Better Landmarks within Reach

Andrew V. Goldberg[*] Haim Kaplan[†] Renato F. Werneck[‡]

Abstract

We study the REAL algorithm for the point-to-point shortest path problem. It combines A^* search search, landmark-based lower bounds, and reach-based pruning. We suggest several improvements to the preprocessing and query phases of the algorithm. In particular, we propose maintaining landmark distances only for high-reach vertices, saving space that can be used to accommodate more landmarks. This leads to a wide range of time-space tradeoffs. The new algorithm is very effective on the road networks of Europe and the USA, each with about 20 million vertices: it can find a random shortest path in one millisecond after one or two hours of preprocessing, using either transit times or travel distances as edge weights. Besides road networks, we also obtain improved performance on two-dimensional grids with random arc lengths.

1 Introduction

We study the *point-to-point shortest path problem* (P2P): given a directed graph $G = (V, A)$ with nonnegative arc lengths and two vertices, the source s and the destination t, find a shortest path from s to t. We allow preprocessing, but limit the size of the precomputed data to a (moderate) constant times the input graph size. Preprocessing time is limited by practical considerations. For example, in our motivating application, driving directions on large road networks, quadratic-time algorithms are impractical. We are interested in exact shortest paths only.

Finding shortest paths is a fundamental problem. The single-source problem with nonnegative arc lengths has been studied most extensively [1, 3, 5, 6, 10, 11, 12, 13, 17, 22, 29, 39, 43]. Near-optimal algorithms are known both in theory, with near-linear time bounds, and in practice, where running times are within a small constant factor of the breadth-first search time.

The P2P problem with no preprocessing has been addressed, for example, in [21, 32, 37, 44]. While no nontrivial theoretical results are known for the general P2P problem, there has been work on the special case of undirected planar graphs with slightly super-linear preprocessing space. The best bound in this context appears in [9]. Algorithms for approximate shortest paths that use preprocessing have been studied; see e.g. [2, 23, 40].

Previous work on exact algorithms with preprocessing includes, e.g., [14, 18, 16, 19, 24, 27, 30, 33, 34, 35, 36, 42]. We focus our discussion here on the most relevant recent developments in preprocessing-based algorithms for road networks. Such methods have two components: a *preprocessing algorithm* that computes auxiliary data and a *query algorithm* that computes an answer for a given s-t pair.

[*]Microsoft Research Silicon Valley, USA. E-mail: goldberg@microsoft.com.

[†]School of Mathematical Sciences, Tel Aviv University, Israel. E-mail: haimk@post.tau.ac.il. Part of this work was done while this authors was visiting Microsoft Research Silicon Valley.

[‡]Microsoft Research Silicon Valley, USA. E-mail: renatow@microsoft.com

Gutman [19] defines the notion of *vertex reach.* Informally, the reach of a vertex v is a number that is large if v is close to the middle of a long shortest path and small otherwise. Gutman shows how to prune an s-t search based on (upper bounds on) vertex reaches and (lower bounds on) vertex distances from s and to t. He uses Euclidean distances as lower bounds, and observes that the idea of reach can be combined with Euclidean-based A^* search to improve efficiency. A^* search uses lower bounds on the distance to the destination to direct the search towards it.

Goldberg and Harrelson [14] (see also [18]) have shown that the performance of A^* (without reaches) can be significantly improved if Euclidean lower bounds are replaced by landmark-based lower bounds. These bounds are obtained by storing (in the preprocessing step) the distances between every vertex and a small (constant-sized) set of special vertices, the landmarks. During queries, one can use this information, together with the triangle inequality, to obtain lower bounds on the distances between any two vertices in the graph. This leads to the ALT (A^* search, landmarks, and triangle inequality) algorithm for the point-to-point problem.

Sanders and Schultes [33] introduce the notion of highway hierarchy and use it to design efficient algorithms for the shortest path problem on road networks. Their latest version [34] is the best overall solution for the problem in the literature. We refer to the algorithms based on highway hierarchy as the HH algorithms. While building the hierarchy structure, these algorithms use a notion of *shortcuts*, which is also useful in other contexts.

In [16], we show how to apply shortcuts in the context of reach-based algorithms to significantly improve both preprocessing and query efficiency. The resulting algorithm is called RE. The paper also shows how the ALT method can be combined with reach pruning in a natural way, leading to an algorithm called REAL. In this paper, we continue our study of reach-based point-to-point shortest paths algorithms, and their combination with A^* search using landmarks.

We introduce the following improvements:

Reach-aware landmarks: In REAL, unless s and t are close to each other, the search visits mostly vertices with high reach. Thus it is more important to have good lower bounds (on distances from the source and to the target) for high-reach vertices. In fact, as we suggested in [16], one can keep landmark data for high-reach vertices only. This can significantly reduce the memory requirements of the algorithm while slightly increasing query times. The space saved by not keeping distances from landmarks to low-reach vertices can be used to store more landmarks instead, thus improving query performance. This allows a wide range of time-memory trade-offs. Of particular interest is the fact that we are able to reduce both time and space requirements for large graphs. We do this by maintaining landmark data for a moderate fraction of vertices and increasing the number of landmarks by a smaller factor.

Shortcuts: Motivated by the work of Sanders and Schultes [34], we develop a preprocessing algorithm that shortcuts vertices of arbitrary (but usually low) degree, instead of just degree-two vertices as our previous method did. This improves both preprocessing and query performance.

Reach computation: We propose an algorithm for computing exact reaches that in practice is more efficient than the standard algorithm, which builds a complete shortest path tree from every vertex in the graph. While it is still not quite practical for large road networks, the improved algorithm does help accelerate the approximate reach computation. We also suggest some slight modifications to our partial tree algorithm (used to compute approximate reaches) that significantly accelerate it in practice.

Improving locality: Since a shortest path computation is largely concentrated on a small number of high-reach vertices, reordering vertices by reach leads to more locality and improved performance.

Readers familiar with our previous paper [16] can skip Sections 2–5, which contain basic definitions and review the algorithms based on reaches and A^* search. We describe our improvements in Section 6 and give experimental results in Section 7.

The bulk of our experiments is on road networks. We conducted experiments on the graph of Western Europe, USA, and subgraphs of USA, using both transit times and travel distances as edge lengths. Experiments confirm that our algorithmic improvements lead to substantial savings in time (especially in preprocessing) and memory. We have practical results for both metrics: both Europe and the USA can be preprocessed in one to two hours and queries with our fastest algorithm take roughly one millisecond.

We have also obtained good results for 2-dimensional grids with random arc lengths. Although not as good as for road networks, the results show that our techniques have more general applicability. To show the limitations of our techniques, we also experimented with 3-dimensional grids. Preprocessing for reach-based algorithms becomes expensive, and query speed-up is small. For higher dimensions, preprocessing becomes impractical, with no gains in query times. For A^* search, preprocessing remains fast for all dimensions, but query times become worse as the dimension increases. The same is true for random graphs.

2 Preliminaries

The input to the preprocessing stage of a P2P algorithm is a directed graph $G = (V, A)$ with n vertices and m arcs, and nonnegative lengths $\ell(a)$ for every arc a. The query stage also gets as inputs a source s and a sink t. The goal is to find a shortest path from s to t. We denote by $\text{dist}(v, w)$ the shortest-path distance from vertex v to vertex w with respect to ℓ. In general, $\text{dist}(v, w) \neq \text{dist}(w, v)$.

The labeling method for the shortest path problem [25, 26] finds shortest paths from the source to all vertices in the graph. The method works as follows (see e.g. [38]). It maintains for every vertex v its distance label $d(v)$, parent $p(v)$, and status $S(v) \in \{\texttt{unreached}, \texttt{labeled}, \texttt{scanned}\}$. Initially $d(v) = \infty$, $p(v) = nil$, and $S(v) = \texttt{unreached}$ for every vertex v. The method starts by setting $d(s) = 0$ and $S(s) = \texttt{labeled}$. While there are labeled vertices, the method picks a labeled vertex v, *relaxes* all arcs out of v, and sets $S(v) = \texttt{scanned}$. To relax an arc (v, w), one checks if $d(w) > d(v) + \ell(v, w)$ and, if true, sets $d(w) = d(v) + \ell(v, w)$, $p(w) = v$, and $S(w) = \texttt{labeled}$.

If the length function is nonnegative, the labeling method terminates with correct shortest path distances and a shortest path tree. Its efficiency depends on the rule to choose a vertex to scan next. We say that $d(v)$ is *exact* if it is equal to the distance from s to v. If one always selects a vertex v such that, at selection time, $d(v)$ is exact, then each vertex is scanned at most once. Dijkstra [6] (and independently Dantzig [3]) observed that if ℓ is nonnegative and v is a labeled vertex with the smallest distance label, then $d(v)$ is exact. We refer to the labeling method with the minimum label selection rule as *Dijkstra's algorithm.* If ℓ is nonnegative then Dijkstra's algorithm scans vertices in nondecreasing order of distance from s and scans each vertex at most once.

For the P2P case, note that when the algorithm is about to scan the sink t, we know that $d(t)$ is exact and the s-t path defined by the parent pointers is a shortest path. We can terminate the algorithm at this point. Intuitively, Dijkstra's algorithm searches a ball with s in the center and t on the boundary.

One can also run Dijkstra's algorithm on the *reverse graph* (the graph with every arc reversed) from the sink. The reverse of the t-s path found is a shortest s-t path in the original graph.

The *bidirectional algorithm* [3, 8, 31] alternates between running the forward and reverse versions of Dijkstra's algorithm, each maintaining its own set of distance labels. We denote by $d_f(v)$ the distance label of a vertex v maintained by the forward version of Dijkstra's algorithm, and by $d_r(v)$ the distance label of a vertex v maintained by the reverse version. (We will still use $d(v)$ when the direction would not matter or is clear from the context.) During initialization, the forward search scans s and the reverse search scans t. The algorithm also maintains the length of the shortest path seen so far, μ, and the corresponding path. Initially, $\mu = \infty$. When an arc (v, w) is relaxed by the forward search and w has already been scanned by the reverse search, we know the shortest s-v and w-t paths have lengths $d_f(v)$ and $d_r(w)$, respectively. If $\mu > d_f(v)+\ell(v,w)+d_r(w)$, we have found a path shorter than those seen before, so we update μ and its path accordingly. We perform similar updates during the reverse search. The algorithm terminates when the search in one direction selects a vertex already scanned in the other. Intuitively, the bidirectional algorithm searches two touching balls centered at s and t.

3 Reach-Based Pruning

Given a path P from s to t and a vertex v on P, the *reach of v with respect to P* is the minimum of the length of the prefix of P (the subpath from s to v) and the length of the suffix of P (the subpath from v to t). The *reach* of v, $r(v)$, is the maximum, over all **shortest** paths P through v, of the reach of v with respect to P.

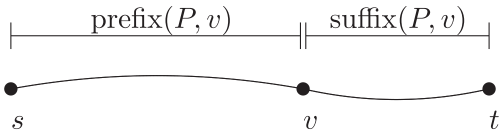


Figure 1: The reach of v with respect to the shortest path P between s and t is the minimum between the lengths of its prefix and its suffix (with respect to v).

For large graphs, computing exact reaches is impractical with current algorithms. Efficient solutions use heuristics to compute only an upper bound on the reach of each vertex. We denote an upper bound on $r(v)$ by $\overline{r}(v)$. Let $\underline{\text{dist}}(v, w)$ denote a lower bound on the distance from v to w. The following fact allows using reaches for pruning Dijkstra's search:

> Suppose $\overline{r}(v) < \underline{\text{dist}}(s, v)$ and $\overline{r}(v) < \underline{\text{dist}}(v, t)$. Then v is not on a shortest path from s to t, and therefore Dijkstra's algorithm does not need to label or scan v.

Note that this holds for the bidirectional algorithm as well.

3.1 Queries Using Upper Bounds on Reaches

In this section we give a quick summary of reach-based query algorithms based on bidirectional Dijkstra's algorithm. For more details, see [15].

As described in Section 3, to prune the search based on the reach of some vertex v, we need a lower bound on the distance of v to the source and a lower bound on the distance of v to the sink.

One way is to use lower bounds implicit in the search itself. Consider the search in the forward direction, and let γ be the smallest distance label of a labeled vertex in the reverse direction (i.e., the topmost label in the reverse heap). If a vertex v has not been scanned in the reverse direction, then γ is a lower bound on the distance from v to the destination t. (The same idea applies to the reverse search: we use the topmost label in the forward heap as a lower bound for unscanned vertices in the reverse direction.) When we are about to scan v we know that $d_f(v)$ is the distance from the source to v. So we can prune the search at v if v has not been scanned in the reverse direction, $\bar{r}(v) < d_f(v)$, and $\bar{r}(v) < \gamma$. When using these bounds, the stopping condition is the same as for the standard bidirectional algorithm (without pruning). We call the resulting procedure the *bidirectional bound* algorithm. See Figure 2.

