of shortcuts. This measure turns out to be of independent interest since it also improves the basic HH algorithm.

Section 5 reports extensive experiments performed using road networks of Western Europe and the USA. Section 6 summarises the results and outlines possible future work.

1.2 More Related Work

There are several other approaches to goal directed search. Our first candidate for combination with highway hierarchies were *Precomputed Cluster Distances* [9]. PCDs allow the computation of upper and lower bounds based on precomputed distances between partitions of the road networks. These lower bounds cannot be used for A^* search since they can produce negative reduced edge weights. The search space can still be pruned by discontinuing search at node v if the lower bound from v to t indicates that the best upper bound seen so far cannot possibly be improved. An advantage of PCDs over landmarks is that they need less space. We did not implement this however since PCDs are rather ineffective for search in the lower levels of the hierarchy and since our distance table optimisation from [6] is already very effective for pruning search at the higher levels of the hierarchy. In contrast, landmarks can be used together with A^* search and thus can direct the search towards the target already in the lower levels of the hierarchy.

An important family of speedup techniques [10–12] associates information with each edge e. This information specifies a superset of the nodes reached via e on some short-

est path. *Geometric containers* [10] require node coordinates and store a simple geometrical object containing all the nodes reached via a shortest path. *Edge flags* partition the graph into regions. For each edge e and each region R one bit specifies whether there is a shortest path via e into region R [11, 12]. Both techniques alone already contain both direction information and hierarchy information so that very big speedups comparable to highway hierarchies can be achieved. However, so far these methods would have forbiddingly large preprocessing times for the largest available road networks. Therefore these approaches looked not so interesting for a first attempt to combine goal directed search with highway hierarchies.

2 Highway Hierarchies

The basic idea of the highway hierarchies approach is that outside some local areas around the source and the target node, only a subset of 'important' edges has to be considered in order to be able to find the shortest path. The concept of a *local area* is formalised by the definition of a neighbourhood node set[1] $\mathcal{N}(v)$ for each node v. Then, the definition of a *highway network* of a graph $G = (V, E)$ that has the property that all shortest paths are preserved is straightforward: an edge $(u, v) \in E$ belongs to the highway network iff there are nodes $s, t \in V$ such that the edge (u, v) appears in the canonical shortest path[2] $\langle s, \ldots, u, v, \ldots, t\rangle$ from s to t in G with the property that $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$.

The size of a highway network (in terms of the number of nodes) can be considerably reduced by a contraction procedure: for each node v, we check a *bypassability criterion* that decides whether v should be *bypassed*—a operation that creates shortcut edges (u, w) representing paths of the form $\langle u, v, w\rangle$. The graph that is induced by the remaining nodes and enriched by the shortcut edges forms the *core* of the highway network. The bypassability criterion takes into account the degree of the node v and the number of shortcuts that would be created if v was bypassed. For details, we refer to [6].

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$. Level 0 corresponds to the original graph G. Level 1 is obtained by computing the *highway network* of level 0, level 2 by computing the highway network of the core of level 1 and so on.

2.1 Highway Query

In [5], we show how the highway hierarchy of a given graph can be constructed efficiently. After that, we can use the *highway query algorithm* [6] to perform s-t queries. It is an adaptation of the bidirectional version of DIJKSTRA's algorithm. The search starts at s and t in level 0. When the neighbourhood of s or t is left, we switch to level 1 and continue the search. Similarly, we switch to the next level if the neighbourhood of the entrance point to the current level is left (Fig. 1). When the core of some level has been entered, we never leave it again: in particular, we do not follow edges that lead to a bypassed node; instead, we use the shortcuts that have been created during the construction.

[1] In [6], we give more details on the definition of neighbourhoods. In particular, we distinguish between a forward and a backward neighbourhood. However, in this context, we would like to slightly simplify the notation and concentrate on the concepts that are important to understand the subsequent sections.

[2] For each connected node pair (s, t), we select a unique *canonical shortest path* in such a way that each subpath of a canonical shortest path is canonical as well. For details, we refer to [5].

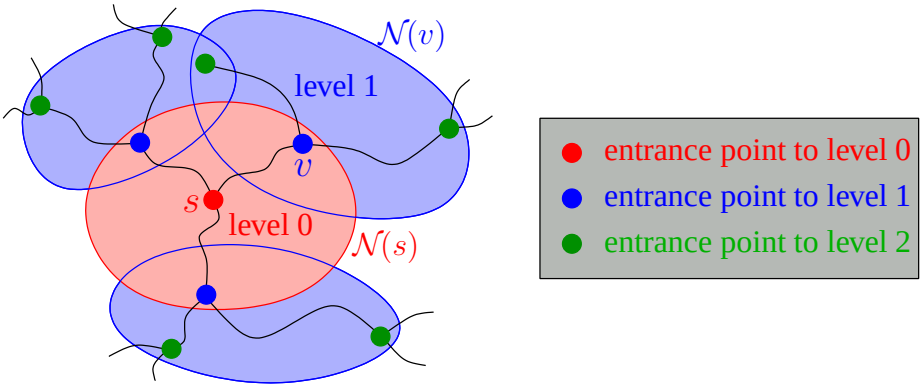


Fig. 1. A schematic diagram of a highway query. Only the forward search started from the source node s is depicted.

At this point, we can observe two interesting properties of the highway query algorithm. First, it is *not* goal-directed. In fact, the forward search 'knows' nothing about the target and the backward search 'knows' nothing about the source, so that both search processes work completely independently and spread into all directions. Second, when both search scopes meet at some point, we cannot easily abort the search—in contrast to the bidirectional version of DIJKSTRA's algorithm, where we can abort immediately after a common node has been settled from both sides. The reason for this is illustrated in Fig. 2. In the upper part of

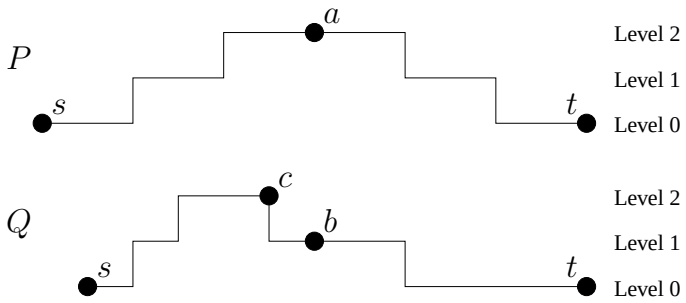


Fig. 2. Schematic profile of a bidirectional highway query.

the figure, the bidirectional query from a node s to a node t along a path P is represented by a profile that shows the level transitions within the highway hierarchy. To get a sequential algorithm, at each iteration we have to decide whether a node from the forward or the backward queue is settled. We assume that a strategy is used that favours the smaller element. Thus, both search processes meet in the middle, at node a. When this happens, a path from s to t has been found. However, we have no guarantee that it is the shortest one. In fact, the lower part of the figure contains the profile of a shorter path Q from s to t, which is less symmetric than the profile of P. Note that the very flexible definition of the neighbourhoods allows such asymmetric profiles. When a on P is settled from both sides, b has been reached on Q by the backwards search, but *not* by the forward search since a search process never goes downwards in the hierarchy: therefore, at node c, the forward search is not continued on the path Q. We find the shorter path Q not until the backward search has reached c—which happens *after* P has been found. Hence, it would be wrong to abort the search, when a has been settled.

In [5], we introduced some rather complicated abort criteria, which we dropped in [6] since they did reduce the search space, but the evaluation of the criteria was too expensive.

2.2 Using a Distance Table

The construction of less levels of the highway hierarchy and the usage of a complete *distance table* for the core of the topmost level can considerably accelerate the query: whenever the forward (backward) search enters the core of the topmost level at some node u, u is added to a node set $\overrightarrow{I}$ ($\overleftarrow{I}$) and the search is not continued from u. Since all distances between the nodes in the sets $\overrightarrow{I}$ and $\overleftarrow{I}$ have been precomputed and stored in a table, we can easily determine the shortest path length by considering all node pairs $(u, v), u \in \overrightarrow{I}, v \in \overleftarrow{I}$, and summing up $d(s, u) + d(u, v) + d(v, t)$. For details, we refer to [6].

Using the distance table can be seen as extreme case of goal-directed search: from the nodes in the set $\overrightarrow{I}$, we directly 'jump' to the nodes in the set $\overleftarrow{I}$, which are close to the target. Thus, we can say that the highway query with the distance table optimisation works in two phases: a strictly non-goal-directed phase till the sets $\overrightarrow{I}$ and $\overleftarrow{I}$ have been determined, followed by a 'goal-directed jump' using the distance table.

2.3 Complete Description of the Shortest Path

So far, we have dealt only with the computation of shortest path *distances*. In order to determine a complete description of the shortest path, a) we have to bridge the gap between the forward and backward core entrance points and b) we have to expand the used shortcuts to obtain the corresponding subpaths in the original graph.

Problem a) can be solved using a simple algorithm: We start with the forward core entrance point u. As long as the backward entrance point v has not been reached, we consider all outgoing edges (u, w) in the topmost core and check whether $d(u, w)+d(w, v) = d(u, v)$; we pick an edge (u, w) that fulfils the equation, and we set $u := w$. The check can be performed using the distance table. It allows us to greedily determine the next hop that leads to the the backward entrance point.

Problem b) can be solved without using any extra data (Variant 1): for each shortcut (u, v), we perform a search from u to v in order to determine the path in the original graph; this search can be accelerated by using the knowledge that the first edge of the path enters a component C of bypassed nodes, the last edge leads to v, and all other edges are situated within the component C.

However, if a fast output routine is required, it is necessary to spend some additional space to accelerate the unpacking process. We use a rather sophisticated data structure to represent unpacking information for the shortcuts in a space-efficient way (Variant 2). In particular, we do not store a sequence of node IDs that describe a path that corresponds to a shortcut, but we store only *hop indices*: for each edge (u, v) on the path that should be represented, we store its index minus the index of the first edge of u. Since in most cases the degree of a node is very small, these hop indices can be stored using only a few bits. The unpacked shortcuts are stored in a recursive way, e.g., the description of a level-2 shortcut may contain several level-1 shortcuts. Accordingly, the unpacking procedure works recursively.

To obtain a further speed-up, we have a variant of the unpacking data structures (Variant 3) that caches the complete descriptions—without recursions—of all shortcuts that belong to the topmost level, i.e., for these important shortcuts that are frequently used, we do not have to use a recursive unpacking procedure, but we can just append the corresponding subpath to the resulting path.

3 A^* Search Using Landmarks

In this section we explain the known technique of A^* search [2] in combination with landmarks. We follow the implementation presented in [4]. In Section 3.2 we introduce a new landmark selection technique called *advancedAvoid*. Furthermore, we present how the selection of landmarks can be accelerated using highway hierarchies.

The search space of DIJKSTRA's algorithm can be visualised as a circle around the source. The idea of goal-directed or A^* search is to push the search towards the target. By adding a 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. A 'good' potential lowers the priority of nodes that lie on a shortest path to the target. It is easy to see that 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.

Bidirectional A^*. At a glance, combining A^* and bidirectional search seems easy. Simply use a feasible potential π_f for the forward and a feasible potential π_r for the backward search. However, this does not work due to the fact that both searches might work on different reduced costs, so that the shortest path might not have been found when both searches meet. This can only be guaranteed if π_f and π_r are *consistent* meaning $w_{\pi_f}(u,v)$ in G is equal to $w_{\pi_r}(v,u)$ in the reverse graph. We use the variant of an average potential function [13] 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. By adding $\pi_r(t)/2$ to the forward and $\pi_f(s)/2$ to the backward search, p_f and p_r provide lower bounds to the target and source, respectively. Note that these potentials are feasible and consistent but provide worse lower bounds than the original ones.

