DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

Reach for A*: Shortest Path Algorithms with Preprocessing

Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck

Abstract. We study the point-to-point shortest path problem with preprocessing. Given an input graph, we preprocess it so as to be able to answer a series of source-to-destination queries efficiently. Our work is motivated by an algorithm of Gutman [ALENEX'04], based on the notion of *reach*, which measures how important each vertex is with respect to shortest paths. We present a simplified version of his algorithm that does not require explicit lower bounds during queries. We also show how the addition of shortcuts to the graph greatly improves the performance of both preprocessing and queries. Finally, we combine a reach-based algorithm with landmark-based A* search to obtain a wide range of space-time trade-offs. For our motivating application, driving directions for road networks, the resulting algorithm is very efficient and practical. The road networks of the USA and Western Europe have roughly 20 million vertices, but on average our algorithm must visit fewer than a thousand to find the distance between two points. Our algorithm also works reasonably well on 2-dimensional grid graphs with random arc weights.

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. Although there is a single input graph, typically there are many source/destination queries. We therefore allow preprocessing of the input graph, but limit the size of the precomputed data to a (moderate) constant times the 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 [**3, 5, 10, 11, 15, 16, 17, 18, 22, 27, 36, 47, 51**]. Near-optimal algorithms are known both in theory, with near-linear time bounds, and in practice, with running times within a small constant factor of the breadth-first search time.

The P2P problem with no preprocessing has been addressed, for example, in [**26, 39, 45, 52**]. 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 is

due to Fakcharoenphol and Rao [**14**]. Algorithms for approximate shortest paths that use preprocessing have also been studied; see e.g., [**4, 28, 48**].

Previous work on exact P2P algorithms with preprocessing includes, e.g., [**19, 23, 24, 30, 33, 37, 40, 43, 44, 50**]. 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*, which computes auxiliary data, and a *query algorithm*, which computes an answer for a given s-t pair.

Gutman [**24**] introduced the notion of *vertex reach*. Informally, the reach of a vertex v is large if v is close to the middle of some long shortest path and small otherwise. Gutman proposes a simple modification of Dijkstra's algorithm that can 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 efficiency can be improved if reaches are combined with Euclidean-based A* search [**25, 38**], which uses lower bounds on the distance to the destination to direct the search towards it.

Goldberg and Harrelson [**19**] (see also [**23**]) have shown that the performance of A* search (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 set of special vertices, the landmarks. During queries, one can use this information, together with the triangle inequality, to obtain lower bounds on the distance between any two vertices in the graph. This leads to the ALT (A* search, landmarks, and triangle inequality) method for the point-to-point problem.

Sanders and Schultes [**40, 41**] use the notion of *highway hierarchies* to design efficient algorithms for road networks. The preprocessing algorithm builds a hierarchy of increasingly sparse *highway networks*; queries start at the original graph and gradually move to upper levels of the hierarchy, greatly reducing the search space. To magnify the natural hierarchy of road networks, the algorithm adds *shortcuts* to the graph: additional edges with the same lengths as the original shortest paths between their endpoints.

In this paper, our first major contribution is to show that reaches can be used to prune the search even when explicit lower bounds (such as those obtained by Euclidean bounds or landmarks) are not available. By making the search bidirectional, we can use the bounds implicit in the search itself to prune it.

The second major contribution of our paper is to explain how shortcuts can be used in the context of reach-based algorithms. This significantly improves both preprocessing and query efficiency. The resulting algorithm is called RE. Although the preprocessing algorithm needs to be modified in order to generate shortcuts, the query algorithm remains the same regardless of whether shortcuts are present or not.

Our third major contribution is to show that the ALT method can be combined with reach-based pruning in a natural way, leading to an algorithm we call REAL. We also show that by maintaining landmark data only for high-reach vertices, one can greatly reduce the memory requirements of REAL. Furthermore, if we use some of the saved space for more landmarks, we can win in both space and time.

In addition, we introduce several improvements to preprocessing and query algorithms for landmark- and reach-based methods.

We evaluate the efficiency of our algorithms through experiments, mostly on road networks with three length functions (travel times, travel distances, and unit lengths). Our experiments show practical results for all three metrics. The road networks of Western Europe or the United States, each with roughly 20 million vertices, can be preprocessed in an hour or less. The average query with our fastest algorithm takes roughly one millisecond on a standard workstation and scans fewer than 1000 vertices.

We have also obtained good results for 2-dimensional grids with random arc lengths. Although not as good as for road networks, the results prove that our techniques have more general applicability. To show the limitations of these techniques, we also experimented with grids of higher dimension and with random graphs. On these inputs, our heuristics do not achieve significant performance gains.

This paper is organized as follows. Section 2 reviews Dijkstra's algorithm and some variants, and establishes the notation used throughout the paper. Section 3 formalizes the definition of *reach* and explains how it can be used to prune a point-to-point shortest path search. Section 4 deals with reach computation: how reaches (or upper bounds on reaches) can be computed in reasonable time during the preprocessing stage of our algorithm. Section 5 reviews the ALT algorithm and shows how it can be combined with reach-based pruning. Section 6 presents our experimental results. Final remarks are made in Section 7, including a brief comparison with recent work presented at the 9th DIMACS Implementation Challenge [**8**].

2. Preliminaries

The input to the preprocessing stage of a P2P shortest path algorithm is a directed graph $G = (V, A)$ with n vertices and m arcs, and nonnegative lengths $\ell(a)$ for every arc a. Besides the source s and the sink t, the query stage takes as input the data produced by the preprocessing stage, which includes the graph itself (potentially modified) and auxiliary information. 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 [**31, 32**] finds shortest paths from the source to all vertices in the graph. It works as follows (see e.g., [**46**]). 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 $S(v) =$ `unreached` for every vertex v. The method starts by setting $d(s) = 0$ and $S(s) =$ `labeled`. While there are labeled vertices, it picks a labeled vertex v and *scans* it by *relaxing* all arcs out of v and setting $S(v) =$ `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 $S(w) =$ `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 [**11**] (and independently Dantzig [**5**]) observed that if ℓ is nonnegative and v is a labeled vertex with the smallest distance label, then $d(v)$ is exact. The labeling method with the minimum label selection rule is known as *Dijkstra's algorithm*.

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* [**5, 13, 38**] 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, 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 that is shorter than those seen before, so we update μ and its path accordingly. We perform similar updates during the reverse search. Note that, to maintain the current best path, it is enough to remember the arc (v, w) whose relaxation gave the current value of μ, as the forward search maintains the s-v path of length $d_f(v)$ and the reverse search maintains the w-t path of length $d_r(w)$. We can alternate between the searches in any order. In our experiments, we strictly alternate between the searches, balancing the work between them.

Intuitively, the bidirectional algorithm searches two touching balls centered at s and t: we can stop when the search in one direction selects a vertex already scanned in the other. A slightly tighter criterion is to terminate the search when $\text{top}_f + \text{top}_r \geq \mu$, where top_f and top_r denote the top keys (values) in the forward and reverse priority queues, respectively (i.e., the smallest labels of unscanned vertices in each direction). To see why this is a valid criterion, suppose there exists an s-t path P whose length is less than μ. Then there must exist an arc (v, w) on this path such that $\text{dist}(s, v) < \text{top}_f$ and $\text{dist}(w, t) < \text{top}_r$, which means that v and w must have been scanned already. Suppose, without loss of generality, that v was scanned first; then, when scanning w, path P would have been detected. Since it was not, P cannot exist and μ must indeed be the length of the shortest s-t path.

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). See Figure 1. 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. Throughout this paper, we assume that shortest paths are unique. If the input has ties, these can be broken in several ways; we describe our tie-breaking procedure in Section 4.6.

The intuition behind the notion of reach is simple. A vertex has high reach only if it is close to the middle of some very long shortest path. On road networks,

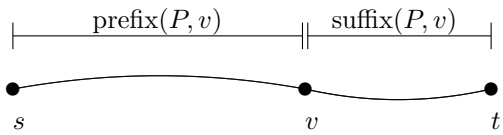

FIGURE 1. The reach of v with respect to the shortest path P between s and t is the minimum between the lengths of its prefix and its suffix (with respect to v).

high-reach vertices roughly correspond to highways, whereas low-reach vertices correspond to local intersections. Take a vertex v representing an intersection between two local roads in a small town. There are shortest paths containing v that start close to v and end somewhere far away, and shortest paths containing v that start far away and end close to v. Usually, however, it is not the case that there is a *shortest* path that starts far away from v, passes through v, and ends far away from v.

The knowledge that a particular vertex v has low reach can be used to prune it during searches. Once we are far away from the source s and the target t, there is no point in visiting v: we know it cannot be on the shortest path from s to t. The remainder of this section will make these observations more formal.

For large graphs, computing exact reaches is impractical with current algorithms. Instead, we efficiently compute *upper bounds* on the reach of every vertex, which is enough for our purposes. Section 4 will explain in detail how this can be done. For now, assume that we have valid reach upper bounds; how they were obtained is immaterial.

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 us to use reaches for pruning an s-t 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.

This holds for the bidirectional algorithm as well.

Note that upper bounds on reaches are not enough: we still need lower bounds on distances from s (which the search itself can provide) and to t. Gutman [**24**] proposed using Euclidean lower bounds to find lower bounds on distances to t. Unfortunately, this only works when vertex coordinates are available, which is not always the case. Even when they are available, as in the case of road networks, the lower bounds are not particularly tight, especially when length functions other than travel distances are used.

We propose a simpler (and more effective) strategy: make the search bidirectional, and extract implicit lower bounds from the bidirectional search itself. During an execution of a bidirectional version of Dijkstra's algorithm, 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 key 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 target t. The same idea applies to the reverse search: we use the

topmost key in the forward heap as a lower bound on the distance from the source to any vertex not yet scanned in the forward 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). As in the original case, we can alternate between the searches in any way. 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 $\bar{r}(v) < \gamma$.

A variant of this method is the *self-bounding* algorithm, which can prune a vertex based on its own distance label, regardless of the other direction. 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.

The self-bounding algorithm can safely ignore the lower bound to the destination because 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, the self-bounding algorithm must check if w has been scanned in the opposite direction; if so, it must check if the candidate path using (v, w) is the shortest path seen so far.

The natural *distance-balanced* 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. 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. This algorithm, which we call RE, is the one we tested in our experiments, given its simplicity and the fact that it can be naturally combined with A* search,

as Section 5.2 will show. Although it could be implemented with only one priority queue, we use two for consistency with the other algorithms we implemented.

3.1. Early Pruning. Our implementation checks whether a vertex w can be pruned not only when scanning it, but also when considering whether to insert it into the heap or not. We call this *early pruning*. When processing an arc (v, w) in the forward search, we are actually evaluating a path from s to t that passes through v and w. The distance from s to w on this path is $d_f(v) + \ell(v, w)$; a lower bound on the distance from w to t is γ (the topmost value in the reverse heap), assuming that w has not been scanned yet. If $\bar{r}(w) < \min\{d_f(v) + \ell(v, w), \gamma\}$, we can prune the search at w. The same procedure applies to the reverse search.

Besides avoiding heap insertions, early pruning can actually reduce the number of arcs we have to scan. Suppose that the arcs in the adjacency list of v are sorted in decreasing order of upper bounds on reaches. In other words, if (v, w) and (v, x) are such that $\bar{r}(w) < \bar{r}(x)$, then (v, x) appears before (v, w) in the adjacency list of v. If, when scanning (v, w), we determine that $\bar{r}(w) < \gamma$ and $\bar{r}(w) < d_f(v)$, then we can not only prune (v, w), but implicitly prune all remaining arcs in the adjacency list of v. Note that there will be cases where $\bar{r}(w)$ will be greater than $d_f(v)$ but smaller than $d_f(v) + \ell(v, w)$; we can still prune w, but we must continue traversing the adjacency list of v. To make this optimization possible, we can sort the adjacency lists when reading the graph for queries (as we did in our experiments) or during the preprocessing step.

To ensure that implicit pruning is indeed correct, there is still a special case we must address. As described, the routine assumes, when processing (v, w), that γ is a valid lower bound on the distance to t from every neighbor z of v that appears after (v, w) in the adjacency list. This may not be true if z has already been scanned: the algorithm will miss the arc (v, z) when scanning v, and it can conceivably be part of the shortest path. Fortunately, this arc will have already been scanned from z during the reverse search. At that point, however, v had not yet been scanned in the forward direction, and therefore the algorithm will not have checked if (v, z) belongs to the shortest path. This can remedied by performing an additional check when scanning v itself: if v is labeled in the other direction, the algorithm checks whether $d_f(v) + d_r(v) \leq \mu$ and updates the shortest path seen so far accordingly.

3.2. 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. During repeated searches, most vertices visited have high 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 has high locality.

Instead, we adopt the following approach to order the vertices. We partition the vertices into two equal-sized sets: the first contains the $n/2$ 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, such as reach-aware landmarks (described in Section 5.4).

4. Reach Computation

Having seen how reach upper bounds are used to prune the search, we now turn to the problem of computing these bounds. The standard algorithm for computing exact reaches builds shortest path trees from each vertex in the graph. The shortest path tree rooted at vertex r compactly represents all shortest paths that start at r.[1] 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 the tree). The reach of v with respect to the entire graph is the maximum reach of v with respect to all shortest path trees.

Building n shortest path trees is too expensive for large road networks. Fortunately, as already mentioned, it is enough to compute upper bounds on reaches. Gutman [**24**] suggested an algorithm for this purpose that works in rounds. Each round tries to find upper bounds for reaches that are smaller than a threshold ϵ by growing *partial shortest path trees* of depth greater than 2ϵ. Intuitively, a high-reach vertex (i.e., a vertex with reach at least ϵ) must be close to the middle of some shortest s-t path of length slightly bigger than 2ϵ. This path will be among those considered when a partial shortest path tree is grown from s. At the end of a round, vertices with bounded reach are removed from the graph, the threshold ϵ is increased (by a multiplicative factor), and the procedure is repeated in the resulting subgraph, now with a larger threshold. This process continues until all reaches have been bounded.

