Reach for A^*: Efficient Point-to-Point Shortest Path Algorithms

Andrew V. Goldberg[1] Haim Kaplan[2] Renato F. Werneck[3]

October 2005

Technical Report
MSR-TR-2005-132

We study the point-to-point shortest path problem in a setting where preprocessing is allowed. We improve the reach-based approach of Gutman [16] in several ways. In particular, we introduce a bidirectional version of the algorithm that uses implicit lower bounds and we add shortcut arcs which reduce vertex reaches. Our modifications greatly reduce both preprocessing and query times. The resulting algorithm is as fast as the best previous method, due to Sanders and Schultes [27]. However, our algorithm is simpler and combines in a natural way with A^* search, which yields significantly better query times.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]Microsoft Research, 1065 La Avenida, Mountain View, CA 94062. Email: `goldberg@microsoft.com`; URL: `http://www.research.microsoft.com/~goldberg/`.

[2]School of Mathematical Sciences, Tel Aviv University, Israel. Part of this work was done while the author was visiting Microsoft Research. Email: `haimk@math.tau.ac.il`.

[3]Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544. Supported by the Aladdin Project, NSF Grant no. CCR-9626862. Part of this work was done while the author was visiting Microsoft Research. Email: `rwerneck@cs.princeton.edu`.

1 Introduction

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

Finding shortest paths is a fundamental problem. The single-source problem with nonnegative arc lengths has been studied most extensively [1, 3, 4, 5, 9, 10, 11, 12, 14, 19, 24, 33, 37]. For this problem, near-optimal algorithms are known both in theory, with near-linear time bounds, and in practice, where running times are within a small constant factor of the breadth-first search time.

The P2P problem with no preprocessing has been addressed, for example, in [18, 26, 31, 38]. While no nontrivial theoretical results are known for the general P2P problem, there has been work on the special case of undirected planar graphs with slightly super-linear preprocessing space. The best bound in this context appears in [8]. Algorithms for approximate shortest paths that use preprocessing have been studied; see e.g. [2, 20, 34]. Previous work on exact algorithms with preprocessing includes those using geometric information [23, 36], hierarchical decomposition [27, 29, 30], the notion of reach [16], and A^* search combined with landmark distances [13, 15].

In this paper we focus on road networks. However, our algorithms do not use any domain-specific information, such as geographical coordinates, and therefore can be applied to any network. Their efficiency, however, needs to be verified experimentally for each particular application. In addition to road networks, we briefly discuss their performance on grid graphs.

We now discuss the most relevant recent developments in preprocessing-based algorithms for road networks. Such methods have two components: a *preprocessing algorithm* that computes auxiliary data and a *query algorithm* that computes an answer for a given *s*-*t* pair.

Gutman [16] defines the notion of vertex *reach*. The reach of a vertex is, informally, a number that is big if the vertex is in the middle of a long shortest path and small otherwise. Gutman shows how to prune an *s*-*t* search based on (upper bounds on) vertex reaches and (lower bounds on) vertex distances from s and to t. He uses Euclidean distances for lower bounds, and observes that the idea of reach can be combined with Euclidean-based A^* search to improve efficiency.

Goldberg and Harrelson [13] (see also [15]) have shown that the performance of A^* (without reaches) can be significantly improved if landmark-based lower bounds are used instead of Euclidean bounds. This leads to the ALT (A^* search, landmarks, and triangle inequality) algorithm for the problem. In [13], it was noted that the ALT method could be combined with reach pruning in a natural way. Not only would the improved lower bounds direct the search better, but they would also make reach pruning more effective.

Sanders and Schultes [27] (see also [29]) introduce an interesting algorithm based on highway hierarchy; we call it the HH algorithm. They describe the algorithm for undirected graphs. They also comment at high level on how the algorithm can be extended to directed graphs, but their implementation and experimental results are for undirected graphs only.[1] We think it is likely that the directed version of HH would not lose much performance. Under this assumption, HH is the most practical of the previously published P2P algorithms for road networks. It has fast query time, relatively small memory overhead, and reasonable preprocessing complexity.

The notions of reach and highway hierarchies have different motivations: The former is aimed at pruning the shortest path search, while the latter takes advantage of inherent road hierarchy to restrict the search to a smaller subgraph. However, as we shall see, the two approaches are related. Vertices pruned by reach have low reach values and as a result belong to a low level of highway hierarchy.

In this paper we study the reach method and its relationship to the HH algorithm. We develop a shortest path algorithm based on improved reach pruning that is competitive with HH. Then we combine it with ALT to make queries even faster. The main contributions of our work are as follows.

1. We introduce several variants of the reach algorithm, including bidirectional variants that do not need explicit lower bounds.

2. We introduce the idea of adding shortcut arcs to reduce vertex reaches. A small number (less than n, the number of vertices) of shortcuts drastically improves the performance of both preprocessing and query stages of the reach-based method.

3. The above-mentioned improvements lead to an implementation which we call RE, whose performance is similar to that of HH.

4. We suggest an interpretation of HH in terms of reach. Our interpretation explains the similarities between the preprocessing algorithms of Gutman, HH, and RE.

5. We show how to combine the techniques behind RE and ALT to obtain a new algorithm, REAL. On road networks, query operation counts and running times for REAL are lower than those for RE and HH.

6. We discuss techniques for improving space and cache efficiency of our algorithms as well as ways of further improving the running time.

Note that while RE combines with ALT in a natural way, the same approach does not work for HH. We discuss the HH algorithm in more detail in Section 7.2.

In short, our results lead to a better understanding of several recent P2P algorithms, leading to simplification and improvement of the underlying techniques. This, in turn, leads to very

[1]This is an important issue because the directed case is more general and real road networks are directed.

practical algorithms. For the graph of North America road network (which has almost 30 million vertices), finding the fastest route between two random points takes less than four milliseconds on a workstation while scanning about fewer than 2000 of vertices on average. Local queries are even faster.

The paper is organized as follows. We give background and definitions in Section 2. Section 3 introduces the concept of reach and discusses how one can use them to speed up shortest path computations. Section 4 describes the reach-based query algorithm in more detail. The preprocessing algorithm is described in 5. In Section 6 we discuss A^* search and its implementation based on landmarks, then show how it can be combined with reaches. Section 7 compares the main aspects of our algorithm with Gutman's and with HH. An experimental evaluation of our algorithm is presented in Section 8. In Section 9 we suggest further improvements that can still be made to the algorithm, and Section 10 contains concluding remarks.

2 Preliminaries

The input to the preprocessing stage of the P2P algorithm is a directed graph $G = (V, A)$ with n vertices and m arcs, and nonnegative lengths $\ell(a)$ for every arc a. The query stage also has as inputs a source s and a sink t. The goal is to find a shortest path from s to t.

Let $\text{dist}(v, w)$ denote 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 [21, 22] finds shortest paths from the source to all vertices in the graph. The method works as follows (see e.g. [32]). It maintains for every vertex v its distance label $d(v)$, parent $p(v)$, and status $S(v) \in \{\texttt{unreached}, \texttt{labeled}, \texttt{scanned}\}$. Initially $d(v) = \infty$, $p(v) = nil$, and $S(v) = \texttt{unreached}$ for every vertex v. The method starts by setting $d(s) = 0$ and $S(s) = \texttt{labeled}$. While there are labeled vertices, the method picks a labeled vertex v, *relaxes* all arcs out of v, and sets $S(v) = \texttt{scanned}$. To relax an arc (v, w), one checks if $d(w) > d(v) + \ell(v, w)$ and, if true, sets $d(w) = d(v) + \ell(v, w)$, $p(w) = v$, and $S(w) = \texttt{labeled}$.

If the length function is nonnegative, the labeling method terminates with correct shortest path distances and a shortest path tree. Its efficiency depends on the rule to choose a vertex to scan next. We say that $d(v)$ is *exact* if it is equal to the distance from s to v. It is easy to see that if one always selects a vertex v such that, at the selection time, $d(v)$ is exact, then each vertex is scanned at most once. Dijkstra [5] (and independently Dantzig [3]) observed that if ℓ is nonnegative and v is a labeled vertex with the smallest distance label, then $d(v)$ is exact. We refer to the labeling method with the minimum label selection rule as *Dijkstra's algorithm*. If ℓ is nonnegative then Dijkstra's algorithm scans vertices in nondecreasing order of distance from s and scans each vertex at most once.

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

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

The *bidirectional algorithm* [3, 7, 25] alternates between running the forward and reverse versions of Dijkstra's algorithm, each maintaining its own set of distance labels. We denote by $d_f(v)$ the distance label of a vertex v maintained by the forward version of Dijkstra's algorithm, and by $d_r(v)$, the distance label of a vertex v maintained by the reverse version. (We will still use $d(v)$ when the direction would not matter or is clear from the context.) During initialization, the forward search scans s and the reverse search scans t. The algorithm also maintains the length of the shortest path seen so far, μ, and the corresponding path. Initially, $\mu = \infty$. When an arc (v, w) is relaxed by the forward search and w has already been scanned by the reverse search, we know the shortest s-v and w-t paths have lengths $d_f(v)$ and $d_r(w)$, respectively. If $\mu > d_f(v)+\ell(v, w)+d_r(w)$, we have found a path shorter than those seen before, so we update μ and its path accordingly. We perform similar updates during the reverse search. The algorithm terminates when the search in one direction selects a vertex already scanned in the other. Intuitively, the bidirectional algorithm searches two touching balls centered at s and t.

A better stopping criterion (see [15]) is as follows:

> Stop the algorithm when the sum of the minimum labels of labeled vertices for the forward and reverse searches is at least μ, the length of the shortest path seen so far.

Note that any alternation strategy works correctly. Balancing the work of the forward and reverse searches is a strategy guaranteed to be within factor of two of the optimal off-line strategy. Also note that remembering μ is necessary, since there is no guarantee that the shortest path will go through the vertex at which the algorithm stops.

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). (For now, assume that the shortest path between any two vertices is unique; Section 5.1 discusses this issue in more detail.) 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.

A straightforward way to compute the reaches of all vertices is to compute all shortest paths and apply the definition. Although polynomial, this is impractical for large graphs. More efficient heuristics compute an upper bound on the reach of every vertex. We denote an upper bound on $r(v)$ by $\bar{r}(v)$. Let $\underline{\text{dist}}(v, w)$ denote a lower bound on the distance from v to w.

The following fact allows using reaches for pruning Dijkstra's search:

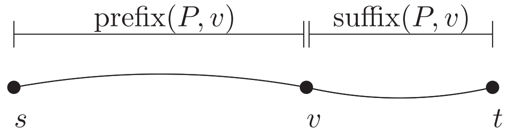

Figure 1: The reach of v with respect to the shortest path P between s and t is the minimum between the lengths of its prefix and its suffix (with respect to v).

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

Note that this also holds for the bidirectional algorithm.

Exact Reaches. As already mentioned, to compute exact reaches it suffices to look at all shortest paths in the graph and apply the definition of reach to each vertex in each path. A more efficient algorithm is as follows. Initialize $r(v) = 0$ for all vertices v. For each vertex x, grow a complete shortest path tree T_x rooted at x. For every vertex v, determine its reach $r_x(v)$ within the tree, given by the minimum between its *depth* (the distance from the root) and its *height* (the distance to its farthest descendant). If $r_x(v) > r(v)$, update $r(v)$. This algorithm runs in $\tilde{O}(nm)$ time, which is still impractical for large graphs. On the largest graph we tested, which has around 30 million vertices, this computation would take years on existing workstations.

Implicit in this algorithm is an efficiently verifiable "certificate" that proves a lower bound on reach, i.e., proves that $r(v) > L$. The certificate is a shortest path tree such that with respect to this tree $r(v) > L$. If L is in fact a lower bound on $r(v)$, such a certificate always exists, and we can verify the validity of a certificate in linear time (check if the tree is a shortest path tree, then compute v's depth and height in the tree). Unfortunately, there does not seem to be a similar certificate for upper bounds on reaches.

In contrast, we found that the following approach produces good reach lower bounds, at least for vertices of high reach. Choose a moderate number of random vertices; for each chosen vertex, compute a shortest path tree rooted at it and vertex reaches with respect to this tree; take the maximum reach over all trees for each vertex.

