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

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

Abstract

We study the point-to-point shortest path problem in a setting where preprocessing is allowed. We improve the reach-based approach of Gutman [17] in several ways. In particular, we introduce a bidirectional version of the algorithm that uses implicit lower bounds and we add shortcut arcs to reduce vertex reaches. Our modifications greatly improve both preprocessing and query times. The resulting algorithm is as fast as the best previous method, due to Sanders and Schultes [28]. However, our algorithm is simpler and combines in a natural way with A^* search, which yields significantly better query times.

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 only. 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, 15, 20, 25, 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 [19, 27, 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, 21, 34]. Previous work on exact algorithms with preprocessing includes those using geometric information [24, 36], hierarchical decomposition [28, 29, 30], the notion of reach [17], and A^* search combined with landmark distances [13, 16].

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 [17] defines the notion of vertex *reach*. Informally, the reach of a vertex is 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 [16]) have shown that the performance of A^* search (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 prob-

*Microsoft Research, 1065 La Avenida, Mountain View, CA 94062. E-mail: goldberg@microsoft.com; URL: http://www.research.microsoft.com/~goldberg/.

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

‡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. E-mail: rwerneck@cs.princeton.edu.

lem. 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 [28] (see also [29]) have recently introduced an interesting algorithm based on highway hierarchy; we call it the HH algorithm. They describe it for undirected graphs, and briefly discuss how to extend it to directed graphs. However, at the time our experiments have been completed and our technical report [14] published, there was no implementation of the directed version of the highway hierarchy algorithm. Assuming that the directed version of HH is not much slower than the undirected version, HH is the most practical of the previously published P2P algorithms for road networks. It has fast queries, relatively small memory overhead, and reasonable preprocessing complexity. Since the directed case is more general, if an algorithm for directed graphs performs well compared to HH then it follows that this algorithm performs well compared to the current state of the art. We compare our new algorithms to HH in Section 8.3.

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 the 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 first contribution of our work is the introduction of several variants of the reach algorithm, including bidirectional variants that do not need explicit lower bounds. We also introduce the idea of adding *shortcut arcs* to reduce vertex reaches. A small number of shortcuts (less than n, the number of vertices) drastically speeds up the preprocessing and the query of the reach-based method. The performance of the algorithm that implements these improvements (which we call RE) is similar to that of HH. We then show that the techniques behind RE and ALT can be combined in a natural way, leading to a new algorithm, REAL. On road networks, the time it takes for REAL to answer a query and the number of vertices it scans are much lower than those for RE and HH.

Furthermore, we suggest an interpretation of HH in terms of reach, which explains the similarities between the preprocessing algorithms of Gutman, HH, and RE. It also shows why HH cannot be combined with ALT as naturally as RE can.

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 practical algorithms. For the graph of the road network of North America (which has almost 30 million vertices), finding the fastest route between two random points takes less than 4 milliseconds on a standard workstation, while scanning fewer than 2 000 vertices on average. Local queries are even faster.

Due to the page limit, we omit some details, proofs, and experimental results. A full version of the paper is available as a technical report [14].

2 Preliminaries

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

The labeling method for the shortest path problem [22, 23] 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. 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 and each vertex is scanned once. 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, 26] 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. A better criterion (see [16]) is to 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. Intuitively, the bidirectional algorithm searches two touching balls centered at s and t.

Alternating between scanning a vertex by the forward search and scanning a vertex by the reverse search balances the number of scanned vertices between these searches. One can, however, coordinate the progress of the two searches in any other way and, as long as we stop according to one of the rules mentioned above, correctness is preserved. Balancing the work of the forward and reverse searches is a strategy guaranteed to be within a factor of two of the optimal strategy, which is the one that splits the work between the searches to minimize the total number of scanned vertices. 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

The following definition of reach is due to Gutman [17]. Given a path P from s to t and a vertex v on P, the *reach of v with respect to P* is the minimum of the length of the prefix of P (the subpath from s to v) and the length of the suffix of P (the subpath from v to t). The *reach* of v, $r(v)$, is the maximum, over all **shortest** paths P that contain v, of the reach of v with respect to P. (For now, assume that the shortest path between any two vertices is unique; Section 5 discusses this issue in more detail.)