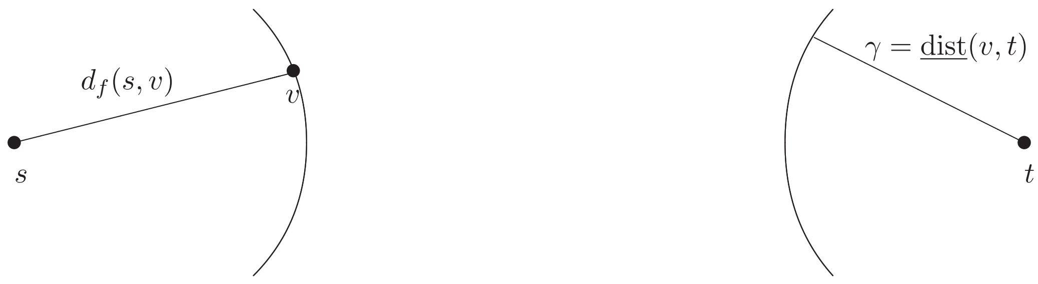


Figure 2: Pruning using implicit bounds. Assume v is about to be scanned in the forward direction, has not yet been scanned in the reverse direction, and that the smallest distance label of a vertex not yet scanned in the reverse direction is γ. Then v can be pruned if $\bar{r}(v) < d_f(v)$ and $\bar{r}(v) < \gamma$.

An alternative is to use the distance label itself for pruning. Assume we are about to scan a vertex v in the forward direction (the procedure in the reverse direction is similar). If $\overline{r}(v) < d_f(v)$, we prune the vertex. Note that if the distance from v to t is at most $\overline{r}(v)$, the vertex will still be scanned in the reverse direction, given the appropriate stopping condition. It is easy to see that the following stopping condition works.

> Stop the search in a given direction when either there are no labeled vertices or the minimum distance label of labeled vertices for the corresponding search is at least half the length of the shortest path seen so far.

We call this the *self-bounding algorithm*. The reason why this algorithm can safely ignore the lower bound to the destination is that it leaves to the other search to visit vertices that are closer to it. Note, however, that when scanning an arc (v,w), even if we end up pruning w, we must check if w had been scanned in the opposite direction and, if so, check if the candidate path using (v,w) is the shortest path seen so far.

The following natural algorithm falls into both of the above categories. It balances the radii of the forward search and the reverse search by scanning in each iteration the labeled vertex with minimum distance label, considering both directions. Note that the distance label of this vertex is also a lower bound on the distance to the target, as the search in the opposite direction has not selected the vertex yet. The algorithm can be implemented with only one priority queue. We refer to this algorithm as *distance-balanced.* Note that one could also use explicit lower bounds in combination with the implicit bounds.

3.2 Preprocessing

The preprocessing algorithm computes upper bounds on reaches and adds *shortcut* arcs to the graph to reduce the reaches of some vertices. Lowering vertex reaches makes pruning more effective during queries, thereby reducing the number of scanned vertices. Our algorithm is similar to the preprocessing algorithm described in [16] with several improvements which we describe in Sections 6.3, 6.4 and 6.5. The remainder of this section gives an overview of the preprocessing of [16].

The algorithm in fact computes upper bounds on the reaches of *arcs*, not vertices. Let P be the shortest path from s to t, and assume it contains an arc (v, w). The *reach of* (v, w) *with respect to* P is the minimum between the distance from s to w and the distance from v to t. The reach of (v, w) (with respect to the entire graph) is the maximum over all shortest paths P of the reach of (v, w) with respect to P. Eventually, when all arc reaches are computed, the algorithm converts them to vertex reaches and stores vertex reaches to save space. The main advantage of computing arc reaches is that is allows for more efficient shortcutting: a high-reach vertex can be bypassed as soon as enough low-reach arcs incident to it are eliminated.

The computation of arc reaches proceeds in iterations, or *levels.* At level i the algorithm tries to bound all arc reaches below some threshold ϵ_i, which grows exponentially with i. Level i starts by removing from the graph all arcs whose reach was bounded in the previous level. It then shortcuts vertices that have exactly two neighbors. Finally, it grows partial shortest path trees from all remaining vertices and uses them to find upper bounds on reaches of arcs with reach less than ϵ_i.

The preprocessing algorithm described in [16] shortcuts a vertex v with neighbors u and w by replacing the arcs (u, v) and (v, w) (if both exist) by an arc (u, w), and the arcs (w, v) and (v, u) (if both exist) by (w, u). The new arcs have length equal to the sum of the lengths of the arcs they replace, thus keeping the original distances unchanged. When breaking ties, however, we consider the new arc to be slightly shorter. As a result, v will no longer be on any shortest path that contains both u and w, which will in effect reduce v's reach.

We construct partial shortest path trees to bound the reach of the arcs that remain in the graph. For these upper bounds to be valid in the original graph, we must take into consideration the arcs that have already been deleted (either in previous iterations or when introducing shortcuts in the current iteration). We do that by associating *penalties* to each vertex v. The *in-penalty* of v is the maximum reach upper bound of an incoming arc that has already been eliminated. The *out-penalty* of v is similar, but considering outgoing arcs from v. When evaluating the partial shortest paths trees, we use the penalties as upper bounds on the lengths of subpaths that no longer belong to the current graph.

The procedure outlined above will proceed until all arc reaches have been bounded. This will happen once the threshold ϵ_i is large enough. At this point, arc reaches are converted to vertex reaches.

This is followed by a *refinement phase*: in order to obtain more accurate reaches, we compute exact reaches on the subgraph induced by the δ vertices with highest reach, where δ is a user-defined parameter (set to $\lceil 10\sqrt{n} \rceil$ in [16]). The remaining vertices are considered in the computation, but only implicitly, as penalties.

Section 6.3, 6.4 and 6.5 will present a more detailed description of various elements of the preprocessing algorithms, including some improvements.

4 A^* Search and the ALT Algorithm

Suppose we need to find shortest paths on a graph G with distance function ℓ. A *potential function* maps vertices to reals. Given a potential function π, the *reduced cost* of an arc is defined as $\ell_\pi(v,w) = \ell(v,w) - \pi(v) + \pi(w)$. Suppose we replace ℓ by ℓ_π. Then for any two vertices x and y, the length of every x-y path (including the shortest) changes by the same amount, $\pi(y) - \pi(x)$. Thus the problem of finding shortest paths in G is equivalent to the problem of finding shortest paths in the transformed graph.

Now suppose we are interested in finding the shortest path from s to t. Let π_f be a (perhaps domain-specific) potential function such that $\pi_f(v)$ gives an estimate on the distance from v to t. In the context of this paper, A^* *search* [7, 20] is an algorithm that works like Dijkstra's algorithm, except that at each step it selects a labeled vertex v with the smallest *key*, defined as $k_f(v) = d_f(v) + \pi_f(v)$, to scan next. It is easy to see that A^* search is equivalent to Dijkstra's algorithm on the graph with length function ℓ_{π_f}. If π_f is such that ℓ_π is nonnegative for all arcs (i.e., if π_f is *feasible*), the algorithm will find the correct shortest paths. We refer to the class of A^* search algorithms that use a feasible function π_f with $\pi_f(t) = 0$ as *lower-bounding algorithms*.

We combine A^* search and bidirectional search as follows. Let π_f be the potential function used in the forward search and let π_r be the one used in the reverse search. Since the latter works in the reverse graph, each original arc (v,w) appears as (w,v), and its reduced cost w.r.t. π_r is $\ell_{\pi_r}(w,v) = \ell(v,w) - \pi_r(w) + \pi_r(v)$, where $\ell(v,w)$ is in the original graph. We say that π_f and π_r are *consistent* if, for all arcs (v,w), $\ell_{\pi_f}(v,w)$ in the original graph is equal to $\ell_{\pi_r}(w,v)$ in the reverse graph. This is equivalent to $\pi_f + \pi_r =$ constant.

If π_f and π_r are not consistent, the forward and reverse searches use different length functions. When the searches meet, we have no guarantee that the shortest path has been found. Assume π_f and π_r give lower bounds to the sink and from the source, respectively. We use the *average function* suggested by Ikeda et al. [21], defined as $p_f(v) = \frac{\pi_f(v) - \pi_r(v)}{2}$ for the forward computation and $p_r(v) = \frac{\pi_r(v) - \pi_f(v)}{2} = -p_f(v)$ for the reverse one. Although p_f and p_r usually do not give lower bounds as good as the original ones, they are feasible and consistent.

The ALT algorithm is an A^*-based algorithm that uses landmarks and triangle inequality to compute feasible lower bounds. We select a small subset of vertices as *landmarks* and, for each vertex in the graph, precompute distances to and from every landmark. Consider a landmark L: if $d(\cdot)$ is the distance *to* L, then, by the triangle inequality, $d(v) - d(w) \leq \text{dist}(v,w)$; if $d(\cdot)$ is the distance *from* L, $d(w) - d(v) \leq \text{dist}(v,w)$. To get the tightest lower bound, one can take the maximum of these bounds, over all landmarks (for efficiency reasons, a query uses only a subset of the available landmarks). Intuitively, the best lower bounds on $\text{dist}(v,w)$ are given by landmarks that appear "before" v or "after" w. The version of ALT algorithm that we use balances the work of the forward search and the reverse search (see Section 2). This version had better performance than other variants.

For more details on landmark selection algorithms and on using a subset of active landmarks for a given query, see [18].

5 Combining Reach and Landmarks

Reach-based pruning can be easily combined with A^* search. The general approach is to run A^* search and prune vertices (or arcs) based on reach conditions. Specifically, when A^* is about to scan a vertex v we extract the length of the shortest path from the source to v from the key of v (recall that for the unidirectional algorithm, $k_f(v) = d_f(v) + p_f(v)$). Furthermore, $\pi_f(v)$ is a lower bound on the distance from v to the destination. If the reach of v is smaller than both $d_f(v)$ and $\pi_f(v)$, we prune the search at v.

The reason why reach-based pruning works is that, although A^* search uses transformed lengths, the shortest paths remain invariant. This applies to bidirectional search as well. In this case, we use $d_f(v)$ and $\pi_f(v)$ to prune in the forward direction, and $d_r(v)$ and $\pi_r(v)$ to prune in the reverse direction. Using pruning by reach does not affect the stopping condition of the algorithm. We still apply the usual condition for A^* search, which is similar to that of the standard bidirectional Dijkstra, but with respect to reduced costs (see [18]). We call our implementation of the bidirectional A^* search algorithm with landmarks and reach-based pruning REAL. As we did for ALT, our implementation of REAL balances the work of the forward search and the reverse search. Implicit bounds cannot be used with A^* search to prune by reach, but landmark bounds can.

Note that REAL has two preprocessing algorithms: the one used by RE (which computes shortcuts and reaches) and the one used by ALT (which chooses landmarks and computes distances between them and all vertices). These two procedures can be independent from each other: since shortcuts do not change distances, landmarks can be generated regardless of what shortcuts are added. Furthermore, the query is still independent of the preprocessing algorithm: the query only takes as input the graph with shortcuts, the reach values, and the distances to and from landmarks. As we will see in the next section, however, it might be useful to take reaches into account when generating landmarks.

6 Algorithmic Improvements

In this section we give a summary of the most significant changes that improved preprocessing or query performance of our algorithms.

6.1 Improving Locality

When reaches are available, a typical point-to-point query spends most of its time scanning high-reach vertices. Except at the very beginning of the search, low-reach vertices are pruned by reach. This suggests an obvious optimization: during preprocessing, reorder the vertices such that high-reach vertices are close together in memory to improve cache locality.

The simplest way to achieve this would be to sort vertices in non-increasing order of reach. This, however, will destroy the locality of the input: in many applications (including road networks), the original vertex order already has high locality.

Instead, we adopt the following approach to order the vertices. We fix a constant $\alpha > 1$ (we use $\alpha = 2$) and partition the vertices into two sets: the first contains the $1/\alpha$ fraction of vertices with highest reach, and the other contains the remaining vertices. We keep the original relative ordering in each part, then recursively process the first part. Besides improving locality, this reordering also facilitates other optimizations described below.