ALT. There exist several techniques [14, 15] how 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:

$$d(u,v) + d(v,L) \geq d(u,L) \quad \text{and} \quad d(L,u) + d(u,v) \geq d(L,v)$$

Therefore, $\underline{d}(u,v) := \max_{L \in S} \max\{d(u,L) - d(v,L), d(L,v) - d(L,u)\}$ provides a lower bound for the distance $d(u,v)$. The quality of the lower bounds highly depends on the quality of the selected landmarks.

Our implementation uses the tuning techniques of *active landmarks*, *pruning* and the enhanced stopping criterion. We stop the search if the sum of minimum keys in the forward and the backward queue exceed $\mu + p_f(s)$, where μ represents the tentative shortest path length and is therefore an upper bound for the shortest path length from s to t. For each s-t query only two landmarks—one 'before' the source and one 'behind' the target—are initially used. At certain checkpoints we decide whether to add an additional landmark to the active set, with a maximal amount of six landmarks. Pruning means that before relaxing an arc (u, v) during the forward search we also check whether $d(s, u) + w(u, v) + \pi_f(v) < \mu$ holds. This technique may be applied to the backward search easily. Note that for pruning, the potential function need not be consistent.

3.1 Landmark-Selection

A crucial point in the success of a high speedup when using ALT is the quality of landmarks. Since finding good landmarks is hard, several heuristics [3, 4] exist. We focus on the best known techniques *avoid* and *maxCover*.

Avoid. This heuristic tries to identify regions of the graph that are not well covered by the current landmark set S. Therefore, a shortest-path tree T_r is grown from a random node r. The *weight* of each node v is the difference between $d(v, r)$ and the lower bound $\underline{d}(v, r)$ obtained by the given landmarks. The *size* of a node v is defined by the sum of its weight and the size of its children in T_r. If the subtree of T_r rooted at v contains a landmark, the size of v is set to zero. Starting from the node with maximum size, T_r is traversed following the child with highest size. The leaf obtained by this traversal is added to S. In this strategy, the first root is picked at random. The following roots are picked with a probability proportional to the square of the distance to its nearest landmark.

MaxCover [4]. The main disadvantage of avoid is the starting phase of the heuristic. The first root is picked at random and the following landmarks are highly dependent on the starting landmark. MaxCover improves on this by first choosing a candidate set of landmarks (using avoid) that is about four times larger than needed. The landmarks actually used are selected from the candidates using several attempts with a local search routine. Each attempt starts with a random initial selection.

3.2 New Selection Techniques

In the following we introduce a new heuristic called *advancedAvoid* to select landmarks. Furthermore, we use the highway hierarchies to speed up the selection of landmarks.

AdvancedAvoid. Another approach to compensate for the disadvantages of avoid is to exchange the first landmarks generated by the avoid heuristic. More precisely, we generate k avoid landmarks, then in each iteration i take the landmarks $(i-1)k' + 1$ to ik' from the set S and generate k' new landmarks using avoid again. We repeat this procedure r times. The advantage of advancedAvoid towards maxCover is the computation time. While maxCover takes about five times longer than avoid, the overhead for advancedAvoid is about 45% for $k = 16$, $k' = 6$, and $r = 1$ on the road network of Western Europe.

Core Landmarks. The computation of landmarks is expensive. Calculating maxCover landmarks on the European network takes about 75 minutes, while constructing the whole highway hierarchy can be done in about 15 minutes. A promising approach is to use the highway hierarchy to reduce the number of possible landmarks: The level-1 core of the European road network has six times fewer nodes than the original network and its construction takes only about three minutes. Using the core as possible positions for landmarks, the computation time for calculating landmarks (all heuristics) can be decreased. Using only the nodes of higher level cores reduces the time for selecting landmarks even more. Figure 3 shows an example of 16 advancedAvoid landmarks, generated on the level-1 core of the European network.

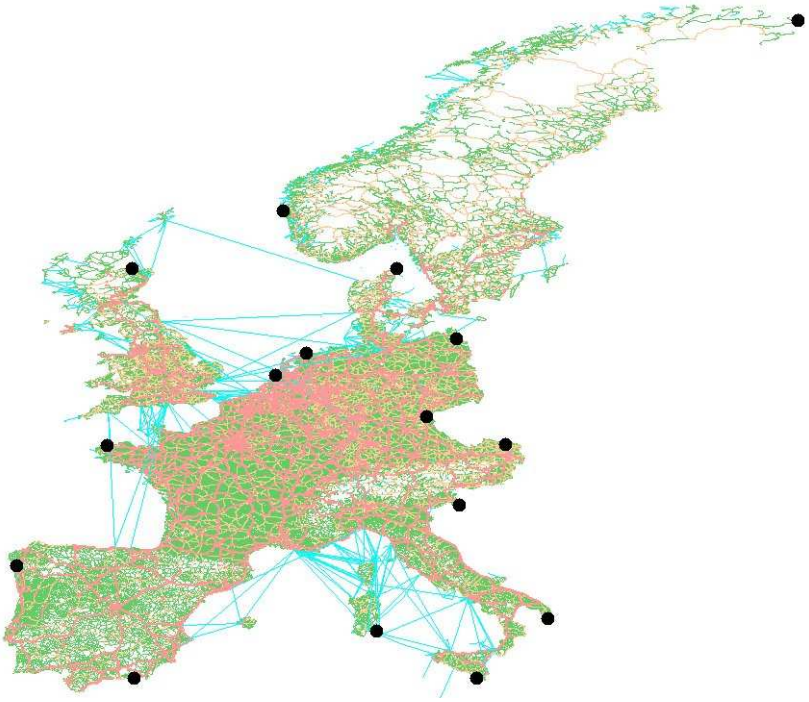


Fig. 3. 16 advancedAvoid core 1 landmarks on the Western European road network

4 Combining Highway Hierarchies and A^* Search

Previously (see Section 2), we strictly separated the search phase to the topmost core from the access to the distance table: first, the sets of entrance points $\overrightarrow{I}$ and $\overleftarrow{I}$ into the core of the topmost level were determined, and afterwards the table look-ups were performed. Now we interweave both phases: whenever a forward core entrance point u is discovered, it is added to $\overrightarrow{I}$ and we immediately consider all pairs $(u, v), v \in \overleftarrow{I}$, in order to check whether the tentative shortest path length μ can be improved. (An analogous procedure applies to the discovery of a backward core entrance point.) This new approach is advantageous since we can use the tentative shortest path length μ as an upper bound on the actual shortest path length. In [5, 6], the highway query algorithm used a strategy that compares the minimum elements of both priority queues and prefers the smaller one in order to sequentialise forward

and backward search. If we want to obtain good upper bounds very fast, this might not be the best choice. For example, if the source node belongs to a densely populated area and the target to a sparsely populated area, the distances from the source and target to the entrance points into the core of the topmost level will be very different. Therefore, we now choose a strategy that balances $|\overrightarrow{I}|$ and $|\overleftarrow{I}|$, preferring the direction that has encountered less entrance points. In case of equality (in particular, in the beginning when $|\overrightarrow{I}| = |\overleftarrow{I}| = 0$), we use a simple alternating strategy.

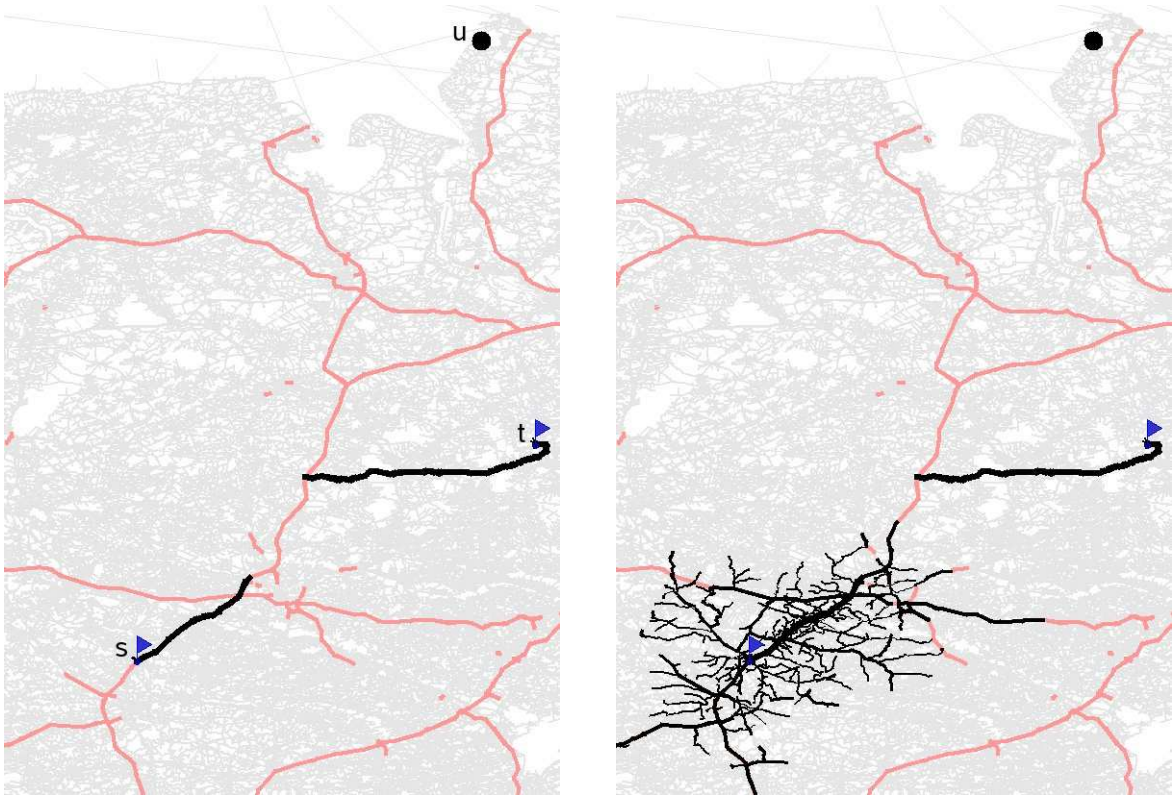
We enhance the highway query algorithm with goal-directed capabilities—obtaining an algorithm that we call HH* *search*—by replacing edge weights by *reduced costs* using potential functions π_f and π_r for forward and backward search. By this means, the search is directed towards the respective target, i.e., we are likely to find some s-t path very soon. However, just using the reduced costs only changes the *order* in which the nodes are settled, it does not reduce the search space. The ideal way to benefit from the early encounter of the forward and backward search would be to abort the search as soon as an s-t path has been found. And, as a matter of fact, in case of the ALT algorithm [3]—even in combination with reach-based routing [8]—it can be shown that an immediate abort is possible without losing correctness if consistent potential functions are used (see Section 3). In contrast, this does not apply to the highway query algorithm since even in the non-goal-directed variant of the algorithm, we cannot abort when both search scopes have met (see Section 2).