Let $\overline{r}(v)$ be an upper bound on $r(v)$, and let $\underline{\text{dist}}(v, w)$ be a lower bound on $\text{dist}(v, w)$. The following fact allows the use of reaches to prune Dijkstra's search:

> Suppose $\overline{r}(v) < \underline{\text{dist}}(s, v)$ and $\overline{r}(v) < \underline{\text{dist}}(v, t)$. Then v is not on the 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.

To compute reaches, it suffices to look at all shortest paths in the graph and apply the definition of reach to each vertex on 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 one we tested, which has around 30 million vertices, this computation would take years on existing workstations.

Note that, if one runs this algorithm from only a few roots, one will obtain valid lower bounds for reaches. Unfortunately, the query algorithm needs good *upper* bounds to work correctly. Upper bounding 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 reaches 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 from the source to v and a lower bound on the distance from 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.

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

Figure 1: Bidirectional bound algorithm. 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 $\bar{r}(v) < \gamma$.

the reverse search: we use the topmost label in the forward heap as a lower bound on the distance from s 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 all the following conditions hold: (1) v has not been scanned in the reverse direction, (2) $\bar{r}(v) < d_f(v)$, and (3) $\bar{r}(v) < \gamma$. When using these bounds, the stopping criterion is the same as for the standard bidirectional algorithm (without pruning). We call the resulting procedure the *bidirectional bound* algorithm. See Figure 1.

An alternative is to use the distance label of the vertex 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. More precisely, we 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 the self-bounding 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 has been scanned in the opposite direction and, if so, check whether the candidate path using (v,w) is the shortest path seen so far.

The following natural algorithm falls into both of the above categories. The algorithm balances the radius of the forward and reverse search regions by picking the labeled vertex with minimum distance label considering both search 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. We refer to this algorithm as *distance-balanced*. Note that one could also use explicit lower bounds in combination with the implicit bounds.

We call our implementation of the bidirectional Dijkstra's algorithm with reach-based pruning RE. The query is distance-balanced and uses two optimizations: early pruning and arc sorting. The former avoids labeling unscanned vertices if reach and distance bounds justify this. The latter uses adjacency lists sorted in decreasing order by the reach of the head vertex, which allows some vertices to be early-pruned without explicitly looking at them. The resulting code is 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 [17], and the third implicit in [28].

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 *shortcuts 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 (which increases with i, the iteration number). Each iteration applies a partial-trees step followed by the shortcut step. By the end of the i-th iteration, the algorithm eliminates every vertex which it can prove has reach less than ϵ_i. If there are still vertices left in the graph, we set $\epsilon_{i+1} = \alpha\,\epsilon_i$ (for some $\alpha > 1$) and proceed to the next iteration.

Approximate reach algorithms, including ours, need the notion of a *canonical path*, which is a shortest path with additional properties. In particular, between every pair (s, t) there is a unique canonical path. 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.

Next we briefly discuss the major components of the algorithm. Due to space limitations, we discuss a variant based on vertex reaches. We indeed use vertex reaches for pruning the query, but our best preprocessing algorithm uses arc reaches instead to gain efficiency (see [14] for details). The main ideas behind our arc-based preprocessing are the same as for the vertex-based version that we describe.

5.1 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. The tree construction stops when for every leaf y of T', one of two conditions holds: (1) y is a leaf of T or (2) the length of the x'-y path in T' is at least 2ϵ, where x' is the vertex adjacent to x on the x-y path in T'.

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 (at least ϵ). 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 algorithm builds the smaller trees as follows. 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 x 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.

Our preprocessing runs the partial-trees algorithm in iterations, multiplying the value of ϵ by a constant α, each time it starts a new iteration. 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 vertices in V_i have reach estimates above ϵ_{i-1} (for $i > 1$). To compute valid upper bounds for them, the partial-trees algorithm must take into account the vertices that have been deleted. It does so by using the concept of *penalties*, which implicitly increase the depths and heights of vertices in the partial trees. This ensures the algorithm will compute correct upper bounds.

Next we introduce arc reaches, which are similar to vertex reaches but carry more information and lead to faster preprocessing. They are useful for defining the penalties as well.

