Better Landmarks within Reach

Andrew V. Goldberg[1], Haim Kaplan[2⋆], and Renato F. Werneck[1]

[1] Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA. {goldberg,renatow}@microsoft.com
[2] School of Mathematical Sciences, Tel Aviv University, Israel. haimk@post.tau.ac.il.

Abstract. We present significant improvements to a practical algorithm for the point-to-point shortest path problem on road networks that combines A^* search, landmark-based lower bounds, and reach-based pruning. Through reach-aware landmarks, better use of cache, and improved algorithms for reach computation, we make preprocessing and queries faster while reducing the overall space requirements. On the road networks of the USA or Europe, the shortest path between two random vertices can be found in about one millisecond after one or two hours of preprocessing. The algorithm is also effective on two-dimensional grids.

1 Introduction

We study the *point-to-point shortest path problem* (P2P): given a directed graph $G = (V, A)$ with nonnegative arc lengths, a source s, and a destination t, find a shortest path from s to t. Preprocessing is allowed, as long as the amount of precomputed data is linear in the input graph size; preprocessing time is limited by practical considerations. The algorithms have two components: a *preprocessing algorithm* that computes auxiliary data and a *query algorithm* that computes an answer for a given s-t pair. We are interested in exact solutions only.

No nontrivial theoretical results are known for the general P2P problem. For the special case of undirected planar graphs, sublinear bounds are known [7]. Experimental work on exact algorithms with preprocessing includes [8, 10–12, 14, 16, 18–20, 22, 23, 25]. Next we discuss the most relevant recent developments.

Gutman [12] introduced the notion of *vertex reach*. Informally, the reach of a vertex v is large if v is close to the middle of some long shortest path and small otherwise. Intuitively, local intersections have low reach and highways have high reach. Gutman showed how to prune an s-t search based on (upper bounds on) reaches and (lower bounds on) vertex distances from s and to t, using Euclidean distances as lower bounds. He also observed that efficiency improves further when reaches are combined with Euclidean-based A^* search, which uses lower bounds on the distance to the destination to direct the search towards it.

Goldberg and Harrelson [8] (see also [11]) have shown that A^* search (without reaches) performs significantly better when landmark-based lower bounds

⋆ Work partially done while this author was visiting Microsoft Research Silicon Valley.

are used instead of Euclidean ones. The preprocessing algorithm computes and stores the distances between every vertex and a small set of special vertices, the landmarks. Queries use the triangle inequality to obtain lower bounds on the distances between any two vertices in the graph. This leads to the ALT (A^* search, landmarks, and triangle inequality) algorithm for the P2P problem.

Sanders and Schultes [19, 20] use the notion of *highway hierarchies* to design efficient algorithms for road networks. The preprocessing algorithm builds a hierarchy of increasingly sparse *highway networks*; queries start at the original graph and gradually move to upper levels of the hierarchy, greatly reducing the search space. To magnify the natural hierarchy of road networks, the algorithm adds *shortcuts* to the graph: additional edges with length equal to the original shortest path between their endpoints.

We have recently shown [10] how shortcuts significantly improve the performance of reach-based algorithms, in terms of both preprocessing and queries. The resulting algorithm, called RE, can be combined with ALT in a natural way, leading to the REAL algorithm. Section 2 presents a more detailed overview of these algorithms. This paper continues our study of reach-based point-to-point shortest paths algorithms and their combination with landmark-based A^* search.

An important observation about RE and REAL is that, unless s and t are very close to each other, an s-t search will visit mostly vertices of high reach. Therefore, as shown in Section 3.1, reordering vertices by reach can significantly improve the locality (and running times) of reach-based queries. In Section 3.2 we develop the concept of *reach-aware landmarks*, based on the intuition that accurate lower bounds are more important for vertices of high reach. In fact, as we suggested in [10], one can keep landmark data for these vertices only. Balancing the number of landmarks and the fraction of distances that is actually kept, a memory-time trade-off is established. For large graphs, we were able to reduce both time and space requirements compared to our previous algorithm.

In addition, motivated by the work of Sanders and Schultes [20], Section 3.4 describes how shortcuts can be used to bypass vertices of arbitrary (but usually low) degree, instead of just degree-two vertices as our previous method did. This improves both preprocessing and query performance. We also study two techniques for accelerating reach computation: a modified version of our partial trees algorithm (for finding approximate reaches) and a novel algorithm for finding exact reaches. They are described in Sections 3.3 and 3.5, respectively.