Fortunately, there is another aspect of goal-directed search that can be exploited, namely *pruning*: finding any s-t path also means finding an upper bound μ on the length of the shortest s-t path. Comparing the lower bounds with the upper bound can be used to prune the search. In Section 3, the pruning of *edges* has already been mentioned. Alternatively, we can prune *nodes*: if the key of a settled node u is greater than the upper bound, we do not have to relax u's edges. Note that, using reduced costs, the key of u is the distance from the corresponding source to u plus the lower bound on the distance from u to the corresponding target.

Since we do not abort when both search scopes have met and because we have the distance table, a very simple implementation of the ALT algorithm is possible. First, we do not have to use consistent potential functions. Instead, we directly use the lower bound to the target as potential for the forward search and, analogously, the lower bound from the source as potential for the backward search. These potential functions make the search processes approach their respective target faster than using consistent potential functions so that we get good upper bounds very early. In addition, the node pruning gets very effective: if one node is pruned, we can conclude that all nodes left in the same priority queue will be pruned as well since we use the same lower bound for pruning and for the potential that is part of the key in the priority queue. Hence, in this case, we can immediately stop the search in the corresponding direction.

Second, it is sufficient to select at the beginning of the query for each search direction only one landmark that yields the best lower bound. Since the search space is limited to a relatively small local area around source and target (due to the distance table optimisation), we do not have to pick more landmarks, in particular, we do not have to add additional landmarks in the course of the query, which would require flushing and rebuilding the priority queues. Thus, adding A^* search to the highway query algorithm (including the distance table optimisation) causes only little overhead per node.

However, there is a considerable drawback. While the goal-directed search (which gives good upper bounds) works very well, the pruning is not very successful when we want to compute *fastest* paths, i.e., when we use a travel time metric, because then the lower bounds are usually too weak. Figure 4 gives an example for this observation, which occurs quite frequently in practice. The first part of the shortest path from s to t corresponds to the first



(a) (b)

Fig. 4. Two snapshots of the search space of an HH* search using a travel time metric. The landmark u of the forward search from s to t is explicitly marked. The landmark used by the backward search is somewhere below s and not included in the chosen clipping area. The search space is black, parts of the shortest path are represented by thick lines. In addition, motorways are highlighted in red.

part of the shortest path from s to the landmark u. Thus, the reduced costs of these edges are zero so that the forward search starts with traversing this common subpath. The backward search behaves in a similar way. Hence, we obtain a perfect upper bound very early (a). Still, the lower bound on $d(s,t)$ is quite bad: we have $d(s,u) - d(t,u) \leq d(s,t)$. Since staying on the motorway and going directly from s to u is much faster than leaving the motorway, driving through the countryside to t and continuing to u, the distance $d(s,t)$ is clearly underestimated. The same applies to lower bounds on $d(v,t)$ for nodes v close to s. Hence, pruning the forward search does not work properly so that the search space still spreads into all directions before the process terminates (b). In contrast, the node s lies on the shortest path (in the reverse graph) from t to the landmark that is used by the backward search. (Since this landmark is very far away to the south, it has not been included in the figure.) Therefore, the lower bound is perfect so that the backward search stops immediately. However, this is a fortunate case that occurs rather rarely.

4.1 Approximate Queries

We pointed out above that in most cases we *find* a (near) shortest path very quickly, but it takes much longer until we *know* that the shortest path has been found. We can adapt to this situation by defining an abort condition that leads to an approximate query algorithm: when a node u is removed from the forward priority queue and we have $(1+\varepsilon)\cdot(d(s,u)+\underline{d}(u,t)) > \mu$ (where $\varepsilon \geq 0$ is a given parameter), then the search is not continued in the forward direction. In this case, we may miss some s-t-paths whose length is $\geq d(s,u)+\underline{d}(u,t)$ since the key of any remaining element v in the priority queue is $\geq d(s,u)+\underline{d}(u,t)$ and it is a lower bound on the length of the shortest path from s via v to t. Thus, if the shortest path is among these paths, we have $d(s,t) \geq d(s,u)+\underline{d}(u,t) > \mu/(1+\varepsilon)$, i.e., we have the guarantee that the best path that we have already found (whose length corresponds to the upper bound μ) is at most $(1+\varepsilon)$ times as long as the shortest path. An analogous stopping rule applies to the backward search.

4.2 Optimisations

Better Upper Bounds. We can use the distance table to get good upper bounds even earlier. So far, the distance table has only been applied to entrance points into the core V'_L of the topmost level. However, in many cases we encounter nodes that belong to V'_L earlier during the search process. Even the source and the target node could belong to the core of the topmost level. Still, we have to be careful since the distance table only contains the shortest path lengths within the topmost core and a path between two nodes in V'_L might be longer if it is restricted to the core of the topmost level than using all edges of the original graph. This is the reason why we have not used such a premature jump to the highest level before. But now, in order to just determine upper bounds, we could use these additional table look-ups. The effect is limited though because finding good upper bounds works very well anyway—the lower bounds are the crucial part. Therefore, the exact algorithm does without the additional look-ups. The approximate algorithm applies this technique to the nodes that remain in the priority queues after the search has been terminated since this might improve the result[3]. For example, we would get an improvement if the goal-directed search led us to the wrong motorway entrance ramp, but the right entrance ramp has at least been inserted into the priority queue.

Reducing Space Consumption. We can save preprocessing time and memory space if we compute and store only the distances between the landmarks and the nodes in the core of some fixed level k. Obviously, this has the drawback that we cannot begin with the goal-directed search immediately since we might start with nodes that do not belong to the level-k core so that the distances to and from the landmarks are not known. Therefore, we introduce an additional *initial query phase*, which works as a normal highway query and is stopped when all entrance points into the core of level k have been encountered. Then, we can determine the distances from s to all landmarks since the distances from s via the level-k core entrance points to the landmarks are known. Analogously, the distances from the landmarks to t can be computed. The same process is repeated for interchanged source and target nodes—i.e., we search forward from t and backward from s—in order to determine the dis-

[3] In a preliminary experiment, the total error observed in a random sample was reduced from 0.096% to 0.053%.

tances from t to the landmarks and from the landmarks to s. Note that this second subphase can be skipped when the first subphase has encountered only bidirected edges.

The priority queues of the *main query phase* are filled with the entrance points that have been found during (the first subphase of) the initial query phase. We use the distances from the source or target node plus the lower bound to the target or source as keys for these initial elements. Since we never leave the level-k core during the main query phase, all required distances to and from the landmarks are known and the goal-directed search works as usual. The final result of the algorithm is the shortest path that has been found during the initial or the main query phase.

Limiting Component Sizes. Since the search processes from the source and target to the level-k core entrance points are often executed twice (once for each direction), it is important to bound this overhead. Therefore, we implemented a limit on the number of hops a shortcut may represent. By this means, the sizes of the components of bypassed nodes are reduced—in particular, the first contraction step tended to create quite large components of bypassed nodes so that it took a long time to leave such a component when the search was started from within it. Interestingly, this measure has also a very positive effect on the worst case analysis in [6]: it turned out that the worst case was caused by very large components of bypassed nodes in some sparsely populated areas, whose sizes now have been considerably reduced by the shortcut hops limit.

5 Experiments

5.1 Environment, Instances, and Parameters

The experiments were done on one core of an AMD Opteron Processor 270 clocked at 2.0 GHz with 4 GB main memory and 2×1 MB L2 cache, running SuSE Linux 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3. We use 32 bit integers to store edge weights and path lengths.

We deal with the road network of Western Europe[4], which has been made available for scientific use by the company PTV AG. Only the largest strongest connected component is considered. The original graph contains for each edge a length and a road category, e.g., motorway, national road, regional road, urban street. We assign average speeds to the road categories, compute for each edge the average travel time, and use it as weight. In addition to this *travel time metric*, we perform experiments on a variant of the European graph with a *distance metric*. We also perform experiments on the US road network (without Alaska and Hawaii), which has been obtained from the TIGER/Line Files [16]. Again, we consider only the largest strongest connected component, and we deal with both a travel time and a distance metric. In contrast to the PTV data, the TIGER graph is undirected, planarised and distinguishes only between four road categories. All graphs[5] have been taken from the DIMACS Challenge website [17]. Table 1 summarises the properties of the used networks.

[4] 14 countries: Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK

[5] Note that the experiments on the TIGER graphs had been performed before the final versions, which use a finer edge costs resolution, were available. We did not repeat the experiments since we expect hardly any change in our measurement results.

Table 1. Properties of the used road networks.

	Europe	USA (Tiger)
#nodes	18 010 173	23 947 347
#directed edges	42 560 279	58 333 344
#road categories	13	4
average speeds [km/h]	10–130	40–100
neighbourhood size H (time)	60	70
neighbourhood size H (dist)	100, 200, 300, ...	

At first, we report only the times needed to compute the shortest path distance between two nodes without outputting the actual route. These times are averages based on 10 000 randomly chosen (s, t)-pairs. In addition to providing average values, we use the methodology from [5] in order to plot query times (and error rates) against the 'distance' of the target from the source, where in this context, the *Dijkstra rank* is used as a measure of distance: for a fixed source s, the Dijkstra rank of a node t is the rank w.r.t. the order which DIJKSTRA's algorithm settles the nodes in. Such plots are based on 1 000 random source nodes. In the last paragraph of Section 5.3, we also give the times needed to traverse the computed shortest paths.

Since it has turned out that a better performance is obtained when the preprocessing starts with a contraction phase, we practically skip the first construction step (by choosing neighbourhood sets that contain only the node itself) so that the first highway network virtually corresponds to the original graph. Then, the first real step is the contraction of level 1 to get its core. Note that in this case, distances within the core of level 1 are equal to the distances between level-1 core nodes in the original graph.

The shortcut hops limit (introduced in Section 4) is set to 10. The neighbourhood size H (introduced in [5, 6]) for the travel time metrics is set to 60 and 70 for the European and the US network, respectively. For the distance metric versions of both graphs, we use the linearly increasing sequence 100, 200, 300, ... as neighbourhood sizes to compute levels 2, 3, 4, ... of the hierarchy.

5.2 Landmarks

Preprocessing First, we analyse the preprocessing of the ALT algorithm with different selection strategies on different cores of the highway hierarchy. We use 16 avoid, advancedAvoid and maxCover landmarks selected from the whole graph and from the core of levels 1–3. For advancedAvoid, we deactivate 6 landmarks once (see Section 3.2). Table 2 gives an overview of the preprocessing of the ALT algorithm on the European network. For the US network, see Tab. 9 in Appendix A.

We observe that the time spent for selecting landmarks decreases significantly when switching to higher cores. Unfortunately, we have to compute the distances from and to all nodes in the original graph if we use core landmarks for the ALT algorithm (on the full graph these distances are computed during selection). In addition, we have to compute the highway information. Nevertheless, the computation of core 1 only takes about three minutes leading to a decrease of total preprocessing with regard to all selection techniques. With regard to preprocessing time, using avoid and advancedAvoid on the cores of level 2 or 3 does not seem reasonable while maxCover benefits from switching to higher cores.