Since the threshold increases substantially as the algorithm progresses, so does the depth of the shortest path trees obtained. Therefore, the efficiency of the algorithm depends crucially on how fast the graph shrinks due to the elimination of low-reach vertices. Intuitively, one would like the number of vertices in the partial trees to remain approximately constant from one round to the next. Unfortunately, when Gutman's algorithm is applied to road networks, the trees increase in size, rendering the algorithm impractical for large graphs.

4.1. Our Approach. Our algorithm is based on the same basic approach (growing partial trees) as Gutman's, but we suggest several improvements that lead to significant speedups.

Our most important improvement is adding *shortcut arcs* (or simply *shortcuts*) to the graph during the preprocessing procedure. A shortcut (v, w) is a new arc with length equal to that of an existing path between v and w. If we break ties appropriately (giving preference to shortcuts), we may decrease the reach of internal vertices on the original v-w path. This speeds up both preprocessing (because more vertices are eliminated after each round) and queries (because vertices with lower reach are more likely to be pruned). The resulting algorithm becomes practical for large road networks, such as those of the USA and Western Europe.

Our method for generating shortcuts is based on the one suggested by Sanders and Schultes [**40, 41**] in the context of highway hierarchies. The idea is to bypass a low-degree vertex v by adding for each pair of arcs $[(u, v), (v, w)]$ an arc (u, w) of length $\ell(u, v) + \ell(v, w)$ and deleting v and all arcs adjacent to it. To avoid introducing too many arcs, we prefer to bypass vertices of low degree. Although this shortcut strategy is local, its repeated application may introduce shortcuts

[1] Note that r can represent either a reach value (as a function) or a tree root. We rely on context to resolve the ambiguity.

representing very long paths. This happens on road networks, where these paths often correspond to portions of highways between two important exits.

A second important novelty of our preprocessing algorithm is that it 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), denoted by $r(v, w)$, is the maximum, over all shortest paths P containing (v, w), of the reach of (v, w) with respect to P. The main advantage of computing arc reaches is that it allows for more efficient shortcutting: a high-reach vertex can be bypassed as soon as enough low-reach arcs incident to it are eliminated.

We now outline the preprocessing algorithm in more detail. Like Gutman's algorithm, it works in *rounds* (or *levels*). At level i, it tries to bound all arc reaches below some threshold ϵ_i, which grows exponentially with i. Level i starts by removing from the graph every arc whose reach was bounded in the previous level. It then eliminates (bypasses) some low-degree vertices by adding shortcuts between its original neighbors. Finally, it grows partial shortest path trees from all remaining vertices, and uses these trees to find upper bounds on the reaches of arcs whose reaches are less then ϵ_i. The algorithm proceeds until all arcs have been eliminated, i.e., until all reaches have been bounded.[2]

At this point, arc reaches are converted to vertex reaches, which are used during queries. We could employ arc reaches during queries, but vertex reaches require less space and are easier to use.

Although the partial shortest path trees grown in a given level contain only vertices and arcs that have not been eliminated from the graph, our goal is to find reach upper bounds that are valid for the original graph. We must therefore 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 implicitly, by associating *penalties* with each vertex v in the current graph, as Section 4.2 will explain in detail. Intuitively, penalties are used to artificially extend the lengths of all shortest paths that start or end at v in the original graph, implicitly accounting for the fact that these paths could be extended to vertices that have been previously eliminated by the preprocessing routine.

Unfortunately, although they help find valid upper bounds, penalties often lead to overly conservative reach estimates. We may end up with upper bounds that are significantly above the actual reach values, which makes pruning less effective during queries. This is especially true for high-reach vertices, the last to have their reaches bounded. This is unfortunate, since they are arguably the most important vertices in the graph: queries prune most low-reach vertices and spend most of their time traversing high-reach ones.

To minimize this issue, the preprocessing algorithm also has a *refinement phase*. After all reach upper bounds are obtained, we compute exact vertex reaches on the subgraph induced by the δ vertices with highest reach upper bound, where δ is a user-defined parameter (set to $2\lceil\sqrt{n}\rceil$ in most of our experiments). The remaining vertices are considered only implicitly, as penalties.

[2]We reiterate that these arcs are "eliminated" during preprocessing only; all original arcs and shortcuts will be present in the final graph, on which queries will be performed.

The remainder of this section describes in detail each component of the preprocessing algorithm: the partial-trees procedure (Section 4.2), shortcut generation (Section 4.3), and the refinement step (Section 4.4). Other implementation details, including parameter choices, are discussed in Section 4.5.

4.2. Approximate Reaches: Growing Partial Trees. We now describe the main routine executed in each iteration of our preprocessing algorithm. Given a graph $G = (V, A)$ and the threshold ϵ, our goal is to find valid reach upper bounds for arcs in A whose actual reaches are smaller than ϵ. (This discussion deals with a single iteration, and therefore assumes that ϵ is fixed.) For the remaining arcs, the upper bound is ∞. While the algorithm is allowed to report false negatives (i.e., it may find an upper bound of ∞ for arcs whose actual reach is less than ϵ), it must never report a false positive.

Fix an arc (v, w). To prove that $r(v, w) < \epsilon$, we must consider all shortest paths that contain (v, w). Fortunately, we do not have to evaluate all such paths explicitly: it suffices to process only *minimal paths*. Let $P_{st} = (s, s', \ldots, v, w, \ldots, t', t)$ be the shortest path between s and t, and assume that (v, w) has reach at least ϵ with respect to this path. Path P_{st} is *ϵ-minimal* with respect to (v, w) if and only if the reaches of (v, w) 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 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 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 $v = r$ or $d(x, v) < \epsilon$, where $d(x, v)$ denotes the distance (in the tree) between x and v. Note that, when v is not an inner vertex, no path P_{rt} starting at r will be ϵ-minimal with respect to a tree arc (u, v): if the reach of (u, v) is greater than ϵ with respect to P_{rt}, it will also be greater than ϵ with respect to P_{xt}.

The tree arcs whose heads are inner vertices are those whose reaches we will try to bound; we call them *inner arcs*. The partial tree T_r must be large enough to include all of them, as well as enough descendants to bound their height accurately. More precisely, we must make sure that every inner vertex v has one of two properties: (1) v has no labeled (unscanned) descendent; or (2) v has at least one scanned descendent whose distance from $p(v)$ (the parent of v in T_r) is ϵ or greater. When these conditions are satisfied, we do not need to grow the tree any further because all ϵ-minimal paths starting at r are already taken into account. In practice, however, this condition often leads to very large partial trees. As Section 4.2.1 will explain, we use a relaxed version of the second condition that only guarantees that the distance (in the partial tree) to every *labeled* vertex from its closest inner ancestor is greater than ϵ. When processing the partial tree, we consider both scanned and labeled vertices (which will be leaves) as belonging to it. This ensures that there are no false positives, but may generate false negatives.

Once the tree is built, processing it is straightforward. For each vertex v, we know its *depth*, i.e., the distance from the root to v. In $O(|T_r|)$ time, one can also compute the *height* of every inner vertex v, defined as the distance within the tree from v to its farthest descendent (either scanned or labeled). The reach of a tree arc (v, w) with respect to the partial tree is the minimum between the depth of w and the height of v. The reach bound for (v, w) with respect to the entire graph is the maximum over all such reaches, considering all partial trees that contain (v, w)

and have w as an inner vertex. If this maximum is at least ϵ, we declare the reach bound to be ∞.

As already observed, the notion of running several rounds of partial-tree computation to find reach upper bounds is due to Gutman [**24**]. Our algorithm improves on his in three important ways. First, we add shortcuts between two rounds of partial-trees computation, which causes the graph to shrink much faster (on road networks). Second, we compute arc reaches instead of vertex reaches, which decreases the degrees of high-reach vertices more quickly during preprocessing and allows them to be bypassed (which reduces their reach). Finally, we only grow partial shortest path trees from vertices that have not been eliminated yet (as the next subsection will explain). Gutman's algorithm, in contrast, also grows trees from eliminated vertices with "live" neighbors, which is significantly slower. Without these modifications, the preprocessing algorithm would be impractical for very large graphs, and query performance would be significantly worse.

4.2.1. *Dealing with penalties.* As described, the algorithm assumes that partial trees are 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 (because they were bypassed or their reach was bounded in a previous iteration), growing partial trees from the remaining vertices only. Eliminated vertices must be taken into account, however, since they may belong to the shortest paths that determine the reaches of the remaining vertices.

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. The intuition behind penalties is simple. Suppose that the reach of an arc (x, y) (still in the graph) is determined by the shortest path between s and t. Some arcs on this path may have already been eliminated from the graph, because they have small reach. But consider the largest subpath s'-t' of s-t that remains in the graph and contains (x, y): if we implicitly "extend" it on both ends (by adding *in-penalty*(s') to the prefix and *out-penalty*(t') to the suffix), the reach of (x, y) with respect to this subpath will be at least as large as the original reach.

To consider penalties when processing partial trees, it suffices to modify some of the definitions used by the procedure. The (redefined) depth of a vertex v within a tree T_r, denoted by $depth_r(v)$, 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. We implicitly attach a *pseudo-leaf* v' to each vertex v in T_r and set the length of the arc between v and v' to be the out-penalty of v. Heights are computed not with respect to T_r, but with respect to the pseudo-tree obtained when the pseudo-leaves are taken into account. As before, the reach of a tree arc (v, w) with respect to T_r is the minimum between the (modified) depth of w and the (modified) height of v.

Next we describe two simple modifications to the way partial trees are grown. They use penalty information to reduce the number of inner vertices in each tree, and with them the number of scanned vertices. This makes the algorithm faster without changing the reach bounds it obtains.

First, when deciding whether v is an inner vertex with respect to a root r, we compute $depth_r(v)$ taking in-penalties into account, i.e., as $d(r, v) + \textit{in-penalty}(r)$. To be considered an inner vertex, v must satisfy one of the following conditions:

(1) $v = r$ or (2) $p(v) = r$ or (3) $depth_x(v) < \epsilon$ (recall that x is the second vertex on the path from r to v). There is one exception to condition (3): if $depth_r(p(v)) < in\text{-}penalty(p(v))$, v will not be considered an inner vertex (because its modified depth will be even higher in the tree rooted at $p(v)$), and neither will its descendents. Note that, together, these definitions imply that the parent of an inner vertex must also be an inner vertex.

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 relevant. To decide whether an outer vertex is relevant, we keep track of the *extension* of each vertex w, denoted by $ext_r(w)$: if u is the last inner vertex on the path from r to w, $ext_r(w)$ is defined as $d(p(u), w)$. An outer vertex v is relevant if its parent is relevant and $ext_r(p(v)) + out\text{-}penalty(p(v)) \leq \epsilon$. This definition ensures that, when v is not relevant, every inner ancestor of v will have height greater than ϵ even if we stop growing the tree at $p(v)$.

Our implementation introduces a third modification: we relax the notion of relevance to allow the algorithm to stop sooner. To be relevant, in addition to the conditions above, a vertex v must be such that $ext_r(v) + out\text{-}penalty(v) \leq \rho\epsilon$, with $\rho \geq 1$ (we used $\rho = 1.1$ in our experiments). Even if we end up not scanning v because of this rule, the algorithm remains correct because, as a labeled vertex, v is still guaranteed to appear in the final partial tree. Unlike the first two modifications, this relaxed definition may lead to worse reach bounds, since the parent u of v in the partial tree when it stops growing may not be v's actual parent (which may remain unscanned). As a result, the reaches of u and its ancestors may appear to be artificially high; in particular, the iteration may end up assigning infinite bounds to vertices whose actual reaches are less than ϵ. In practice, we observed that the heuristic stopping criterion makes the algorithm significantly faster and has little effect on the quality of the bounds.

4.2.2. *Improvable arcs.* When an iteration of the reach algorithm starts, we assume that all the arcs that remain in the graph have reach estimate $\bar{r}(v, w) = 0$. We grow partial shortest path trees one at a time. For each inner arc (v, w) in a tree, we check if its reach with respect to the tree is greater than $\bar{r}(v, w)$. If so, we update $\bar{r}(v, w)$ to its reach value in the current tree (or to ∞, if the reach in the current tree is greater than ϵ).

The fact that the reach estimate can only increase as the algorithm progresses (within a single iteration) can be used to speed up the computation. Assume we have already grown a few partial shortest path trees and the current reach estimate of (v, w) is $\bar{r}(v, w)$. When growing a new partial shortest path tree T_r, suppose (v, w) is again an inner arc. Without any further processing, we know that the reach of (v, w) with respect to this tree will be no greater than $depth_r(w)$. If this value is smaller than (or equal to) $\bar{r}(v, w)$, we can safely say that T_r is irrelevant for computing the reach of (v, w). To formalize this notion, we say that a vertex v is *improvable* with respect to T_r if $depth_r(v) > \bar{r}(p(v), v)$.

We can now redefine the *extension* $ext_r(w)$ of a vertex w as $d(u, w)$, where u is the last inner vertex on the path from r to w that is improvable. Compared with the previous definition, the extension of a vertex can only be larger (or remain the same). As a result, fewer vertices will be considered relevant, thus allowing the search to stop sooner.

4.2.3. *Dealing with long arcs.* The partial-trees algorithm aims at analyzing ϵ-minimal paths. Typically, these paths have length close to 2ϵ, which is the usual depth of the trees examined. In fact, if all arc lengths are much smaller than ϵ, it is not hard to see that all partial trees will have comparable depth.

Some road networks, however, contain arcs that are significantly longer than average. An obvious example is an arc representing a ferry route. Partial shortest path trees that include these arcs tend to be much deeper. Since for road networks the total number of vertices in a tree is roughly quadratic in the depth, this can significantly slow down the preprocessing procedure.

To speed up the algorithm, we used a hard bound on the total depth of the tree. We set it to 4ϵ, which is large enough so that most trees are unaffected, but small enough to prune exceptional cases significantly. Unfortunately, we cannot simply stop growing the tree when it reaches this value and process the result. Because the standard stopping criteria will not be observed, this might lead to false positives (i.e., some reaches will be underestimated).