5.2 Arc Reaches. Let (v, w) be an arc on the shortest path P between s and t. The reach of this arc with respect to P is the minimum of the length of the prefix of P (the distance between s and w) and the length of the suffix of P (the distance between v and t). Note that the arc belongs to both the prefix and the suffix (a definition that excluded the arc from both would be equivalent). The arc reach of (v, w) with respect to the entire graph, denoted by $r(v, w)$, is the maximum reach of this arc with respect to all shortest paths P containing it.

During the partial-trees algorithm, we actually try to bound arc reaches instead of vertex reaches—the procedure is essentially the same as described before, and arc reaches are more powerful (the reach of an arc may be much smaller than the reaches of its endpoints). Once all arc reaches are bounded, they are converted into vertex reaches: a valid upper bound on the reach of a vertex can be obtained from upper bounds on the reaches of all incident arcs.

Penalties are computed as follows. The *in-penalty* of a vertex $v \in V_i$ is defined as

$$\textit{in-penalty}(v) = \max_{(u,v)\in A^+:(u,v)\notin A_i} \{\bar{r}(u, v)\},$$

if v has at least one eliminated incoming arc, and zero otherwise. In this expression, A^+ is the set of original arcs augmented by the shortcuts added up to iteration i. The *out-penalty* of v is defined similarly, considering outgoing arcs instead of incoming arcs:

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

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

The partial-trees algorithm works as described above, but increases the lengths of path suffixes and prefixes by out- and in-penalties, respectively, for the purpose of reach computation.

5.3 Shortcut Step. We call a vertex v *bypassable* if it has exactly two neighbors (u and w) and one of the following condition holds: (1) v has exactly one incoming arc, (u, v), and one outgoing arc, (v, w); or (2) v has exactly two outgoing arcs, (v, u) and (v, w), and exactly two incoming arcs, (u, v) and (w, v). 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 are 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, which can either be *one-way* or *two-way*. 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 can simply add a shortcut (u, w) (and (w, u), in case it is a two-way line). The length and the perturbation associated with the shortcut is the sum of the corresponding values of the arcs it bypasses. We break the tie thus created by making the shortcut *preferred* (i.e., implicitly shorter). If v is a bypassed vertex, any shortest path that passes through u and w will no longer contain v. This potentially reduces the reach of v. If the line has more than two arcs, we actually add "sub-lines" as well: we recursively process the left half, then the right half, and finally bypass the entire line. This reduces reaches even further, as the example in Figure 2 shows.

Once a vertex is bypassed, we immediately delete it from the graph to speed up the reach computation. As long as the appropriate penalties are assigned to its neighbors, the computation will still find valid upper bounds on all reaches.

One issue with the addition of shortcuts is that they may be very long, which can hurt the performance of the partial-trees algorithm in future iterations. To avoid this, we limit the length of shortcuts that may be added in iteration i to at most $\epsilon_{i+1}/2$.

5.4 The Refinement Phase. The fact that penalties are used to help compute valid upper bounds tends to make the upper 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. 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: it recomputes the reach estimates of the δ vertices with highest (upper bounds on) reaches found during the main step, where δ is a user-defined parameter (we used $\delta = \lceil 10\sqrt{n} \rceil$).

Let V_δ be this set of high-reach vertices of G. To

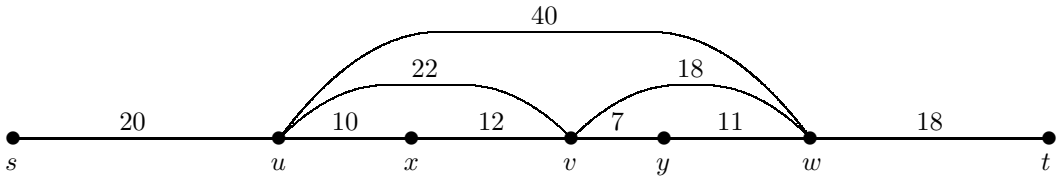

Figure 2: In this graph, (s,u), (u,x), (x,v), (v,y), (y,w), and (w,t) are the original edges (for simplicity, the graph is undirected). Without shortcuts, their reaches are $r(s) = 0$, $r(u) = 20$, $r(x) = 30$, $r(v) = 36$, $r(y) = 29$, $r(w) = 18$, and $r(t) = 0$. If we add just shortcut (u,w), the 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 $r(x) = r(y) = 0$.

recompute the reaches, we first determine 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 vertex 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 penalties to account for the remaining vertices.