6.2 Reach-Aware Landmarks

Here we investigate the interaction of landmarks and reaches. We can reduce the memory requirements of the algorithm by storing landmark distances only for high-reach vertices. As we shall see, however, this results in some degradation of query performance. If we add more landmarks, we get a wide range of trade-offs between query performance and memory requirement. We call the resulting method, a variant of REAL, the *partial landmark algorithm*.

Queries for the partial landmark algorithm work as follows. Let R be the reach threshold: we store landmark distances for all vertices with high reach, i.e., with reach at least R. We start a query by running the bidirectional Dijkstra algorithm, with reach pruning but without A^* search, until either the algorithm terminates or both balls searched have radius R. In the latter case, we know that, from this point on, we need to examine only vertices with reach R or more. We switch to A^* search by removing labeled vertices from the heaps and re-inserting them using new keys that incorporate lower bound values.

Recall that, for every vertex v it visits, A^* search may need lower bounds on the distance from v to t (in the forward search) or from s to v (in the reverse search). They are computed with the triangle inequality, which requires distances between these vertices (v, s, and t) and the landmarks. These are guaranteed to be available for v, which has high reach, but not for s or t, which are arbitrary vertices. We need to specify how to compute a lower bound on the distance between a low-reach vertex (s or t) and a high-reach one (v).

Suppose s has low reach (t is treated symmetrically). Let s', the *proxy* for s, be the high-reach vertex that is closest to s.[1] One can compute proxies during preprocessing and store them, or compute them during the initialization phase of the query algorithm; we choose the latter approach. Two executions of a multiple-source version of Dijkstra's algorithm (one in the forward graph and one in the reverse graph) suffice to compute both the proxies and the appropriate distances.

As already mentioned, when processing a high-reach vertex v, the A^* search needs lower bounds on $\text{dist}(s, v)$ and $\text{dist}(v, t)$. Given a landmark L, we can obtain these bounds using either distances *from* L or distances *to* L. With the help of proxies, these bounds can be easily computed.

A lower bound on $\text{dist}(s, v)$ using distances *to* L is given by

$$\text{dist}(s, v) \geq \text{dist}(s', L) - \text{dist}(v, L) - \text{dist}(s', s). \tag{1}$$

Using distances *from* L, the lower bound can be computed as follows:

$$\text{dist}(s, v) \geq \text{dist}(L, v) - \text{dist}(L, s') - \text{dist}(s', s). \tag{2}$$

Lower bounds on distances from v to the target t can be computed similarly. Using distances *from* landmarks, the following relation applies:

$$\text{dist}(v, t) \geq \text{dist}(L, t') - \text{dist}(L, v) - \text{dist}(t, t'). \tag{3}$$

With distances *to* landmarks, the appropriate expression is

$$\text{dist}(v, t) \geq \text{dist}(v, L) - \text{dist}(t', L) - \text{dist}(t, t'). \tag{4}$$

[1]Each vertex s actually has two proxies: the high-reach vertex s' that minimizes $\text{dist}(s', s)$ and the high-reach vertex s'' that minimizes $\text{dist}(s, s'')$. We will assume they are the same to simplify the discussion, but they need not be.

Note that distances between L and v, s' and t' are computed during the preprocessing stage, since all three vertices have high reach. As already mentioned, the distances between every vertex and its proxy (or, more precisely, proxies) are computed in the initialization phase of the query algorithm.

The quality of the lower bounds obtained by the partial landmark algorithm depends not only on the number of landmarks available, but also on the value of R. In general, the higher the reach threshold, the farther the proxy s' will be from s (and t' from t), thus decreasing the accuracy of the lower bounds. If all landmark distances are available, R will be zero, and the algorithm will behave exactly as the standard REAL method. By decreasing the number of distances per landmark (representing distances only to higher-reach vertices), R will increase; a trade-off between memory usage and query efficiency is thus established.

It turns out, however, that we can often improve both memory usage and query times. Starting from the base algorithm, we can increase the number of landmarks while decreasing the number of distances per landmark so that the total memory usage is lower. On large road networks, the availability of more landmarks makes up for the fact that only a fraction of all distances is available, and query times in fact improve.

In our experiments, we generate up to 64 landmarks, as opposed to only 16 in our previous study [16]. We use the *avoid* algorithm to select landmarks. Among the landmark selection algorithms studied in [18], *avoid* is the second best in terms of solution quality. It is only surpassed by *maxcover*, which is essentially *avoid* followed by local search and is about five times slower. We have discovered a minor issue with our previous implementation of *avoid* that caused it to obtain slightly worse landmarks than it should. The fixed version of *avoid* provides slightly better landmarks, and is almost as good as *maxcover*.

The *avoid* method works by adding one landmark at a time. In each iteration, it tries to pick a landmark in a region that is still not well-covered by existing landmarks. To do so, it first builds a complete shortest path tree T_r rooted at a vertex r picked in a randomized fashion. It then assigns a weight to each vertex v, which depends on the quality of the lower bound on the distance from r to v (based on previously selected landmarks). Vertices with bad lower bounds will have higher weights. The algorithm determines the vertex p that, among all vertices that have no landmark as a descendent, maximizes the sum of the weights of its descendents in T_r. A leaf of the subtree of T_r rooted at p is then selected as the new landmark.

Computing the lower bound on the distance from r to a particular vertex takes time proportional to the number of landmarks already selected, which makes *avoid* quadratic in the number k of landmarks. This is not a major issue when only 16 landmarks are selected, but the algorithm does get measurably slower with 64 or more landmarks. We propose a simple modification to the algorithm to make it linear in the number of landmarks.

When processing the shortest path tree rooted at r, we still define the weight of v to be the difference between the distance from r to v and the current lower bound on this distance, but only if v belongs to a set of n/k *relevant vertices*. For all other vertices, we define the weight as zero. If no reach information is available, the set of relevant vertices is picked uniformly at random. If reaches are available, the set is defined as the n/k vertices with smallest labels. Because vertices are approximately sorted in decreasing order of reach (as seen in the previous subsection), this means that only vertices with high reach are taken into account. These changes made the algorithm substantially faster, with no discernible decrease in solution quality.

6.3 Adding Shortcuts

Our previous implementation of the preprocessing procedure only shortcuts lines, i.e., connected subgraphs induced by vertices with exactly two neighbors. Sanders and Schultes [34] suggested shortcutting not only degree-two vertices, but other vertices of small degree as well. This approach is more general and works better for both preprocessing and queries.

We can shortcut a vertex v as follows. First, we examine all pairs of incoming/outgoing arcs $((u,v),(v,w))$ with $u \neq w$. For each pair, if the arc (u,w) is not in the graph, we add an arc (u,w) of length $\ell(u,v)+\ell(v,w)$. Otherwise, we set $\ell(u,w) \leftarrow \min\{\ell(u,w), \ell(u,v)+\ell(v,w)\}$. Finally, we delete v and all arcs adjacent to it.

In principle, any vertex in the graph could be subject to this procedure. However, bypassing high-degree vertices would significantly increase the number of arcs in the graph. To avoid an excessive expansion, we consider the ratio c_v between the number of new arcs added and the number of arcs deleted by the procedure above. A vertex is deemed *bypassable* if $c_v \leq c$, where c is a user-defined parameter.[2] Higher values of c will cause the graph to shrink faster during preprocessing, but may increase the final number of arcs substantially. We used values of c from 0.5 to 2.0 in our experiments. For road networks, we used 0.5 on the first level, 1.0 on the second, and 1.5 on the remaining levels. This prevents the algorithm from adding too many shortcuts at the beginning of the preprocessing algorithm (when the graph is larger but shrinks faster) and ensures that the graph will shrink fast enough as the algorithm progresses.

We impose some additional constraints on a vertex v to deem it bypassable (besides having a low value of c_v). First, we require both its in-degree and its out-degree to be bounded by a constant (5 in our experiments). This guarantees that the total number of arcs added by the algorithm will be linear in n. We also consider two additional measures (besides c_v) related to v: the length of the longest shortcut arc introduced when v is bypassed, and the largest reach of an arc adjacent to v (which will be removed). The maximum between these two values is the *cost* of v, and it must be bounded by $\epsilon_i/2$ during iteration i for the vertex to be considered bypassable (recall that ϵ_i is the threshold for bounding reaches at that iteration). As explained in [16], long arcs and large penalties can decrease the quality of the reach upper bounds provided by the preprocessing algorithm. Imposing these additional bypassability criteria prevent such long arcs and large penalties from appearing too soon.

On any given graph, many vertices may be bypassable. When a vertex is bypassed, the fact that we remove existing arcs and add new ones may affect the bypassability of its neighbors. Therefore, the order in which the vertices are processed matters. Vertices with low expansion (c_v) and low cost are preferred, since they are the least likely to affect the bypassability of their neighbors. When deciding which vertex v to bypass next, we take the vertex that minimizes the product of these two measures (expansion and cost). When a vertex is bypassed, its neighbors must have their priorities updated. We use a heap to efficiently determine which vertex to bypass next.

6.4 Growing Partial Trees

This section reviews the basic aspects of the main routine of our preprocessing algorithm, originally described in [16]. We then mention a few simple modifications in the way penalties are handled that make the procedure roughly twice as fast in practice. For simplicity, we describe the algorithm as if it computed vertex reaches; the procedure for computing arc reaches essentially the same.

[2]We follow the notation proposed by Sanders and Schultes.

Given a graph $G = (V, A)$ and a constant ϵ (the *threshold*), our goal is to find valid reach upper bounds for the vertices in V whose actual reach is smaller than ϵ. For the remaining vertices, the upper bound is ∞. While the algorithm is allowed to report false negatives (i.e., it may find an upper bound of ∞ for vertices whose actual reach is less than ϵ), it must never report a false positive.

Fix a vertex v. To prove that $r(v) < \epsilon$, we must consider all shortest paths through v. Fortunately, we do not have to evalute all such paths explicitly: we only need to consider *minimal paths*. Consider a shortest path $P_{st} = (s, s', \ldots, v, \ldots, t', t)$ between s and t, and assume that v has reach at least ϵ with respect to this path. Path P_{st} is *ϵ-minimal* with respect to v if and only if the reaches of v with respect to $P_{s't}$ and $P_{st'}$ are both smaller than ϵ.

The algorithm works by growing a *partial tree* T_r from each vertex $r \in V$. It runs Dijkstra's algorithm from r, but stops it as soon as it can prove that all minimal paths starting at r are part of the tree. In order to determine when to stop growing the tree, we need the notion of *inner vertices*. Let $v \neq r$ be a vertex in this tree, and let x be the first vertex (besides r) on the path from r to v. We say that v is an *inner vertex* if either (1) $d(r, v) \leq \epsilon$ or (2) $d(r, v) > \epsilon$ and $d(x, v) < \epsilon$. The root r is defined to be an inner vertex as well. When v is not an inner vertex, no path P_{rw} starting at r will be ϵ-minimal with respect to v: if v's reach is greater than ϵ with respect to P_{rw}, it will also be greater than ϵ with respect to P_{xw}.

The partial tree T_r must be large enough to include all inner vertices—these will be the vertices whose reaches we will try to bound. To get accurate bounds on reaches, we must make sure that every inner vertex v has one of two properties: (1) v has no labeled (unscanned) descendents; or (2) v has at least one scanned descendent whose distance to v is ϵ or greater. We actually use a relaxed version of the second condition: we stop growing the tree when every labeled vertex is within distance greater than ϵ from the closest inner vertex.

Once the tree is built, processing it is straightforward. For each inner vertex, we know its *depth*, i.e., the distance from the root. In $O(|T_r|)$ time, one can also compute the *height* of every inner vertex v, defined as the distance from v to its farthest descendent. Clearly, the reach of v with respect to the partial tree is the minimum between its depth and its height. The reach of v with respect to the entire graph is the maximum over all such reaches. If this maximum is at least ϵ, we declare the reach to be ∞.

Dealing with penalties. As described, the algorithm assumes that partial trees will be grown from every vertex in the graph. We would like, however, to run the partial-trees routine even after some of the vertices have been eliminated (either because they were bypassed, or because their reach was bounded in a previous iteration). Eliminated vertices must be taken into account, since they may belong to paths that determine the reach of the remaining vertices. However, growing partial trees from them would be too expensive.

Instead, we use the notion of *penalties* to account for the eliminated vertices. If v is a vertex that remains in the graph, its *in-penalty* is the maximum over the reaches of all arcs (u, v) that have already been eliminated. Similarly, the *out-penalty* of v is the maximum reach of all arcs (v, w) that have already been eliminated. (It is possible to define penalties in terms of vertex reaches, but the definition in terms of arc reaches is simpler.)