Consider what happens, for instance, when we grow a partial tree from a root r, and let (r, v) be a very long arc (e.g., $\ell(r, v) > 4\epsilon$) incident to r. Suppose that the shortest path from r to some vertex x consists of the arcs (r, v), (v, w), and (w, x), with $\ell(v, w) < \epsilon$ and $\ell(w, x) \gg \epsilon$. Clearly, $r(v, w)$ is greater than ϵ. An algorithm that simply stops growing partial trees when their depth reaches 4ϵ will not detect this path, however. When growing the tree from r, it would stop after its depth reached 4ϵ. At this point, v will be labeled (with parent r), but not scanned, and vertex w may not have been visited at all. When growing a tree from v, we would only see the path (v, w, x), where the reach of (v, w) is smaller than ϵ.

To avoid this situation, we must use a modified notion of in-penalty when growing partial trees. For every vertex v, we determine the length $\ell(u, v)$ of its longest remaining incoming arc. If it is ϵ or larger, we set $\textit{in-penalty}'(v) \leftarrow \max\{\textit{in-penalty}(v), \ell(u, v)\}$. Intuitively, whenever there is a long arc incident to v, we ensure that the in-penalty of v is large enough to "catch" all necessary ϵ-minimal paths. In the example above, even though we would abort the search too soon when growing a tree from r, we would still determine that (v, w) has high reach when growing the tree from v itself.

Of course, the downside of this approach is that it may lead to more false negatives, since in some cases arc (u, v) will not be on the shortest paths between u and the inner vertices of the partial tree rooted at v. Very long arcs are relatively rare, however, and the considerable speed-up allowed by this technique makes it worthwhile in practice.

4.2.4. *Converting arc reaches to vertex reaches.* Before the refinement step, we convert the upper bounds on arc reaches into upper bounds on vertex reaches. Consider a vertex v, and let P be the shortest path that determines $r(v)$. Assume that $r(v) > 0$, i.e., that v is not an endpoint of P. Let (u, v) and (v, w) be the arcs of P entering and leaving v. Clearly, the reaches of these arcs with respect to P are at least $r(v)$; conversely, $r(v) \leq \min\{r(u, v), r(v, w)\}$.

Unfortunately, we do not know which neighbors of v are the ones that determine the reach (i.e., which ones are u and w). But it is easy to verify that $r(v) \leq \min\{\max_x\{r(x, v)\}, \max_y\{r(v, y)\}\}$. In other words, a valid upper bound for $r(v)$ is the minimum over the highest incoming arc reach and the highest outgoing arc reach.

Often, however, both maxima are achieved at the same neighbor $x = y$. Although the upper bound in this case is still valid, it may be much higher than necessary, since we know that $u \neq w$ on the path that determines the highest reach. We can exclude this case as follows:

(1) Find x', the incoming neighbor of v that maximizes $r(x', v)$, then find y', the outgoing neighbor of v that is *different from* x' and maximizes $r(v, y')$. Let δ' be the minimum of $r(x', v)$ and $r(v, y')$.
(2) Find the outgoing neighbor y'' of v such that $r(v, y'')$ is maximized, and the incoming neighbor x'' of v that is different from y'' and maximizes $r(x'', v)$. Let δ'' be the minimum of $r(x'', v)$ and $r(v, y'')$.

Note that δ' and δ'' may be different from each other in a directed graph. A valid upper bound on $r(v)$ is the maximum of δ' and δ''. As a special case, if v has only one neighbor (even if it is both incoming and outgoing), the reach of v will be zero.

4.3. Adding Shortcuts. To bypass a vertex v, we first examine all pairs $[(u, v), (v, w)]$ of incoming/outgoing arcs 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. Bypassing high-degree vertices, however, 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 (we follow the notation proposed by Sanders and Schultes [**41**]). Higher values of c will cause the graph to shrink faster during preprocessing, but may increase the final number of arcs substantially. For road networks, we used 0.5 for the first level, 1.0 for the second, and 1.5 for 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. For grids, which do not have a natural hierarchy, we used higher values of c from the beginning.

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 (when v is about to 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 iteration i). As explained in Section 4.2.3, long arcs and large penalties can decrease the quality of the reach upper bounds provided by the preprocessing algorithm. Imposing these additional bypassability criteria prevents 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 priority queue to efficiently determine which vertex to bypass next.

4.3.1. *Computing reaches of deleted arcs.* As already mentioned, when a vertex v is bypassed, it is deleted from the graph alongside all arcs currently incident to it. At this moment, we must find a valid upper bound on the reaches of these deleted arcs. Consider an incoming arc (u, v). We know that any shortest path P containing this arc will *not* continue beyond v using one of the existing outgoing arcs (since we break ties by preferring shortcuts, such path would contain one of the newly inserted shortcuts instead). Therefore, P will either stop at v or proceed through a previously deleted arc. In the latter case, *out-penalty*(v) bounds the length of the suffix of P that starts from v. It follows that we can safely set the upper bound $\bar{r}(u, v)$ on the reach of (u, v) to $\ell(u, v) +$ *out-penalty*(v). The same argument applies to an outgoing arc (v, w): we set $\bar{r}(v, w) \leftarrow \ell(v, w) +$ *in-penalty*(v).

The penalties associated with the neighbors of v must also be updated to take the reaches of the newly eliminated arcs into account.

4.4. Exact Reaches: Refinement Step. The refinement step computes exact vertex reaches on the subgraph induced by the δ vertices with highest reach, where δ is a user-defined parameter. Vertices and arcs not in this induced subgraph are considered implicitly, as penalties associated with every remaining vertex in the graph. To simplify notation, in this section we denote by n the number of vertices of the induced subgraph in which we compute exact reaches.

As already mentioned, exact reaches can be computed by growing shortest path trees from each of the n vertices in the graph, then computing the depth and height of each vertex within these trees. We developed an algorithm that has the same worst-case complexity, but can be significantly faster in practice on road networks. Even though it is still prohibitively expensive for large road networks, it is very useful when applied in the refinement step on a much smaller subgraph.

Our method follows the same principle as the basic algorithm: 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 previously found subtrees.

The algorithm works as follows. First, it partitions 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 $w \notin A$. The remainder of the region consists of *internal 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 4.1. *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 some vertex 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 v 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 on this path, which contradicts our initial assumption that the parent of v in all trees rooted at vertices of $f(A)$ is $p_{f(A)}(v) \neq p_r(v)$. □

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 holds, 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 a shortest path tree from any internal vertex of A as well.

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 *truncated trees* from every internal vertex of A (i.e., every vertex in $A \setminus f(A)$). These truncated 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 $p(w)$ will be the same (tainted) vertex in every shortest path tree rooted at A. Therefore, w imposes an *implicit penalty* on $p(w)$ equal to w's own height plus the length of the arc $(p(w), w)$. 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 internal 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, but 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 truncated shortest path trees, we remember the maximum depth of each tainted vertex, we can later compute (in linear time) the maximum depth of all untainted vertices. Since their heights are known, their reaches (with respect to the trees rooted at A) can be easily determined.

4.4.1. *Regions.* The algorithm above is correct regardless of how the regions are chosen. In particular, if each region has exactly one vertex, we have 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, it also increases 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. If there are unreachable vertices, they 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 compute regions quickly with relatively smaller frontiers.

4.5. Other Details. As already mentioned, the reach threshold grows exponentially as the preprocessing algorithm progresses: we set $\epsilon_{i+1} \leftarrow 3\epsilon_i$ for each round i. It remains to determine ϵ_1, the threshold during the first round. Ideally, our choice should be such that the first iteration takes roughly as much time as each of the remaining iterations. A large value will make the first iteration comparatively slow and will not give the algorithm the chance to add shortcuts when needed. In contrast, a very small value will introduce penalties too early, which will decrease the accuracy of the reach upper bounds computed in subsequent iterations.

The value of ϵ_1 depends on an integer input parameter k_1 (we used $k_1 = 1000$ in our experiments). We pick $\lfloor n/k_1 \rfloor$ root vertices at random. For each root, we grow a partial shortest path tree with exactly k_1 vertices and take note of its radius (the distance label of the last scanned vertex). We set ϵ_1 to be half the minimum among all such radii. This ensures that, during the first iteration of the preprocessing algorithm, not many partial shortest path trees have more than k_1 vertices. Our choice of k_1 is fairly robust for road networks: small changes do not have much effect on the performance of either preprocessing or queries.

4.6. Correctness. We argued that the transformation of arc reaches into vertex reaches and the refinement step are correct. For completeness, we now present a (somewhat tedious but straightforward) proof that the main stage of our algorithm, which grows partial trees and adds shortcuts to the graph, is correct: it finds valid upper bounds on reaches. We focus on the basic version of the partial-trees algorithm, which takes penalties into account when processing the tree but not when growing it. The three improvements described in Section 4.2.1 are not taken into account, and neither are the special stopping criteria introduced in Section 4.2.3 to handle very long arcs. We have already argued why these accelerations do not affect correctness.

Before we proceed to the proof, we must explain in more detail an important element of the partial-trees algorithm. Recall that we assume that ties are broken so that a shortcut is always preferred to the arcs it replaces; furthermore, we assume that all original shortest paths are unique. We deal with these issues by working with *canonical paths*. A canonical path is a shortest path with additional properties,

including uniqueness. We require the following properties, which are sufficient and easy to work with, but may not be necessary.

(1) A canonical path is a simple shortest path.
(2) For every pair (s, t), there is a unique canonical path between s and t.
(3) A subpath of a canonical path is a canonical path.
(4) There is an implementation of Dijkstra's algorithm that always finds canonical paths.
(5) (Non-shortcut property.) A path Q is *not* a canonical path if Q contains a subpath P with more than one arc such that the graph contains a shortcut arc for P.

Regarding Property 1, note that, if the graph has no cycles of zero-length arcs, all shortest paths are simple. Property 5 ensures that adding shortcut arcs decreases vertex reaches.

We implement canonical paths as follows. For each original arc a, we generate a length perturbation $\ell'(a)$. When computing the length of a path, we separately sum lengths and perturbations along the path, and use the perturbation to break ties in path lengths.

First suppose there are no shortcut arcs. If the perturbations are chosen uniformly at random from a big enough range of integers, with high probability all shortest paths (with respect to length and perturbations) are canonical paths. In our implementation, perturbations were picked uniformly at random from the range $[1, 65535]$. Shortcut arcs are added after the perturbations are introduced. The length and the perturbation of a shortcut arc are equal to the sum of the corresponding values for the arcs on the path that we shortcut. To break ties in a graph with shortcuts, we use the number of hops of the paths (fewer hops are better) after considering perturbations. Note that Dijkstra's algorithm can easily maintain the number of hops of candidate paths. It is not hard to see that the shortest paths that win the tie breaking are canonical.

Our way of dealing with canonical paths is conceptually simple but has two disadvantages: the memory overhead for storing perturbations during queries (which is minor) and a small probability of failure due to ties in sums of perturbations. Regarding the latter issue, we have never observed the algorithm working incorrectly during our extensive experiments. In general, our method can use any tie breaking rule that gives preference to paths with shortcuts and for which a (suitably modified) Dijkstra's algorithm finds the corresponding canonical paths. It is possible that other tie-breaking approaches, such as that of [**9**], will work. However, this is not completely obvious.

Our preprocessing algorithm computes upper bounds on reaches with respect to the set of canonical paths as defined above using tie-breaking by perturbations and hops. During queries, however, we can completely ignore both tie-breaking rules, and simply prune by reach. If we only prune a vertex v if $r(v) < \text{dist}(s, v)$ and $r(v) < \text{dist}(v, t)$ (as our algorithms do), the canonical path will not be pruned. The algorithm may not necessarily return the canonical path as its answer, but it is guaranteed to find a path of the same length.

Once we have the notion of canonical shortest paths, we can prove that the upper bounds on arc reaches computed during the first stage of the preprocessing algorithm are indeed valid. The proof is by a straightforward induction with a somewhat tedious case analysis.

THEOREM 4.2. *For every arc (v, w) we compute an upper bound $\overline{r}(v, w)$ on the reach of (v, w) in the graph obtained from the original graph G by adding all shortcuts.*

PROOF. We break the process into its elementary steps, where an elementary step is either bypassing a vertex or a round of partial tree computations (after which we delete every arc whose reach is finitely bounded). Let G_i be the original graph with all shortcuts added up to the end of step i, and let G'_i be the graph that the algorithm is left with after step i. We prove by induction on the steps that the following invariants hold.

(1) For every arc (v, w) that we have deleted until (and including) step i (i.e., $(v, w) \notin G'_i$), $\overline{r}(v, w)$ upper bounds the reach of (v, w) in G_i.
(2) For every vertex $v \in G'_i$, *out-penalty*(v) upper bounds the reach in G_i of any arc (v, w) that we have already deleted.
(3) For every vertex $v \in G'_i$, *in-penalty*(v) upper bounds the reach in G_i of any arc (u, v) that we have already deleted.

Note that Invariants (2) and (3) immediately follow from Invariant (1) because of the way we update the penalties.

We outline a proof of the induction step. The basis is trivial. Assuming these invariants are true at the end of step $i - 1$, we must prove that they also hold at the end of step i.

Since adding shortcuts can only reduce reaches, Invariant (1) continues to hold for all arcs deleted prior to step i. We will establish Invariant (1) for every arc that we delete during step i.

Assume step i is bypassing a vertex v. Let (v, w) (or (w, v)) be an arc that we delete when bypassing v. Then, from the correctness of the penalties after step $i - 1$ and the discussion in Section 4.3, it follows that $\overline{r}(v, w)$ indeed bounds the reach of (v, w) in the graph G_i, which includes also all shortcut arcs added when we bypassed v.