5.5 Additional Parameters. The choice of ϵ_1 and α is a tradeoff between preprocessing efficiency and the quality of reaches and shortcuts. To choose ϵ_1, we first pick $k = \min\{500, \lfloor\lceil\sqrt{n}\rceil/3\rfloor\}$ vertices at random. For each vertex, we compute the radius of a partial shortest path tree with exactly $\lfloor n/k \rfloor$ scanned vertices. (This radius is the distance label of the last scanned vertex.) Then we set ϵ_1 to be twice the minimum of all k radii. We use $\alpha = 3.0$ until we reach an iteration in the main phase where the number of vertices is smaller than δ, then we reduce it to 1.5. This change allows the algorithm to add more shortcuts in the final iterations. The refinement step ensures that the reach bounds of the last δ vertices are still good.

6 Reach and the ALT Algorithm

6.1 A^* Search and the ALT Algorithm. A *potential function* is a function from the vertices of a graph G 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 the original distance function 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, 18] 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_f} 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*. As shown in [16], better estimates lead to fewer vertices being scanned. In particular, a lower-bounding algorithm with a nonnegative potential function visits no more vertices than Dijkstra's algorithm, which uses 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. [19], defined as $p_f(v) = (\pi_f(v) - \pi_r(v))/2$ for the forward computation and as $p_r(v) = (\pi_r(v) - \pi_f(v))/2 = -p_f(v)$ 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 [13, 16] is based on A^* and 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 bound on $\text{dist}(v, w)$ is given by a landmark that appears "before" v or "after" w. We use the version of ALT algorithm used in [16], which balances the work of the forward search and the reverse search.

6.2 Reach and A^* search. Reach-based pruning can be easily combined with A^* search. Gutman [17] 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. When A^* is about to scan a vertex v, we can extract the length of the shortest path from the source to v from the key of v (recall that $k_f(v) = d_f(v) + \pi_f(v)$). 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. 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 [16]. We call our implementation of the bidirectional A^* search algorithm with landmarks and reach-based pruning REAL. As in ALT, we used a version of REAL that balances the work of the forward search and the reverse search. Our implementation of REAL uses variants of early pruning and arc sorting, modified for the context of A^* 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. This 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. The actual algorithms used to obtain this data can be changed at will.

7 Other Reach Definitions and Related Work

7.1 Gutman's Algorithm. In [17], 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 it 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 some other differences in the preprocessing algorithm, but their effect on performance is less significant. 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. Since HH is described for undirected graphs, we restrict the discussion to this case.

We introduce a variant of reach that we call *c-reach* (cardinality reach). Given a vertex v on a shortest path P, grow equal-cardinality balls centered at its endpoints until v belongs to one of the balls. Let $c_P(v)$ be the

Table 1: Road Networks

NAME	DESCRIPTION	VERTICES	ARCS	LATITUDE (N)	LONGITUDE (W)
NA	North America	29 883 886	70 297 895	$[-\infty, +\infty]$	$[-\infty, +\infty]$
E	Eastern USA	4 256 990	10 088 732	[24.0; 50.0]	$[-\infty; 79.0]$
NW	Northwest USA	1 649 045	3 778 225	[42.0; 50.0]	[116.0; 126.0]
COL	Colorado	585 950	1 396 345	[37.0; 41.0]	[102.0; 109.0]
BAY	Bay Area	330 024	793 681	[37.0; 39.0]	[121; 123]

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

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

An important property of the HH query algorithm, which makes it similar to the self-bounding algorithm discussed in Section 4, is that the search in a given direction never goes to a lower level of the hierarchy. Our self-bounding algorithm can be seen as having a "continuous hierarchy" of reaches: once a search leaves a reach level, it never comes back to it. Like the self-bounding algorithm, HH cannot be combined with A^* search in a natural way.