Experimental results are presented in Section 4. On road networks, the algorithmic improvements lead to substantial savings in time, especially for preprocessing, and memory. The road networks of Europe or the USA can be preprocessed in one or two hours, and queries take about one millisecond (while Dijkstra's algorithm takes seconds). We also obtain reasonably good results for 2-dimensional grids. For grids of higher dimension and for random graphs, however, preprocessing becomes too expensive and query speedups are only marginal.

A preliminary version of this paper [9] was presented at the 9th DIMACS Implementation Challenge: Shortest Paths [5], where several other algorithms were introduced [1, 3, 4, 15, 17, 21]. Section 5 discusses these recent developments.

2 Algorithm Overview

Our algorithms have four main components: Dijkstra's algorithm, reach-based pruning, A^* search, and their combination. We discuss each in turn.

Dijkstra's algorithm. The *labeling method* finds shortest paths from a source s to all vertices in the graph as follows (see e.g. [24]). It keeps 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, it 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 $S(w) = \texttt{labeled}$. By always scanning the vertex with the smallest label, *Dijkstra's algorithm* [6] ensures that no vertex is scanned more than once. The P2P version can stop when it is about to scan the target t: the s-t path defined by the parent pointers is the solution.

One can also run Dijkstra's algorithm from the target on the reverse graph to find the shortest t-s path. The *bidirectional algorithm* alternates between the forward and reverse searches, each maintaining its own set of distance labels (denoted by $d_f(\cdot)$ and $d_r(\cdot)$, respectively). 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 $d_f(v) + \ell(v, w) + d_r(w)$ is less than the shortest path distance found so far (initially ∞), we update it. We perform similar updates during the reverse search. The algorithm stops when the two searches meet.

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 lengths of the s-v and v-t subpaths of P. 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 [12]. We assume shortest paths are unique, which can be achieved through perturbation. Computing exact reaches is impractical for large graphs; we must resort to upper bounds instead. We denote an upper bound on $r(v)$ by $\overline{r}(v)$, and a lower bound on the distance $\text{dist}(v, w)$ from v to w by $\underline{\text{dist}}(v, w)$. If, during an s-t query, we observe that $\overline{r}(v) < \underline{\text{dist}}(s, v)$ 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 can prune the search at v.

To apply this, we need lower bounds on $\text{dist}(s, v)$ and $\text{dist}(v, t)$. During a bidirectional search, one can use the bounds implicit in the search itself [10]. Consider the forward direction (the reverse case is similar), 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 t. We can prune the search at v if v has not been scanned in the reverse direction, $\bar{r}(v) < d_f(v)$, and $\bar{r}(v) < \gamma$. The algorithm can still stop when the searches meet.

Reach upper bounds are computed during the preprocessing phase. It works in rounds, each associated with a threshold ϵ_i (which grows exponentially with i). Round i bounds the reach of vertices whose reach is at most ϵ_i. It does so

by growing *partial trees* from each vertex, each with depth roughly $2\epsilon_i$. A vertex whose reach is bounded is eliminated from the graph and considered only indirectly in subsequent rounds. We also add *shortcuts* to the graph before each round [10]. Given two edges (u, v) and (v, w), a shortcut is a new edge (u, w) with length $\ell(u, v) + \ell(v, w)$ ($\ell(\cdot, \cdot)$ denotes the length of an edge). When ties are broken appropriately, the shortcut will ensure that v does not belong to the shortest path between u and w, potentially reducing v's reach, thus making pruning more effective and speeding up preprocessing. After all reaches are bounded, a *refinement step* builds the graph induced by the $\lceil 5\sqrt{n} \rceil$ vertices with highest reach bounds and recomputes the reaches using an exact algorithm. The preprocessing algorithm, as well as improvements relative to [10], are discussed in detail in Sections 3.3, 3.4, and 3.5.