Assume step i is a partial tree computation. Let ϵ be the threshold of the current level, and let (v, w) be an arc whose reach we bound at this step ($\overline{r}(v, w) < \epsilon$). Assume (to get a contradiction) that the reach of (v, w) in G_i is $r(v, w) > \overline{r}(v, w)$. Note that the partial tree computation is done in the graph G'_{i-1} and, since we do not bypass vertices in this step, $G_i = G_{i-1}$.

Assume first that $r(v, w) \geq \epsilon$, and let $P = (s, \ldots, v, w, \ldots, t)$ be an ϵ-minimal path with respect to (v, w) in G_i. For a vertex $x \in P$, let $p(x)$ be the vertex preceding x on P and let $f(x)$ be the vertex following x on P. Let $P' = (z, \ldots, v, w, \ldots, y)$ be the maximal subpath of P in G'_{i-1} containing (v, w). Since if $z \neq s$ the reach of the arc $(p(z), z)$ in G_{i-1} is at least $\min\{\text{dist}(s, z), \text{dist}(p(z), t)\}$, it follows from the correctness of the penalties after step $i - 1$ that *in-penalty*$(z) + \text{dist}(z, w) \geq \epsilon$. Let z' be the last vertex on the prefix of P' up to and including v such that *in-penalty*$(z') + \text{dist}(z', w) \geq \epsilon$. In the partial tree rooted at z', w is inner. (Note that this is true also if $z' = v$.)

If $y \neq t$ then the reach of $(y, f(y))$ is at least $\min\{\text{dist}(y, t), \text{dist}(s, f(y))\}$. So (regardless of whether y is t or not) by the correctness of the penalties after step $t - 1$ we get that $\text{dist}(v, y) + \textit{out-penalty}(y) \geq \epsilon$. Let y' be the last vertex after v on P' such that $\text{dist}(v, y') + \textit{out-penalty}(y') < \epsilon$ (or v if there is no such vertex). Then y' must be scanned, and $f(y')$ must be a labeled child of y', during the shortest path computation from z'. Therefore, in the partial tree rooted at z' the depth of

w and the height of v are both at least ϵ. So in this tree the arc (v, w) should have gotten a bound of ∞ on its reach, which is a contradiction.

If $r(v, w) < \epsilon$, let $P = (s, \ldots, v, w, \ldots, t)$ be an $r(v, w)$-minimal path. A similar argument shows that there is a partial tree in which the bound on the reach of (v, w) is at least $r(v, w)$, which gives a contradiction. □

4.7. Cardinality Reach and Highway Hierarchies. Having fully described our reach-based algorithm (RE), we now discuss its relationship to the HH algorithm of Sanders and Schultes [**40, 41**]. We show similarities between the algorithms other than those pointed out in [**40, 41**].

We introduce a variant of reach that we call *c-reach* (cardinality reach). Given a vertex v on a shortest path P, grow equal-cardinality balls centered at the endpoints of P until v belongs to one of the balls. Let $c_P(v)$ be the cardinality of each of the balls at this point. The c-reach of v, $c(v)$, is the maximum, over all shortest paths P, of $c_P(v)$. Note that if we replace cardinality with radius, we get the definition of reach. To use c-reach for pruning the search, we need the following values. For a vertex v and a nonnegative integer i, let $\rho(v, i)$ be the radius of the smallest ball centered at v that contains i vertices. Consider a search for the shortest path from s to t and a vertex v. We do not need to scan v if $\rho(s, c(v)) < \text{dist}(s, v)$ and $\rho(t, c(v)) < \text{dist}(v, t)$. A direct implementation of this pruning method would require maintaining $n - 1$ values of ρ for every vertex.

The main idea behind HH preprocessing is to use the partial-trees algorithm for c-reaches instead of reaches. Given a threshold h, the algorithm identifies vertices that have c-reach below h (local vertices). Consider a bidirectional search. During the search from s, once the search radius advances past $\rho(s, h)$, one can prune local vertices in this search. One can do similar pruning for the reverse search. This idea is applied recursively to the graph with low c-reach vertices deleted. This gives a hierarchy of vertices, in which each vertex needs to store a ρ-value for each level of the hierarchy it is present at. The preprocessing phase of HH also adds shortcuts and uses other heuristics to reduce the graph size at each iteration.

An important property of the HH query algorithm, which makes it similar to the self-bounding algorithm discussed in Section 3, is that the search in a given direction never goes to a lower level of the hierarchy. Our self-bounding algorithm can be seen as having a "continuous hierarchy" of reaches: once a search leaves a reach level, it never comes back to it.

5. Reaches and A* Search

This section reviews A* search and the ALT algorithm, and shows how the latter can be combined with reach pruning in a natural way.

5.1. A* Search. Suppose we need to find shortest paths in 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* [**12, 25**] 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 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*, since $\pi_f(v)$ is guaranteed to be a lower bound on the distance from any vertex v to t.

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 the length 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 forward graph is equal to $\ell_{\pi_r}(w, v)$ in the reverse graph. This is equivalent to $\pi_f + \pi_r = \text{constant}$.

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 are feasible (but not necessarily consistent) and give lower bounds to the sink and from the source, respectively. Ikeda et al. [**26**] suggest using an *average function*, 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. To make the algorithm more intuitive, we add $\pi_r(t)/2$ to the forward function (which ensures that $p_f(t) = 0$) and $\pi_f(s)/2$ to the reverse function (making it zero at s). Because the added terms are constant, the functions remain consistent. Although p_f and p_r usually do not give lower bounds as good as the original ones, they are feasible and consistent.

Recall that the standard bidirectional algorithm can stop as soon as $\text{top}_f + \text{top}_r \geq \mu$, where top_f is the top key in the forward heap, top_r is the top key in the reverse heap, and μ is the length of the shortest path found so far. For bidirectional A* with consistent lower bounds, we can use a similar stopping criterion, but in the transformed graph.

Let v_f and v_r be the top vertices in each heap (with keys top_f and top_r, respectively). The standard stopping criterion states that we can stop as soon as $\text{dist}(s, v_f) + \text{dist}(v_r, t) \geq$ [best path seen so far]. In the transformed graph, the distance between s and v_f corresponds to $d_f(v_f) + p_f(v_f) - p_f(s) = k_f(v_f) - p_f(s) = \text{top}_f - p_f(s)$. Similarly, the distance between v_r and t in the transformed graph is $d_r(v_r) + p_r(v_r) - p_r(t) = k_r(v_r) - p_r(t) = \text{top}_r - p_r(t)$. Finally, the length of the shortest path seen so far is $\mu - p_f(s) + p_f(t)$. The stopping criterion then becomes:

$$[\text{top}_f - p_f(s)] + [\text{top}_r - p_r(t)] \geq \mu - p_f(s) + p_f(t).$$

Since we fixed $p_f(t) = 0$, this translates into

$$\text{top}_f + \text{top}_r \geq \mu + p_r(t).$$

5.2. The ALT Algorithm. The ALT algorithm [**19, 23**] is an A*-based method that uses landmarks and triangle inequality to compute feasible lower bounds. We select a small subset of vertices (a constant number) as *landmarks* and, for each vertex in the graph, precompute distances to and from every landmark. Consider

a landmark L. If we are using distances *to* L, then, by the triangle inequality, $\text{dist}(v, L) - \text{dist}(w, L) \leq \text{dist}(v, w)$; if we use distances *from* L, then $\text{dist}(L, w) - \text{dist}(L, v) \leq \text{dist}(v, w)$. To get the tightest lower bound, one can take the maximum of these bounds, over all *active* landmarks (i.e., those in use for the current search). These bounds are feasible, and we use the average function to ensure consistency. 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 overall performance than other variants [**21**].

5.2.1. *Active Landmarks.* Each s-t query starts with just two active landmarks, those that give the best lower bounds on the distance from s to t (one using distances *to* the landmark, the other using distances *from* the landmark). Periodically, the algorithm checks whether adding another landmark to the set of active ones would be advantageous. We call this a *checkpoint*.

Of course, checkpoints are expensive, so we avoid running them too often. We allow each search (forward and reverse) to have at most ten checkpoints. In addition, we require that at least $8k$ vertices be scanned between checkpoints in the same direction, where k is the total number landmarks available. For details, see [**23**].

5.2.2. *Landmark Generation.* In our experiments, we use the *avoid* algorithm to select landmarks. Among the landmark selection algorithms studied in [**23**], *avoid* is the second best in terms of solution quality. It is only surpassed by *max-cover*, which is essentially *avoid* followed by a simple 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. More precisely, r is picked with probability proportional to the square of its distance to the closest existing landmark (if there is no landmark yet, r is picked uniformly at random). The algorithm then assigns a weight to each vertex v, defined as the difference between the distance from r to v and the lower bound on that distance given by the existing landmarks (if there are no landmarks yet, the weight is constant). Therefore, vertices with weak lower bounds will have larger 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 (we call this sum the *size* of p). A leaf of the subtree of T_r rooted at p is then selected as the new landmark. More precisely, we follow a path from p to a leaf by picking in each step a child with maximum size.

Computing the lower bound on the distance from the root 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 (as in most of our experiments), 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. The set of relevant vertices is picked uniformly at random (unless reaches are available, as Section 5.4 will explain). For the range of values of k we tested (up to 64), the landmarks produced by this method were not measurably worse than those obtained when n vertices are considered relevant, as in [**21**].

As observed in [**23**], we create a separate file for each landmark. For each vertex (in order), it contains the distances *to* and *from* the landmark stored contiguously. To save space, we use a simple compression scheme that exploits the fact that, on road networks, distances to and from the same landmark are usually very similar, and that vertices with similar identifiers are usually close together in the graph. Although each (uncompressed) distance could be stored as a 32-bit value, there are long runs in which the first 16 bits in each distance are identical. These 16 bits are represented only once, together with an 8-bit count on the number of times it is repeated (runs with more than 255 elements are split); each distance in the run is then represented by its 16 least significant bits. This approach, which is a run-length encoding scheme,[3] leads to compression ratios that are close to 50%. We observe that compression is used only in disk. Once in main memory, landmark distances are kept uncompressed as 32-bit integers. We still keep a separate array for each landmark, with "from" and "to" distances stored contiguously.

5.3. Combining Reaches and Landmarks. Reach-based pruning can be easily combined with A* search: the basic idea is to just run A* search and prune vertices (or arcs) based on reach conditions. Specifically, when A* search is about to scan a vertex v, we extract from the key of v the length of the shortest path from the source to 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.

Note that we must actually use landmark-based lower bounds for pruning. Implicit bounds cannot be used with A* search because the search grows balls with respect to reduced costs, which have little correlation with the original lengths.

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. 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. Pruning by reach does not affect the stopping condition of the algorithm: the usual condition for A* search can still be applied. 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 and reverse searches.

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, queries are still independent of the preprocessing algorithm: they only take as input the graph

[3]See e.g., http://www.data-compression.info/Algorithms/RLE/.

with shortcuts, the reach values, and the distances to and from landmarks. How this data was obtained is immaterial. We will see in Section 5.4, however, that it might be useful to take reaches into account when generating landmarks.

5.3.1. *Optimizations.* Our implementation of REAL actually uses early pruning, as in RE. When scanning arc (v, w) in the forward direction, we prune w if $\overline{r}(w) < \min\{d_f(v) + \ell(v, w), \pi_f(w)\}$. (We restrict our discussion to the forward search; the reverse one is similar.) Note that this computation requires knowledge of $\pi_f(w)$, a lower bound on the distance from w to the target. Computing it requires retrieving from memory the distances between w and all active landmarks, and computing the triangle inequality in each case. Although this takes constant time, it is relatively expensive in practice. Before actually computing it, we use $\pi_f(v) - \ell(v, w)$ as a lower bound on the distance from w to t; since v is the vertex being scanned, $\pi_f(v)$ is readily available. If (1) $\overline{r}(w) < d_f(v) + \ell(v, w)$ and (2) $\overline{r}(w) < \pi_f(v) - \ell(v, w)$, we can prune w. Only if condition (2) fails and condition (1) succeeds do we need to compute $\pi_f(w)$.

Also as in RE, we can use arc sorting to prune some arcs implicitly. We sort the arcs (v, w) in the adjacency list of v by $\overline{r}(w) + \ell(v, w)$ (in non-increasing order). Suppose that, while scanning v, we find an arc (v, w) such that (1′) $\overline{r}(w) + \ell(v, w) < d_f(v)$ and (2′) $\overline{r}(w) + \ell(v, w) < \pi_f(v)$. This implies that conditions (1) and (2) above are true for w and therefore (v, w)—and all arcs that succeed it—can be pruned. Note that conditions (2) and (2′) are equivalent, but condition (1) may succeed while (1′) fails. In this case we still prune the arc, but keep traversing the adjacency list.

We also try to prune a vertex after we remove it from the heap (and before we scan it). This is still useful because the lower bound on the distance to the target may have improved since the vertex was inserted into the heap, due to the activation of new landmarks.

5.4. Reach-Aware Landmarks. Although landmark generation and reach computation can be completely independent, we can use landmarks more efficiently when reaches are available. 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 (still with pruning by reach) by removing all labeled vertices from the heaps and reinserting 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

therefore 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.[4] 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

$$\text{dist}(s, v) \geq \text{dist}(s', L) - \text{dist}(v, L) - \text{dist}(s', s). \tag{5.1}$$

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

$$\text{dist}(s, v) \geq \text{dist}(L, v) - \text{dist}(L, s') - \text{dist}(s', s). \tag{5.2}$$

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

$$\text{dist}(v, t) \geq \text{dist}(L, t') - \text{dist}(L, v) - \text{dist}(t, t'). \tag{5.3}$$

With distances *to* landmarks, the appropriate expression is

$$\text{dist}(v, t) \geq \text{dist}(v, L) - \text{dist}(t', L) - \text{dist}(t, t'). \tag{5.4}$$

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, we can find parameter values for which both memory use and query times decrease.

We note that, in our experiments, we do not pick the reach threshold R explicitly. Instead, we actually pick how many vertex distances we want to store per landmark; the value of R will be fully determined by this choice.

[4]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.