Our original algorithm [16] did not take penalties into account when growing partial trees. Penalties were only used to process the partial tree. The (redefined) depth of a vertex v within a tree T_r is the distance from r to v plus the in-penalty of r. Similarly, the height of a vertex is redefined to take out-penalties into account. For that, we consider that each vertex v in T_r is attached to a *pseudo-leaf* v', and that the length of the edge between v and v' is equal to the out-penalty of v. Heights are computed not with respect to T_r, but with respect to the pseudo-tree obtained if the pseudo-leaves are taken into account.

Although taking penalties into account when growing partial trees is not required for correctness, it turns out to be helpful to accelerate the algorithm. We therefore make two simple modifications to the partial-trees algorithm.

First, when deciding whether v is an inner vertex with respect to a root r, we must compute $\mathit{depth}_r(v)$ taking in-penalties into account, i.e., as $\text{dist}(r, v) + \mathit{in\text{-}penalty}(r)$. Vertex v will be considered an inner vertex if either (1) $\mathit{depth}_r(v) \leq \epsilon$ or (2) $\mathit{depth}_r(v) > \epsilon$ and $\mathit{depth}_x(v) < \epsilon$ (recall that x is the second vertex on the path from r to v). Here, $\mathit{depth}_x(v)$ also takes $\mathit{in\text{-}penalty}(x)$ into account. There is one exception to the two criteria above: if $\mathit{depth}_r(v) < \mathit{in\text{-}penalty}(v)$, v will not be considered an inner vertex (because its modified depth will be even higher in the tree rooted at v), and neither will its descendents.

The second modification is in the stopping criterion. We grow the tree until none of the labeled (unscanned) vertices is *relevant*. All inner vertices are considered relevant. In addition, an outer vertex v may also be consider relevant if it satisfies certain constraints. Let u be the inner ancestor of v in the current tree that is closest to v. Let $d(u, v)$ be the distance (in the tree) between u and v. Vertex v will be considered relevant if $d(u, v) + \mathit{out\text{-}penalty}(v) \leq \epsilon$ (note that this is the expression used to compute the modified depth of u).

Note that this stopping criterion is not exact, and therefore may lead to false negatives: the algorithm may find infinite upper bounds for some reaches that are less than ϵ. For a more precise computation, one should stop only after ensuring that every inner vertex u either has no labeled descendent, or has at least one *scanned* descendent v such that $d(u, v) + \mathit{out\text{-}penalty}(v) > \epsilon$. The heuristic stopping criterion makes the algorithm much faster, however, with little effect on the quality of the upper bounds obtained.

Once again, we note that we actually use the algorithms to compute *arc reaches* instead of vertex reaches. The procedure is essentially the same.

6.5 Approximate vs. Exact Reach Computation

As our preprocessing algorithm proceeds, the partial trees become large relative to the graph size, which makes the computation slower. Therefore, when the graph is small enough, computing exact reaches is more cost-effective. The computation produces accurate reaches even faster than the partial-trees algorithm does.

Standard algorithm. The standard algorithm for computing exact reaches processes all n^2 shortest paths by building shortest path trees from each vertex in the graph. The shortest path tree rooted at a vertex r implicitly represents all shortest paths that start at r. The reach of a vertex v restricted to these paths is given by the minimum between its *depth* (the distance from r) and its *height* (the distance to its farthest descendent) in this tree. The reach of v with respect to all paths is the maximum reach v obtains in any of these shortest path trees.

Improved algorithm. We developed an algorithm that has the same worst-case performance as the method above, but can be significantly faster in pratice on road networks. It follows the same basic principle: build a shortest path tree from each vertex in the graph and compute reaches within these trees. Our improvement consists of building parts of these trees *implicitly* by reusing subtrees previously found.

The algorithm works as follows. First, partition the vertices of the graph into k subsets, for a given parameter k (usually around $\sqrt{n}$). The algorithm will work with any such partition, but some are better than others, as we shall see. We call each subset a *region* of the graph. The *frontier* of a region A, denoted by $f(A)$, is the set of vertices $v \in A$ such that there exists at least one arc (v, w) with $w \neq A$. The remainder of the region consists of *inner vertices*.

Given any set $S \subseteq V$, we say that a vertex v is *stable* with respect to S if it has the same parent in all shortest path trees rooted at vertices in S; otherwise, we call it *unstable*. For any region A, the following holds:

Lemma 1 *If $v \in V \setminus A$ is stable with respect to $f(A)$, then v is stable with respect to A.*

Proof. Let $p_r(v)$ denote the parent of v in T_r (the shortest path tree rooted at r) and $p_{f(A)}(v)$ denote the common parent of v in all shortest path trees rooted at $f(A)$. Suppose the lemma is not true, i.e., that there exist a vertex $r \in A$ and a vertex $v \in V \setminus A$ such that $p_r(v) \neq p_{f(A)}(v)$. Consider the path P from r to v in T_r. Because $r \in A$ and $v \notin A$, at least one vertex in P must belong to $f(A)$. Let s be the last such vertex, i.e., the one closest to r. The subpath of P from s to r is itself a shortest path, and therefore it must appear in the shortest path tree rooted at s (we assume all shortest paths are unique). But recall that v has $p_r(v)$ as its parent in this path, and our initial assumption was that its parent on all trees rooted at $f(A)$ was $p_{f(A)}(v) \neq p_r(v)$. We have reached a contradiction. $\square$

A vertex $v \in V$ is considered *tainted* with respect to $f(A)$ if at least one of the following conditions holds: (1) v is unstable with respect to $f(A)$; (2) v has an unstable descendent in at least one of the $|f(A)|$ trees; or (3) $v \in A$. If none of these conditions hold, v is *untainted*. An untainted vertex v will be the root of the exact same subtree in every shortest path tree rooted at $f(A)$. The lemma above also ensures that it would be the root of the same subtree if we grew shortest path trees from the inner vertices of A as well. We know that even without actually growing the trees.

Our algorithm takes advantage of this. In its first stage, it grows full shortest path trees from every vertex in $f(A)$. For these trees, it computes the height and the depth of every vertex, as usual. The second stage of the algorithm grows *partial trees* from every inner vertex of A (i.e., every vertex in $A \setminus f(A)$). These partial trees contain only tainted vertices; no untainted vertex is ever visited. Even so, it is still possible to compute the reach of the tainted vertices as if we had grown the entire tree. The depth can be computed as before. For the height, we need to consider vertices that were not visited.

This is done as follows. Consider a maximal untainted subtree rooted at a vertex w. The height of this tree can be easily precomputed. Because w is untainted, its parent p will be the same (tainted) vertex in every shortest path tree rooted at A. Therefore, w imposes an *implicit penalty* on p equal to w's own height plus the length of the arc (p, w). The *extended penalty* of a tainted vertex is defined as the maximum between its own out-penalty and the implicit penalties associated with its untainted children. Note that the extended penalty needs to be computed only once, after all trees rooted at the frontier are built. When trees are built from inner vertices, the height of each vertex visited (which must be tainted) is computed as usual, using extended penalties instead of out-penalties.

Our description so far allows us to correctly compute the reach of each tainted vertex. We also need to determine the reach of the untainted vertices. Although the height of an untainted vertex v is the same across all trees, the depth varies, and so does the reach. Fortunately, we do not need to know the reach of v within each tree; it suffices to know the *maximum* reach. Since the height is constant, the maximum is realized in the tree that maximizes the depth of v. If, when growing full and partial shortest path trees, we remember the maximum depth of each tainted vertex, we can later compute (in linear time) the maximum depth of the untainted vertices. Since their heights are known, their reaches (with respect to the trees rooted at A) can be easily determined.

Regions. The algorithm above is correct regardless of how the regions are chosen. In particular, if each region has exactly one vertex, we have exactly the standard algorithm. Of course, there are better choices of regions. There are two main goals to achieve: (1) the size of the frontier should be small compared to the size of the entire region and (2) the number of tainted vertices should be minimized. On road networks, these two goals are conflicting. In general, a larger region will have a smaller fraction of its vertices in the frontier. However, they will also increase the probability of an external vertex being tainted. A good compromise is to choose regions with roughly $\sqrt{n}$ vertices.

To create a partition with at least k sets, we pick k vertices at random to be centers and determine their Voronoi regions (we use $k = 2\lceil\sqrt{n}\rceil$). Recall that the Voronoi region associated with a center v is the set of vertices w that are closer to v than to any other center (ties are broken arbitrarily). One can compute the Voronoi diagram of a graph with a multiple-source version of Dijkstra's algorithm. Unreachable vertices are assigned to regions by themselves.

Although the Voronoi diagram is a simple way of defining the regions, it is certainly not the best conceivable partition. A topic for future research is to determine regions with relatively smaller frontiers.

Using the algorithm. As already mentioned, having a faster exact algorithm allows us to switch to the exact algorithm earlier when the graph is larger. Specifically, we switch when either of two conditions is satisfied: (1) the number of remaining vertices is less than $\lceil 5\sqrt{n}/2\rceil$; or (2) after a level in which average size of partial tree was greater than one eighth of the total number of remaining vertices.

Of course, the exact algorithm finds reaches for all arcs that remain in the graph. It is not in our interest to stop the algorithm at this point, however: we would like to add more shortcuts in order to improve query performance. Therefore, we only take into account the bottom third of the computed reaches. The corresponding arcs are eliminated from the graph; the other two-thirds remain, and we ignore their reaches. The next iteration will then add shortcuts which would not be added otherwise. These shortcuts reduce reaches of high-reach vertices and improve query performance.

The new version of the exact algorithm is also used during the *refinement step* of the preprocessing algorithm. After all reaches are computed, we build the graph induced by the $\lceil 5\sqrt{n}\rceil$ vertices with highest reach and computing exact reaches on it. The remaining vertices are accounted for as penalties. This significantly improves the reach bounds for these vertices, which are the ones visited most often during the queries.

7 Experimental Results

We implemented our algorithms in C++ and compiled them with Microsoft Visual C++ 2005. All tests were performed on a dual-core, 2.4 GHz AMD Opteron machine running Microsoft Windows

Server 2003 with 16 GB of RAM, 32 KB instruction and 32 KB data level 1 cache per processor, and 2 MB of level 2 cache. Our code is single-threaded and runs on a single core.

The main goals of our experimental analysis are:

- Measure algorithm performance on the 9th DIMACS Challenge road networks.
- Determine how much various improvements contribute to performance gain.
- Study the preprocessing/query as well as time/space trade-offs.
- Determine if our heuristics work on some P2P problems other than road networks.

We ran ALT, RE and REAL-(i, j) algorithms, where the REAL-(i, j) algorithm uses i landmarks and maintains landmark data for n/j highest-reach vertices. We call parameter j the *sparsity* of the landmarks. For instance, REAL-(64,16) maintains 64 landmarks, but with distances only to the $n/16$ vertices with highest reach; REAL-(16,1) uses 16 landmarks, each with distances to all vertices in the graph. On grid graphs, we also ran our own implementation of the bidirectional version of Dijkstra's algorithm, denoted by BD. For machine calibration purposes, we ran the DIMACS Challenge implementation of the P2P version of Dijkstra's algorithm, denoted by D, on the largest road networks.

For Europe and USA graphs with time-based lengths, additional data from previous works is available. We give the data for our previous implementations, RE-OLD and REAL-OLD from [16] (run on the same machine). REAL-OLD uses 16 landmarks selected with the *maxcover* method. In addition, we give the data for the highway hierarchy-based algorithm of Sanders and Schultes [34], where a slightly slower 2.0 GHz AMD Opteron machine was used. (The running times should be roughly comparable.) There are two versions their algorithm: HH-mem, entirely based on highway hierarchies, and HH, which replaces high levels of the hierarchy by a table with distances between all pairs of vertices in the corresponding graph.

For USA and Europe, we give data for both random and local queries. For the USA subgraphs, we give only random query data, as local query performance is mostly independent of the graph size.

We also tested grid graphs with 2 and 3 dimensions. These are square- and cube-shaped grids, with each vertex connected by two directed arcs (one in each direction) to each neighbor. Lengths are chosen uniformly at random.

We also tried grids of higher dimensions (including hypercubes) and random graphs, but on those reach-based techniques offer no significant improvement compared to the bidirectional version Dijkstra's algorithm. Detailed results are therefore omitted. We give data on the 3-dimensional grids (even though our algorithm is not substantially faster that bidirectional Dijkstra) as it is more likely that progress will be made for these graphs in the near future.