A^* search and the ALT *algorithm.* A *potential function* maps vertices to reals. Given a potential function π, the *reduced cost* of an arc is defined as $\ell_\pi(v, w) = \ell(v, w) - \pi(v) + \pi(w)$. Suppose we replace ℓ by ℓ_π. The length of every s-t path changes by the same amount, $\pi(t) - \pi(s)$, so finding shortest paths in the original graph is equivalent to finding shortest paths in the transformed graph. Let π_f be a 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* [13] is an algorithm that works like Dijkstra's algorithm, but at each step selects a labeled vertex v with the smallest *key*, defined as $k_f(v) = d_f(v) + \pi_f(v)$, to scan next. This effectively guides the search towards t. 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 use as potential functions lower bounds on the distance from v to the target t. During the preprocessing stage, we pick a constant number of vertices as *landmarks* and store distances between them and every vertex in the graph; queries use these distances, together with the triangle inequality, to compute the lower bounds. The ALT algorithm is a bidirectional version of A^* search with landmark bounds and triangle inequality.

Combining reaches and landmarks. We can combine A^* search and reaches in the obvious way: running A^* search and pruning vertices based on reach conditions. Specifically, when A^* search is about to scan a vertex v with key $k_f(v) = d_f(v) + \pi_f(v)$, it can prune the search at v if $\bar{r}(v) < d_f(v)$ and $\bar{r}(v) < \pi_f(v)$. Note that this method (which we call REAL) has two preprocessing algorithms: reach computation and landmark generation. Although they are in principle independent, Section 3.2 will show that it might be useful to take reaches into account when generating landmarks.

3 Algorithmic Improvements

3.1 Improving Locality

When reaches are available, a typical point-to-point query spends most of its time scanning high-reach vertices. Except at the very beginning of the search,

low-reach vertices are pruned by reach. This suggests an obvious optimization: during preprocessing, reorder the vertices such that high-reach vertices are close together in memory to improve cache locality. Simply sorting vertices in non-increasing order of reach would destroy the locality of the input, which is often quite high. Instead, we partition the vertices into equal-sized sets, the first with the $n/2$ higher-reach vertices, and the other with the rest. We keep the original relative ordering in each part, then recursively process the first (high-reach) part. This also facilitates the optimizations described below.

3.2 Reach-Aware Landmarks

We can reduce the memory requirements of the algorithm by storing landmark distances only for high-reach vertices, with marginal performance degradation. If we use the saved space to add more landmarks, we get a wide range of trade-offs between query performance and memory requirement. We call the resulting method, a variant of REAL, the *partial landmark algorithm*.

Queries for this algorithm work as follows. Let R, the *reach threshold*, be the smallest value such that *all* vertices with reach at least R have landmark distances available. We say these vertices have *high reach*. Queries start as RE, with reach pruning but without A^* search, until both balls searched have radius R (or the algorithm terminates). From this point on, only vertices with reach R or higher will be scanned. We switch to REAL by removing labeled vertices from the heaps and reinserting them with new keys that incorporate lower bounds.

To process a vertex v, A^* search needs lower bounds on the distance from v to t (in the forward search) or from s to v (in the reverse search). They are computed with the triangle inequality, which requires distances between these vertices (v, s, and t) and the landmarks. These are guaranteed to be available for v, which has high reach, but not for s or t; for them, we must use *proxies*. The proxy for s, which we denote by s', is the high-reach vertex that is closest to s (t is treated similarly). A lower bound on $\text{dist}(s, v)$ using distances *to* a landmark L is given by $\text{dist}(s, v) \geq \text{dist}(s', L) - \text{dist}(v, L) - \text{dist}(s', s)$. Other bounds (using distances from landmarks and involving s and t) can be computed in a similar way. Proxies (and related distances) are computed during the initialization phase of the query algorithm with a multiple-source version of Dijkstra's algorithm.

We use the *avoid* algorithm [11] to select landmarks: it picks landmarks one by one, always in regions of the graph not already "covered" by existing landmarks. It does so by assigning to each vertex a *weight* that measures how well-covered it is. We changed the algorithm slightly from [11]: instead of computing the weights of all vertices in the graph, we only consider n/k of them (where k is the number of landmarks). This makes the algorithm linear in k, instead of quadratic, without a significant effect on solution quality.

3.3 Growing Partial Trees

Next we describe the partial-trees routine executed in each iteration of our preprocessing algorithm. For simplicity, we describe the algorithm as if it computed

vertex reaches; it actually computes *arc reaches* and eventually converts them to vertex reaches [10]. (The reach of an arc (v, w) with respect to a shortest s-t path containing it is $\min\{\text{dist}(s, w), \text{dist}(v, t)\}$.) In each round, we are given a graph $G = (V, A)$ and a threshold ϵ, and our goal is to find upper bounds on the reaches of vertices in V whose actual reaches are smaller than ϵ. The remaining reaches will be bounded in subsequent rounds, when ϵ will be larger.