5.4.1. *Landmark Generation.* As observed in Section 5.2.2, we use a fast version of the *avoid* method to generate landmarks. We also use it to generate partial landmarks, with a small modification. Instead of picking the set of relevant vertices uniformly at random, we pick the first n/k vertices in the vertex list. Because vertices are approximately sorted in decreasing order of reach (as seen in Section 3.2), only vertices with high reach are taken into account, which greatly simplifies the implementation when only partial landmark data is stored. In terms of solution quality, these two approaches are roughly equivalent.

6. Experimental Results

We implemented our algorithms in C++ and compiled them with Microsoft Visual C++ 2005. All tests were performed on a dual-processor, 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 processor at a time (but is not pinned to a particular processor).

Our implementation uses two kinds of priority queues. We use 4-heaps when the number of elements in a priority queue is small, such as during queries and when building partial trees during preprocessing. For single-source shortest path computations on the entire graph, we use multi-level buckets [**17, 22**]. Multi-level buckes are faster in general, but 4-heaps are competitive when the number of elements is small, and have a smaller memory overhead.

One of the main goals of our experimental analysis is to measure the performance of our algorithm on the road networks of the USA and Europe, made available for the 9th DIMACS Implementation Challenge. The experiments, reported in Section 6.1, include both random and local queries. We also investigated how much various ingredients of our algorithms contribute to the overall performance. In particular, we study the preprocessing/query and the time/space trade-offs.

Section 6.2 considers how our algorithms behave on other graph classes. In particular, we present results on grid graphs in 2 and 3 dimensions with random arc weights. We also conducted preliminary experiments on higher-dimensional grids and random graphs. Since our algorithm performed poorly on these graphs, we omit the detailed results.

We ran the ALT (with 16 landmarks), RE and REAL-(i, j) algorithms, where REAL-(i, j) uses i landmarks and maintains landmark data for n/j highest-reach vertices. We call the parameter j the *sparsity* of the landmarks. For instance, REAL-(64,16) maintains 64 landmarks, but with distances only to $n/16$ vertices with high reaches; REAL-(16,1) uses 16 landmarks, each with distances to all vertices in the graph. All landmarks (for ALT and all variants of REAL) were generated with the *avoid* method. 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 the graphs of the USA and Europe with travel times as arc lengths, additional data from previous works was available at the time of writing. In particular, we report the results obtained by the highway hierarchy-based algorithm of Sanders and Schultes, as reported in [**41**]. Their sequential code was run on a dual-core 2.0

GHz AMD Opteron machine, which is comparable to our machine. In fact, because of a different memory architecture, D runs about 2% faster on their machine. Their algorithm has two versions [**41**]: 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.

6.1. Road Networks. The graphs representing the USA (Tiger/Line) [**49**] and (Western) Europe [**35**] road networks belong to the 9th Implementation DIMACS Challenge [**8**] data set. The USA is symmetric and has 23 947 347 vertices (road intersections) and 58 333 444 arcs (directed road segments); Europe is directed, with 18 010 173 vertices and 42 560 279 arcs. To evaluate the performance on smaller graphs, we tested the subgraphs of USA described in Table 1. All graphs are strongly connected. We used two natural metrics as length functions: travel times and travel distances. In addition, to test the robustness of our algorithm, we considered a third metric (uniform) in which all arcs have unit length. For all metrics, lengths were represented as 32-bit integers.

TABLE 1. USA road networks from the TIGER/Line collection.

NAME	DESCRIPTION	VERTICES	ARCS	LAT. (N)	LONG. (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]

6.1.1. *Random queries.* Our first experiment consists of running our algorithms on 1000 random queries. In each case, the source and the target are picked independently and uniformly at random from the set of all available vertices. Table 2 presents results for the USA graph, and Table 3 shows the corresponding results for the European road network. In each case, all three metrics (length functions) are considered.

For each algorithm, 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 disk space (in megabytes) required by the preprocessed data. For D, this is the space required to store only the graph itself; for ALT, this includes the graph and landmark data; for RE, it includes the graph with shortcuts (which has roughly two-thirds more arcs) and an array of vertex reaches; finally, the data for REAL includes the graph with shortcuts, the array of reaches, and landmark data. All files are stored in binary format, with 32-bit lengths, distances, and vertex identifiers.

TABLE 2. Random queries on the USA graph.

METRIC	METHOD	PR. TIME (min)	DISK SPACE (MB)	QUERY AVG SC.	QUERY MAX SC.	QUERY TIME (ms)
TIMES	ALT	17.6	2563	187968	2183718	295.44
	RE	27.9	893	2405	4813	1.77
	REAL-(16,1)	45.5	3032	592	2668	0.80
	REAL-(64,16)	113.9	1579	538	2534	0.86
	HH	18	1686	1076	—	0.88
	HH-mem	65	919	2217	—	1.60
	D	—	536	11808864	—	5440.49
DISTANCES	ALT	15.2	2417	276195	2910133	410.73
	RE	46.4	918	7311	13886	5.78
	REAL-(16,1)	61.5	2923	905	5510	1.41
	REAL-(64,16)	120.5	1575	670	3499	1.22
	D	—	536	11782104	—	4576.02
UNIFORM	ALT	14.0	1992	240801	3922923	312.97
	RE	28.8	865	3496	6830	2.58
	REAL-(16,1)	42.7	2321	790	2968	1.09
	REAL-(64,16)	96.6	1315	573	3067	0.95
	D	—	536	11724870	—	3636.92

TABLE 3. Random queries on Europe.

METRIC	METHOD	PR. TIME (min)	DISK SPACE (MB)	QUERY AVG SC.	QUERY MAX SC.	QUERY TIME (ms)
TIMES	ALT	12.5	1597	82348	993015	120.09
	RE	45.1	648	4371	8486	3.06
	REAL-(16,1)	57.7	1869	714	3387	0.89
	REAL-(64,16)	102.6	1037	610	2998	0.91
	HH	15	1570	884	—	0.8
	HH-mem	55	692	1976	—	1.4
	D	—	393	8984289	—	4365.81
DISTANCES	ALT	9.6	1622	240750	3306755	321.11
	RE	31.5	681	7259	13059	5.20
	REAL-(16,1)	41.1	1938	855	4867	1.20
	REAL-(64,16)	76.0	1084	562	2596	0.91
	D	—	393	8991955	—	2934.24
UNIFORM	ALT	10.6	1488	140291	2137518	188.84
	RE	37.5	625	4109	10574	2.84
	REAL-(16,1)	48.1	1720	1048	5888	1.28
	REAL-(64,16)	88.7	964	644	3631	0.94
	D	—	393	9054599	—	3000.50

Note that for ALT, RE and REAL, query performance is worst when lengths are travel distances, but not drastically so. This can be explained by the fact that the natural hierarchy of road networks, which is exploited by both ALT and RE, is more pronounced with travel times (since highways usually have higher speed limits). With travel distances, a local road running alongside a major freeway may seem more attractive. Even with unit lengths the hierarchy is more pronounced than with

travel distances, as the number of vertices scanned by RE shows. Preprocessing, however, can be slower—probably because the partial trees become too dense with our default choice of parameters. We made no attempt to tune the algorithm for unit lengths.

Although the two graphs have similar sizes, the algorithms behave differently when lengths are travel times. ALT is relatively more efficient on Europe than on USA, while RE is faster on the USA graph. By combining ALT and RE, REAL is more robust than either method, and has similar performance on both graphs. In fact, even if we consider all six combinations of graphs and metrics, REAL-(64,16) was remarkably consistent: the average number of scans ranged from 538 to 670.

For a fixed graph, REAL-(16,1) and REAL-(64,16) have almost identical query performance, the latter being slightly better on most metrics. Given that REAL-(64,16) requires about half as much disk space, it has the edge for these queries. However, preprocessing for REAL-(64,16) takes roughly twice as long. As already observed, RE is less robust than REAL: the query times of RE on USA, for example, vary from 1.77 ms for travel times to 5.78 ms for travel distances, while the corresponding numbers for REAL-(64,16) are 0.86 and 1.22 ms.

We can compare RE and REAL to HH-mem and HH for the travel-time metric, for which the data is available. RE performs worse than HH-mem, but the difference is small, particularly on the USA graph. For queries, the performance of REAL-(64,16) is similar to that of HH, and REAL-(64,16) uses a little less space. Preprocessing for HH, however, is faster, especially on Europe.

6.1.2. *Graph size dependence.* To test how the performance of our algorithms depends on graph size, we ran 1000 random queries on subgraphs of the USA road network. The results are reported in Tables 4 and 5. Although query times tend to increase as the graph grows, they are not strictly monotone: the graph structure affects the performance of the algorithm. It is clear, however, 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 for travel times, and more for travel distances. The speed advantage of REAL-(64,16) relative to REAL-(16,1) holds only for large graphs: for smaller ones, the loss of precision due to proxies (defined in Section 5.4) is relatively higher, since the average shortest path is shorter.

Regarding preprocessing time, with a fixed number of landmarks ALT is roughly linear in the graph size. For 16 landmarks, the preprocessing of ALT is faster than the preprocessing of RE, and the ratio of the two does not change much 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) as for RE.

6.1.3. *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 [**40**], but it has a similar purpose. For each of the large graphs (Europe and

TABLE 4. Data for subgraphs of USA with travel times.

		PREP. TIME	DISK SPACE	QUERY		
GRAPH	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
NY	ALT	0.1	25	2735	23739	1.73
	RE	0.6	11	1104	2634	0.56
	REAL-(16,1)	0.7	30	226	1601	0.22
	REAL-(64,16)	1.1	17	383	1099	0.38
BAY	ALT	0.1	31	3251	31953	2.00
	RE	0.3	12	770	2114	0.42
	REAL-(16,1)	0.4	37	183	921	0.20
	REAL-(64,16)	0.9	20	238	939	0.25
COL	ALT	0.2	45	6533	70857	4.80
	RE	0.4	16	756	2503	0.39
	REAL-(16,1)	0.5	53	172	780	0.19
	REAL-(64,16)	1.2	28	240	1228	0.25
FLA	ALT	0.5	106	8195	126608	5.77
	RE	0.9	39	814	1751	0.47
	REAL-(16,1)	1.4	125	197	1067	0.19
	REAL-(64,16)	3.1	68	210	1131	0.22
NW	ALT	0.6	125	13106	166870	9.91
	RE	1.0	43	1091	3551	0.66
	REAL-(16,1)	1.6	149	211	1115	0.27
	REAL-(64,16)	3.8	78	230	1379	0.28
NE	ALT	0.7	152	12161	148647	10.91
	RE	1.8	58	1474	3466	0.86
	REAL-(16,1)	2.6	182	298	1527	0.33
	REAL-(64,16)	5.5	98	324	1234	0.42
CAL	ALT	1.0	198	22418	227812	23.28
	RE	1.9	70	1220	3700	0.73
	REAL-(16,1)	2.9	234	296	1581	0.36
	REAL-(64,16)	6.4	123	362	2123	0.44
LKS	ALT	1.5	292	21616	218084	22.78
	RE	3.2	105	1687	4015	1.12
	REAL-(16,1)	4.8	347	331	1669	0.39
	REAL-(64,16)	10.0	183	368	1800	0.45
E	ALT	2.1	375	26712	516585	29.62
	RE	3.9	132	1562	3410	1.03
	REAL-(16,1)	5.9	444	322	1575	0.42
	REAL-(64,16)	13.5	232	322	1749	0.41
W	ALT	4.1	683	75494	888950	97.61
	RE	6.4	233	1675	4223	1.03
	REAL-(16,1)	10.4	802	405	2028	0.56
	REAL-(64,16)	24.8	414	400	1782	0.58
CTR	ALT	13.4	1613	119303	1335378	234.89
	RE	25.3	522	2277	5237	2.14
	REAL-(16,1)	38.7	1880	545	2559	1.00
	REAL-(64,16)	86.9	937	453	2227	0.97
USA	ALT	17.6	2563	187968	2183718	295.44
	RE	27.9	893	2405	4813	1.77
	REAL-(16,1)	45.5	3032	592	2668	0.80
	REAL-(64,16)	113.9	1579	538	2534	0.86

TABLE 5. Data for subgraphs of USA with travel distances.

		PREP. TIME	DISK SPACE	QUERY		
GRAPH	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
NY	ALT	0.1	24	3083	35210	2.08
	RE	0.7	11	1622	3346	0.92
	REAL-(16,1)	0.8	29	222	1321	0.25
	REAL-(64,16)	1.2	17	473	1498	0.44
BAY	ALT	0.1	29	4875	80310	3.47
	RE	0.3	12	968	2252	0.48
	REAL-(16,1)	0.5	35	192	965	0.20
	REAL-(64,16)	0.9	20	264	1268	0.25
COL	ALT	0.2	42	5774	78677	3.52
	RE	0.4	16	1087	3405	0.58
	REAL-(16,1)	0.6	51	178	1181	0.19
	REAL-(64,16)	1.2	28	267	1446	0.27
FLA	ALT	0.5	101	12394	153621	9.95
	RE	1.1	39	1196	2776	0.62
	REAL-(16,1)	1.6	121	239	1162	0.28
	REAL-(64,16)	3.2	66	246	1128	0.31
NW	ALT	0.5	117	13994	131191	10.89
	RE	1.1	43	1391	4015	0.83
	REAL-(16,1)	1.6	142	219	1143	0.28
	REAL-(64,16)	3.6	76	247	1439	0.31
NE	ALT	0.7	144	15533	214572	12.88
	RE	2.4	61	2873	6170	1.72
	REAL-(16,1)	3.1	176	327	2099	0.45
	REAL-(64,16)	5.7	99	395	2095	0.52
CAL	ALT	0.9	186	27524	315506	26.33
	RE	2.3	72	1881	5156	1.11
	REAL-(16,1)	3.2	224	331	1794	0.42
	REAL-(64,16)	6.2	122	378	1799	0.47
LKS	ALT	1.4	276	41832	622476	45.78
	RE	5.2	108	3959	8475	2.91
	REAL-(16,1)	6.7	335	470	2542	0.67
	REAL-(64,16)	11.4	183	431	2926	0.66
E	ALT	1.9	354	43539	674704	52.80
	RE	5.2	136	3486	7078	2.58
	REAL-(16,1)	7.1	427	401	2066	0.62
	REAL-(64,16)	13.9	231	382	1957	0.58
W	ALT	3.4	642	75682	669930	86.42
	RE	7.9	239	2849	6596	2.02
	REAL-(16,1)	11.3	771	456	3501	0.66
	REAL-(64,16)	24.1	413	415	2575	0.62
CTR	ALT	11.9	1540	154980	1859858	276.11
	RE	42.8	538	6827	12937	6.44
	REAL-(16,1)	54.7	1845	756	4523	1.56
	REAL-(64,16)	97.8	948	563	3054	1.44
USA	ALT	15.2	2417	276195	2910133	410.73
	RE	46.4	918	7311	13886	5.78
	REAL-(16,1)	61.5	2923	905	5510	1.41
	REAL-(64,16)	120.5	1575	670	3499	1.22

USA, with both travel times and travel distances as length functions), we generated 1000 random queries for each Dijkstra rank between 8 and 24. Figures 3 to 6 report the average running times and average number of nodes scanned for each graph and each length function.