7.1 Road Networks

The USA (Tiger) [41] and Europe [28] graphs are those from the DIMACS Challenge [4] data set. We worked with the largest strongly connected components of each graph. The USA is symmetric and has 23 947 347 vertices and 58 333 444 arcs; Europe is directed, with 18 010 173 vertices and 42 560 279 arcs. (These sizes refer to the largest strongly connected component.) To test performance on smaller graphs, we used the subgraphs of USA described in Table 1. We used both transit times and distances as length functions.

Table 1: USA Road Networks derived from the TIGER/Line collection.

NAME	DESCRIPTION	VERTICES	ARCS	LATITUDE (N)	LONGITUDE (W)
USA	—	23 947 347	58 333 344	—	—
CTR	Central USA	14 081 816	34 292 496	[25.0; 50.0]	[79.0; 100.0]
W	Western USA	6 262 104	15 248 146	[27.0; 50.0]	[100.0; 130.0]
E	Eastern USA	3 598 623	8 778 114	[24.0; 50.0]	[$-\infty$; 79.0]
LKS	Great Lakes	2 758 119	6 885 658	[41.0; 50.0]	[74.0; 93.0]
CAL	California and Nevada	1 890 815	4 657 742	[32.5; 42.0]	[114.0; 125.0]
NE	Northeast USA	1 524 453	3 897 636	[39.5, 43.0]	[$-\infty$; 76.0]
NW	Northwest USA	1 207 945	2 840 208	[42.0; 50.0]	[116.0; 126.0]
FLA	Florida	1 070 376	2 712 798	[24.0; 31.0]	[79; 87.5]
COL	Colorado	435 666	1 057 066	[37.0; 41.0]	[102.0; 109.0]
BAY	Bay Area	321 270	800 172	[37.0; 39.0]	[121; 123]
NY	New York City	264 346	733 846	[40.3; 41.3]	[73.5; 74.5]

Random queries. Table 2 gives data for random queries on the two largest road graphs available, Europe and USA, each with both length functions.

Our algorithms were run on 1 000 queries picked uniformly at random. We report the average query time (in milliseconds), the average number of scanned vertices and, when available, the maximum number of scanned vertices (over the queries in the set). Also shown are the total preprocessing time (in minutes) and the total space in disk (in megabytes) required by the preprocessed data. For D, this is only the graph itself; for ALT, this includes the graph and landmark data; for RE, it includes the graph with shortcuts (which is larger) and an array of vertex reaches; finally, the data for REAL includes the graph with shortcuts, the array of reaches, and landmark data.

Note that for ALT, RE and REAL, query performance is worse for the distance-based metric, but not drastically so. Preprocessing performance is similar, and depends both on the base graph and on the length function. Comparing REAL-(16,1) to REAL-(64,16), we see they have almost identical query performance with transit times, and that REAL-(64,16) is slightly better with travel distances. Given that REAL-(64,16) requires about half as much disk space, it has the edge for these queries. On graphs with the distance metric, REAL-(64,16) wins both in time and in space. However, preprocessing for it takes roughly twice as long. RE is less robust than REAL.

We can compare the new methods to previous ones when travel times are used as the length function. RE and REAL substantially improve on their old counterparts, especially in terms of preprocessing time. RE has worse performance than HH-mem, but not by much. For queries, REAL-(64,16) performs similarly to HH: REAL-(64,16) uses a little less space, while HH is slightly faster. Preprocessing for HH, however, is significantly faster. With no data available, we cannot comment on the performance of HH and HH-mem on graphs with travel distances.

Graph size dependence. Tables 3 and 4 show how the performance of our algorithms scales with graph size. Although times tend to increase as the graph grows, they are not strictly monotone: the graph structure and the way the reach computation switches to the exact variant affect query performance. However, it is clear that reach-based algorithms have better asymptotic performance than ALT. As the graph size increases by two orders of magnitude, so does the time for ALT queries. Running times of RE and REAL change by a factor of six or less.

Regarding preprocessing, with a fixed number of landmarks ALT is roughly linear in the graph

Table 2: Data for random queries on Europe and USA graphs.

GRAPH	METHOD	PREP. TIME (min)	DISK SPACE (MB)	QUERY AVG SC.	QUERY MAX SC.	QUERY TIME (ms)
EUROPE (times)	ALT	13.4	1 631	82 348	993 015	158.33
	RE	82.8	626	4 563	8 866	3.48
	REAL-(16,1)	96.2	1 864	741	3 518	1.17
	REAL-(64,16)	143.9	935	683	2 817	1.14
	RE-OLD	1558	570	16 122	34 118	13.5
	REAL-OLD	1625	1793	1 867	8 499	2.8
	HH	15	1 570	884	—	0.8
	HH-mem	55	692	1 976	—	1.4
	D	—	393	8 984 289	—	4 365.81
USA (times)	ALT	18.5	2 608	187 968	2 183 718	399.21
	RE	48.6	890	2 264	4 667	1.86
	REAL-(16,1)	67.2	2 962	581	2 732	1.03
	REAL-(64,16)	142.5	1 408	543	2 509	1.08
	RE-OLD	366	830	3 851	8 722	4.50
	REAL-OLD	459	2 392	891	3 667	1.84
	HH	18	1 686	1 076	—	0.88
	HH-mem	65	919	2 217	—	1.60
	D	—	536	11 808 864	—	5 440.49
EUROPE (distances)	ALT	10.5	1 656	240 750	3 306 755	417.83
	RE	50.3	664	7 007	12 940	5.81
	REAL-(16,1)	60.8	1 926	847	3 996	1.47
	REAL-(64,16)	98.0	979	575	2 876	1.17
	D	—	393	8 991 955	—	2 934.24
USA (distances)	ALT	16.2	2463	276 195	2 910 133	533.53
	RE	73.8	928	6 866	13 589	6.06
	REAL-(16,1)	90.0	2 854	872	5 563	1.83
	REAL-(64,16)	153.2	1 410	633	3 312	1.52
	D	—	536	11 782 104	—	4 576.02

size. For 16 landmarks, ALT preprocessing is faster than RE, and the ratio of the two remains roughly constant as the graph size increases. With 64 landmarks, the times for landmark selection and reach computation are roughly the same: preprocessing takes about twice as long for REAL-(64,16) than for RE.

Local queries. Up to this point, we have reported data only on random queries. A more realistic assumption for road networks is that most queries will be more local. To define the notion of local queries formally, we use the concept of *Dijkstra rank*. Suppose we run Dijkstra's algorithm from s, and let v be the k-th vertex it scans. Then the Dijkstra rank of v with respect to s is $\lfloor \log_2 k \rfloor$. To generate a local query with rank r, we pick s uniformly at random from V, and pick t uniformly at random from positions $[2^r, 2^{r+1})$ in the scanning order. Note that our definition of Dijkstra rank differs slightly from the one proposed by Sanders and Schultes [33], but it has a similar purpose.

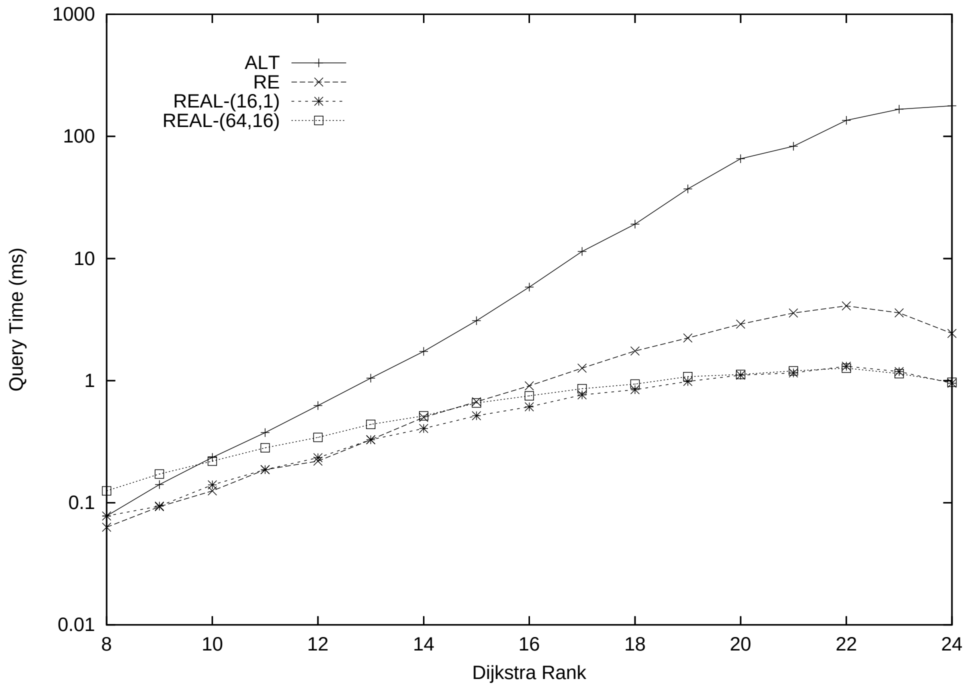
For each of the large graphs (Europe and USA, with both length functions), we generated 1 000 random queries for each Dijkstra rank between 2^8 and 2^{24}. The results are reported in

Table 3: Data for USA graphs with travel times.

		PREP. TIME	DISK SPACE	QUERY		
GRAPH	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
NY	ALT	0.1	26	2735	23739	2.16
	RE	0.9	10	1075	2845	0.67
	REAL-(16,1)	1.0	29	227	1171	0.25
	REAL-(64,16)	1.4	15	426	1253	0.39
BAY	ALT	0.2	31	3251	31953	2.28
	RE	0.5	11	751	2037	0.41
	REAL-(16,1)	0.6	35	184	877	0.22
	REAL-(64,16)	1.1	17	274	1117	0.28
COL	ALT	0.2	46	6533	70857	4.95
	RE	0.5	16	738	2404	0.38
	REAL-(16,1)	0.8	52	172	807	0.23
	REAL-(64,16)	1.4	25	227	1162	0.22
FLA	ALT	0.5	108	8195	126608	7.03
	RE	1.9	37	766	1576	0.48
	REAL-(16,1)	2.4	121	199	1037	0.28
	REAL-(64,16)	4.1	58	226	920	0.28
NW	ALT	0.6	127	13106	166870	13.00
	RE	1.8	42	1051	3609	0.62
	REAL-(16,1)	2.4	143	211	1095	0.30
	REAL-(64,16)	4.7	67	223	1277	0.31
NE	ALT	0.8	155	12161	148647	13.83
	RE	3.7	56	1455	3561	0.98
	REAL-(16,1)	4.5	175	300	1596	0.39
	REAL-(64,16)	7.6	86	328	1267	0.44
CAL	ALT	1.0	201	22418	227812	29.08
	RE	4.4	68	1109	3558	0.77
	REAL-(16,1)	5.4	226	302	1591	0.45
	REAL-(64,16)	9.0	107	362	2120	0.41
LKS	ALT	1.6	297	21616	218084	26.98
	RE	6.5	101	1522	3893	1.12
	REAL-(16,1)	8.1	335	328	1708	0.52
	REAL-(64,16)	13.9	160	357	1540	0.55
E	ALT	2.2	382	26712	516585	36.30
	RE	8.6	126	1410	3434	1.05
	REAL-(16,1)	10.7	427	321	1565	0.52
	REAL-(64,16)	18.6	202	331	1532	0.56
W	ALT	3.9	695	75494	888950	132.89
	RE	13.4	226	1636	4154	1.14
	REAL-(16,1)	17.2	781	409	2026	0.64
	REAL-(64,16)	33.5	365	394	1710	0.69
CTR	ALT	14.1	1640	119303	1335378	332.89
	RE	40.1	522	2129	5003	2.20
	REAL-(16,1)	54.2	1846	532	2931	1.33
	REAL-(64,16)	107.6	853	447	2154	1.27
USA	ALT	18.5	2608	187968	2183718	399.21
	RE	48.6	890	2264	4667	1.86
	REAL-(16,1)	67.2	2962	581	2732	1.03
	REAL-(64,16)	142.5	1408	543	2509	1.08

Table 4: Data for USA graphs with travel distances.