Table 2. Overview of the preprocessing time for different selection strategies on the European network. All figures are given in minutes of computation time. Generating 16 maxCover landmarks on the whole graph requires more than 4 GB RAM. Therefore, these landmarks were generated on an AMD Opteron Processor 252 clocked at 2.6 GHz with 16 GB main memory.

		full graph			core-1			core-2			core-3		
metric	preproc. [min]	avoid	adv.av.	maxCov	avoid	adv.av.	maxCov	avoid.	adv.av.	maxCov	avoid	adv.av.	maxCov
time	highway info	–	–	–	2.7	2.7	2.7	11.5	11.5	11.5	13.7	13.7	13.7
	selection	15.8	23.2	88.3	2.5	3.6	21.2	0.4	0.5	3.3	0.1	0.1	0.8
	distances	–	–	–	6.3	6.3	6.3	6.3	6.3	6.3	6.3	6.3	6.3
dist	highway info	–	–	–	2.7	2.7	2.7	13.6	13.6	13.6	20.1	20.1	20.1
	selection	13.5	19.2	75.3	2.1	3.0	19.5	0.4	0.5	2.4	0.1	0.1	1.2
	distances	–	–	–	4.2	4.2	4.2	4.2	4.2	4.2	4.2	4.2	4.2

Another advantage when switching to higher cores is memory consumption. While about 2.3 GB of RAM are needed for the distances from and to all nodes when selecting 16 avoid landmarks on the full graph, 384 MB are sufficent when using the core of level 1. Using the core-2 (core-3) even further reduces the memory consumption to 64 (17) MB. Note, that we use 32 bit integers for keeping the distances in the main memory.

Search Space. Table 3 gives an overview of the average search space for 1 000 random s-t queries on the road network of Western Europe and the US. For each selection strategy and core we generated 10 different sets of 16 landmarks. We report the average, minimum and maximum of the average search space.

Table 3. Overview of the average number of nodes settled by the ALT algorithm for 1 000 random queries on the road networks of Western Europe and the US for travel times and—in parentheses—distances. The figures are based on 10 different sets of landmarks.

	Europe			USA		
landmarks	average	min	max	average	min	max
avoid	93520 (253552)	72720 (241609)	103929 (264822)	220333 (308823)	177826 (261037)	276709 (345416)
adv.av.	86340 (256511)	72004 (218335)	95663 (283911)	210703 (302521)	183542 (278157)	240971 (338930)
maxCov	75220 (230110)	71061 (212641)	77556 (254339)	175359 (282162)	160635 (255140)	186457 (297818)
avoid-c1	84515 (254596)	67895 (224111)	96775 (279603)	218313 (309200)	162054 (271835)	279510 (346570)
adv.av.-c1	82423 (252002)	71084 (226088)	98963 (275779)	204800 (306364)	187410 (263238)	247013 (367764)
maxCov-c1	75992 (230979)	74640 (209605)	78007 (257163)	177304 (277981)	157530 (268946)	190396 (288383)
avoid-c2	89001 (259145)	74980 (242489)	97764 (277761)	206188 (310958)	170539 (265233)	233813 (366833)
adv.av.-c2	86611 (257963)	75450 (218037)	99107 (275780)	221356 (306553)	175679 (252837)	250045 (360645)
maxCov-c2	75379 (230310)	71551 (211168)	80815 (250145)	187644 (281465)	173851 (254751)	200721 (309360)
avoid-c3	91201 (264821)	76681 (245809)	99667 (296217)	237615 (313672)	193502 (270129)	277167 (351791)
adv.av.-c3	91163 (275991)	84116 (263978)	99779 (301018)	234385 (321328)	200155 (293913)	266757 (354027)
maxCov-c3	72310 (239584)	68348 (209720)	76770 (259185)	194707 (283086)	172334 (257488)	205618 (307022)

We see that for distances the quality of landmarks is almost independent of the chosen level of the hierarchy. Only when switching from level 2 to 3 we observe a mild increase of the search space when using advancedAvoid landmarks. However, for travel times on the European network an interesting phenomenon is that avoid gets better when switching from the whole graph to core 1 but gets worse and worse with higher levels on which landmarks are selected. On the US network, the search space reduces when switching to core 2 in combination with avoid landmarks. MaxCover is nearly independent of the chosen level on the European network while on the US network a slight loss of quality can be observed with higher levels.

There seem to be two counteracting effects here: On higher levels of the hierarchy, we loose information. For example, peripheral nodes that are candidates for good landmarks are dropped. On the other hand, concentrating on higher level edges in landmark selection heuristics could be beneficial since these are edges needed by many shortest paths.

In general, maxCover outperforms avoid and advancedAvoid regarding the average quality of the obtained landmarks. Nevertheless, in most cases the minimum average search space is nearly the same for all selection strategies within a core, while some sets of avoid and advancedAvoid landmarks lead to search spaces 25% higher than the worst maxCover landmarks. So, the maxCover routine seems to be more robust than avoid or advancedAvoid. Comparing avoid and advancedAvoid we observe just a mild improvement in quality. Thus, the additional computation time of advancedAvoid is not worth the effort.

Combining the results from Tabs. 2 and 3, another strategy seems promising: maxCover landmarks from the core of level 2 or 3 outperform avoid landmarks from the full graph and their computation—including the highway information—needs only additional 5 minutes compared to avoid landmarks from the full graph. For this reason, we use such landmarks for our further experiments.

Efficiency and Approximation Table 4 indicates the efficiency of our implementation by reporting query times in comparison to the bidirectional variant of DIJKSTRA's algorithm. For comparison with approximate HH queries we also provide the results for an approximate ALT algorithm: Stop the query if the sum of the minimum keys in the forward and the backward queue exceed $\mu/(1+\varepsilon) + p_f(s)$ with $\varepsilon = 0.1$. This stopping criterion keeps the error rate below 10%.

Table 4. Comparison of the bidirectional variant of DIJKSTRA's algorithm, the ALT algorithm, and the approximate ALT algorithm concerning search space, query times and error rate. The landmarks are 16 maxCover core-3 landmarks. The figures are based on 1 000 random queries.

		Europe			USA		
metric		bi.Dij.	ALT	approx.ALT	bi.Dij.	ALT	approx.ALT
time	#settled nodes	$4.68 \cdot 10^6$	73 563	61 939	$7.42 \cdot 10^6$	192 938	182 426
	query time [ms]	2 707	55.2	45.8	3 808	129.2	116.9
	inaccurate queries	–	–	12.1%	–	–	8.9%
dist	#settled nodes	$5.27 \cdot 10^6$	241 476	219 124	$8.11 \cdot 10^6$	281 335	263 375
	query time [ms]	2 013	169.2	150.9	3 437	177.1	163.5
	inaccurate queries	–	–	33.7%	–	–	24.8%

Analysing the speedups compared to the bidirectional variant of DIJKSTRA's algorithm, we observe a search space reduction for Europe (travel times) by a factor of about 63.6. This reduction leads to a speedup factor of 49.0 concerning query times. For the USA (travel times), speedup concerning search space and query times is smaller than for Europe. We observe a factor of 38.5 for search space and 29.5 for query times. The reason for this discrepancy is the overhead for computing the potential and is also reported in [3, 4, 8].

For the distance metric on the European network we observe a reduction in search space of factor 21.8, leading to a speedup factor of 11.8. The corresponding figures for the US are 28.8 and 19.4. Thus, the situation is vice versa to travel times. Here, speedups are better on the US network than on the European network. The higher speedups for travel times are due to the fact that for distances the advantage of taking fast highways instead of slow

streets is smaller than for travel times. Since the difference between the slowest and fastest road category (see Tab. 1) is bigger for Europe, the ALT algorithm performs better on this network than on the US network when using travel times.

Comparing our results with the ones from [8] we have about 10% higher search spaces on the US network (travel times). This derives from the fact that on the US network with travel times the quality of maxCover landmarks slightly decreases when switching to higher cores (see Tab. 3). Nevertheless, our average query times in this instance are 2.49 (129 ms to 322 ms) times faster, although we are using a slower computer. A reason for this is a different overhead factor.While our implementation has an overhead of factor 1.3, the figures from [8] suggest an overhead of 2.

For the travel time metric, approximate queries perform only 20% better on Europe and 10% better on the US than exact ones. The percentage of inaccurate queries is 12 and 8%, respectively. For the distance metric, the speedup for approximate queries is even less and the percentage of inaccurate queries is much higher, namely 33.7 and 24.8% for the European and US network, respectively. These high numbers of wrong queries are due to the fact that for the distance metric there are more possibilities of short paths with similar lengths since the difference between taking fast highways and driving on slow streets fades. So, approximation for ALT adds only a small speedup not justifying the loss of correctness. For a detailed analysis of the approximation error see Tab. 10 and Figs. 11–14 in Appendix A.

Local Queries. Figure 5 gives an overview of the query times in relation to the Dijkstra rank. For the same analysis of the approximate ALT algorithm, see Fig. 8 in Appendix A. The results for the distance metric are also located in Appendix A (Figures 9 and 10).

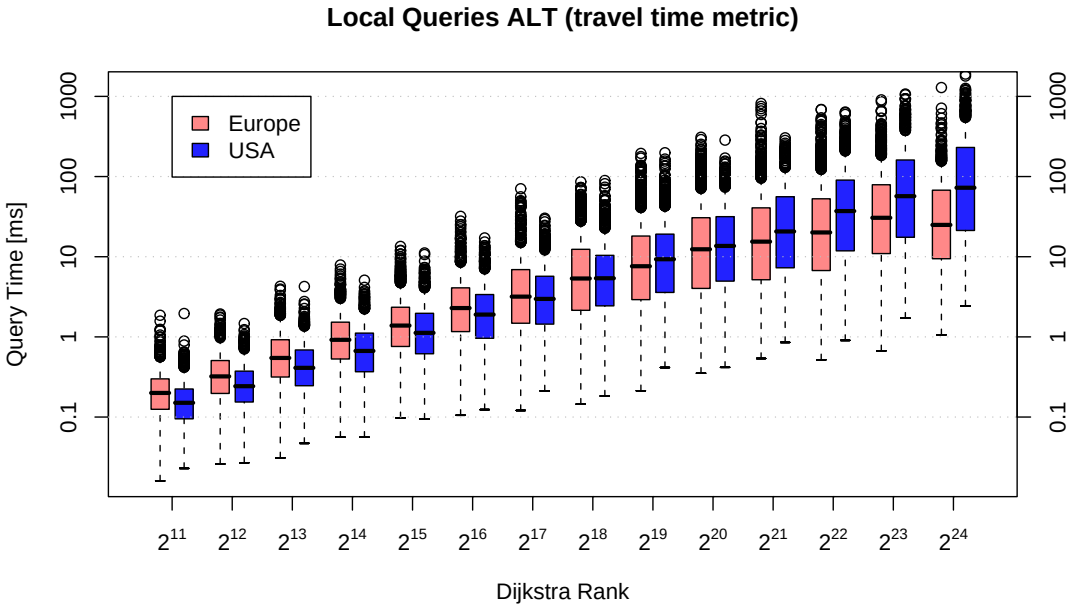


Fig. 5. Comparison of the query times using the Dijkstra rank methodology on the road network of Western Europe and the US. The landmarks are chosen from the level-3 core using maxCover. The results are represented as box-and-whisker plot [18]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

The fluctuations in query time both between different Dijkstra ranks and with fixed Dijkstra rank are so big that we had to use a logarithmic scale. Even typical query times vary by an order of magnitude for large Dijkstra ranks. The slowest queries for most Dijkstra ranks are two orders of magnitude slower than the median query times.

An interesting observation is also that for small ranks ALT is faster on the network of the US whereas for ranks higher than 2^{21}, queries are faster on the European network. A plausible explanation seems to be the different geometry of the two continents. Queries within the (pen)insulae of Iberia, Britain, Italy, or Scandinavia lack landmarks in many directions. For example, a user in Scotland might make the queer experience, that queries in north-south direction are consistently faster than queries in east-west direction (see Fig. 3). In contrast, long distance routes often have to go through bottlenecks which simplify search. In the US, such effects are rare.