8 Experimental Results

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 use a standard cache-efficient graph representation. All arcs are stored in a single array, with each arc represented by its head and its length.[1] The array is sorted by arc tail, so all outgoing arcs from a vertex appear consecutively. An array of vertices maps the identifier of a vertex to the position (in the list of arcs) of the first element of its adjacency list. All query algorithms use standard four-way heaps.

We conduct most of our tests on road networks. We test our algorithm on the five 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 four instances are representative subgraphs of NA (for tests on more subgraphs, see [14]). All graphs are directed and biconnected. We ran tests with two length functions: travel times and travel distances.

For a comparison with HH, we use the graph of the United States built by Sanders and Schultes [28] based on Tiger-Line data [35]. Because our implementations of ALT and REAL assume the graph to be connected (to simplify implementation), 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 performed 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., $x = y$).

Unless otherwise noted, in each experiment we run the algorithms with a fixed set of parameters. For ALT we use the same parameters as in [16]: for each graph we generated one set of 16 *maxcover* landmarks, and each s-t search uses dynamic selection to pick between two and

[1] The length is stored as a 16-bit integer on the original graphs and as a 32-bit integer for the graphs with shortcuts. The head is always a 32-bit integer.

six of those. The same set of landmarks was also used by 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.

8.2 Road Networks. Tables 2 and 3 present the results obtained by our algorithms when applied to the `Mappoint.NET` graphs with the travel-time and travel-distance metrics, respectively. In these experiments, we used 1000 random s-t pairs for each graph. We give results for preprocessing, average-case performance, and worst-case performance. For queries, we give both absolute numbers and the *speedup* with respect to an implementation of the bidirectional Dijkstra's algorithm (to which we refer as B).

For queries, the running time generally increases with graph size. While the complexity of ALT grows roughly linearly with the graph size, RE and REAL scale better. For small graphs, ALT is competitive with RE, but for large graphs the latter is more than 20 times faster. REAL is 3 to 4 times faster than RE.

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, used to convert shortcuts into its constituent arcs. The time for actually performing this conversion after each query is not taken into account in our experiments, since not all applications require it.

Next we compare the results for the two metrics. With the travel distance metric, the superiority of highways over local roads becomes much less pronounced than with travel times. As a result, RE become twice as slow for queries, and preprocessing takes 2.5 times longer on NA. On the other hand, ALT queries slow down only by about 20%, and preprocessing slows down even less. Changes in the performance of REAL fall in-between, which implies that its speedup with respect to RE becomes higher (on NA, REAL visits less than one tenth as many vertices as RE on average). All algorithms require a similar amount of space for travel times and for travel distances. While not quite as good as with travel times, the performance for travel distances is still excellent: REAL can find a shortest path on NA in less than 6 milliseconds on average.

While s and t are usually far apart on random s-t pairs, queries for driving directions tend to be more local. We used an idea from [28] to generate queries with different degrees of locality for NA. See Figure 3. When s and t are close together, ALT visits fewer vertices than RE. However, since the asymptotic performance of ALT is worse, RE quickly surpasses it as s and t get farther apart. REAL is the best algorithm in every case. Comparing plots for travel time and distance metrics, we note that ALT is less affected by the metric change than the other algorithms.

8.3 Comparison to Highway Hierarchies. As already mentioned, HH is the most practical of the previous P2P algorithms. Recall that HH works for undirected graphs only, while our algorithms work on directed graphs (which are more general). To compare our algorithms with HH, we use the (undirected) USA graph. Data for HH on USA, which we take from [28] and from a personal communication from Dominik Schultes, is available for the travel time metric only.

We compare both operation counts and running times. Since for all algorithms queries are based on the bidirectional Dijkstra's algorithm, comparing the number of vertices scanned is informative. For the running times, note that the HH experiments were conducted on a somewhat different machine. It was slightly slower than ours: an AMD Opteron running at 2.2 GHz (ours is an AMD Opteron running at 2.4 GHz) using the Linux operating system (ours uses Windows). Furthermore, implementation styles may be different. This introduces an extra error margin in the running time comparison. (To emphasize this, we use $\approx$ when stating running times for HH in Table 4.) However, comparing running times gives a good sanity check, and is necessary for preprocessing algorithms, which differ more than the query algorithms.