Unfortunately, the query algorithm needs good *upper* bounds to work correctly, not lower bounds. Upper bound algorithms are considerably more complex, as Section 5 will show.

4 Queries Using Upper Bounds on Reaches

In this section, we describe how to make the bidirectional Dijkstra's algorithm more efficient assuming we have upper bounds on the reach of every vertex. As described in Section 3, to prune the search based on the reach of some vertex v, we need a lower bound on the distance of v to the source and a lower bound on the distance of v to the sink. We show how we can use lower bounds implicit in the search itself to do the pruning, thus obtaining a new algorithm.

4.1 Implicit Bounds on Distances

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

$d_f(s, v)$
v
s
$\gamma = \underline{\text{dist}}(v, t)$
t

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

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

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

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

Remark. The self-bounding method has the following property. If a search in a certain direction prunes a vertex v, all vertices scanned by the search afterwards have reaches greater than v. This defines a "continuous hierarchy" of reaches: Once the search leaves a reach level, it never comes back to it.

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

4.2 Further Optimizations

As described earlier, reach-based pruning applies the reach test to a labeled vertex about to be scanned. In addition, we apply the test to an unreached vertex that is about to be labeled (and added to the search queue). If both $d(v)$ and the lower bound on v's distance to the target are greater than $\overline{r}(v)$, we do not add v to the queue. We call this optimization *early pruning*. If at a later stage $d(v)$ decreases to a value below $\overline{r}(v)$, then we will place v on the queue at that stage, maintaining correctness of the algorithm. Note that it is still useful to test the pruning condition before the vertex is scanned, because the lower bound on its distance to the target may have increased (because γ increased), allowing us to prune the vertex.

Another optimization, *arc sorting*, aims at early-pruning vertices without explicitly looking at them. We sort the arcs that leave a vertex by the reach bound of the head (in non-increasing order). Suppose we have just scanned an arc (v, w) and looked up $\overline{r}(w)$. Then $\overline{r}(w)$ is an upper bound on the reaches of the neighbors of v that follow w.

Arc sorting allows us to eliminate some outgoing arcs without even looking at them. If $\overline{r}(w) < d(v)$ and $\overline{r}(w) < \gamma$ (where γ is the radius of the ball searched in the opposite direction), we do not need to look at any more neighbors of v explicitly: we know they will all be pruned. Even if only the second condition holds ($\overline{r}(w) < \gamma$), we might still be able to early-prune w if $\overline{r}(w) < d(v) + \ell(v, w)$, but we must still look at the remaining neighbors of v.

We call our implementation of the bidirectional Dijkstra's algorithm with reach-based pruning RE. The query is distance-balanced and uses early pruning and arc sorting. Note that its code is very simple, with just a few tests added to the implementation of the bidirectional Dijkstra's algorithm.

5 Preprocessing

In this section we present an algorithm for efficiently computing upper bounds on vertex reaches. Our algorithm combines three main ideas, two introduced in [16], and the third implicit in [27].

The first idea is the use of *partial trees.* Instead of running a full shortest path computation from each vertex, which is expensive, we stop these computations early and use the resulting partial shortest path trees, which contain all shortest paths with length lower than a certain threshold. These trees allow us to divide vertices into two sets, those with small reaches and those with large reaches. We obtain upper bounds on the reaches of the former vertices. The second idea is to delete these low-reach vertices from the graph, replacing them by penalties used in the rest of the computation. Then we recursively bound reaches of the remaining vertices. The third idea is to introduce *shortcut arcs* to reduce the reach of some vertices. This speeds up both the preprocessing (since the graph will shrink faster) and the queries (since more vertices will be pruned).

The preprocessing algorithm works in two phases: during the *main phase*, partial trees are grown and shortcuts are added; this is followed by the *refinement phase*, when high-reach vertices are re-evaluated in order to improve their reach bounds.

The main phase uses two subroutines: one adds shortcuts to the graph (*shortcut step*), and the other runs the partial-trees algorithm and eliminates low-reach vertices (*partial-trees step*). The main phase starts by applying the shortcut step. Then it proceeds in iterations, each associated with a threshold ϵ_i (an increasing function of i, the iteration number). By the end of the i-th iteration, the algorithm will have eliminated every vertex whose reach it can prove is less than ϵ_i. Each iteration applies a partial-trees step followed by a shortcut step. If there are still vertices left in the graph after iteration i, we set $\epsilon_{i+1} = \alpha\,\epsilon_i$ (for some $\alpha > 1$) and proceed to the next iteration.

The remainder of this section describes in detail each of the main components of the algorithm: growing partial trees (Section 5.2), adding shortcuts (Section 5.3), and the refinement phase (Section 5.4). Before that, we introduce the concept of canonical paths, which are essential to our preprocessing algorithm.

5.1 Canonical Paths and Shortcuts

Consider the graph $G = (\{a, b, c, d\}, \{(a, b), (b, c), (c, d)\})$ with $\ell(a, b) = \ell(c, d) = 100$ and $\ell(b, c) = 1$. It is easy to see that $r(a) = r(d) = 0$ and $r(b) = r(c) = 100$. Suppose we add an arc (a, d) with length $\ell(a, d) = 201$, equal to the distance from a to d in the original graph. No distances change. However, if we break the tie by making the path using the new arc preferable (i.e., implicitly shorter than the original path), the reaches of b and c decrease from 100 to 1. Thus, introducing "preferred" shortcuts can drastically reduce vertex reaches. See Figure 3.

Given a path P from v to w, we say that an arc (v, w) is a *shortcut arc for P* if the length of

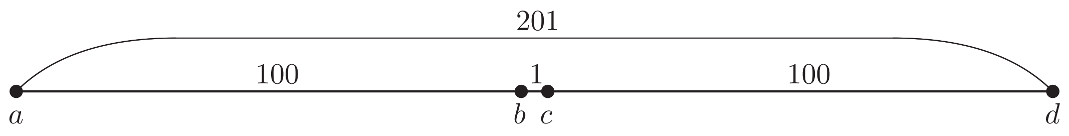

Figure 3: Without the shortcut (a, d), $r(b) = r(c) = 100$; with the shortcut, $r(b) = r(c) = 1$. Both a and d have reach 0 with or without the shortcut.

the arc is equal to the length of the path.

Approximate reach algorithms, including ours, need the notion of *canonical path*, which is a shortest path with additional properties. 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]

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.

6. (Nesting property.) Original graph paths corresponding to shortcuts are nested: for any pair of such paths, either they do not intersect, or one is contained in the other.

We need Property 5 of canonical paths to ensure that adding shortcut arcs decreases vertex reaches. Property 6 simplifies the correctness proof and, if no parallel shortcut arcs are introduced, bounds the number of shortcut arcs by $2n$.

We implement canonical paths as follows. For each 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.[3] 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 on the paths (fewer hops are better) in addition to perturbations. Note that Dijkstra's algorithm can maintain the number of hops of candidate paths. Because of the nesting property of shortcut paths one can see that there are no ties remaining

[2]If the graph has no cycles of zero-length arcs, all shortest paths are simple.

[3]In our implementation, perturbations were picked uniformly at random from the range $[1, 65535]$.

after breaking ties by perturbations and hops even in the presence of shortcuts. Furthermore, 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: a small probability of failure due to ties in sums of perturbations and memory overhead for storing perturbations. Regarding the issue former, the algorithm worked correctly for every one of the tens of thousands instances used in our experiments. As for the latter, it is an issue in the preprocessing phase only, however. During the query phase, where memory efficiency is most important, we can completely ignore the tie-breaking rules; in fact, we do not need perturbations at all to ensure correctness.

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. The following fact justifies pruning vertices based on these reaches.

> Suppose r is a reach function with respect to some choice of canonical paths. Consider an implementation of Dijkstra's algorithm (unidirectional or bidirectional) that prunes vertices v such that $r(v) < \mathrm{dist}(s, v)$ and $r(v) < \mathrm{dist}(v, t)$. Then this implementation of Dijkstra's algorithm is correct.

This observation follows from the fact that vertices on the canonical path from s to t are not pruned, and therefore the algorithm will find some shortest path.

5.2 Growing Partial Trees

To gain intuition on the construction and use of partial trees, we consider a graph such that all shortest paths are unique (and therefore canonical) and a parameter ϵ. We outline an algorithm that partitions vertices into two groups, those with high reach (ϵ or more) and those with low reach (less than ϵ). For each vertex x in the graph, the algorithm runs Dijkstra's shortest path algorithm from x with an early termination condition. Let T be the current tentative shortest path tree maintained by the algorithm, and let T' be the subtree of T induced by the scanned vertices. Note that any path in T' is a shortest path in the graph. The tree construction stops when for every leaf y of T', one of the following two conditions holds:

1. y is a leaf of T.
2. if x' is the vertex adjacent to x on the x-y path in T', then the length of the x'-y path in T' is at least 2ϵ.

Let T_x, the *partial tree of* x, denote T' at the time the tree construction stops. The algorithm marks all vertices that have reach at least ϵ with respect to a path in T_x as high-reach vertices.

It is clear that the algorithm will never mark a vertex whose reach is less than ϵ, since its reach restricted to the partial trees cannot be greater than its actual reach. Therefore, to prove the correctness of the algorithm, it is enough to show that every vertex v with high reach is marked at the end. Consider a minimal canonical path P such that the reach of v with respect to P is high. Let x and y be the first and the last vertices of P, respectively. Consider T_x. By uniqueness of shortest paths, either P is a path in T_x, or P contains a subpath of T_x that starts at x and ends at a leaf, z, of T_x. In the former case, v is marked. For the latter case, note that z cannot be a leaf of T as z has been scanned and the shortest path P continues past z. The distance from x to v is at least ϵ and the distance from x', the successor of x on P, to v is less than ϵ (otherwise P would not be minimal). By the algorithm, the distance from x' to z is at least 2ϵ and therefore the distance from v to z is at least ϵ. Thus in this case v is also marked.

Note that long arcs pose an efficiency problem for this approach. For example, if x has an arc with length 100ϵ adjacent to it, the depth of T_x is at least 102ϵ. Building T_x will be expensive. All partial-tree-based preprocessing algorithms, including ours, deal with this problem by building smaller trees in such cases and potentially classifying some low-reach vertices as having high reach. This results in weaker upper bounds on reaches and potentially slower query times, but correctness is preserved.

Our preprocessing routine runs the partial-trees algorithm in iterations, multiplying the value of ϵ by a constant α each time it starts a new iteration. In each subsequent iteration the algorithm runs the partial-trees algorithm only on the subgraph consisting of arcs whose reach bound is larger than ϵ, incorporating penalties inherited from arcs deleted in previous iterations. Note that, to facilitate the addition of shortcuts, we use arc reaches, which are closely related to vertex reaches. After all iterations are finished, we convert these arc reaches to vertex reaches. Section 5.2.2 will describe the algorithm in more detail. Before we start, however, we need the definition of *arc reach*.

5.2.1 Arc Reach

The notion of *arc reach* (implicit in [27]) is similar to that of vertex reaches. (We shall refer to our original definition as *vertex reach* whenever there is ambiguity.) Given a path P from s to t and an arc (v, w) on P, we define the *reach of* (v, w) *with respect to* P to be the minimum of the length of the prefix of P from s to w and the length of the suffix of P from v to t.[4] The *reach* of (v, w), 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. We denote an upper bound on $r(v, w)$ by $\overline{r}(v, w)$. Pruning based on arc reaches is similar to pruning based on vertex reaches. The following fact allows using arc reaches for pruning:

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

[4]Note that arc (v, w) is included in both the prefix and the suffix. A natural alternative definition would be to exclude the arc from both. These two definitions are equivalent, since it suffices to add or subtract $\ell(v, w)$ to get from one to the other.

Arc reaches are more powerful than vertex reaches. If $r(v, w) = R$, then $r(w) \geq R - \ell(v, w)$. This implies that, if one of the reach pruning conditions fails, then the corresponding arc-reach pruning condition also fails. Thus if a search that uses arc reaches scans (v, w), the search that uses vertex reaches will not early-prune w from v. However, one can construct a graph where $r(v, w)$ is much smaller than $r(v)$ and $r(w)$, so the search based on arc reaches will prune an arc (v, w) but the search based on vertex reaches will prune neither v nor w.