5.3 Highway Hierarchies and A^* Search

Default Settings. Unless otherwise stated, we use the following default settings. After the level-5 core has been determined, the construction of the hierarchy is stopped. A complete distance table is computed on the level-5 core. For distance metrics, we stop at the level-6 core instead. We use 16 maxCover landmarks that have been computed in the level-3 core. The approximate query algorithm uses a maximum error rate of 10%, i.e., $\varepsilon = 0.1$.

Using a Distance Table and/or Landmarks. As described in Section 2, using a distance table can be seen as adding a very strong sense of goal direction after the core of the topmost level has been reached. If the highway query algorithm (without distance table) is enhanced by the ALT algorithm, the goal direction comes into effect much earlier. Still, the most considerable pruning effect occurs in the middle of rather long paths: close to the source and the target, the lower bounds are too weak to prune the search. Thus, both optimisations, distance tables and ALT, have a quite similar effect on the search space: using either of both techniques, in case of the European network with the *travel time metric*, the search space size is reduced from 1 662 to 916 (see Tab. 5). (Note that a slightly more effective reduction of the search space is obtained when all landmarks are used to compute lower bounds instead of selecting only one landmark for each direction, namely to 903 instead of 916.) When we consider other aspects like preprocessing time, memory usage, and query time, we can conclude that the distance table is somewhat superior to the landmarks optimisation. Since both techniques

Table 5. Comparison of all variants of the highway query algorithm using no optimisation (∅), a distance table (DT), ALT, or both techniques. Values in parentheses refer to *approximate* queries. Note that the *disk space* includes the memory that is needed to store the original graph.

		Europe				USA			
metric		∅	DT	ALT	both	∅	DT	ALT	both
time	preproc. time [min]	16	18	19	21	22	25	26	27
	total disk space [MB]	886	1 273	1 326	1 713	1 129	1 574	1 743	2 188
	#settled nodes	1 662	916	916	686 (176)	1 966	1 098	1 027	787 (162)
	query time [ms]	1.49	0.79	1.04	0.68 (0.21)	1.58	0.89	1.05	0.73 (0.21)
dist	preproc. time [min]	46	46	49	48	54	56	58	58
	total disk space [MB]	894	1 506	1 337	1 948	1 140	1 721	1 754	2 335
	#settled nodes	10 284	5 067	3 347	2 138 (177)	9 706	5 477	2 784	2 021 (169)
	query time [ms]	10.93	6.02	4.33	2.54 (0.30)	9.74	6.24	3.42	2.23 (0.33)

have a similar point of application, a combination of the highway query algorithm with both optimisations gives only a comparatively small improvement compared to using only one optimisation. In contrast to the exact algorithm, the approximate variant reduces the search space size and the query time considerably—e.g., to 19% and 27% in case of Europe (relative to using only the distance table optimisation)—, while guaranteeing a maximum error of 10% and achieving a total error of 0.056% in our random sample of 1 000 000 (s, t)-pairs (refer to Tab. 7).

Using a *distance metric*, ALT gets more effective and beats the distance table optimisation since much better lower bounds are produced: the negative effect described in Fig. 4 is weakened. Furthermore, in this case, a combination with both optimisations is worthwhile: the search space size and the query time are reduced to 42% in case of Europe (relative to using only the distance table optimisation). While the highway query algorithm enhanced with a distance table has 7.6 times slower query times when applied to the European graph with the distance metric instead of using the travel time metric, the combination with both optimisations reduces this performance gap to a factor of 3.7—or even 1.4 when the approximate variant is used.

Different Landmark Sets. In Tab. 6, we compare different sets of landmarks. Obviously, an increase of the number of landmarks improves the query performance. However, the rate of improvement is rather moderate so that using only 16 landmarks and thus, saving some memory and preprocessing time seems to be a good option. The quality of the selected landmarks is very similar for the two landmark selection methods that we have considered. Since the preprocessing times are similar as well, we prefer using the maxCover landmarks since they are slightly better.

Table 6. Comparison of the search spaces (in terms of number of settled nodes) of the highway query algorithm using different landmark sets. For each road network (with the travel time metric), the first column contains the search space size if the A^* search is *not* used. Values in parentheses refer to the search space sizes of approximate queries.

	Europe				USA			
#landmarks	0	16	24	32	0	16	24	32
core-1 avoid	916	687 (179)	665 (161)	651 (147)	1 098	808 (189)	762 (144)	736 (127)
core-3 maxCover		686 (176)	697 (177)	649 (140)		787 (162)	758 (134)	736 (121)

Local Queries. In Fig. 6, we compare the exact and the approximate HH^* search in case of the European network with the travel time metric. (For the US network the results are similar. We refer to Fig. 15 in Appendix A.) In the exact case, for Dijkstra ranks up to 2^{20}, we observe a continuous increase of the query times: since the distance between source and target grows, it takes longer till both search scopes meet. For greater Dijkstra ranks, we observe no significant further rise of the query times. This can be explained by the distance table that bridges the gap between the forward and backward search for long-distance queries very efficiently, no matter whether we deal with a long or a very long path.

Up to a Dijkstra rank of 2^{18}, the approximate variant shows a very similar behaviour—even though at a somewhat lower level. Then, the query times *decrease*, reaching very small values for very long paths (Dijkstra ranks 2^{22}–2^{24}). This is due to the fact that the *relative* inaccuracy of the lower bounds, which is crucial for the stop condition of the approximate

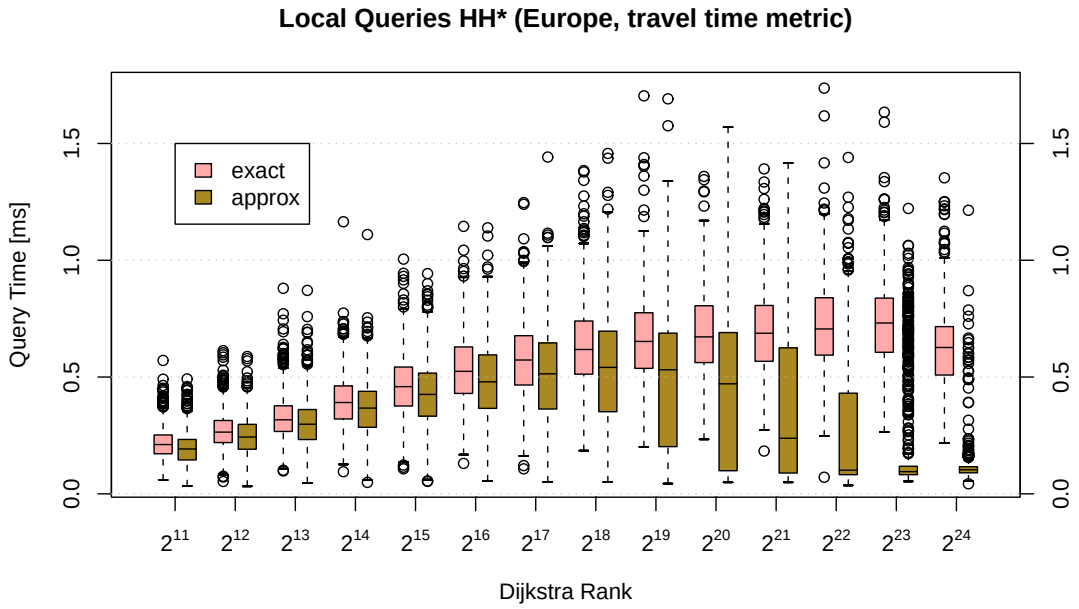


Fig. 6. Comparison of the query times of the exact and the approximate HH^* search using the Dijkstra rank methodology.

algorithm, is less distinct for very long paths: hence, most of the time, the lower bounds are sufficiently strong to stop very early. However, the large number and high amplitude of outliers indicates that sometimes goal direction does not work well even for approximate queries.

Approximation Error. Figure 7 shows the actual distribution of the approximation error for a random sample in the European network with the travel time metric, grouped by Dijkstra rank. (For the European network with the distance metric and the US network with both metrics, see Figs. 16–18 in Appendix A.) For paths up to a moderate length (Dijkstra rank 2^{16}), at least 99% of all queries in the random sample returned an accurate result. Only very few queries approach the guaranteed maximum error rate of 10%. For longer paths, still more than 94% of the queries give the correct result, and almost 99% of the queries find paths that are at most 2% longer than the shortest path. The fact that we get more errors for longer paths corresponds to the running times depicted in Fig. 6: in the case of large Dijkstra ranks, we usually stop the search quite early, which increases the likelihood of an inaccuracy.

While the approximate variant of the ALT algorithm gives only a small speedup (compare Fig. 5 with Fig. 8 in Appendix A) and produces a considerable amount of inaccurate results (in particular for short paths, see Figs. 11 and 13), the approximate HH^* algorithm is much faster than the exact version (in particular for long paths) and produces a comparatively small amount of inaccurate results. This difference is mainly due to the distance table, which allows a fast determination of upper bounds—and thus, in many cases early aborts—and provides accurate long-distance subpaths, i.e., the only thing that can go wrong is that the search processes in the local area around source and target do not find the right core entrance points.

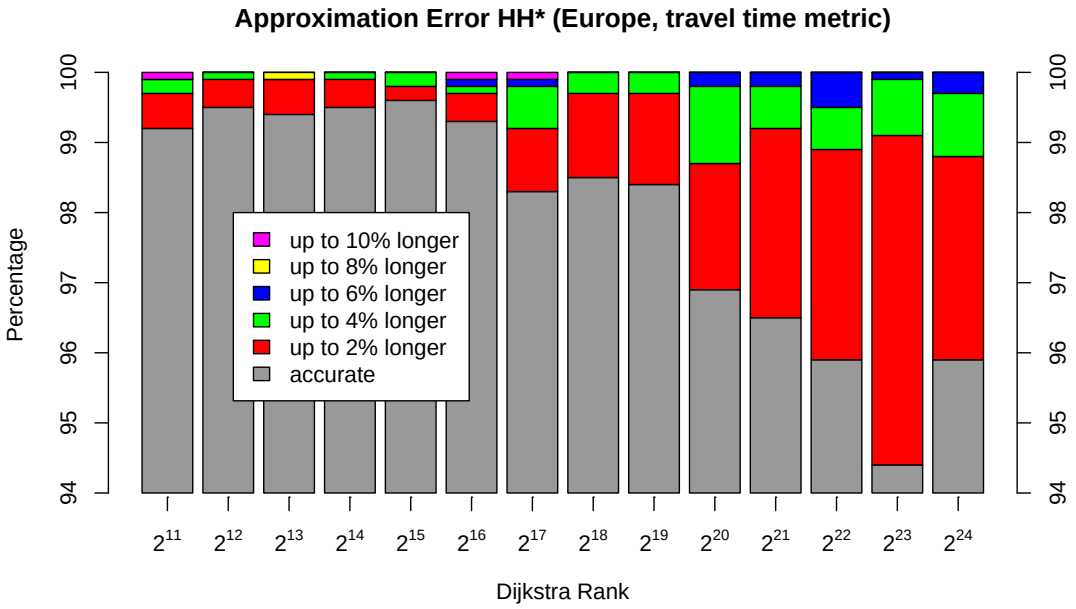


Fig. 7. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 94%, i.e., at least 94% of all queries returned an accurate result.