Comparing the two plots in each figure to one another, we observe that the curves showing running times and the curves showing scan counts are very similar, but shifted by a constant that is related to the time per vertex scan of each method.

As random queries tend to involve pairs of vertices 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 the higher overhead of accessing landmark information its running time can be 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 the higher overhead. ALT clearly 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 methods, and RE becomes consistently 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) and REAL-(64,16) are the best options. REAL-(64,16) is somewhat worse for very local queries, but slightly better for higher Dijkstra ranks. Its main advantage, however, is its lower space requirement.

6.1.4. *Reach-aware landmarks.* We now study how landmark sparsity influences query times and space usage. For 16 and 64 landmarks, we vary the fraction of vertices for which landmark data is maintained. Tables 6 and 7 show that, as the sparsity increases, we get a substantial reduction in the memory overhead associated with landmarks. The number of vertex scans increases steadily (but slowly).

Running times increase substantially when the sparsity increases from 1 to 2, despite the fact that the number of scans barely changes. This is because sparsity 1 corresponds to the standard REAL algorithm: all distances to and from landmarks are available, and therefore the algorithm does not need to deal with proxies, which slow down the computation. With sparsity 2, proxies must be taken into account. Although the lower bounds have similar quality (as the number of scans shows), computing them is more expensive.

Curiously, when we further increase the sparsity the effect on the running time is minor, despite the increase in the number of vertex scans. With increased sparsity, a relatively greater portion of the vertices will be scanned at the beginning of the search, before the algorithm starts using A* search. The algorithm will behave essentially like RE at the beginning, and we have seen that RE has significantly lower overhead per scan than REAL.

One can win in all measures of query performance. As already mentioned, compared to REAL-(16,1), which is not landmark-aware, REAL-(64,16) uses less space and often runs faster. Note that our current preprocessing algorithm runs landmark generation on the full graph; for the landmark-aware case, one could eliminate low-reach vertices to improve performance.

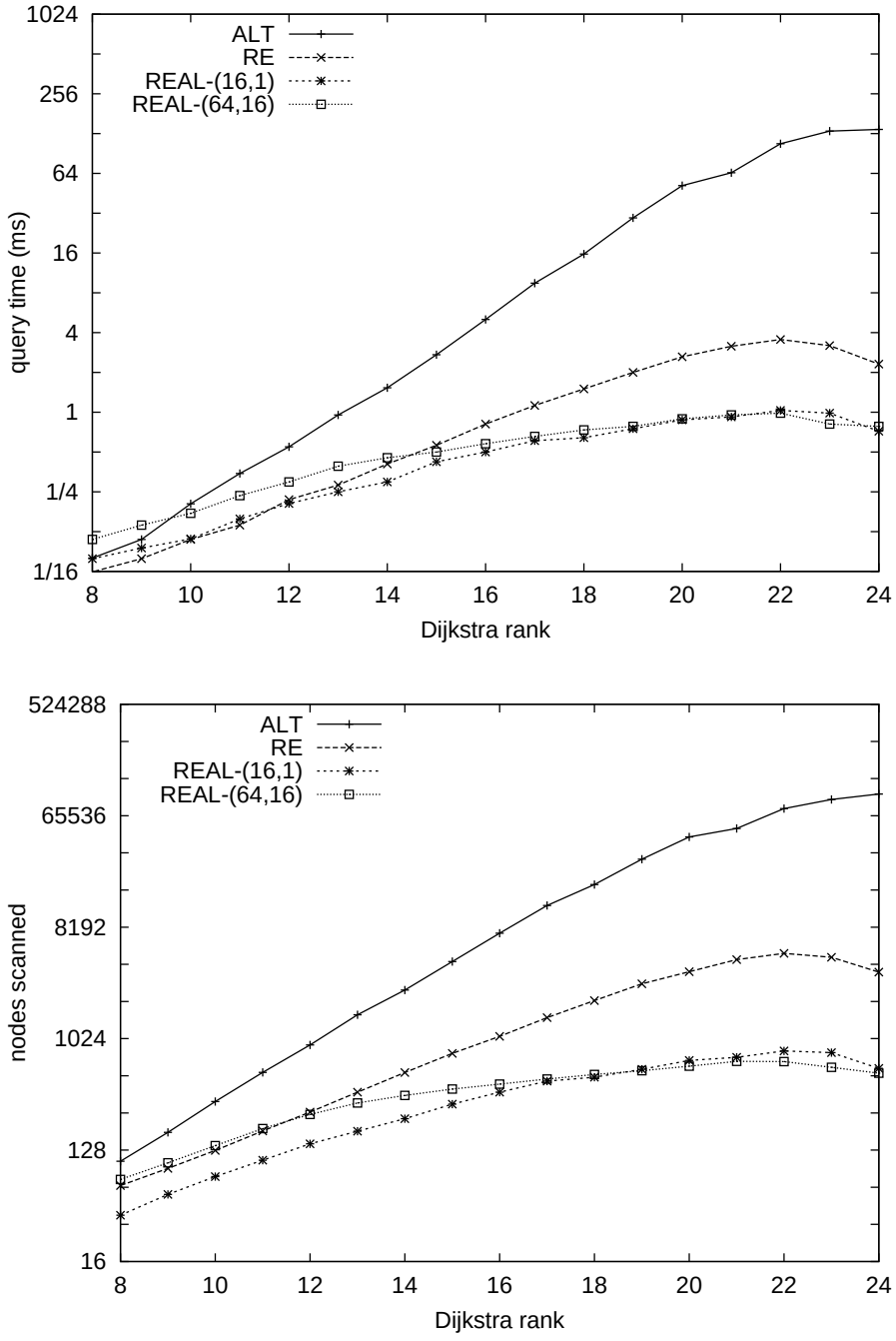

FIGURE 3. Local queries on Europe with travel times: running times and scan counts.

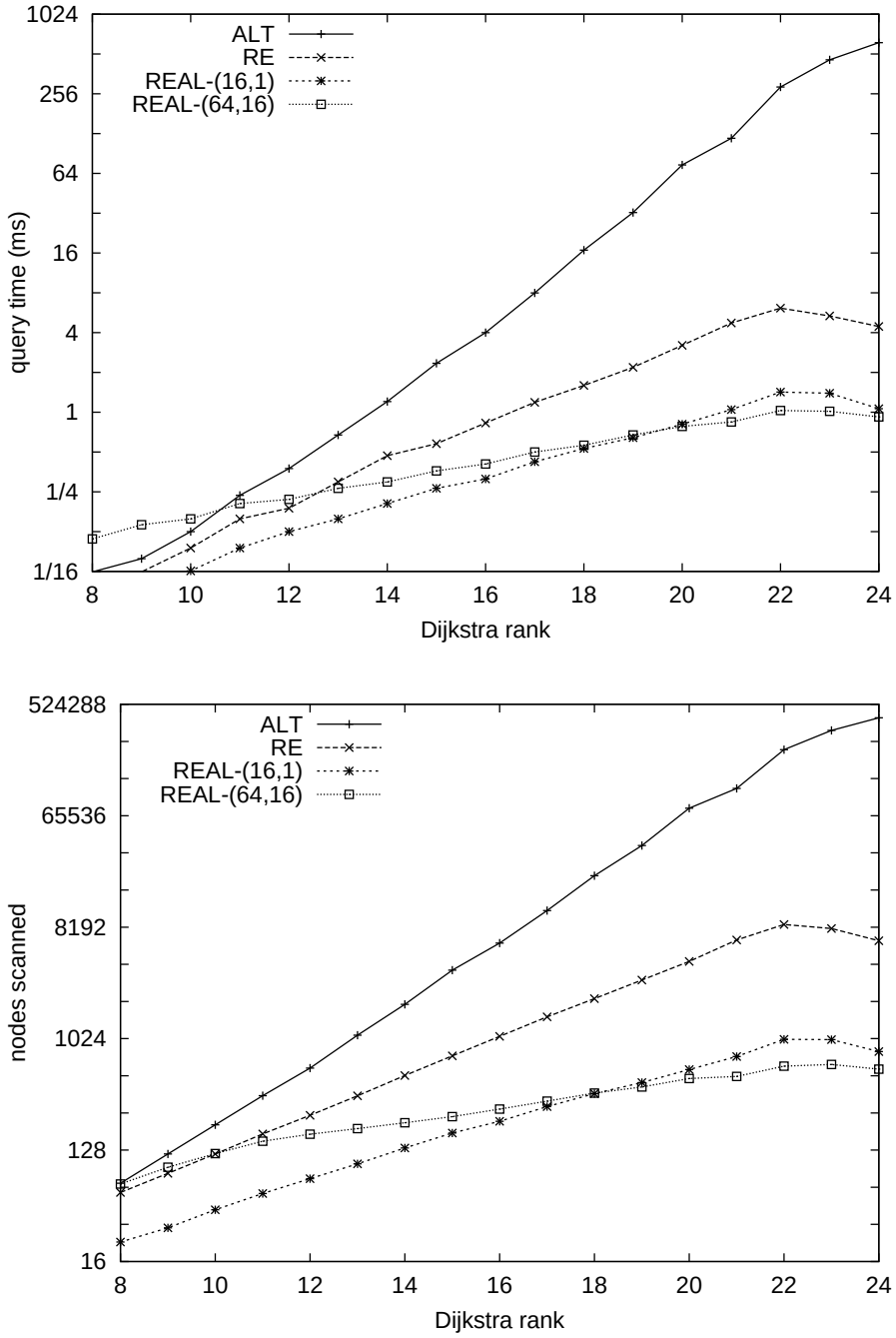

FIGURE 4. Local queries on Europe with travel distances: running times and scan counts.

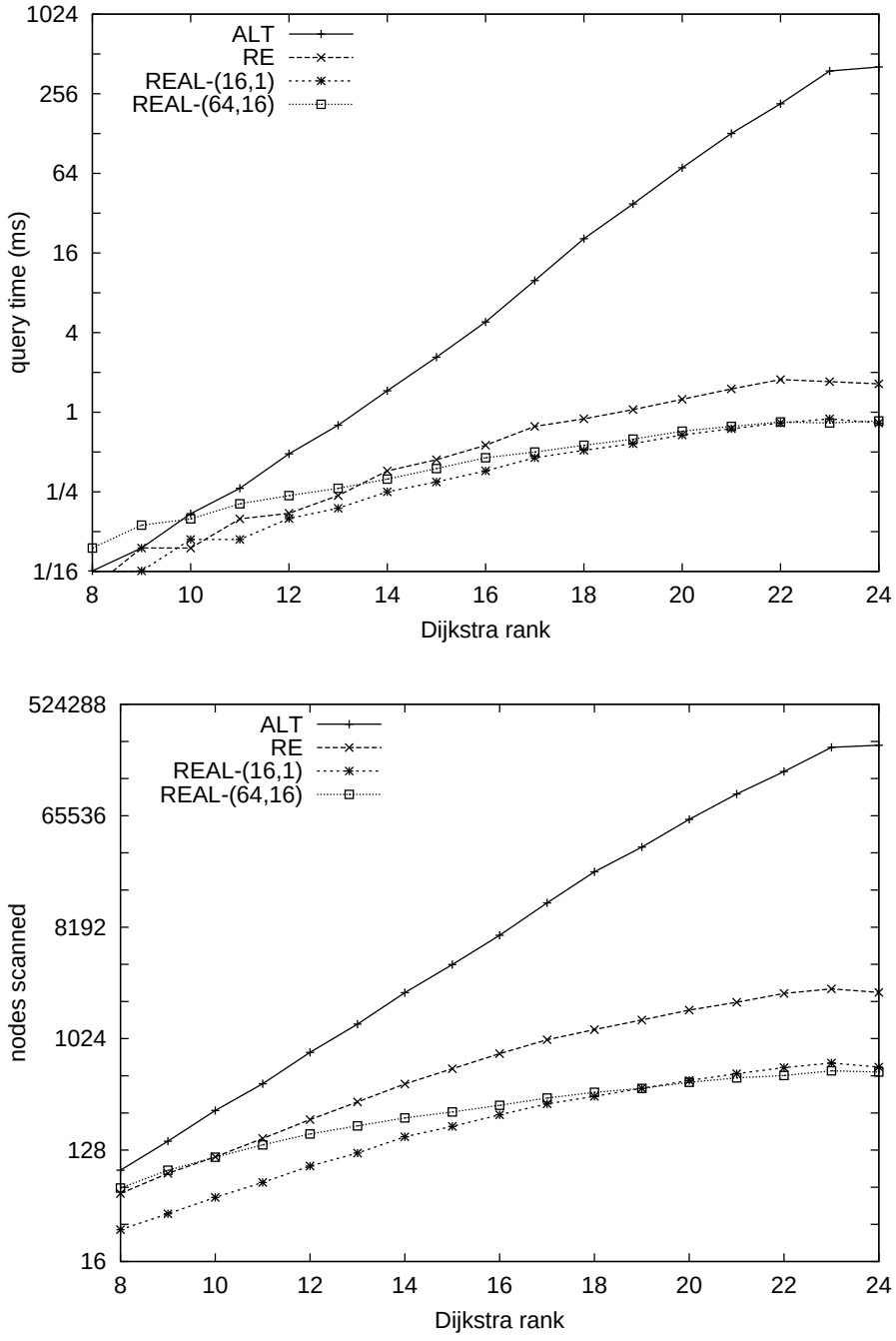

FIGURE 5. Local queries on USA with travel times: running times and scan counts.