Although arc reaches are more effective, they are more expensive to store, for two reasons. First, the number of arcs is bigger than the number of vertices. Second, since each arc appears twice (in the forward graph and in the reverse graph), we must either replicate the reach value in two locations or assign identifiers to the arcs (and store them somewhere). We use arc reaches in preprocessing, where memory is less of an issue and the performance improvement is more significant. For queries, we use vertex reaches.

To do that, we must convert the upper bounds we obtain on arc reaches into upper bounds on vertex reaches. Consider a vertex v, an arc (v, w), and a path P that determines $r(v)$. Assume that $r(v) > 0$ and therefore 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 reach of these arcs with respect to P is 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 a straightforward bound on $r(u, v)$ and $r(v, w)$ yields $r(v) \leq \min\{\max_x\{r(x, v)\}, \max_y\{r(v, y)\}\}$. In other words, a valid bound for $r(v)$ is the minimum over the highest incoming arc reach and the highest outgoing arc reach.

Often, however, the two maximums are achieved for $x = y$. Although the upper bound in this case is still valid, it may be much higher than necessary. Since we know that $u \neq w$, we can exclude this case as follows. First, we find the maximum of $r(x, v)$ over all x. Let x' be the neighbor of v such that this maximum is $r(x', v)$. Then we find the maximum of $r(v, y)$ over all $y \neq x'$. Let y' be the outgoing neighbor of v such that $r(v, y')$ is this maximum. Let δ_1 be the minimum of $r(x', v)$ and $r(v, y')$. Second, we find the maximum of $r(v, y)$ over all y, and the vertex y' such that $r(v, y')$ is this maximum. Then we find the maximum of $r(x, v)$ over all $x \neq y'$, and the vertex x' such that $r(x', v)$ is this maximum. Let δ_2 be the minimum of $r(x', v)$ and $r(v, y')$. Note that δ_1 and δ_2 may be different from each other in a directed graph. It is not hard to see that a valid upper bound on $r(v)$ is the maximum of δ_1 and δ_2.

5.2.2 Partial Trees for Arc Reaches

We now describe how one can use partial trees to find arcs whose reaches are smaller than a threshold ϵ. Let $G' = (V', A')$ be the graph we are working with. We initialize the reach estimates of all arcs in A' to zero. We then grow the partial trees from each vertex in V', compute the reaches of the appropriate arcs within these trees, and update the reach estimates accordingly. For each arc, the reach will end up being the maximum reach observed within all relevant partial trees.

Consider a partial shortest path tree T_x rooted at a vertex x, and let $v \neq x$ be a vertex in this tree. Let $f(v)$ be the vertex adjacent to v on the shortest path from x to v. The *inner circle* of T_x is the set containing the root x and all vertices $v \in T_x$ such that $d(v) - \ell(x, f(v)) \leq \epsilon$. We call vertices in the inner circle *inner vertices*; all other vertices in T_x are *outer vertices*. The *distance* from an outer vertex w to the inner circle is defined in the obvious way, as the length of the path (in T_x) between the closest (to w) inner vertex and w itself. The partial tree stops growing when all labeled vertices are outer vertices and have distance to the inner circle greater than ϵ.

Remark. Note that the criterion we use to stop growing the tree is heuristic. To guarantee that all short paths are considered, we must grow the tree until the *parent* of every labeled vertex is within distance greater than ϵ from the inner circle. Our stopping criterion makes the search faster, but it may cause the reach estimates of some vertices to be higher than the actual value.

Once the partial tree is built, we compute the reach (within the tree) of all arcs whose heads belong to the inner circle. For that, we define the *depth* of v, denoted by $\mathit{depth}(v)$, as the distance from the root x to v within the tree. The *height* of v, denoted by $\mathit{height}(v)$, is defined as the distance from v to its farthest descendant, *as long as no descendant is labeled* (i.e., if they are all scanned). If v does have a labeled descendant, then $\mathit{height}(v) = \infty$. The reach of an arc (u, v) with respect to the tree is defined as $r((u, v), T_x) = \min\{\mathit{depth}(v), \mathit{height}(v) + \ell(u, v)\}$. One can compute the reach of all inner arcs (arcs whose endpoints are both inner vertices) in $O(|T_x|)$ total time. For each such arc, we verify if the reach within the tree is greater than the current estimate; if it is, the estimate is updated.

After all partial trees are grown, every reach estimate with value at most ϵ will be valid. We then eliminate all arcs with reach estimate below ϵ from the graph. The remaining arcs, i.e., those with estimate greater than ϵ (including ∞), are kept in the graph for the next iteration.

In general, iteration i applies the partial-trees algorithm to a graph $G_i = (V_i, A_i)$. This is the graph induced by all arcs that have not been eliminated yet (considering not only the original arcs, but also shortcuts added in previous iterations). All arcs in A_i have reach estimates above ϵ_{i-1} (for $i > 1$). To compute valid upper bounds for these arcs, the partial-trees algorithm must take into account the deleted arcs. It does so using *penalties*, which we define next.

5.2.3 Penalties

Let $G_i = (V_i, A_i)$ be the subgraph processed by the partial-trees algorithm at iteration i. We define the *in-penalty* of a vertex $v \in V_i$ as

$$\mathit{in\text{-}penalty}(v) = \max_{(u,v) \in A \setminus A_i} \{\bar{r}(u, v)\},$$

if v has at least one eliminated incoming arc, and zero otherwise. The *out-penalty* of v is defined similarly, considering outgoing arcs instead of incoming arcs:

$$\textit{out-penalty}(v) = \max_{(v,w)\in A\setminus A_i} \{\bar{r}(v,w)\}.$$

If there is no outgoing arc, the out-penalty is zero.

The partial-trees algorithm will work as before, but *depth* and *height* must be redefined. Given a partial tree T_x rooted at x, the depth of a vertex $v \in T_x$ is now defined as

$$\textit{depth}(v) = d(v) + \textit{in-penalty}(x),$$

where $d(v)$ is the distance between x and v in the tree.

To define the height of a vertex, we need the concept of *pseudo-leaves*. For each vertex v in the partial tree, create a new child v' (the pseudo-leaf) and an arc (v, v') with length equal to *out-penalty*(v). Intuitively, v' acts as a representative of all original arcs incident to v that are not present in the current subgraph. We conservatively assume that the shortest path from x to v could actually be extended to v'. As this is not always true, we end up computing only upper bounds on reaches.

The height of a vertex v is now defined as the distance between v and the farthest pseudo-leaf. We stress that pseudo-leaves need to be added only implicitly, and only after the partial tree has been built. When growing a partial tree, we do not need to worry about pseudo-leaves at all.

Remark. At the end of iteration i, an arc (u, v) may end up with a reach estimate that is greater than ϵ_i, but finite. This arc *cannot* be eliminated, since its actual reach may be higher than this estimate. The algorithm might miss the path x-y that proves this is the case because v may not belong to the inner circle of x.

5.3 Shortcuts

The shortcut step looks for bypassable vertices. We call a vertex v *bypassable* if one of the following conditions holds:

- v has exactly one incoming arc (u, v) and one outgoing arc (v, w);
- v has exactly two outgoing arcs, (v, u) and (v, w), and exactly two incoming arcs, (u, v) and (w, v);

In both cases, we assume that $u \neq w$, which means that v has exactly two neighbors. In the first case, we say v is a candidate for a *one-way bypass*; in the second, v is a candidate for a *two-way bypass*. Shortcuts will be used to go around bypassable vertices.

A *line* is a path in the graph containing at least three vertices such that all vertices, except the first and the last, are bypassable. Every bypassable vertex belongs to exactly one line. Lines can be either *one-way* or *two-way*, depending on the type of bypassable vertices it contains (it is easy to see that there can be only one vertex type in a line).

Once a line is identified, we may bypass it. The simplest approach would be to do it in a single step: if its first vertex is u and the last one is w, we could simply add a shortcut (u, w) (and (w, u), in case it is a two-way line). However, if the line has several arcs, it is a good idea to shortcut "sub-lines" as well. This will reduce the reaches even further, as the example in Figure 4 shows.

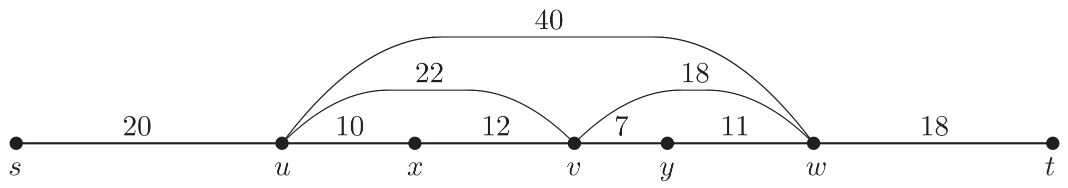

Figure 4: In this graph, (s, u), (u, x), (x, v), (v, y), (y, w), and (w, t) are the original edges (for simplicity, the graph is undirected). Without shortcuts, their reaches are $r(s) = 0$, $r(u) = 20$, $r(x) = 30$, $r(v) = 36$, $r(y) = 29$, $r(w) = 18$, and $r(t) = 0$. If we add just shortcut (u, w), the reaches of three vertices are reduced: $r(x) = 19$, $r(v) = 12$, and $r(y) = 19$. If we also add shortcuts (u, v) and (v, w), the reaches of x and y are reduced even further, to $r(x) = r(y) = 0$.

More precisely, we proceed as follows. Let $\Lambda = (u \cdots w)$ be a line with $k \geq 2$ segments ($k + 1$ vertices). If $k = 2$, we bypass the only internal vertex by adding a new arc (u, w). If $k > 2$, we find the internal vertex v that is closest to the median of path $u \cdots w$ (with respect to arc lengths, not number of vertices).[5] Then we recursively process subpaths $(u \cdots v)$ and $(v \cdots w)$, causing shortcuts (u, v) and (v, w) to be added to the graph (unless the corresponding arcs already belong to the graph). Finally, we create the shortcut (u, w).

5.3.1 Avoiding Long Arcs

The method above tends to create long shortcuts, since there are often lines with dozens of arcs. This may cause the partial-trees algorithm to be less effective in future iterations. To prevent this from happening, for each iteration i we define a maximum length λ_i that new arcs may have. In our experiments, we set $\lambda_i = \epsilon_{i+1}/2$ for $i \geq 0$ (note that λ_0 is the value used by the shortcut step that precedes the first iteration).

To take this bound into account, we modify the procedure above in the obvious way. Consider a line $\Lambda = (u \cdots w)$ with k segments. If $k = 2$, we only create a shortcut if the length of the line is smaller than λ_i; if it is longer, we do nothing. If $k > 3$, we make the recursive calls regardless of the length of the line; however, the final shortcut (linking u and w) is only added if its length is less than λ_i.

[5]When finding v on a two-way line, we consider the length of a segment (x, y) to be $\max\{\ell(x, y), \ell(y, x)\}$; on road networks, these two lengths are usually the same or very close to each other.

To decide whether to add shortcuts or not, we compute the length of a bidirectional line by taking the maximum length of the two arcs in each segment and adding them all. As a result, if the algorithm decides to create a shortcut in one direction, it will also create one in the other.

5.3.2 Eliminating Bypassed Vertices

Consider a one-way line composed of three vertices, u, v, and w. When we add a shortcut from u to w, we know that arc (u, v) will never be used on a shortest path that goes through u to w anymore. Any path that uses (u, v) will end either in v or in some low-reach area neighboring v. Therefore, a valid upper bound for the reach of (u, v) is $\overline{r}(u, v) = \ell(u, v) + \textit{out-penalty}(v)$. Similarly, a valid upper bound for the reach of (v, w) will be $\overline{r}(v, w) = \ell(v, w) + \textit{in-penalty}(v)$. Once we have these bounds, we can immediately remove v, as well as (u, v) and (v, w), from the graph and update the appropriate penalties. A similar procedure can be adopted for a two-way line. The two bounds above remain valid, and we can also say that $\overline{r}(w, v) = \ell(w, v) + \textit{out-penalty}(v)$ and $\overline{r}(v, u) = \ell(v, u) + \textit{in-penalty}(v)$.

Although we could generalize this for lines with more than three vertices, there is no need to. Our algorithm never creates a shortcut that bypasses more than one vertex; whenever it encounters a longer line, it finds a vertex v close to the median, divides the line into two parts, and recursively replaces each part by a single segment. When the recursive call is finished, the neighbors of v will be the extremes of the original line.