In Tab. 7, we compared the effect of different maximum error rates ε. We obtained the expected result that a larger maximum error rate reduces the search space size considerably. Furthermore, we had a look at the actual error that occurs in our random sample: we divided the sum of all path lengths that were obtained by the approximate algorithm by the sum of the shortest path lengths. We find that the resulting total error is *very* small, e.g., only 0.056% in case of the European network with the travel time metric when we allow a maximum error rate of 10%. Similar to the results in Section 5.2, we observe that the total error and the percentage of inaccurate queries (see Figs. 16 and 18) are much higher when using the distance metric instead of the travel time metric.

Table 7. Comparison of different maximum error rates ε. By the *total error*, we give the sum of the path lengths obtained by the approximate algorithm divided by the sum of the shortest path lengths. Note that these values are given in percent. This table is based on 1 000 000 random (s, t)-pairs (instead of the usual 10 000 pairs).

		Europe						USA					
metric	ε [%]	0	1	2	5	10	20	0	1	2	5	10	20
time	#settled nodes	685	612	523	319	177	103	784	632	516	307	162	86
	total error [%]	0	0.0002	0.0015	0.018	0.056	0.112	0	0.0013	0.0073	0.034	0.082	0.144
dist	#settled nodes	2131	1302	843	333	184	143	2021	1101	672	277	169	134
	total error [%]	0	0.0112	0.0383	0.172	0.329	0.526	0	0.0108	0.0441	0.132	0.193	0.240

Complete Description of the Shortest Path. So far, we have reported only the times needed to compute the shortest path *distance* between two nodes. Now, we determine a complete description of the shortest path. In Tab. 8 we give the additional preprocessing time and the additional disk space for the unpacking data structures. Furthermore, we report the additional

time that is needed to determine a complete description of the shortest path and to traverse[6] it summing up the weights of all edges as a sanity check—assuming that the distance query has already been performed. That means that the total average time to determine a shortest path is the time given in Tab. 8 plus the query time given in previous tables. We can conclude that even Variant 3 uses comparatively little preprocessing time and space. With Variant 3, the time for outputting the path remains considerably smaller than the query time itself and a factor 3–5 smaller than using Variant 2. The USA graph profits more than the European graph since it has paths with considerably larger hop counts, perhaps due to a larger number of degree two nodes in the input. Note that due to cache effects, the time for outputting the path using preprocessed shortcuts is likely to be considerably smaller than the time for traversing the shortest path in the original graph.

Table 8. Additional preprocessing time, additional disk space and query time that is needed to determine a complete description of the shortest path and to traverse it summing up the weights of all edges—assuming that the query to determine its lengths has already been performed. Moreover, the average number of hops—i.e., the average path length in terms of number of nodes—is given. These figures refer to experiments on the graphs with the travel time metric. Note that the experiments for Variant 1 have been performed without using a distance table for the topmost level.

	Europe				USA			
	preproc. [s]	space [MB]	query [ms]	# hops (avg.)	preproc. [s]	space [MB]	query [ms]	# hops (avg.)
Variant 1	0	0	16.70	1 370	0	0	40.64	4 537
Variant 2	71	112	0.45	1 370	71	134	1.32	4 537
Variant 3	75	180	0.17	1 370	75	200	0.27	4 537

6 Discussion

We have learned a few things about landmark A^* (ALT) that are interesting independently of highway hierarchies. We have explained why the lower bounds provided by ALT are often quite weak and why there are very high fluctuations in query performance. There are also considerable differences between Western Europe and the US. In Europe, we have *larger* execution times for local queries than in the US whereas for long range (average case) queries, times are *smaller*. Executing landmark selection on a graph where sparse subgraphs have been contracted is profitable in terms of preprocessing time even if we do not want highway hierarchies. Similarly, storing distances to landmarks only on this contracted graph considerably reduces the space overhead of ALT.

For highway hierarchies we have learned that they can also handle the case of travel distances. Compared to the case of travel times, space consumption is about the same whereas preprocessing time and query time increase by a factor of about three. It is to be expected that any other cost metric that represents some compromise of travel time, distance, fuel consumption and tolls will have performance somewhere within this range. Highway hierarchies can be augmented to output shortest paths in a time below the time needed for computing the distances.

There is a complex interplay between highway hierarchies and the optimisations of distance tables and ALT. For exact queries using the travel time metric, distance tables are a

[6] Note that we do *not* traverse the path in the original graph, but we directly scan the assembled description of the path.

better investment into preprocessing time and space than ALT. One incompatibility between highway hierarchies and ALT is that the search cannot be stopped when search frontiers meet. For approximate queries or for the distance metric, all three techniques work together very well yielding a speedup around four over highway hierarchies alone: Highway hierarchies save space and time for landmark preprocessing; distance tables obviate search in higher levels and allow simpler and faster ALT search with very effective goal direction. ALT provides good pruning opportunities for the distance metric and an excellent sense of goal direction for approximate queries yielding high quality routes most of the time while never computing very bad routes.

An interesting route of future research is to consider a combination of highway hierarchies with geometric containers or edge flags [10–12]. Highway hierarchies might harmonise better with these methods than with ALT because similar to highway hierarchies they are based on truncating search at certain edges. There is also hope that their high preprocessing costs might be reduced by exploiting the highway hierarchy.

Very recently, *transit node routing* (TNR) and related approaches [19, 20] has accelerated shortest path queries by another two orders of magnitude. Roughly, TNR precomputes shortest path distances to *access points* in a transit node set T (e.g., the nodes at the highest level of the highway hierarchy). During a query between "sufficiently distant" nodes, a distance table for T can be used to bridge the gap between the access points of source and target. However, TNR needs considerably more preprocessing time than the approach described in this paper. Furthermore, the currently best implementation of TNR uses highway hierarchies for preprocessing and local queries. It is likely that also landmarks might turn out to be useful in future versions of TNR. On the one hand, landmarks yield lower bounds that can be used for *locality filters* needed in TNR. On the other hand, the precomputed distances to access points could be used as landmark information for speeding up local search.

Acknowledgements

We would like to thank Timo Bingmann for work on visualisation tools, which were very helpful.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on System Science and Cybernetics **4** (1968) 100–107
3. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
4. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering and Experimentation. (2005) 26–40
5. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms (ESA). Volume 3669 of LNCS., Springer (2005) 568–579
6. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms (ESA). Volume 4168 of LNCS., Springer (2006) 804–816 to appear.
7. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. (2004) 100–111
8. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami (2006) 129–143
9. Maue, J., Sanders, P., Matijevic, D.: Goal directed shortest path queries using Precomputed Cluster Distances. In: 5th Workshop on Experimental Algorithms (WEA). Number 4007 in LNCS, Springer (2006) 316–328

10. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: 11th European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003) 776–787
11. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. (2005) 189–202
12. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung. Volume 22., IfGI prints, Institut für Geoinformatik, Münster (2004) 219–230
13. 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. IEEE. (1994)
14. Sedgewick, R., Vitter, J.S.: Shortest paths in Euclidean space. Algorithmica **1** (1986) 31–48
15. Willhalm, T.: Engineering Shortest Path and Layout Algorithms for Large Graphs. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik (2005)
16. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html (2002)
17. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/∼challenge9/ (2006)
18. R Development Core Team: R: A Language and Environment for Statistical Computing. http://www.r-project.org (2004)
19. Müller, K.: Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs. Master's thesis, Universtät Karlsruhe (2006) supervised by D. Delling, M. Holzer, F. Schulz, and D. Wagner.
20. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant time shortest-path queries in road networks. submitted for publication. (2006)

A Further Experiments

Table 9. Overview of the preprocessing for different selection strategies on the US network. All figures are given in minutes of computation time. Generating 16 maxCover landmarks on the whole graph requires more than 4 GB RAM. Therefore, these landmarks were generated on an AMD Opteron Processor 252 clocked at 2.6 GHz with 16 GB main memory.

		full graph			core-1			core-2			core-3		
metric	preproc. [min]	avoid	adv.av.	maxCov	avoid	adv.av.	maxCov	avoid.	adv.av.	maxCov	avoid	adv.av.	maxCov
	highway info	–	–	–	3.4	3.4	3.4	14.9	14.9	14.9	18.5	18.5	18.5
time	selection	20.5	30.5	105.2	3.1	4.5	28.4	0.5	0.7	5.6	0.1	0.2	1.2
	distances	–	–	–	7.1	7.1	7.1	7.1	7.1	7.1	7.1	7.1	7.1
	highway info	–	–	–	3.1	3.1	3.1	17.4	17.4	17.4	26.3	26.3	26.3
dist	selection	18.3	26.4	97.2	2.9	4.2	28.2	0.6	0.9	5.8	0.2	0.2	1.5
	distances	–	–	–	5.8	5.8	5.8	5.8	5.8	5.8	5.8	5.8	5.8

Table 10. Comparison of the exact and approximate ALT algorithm. The landmarks are taken from the full graph. The figures are based on 1 000 random queries on 10 different sets of 16 landmarks.

		Europe			USA		
		#settled nodes		inaccurate queries	#settled nodes		inaccurate queries
metric		exact	approx.	min – max	exact	approx.	min – max
	avoid	93 520	81 582	9.8% – 11.9%	220 333	206 165	7.4% – 10.1%
time	adv.av.	86 340	74 706	9.3% – 12.6%	210 703	194 920	7.6% – 9.6%
	maxCover	75 220	63 112	10.7% – 11.7%	175 359	161 230	7.6% – 9.6%
	avoid	253 552	225 618	31.5% – 38.4%	308 823	289 701	24.8% – 29.9%
dist	adv.av.	256 511	227 779	30.9% – 38.0%	302 521	282 410	24.3% – 29.3%
	maxCover	230 110	203 564	31.3% – 34.9%	282 162	265 091	27.3% – 22.3%

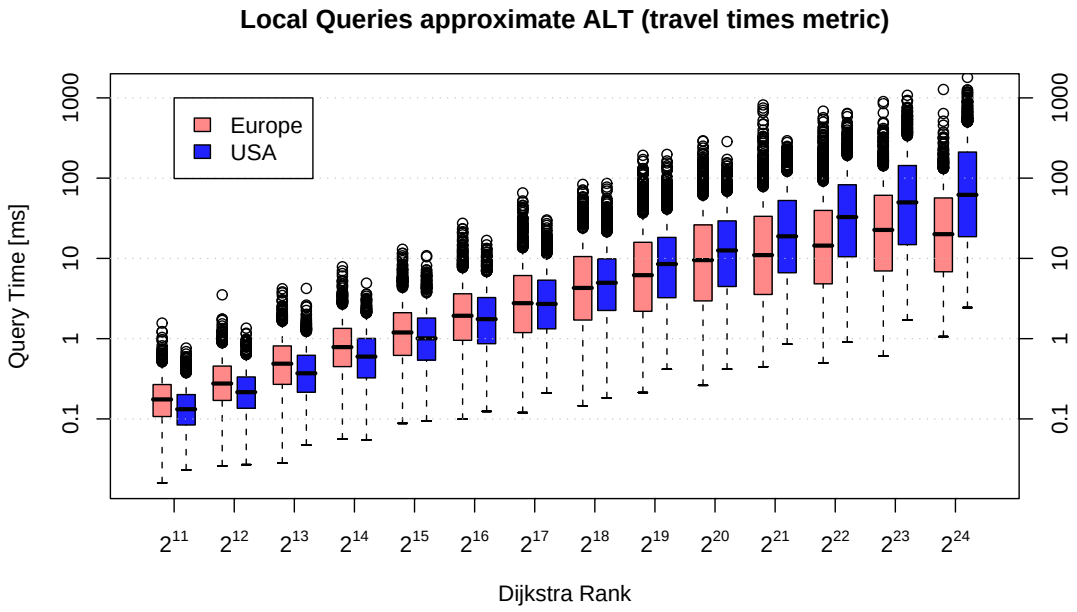


Fig. 8. Comparison of the query times on the road network of Western Europe and the USA using the approximate ALT algorithm. The landmarks are chosen from the core-3 using maxCover.