		PREP. TIME	DISK SPACE	QUERY		
GRAPH	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
NY	ALT	0.1	25	3083	35210	2.52
	RE	1.1	11	1638	3492	1.00
	REAL-(16,1)	1.2	29	225	1403	0.33
	REAL-(64,16)	1.6	15	533	1714	0.50
BAY	ALT	0.1	30	4875	80310	4.08
	RE	0.5	12	952	2205	0.58
	REAL-(16,1)	0.7	35	192	932	0.28
	REAL-(64,16)	1.1	18	282	1149	0.31
COL	ALT	0.2	43	5774	78677	4.88
	RE	0.7	16	1066	3380	0.66
	REAL-(16,1)	0.9	49	179	1186	0.23
	REAL-(64,16)	1.5	24	259	1431	0.31
FLA	ALT	0.5	103	12394	153621	12.38
	RE	2.3	38	1073	2654	0.73
	REAL-(16,1)	2.8	117	243	1196	0.38
	REAL-(64,16)	4.4	58	279	1611	0.36
NW	ALT	0.6	120	13994	131191	12.91
	RE	2.1	43	1285	3966	0.89
	REAL-(16,1)	2.6	136	216	1112	0.36
	REAL-(64,16)	4.7	66	250	1531	0.34
NE	ALT	0.8	147	15533	214572	18.27
	RE	4.8	58	2823	6294	2.03
	REAL-(16,1)	5.6	170	337	2214	0.59
	REAL-(64,16)	8.4	86	407	1952	0.59
CAL	ALT	1.0	189	27524	315506	36.11
	RE	4.5	70	1767	5302	1.23
	REAL-(16,1)	5.5	217	337	2060	0.52
	REAL-(64,16)	8.8	107	396	1904	0.55
LKS	ALT	1.5	281	41832	622476	60.44
	RE	9.5	106	3861	9055	3.14
	REAL-(16,1)	11.0	324	473	2774	0.81
	REAL-(64,16)	16.3	160	458	2668	0.75
E	ALT	1.9	360	43539	674704	67.28
	RE	10.1	132	3202	7117	2.64
	REAL-(16,1)	12.1	412	397	2202	0.81
	REAL-(64,16)	19.3	202	401	1985	0.70
W	ALT	3.6	654	75682	669930	109.23
	RE	14.4	234	2580	6316	1.94
	REAL-(16,1)	18.0	747	453	3537	0.78
	REAL-(64,16)	31.4	362	408	2163	0.77
CTR	ALT	12.7	1567	154980	1859858	369.74
	RE	62.9	547	6539	12619	6.86
	REAL-(16,1)	75.5	1798	735	4270	2.22
	REAL-(64,16)	120.1	859	519	2773	1.69
USA	ALT	16.2	2463	276195	2910133	533.53
	RE	73.8	928	6866	13589	6.06
	REAL-(16,1)	90.0	2854	872	5563	1.83
	REAL-(64,16)	153.2	1410	633	3312	1.52



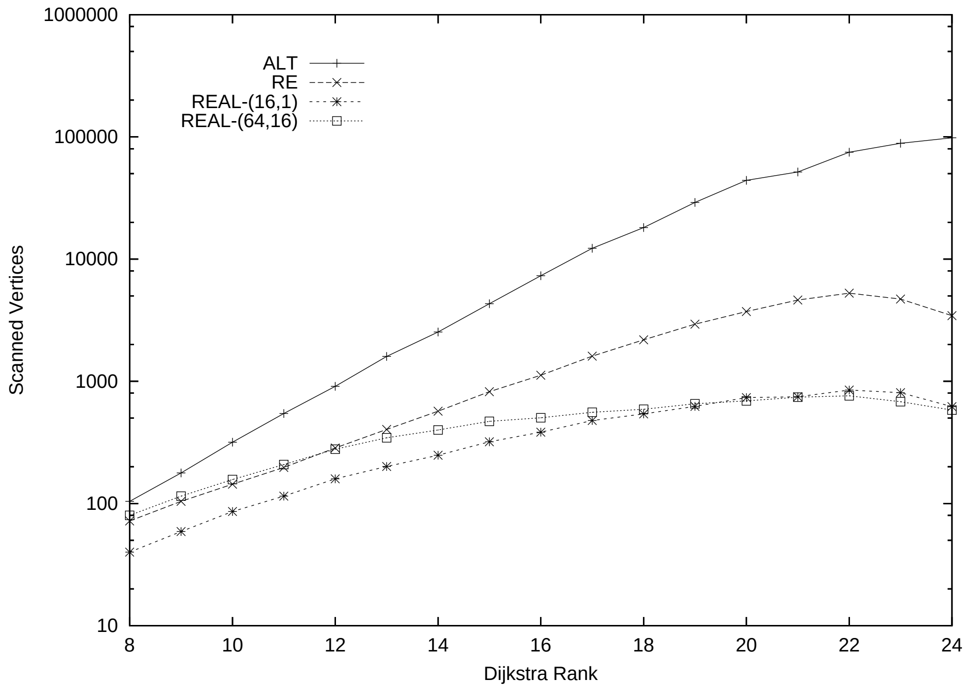
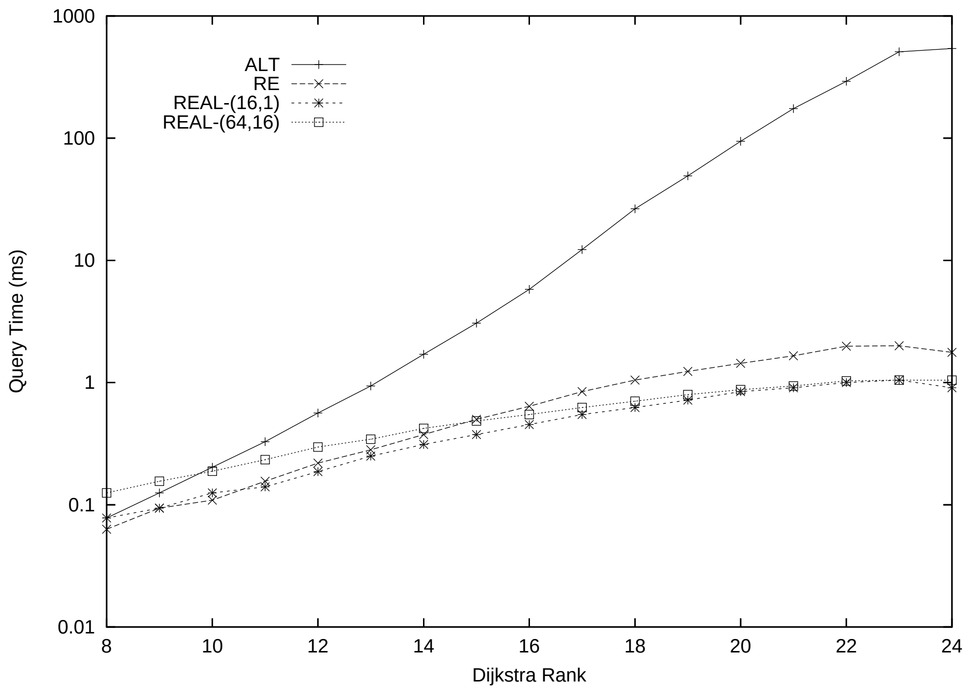


Figure 3: Local queries on Europe with travel times: running times (top) and scan counts (bottom).



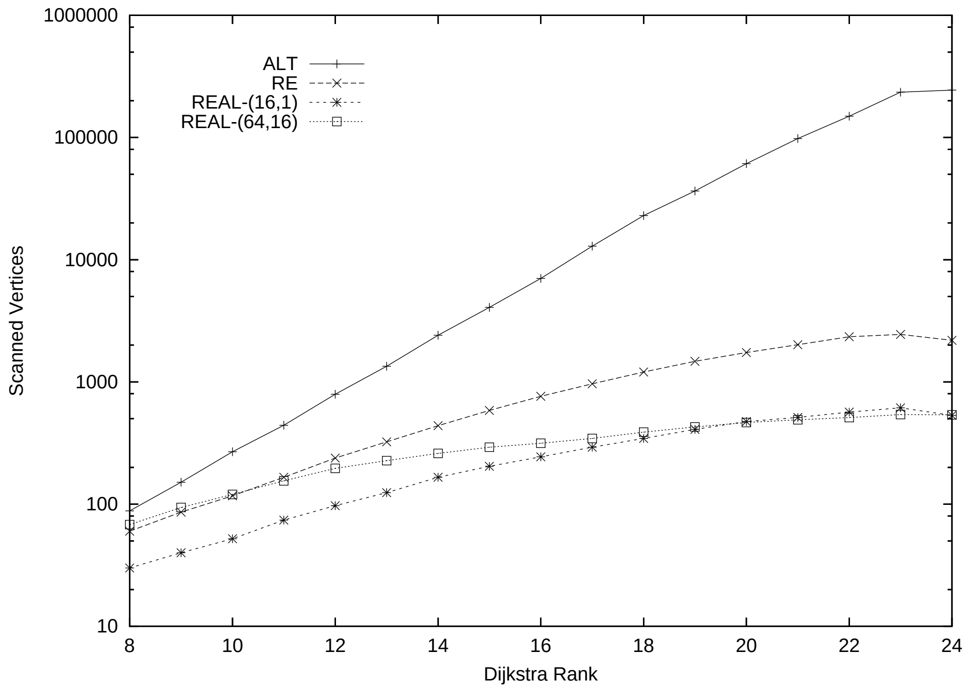


Figure 4: Local queries: running times (top) and average number of vertices scanned (bottom) for USA with travel times.

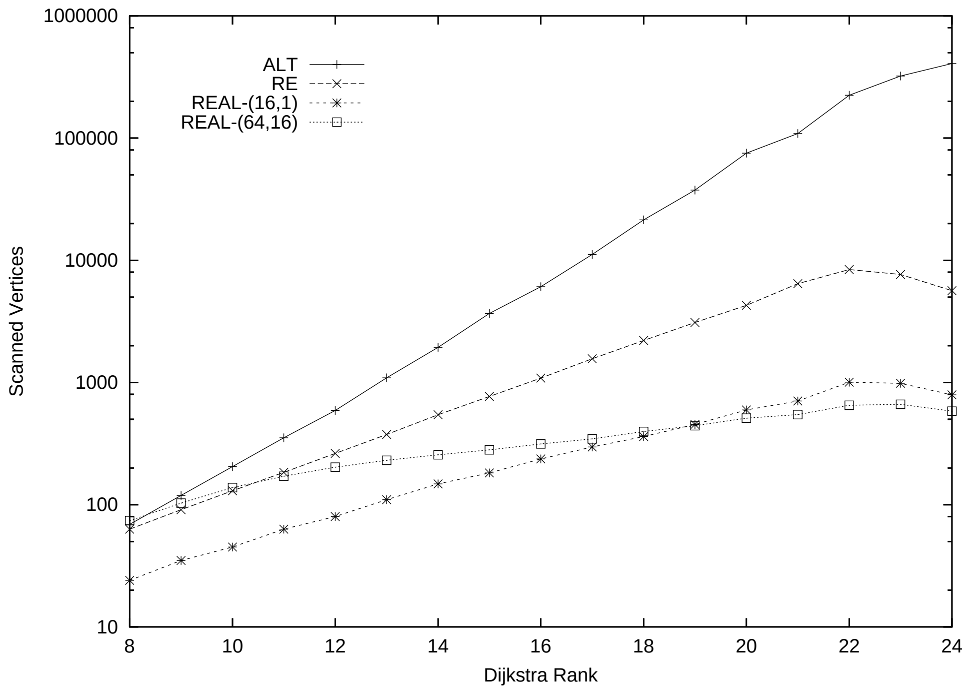


Figure 5: Local queries: average number of vertices scanned on Europe with travel distances.

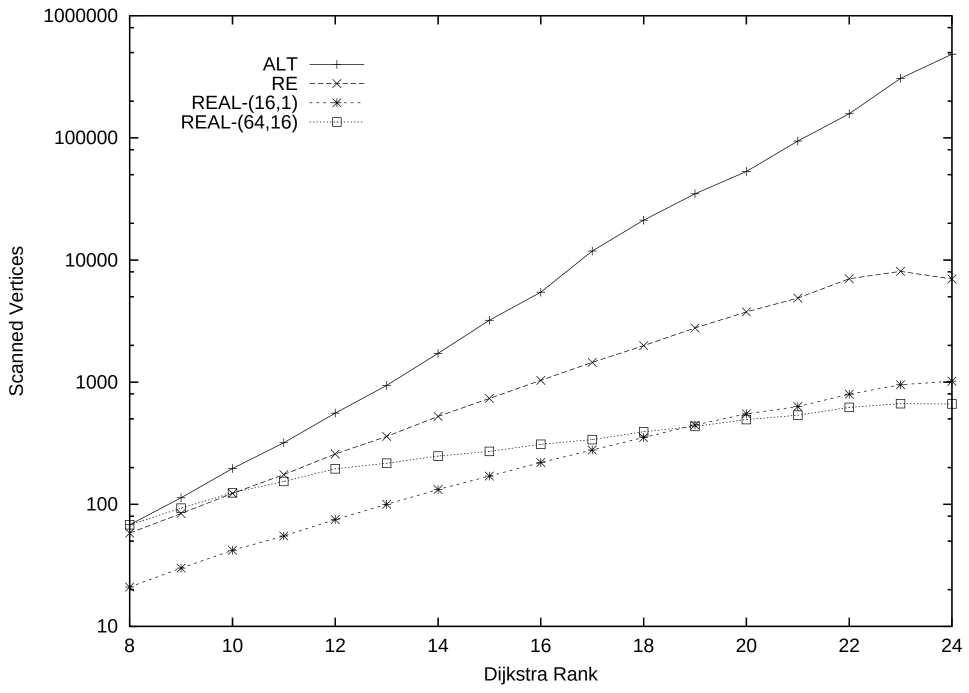


Figure 6: Local queries: average number of vertices scanned on USA with travel distances.