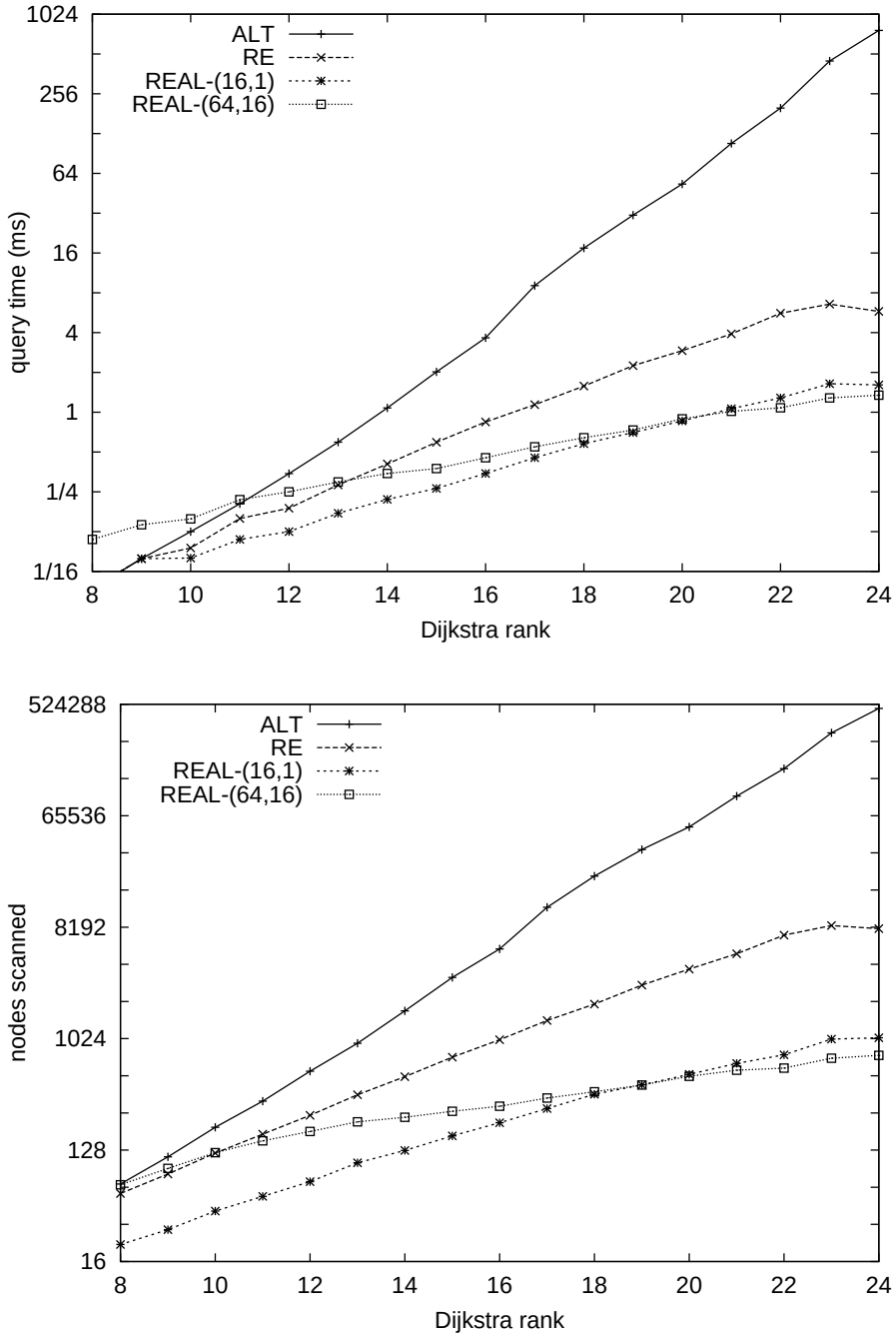

FIGURE 6. Local queries on USA with travel distances: running times and scan counts.

TABLE 6. Data for REAL on USA for various landmark/sparsity combinations.

METRIC	LAND.	SPAR.	PREP. TIME (min)	DISK SPACE (MB)	QUERY AVG SCANS	QUERY MAX SCANS	QUERY TIME (ms)
DISTANCES	16	1	61.5	2923	905	5510	1.45
		2	61.5	1960	915	5512	1.83
		4	61.5	1472	921	5696	1.84
		8	61.5	1216	933	5684	1.81
		16	61.5	1082	970	5578	1.80
		32	61.5	1014	1086	5443	1.81
		64	61.5	978	1292	5356	1.89
	64	4	120.5	3134	610	3526	1.20
		8	120.5	2109	624	3523	1.20
		16	120.5	1575	670	3499	1.16
		32	120.5	1300	810	3459	1.27
		64	120.5	1158	1049	3861	1.39
TIMES	16	1	45.5	3032	592	2668	0.81
		2	45.5	2001	600	2741	1.02
		4	45.5	1477	607	2660	0.97
		8	45.5	1206	618	2631	0.97
		16	45.5	1065	645	2696	1.00
		32	45.5	991	727	2843	1.03
		64	45.5	954	795	2968	1.05
	64	4	113.9	3229	497	2528	0.86
		8	113.9	2145	507	2536	0.81
		16	113.9	1579	538	2534	0.83
		32	113.9	1287	636	2495	0.88
		64	113.9	1137	713	2542	0.91

On a final note, observe that the number of landmark distances that must be stored depends on the ratio between the number of landmarks and the sparsity. In particular, when these ratios are the same, the number of distances stored will also be the same; this is the case of REAL-(16,8) and REAL-(64,32), for example. According to the tables, however, REAL-(64,32) actually requires slightly more space to represent the information. To understand why, recall (from Section 5.2.2) that landmark files are compressed. The compression ratio is better when vertices with similar identifiers are close to each other in the graph, which is generally true in the original graph. When vertices are partially reordered by reach, however, this is no longer true, especially for very high reaches (i.e., very low vertex identifiers): the compression ratio is much worse in the beginning of the file (high-reach vertices) than towards the end (low-reach vertices). REAL-(16,8) stores 16 files, each with $n/8$ pairs of distances (to and from the landmark); REAL-(64,32), on the other hand, must store 64 files, each with only $n/32$ pairs of distances. The average reach in the latter case is much higher, which explains the worse compression ratio.

6.1.5. *Shortcut generation.* We now examine how the performance of our algorithm is affected by the choice of shortcuts. Recall that we only allow a vertex to be bypassed if (among other criteria) the ratio of the number of new shortcuts created to the number of arcs eliminated it at most some constant c. In our algorithm, we set c to 0.5 in the first iteration, 1.0 in the second, and 1.5 for the remaining ones.

TABLE 7. Data for REAL on Europe for various landmark/sparsity combinations.

			PREP.	DISK	QUERY		
			TIME	SPACE	AVG	MAX	TIME
METRIC	LAND.	SPAR.	(min)	(MB)	SCANS	SCANS	(ms)
DISTANCES	16	1	41.1	1938	855	4867	1.20
		2	41.1	1326	865	4865	1.55
		4	41.1	1019	872	4633	1.52
		8	41.1	862	884	4659	1.44
		16	41.1	782	909	4779	1.47
		32	41.1	741	1007	4774	1.55
		64	41.1	720	1353	4636	1.67
	64	4	76.0	2026	515	2748	0.89
		8	76.0	1402	527	2711	0.89
		16	76.0	1084	562	2596	0.91
		32	76.0	920	692	2858	1.03
		64	76.0	837	1095	3214	1.22
TIMES	16	1	57.7	1869	714	3387	0.84
		2	57.7	1270	731	3386	1.12
		4	57.7	973	745	3421	1.16
		8	57.7	822	757	3440	1.12
		16	57.7	745	810	3467	1.11
		32	57.7	706	1038	3468	1.22
		64	57.7	686	1400	4754	1.47
	64	4	102.6	1944	534	3114	0.84
		8	102.6	1343	547	3107	0.84
		16	102.6	1037	610	2998	0.88
		32	102.6	880	876	3097	1.06
		64	102.6	800	1289	4754	1.31

We now consider what happens when the same (fixed) value of c is used in every iteration. We considered values of c from 0.5 to 2.5 and tested our four main graphs (USA and Europe with travel times and travel distances). For each case, Table 8 shows the preprocessing time, the number of shortcuts added (as a percentage of the original number of arcs), and the corresponding query results (aggregated over 1000 random pairs of vertices).

As anticipated, larger values of c usually result in more shortcuts being added, which speeds up both preprocessing (because the graph shrinks faster) and queries (because shortcuts reduce the reaches of eliminated arcs). The differences are most noticeable when c increases from 0.5 to 1.0; the results for $c = 2.0$ and $c = 2.5$, in contrast, are almost indistinguishable, and very close to those obtained with $c = 1.5$.

Recall that our original adaptive scheme uses $c = 0.5$ for the first iteration, 1.0 for the second, and 1.5 for the remaining ones. Comparing the results in Tables 2 and 3 with those in Table 8, we note that queries in the adaptive setting are very similar with those using $c = 1.5$ throughout the algorithm, and preprocessing is slightly slower. The main advantage of the adaptive scheme is that it adds fewer shortcuts. For USA, it adds 65.4% with travel times and 72.9% with travel distances; for Europe, the corresponding numbers are 62.2% and 74.9%. Note that

TABLE 8. Performance of RE for different shortcutting schemes. Only vertices with expansion at most c can be bypassed in each iteration.

			PR. TIME	SHORTCUTS	QUERY		
GRAPH	METRIC	c	(min)	ADDED (%)	AVG SC.	MAX SC.	TIME (ms)
USA	TIMES	0.5	43.6	43.4	3893	7712	2.53
		1.0	29.5	64.7	2562	5189	1.84
		1.5	27.2	74.9	2427	4863	1.75
		2.0	26.8	77.9	2377	4837	1.77
		2.5	27.1	77.9	2375	4846	1.75
	DISTANCES	0.5	201.2	47.1	22201	42304	15.14
		1.0	71.1	72.9	10564	21812	7.73
		1.5	44.1	86.0	7265	13781	5.75
		2.0	39.4	90.2	6486	12252	5.33
		2.5	39.5	90.2	6492	12255	5.42
Europe	TIMES	0.5	155.7	35.0	10725	22437	6.00
		1.0	53.2	60.3	4924	9686	3.19
		1.5	44.8	68.6	4374	8484	3.05
		2.0	43.0	71.2	4253	8323	3.03
		2.5	43.1	71.3	4250	8323	3.03
	DISTANCES	0.5	256.5	41.5	31872	58167	18.77
		1.0	44.6	72.5	9922	18247	6.84
		1.5	30.8	82.8	7218	12983	5.26
		2.0	28.1	86.4	6521	11748	4.91
		2.5	27.6	86.5	6499	11682	4.94

these are close to the results obtained with $c = 1.0$ (fixed), but with better query performance and preprocessing time.

6.1.6. *Exact reaches and the refinement step.* Recall that the refinement step of the preprocessing algorithm recomputes δ highest reaches with an exact algorithm. In the previous experiments, we set $\delta = 2\lceil\sqrt{n}\rceil$. For the USA graph, in particular, we set δ to 9788. Table 9 shows the trade-off between preprocessing and query times as δ varies for the USA graph (with both travel times and travel distances as length functions). We tested δ values ranging from 0 to $10\lceil\sqrt{n}\rceil$; in each case, 100 000 random queries were run.

The table shows that increasing δ does improve queries, but significantly slows down the preprocessing routine. Using $\delta = 10\lceil\sqrt{n}\rceil$, queries are up to 20% faster than when no refinement step is used, but preprocessing is more than twice as slow. Recomputing all reaches would be ideal to speed up queries, but preprocessing would be prohibitively expensive. Even on Bay Area, a subgraph of USA with only 321 270 vertices, exact reach computation takes almost 2.5 hours with our new exact algorithm (the standard one takes more than 10 hours). But queries do become 40% faster with exact reaches. How much time, if any, to spend in the refinement phase of the preprocessing algorithm depends on the requirements of the application. We chose $\delta = 2\lceil\sqrt{n}\rceil$ for most of our experiments because its effect on the preprocessing time is negligible.

6.1.7. *Retrieving the shortest path.* The query times reported so far for RE and REAL consider only the task of finding the shortest path in the graph with shortcuts. Although this path has the same length as the corresponding path in the original

TABLE 9. Performance of RE on the USA graph as a function of δ (the number of vertices whose reaches are recomputed).

		PR. TIME	QUERY		
METRIC	δ	(min)	AVG SC.	MAX SC.	TIME (ms)
TIMES	0	25.1	2505	6176	1.84
	4894	26.3	2480	6171	1.83
	9788	27.7	2376	6048	1.77
	14682	29.7	2337	6000	1.75
	19576	32.5	2318	5978	1.72
	24470	36.0	2305	5952	1.70
	29364	39.8	2291	5941	1.71
	34258	44.0	2274	5911	1.68
	39152	49.4	2256	5880	1.70
	44046	55.2	2231	5847	1.66
	48940	60.3	2189	5767	1.63
DISTANCES	0	42.7	7394	15549	5.86
	4894	44.0	7391	15549	5.85
	9788	46.3	7260	15477	5.73
	14682	49.5	7213	15456	5.69
	19576	53.0	7166	15404	5.69
	24470	58.9	7087	15342	5.63
	29364	65.8	6945	15166	5.51
	34258	71.1	6638	14613	5.27
	39152	78.6	6501	14358	5.14
	44046	87.5	6422	14175	5.07
	48940	97.9	6376	14087	5.04

graph, it has much fewer arcs. On USA with travel times, for example, the shortest path between a random pair of vertices has more than 4500 vertices in the original graph, but fewer than 30 in the graph with shortcuts. With travel distances, these values increase to about 5000 and 43, respectively.