5.4 The Refinement Phase

The fact that penalties are used to help compute valid upper bounds tends to make the bounds less tight (in absolute terms) as the algorithm progresses, since penalties become higher. Therefore, additive errors tend to be larger for vertices that remain in the graph after several iterations. This is unfortunate because these are arguably the most important vertices in the graph. Since they have high reach, they are visited by more queries than other vertices. If we could make these reaches more precise, the query would be able to prune more vertices.

This is the goal of the *refinement phase* of our algorithm. After finding valid upper bounds using the partial-trees algorithm, the refinement step will recompute the reach of the δ vertices with highest (upper bounds on) reaches, where δ is a user-defined parameter.

Let V_δ be this set of high-reach vertices of G. To recompute the reaches, we first compute the subgraph $G_\delta = (V_\delta, A_\delta)$ induced by V_δ. This graph contains not only original arcs, but also the shortcuts between vertices in V_δ added during the main phase. We then run an exact reach computation on G_δ by growing a complete shortest path tree from each vertex in V_δ. Because these shortest path trees include vertices in G_δ only, we still have to use in- and out-penalties to account for the remaining vertices. But these penalties tend to be smaller, since the arcs with highest reaches will be inside G_δ.

During the refinement phase, we compute vertex reaches directly (instead of arc reaches), since we need not worry about making the graph shrink faster at this point. No new shortcuts are added (even though they could be). Since the refinement phase grows a shortest path tree from every vertex in V_δ, its running time is at least $\Omega(\delta^2)$. For our experiments, we chose $\delta = \lceil 10\sqrt{n} \rceil$, which made the refinement phase take a fraction of the time taken by the main phase of the preprocessing algorithm (usually around 30% for road networks with travel times, less for other graphs).

5.5 Correctness

By now we have completely described our preprocessing algorithm. We start with a shortcut step, and then proceed in iterations. Each iteration applies a partial-trees step to compute upper bounds on arc reaches, followed by a shortcut step that adds shortcuts and deletes bypassed arcs (while bounding their reach). We then convert arc reaches to vertex reaches as described in Section 5.2.1, and, finally, compute better upper bounds for high-reach vertices in the refinement phase. The following theorem establishes the correctness of our preprocessing algorithm.

For the proof we need the notion of penalty reaches. We define *penalty reaches* with respect to a set of canonical paths, in a graph where each vertex v has an *in-penalty*(v), and an *out-penalty*(v) associated with it as follows. Let P be a canonical path from vertex y to vertex z containing arc (v, w). The *penalty reach of* (v, w) *with respect to* P is the minimum between the length of the prefix of P from y to w plus *in-penalty*(y), and the length of the suffix of P from v to z plus *out-penalty*(z). The *penalty reach* of an arc (v, w) is the maximum over all canonical paths P containing (v, w) of the *penalty reach* of (v, w) with respect to P. Analogously we define a penalty reach for each vertex in G.

Theorem 5.1 *Let $G' = (V, A')$ be the original graph $G = (V, A)$ with all shortcuts added. The reach bounds computed by our preprocessing algorithm are valid with respect to canonical paths in G'.*

Proof. We start by proving that upper bounds on arc reaches computed by the main phase of the algorithm are correct with respect to canonical paths in G'.

We break the iterations of the algorithm into steps and prove the theorem by induction on number of steps. We define a *step* to be either (1) a partial trees computation followed by elimination of all arcs with reach smaller than the appropriate threshold, or (2) an addition of a single shortcut and the elimination of the bypassed arcs.

We denote by G_t the graph on which we perform step t: G_t includes all original arcs and shortcuts that have been added up to step t and have not been eliminated by step t. Let G'_t be the original graph along with all shortcuts added up to step t. Note that G_t is a subgraph of G'_t. We prove by induction on the number of steps that

1. Upper bounds on reaches computed for arcs eliminated up to (but not including) step t are valid with respect to canonical paths in G'_t, and
2. Suppose (v, w) is present in G_t. Let P' be a canonical path in G'_t through (v, w). Let P be the maximal subpath of P' in G_t containing (v, w). Then the penalty reach of (v, w) in G_t with respect to P is no smaller than the reach of (v, w) with respect to P' in G'_t.

In the following we assume that the induction hypothesis holds before step t (for G_t and G'_t), and show that it holds after step t (for G_{t+1} and G'_{t+1}).

Partial tree step, first part. Assume first that step t is a partial tree computation at the beginning of iteration i. It is straightforward to verify that our partial tree algorithm computes upper bounds on penalty reaches in G_t for every arc whose penalty reach is at most ϵ_i. Since by our induction hypothesis these penalty reaches are upper bounds on the reaches of the corresponding arcs in G'_t, and $G'_t = G'_{t+1}$ (no shortcuts are added in this step), we establish the first part of the induction hypothesis for G'_{t+1}.

Partial tree step, second part. To prove the second part of the induction hypothesis, consider a canonical path P in G_t that goes through (v, w). Let P^- be the maximal subpath of P in G_{t+1} that contains (v, w). We claim that the penalty reach of (v, w) in G_{t+1} with respect to P^- is at least as large as the penalty reach of (v, w) in G_t with respect to P. The second part of the induction hypothesis holds for G'_{t+1} because of this claim and because it holds for G'_t. Let x be the first vertex on P^- and let y be the last vertex on P^-. The correctness of the claim follows from the following observations.

1. The in-penalty of x in G_{t+1} is at least as large as the in-penalty of x in G_t, and at least as large as the penalty reach in G_t of every incoming arc into x that has been eliminated in step t.
2. The out-penalty of y in G_{t+1} is at least as large as the out-penalty of y in G_t and at least as large as the penalty reach in G_t of every arc outgoing from y that has been eliminated in step t.
3. Suppose x is not the first vertex of P and let x' be the predecessor of x on P. Then (i) the arc (x', x) has been eliminated in step t, and (ii) the penalty reach of (x', x) in G_t plus the length of the subpath of P from x to w is at least as large as the penalty reach of (v, w) in G_t. Therefore from item 1 above we obtain that *in-penalty*(x) plus the length of the subpath of P from x to w is at least as large as the penalty reach of (v, w) in G_t.
4. Suppose y is not the last vertex of P and let y' be the successor of y on P. Then (i) the arc (y, y') has been eliminated in step t, and (ii) the penalty reach of (y, y') in G_t plus the

length of the subpath of P from v to y is at least as large as the penalty reach of (v, w) in G_t. Therefore from item 2 above we obtain that *out-penalty*(y) plus the length of the subpath of P from v to y is at least as large as the penalty reach of (v, w) in G_t.

Shortcut step, first part. Next we establish the first part of the induction claim for the case where step t is an addition of a shortcut (u, w) bypassing a vertex v. Assume that (u, w) is a one-way shortcut (the proof for a two-way shortcut is analogous). In this case G'_{t+1} is obtained from G'_t by adding (u, w). Consider an arc (x, y) eliminated before step t and let P' be a canonical path in G'_{t+1} through (x, y). Then either P' is a canonical path in G'_t or P' contains (u, w). In the latter case, the path obtained from P' by replacing (u, w) by the concatenation of (u, v) and (v, w) is a canonical path in G'_t. Therefore the reach of (x, y) in G'_{t+1} is no greater than its reach in G'_t, and the upper bound remains valid.

To completely establish the first part of the induction hypothesis we also have to prove that the upper bounds on the reaches of the bypassed (and eliminated) arcs, (u, v) and (v, w), are correct in G'_{t+1}. Consider one such arc, say (v, w). (The proof for (u, v) is analogous.) Every canonical path P' going through (v, w) in G'_{t+1} does not go through (u, w), and therefore P' is also a canonical path in G'_t. (Note that P' cannot contain (u, v) either.) Let P be the maximal subpath of P' in G_t containing (v, w). By the induction hypothesis the penalty reach of (v, w) with respect to P in G_t upper bounds the reach of (v, w) with respect to P' in G'_t. Since (u, v) is the only incoming arc into v in G_t, and P' does not contain (u, v), the penalty reach of (v, w) with respect to P in G_t is bounded by *in-penalty*$(v)+\ell(v, w)$.

Shortcut step, second part. To prove the second part of the induction hypothesis, consider a shortcut (u, w) added at step t, an arc (g, h) in G_{t+1}, and a canonical path P' in G'_{t+1} containing (g, h). If P' does not contain (u, w), (u, v), or (v, w), then P' is also a canonical path in G'_t. Furthermore, the maximal subpath P of P' that contains (g, h) in G_{t+1} is the same as the maximal subpath containing (g, h) in G_t. The penalty reach of (g, h) with respect to P is also the same in G_t and in G_{t+1}. So the penalty reach of (g, h) with respect to P in G_{t+1} upper bounds the reach of (g, h) with respect to P' in G'_{t+1} by the induction hypothesis.

Suppose that P' contains (v, w). (The proof in the case that P' contains (u, v) is analogous.) Then P' cannot contain (u, w) or (u, v). The subpath P of P' containing (g, h) in G_t starts at v, and the subpath P of P' containing (g, h) in G_{t+1} starts at w. Our upper bound on the reach of (v, w) plus the length of the subpath of P from w to h is at least as large as the penalty reach of (g, h) with respect to P in G_t. Since the in-penalty of w in G_{t+1} is not smaller than our upper bound on the reach of (v, w), the induction statement holds with respect to P' and (g, h) in G'_{t+1}.

Now suppose that P' contains (u, w). Such a path was not a canonical path in G'_t. However there was a canonical path $\overline{P}'$ in G'_t identical to P' except that it had the arcs (u, v) and (v, w) in place of the arc (u, w) in P'. Let P be the maximal subpath of P' containing (g, h) in G_{t+1} and let $\overline{P}$ be maximal subpath of $\overline{P}'$ containing (g, h) in G_t. Again P and $\overline{P}$ are identical except that if P contains (u, w) then $\overline{P}$ contains (u, v) and (v, w) instead. If $(g, h) \neq (u, w)$, then the reach of (g, h) in G'_{t+1} with respect to P' is the same as the reach of (g, h) in G'_t with respect to $\overline{P}'$. Similarly the penalty reach of (g, h) in G_{t+1} with respect to P is the same as the penalty reach of (g, h) in G_t with respect to $\overline{P}$, and the inductive claim for P' and P follows from the induction hypothesis applied to $\overline{P}'$ and $\overline{P}$.

The last case to consider is when $(g, h) = (u, w)$. Assume that the reach of (u, w) with respect to P' is the length of the suffix of P' from u. (The other case is analogous.) Then the reach of (v, w) with respect to $\overline{P}'$ in G'_t is the length of the suffix of $\overline{P}'$ from v. Since by the induction hypothesis the penalty reach of (v, w) with respect to $\overline{P}$ in G_t is as large as the reach of (v, w) with respect to $\overline{P}'$ in G'_t, we have

1. The length of the suffix of $\overline{P}$ from v plus the out-penalty of the last vertex on $\overline{P}$ is at least as large as the length of the suffix of $\overline{P}'$ from v.

2. The length of the prefix of $\overline{P}$ until w plus the in-penalty of the first vertex on $\overline{P}$ is at least as large as the length of the the suffix of $\overline{P}'$ from v.

It is easy to see that these two facts imply that the reach of (u, w) with respect to P' in G'_{t+1} is upper bounded by the penalty reach of (u, w) with respect to P in G_{t+1}.

To complete the proof we argue that the procedure that converts arc reaches into vertex reaches is correct and that the refinement algorithm is correct. The former is straight-forward to verify. One can prove the latter by showing that penalty reaches of the vertices in the graph to which we apply the refinement step are upper bounds on their reaches in G'. □

We stress that the preprocessing phase of the algorithm is completely independent of the query phase: the only information passed from the former to the latter is the graph with shortcuts and the reach values. For example, the preprocessing algorithm does not pass on perturbations introduced to make shortest paths unique. Correctness of our query algorithm follows from Theorem 5.1.

5.6 Parameters

Choosing ϵ_1. Our goal in choosing ϵ_1 is to make the first iteration take roughly as much time as the iterations that follow. A large value will make the first iteration comparatively slow and will not give the algorithm the chance to add shortcuts when needed. On the other hand, a very small ϵ_1 will introduce penalties too early, which will make reaches computed in subsequent iterations more inaccurate.