Figures 3 to 6.

Figures 3 and 4 present results for Europe and USA with travel times as the length function. Comparing the two plots in each figure to one another, we observe that the curves representing running times and operation counts are essentially shifted by a constant that is related to the time per vertex scan of each method. Therefore, we discuss operations counts, unless mentioned otherwise. In particular, for travel distances we report only operation counts (Figures 5 and 6).

As random queries tend to produce vertex pairs that are very far apart, our previous discussion applies to local queries with high Dijkstra rank. In particular, in this context ALT is the slowest of the four algorithms, and the REAL variants are the fastest (with very similar performance).

Now consider very local queries, with small Dijkstra rank. REAL-(16,1) scans the fewest vertices, but due to higher overhead its running time is slightly worse than that of RE. REAL-(64,16) mostly visits low-reach vertices and thus fails to take advantage of the landmark data. It scans about the same number of vertices as RE, but is slower due to higher overhead. ALT has the worst asymptotic performance as a function of the Dijkstra rank, but for small ranks it scans only slightly more vertices than RE. As the rank grows, ALT becomes worse than the other codes, and RE becomes worse than REAL-(16,1). REAL-(64,16) improves, and catches up with REAL-(16,1) for large ranks.

In terms of query times and scan counts, REAL-(16,1) is the best code. REAL-(64,16) is somewhat worse for very local queries, but performs similarly for higher Dijkstra ranks. Its main advantage is its lower space requirement.

Table 5: Data for REAL on USA for various landmark/sparsity combinations.

		PREP.	DISK	QUERY		
		TIME	SPACE	AVG	MAX	TIME
METRIC	METHOD	(min)	(MB)	SCANS	SCANS	(ms)
times	REAL-(16,1)	67.2	2962	581	2732	1.03
	REAL-(16,2)	67.2	1926	591	2724	1.30
	REAL-(16,4)	67.2	1408	598	2650	1.27
	REAL-(16,8)	67.2	1148	610	2627	1.33
	REAL-(16,16)	67.2	1019	647	2735	1.28
	REAL-(64,4)	142.5	2962	495	3529	1.09
	REAL-(64,8)	142.5	1926	505	2477	1.06
	REAL-(64,16)	142.5	1408	543	2509	1.01
	REAL-(64,32)	142.5	1148	612	2557	1.09
	REAL-(64,64)	142.5	1019	744	2736	1.14
distances	REAL-(16,1)	90.0	2854	872	5563	1.84
	REAL-(16,2)	90.0	1891	887	5565	2.41
	REAL-(16,4)	90.0	1410	897	5564	2.44
	REAL-(16,8)	90.0	1169	911	5554	2.34
	REAL-(16,16)	90.0	1048	947	5495	2.36
	REAL-(64,4)	153.2	2854	569	3551	1.53
	REAL-(64,8)	153.2	1891	583	3063	1.50
	REAL-(64,16)	153.2	1410	633	3312	1.52
	REAL-(64,32)	153.2	1169	780	3972	1.56
	REAL-(64,64)	153.2	1048	975	3858	1.72

Table 6: Data for REAL on Europe for various landmark/sparsity combinations.

		PREP.	DISK	QUERY		
		TIME	SPACE	AVG	MAX	TIME
METRIC	METHOD	(min)	(MB)	SCANS	SCANS	(ms)
times	REAL-(16,1)	96.2	1864	741	3518	1.17
	REAL-(16,2)	96.2	1245	763	3529	1.52
	REAL-(16,4)	96.2	935	777	3524	1.52
	REAL-(16,8)	96.2	781	790	3543	1.50
	REAL-(16,16)	96.2	703	878	3562	1.53
	REAL-(64,4)	143.9	1864	570	2855	1.12
	REAL-(64,8)	143.9	1245	585	2815	1.11
	REAL-(64,16)	143.9	935	683	2817	1.12
	REAL-(64,32)	143.9	781	909	3104	1.22
	REAL-(64,64)	143.9	703	1591	5746	1.80
distances	REAL-(16,1)	60.8	1926	847	3996	1.50
	REAL-(16,2)	60.8	1295	860	4004	1.98
	REAL-(16,4)	60.8	979	868	3993	1.98
	REAL-(16,8)	60.8	822	883	4001	1.97
	REAL-(16,16)	60.8	743	920	3912	1.95
	REAL-(64,4)	98.0	1926	503	3843	1.11
	REAL-(64,8)	98.0	1295	522	3755	1.08
	REAL-(64,16)	98.0	979	575	2876	1.17
	REAL-(64,32)	98.0	822	681	2856	1.20
	REAL-(64,64)	98.0	743	1053	2868	1.52

Reach-aware landmarks. Next consider Tables 5 and 6. For 16 and 64 landmarks, we vary the fraction of vertices for which landmark data is maintained. As the fraction decreases, we get a substantial improvement in the memory overhead associated with landmarks. The number of vertex scans increases only slightly except for the last steps ($(16, 8)$ to $(16, 16)$ and $(64, 32)$ to $(64, 64)$), where the increase is moderate. The running time remains nearly constant except for the last step because as the fraction decreases, the proportion of cheaper vertex scans that do not use landmarks increases.

One can win in all measures of query complexity. For example, compared to REAL-(16,1) which is not landmark-aware, REAL-(64,16) uses less space and runs faster. Note that our current preprocessing algorithm runs landmark generation on the full graph; for the landmark-aware case, one could work with the graph induced by the high-reach vertices, improving performance.

7.2 Grid Graphs

Grid graphs with uniform random arc lengths are interesting because they do not have an obvious highway hierarchy. We used square 2-dimensional and cube-shaped 3-dimensional grids in our experiments. The 2-dimensional grids were generated using the `spgrid` generator, available at the 9th DIMACS Challenge download page. The 3-dimensional grids were generated by our own generator, `kGrid`. Both families are directed (not symmetric), with a vertex connected to its neighbors in the grid with arcs of length chosen uniformly at random from the range $[1, n]$, where n is the number of vertices. We generated five grids of each size, and report the average results obtained; the only exception is the maximum number of nodes scanned, which is taken over all

Table 7: Data for 2-dimensional grids.

GRAPH	METHOD	PREP. TIME (min)	DISK SPACE (MB)	QUERY AVG SC.	QUERY MAX SC.	QUERY TIME (ms)
65536	D	—	2.2	33752	—	14.83
	BD	—	2.2	21358	51819	16.69
	ALT	0.05	11.7	840	10003	1.30
	RE	1.76	3.4	2147	3706	1.53
	REAL-(16,1)	1.81	12.8	219	1775	0.37
	REAL-(64,16)	1.97	5.8	1947	3151	1.78
131044	D	—	4.5	66865	—	30.72
	BD	—	4.5	41682	103770	40.48
	ALT	0.10	24.1	1407	18232	3.36
	RE	4.70	6.8	3032	5024	2.44
	REAL-(16,1)	4.80	26.3	284	2050	0.65
	REAL-(64,16)	5.14	11.7	2352	3516	2.56
262144	D	—	9.0	134492	—	63.58
	BD	—	9.0	85587	205668	78.80
	ALT	0.21	48.7	2666	34039	7.56
	RE	14.38	13.3	4391	7022	4.00
	REAL-(16,1)	14.58	53.0	384	2701	1.01
	REAL-(64,16)	15.31	23.2	2283	3445	2.93
524176	D	—	18.0	275589	—	112.80
	BD	—	18.0	174150	416925	160.14
	ALT	0.38	97.9	5459	72601	13.68
	RE	33.13	26.0	6288	10074	5.79
	REAL-(16,1)	33.50	105.8	537	3521	1.32
	REAL-(64,16)	34.84	45.9	1971	3213	2.71

five instances.

Table 7 shows computational results for 2-dimensional grid graphs, similar to those used in our previous paper [16]. The table includes results for ALT, RE, two versions of REAL, D (the reference implementation of Dijkstra's algorithm) and BD (our own implementation of the bidirectional version of Dijkstra's algorithm). Note that BD scans fewer nodes than D, but has higher overhead per scan, due to the fact that our implementation of BD is compatible with our more elaborate algorithms, whereas D is more restricted.

Our new preprocessing algorithm (to computes reaches and shortcuts) is an order of magnitude faster on these graphs. The algorithm is still much slower than for road graphs of similar size. We did not, however, attempt to tune the algorithms for grid graphs, except by setting $c = 1.0$ for all iterations of the reach generation algorithm (the use of increasing values of c, starting at 0.5, is tuned for road networks). Query times improve as well, by about a factor of 6. This makes RE competitive with ALT; in fact, RE appears to be asymptotically faster. Performance of REAL-(16,1) improves as well.

Unlike what happens for random queries on road networks, on 2-dimensional grids REAL-(64,16) performs substantially worse than REAL-(16,1). This is probably because the graphs are relatively small and well-structured, thus making an increase in the number of landmarks not as advantageous as for large road networks. The fact that the relative performance gap between the two algorithms narrows as the graph size increases confirms this conjecture. For the smallest grid in our test, REAL-(64,16) is almost five times slower than REAL-(16,1) and even slower than RE. For the largest grid, however, it is faster than RE and only about twice as slow as REAL-(16,1).

The experiments on 2-dimensional grids show that reaches help even when a graph does not have an obvious highway hierarchy, and that the applicability of REAL is not restricted to road networks. On the largest grid, it is five times faster than ALT, and two orders of magnitude faster than the bidirectional Dijkstra algorithm.

Table 8: Data for 3-dimensional grids.

		PREP. TIME	DISK SPACE	QUERY		
GRAPH	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
32768	D	—	1.6	16747	—	6.90
	BD	—	1.6	5840	18588	4.10
	ALT	0.02	4.5	483	5264	0.61
	RE	4.76	2.6	3166	7433	2.83
	REAL-(16,1)	4.78	5.5	350	2609	0.74
	REAL-(64,16)	4.86	3.3	3819	8056	3.73
64000	D	—	3.1	31759	—	13.54
	BD	—	3.1	11338	36732	8.63
	ALT	0.04	10.1	723	8931	1.03
	RE	12.70	5.0	5053	11860	5.07
	REAL-(16,1)	12.74	12.0	473	3824	1.17
	REAL-(64,16)	12.91	6.8	5926	12839	7.00
132651	D	—	6.5	66045	—	32.98
	BD	—	6.5	23738	79600	23.64
	ALT	0.09	24.0	1183	12104	2.01
	RE	39.45	10.3	8463	18379	10.01
	REAL-(16,1)	39.55	27.9	688	5605	2.05
	REAL-(64,16)	39.96	14.7	9674	20103	13.58
262144	D	—	12.8	133552	—	89.65
	BD	—	12.8	48699	161036	52.31
	ALT	0.23	50.1	2286	29926	9.20
	RE	116.35	19.9	13156	27434	19.29
	REAL-(16,1)	116.57	57.3	1077	7926	5.12
	REAL-(64,16)	117.47	29.3	14734	29136	24.56

Next we discuss 3-dimensional grid data, given in Table 8. Since these graphs have higher degree, we used $c = 2.0$ when generating shortcuts. The table shows that both reach and landmark heuristics are less effective as on 2-dimensional grids. ALT queries are only modestly slower, however, and ALT preprocessing time is not affected much. In contrast, RE preprocessing becomes asymptotically slower—the time roughly triples when the graph size doubles. With this rate of growth, it would take about two months to preprocess a grid with 16 000 000 vertices, comparable in size to the Europe graph.