Our algorithm can retrieve the original path from the path with shortcuts in time proportional to the number of arcs on the original path. For this, it uses an *arc map*, which maps each shortcut 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 of our 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. Retrieving the original list of arcs on the USA with travel times takes roughly one millisecond, which is comparable to the time REAL takes to actually determine the shortest path distance with travel times. With travel distances, REAL is comparatively slower, but retrieving the original path still takes about one millisecond.

On the Europe graph, the overhead of retrieving the shortest path is somewhat smaller, mainly because there are fewer arcs in the original paths: less than 1400

with travel times, and less than 3400 with travel distances. The paths with shortcuts have roughly 25 and 36 arcs, respectively. Retrieving the shortest paths takes 0.6 milliseconds with travel distances, and less than 0.2 milliseconds with travel times.

6.2. Grid Graphs. Computing driving directions was the main motivation for our work, and the experiments in the previous section show that we can solve this problem efficiently. In this section, we verify whether our algorithm is effective on other classes of graphs. It does run correctly on any input graph, since it does not use any property that is specific to road networks (such as being almost planar or having a known planar embedding). Its efficiency, however, may vary.

We tested our algorithm on grid graphs with arc lengths picked uniformly at random. Unlike road networks, these graphs 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 Implementation Challenge download page. The 3-dimensional grids were generated by our own generator. Both families are directed (not symmetric), with a vertex connected to its neighbors in the grid by 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 vertices scanned, which is taken over all five instances.

TABLE 10. Data for 2-dimensional grids.

		PR. TIME	DISK SPACE	QUERY		
VERTICES	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
65536	ALT	0.04	11.5	851	6563	1.19
	RE	1.36	3.4	2192	3666	1.59
	REAL-(16,1)	1.40	12.7	222	1013	0.34
	REAL-(64,16)	1.57	5.9	1987	3135	1.77
	BD	—	2.2	21358	51819	16.69
	D	—	2.2	33752	—	14.83
131044	ALT	0.10	23.8	1404	11535	2.56
	RE	3.08	6.8	3058	5072	2.67
	REAL-(16,1)	3.17	26.2	288	1632	0.52
	REAL-(64,16)	3.49	12.1	2359	3457	2.59
	BD	—	4.5	41682	103770	40.48
	D	—	4.5	66865	—	30.72
262144	ALT	0.21	48.3	2439	27936	4.36
	RE	6.82	13.8	4466	7210	4.09
	REAL-(16,1)	7.03	53.1	366	1969	0.67
	REAL-(64,16)	7.67	24.5	2666	3398	3.36
	BD	—	9.0	85587	205668	78.80
	D	—	9.0	134492	—	63.58
524176	ALT	0.26	96.6	6057	65664	6.28
	RE	7.77	27.7	6458	10049	4.75
	REAL-(16,1)	8.03	106.3	558	3189	0.89
	REAL-(64,16)	9.14	49.2	2823	3711	2.67
	BD	—	18.0	174150	416925	160.14
	D	—	18.0	275589	—	112.80

Table 10 shows computational results for 2-dimensional grids. The table includes results for ALT, RE, two versions of REAL, D (the reference implementation of Dijkstra's algorithm made available by the 9th DIMACS Implementation Challenge) and BD (our own implementation of the bidirectional version of Dijkstra's algorithm). Note that BD scans fewer vertices 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. We did not attempt to tune the algorithms for these graphs, except by setting the parameter c (defined in Section 4.3) to 1.0 for all iterations of the reach computation algorithm (the use of increasing values of c, starting at 0.5, is tuned for road networks). It is possible that additional tuning may help, especially if it takes into account the structure of these graphs.

Compared to BD, both RE and REAL are significantly faster. Queries and preprocessing times, however, are clearly not as good as those for road networks of similar size. But the fact that there is an improvement at all is noteworthy, given the absense of a clear hierarchy on grids with random edge weights.

For the graph sizes we consider, RE and ALT have comparable query times, but RE has better asymptotic performance. Preprocessing, however, is much more expensive for RE. REAL-(16,1) benefits from the performance improvements of both RE and ALT and therefore has much better query times (although its preprocessing times are also high, due to the reach computation).

Unlike on large road networks, REAL-(64,16) queries pay a significant performance penalty compared to REAL-(16,1). This is probably because the graphs are relatively small and well-structured, which makes increasing 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 supports this conjecture. For the smallest grid in our test, REAL-(64,16) is more than 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 three times as slow as REAL-(16,1). REAL-(64,16) uses significantly less space than REAL-(16,1), however.

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 over seven times faster than ALT, and two orders of magnitude faster than the bidirectional Dijkstra algorithm.

Next we discuss the results of running our algorithms on 3-dimensional grids, shown in Table 11. Since these graphs have higher average degree, we used $c = 2.0$ when generating shortcuts. The table shows that both reach-based and landmark-based algorithms are less effective than 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 comparable in size to the European road network.

Consider queries of RE and ALT. The average number of scans for RE is about seven times greater than for ALT. The running time is about five times greater, except for the largest problem, where it is greater by a factor of slightly over three. The average number of scans suggests that REAL-(16,1) has a small asymptotic

TABLE 11. Data for 3-dimensional grids.

		PR. TIME	DISK SPACE	QUERY		
VERTICES	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
32768	ALT	0.02	4.5	472	3968	0.52
	RE	4.11	2.5	3141	7017	2.67
	REAL-(16,1)	4.13	5.5	349	1866	0.62
	REAL-(64,16)	4.20	3.4	3753	7729	3.42
	BD	—	1.6	5840	18588	4.10
	D	—	1.6	16747	—	6.90
64000	ALT	0.04	10.1	707	7560	0.91
	RE	12.23	4.9	4965	11711	4.89
	REAL-(16,1)	12.27	12.2	489	3380	0.95
	REAL-(64,16)	12.42	7.0	5872	13081	6.30
	BD	—	3.1	11338	36732	8.63
	D	—	3.1	31759	—	13.54
132651	ALT	0.09	23.5	1200	11282	1.84
	RE	38.18	10.3	8774	20467	10.05
	REAL-(16,1)	38.27	27.6	730	4584	1.80
	REAL-(64,16)	38.64	15.2	10159	21972	12.89
	BD	—	6.5	23738	79600	23.64
	D	—	6.5	66045	—	32.98
262144	ALT	0.22	49.6	2216	18157	5.31
	RE	113.15	20.3	14849	31819	18.84
	REAL-(16,1)	113.37	57.5	1159	6524	3.39
	REAL-(64,16)	114.17	30.7	16576	33169	24.78
	BD	—	12.8	48699	161036	52.31
	D	—	12.8	133552	—	89.65

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. This might be explained by the very low quality of the lower bounds provided by sparse landmarks on graphs of small diameter.

These results show that reach-based methods are barely useful for 3-dimensional grids. Preprocessing does make queries faster for large graphs, but it is very expensive. Whether this is a basic limitation of the method, or a limitation of our preprocessing algorithm, is an interesting open question.

7. Concluding Remarks

The experimental analysis has shown that our algorithms are definitely practical for computing driving directions on large road networks. On the USA and European road networks, the average shortest path can be found while scanning less than 1000 vertices, which takes about one millisecond on a standard server. By applying techniques similar to those reported by Goldberg and Werneck [**23**], we can make our algorithm work from external memory, maintaining in RAM only information about vertices actually visited during the search. Since there are so few of those, a PocketPC implementation of RE (with the preprocessed data stored on a flash card) is practical enough for everyday use. Random queries take less than

10 seconds on a PocketPC with a 400 MHz ARM processor and 128 MB of RAM, and local queries (which should be much more common) are significantly faster.

An interesting direction of future research is to make the algorithms effective on a wider range of graphs. Although our methods work reasonably well on 2-dimensional grids, we have shown that they have only limited applicability to 3-dimensional grids, for instance.

Many other open problems remain. The performance of our best query algorithms depend crucially on the quality of the reaches available. Faster algorithms to compute exact reaches (or at least better upper bounds on reaches) would speed up our queries. 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 that they find the best possible shortcuts. 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 use. 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.

Recent work. A preliminary version of this paper [**20**] was presented at the 9th DIMACS Implementation Challenge [**8**]. Several other papers presented also dealt with the point-to-point shortest path problem. Lauther [**34**] and Köhler et al. [**29**] presented algorithms based on arc flags, but their running times (for both preprocessing and queries) are dominated by REAL and HH. Delling, Sanders, et al. [**7**] presented a variant of the partial landmarks algorithm in the context of highway hierarchies, but with only modest speedups (for technical reasons A* search cannot be combined naturally with HH). In a different paper, Delling, Holzer, et al. [**6**] showed how multi-level graphs could be used to perform random queries in less than 1 ms, but only after weeks of preprocessing time.

The fastest queries were achieved by algorithms based on *transit node routing*, a concept introduced by Bast et al. [**1**] and combined with highway hierarchies by Sanders and Schultes [**42**] (these papers were later merged [**2**]). The intuition is simple: when following the shortest path from a fixed source s to any point "far away," the path leaves the source's "local area" via one of a very limited number of *access nodes*. The set of all access nodes, considering all possible sources, are the transit nodes of the graph. On USA and Europe with travel times, there are only a few thousand of those. Preprocessing computes the access nodes and the distances between them, and most queries consist of a few table lookups. Both graphs can be processed in about three hours, and random queries take in $6\,\mu$s on average (local queries are slightly slower, at $20\,\mu$s). If preprocessing time is limited to about an hour, average query times are still only $11\,\mu$s (0.3 ms for local queries). The effectiveness of transit nodes depends strongly on the natural hierarchy of the underlying road network. If travel times are replaced by travel distances, preprocessing time increases to about eight hours, and average query times are close to 0.1 ms. Performance would probably be significantly worse on other graph classes, such as grids.

Even for travel distances, queries with transit-node routing are significantly faster than with REAL. Our method does appear to be more robust, however, to

changes in the length function. Moreover, these approaches are not mutually exclusive. As Bast et al. observe in [**2**], a reach-based approach could be used instead of highway hierarchies to compute a suitable set of transit nodes and the corresponding distance tables. An actual implementation of the combined algorithm is an interesting topic for future research. It should be noted, however, that speeding up our algorithm would only make sense if a full description of the shortest path is not required; traversing the shortest path already costs about as much as finding its length.

Acknowledgements

We would like to thank Daniel Delling, Peter Sanders, and Dominik Schultes for many useful discussions. We also thank an anonymous referee for comments that helped improve the presentation of our paper.

References

[1] H. Bast, S. Funke, and D. Matijevic. TRANSIT: Ultrafast shortest-path queries with linear-time preprocessing. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.

[2] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proc. 9th International Workshop on Algorithm Engineering and Experiments*, pages 46–59. SIAM, 2006. Available at http://www.mpi-inf.mpg.de/ bast/tmp/transit.pdf.

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

[4] 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.

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

[6] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-performance multi-level graphs. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.

[7] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.

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

[9] C. Demetrescu and P. Italiano. A New Approach to Dynamic All Pairs Shortest Paths. *J. Assoc. Comput. Mach.*, 51:968–992, 2004.

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

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

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

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

[14] 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.

[15] 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.

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

[17] A. V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. In *Proc. 9th Annual European Symposium Algorithms*, pages 230–241. Springer-Verlag, 2001.

[18] A. V. Goldberg. Shortest Path Algorithms: Engineering Aspects. In *Proc. 12th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 2001.

[19] 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.

[20] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better landmarks within reach. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.

[21] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proc. 8th International Workshop on Algorithm Engineering and Experiments*, pages 38–51. SIAM, 2006.

[22] 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.

[23] 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.

[24] 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.

[25] 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):100–107, 1968.

[26] 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*, pages 2037–2044. IEEE, 1994.

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

[28] 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.

[29] E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.

[30] E. Köhler, R.H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Proc. 4th International Workshop on Efficient and Experimental Algorithms*, pages 126–138, 2005.

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

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

[33] 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.

[34] U. Lauther. An experimental evaluation of point-to-point shortest path calculation on road-networks with precalculated edge-flags. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.

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

[36] 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.

[37] R.H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *Proc. 4th International Workshop on Efficient and Experimental Algorithms*, pages 189–202, 2005.

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

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

[40] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proc. 13th Annual European Symposium Algorithms*, pages 568–579, 2005.
[41] P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proc. 14th Annual European Symposium Algorithms*, pages 804–816, 2006.
[42] P. Sanders and D. Schultes. Robust, almost constant time shortest-path queries on road networks. Presented at the 9th DIMACS Implementation Challenge, available at `http://www.dis.uniroma1.it/~challenge9/`. A revised version appears in this book, 2006.
[43] D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. Master's thesis, Department of Computer Science, Universität des Saarlandes, Germany, 2005.
[44] 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.
[45] R. Sedgewick and J.S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, 1:31–48, 1986.
[46] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
[47] M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. Assoc. Comput. Mach.*, 46:362–394, 1999.
[48] 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.
[49] DC US Census Bureau, Washington. UA Census 2000 TIGER/Line files. `http://www.census.gov/geo/www/tiger/tigerua/ua.tgr2k.html`, 2002.
[50] D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proc. 11th Annual European Symposium Algorithms*, pages 776–787, 2003.
[51] F. B. Zhan and C. E. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transp. Sci.*, 32:65–73, 1998.
[52] 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*, 4:1–11, 2000.

Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA.
E-mail address: `goldberg@microsoft.com`

School of Mathematical Sciences, Tel Aviv University, Israel. Part of this work was done while the author was visiting Microsoft Research Silicon Valley.
E-mail address: `haimk@post.tau.ac.il`

Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA.
E-mail address: `renatow@microsoft.com`