The choice of ϵ_1 is made as follows. Pick k vertices at random, for some parameter k. For each vertex, grow a partial shortest path tree with exactly $\lfloor n/k \rfloor$ scanned vertices and take note of its radius (given by the distance label of the last scanned vertex). Set ϵ_1 to be twice the minimum of all k radii. We used $k = \min\{500, \lfloor \lceil \sqrt{n} \rceil / 3 \rfloor\}$.

Note that the density of road networks tends to vary a lot: a ball with radius R in New York City will have many more vertices than one in rural North Dakota. The above heuristic identifies dense regions of the graph and chooses ϵ_1 so that growing partial shortest path trees in dense areas is not too expensive.

Choosing α. Once we pick the appropriate value of ϵ_1, we also have to choose a multiplier α to determine the thresholds to be used in the remaining iterations (recall that $\epsilon_i = \alpha^{i-1}\epsilon_1$). The choice of α is related to three important aspects of the algorithm. First, it affects the running time: the smaller α is, the more iterations will be performed by the algorithm; on the other hand, if α is too large, iterations will typically take longer (since vertices are eliminated less frequently). Second, the choice of α helps determine how many shortcuts are added: if α is relatively small, the algorithm will have a better chance of shortcutting vertices before they are eliminated. Finally, and most importantly, the choice of α determines how good the upper bounds will be. The error in an arc reach estimated during iteration i depends on the penalties, which in turn depend on the maximum reaches of arcs eliminated in previous iterations. The larger α is, the smaller the summation $\sum_{j<i} \epsilon_j$ will be with respect to ϵ_i.

Empirically, we have determined that using $\alpha = 3.0$ provides a good balance between these conflicting factors. However, as soon as we reach an iteration in the main phase where the number of vertices is smaller than δ (the parameter defined in Section 5.4), we reduce the multiplier from 3.0 to 1.5. This gives the algorithm the opportunity to add more shortcuts. Although this increases the running time, it does so by a small factor (assuming $\delta \ll n$). Note that the reaches obtained during the main phase for the last δ vertices may not be as good as if the multiplier were higher; since those reaches will be recomputed during the refinement phase, this is not a problem.

6 Reach and the ALT Algorithm

6.1 A^* Search and ALT Algorithms

Suppose we need to find shortest paths on a graph G with distance function ℓ. A *potential function* is a function from 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* [6, 17] is an algorithm that works like Dijkstra's algorithm, except that at each step it selects a labeled vertex v with the smallest *key*, defined as $k_f(v) = d_f(v) + \pi_f(v)$, to scan next. It is easy to see that A^* search is equivalent to Dijkstra's algorithm on the graph with length function ℓ_{π_f}. If π_f is such that ℓ_π is nonnegative for all arcs (i.e., if π_f is *feasible*), the algorithm will find the correct shortest paths. We refer to the class of A^* search algorithms that use a feasible function π_f with $\pi_f(t) = 0$ as *lower-bounding algorithms.*

Intuitively, better estimates lead to fewer vertices being scanned. More precisely, consider an instance of the P2P problem and let π_f and π'_f be two feasible potential functions such that $\pi_f(t) = \pi'_f(t) = 0$ and, for any vertex v, $\pi'_f(v) \geq \pi_f(v)$ (i.e., π'_f dominates π_f). If ties are broken consistently when selecting the next vertex to scan, the following holds.

Theorem 6.1 [13] *The set of vertices scanned by A^* search using π'_f is contained in the set of vertices scanned by A^* search using π_f.*

Note that the theorem implies that any lower-bounding algorithm with a nonnegative potential function visits no more vertices than Dijkstra's algorithm, which is equivalent to the lower-bounding algorithm with the zero potential function.

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

If π_f and π_r are not consistent, the forward and reverse searches use different length functions. When the searches meet, we have no guarantee that the shortest path has been found. Assume π_f and π_r give lower bounds to the sink and from the source, respectively. We use the *average function* suggested by Ikeda et al. [18], defined as $p_f(v) = \frac{\pi_f(v) - \pi_r(v)}{2}$ for the forward computation and $p_r(v) = \frac{\pi_r(v) - \pi_f(v)}{2} = -p_f(v)$ for the reverse one. Although p_f and p_r usually do not give lower bounds as good as the original ones, they are feasible and consistent.

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

Finding good landmarks is important for the overall performance of ALT algorithms. Our implementation uses the *maxcover* heuristic discussed in [15]. In our tests, we always picked 16 landmarks in total, unless mentioned otherwise.

For a given *s*-*t* pair, some of the landmarks give good lower bounds on distances while others do not. Our ALT implementation uses dynamic selection of active landmarks, i.e., it picks a only a subset of the landmarks to be used during each specific s-t search (initially two, but up to six if necessary). See [15] for details.

6.2 Reach and A^* search

Reach-based pruning can be easily combined with A^* search. Gutman [16] noticed this in the context of unidirectional search. The general approach is to run A^* search and prune vertices (or arcs) based on reach conditions. Specifically, when A^* is about to scan a vertex v we extract the length of the shortest path from the source to v from the key of v (recall that for the unidirectional algorithm, $k_f(v) = d_f(v) + p_f(v)$). Furthermore, $\pi_f(v)$ is a lower bound on the distance from v to the destination. If the reach of v is smaller than both $d_f(v)$ and $\pi_f(v)$, we prune the search at v.

The reason why reach-based pruning works is that, although A^* search uses transformed lengths, the shortest paths remain invariant. This applies to bidirectional search as well. In this case, we use $d_f(v)$ and $\pi_f(v)$ to prune in the forward direction, and $d_r(v)$ and $\pi_r(v)$ to prune in the reverse direction. Using pruning by reach does not affect the stopping condition of the algorithm. We still use the usual condition for A^* search, which is similar to that of the standard bidirectional Dijkstra, but with respect to reduced costs (see [15]). We call our implementation of the bidirectional A^* search algorithm with landmarks and reach-based pruning REAL. As for ALT, we used a version of REAL that balances the work of the forward search and the reverse search.

Note that we cannot use implicit bounds with A^* search. The implicit bound based on the radius of the ball searched in the opposite direction does not apply because the ball is in the transformed space. The self-bounding algorithm cannot be combined with A^* search in a useful way, because it assumes that the two searches will process balls of radius equal to half of the s-t distance. However, processing these balls defeats the purpose of A^* search, which aims at processing a smaller set.

The main gain in the performance of A^* search comes from the fact that it directs the two searches towards their goals, reducing the search space. Reach-based pruning sparsifies search regions, and this sparsification is effective for regions searched by both Dijkstra's algorithm and A^* search.

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 from all vertices to it). These two procedures are independent from each other: since shortcuts do not change distances, landmarks can be generated regardless of what shortcuts are added. Furthermore, the query is still independent of the preprocessing algorithm: the query only takes as input the graph with shortcuts, the reach values, and the distances to and from landmarks.

6.3 Further Optimizations

Our implementation of REAL uses early pruning: when scanning an arc (v, w), we try to prune w by reach before actually inserting it into the queue (see Section 4.2). Note that the test will require a lower bound $L(w)$ on the distance from w to the destination. Since obtaining this value is relatively expensive (it requires retrieving several distances to and from landmarks), we first use $L(v) - \ell(v, w)$ as the lower bound. More precisely, we test if (1) $d(v) + \ell(v, w) > \overline{r}(w)$ and (2) $L(v) - \ell(v, w) > \overline{r}(w)$. If so, we prune. This test is weaker, but cheaper (since at this point we know $L(v)$). We compute $L(w)$ only if the test fails to prune the vertex.

Our implementation also uses arc sorting. However, we must use a more sophisticated (and somewhat weaker) version of the algorithm presented in Section 4.2. Instead of sorting the outgoing arcs of (v, w) by $\overline{r}(w)$, we sort by $\overline{r}(w) + \ell(v, w)$ (in non-increasing order). Suppose that, while scanning v, we find an arc (v, w) such that $(1')$ $\overline{r}(w) + \ell(v, w) < d(v)$ and $(2')$ $\overline{r}(w) + \ell(v, w) < L(v)$. 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 scanning it). This is still useful because the lower bound on the distance to the target may have changed since the vertex was inserted, due to the possibility of new landmarks being activated.

7 Alternative Reach Definitions and Related Work

7.1 Gutman's Algorithm

In [16] Gutman computes shortest routes with respect to travel times. However, his algorithm, which is unidirectional, uses Euclidean bounds on travel *distances*, not times. This requires a more general definition of reach, which involves, in addition to the metric induced by graph distances (*native metric*), another metric M, which can be different. To define reach, one considers native shortest paths, but takes subpath lengths and computes reach values for M-distances. It is easy to see how these reaches can be used for pruning. Note that Gutman's algorithm can benefit from shortcuts, although he does not use them. All our algorithms have natural distance bounds for the native metric, so we use this metric as M.

Other major differences between RE and Gutman's algorithm are as follows. First, RE is bidirectional, and bidirectional shortest path algorithms tend to scan fewer vertices than unidirectional ones. Second, RE uses implicit lower bounds and thus does not need the vertex coordinates required by Gutman's algorithm. Finally, RE preprocessing creates shortcuts, which Gutman's algorithm does not. There are also minor differences in the preprocessing algorithm, which have less effect on performance. In particular, we do not grow partial trees from eliminated vertices, which requires a slightly different interpretation of penalties.

A variant of Gutman's algorithm uses A^* search with Euclidean lower bounds. In addition to the differences mentioned in the previous paragraph, REAL differs in using tighter landmark-based lower bounds.

7.2 Cardinality Reach and Highway Hierarchies

We now discuss the relationship between our reach-based algorithm (RE) and the HH algorithm of Sanders and Schultes. Although the HH algorithm is stated in terms of arc pruning, we outline a vertex-pruning variant, which is simpler, and refer the reader to [29] for the original version. Since HH is described for undirected graphs, we restrict the discussion to them. Most of what we say applies to the directed case as well.

We introduce the variant of reach, *c-reach* (cardinality reach). Given a vertex v on a shortest path P, grow equal-cardinality balls centered at its endpoints until v belongs to one of the balls. Let c_P be the cardinality of each of the balls. The c-reach of v, $c(v)$, is the maximum, over all shortest paths P, of c_P. 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 $\rho(t, c(v)) < \text{dist}(v, t)$. Intuitively, if v has low c-reach, it must be visited very early in any s-t search; if the search grows past a certain radius and v has not been visited yet, we know for sure v will not be on the shortest path. 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). The remaining ones are highway 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. In other words, after an initial local search, we can restrict ourselves to the subgraph induced by the highway vertices.

This idea is applied recursively to the graph with low-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 HH algorithm preprocessing phase also shortcuts lines and uses other heuristics to reduce the graph size at each iteration. On road networks, the graph shrinks substantially at each level, and the total number of levels is very small.

An important property of the HH query algorithm, which makes it similar to the self-bounding algorithm discussed in Section 4.1, is that the search in a given direction never goes to a lower level of the hierarchy. The intuition is that if the actual shortest path does go through the lower level, the remaining portion of the path will be found by the search in the opposite direction.

Our partial description of HH brings out its similarity to the reach-based algorithms. We have omitted the details of HH which have no equivalent in RE, which has a simpler query algorithm. For example, if a search from s advances to the level of hierarchy where s is not present, then at this level $\rho(s, h)$ is not defined. The query algorithm needs to deal with such cases, making it more complex. See [27, 29] for details.

Note that, like the self-bounding algorithm, HH cannot be combined with A^* search in the usual way.

The biggest difference between HH and our approach is the use of c-reach instead of reach. This requires HH to use ρ values. An efficient way to maintain these values is implicitly, with the help of the highway hierarchy. A potential advantage of c-reaches is that the size of a ball to be searched before advancing to the next level of the hierarchy is explicitly bounded. The disadvantage is the added complexity and the loss of flexibility, which in particular prevents a natural combination with A^* search.

8 Experimental Results

Our computational experiments have several goals. First, we study the relative performance of two previous algorithms (ALT and B) and our new algorithms (RE and REAL) in the context of road networks. Note that B, the bidirectional Dijkstra's algorithm, is a natural candidate for comparison as it is simple to implement and the most robust algorithm for general graphs with no preprocessing. This comparison also shows how much A^* search and landmarks improve the performance of RE. We also make a comparison with HH, the best previous algorithm.