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

The algorithm works by growing a *partial tree* T_r from each vertex $r \in V$. It runs Dijkstra's algorithm from r, but stops as soon as it can prove that all ϵ-minimal paths starting at r have been considered. Let v be a vertex in this tree, and let x be the first vertex (besides r) on the path from r to v. We say that v is an *inner vertex* if either (1) $v = r$ or (2) $d(x, v) < \epsilon$, where $d(x, v)$ denotes the distance between x and v in the tree. The inner vertices are those whose reaches we will try to bound. When v is not an inner vertex, no path P_{rw} starting at r will be ϵ-minimal with respect to v: if v's reach is greater than ϵ with respect to P_{rw}, it will also be greater than ϵ with respect to P_{xw}. To get accurate bounds on reaches, we must grow the tree until every inner vertex v has one of two properties: (1) v has no labeled (unscanned) descendents; or (2) v has at least one scanned descendent whose distance to v is ϵ or greater. For efficiency, we relax the second condition and stop when every labeled vertex is within distance greater than ϵ from the closest inner vertex.

Once the tree is built, we process it. Given an inner vertex v, we know its *depth*, equal to $d(r, v)$. In $O(|T_r|)$ time, we can also compute its *height*, defined as the distance from v to its farthest descendent (labeled or scanned). The reach of v with respect to T_r is the minimum between its depth and its height, and the reach of v with respect to the entire graph is the maximum over all such reaches. If this maximum is ϵ or greater, we declare the reach to be ∞.

Penalties. As described, the algorithm assumes that partial trees will be grown from every vertex in the graph. We would like, however, to run the partial-trees routine even after some of the vertices have been eliminated in a previous iteration. We use *penalties* to account for the eliminated vertices. The *in-penalty* of a vertex v is the maximum over the reaches of all arcs (u, v) that have already been eliminated; *out-penalties* are defined similarly, considering outgoing arcs instead. Partial trees are processed as before, but the definitions of *height* and *depth* must change. The (redefined) depth of a vertex v within a tree T_r, denoted by $\textit{depth}_r(v)$, is the distance from r to v plus the in-penalty of r. The (modified) height of v is the distance to its farthest descendent not in T_r itself, but in a *pseudo-tree* in which each vertex v in T_r is attached to a *pseudo-leaf* v' by an arc whose length is equal to the out-penalty of v.

For correctness, it is enough to take penalties into account only when processing the tree, as we did in [10]. We can, however, use penalties to stop growing

partial trees sooner, thus speeding up preprocessing. First, we now consider vertex v to be an inner vertex if either (1) $v = r$ or (2) $depth_x(v) < \epsilon$ (recall that x is the second vertex on the path from r to v). If $v \neq r$ and $depth_r(v) < \textit{in-penalty}(v)$, however, v will not be considered an inner vertex (because its modified depth will be even higher in the tree rooted at v), and neither will its descendents. Second, we stop growing the tree when no labeled (unscanned) vertex is *relevant*. All inner vertices are considered relevant; an outer vertex v is relevant if $d(u, v)+\textit{out-penalty}(v) \leq \epsilon$, where u is the closest inner ancestor of v.

3.4 Adding Shortcuts

Our previous implementation of the preprocessing procedure [10] only shortcuts vertices with degree two. Sanders and Schultes [20] suggested shortcutting other vertices of small degree as well, which is more general and works better for both preprocessing and queries. A vertex v can be *bypassed* as follows. First, we examine all pairs of incoming/outgoing arcs $((u, v), (v, w))$ with $u \neq w$. For each pair, if the arc (u, w) is not in the graph, we add a shortcut arc (u, w) of length $\ell(u, v) + \ell(v, w)$. Otherwise, we set $\ell(u, w) \leftarrow \min\{\ell(u, w), \ell(u, v) + \ell(v, w)\}$. Finally, we delete v and all arcs adjacent to it from the current graph.

The processing algorithm will produce a graph containing all original arcs and all shortcuts. To prevent the graph from being too large, we only bypass v if both its in-degree and its out-degree are bounded by a constant (5 in our experiments); this ensures that only $O(n)$ arcs will be added. In addition, we consider the ratio c_v between the number of new arcs added and the number of arcs deleted by the procedure above. A vertex v is deemed bypassable only if $c_v \leq c$, where c is a user-defined parameter. For road networks, we used $c = 0.5$ in the first round of preprocessing, 1.0 in the second, and 1.5 in the remaining rounds. As a result, relatively few shortcuts are added during the first two rounds, when the graph is larger but shrinks faster. For two- and three-dimensional grids, which do not shrink as fast, we fixed c at 1.0 and 2.0, respectively.