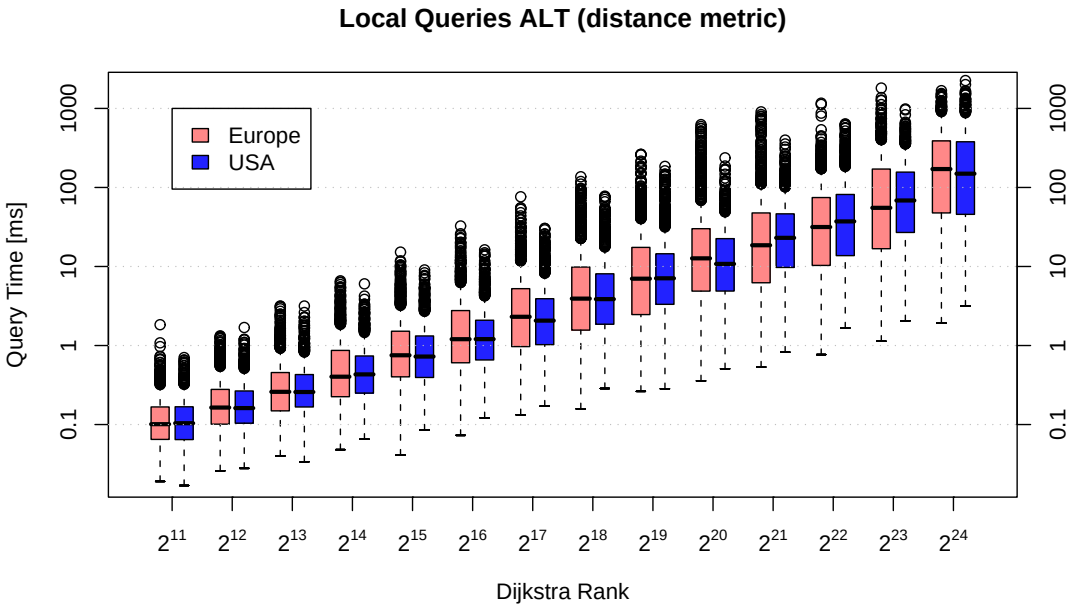


Fig. 9. Comparison of the query times on the road network of Western Europe and the USA using the ALT algorithm. The landmarks are chosen from the core-3 using maxCover.

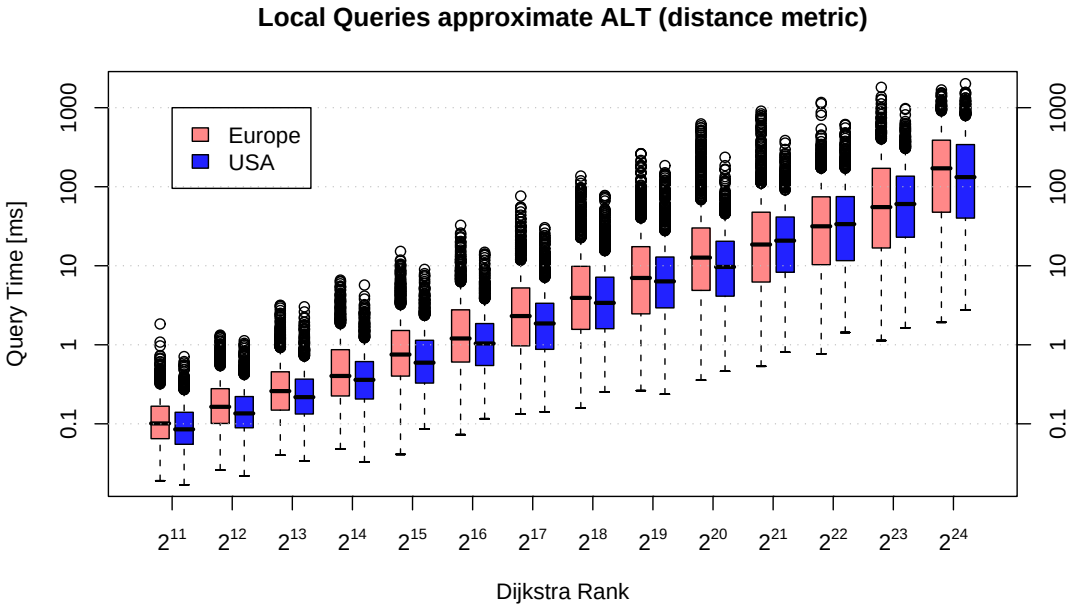


Fig. 10. Comparison of the query times on the road network of Western Europe and the USA using the approximate ALT algorithm. The landmarks are chosen from the core-3 using maxCover.

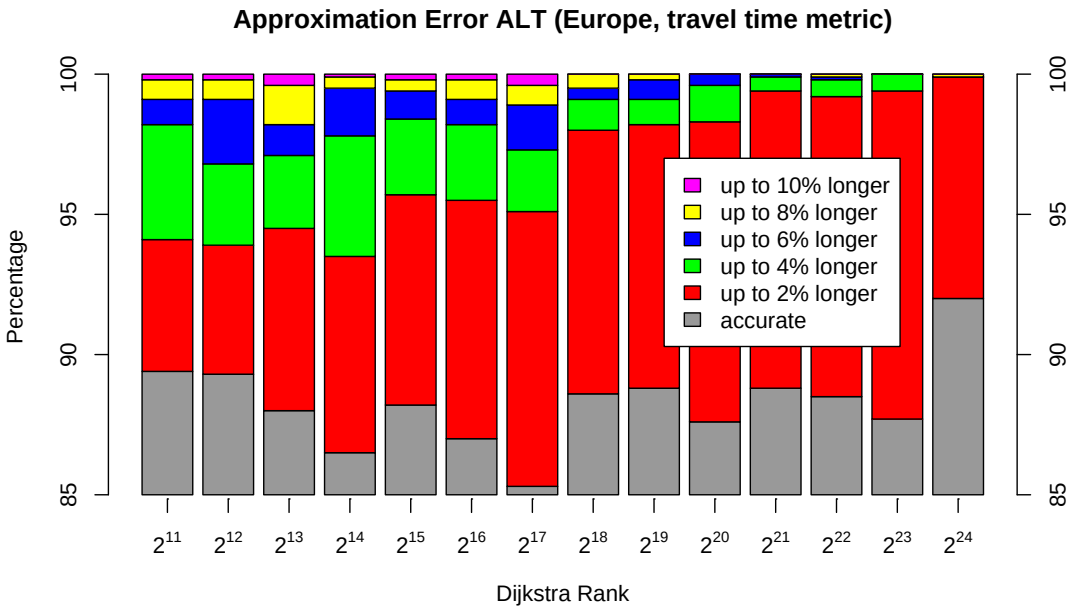


Fig. 11. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 85%, i.e., at least 50% of all queries returned an accurate result.

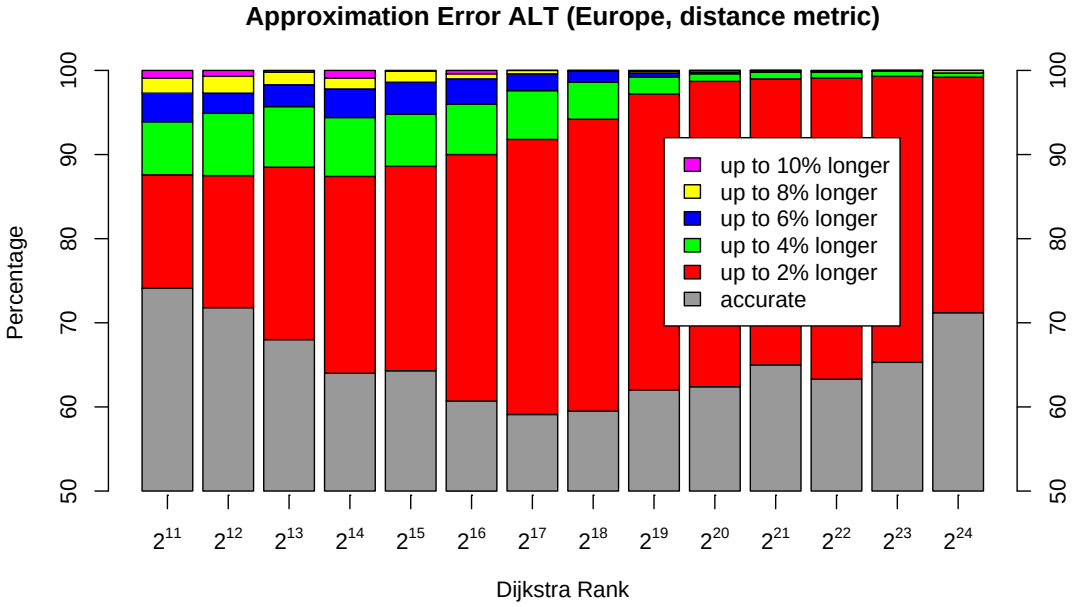


Fig. 12. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 50%, i.e., at least 50% of all queries returned an accurate result.

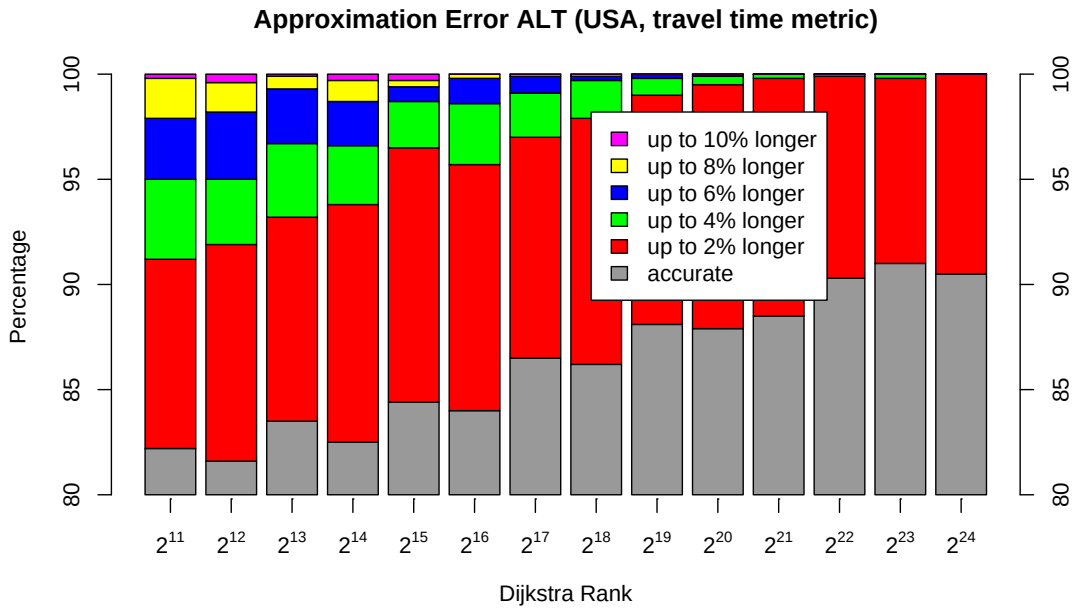


Fig. 13. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 80%, i.e., at least 80% of all queries returned an accurate result.