Then we study how much shortcuts help improve the performance of both preprocessing and queries, and also how the use of upper bounds on reaches (instead of exact values) affects the algorithms. Next, we assess how the number of landmarks affects the performance of REAL. Finally, we briefly study how the algorithms fare on grid graphs, which have no natural hierarchy.

8.1 Experimental Setup

We implemented our algorithms in C++ and compiled them with Microsoft Visual C++ 7.0. All tests were performed on an AMD Opteron with 16 GB of RAM running Microsoft Windows Server 2003 at 2.4 GHz.

We conduct most of our tests on road networks. We test our algorithm on the 13 graphs described in Table 1. The first graph in the table, North America (NA), was extracted from `Mappoint.NET` data and represents Canada, the United States (including Alaska), and the main roads of Mexico. The other 12 instances are representative subgraphs of NA. All graphs are directed and biconnected. For each of these graphs, we use one of two types of length function: travel times and travel distances.

For a direct comparison with HH, we use the graph of the United States built by Sanders and Schultes [27] based on Tiger-Line data [35]. Because our implementations of ALT and REAL

Table 1: Road Networks

NAME	DESCRIPTION	VERTICES	ARCS	LATITUDE (N)	LONGITUDE (W)
NA	North America	29 883 886	70 297 895	$[-\infty, +\infty]$	$[-\infty, +\infty]$
CTR	Central USA	16 797 663	39 499 505	[25.0; 50.0]	[79.0; 100.0]
W	Western USA	8 434 514	19 730 764	[27.0; 50.0]	[100.0; 130.0]
E	Eastern USA	4 256 990	10 088 732	[24.0; 50.0]	$[-\infty; 79.0]$
LKS	Great Lakes	3 499 752	8 439 615	[41.0; 50.0]	[74.0; 93.0]
CAL	California and Nevada	2 134 828	5 031 072	[32.5; 42.0]	[114.0; 125.0]
NW	Northwest USA	1 649 045	3 778 225	[42.0; 50.0]	[116.0; 126.0]
NE	Northeast USA	1 575 447	3 841 303	[39.5, 43.0]	$[-\infty; 76.0]$
FLA	Florida	1 228 116	2 999 092	[24.0; 31.0]	[79; 87.5]
COL	Colorado	585 950	1 396 345	[37.0; 41.0]	[102.0; 109.0]
CAN	Western Canada	513 914	1 288 194	$[49.0; +\infty]$	$[39; +\infty]$
BAY	Bay Area	330 024	793 681	[37.0; 39.0]	[121; 123]
NYC	New York City	277 863	697 641	[40.3; 41.3]	[73.5; 74.5]

assume the graph to be connected,[6] we only take the largest connected component of this graph, which contains more than 98.6% of the vertices. The graph is undirected, and we replace each edge $\{v, w\}$ by arcs (v, w) and (w, v). Our version of the graph (which we call USA) has 23 947 347 vertices and 57 708 624 arcs.

We also do limited experiments with grid graphs. Vertices of an $x \times y$ grid graph correspond to points on a two-dimensional grid with coordinates i, j for $0 \leq i < x$ and $0 \leq j < y$. Each vertex has arcs to the vertices to its left, right, up, and down neighbors, if present. Arc lengths are integers chosen uniformly at random from $[1, 1024]$. We use square grids (i.e., $x = y$).

Choice of parameters. Unless otherwise noted, in each experiment we run the algorithms with a fixed set of parameters.

For each graph, we generated one set of 16 *maxcover* landmarks (as described in [15]). The same set was used both by ALT and REAL. Upper bounds on reaches were generated with the algorithm described in Section 5. The reaches thus obtained (alongside with the corresponding shortcuts) were used by both RE and REAL.

At query time, both ALT and REAL use dynamic selection of active landmarks, with up to six landmarks used in any particular search (again, as suggested in [15]). RE and REAL use both early pruning and arc sorting.

[6]We assume the graphs are connected only to simplify implementation.

8.2 Random Queries on Road Networks

Table 2 presents the results obtained by our algorithms when applied to the `Mappoint.NET` graphs with the travel-time metric. For each graph and each algorithm, we show performance information related to preprocessing and to random queries.

First, we show the total time (in minutes) required by the preprocessing procedure; for ALT, this corresponds to landmark generation; for RE, reach computation; and for REAL, both.

Then we present the total space required by the preprocessed data (in megabytes). That includes the graph itself (for all three algorithms), the reach information (for RE and REAL), and the landmark information (for ALT and REAL). Recall that ALT uses the original graph, whereas the other algorithms use a version of the graph containing shortcuts, which is larger. The total space also accounts for the size of the *translation map* used to convert shortcuts into their constituent arcs. As in [27], our query times do not include the time to actually perform this conversion, since this step may not be necessary in some applications.

The last six columns present query data. For each graph, we picked 1000 pairs of vertices uniformly at random and ran each algorithm on them. We show the average number of vertices scanned, the maximum number of vertices scanned (among the 1000 searches), and the average running time per search. In all cases, we show both the absolute values (times are in milliseconds) and the *speedup* with respect to our implementation of the standard bidirectional Dijkstra's algorithm (B). The speedup is the ratio between the result for B and the result for the algorithm being tested.

Comparing our algorithms with B is important to provide a baseline and because B is relatively stable: on these graphs, B visits between 31% and 43% of the vertices on average, and between 69% and 96% in the worst case. The lower bounds in both ranges are achieved on NYC, and the upper bounds on LKS.

We start the analysis of the results by considering the average behavior of our query algorithms, in terms of both running time and number of vertices scanned. The table suggests that ALT has worse asymptotic performance than the other algorithms. Although competitive with RE on smaller graphs, ALT is roughly 20 times worse on the largest one according to both measures. In [15], it was observed that, on road networks, ALT visits a constant fraction of the vertices on average, regardless of graph size. This is indeed what we observe here: roughly 1% of the vertices are visited in all cases. Since the performance of B is largely independent of graph size, so is the speedup.

Contrast this with RE: the average fraction of vertices it scans tends to decrease with graph size. On the largest graph, RE scans about 6% as many vertices as ALT. REAL is consistently faster than RE on average, but the two algorithms seem to be asymptotically very close to each other: on all graphs, RE scanned 5 to 7 times as many vertices as REAL, and the average time it takes is larger by a factor of 3 to 4 on most graphs. This indicates that REAL does more work per vertex than RE.

Table 2: Algorithm performance on road networks with travel times as arc lengths: total preprocessing time, total space in disk required by the preprocessed data (in megabytes), average number of vertices scanned per query (over 1000 random queries), maximum number of vertices scanned (over the same queries), and average running times. Query data shown as both absolute values and as speedups with respect to the bidirectional Dijkstra algorithm.

		PREP.	DISK	QUERY					
		TIME	SPACE	AVG SCANS		MAX SCANS		AVG TIME	
GRAPH	METHOD	(min)	(MB)	COUNT	SPD	COUNT	SPD	msec	SPD
NYC	ALT	0.7	22	2 557	32	28 279	7	2.77	14
	RE	6.3	16	2 588	32	4 977	39	2.03	19
	REAL	7.0	34	316	263	1 411	136	0.48	82
BAY	ALT	0.7	26	4 052	29	54 818	5	3.39	16
	RE	3.2	19	1 590	74	3 438	85	1.17	48
	REAL	3.9	40	290	404	1 691	172	0.45	123
CAN	ALT	1.3	42	5 975	36	61 367	8	4.28	21
	RE	4.8	32	2 457	88	6 306	74	1.91	47
	REAL	6.1	64	281	768	1 850	253	0.50	177
COL	ALT	1.6	47	7 373	26	85 246	6	5.84	15
	RE	5.2	36	2 181	88	5 074	103	1.80	49
	REAL	6.9	73	306	624	1 612	324	0.59	149
FLA	ALT	3.4	99	15 565	30	200 818	6	15.52	16
	RE	11.0	73	2 158	215	4 389	253	1.81	134
	REAL	14.4	151	392	1 183	2 207	504	0.72	337
NE	ALT	4.2	127	14 699	39	158 867	9	17.39	20
	RE	32.9	95	5 372	108	10 610	135	4.80	71
	REAL	37.1	193	648	895	3 735	384	1.20	283
NW	ALT	3.9	132	14 178	36	144 082	8	12.52	21
	RE	17.5	100	2 804	184	5 877	203	2.39	112
	REAL	21.4	204	367	1 408	1 513	789	0.73	365
CAL	ALT	5.2	171	24 603	34	307 596	6	24.19	20
	RE	23.3	129	3 159	268	6 960	283	3.03	158
	REAL	28.5	263	518	1 632	3 677	536	0.83	577
LKS	ALT	10.9	282	46 130	33	551 193	6	52.74	14
	RE	61.1	217	5 991	252	9 714	346	5.83	130
	REAL	72.0	437	791	1 911	3 800	885	1.52	501
E	ALT	15.2	342	35 044	42	487 194	8	44.47	18
	RE	84.7	255	6 925	212	13 857	277	7.06	116
	REAL	99.9	523	795	1 843	4 543	844	1.61	510
W	ALT	26.7	677	97 073	31	1 196 364	6	111.27	15
	RE	124.7	526	6 222	480	12 112	586	6.75	252
	REAL	151.5	1 058	915	3 262	5 382	1 318	1.84	922
CTR	ALT	55.5	1 346	147 040	37	2 376 017	5	224.65	16
	RE	400.9	1 038	11 721	464	20 319	630	13.27	274
	REAL	456.4	2 094	1 275	4 263	6 594	1 940	2.84	1 276
NA	ALT	95.3	2 398	250 381	41	3 584 377	8	393.41	19
	RE	678.8	1 844	14 684	698	24 618	1 104	17.38	439
	REAL	774.2	3 726	1 595	6 430	7 450	3 647	3.67	2 080

Indeed, if one takes the ratio between the speedup in terms of number of vertices scanned and the time speedup, one gets an estimate on the relative work the algorithm performs per vertex scan (compared to B). For RE, the work is between 1.5 and 2.0 times greater than for B. For ALT, the ratio is usually greater than 2.0. For REAL, it is closer to 3.0.

When comparing the maximum number of vertices scanned, RE and REAL exhibit a consistent behavior, with REAL being 2 to 3 times better on all graphs. Note that this difference is smaller than in the average case. ALT is significantly worse—it is the only algorithm where the worst case grows linearly with graph size: bad cases make ALT visit 10% to 15% of the vertices. The other algorithms are strongly sublinear in this measure: a bad case on NA is only five times worse than a bad case on NYC, a graph 100 times smaller.

Although query times generally increase with graph size, it is not a perfect predictor of how well each algorithm will fare, particularly for RE. Take NE and NW, for instance. Both graphs have roughly 1.6 million vertices, but RE is almost twice as fast on NW; ALT and REAL are also faster on NW, but by a smaller margin. We conjecture the difference is due to the fact that NW has a more natural highway hierarchy than NE. Similarly, NYC, the smallest graph in our test set, is harder for RE than any of the next four graphs—including FLA, which has more than four times as many vertices.

In terms of preprocessing, we note that computing landmarks is significantly faster than finding good upper bounds on reaches. However, landmark data (with a reasonable number of landmarks) takes up more space than reach data; compare the space usage of RE and ALT. In fact, the reaches themselves are a minor part (less than 20%) of the total space required by RE. The rest of the space is used up by the graph with shortcuts (typically, the number of arcs increases by 35% to 55%) and by the shortcut translation map.

Travel distances. Table 3 is similar to Table 2, but with travel distances (as opposed to travel times) as arc lengths. The most obvious difference in the query results is that ALT, RE, and REAL have worse performance in this case, on all measures. All these algorithms benefit from the existence of a natural hierarchy in the underlying network, and such a hierarchy is much more pronounced when travel times are used: if one considers distances only, a local road will look just as good as the major freeway it runs parallel to.

However, not all algorithms are hit equally hard by the change in metric. For large graphs, the average performance of ALT (measured by both running time and vertices scanned) becomes roughly 20% worse, and the worst-case performance is not much affected at all. On the other hand, it is not uncommon for RE to become more than twice as slow when travel distances are used. REAL falls somewhere in between.