While in all our experiments we give the maximum number of vertices visited during the 1 000 queries we tested, Sanders and Schultes also obtain an upper bound on the worst-case number by running the search from each vertex in the graph to an unreachable dummy vertex and doubling the maximum number of vertices scanned. We did the same for RE. Note that this approach does not work for the landmark-based algorithms, as preprocessing would determine that no landmark is reachable to or from the dummy vertex. For both metrics, the upper bound is about a factor 1.5 higher than the lower bound given by the maximum over 1 000 trials, suggesting that the latter is a reasonable approximation of the worst-case behavior.

Data presented in Table 4 for the travel time metric suggests that RE and HH have similar performance and memory requirements. REAL queries are faster, but

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 in both absolute values and as a speedup 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	ms	SPD
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
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
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
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
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

Table 3: Algorithm performance on road networks with travel distances 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 in both absolute values and as a speedup 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	ms	SPD
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
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
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
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
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

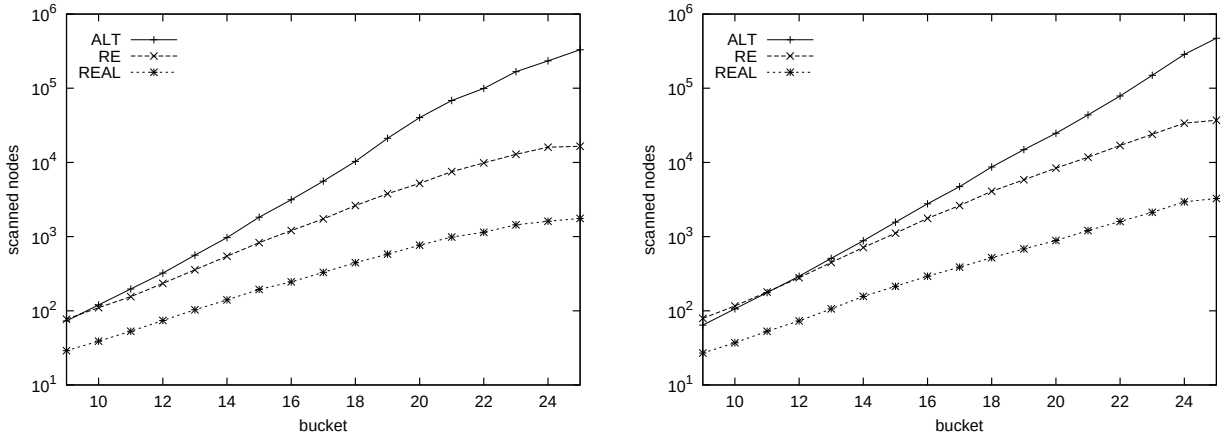

Figure 3: Average number of scanned vertices for local queries on NA with travel times (left) and distances (right). The horizontal axis refers to *buckets* with 1 000 pairs each. Each pair s-t in bucket i is such that s is chosen at random and t is the j-th farthest vertex from s, where j is selected uniformly at random from the range $(2^{i-1}, 2^i]$. The vertical axis is in log scale.

Table 4: Results for the undirected USA graph (same measures as in Table 2). For HH, averages are taken over 10 000 random queries (but the maximum is still taken over 1 000). For HH and RE we also give an upper bound on the maximum number of scans (UB). Data for HH with travel distances is not available.

		PREP.	DISK	QUERY						
		TIME	SPACE	AVG SCANS		MAX SCANS			AVG TIME	
METRIC	METHOD	(min)	(MB)	COUNT	SPD	COUNT	SPD	UB	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	13 364	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	5 955	3 412	8 678	≈ 7.04	≈ 937
DISTANCES	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	68 672	25.59	236
	REAL	1 081.4	3 040	2 119	3 973	11 163	1 977	—	4.89	1 235

it needs more memory. ALT queries are substantially slower, but preprocessing is faster.