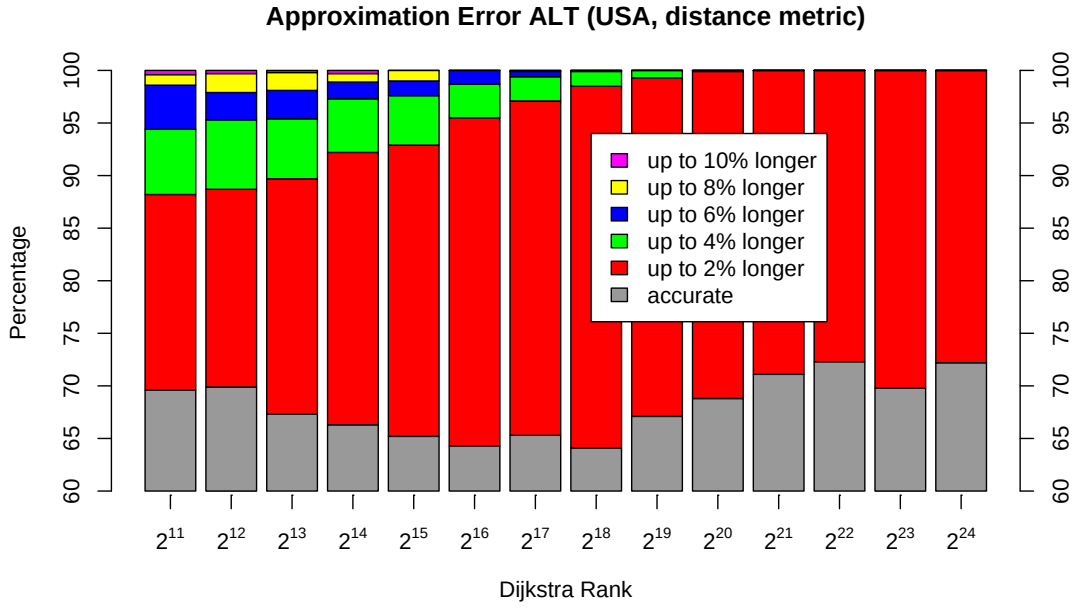


Fig. 14. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 60%, i.e., at least 60% of all queries returned an accurate result.

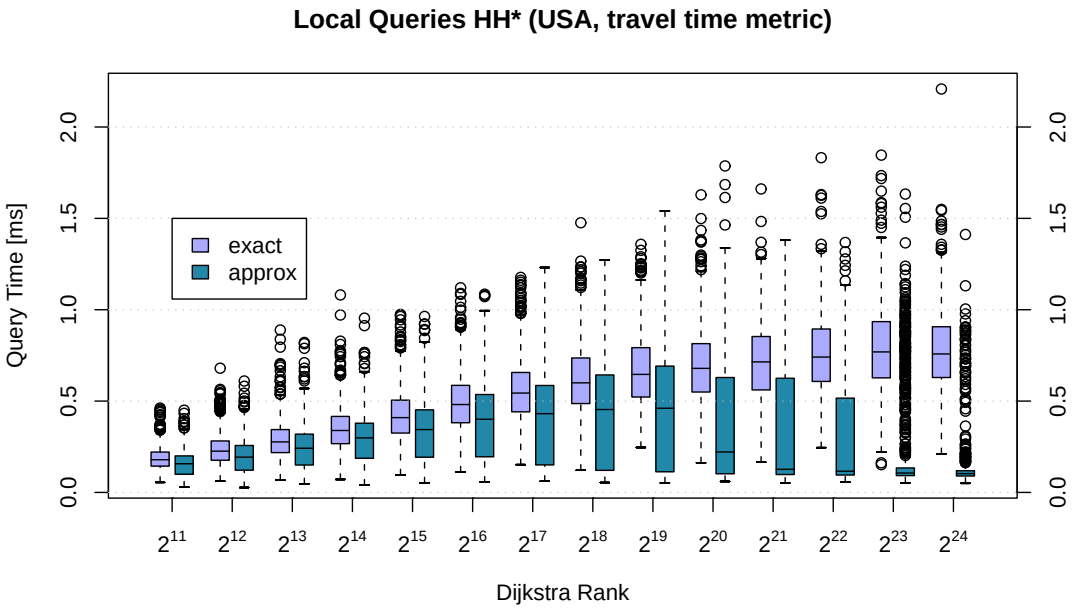


Fig. 15. Comparison of the query times of the exact and the approximate HH* search.

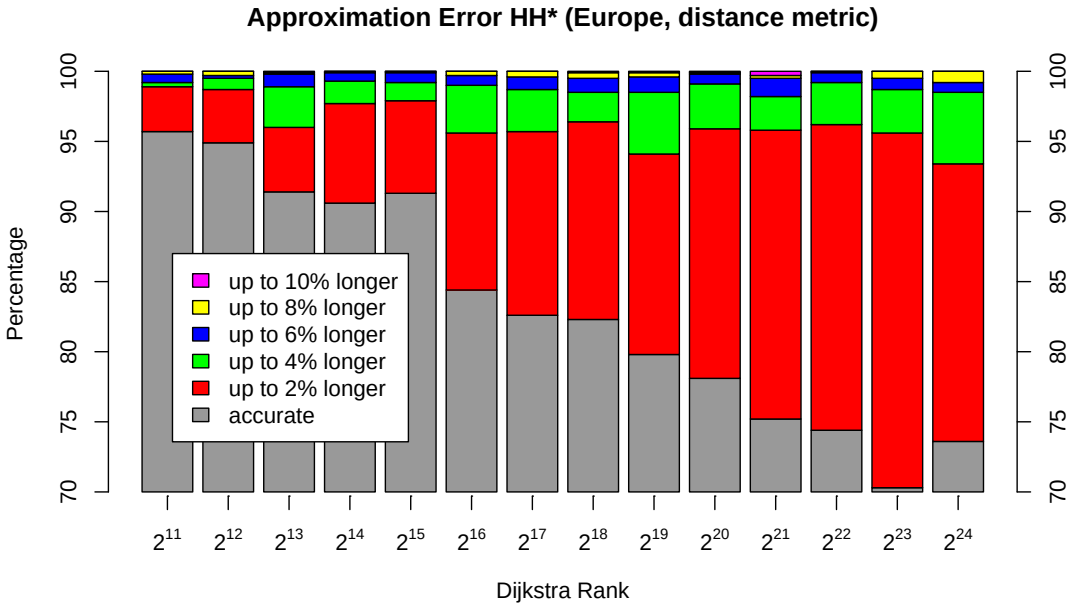


Fig. 16. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 70%, i.e., at least 70% of all queries returned an accurate result.

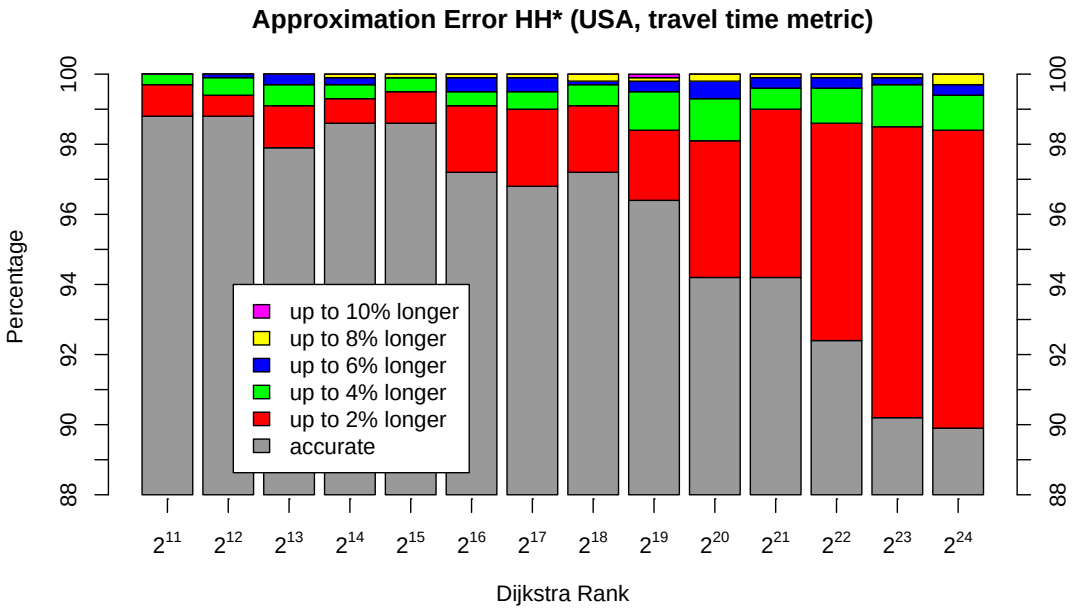


Fig. 17. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 88%, i.e., at least 88% of all queries returned an accurate result.

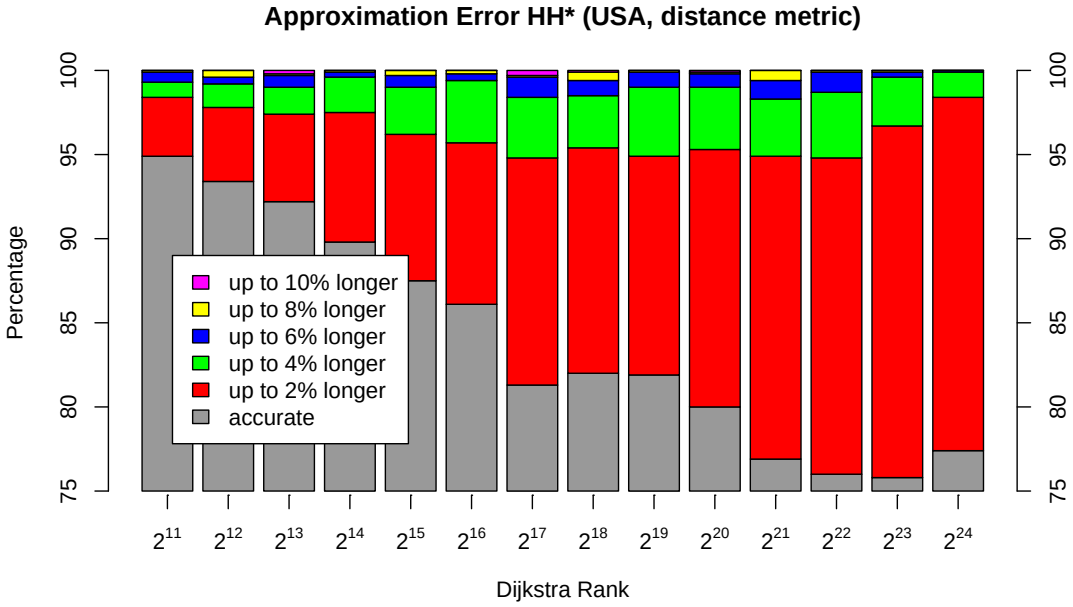


Fig. 18. Actual distribution of the approximation error for a random sample, grouped by Dijkstra rank. Note that, in order to increase readability, the y-axis starts at 75%, i.e., at least 75% of all queries returned an accurate result.