In terms of preprocessing time, once again the algorithms have different behavior. While landmark computation remains largely unaffected, reach computation becomes significantly worse: up to 2.5 times slower on large graphs. The total space used does not change much in either case.

Table 3: Algorithm performance on road networks with distances as arc lengths. For each graph and each method, we first show the total time spent in preprocessing (in minutes) and the total size of the data stored on disk after preprocessing (including the graph itself). Then we present the results for queries on 1000 random graphs: average number of vertices scanned per query, the maximum number of vertices scanned (over all queries ran), and the average running time. In each case, we show both the actual value and the speedup (SPD) with respect to B.

		PREP.	DISK	QUERY					
		TIME	SPACE	AVG SCANS		MAX SCANS		AVG TIME	
GRAPH	METHOD	(min)	(MB)	COUNT	SPD	COUNT	SPD	msec	SPD
NYC	ALT	0.8	23	2 959	29	27 070	7	3.25	14
	RE	12.5	17	4 945	17	9 317	21	3.72	12
	REAL	13.3	35	454	189	2 382	82	0.80	58
BAY	ALT	0.8	27	3 383	35	42 192	7	3.25	18
	RE	4.6	19	2 761	43	6 313	45	2.05	28
	REAL	5.4	41	335	356	2 717	105	0.45	128
CAN	ALT	1.5	44	7 841	25	89 602	5	5.89	15
	RE	8.1	32	3 566	55	9 216	52	2.86	30
	REAL	9.6	67	390	505	2 685	180	0.73	117
COL	ALT	1.8	48	7 793	24	126 755	4	6.34	14
	RE	9.7	36	3 792	50	10 067	50	3.16	28
	REAL	11.5	75	406	469	2 805	178	0.72	123
FLA	ALT	3.7	102	11 203	42	130 532	9	11.06	21
	RE	16.6	74	3 544	133	8 248	137	2.88	80
	REAL	20.3	154	440	1 074	3 113	362	0.78	296
NE	ALT	4.3	132	14 334	41	114 846	13	17.08	21
	RE	60.8	96	10 460	57	22 561	64	9.28	38
	REAL	65.1	199	819	726	4 496	322	1.22	289
NW	ALT	4.2	136	20 662	26	426 069	3	21.61	12
	RE	21.3	101	4 217	125	10 630	121	3.81	71
	REAL	25.4	208	478	1 103	3 058	419	0.89	302
CAL	ALT	5.5	177	31 033	28	476 091	4	40.31	12
	RE	38.8	130	5 802	149	16 184	120	5.19	91
	REAL	44.3	270	762	1 136	5 532	352	1.33	356
LKS	ALT	10.7	292	48 417	31	472 060	7	57.99	15
	RE	145.1	221	13 060	115	24 725	136	12.02	71
	REAL	155.8	451	1 201	1 254	7 371	458	2.16	393
E	ALT	14.6	353	43 737	35	582 663	7	61.98	15
	RE	158.9	258	14 025	108	28 144	141	13.28	69
	REAL	173.4	537	1 142	1 323	7 097	560	2.27	404
W	ALT	25.6	700	91 697	33	748 710	10	117.42	17
	RE	195.9	531	10 385	289	24 065	298	10.61	186
	REAL	221.6	1 086	1 235	2 431	7 703	931	2.36	834
CTR	ALT	60.7	1 395	167 753	33	2 351 001	6	251.34	16
	RE	977.0	1 051	26 549	206	50 127	258	28.52	144
	REAL	1 037.7	2 156	2 160	2 537	11 771	1 100	4.66	883
NA	ALT	97.2	2 511	292 777	36	3 588 684	8	476.86	17
	RE	1 623.0	1 866	30 962	336	56 794	485	34.92	231
	REAL	1 720.2	3 860	2 653	3 922	17 527	1 570	5.97	1 351

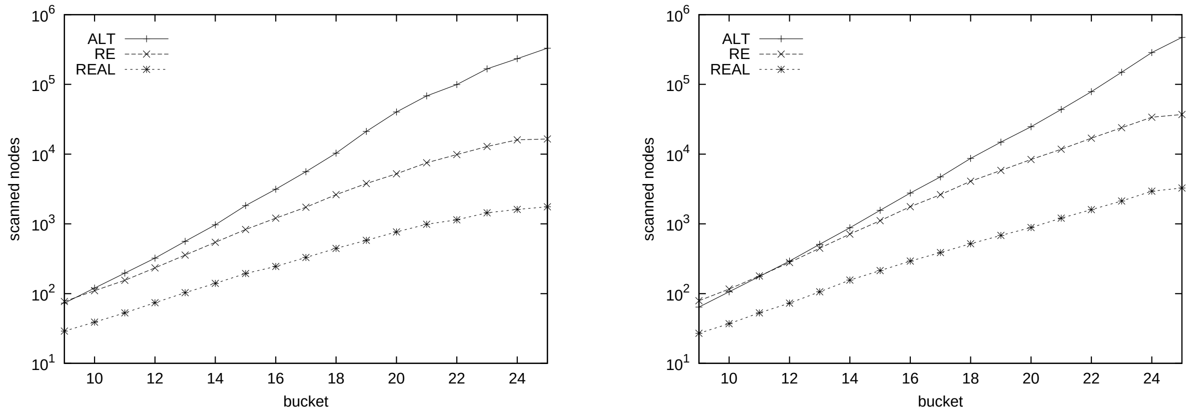

Figure 5: Average number of scanned vertices for local queries on NA with travel times (left) and travel distances (right). The horizontal axis corresponds to ranges of ranks. Position i represents s-t pairs such that s is chosen at random and t is the j-th farthest vertex from s, where j is selected uniformly at random from the range $(2^{i-1}, 2^i]$. The vertical axis is in log scale.

8.3 Local Queries on Road Networks

So far, we have tested the algorithms only on random s-t pairs. In this distribution, source and destination are on average very far from each other. In practice, it is reasonable to expect queries to be more local in nature. To evaluate the behavior of the algorithms for local queries, we use a slight modification of the "local" s-t pair generation process introduced in [27] and apply it to the NA graph.

To describe the process, we need the following definition. Given a vertex s, sort all vertices with respect to their distance from s, in non-decreasing order. Define $\text{rank}_s(v)$ to be the index of v in the resulting sequence. For a parameter i, we generate 1000 s-t pairs by (1) picking s uniformly at random, (2) picking j uniformly at random from $(2^{i-1}, 2^i]$, and (3) selecting t so that $\text{rank}_s(t) = j$. We do this for $i \in \{9, 10, \ldots, 25 = \lceil \log_2 n \rceil\}$, where n is the number of vertices in the NA graph. We then group the pairs thus generated into *buckets*: all pairs generated with parameter i are assigned to bucket i.

Figure 5 shows the average number of vertices scanned when each query algorithm was run on each bucket. The first plot uses travel times and the second uses travel distances. Note that the vertical axis is in logarithmic scale. In both plots, the asymptotic behavior of ALT is clearly worse than that of RE. For pairs of vertices that are far apart, ALT scans around 100 times as many vertices as RE (slightly more than that with travel times, slightly less with travel distances). Note, however, that ALT scans fewer vertices than RE when s and t are very close to each other. The crossover point is larger with travel distances. Not surprisingly, REAL is better than both algorithms for all sizes, and its advantage over RE even increases as pairs get farther.

A comparison between these plots and Tables 2 and 3 shows that all three algorithms have average performance on the uniform distribution that is similar to that of bucket 24, which confirms that the uniform distribution is indeed biased towards pairs of vertices that are far from each other.

8.4 Comparison to Highway Hierarchies

To compare our algorithms with HH, we use the USA graph. Unlike the other graphs we tested, USA is undirected. However, our algorithms do not exploit this fact explicitly: they simply interpret each (undirected) edge as two arcs of equal length in opposite directions. We are interested in the more general case of directed graphs, so we have not investigated how much improvement one can get on undirected graphs.

The HH algorithm, however, does use the fact that the graph is undirected. This makes its implementation simpler, and maybe more efficient. When discussing its performance, for both preprocessing and query, we assume that the performance does not get much worse when the algorithm is extended to directed graphs. We believe this is true, but this assumption has not been verified.

For HH, we use the results reported in [27]. Averages for this algorithm are taken over 10 000 executions (we use 1000 for our methods). Running times for HH were obtained on a machine slightly slower than ours: an AMD Opteron running at 2.2 GHz (ours is an AMD Opteron running at 2.4 GHz). The machine used the Linux operating system (ours uses Windows).

Table 4 compares the performance of the algorithms when executing random queries on the USA graph. The table shows data for both travel times and travel distances. Five measures of performance are presented: preprocessing time, size of preprocessed data, average number of vertices scanned, maximum number of vertices scanned, and average query time. All query values are given in absolute terms and as a speedup relative to the bidirectional Dijkstra algorithm. The maximum number of vertices scanned reported for HH is an upper bound obtained from unidirectional queries; the actual worst pair from the sample was not reported in HH. The total size of the preprocessed data was not reported in [27] either. We got the number from the authors [28], who also mentioned that it can be improved to 1309 MB with little effect on query times. The results for HH with travel distances are not presented in [27], and therefore are omitted from the table.

First consider the results with the travel-time metric. In terms of average number of vertices visited, RE and HH have remarkably similar performance. The average running time for RE is lower, which we attribute to the fact that RE is a simpler algorithm and its implementation has smaller constants. The maximum number of vertices visited, on the other hand, is higher for RE, but not by much. The preprocessing time of HH is about 30% smaller than that of RE, but the total space usage is similar.

As with the other road networks, REAL is better than RE by over a factor of two for all performance measures, and ALT is significantly worse.

Table 4: Results for the undirected USA graph: preprocessing time, preprocessing space, average number of vertices scanned, maximum number of vertices scanned, and average query time (for the last three, we present both absolute values and speedups with respect to B). Data for HH with travel distances is not available; for travel times, maximum number of vertices scanned is an upper bound. Running times for HH were obtained on a slightly slower machine.

		PREP.	DISK	QUERY					
		TIME	SPACE	AVG SCANS		MAX SCANS		AVG TIME	
METRIC	METHOD	(min)	(MB)	COUNT	SPD	COUNT	SPD	ms	SPD
TIMES	ALT	92.7	1 984	177 028	44	2 587 562	8	322.78	21
	RE	365.9	1 476	3 851	2 000	8 722	2 330	4.50	1 475
	REAL	458.5	3 038	891	8 646	3 667	5 541	1.84	3 601
	HH	≈ 258.0	1 457	3 912	1 969	$\leq 8\,678$	$\geq 2\,341$	≈ 7.08	≈ 937
DIST.	ALT	99.9	1 959	256 507	33	2 674 150	8	392.84	15
	RE	981.5	1 503	22 377	376	44 130	500	25.59	236
	REAL	1 081.4	3 040	2 119	3 973	11 163	1 977	4.89	1 235

We note that the USA graph with transit times appears to be much "easier" for RE than the graphs listed in Table 2. In terms of total number of vertices, this graph is half-way between CTR and NA, the two largest graphs in that table. But the performance (in terms of vertices scanned on the average and worst cases) is more similar to CAL or LKS, which are significantly smaller. We conjecture that this difference is due mainly to the fact that the number of road categories in the USA graph is very small: only four, as reported in [27]. In contrast, `Mappoint.NET` data has more than 80 categories; although this does not mean there are 80 different speed limits, they are certainly more numerous. As a result, the hierarchy in USA seems to be clearer than in other graphs, which benefits RE and HH.

With travel distances, RE is almost six times slower than with travel times. This loss in performance is much more pronounced than on the `Mappoint.NET` graphs (as shown in Tables 2 and 3). In fact, with travel distances the USA graph does not look any easier than other graphs of comparable size. This supports our claim that USA with travel times is easy mainly because of an artificially pronounced hierarchy. Our fastest algorithm, REAL, also becomes slower with travel distances, but by a factor of only two. The time for ALT also increases, but only slightly.

With travel distances, Sanders and Schultes [29] only present data for the road network of Germany, which has 4.3 million vertices. For better performance, they tune some parameters of their algorithm for travel distances differently from travel times, whereas we use the same parameters for all graphs. Only the average query time is reported: 32 ms (compared to 5.2 ms with travel time lengths). The ratio between the average query time for the distance lengths and the average query time for the travel time lengths is about six, which is similar to our results for RE on USA. The preprocessing time for Germany with travel distances was roughly four times higher than with travel times (2.1 instead of 0.5 hours) [28]. It is not clear how well this would scale for larger graphs.