Lacking data for HH, we cannot compare it to our algorithms for the travel distance metric. Performance of our algorithms on USA with this metric is similar to that on NA with the same metric. This suggests that directed graphs are not much harder than undirected ones for our algorithms. In contrast, with the travel time metric, the performance of RE (and, to a lesser extent, REAL) is much better on USA than on NA. This suggests that the hierarchy on the USA graph with travel times is more evident than on NA, probably because USA has a small number of road categories.

8.4 Grids. Although road networks are our motivating application, we also tested our algorithms on grid graphs. As with road networks, for each graph we generated 1 000 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 5.

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 grid size, and it is likely that for very large grids RE would be faster.

An interesting observation is that REAL remains the

Table 5: 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 1 000 random queries), the maximum number of vertices scanned (over the same queries), 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

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 during the preprocessing algorithm). Such an improvement may lead to a better preprocessing algorithm for road networks as well.

8.5 Additional Experiments. We ran our preprocessing algorithm on BAY with and without shortcut generation. The results are shown in Table 6. Without shortcuts, queries visited almost 10 times as many vertices, and preprocessing was more than 15 times slower; for larger graphs, the relative performance is even worse. Without shortcuts, preprocessing NA is impractical. The table also compares approximate and exact reach computations. Again, preprocessing for exact reaches is extremely expensive, and of course shortcuts do not make it any faster (note that the shortcuts in this case are the ones added by the approximate algorithm). Fortunately, our upper bounding heuristic seems to do a good enough job: on BAY, exact reaches improved queries by less than 25%.

We also experimented with the number of landmarks REAL uses on NA. With as few as four landmarks, REAL is already twice as fast as RE on average (while visiting less than one third of the vertices). In general, more landmarks give better results, but with more than 16 landmarks the additional speedup does not seem to be worth the extra amount of space required.

9 Conclusion and Future Work

The reach-based shortest path approach leads to simple query algorithms with efficient implementations. Adding shortcuts greatly improves the performance of these algorithms on road networks. We have shown that the algorithm RE, based on these ideas, is competitive with the best previous method. Moreover, it combines naturally with A^* search. The resulting algorithm, REAL, improves query times even more: an average query in North America takes less than 4 milliseconds.

However, we believe there is still room for improvement. In particular, we could make the algorithm more cache-efficient by reordering the vertices so that those with high reach appear close to each other. There are few of those, and they are much more likely to be visited during any particular search than low-reach vertices.

The number of vertices visited could also be reduced. With shortcuts added, a shortest path on NA with travel times has on average less than 100 vertices,

Table 6: Results for RE with different reach values on BAY, both with and without shortcuts.

METRIC	SHORTCUTS	REACHES	PREP. TIME (min)	QUERY AVG SCANS	QUERY MAX SCANS	QUERY TIME (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

but an average REAL search scans more than 1500 vertices. Simply adding more landmarks would require too much space, however. To overcome this, one could store landmark distances only for a fraction (e.g., 20%) of the vertices, those with reach greater than some threshold R. The query algorithm would first search balls of radius R around s and t without using landmarks, then would start using landmarks from that point on. Another potential improvement would be to pick a set of landmarks specific to REAL (in our current implementation, REAL uses the same landmarks as ALT).

Also, one could 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.

Our query algorithm is independent of the preprocessing algorithm, allowing us to state natural subproblems for the latter. What is a good number of shortcuts to add? Where to add them? How to do it efficiently?

Another natural problem, originally raised by Gutman [17], is that of efficient reach computation. Can one compute reaches in less than $\Theta(nm)$ time? What about provably good upper bounds on reaches? Our results add another dimension to this direction of research by allowing shortcuts to be added 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 more robust over that class.

Acknowledgments

We would like to thank Peter Sanders and Dominik Schultes for their help with the USA graph data.

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, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Al-

gorithms. Technical Report MSR-TR-2005-132, Microsoft Research, 2005.

[15] 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.

[16] 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.

[17] 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. SIAM, 2004.

[18] 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.

[19] T. Ikeda, M.-Y. 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.

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

[21] 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.

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

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

[24] 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.

[25] 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.

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

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

[28] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proc. 13th Annual European Symposium Algorithms*, volume 3669 of *LNCS*, pages 568–579. Springer, 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. 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] D. 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.