Next we consider RE and ALT queries. The average number of scans for RE is six to seven times greater than that for ALT. The running time is about five times greater, except for the biggest problem, where it is greater by only slightly over a factor of two. The average number of scans suggest that REAL-(16,1) has a small asymptotic advantage over ALT and RE. It is slightly slower than ALT on the smallest problem and a little faster on the largest one.

REAL-(64,16) is the slowest code and on average scans more vertices than RE. We note that

REAL and RE use different strategies for balancing the forward and the reverse search. REAL strictly alternates between the two searches, while RE balances the radii of the two balls. This explains the discrepancy.

These results show that 3-dimensional grids are on the edge of usefulness of current reach-based methods. Preprocessing is useful only for large graphs, but on those it is expensive. Whether this is a basic limitation of the method, or a limitation of our preprocessing algorithm, is an interesting open question.

7.3 Exact Reaches and the Refinement Step

To illustrate the effectiveness of our new algorithm for computing exact reaches, we ran it on BAY (with travel times), which has 321 270 vertices. The standard algorithm for computing exact reaches on this graph takes almost 20 hours; the new algorithm takes 2 hours and 18 minutes. The speedup is even bigger for larger graphs. Unfortunately, the algorithm is still far from being practical. The partial trees algorithm takes only 30 seconds and, because it changes the input graph (by adding shorcuts), it actually finds better reaches. As we have seen, with the normal preprocessing algorithm RE scans 751 vertices and takes $0.41ms$ on average. With exact reaches (but no shortcuts), these figures increase to $6,345$ and 2.92 ms, respectively.

Of course, one can always use the partial-trees algorithm to generate shortcuts and then compute exact reaches in the resulting graph. Even though the graph with shortcuts has about 50% more arcs than the original one, computing exact reaches on it took 2 hours and 19 minutes, about the same as before. But RE queries became significantly better: the number of scanns is reduced to 473, and the average time to 0.25 ms. This confirms the importance of shortcuts.

Even though computing exact reaches does improve queries, the preprocessing time is prohibitive. A more practical use of the exact reach algorithms is as an addition to the partial trees heuristic. As already mentioned, it is used in the *refinement step* of our algorithm. We perform exact reach computation on the subgraph induced by the δ vertices with highest reach. All the experiments we have presented so far use $\delta = \lceil 5\sqrt{n} \rceil$, which provides a good balance between reach quality and preprocessing time.

As already mentioned, when there is a refinement step, the algorithm switches to exact reach computation during its first phase as soon as it starts an iteration where the number of vertices left is at most $\delta/2$. Once the exact arc reaches are computed, the algorithm eliminates the 1/3 that have small reach and adds shortcuts to the graph. This process is repeated until there are no arcs left. As a result, significantly more shortcuts are added at the final iterations; the effect is to reduce the reaches of high-reach vertices, at the expense of making the subgraph induced by them denser.

Table 9 shows how the choice of δ affects the performance of the RE algorithm on the USA graph with travel times, in terms of both preprocessing and queries. Besides the original value of δ (24 469), we also tested 97 873 (which is equal to $\lceil 20\sqrt{n} \rceil$) and zero (i.e., no refinement step). For each value, the table shows the preprocessing time and the query performance on 1 000 random pairs.

As one would expect, one can trade longer preprocessing for better query times. With $\delta =$ 24 469, the algorithm scans 10% fewer vertices on average than it would without the refinement step. The running time, however, does not decrease accordingly because high-reach vertices have higher degree. A more significant increase in δ, however, can substantially decrease the running time.

The results of this section show that there is still room for improvement in our algorithm. At

Table 9: Performance of RE on the USA with travel times when the size of the refinement step varies.

δ	PREP. TIME	QUERY		
	(min)	AVG SC.	MAX SC.	TIME (ms)
0	32.4	2555	5242	1.88
24469	48.6	2264	4667	1.86
97873	177.3	1803	4228	1.53

the very least, it could be better tuned. Ideally, however, one would like to have a more stable algorithm for generating and computing reaches once the graph is small enough (and only vertices of very high reach remain).

7.4 Retrieving the Shortest Path

The query times reported so far for RE and REAL consider only the task of finding the shortest path on the graph with shortcuts. Although this path has the same length as the corresponding path in the original graph, it has much fewer arcs. On USA, for example, the shortest path between a random pair of vertices has around 5 000 vertices in the original graph, but only about 30 in the graph with shortcuts.

Our algorithm can retrieve the original path from the path with shortcuts in time proportional to the size of the original path. For this, it uses an *arc map*, which maps each arc to the two arcs it replaces. The arc map is built during the preprocessing step and has roughly the same size as the graph with shortcuts, but it is *not* included in the "disk space" column in previous tables, since not all applications require it.

Since paths in the original graph have many more arcs than the paths found by RE or REAL, retrieving the original path can be relatively costly. To measure this, we reran the RE algorithm. After each query, we dumped the list of arc identifiers to an array, and at the same time computed the sum of the costs of these arcs. Even though we already know what the sum will be, this procedure is a good approximation of what an actual application might do.

Table 10 shows the running time of the standard RE algorithm on all road networks we studied, together with the time that would be necessary to retrieve the original paths. Note that retrieving the original path would increase the query times of RE by up to 50%. With REAL, the absolute time to retrieve the path would be the same; since REAL is faster than RE, path retrieval would represent an even higher fraction of the total time. In particular, if one finds and retrieves a shortest path on the USA graph with the distance metric using REAL-(64,16), 45% of the time will be spent retrieving the original path, almost as much as actually finding it.

8 Concluding Remarks

For computing driving directions, our algorithms are extremely practical for large road networks, both for servers and, because they scan so few vertices, even portable devices. Our improvements made the algorithms faster and reduced memory requirements. The algorithms also work well on 2-dimensional grids.

An interesting direction of future research is to make the algorithm effective on a wider range of graphs. We have shown that it has only limited applicability to 3-dimensional grids, for instance.

Table 10: Running times of RE algorithm on random pairs: SEARCH ONLY is the time to find the path on the graph with shortcuts (in milliseconds); PATH RETRIEVAL is the additional time necessary to retrieve the original path, given both in milliseconds and as a fraction of the search time. Note that the search time does *not* include the path retrieval time. The top half of the table uses the travel time metric; the bottom half uses travel distances.

GRAPH	SEARCH ONLY	PATH RETRIEVAL	
	(ms)	TIME (ms)	OVERHEAD (%)
NY	0.672	0.031	4.6
BAY	0.407	0.061	15.0
COL	0.375	0.172	45.9
FLA	0.484	0.094	19.4
NW	0.625	0.156	25.0
NE	0.985	0.155	15.7
CAL	0.765	0.345	45.1
LKS	1.125	0.188	16.7
E	1.047	0.375	35.8
W	1.140	0.595	52.2
CTR	2.203	0.687	31.2
USA	1.860	0.796	42.8
Europe	3.485	0.421	12.1
NY	1.000	0.172	17.2
BAY	0.578	0.031	5.4
COL	0.656	0.172	26.2
FLA	0.734	0.219	29.8
NW	0.891	0.203	22.8
NE	2.031	0.406	20.0
CAL	1.234	0.235	19.0
LKS	3.141	0.749	23.8
E	2.641	0.625	23.7
W	1.938	0.453	23.4
CTR	6.859	1.047	15.3
USA	6.063	1.250	20.6
Europe	5.812	1.048	18.0

Many open problems remain. The performance of our best query algorithms depend crucially on the quality of the reaches available. Devising faster algorithms to compute exact reaches (or at least better reaches) might make our queries even faster. More importantly, the problem of finding good shortcuts deserves a more detailed study. Our current algorithm uses a series of heuristics to determine when to add shortcuts, but there is no reason to believe they are the best that can be done. It is also desirable to get theoretical justification for the good practical performance of our algorithms.

Finally, it would be interesting to find other applications for the concepts and techniques we developed. For example, reach information may also be useful for highway design: high-reach local roads are natural candidates for becoming highways with increased speed limits.

References

[1] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.

[2] L. J. Cowen and C. G. Wagner. Compact Roundtrip Routing in Directed Networks. In *Proc. Symp. on Principles of Distributed Computation*, pages 51–59, 2000.

[3] G. B. Dantzig. *Linear Programming and Extensions.* Princeton Univ. Press, Princeton, NJ, 1962.

[4] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9/, 2006.

[5] E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[6] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

[7] J. Doran. An Approach to Automatic Problem-Solving. *Machine Intelligence*, 1:105–127, 1967.

[8] D. Dreyfus. An Appraisal of Some Shortest Path Algorithms. Technical Report RM-5433, Rand Corporation, Santa Monica, CA, 1967.

[9] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. 42nd IEEE Annual Symposium on Foundations of Computer Science*, pages 232–241, 2001.

[10] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[11] G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.

[12] A. V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. In *Proc. 9th ESA, Lecture Notes in Computer Science LNCS 2161*, pages 230–241. Springer-Verlag, 2001.

[13] A. V. Goldberg. Shortest Path Algorithms: Engineering Aspects. In *Proc. ESAAC '01, Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[14] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[15] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. Technical Report MSR-TR-2005-132, Microsoft Research, 2005.

[16] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proc. 7th International Workshop on Algorithm Engineering and Experiments.* SIAM, 2006.

[17] A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. In P. M. Pardalos, D. W. Hearn, and W. W. Hages, editors, *Lecture Notes in Economics and Mathematical Systems 450 (Refereed Proceedings)*, pages 292–327. Springer Verlag, 1997.

[18] A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proc. 7th International Workshop on Algorithm Engineering and Experiments*, pages 26–40. SIAM, 2005.

[19] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. 6th International Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[20] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on System Science and Cybernetics*, SSC-4(2), 1968.

[21] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A Fast Algorithm for Finding Better Routes by AI Search Techniques. In *Proc. Vehicle Navigation and Information Systems Conference*. IEEE, 1994.

[22] R. Jacob, M.V. Marathe, and K. Nagel. A Computational Study of Routing Algorithms for Realistic Transportation Networks. *Oper. Res.*, 10:476–499, 1962.

[23] P. Klein. Preprocessing an Undirected Planar Network to Enable Fast Approximate Distance Queries. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 820–827, 2002.

[24] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of shortest path and constrained shortest path computation. In *WEA*, pages 126–138, 2005.

[25] Jr. L. R. Ford. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956.

[26] Jr. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[27] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *IfGIprints 22, Institut fuer Geoinformatik, Universitaet Muenster (ISBN 3-936616-22-1)*, pages 219–230, 2004.

[28] PTV Traffic Mobility Logistic. Western europe road network. http://www.ptv.de/, 2006.

[29] U. Meyer. Single-Source Shortest Paths on Arbitrary Directed Graphs in Linear Average Time. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 797–806, 2001.

[30] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *WEA*, pages 189–202, 2005.

[31] T. A. J. Nicholson. Finding the Shortest Route Between Two Points in a Network. *Computer J.*, 9:275–280, 1966.

[32] I. Pohl. Bi-directional Search. In *Machine Intelligence*, volume 6, pages 124–140. Edinburgh Univ. Press, Edinburgh, 1971.

[33] P. Sanders and D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. In *Proc. 13th Annual European Symposium Algorithms*, 2005.

[34] P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proc. 14th Annual European Symposium Algorithms*, 2006.

[35] D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. Master's thesis, Department of Computer Science, Universitt des Saarlandes, Germany, 2005.

[36] F. Schulz, D. Wagner, and K. Weihe. Using Multi-Level Graphs for Timetable Information. In *Proc. 4th International Workshop on Algorithm Engineering and Experiments*, pages 43–59. LNCS, Springer, 2002.

[37] R. Sedgewick and J.S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, 1:31–48, 1986.

[38] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[39] M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. Assoc. Comput. Mach.*, 46:362–394, 1999.

[40] M. Thorup. Compact Oracles for Reachability and Approximate Distances in Planar Digraphs. In *Proc. 42nd IEEE Annual Symposium on Foundations of Computer Science*, pages 242–251, 2001.

[41] DC US Census Bureau, Washington. UA Census 2000 TIGER/Line files. http://www.census.gov/geo/www/tiger/tigerua/ua.tgr2k.html, 2002.

[42] D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *European Symposium on Algorithms*, 2003.

[43] F. B. Zhan and C. E. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transp. Sci.*, 32:65–73, 1998.

[44] F. B. Zhan and C. E. Noon. A Comparison Between Label-Setting and Label-Correcting Algorithms for Computing One-to-One Shortest Paths. *Journal of Geographic Information and Decision Analysis*, 4, 2000.