8.5 Exact Reaches and Shortcuts

Recall that our preprocessing algorithm does not compute exact reaches, but upper bounds. This inevitably leads to less efficient queries. To assess how much is actually lost, we took a relatively small instance (BAY) and computed exact reaches on the original graph and on the version with shortcuts for both metrics (travel times and travel distances). We also ran a modified version of our preprocessing procedure that does *not* add shortcuts to the graph. Table 5 summarizes the results we obtained when running RE with those reaches. We also show the results obtained by the approximate reach computation, already reported in Tables 2 and 3.

Table 5: Results for RE with different reach values on BAY, both with and without shortcuts.

			PREP.	QUERY		
			TIME	AVG	MAX	TIME
METRIC	SHORTCUTS	REACHES	(min)	SCANS	SCANS	(ms)
TIMES	NO	APPROX.	52.8	13 369	28 420	6.44
		EXACT	966.1	11 194	24 358	6.05
	YES	APPROX.	3.2	1 590	3 438	1.17
		EXACT	980.7	1 383	3 056	0.97
DISTANCES	NO	APPROX.	82.5	17 448	37 171	9.47
		EXACT	956.9	13 986	30 788	7.61
	YES	APPROX.	4.6	2 761	6 313	2.05
		EXACT	1 078.9	2 208	5 159	1.55

The table makes it clear that computing exact shortcuts is prohibitively expensive. Even though the graph has only 330 024 vertices, the computation took more than 16 hours. On our largest instance, a similar computation would take years (it has almost 100 times more vertices and the procedure is quadratic). Fortunately, approximate reaches appear to be good enough for the query algorithm. Even though having exact values does make the query more efficient (especially for travel distances) it does so by only a small margin—less than 25% for all measurements.

The addition of shortcuts, on the other hand, has a much stronger effect on the algorithm. Without them, the preprocessing procedure for approximate reaches becomes more than 15 times slower on this graph. More importantly, the query algorithm gets at least five times worse on all counts.

Although we could not generate exact reaches for larger graphs (because of the prohibitive time), we did generate approximate reaches without shortcuts for on NW (which has 1.6 million vertices) with travel times. The preprocessing time increased from 17.5 minutes (as reported in

Table 2) to more than 18 hours—a factor of more than sixty. With random queries, RE visited 53 888 vertices on average (in 30.6 ms) and 106 288 in the worst case. Each of these measures is at least 10 times worse that the corresponding value for RE in Table 2. In fact, the algorithm becomes worse than ALT on average.

The fact that exact reaches are not very helpful does not mean the reach-based algorithm has little room for improvement. The reaches are exact, but the shortcuts were found by a heuristic, and a significantly better set of shortcuts may exist.

8.6 Number of Landmarks

In this experiment, we study how the number of landmarks affects the performance of REAL. We generate sets of $2, 4, 8, 16$, and 32 landmarks and run REAL on NA with transit time metric, with the same 1000 randomly selected pairs of vertices. The landmarks are generated with the *maxcover* algorithm [15]. We also run RE, which can be viewed as REAL with zero landmarks. Table 6 summarizes the results.

Table 6: Results for REAL on NA with travel times when varying the number of landmarks. For each number of landmarks, we show the time to generate landmarks, disk space used by landmarks, average number of vertices scanned (over 1000 queries), maximum number of vertices scanned, and average query time. Data for 0 landmarks refers to RE.

	GEN.	LAND.	QUERY		
	TIME	SPACE	AVG	MAX	TIME
LAND.	(min)	(MB)	SCANS	SCANS	(ms)
0	—	—	14 684	24 618	17.38
2	8.7	235.3	9 055	37 591	18.42
4	16.6	470.7	3 810	13 267	8.66
8	41.2	941.4	2 472	11 621	5.53
16	95.3	1 882.8	1 595	7 450	3.67
32	300.0	3 765.6	1 257	5 867	3.17

As shown in [15], increasing the number of landmarks improves the performance of ALT. The same holds for REAL, with one exception: the algorithm becomes slower when the number of landmarks increases from zero to two. The decrease in the average number of nodes scanned is not sufficient to pay for the overhead of computing distance lower bounds. The maximum number of scanned vertices also increases in this case.

With four or more landmarks, the performance of REAL only improves as the of landmarks increases. This is to be expected, since a larger number of landmarks can provide better bounds for the A^* search. However, as the table shows, after 16 landmarks, simply adding more is not a particularly efficient way of improving the bounds. The decrease in the running time is small compared to the increase in space. But the experiment does suggest that REAL may benefit from a way of generating landmarks that takes reaches into account. We discuss this in more detail in Section 10.

8.7 Grids

Although road networks are our motivating application, we also tested our algorithms on grid graphs. As for road networks, for each graph we generated 1000 pairs of vertices, each selected uniformly at random. These graphs have no natural hierarchy of shortest paths, which results in a large fraction of the vertices having high reach. For these tests, we used the same parameter settings as for road networks. It is unclear how much one can increase performance by tuning parameter values. As preprocessing for grids is fairly expensive, we limited the maximum grid size to about half a million vertices. The results are shown in Table 7.

Table 7: Algorithm performance on grid graphs with random arc lengths. For each graph and each method, the table shows the total time spent in preprocessing, the total size of the data stored on disk after preprocessing, the average number of vertices scanned (over 1000 random queries), the maximum number of vertices scanned, and the average running time. For the last three measures, we show both the actual value and the speedup (SPD) with respect to B.

		PREP.	DISK	QUERY					
		TIME	SPACE	AVG SCANS		MAX SCANS		AVG TIME	
VERTICES	METHOD	(min)	(MB)	COUNT	SPD	COUNT	SPD	msec	SPD
65 536	ALT	0.2	6.2	686	29.6	8 766	5.5	0.52	17.6
	RE	12.3	5.2	5 514	3.7	10 036	4.8	3.09	2.9
	REAL	12.5	9.6	363	55.9	2 630	18.4	0.34	26.4
131 044	ALT	0.6	12.4	1 307	32.6	14 400	7.2	1.42	13.9
	RE	44.7	10.4	9 369	4.6	16 247	6.4	5.94	3.3
	REAL	45.3	19.3	551	77.4	3 174	32.6	0.77	25.8
262 144	ALT	0.9	25.1	2 382	35.9	27 399	7.3	2.81	16.1
	RE	131.4	20.7	14 449	5.9	24 248	8.3	9.75	4.6
	REAL	132.3	38.8	791	108.0	5 020	39.9	1.22	37.1
524 176	ALT	1.9	50.2	4 416	38.8	40 568	9.9	5.25	17.5
	RE	232.1	41.4	23 201	7.4	39 433	10.2	17.47	5.3
	REAL	234.1	77.7	1 172	146.3	7 702	52.3	1.61	57.2

As expected, RE does not get nearly as much speedup on grids as it does on road networks (see Tables 2 and 3). However, there is some speedup, and it does grow (albeit slowly) with grid size. ALT is significantly faster than RE: in fact, its speedup on grids is comparable to that on road networks. However, the speedup does not appear to change much with the grid size, and it is likely that for very large grids RE would be faster.

An interesting observation is that REAL remains the best algorithm in this test, and its speedup grows with grid size. For our largest grid, queries for REAL improve on ALT by about a factor of four for all performance measures that we considered. The space penalty of REAL with respect to

ALT is a factor of about 1.5. REAL is over 50 times better than B. This shows that the combination of reaches and landmarks is more robust than either ALT or RE individually.

The most important downside of the reach-based approach on grids is its large preprocessing time. An interesting question is whether this can be improved. This would require a more elaborate procedure for adding shortcuts to a graph (instead of just waiting for lines to appear). Such an improvement may lead to a better preprocessing algorithm for road networks as well.

9 Potential Improvements

Below we mention several modifications of RE and REAL aimed at improving performance and reducing the memory overhead. We did not have a chance to implement these ideas, but they are promising.

The number of high-reach vertices is small, but during a query these vertices are visited much more often that other vertices. This observation can be used to improve the locality of the algorithm. One approach is to reorder vertices (and their arcs) in non-increasing order of reaches. This places high-reach vertex data in the same cache lines and memory pages. If the original vertex ordering uses spatial locality, as is often the case with road networks, one can use a variant of this idea. One can use an approximate sorting based on threshold values, and preserve vertex ordering within groups of vertices between two threshold values.

We can reduce the space required to store $\overline{r}$ values by picking a constant γ, rounding $\overline{r}$'s up to the nearest integer power of γ, and storing the logarithms to the base γ of the $\overline{r}$'s. Furthermore, we can reorder vertices according to the values of these logarithms and store the positions in the sorted list where the values increase. Then the rounded reach of a vertex can be computed from the position of the vertex in the sorted list. If γ is sufficiently small, the rounding has small effect on the precision of the bounds, and the number of vertices pruned during a query does not decrease by much.

Another way to save space is to store landmark distances only for the fraction (e.g., 20%) of vertices with reach greater than a threshold R. The query algorithm first searches balls of radius R around s and t without using landmarks. If the shortest path is not found, the algorithm starts using landmarks. Dynamic landmark selection allows for an easy implementation of this strategy. There are two reasons why the relative time penalty of this method compared to REAL may be small. First, both reaches and A^* search are not very effective in pruning vertices close to s and t and may already visit most vertices in the balls of radius R. Second, if R is small, the cardinality of the balls is small and the modified algorithm scans a small number of extra vertices.

For road network applications, the graph, although directed, has a lot of symmetry: most road segments are two-way and have the same length in each direction. One can take advantage of this fact to obtain a more compact representation of the graph.

Our implementation of REAL uses the same landmarks as ALT. However, it is possible that a selection strategy that is specific to REAL will work better. Bad queries for the RE algorithm are those where s or t are in or close to dense portions of the graph. Placing landmarks in the densest areas may help to deal with such cases. Another way to tailor landmarks to high-reach vertices is to generate landmarks on a subgraph induced by vertices with reach greater than a certain threshold.

10 Concluding Remarks

The reach-based shortest path approach leads to simple query algorithms with efficient implementations. The shortcuts greatly improve performance of the reach-based algorithms on road networks. We have shown that the algorithm RE, based on these ideas, is competitive with the best previous method. The reach-based approach combines naturally with A^* search. The resulting algorithm, REAL, has improved query time compared to RE.

Goldberg and Harrelson [13] suggest efficiency as a machine- and implementation-independent measure of performance, where efficiency of a shortest path computation is the number of arcs on the shortest path divided by the number of vertices scanned. With no shortcuts, the algorithm needs to scan all vertices on the shortest path except for the last one, so efficiency is at most one. Efficiency of 0.5 means that the number of scans is within a factor of two from the lower bound. With shortcuts, this is no longer the case. We can find a shortest path in the original graph while scanning fewer vertices than the number of arcs on the path. Efficiency with respect to the graph with shortcuts depends on the shortcuts, which makes in unusable for comparing different implementations. However, this efficiency is small, indicating that there is room for query algorithm improvement. For example, for NA with the travel-distance metric, the average number of arcs on shortest paths for random queries is 109, and the average number of scanned vertices is 2653.

Our query algorithm is independent of the preprocessing algorithm, allowing us to state natural subproblems for the latter. One such question is that of optimal shortcut selection. What is a good number of shortcuts to add? Where to add them? How to do so efficiently? Of course, the number of shortcuts to add depends on the amount of memory and preprocessing time available. Note that adding a shortcut for every pair of vertices reduces the reach of every vertex to zero and allows queries to be answered in constant time, but is impractical for large road networks.

Another natural problem, raised in Gutman's paper, is that of efficient reach computation. Can one compute reaches in less than $\Theta(nm)$ time? What about provably good upper bound on reaches? Our results add another dimension to this direction of research by allowing to add shortcuts to improve performance.

Another interesting direction of research is to identify a wider class of graphs for which these techniques work well, and to make the algorithms robust over this family.

Acknowledgments

We would like to thank Frank McSherry, Peter Sanders, and Dominik Schultes for helpful discussions.

References

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

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

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

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

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

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

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

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

[28] P. Sanders and D. Schultes. Personal communication. 2005.

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

[30] 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.

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

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

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

[34] 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.

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

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

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

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