We also consider two additional measures (besides c_v) related to v: the length of the longest shortcut arc introduced when v is bypassed, and the largest reach of an arc adjacent to v (which will be removed). The maximum between these two values is the *cost* of v, and it must be bounded by $\epsilon_i/2$ during iteration i for the vertex to be considered bypassable. As explained in [10], long arcs and large penalties can decrease the quality of the reach upper bounds provided by the preprocessing algorithm; they should not appear too soon. When deciding which vertex to bypass next, we take those that minimize the product between c_v and cost, since they are less likely to affect the bypassability of their neighbors.

3.5 Exact Reach Computation

The standard algorithm for computing exact reaches (during the refinement step) builds a shortest path tree from each vertex r in the graph and computes the minimum between the depth and the height of each vertex v in the tree. The maximum of these minima over all trees will be the reach of v. We developed

an algorithm that has the same worst-case complexity, but can be significantly faster in practice on road networks. It follows the same basic principle, but builds parts of some of the shortest path trees *implicitly* by reusing previously found subtrees. The algorithm partitions the vertices of the graph into k regions. (In our experiments, we picked the regions of the Voronoi diagram of $k = \sqrt{n}$ randomly selected vertices.) The *frontier* of a region A is the set of vertices $v \in A$ such that there exists at least one arc (v, w) with $w \neq A$. When processing a region, the algorithm first grows full shortest path trees from the frontier. Typically, they will have large subtrees in common, and it is easy to see that the same subtrees would appear in the full shortest path trees rooted at non-frontier vertices as well. It therefore suffices to grow *truncated* trees from these vertices, which account for the common subtrees only implicitly.

4 Experimental Results

Our code was written in C++ and compiled with Microsoft Visual C++ 2005. All tests were performed on a dual-processor, 2.4 GHz AMD Opteron running Microsoft Windows Server 2003 with 16 GB of RAM, 32 KB instruction and 32 KB data level 1 cache per processor, and 2 MB of level 2 cache. Our code is single-threaded, but an execution is not pinned to a single processor.

We tested the ALT (with 16 landmarks), RE, REAL-(16, 1), and REAL-(64, 16) algorithms. Here REAL-(i, j) denotes an algorithm that uses i landmarks but maintains landmark data for n/j highest-reach vertices only (when $j = 1$, all landmark distances are kept, as in the original REAL algorithm). Due to space constraints, we omit detailed results for other values of i and j. In general, however, we observed that moderately increasing sparsity does affect running times, but not too much: on large road networks, REAL-(16,16) is less than 30% slower than REAL-(16,1), for instance. The sparser the landmarks, the more the algorithm will rely on RE (at the beginning); the slight increase in the number of vertices scanned is offset by the fact that scans are on average faster, since RE does not need to access landmark data.

For machine calibration purposes, we also ran the DIMACS Challenge implementation of the P2P version of Dijkstra's algorithm, denoted by D, on the largest road networks. Our experiments on road networks are described in Section 4.1, and experiments on grid graphs are reported in Section 4.2.

4.1 Road Networks

We first test our algorithm on the road networks of the USA and Europe, which belong to the DIMACS Challenge [5] data set. The USA is symmetric and has 23 947 347 vertices and 58 333 444 arcs; Europe is directed, with 18 010 173 vertices and 42 560 279 arcs. Both graphs are strongly connected. Two length functions are available in each case: travel times and travel distances.

Random queries. For each graph and each length function, we tested the algorithms on 1 000 pairs of vertices picked uniformly at random. Table 1 reports

Table 1. Data for random queries on Europe and USA graphs.

		PREP. TIME	DISK SPACE	QUERY		
GRAPH	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
Europe	ALT	13.2	1597	82348	993015	160.34
(times)	RE	82.7	626	4643	8989	3.47
	REAL-(16,1)	96.8	1849	814	4709	1.22
	REAL-(64,16)	140.8	1015	679	2955	1.11
	RE-OLD	1558	570	16122	34118	13.5
	REAL-OLD	1625	1793	1867	8499	2.8
	HH	15	1570	884	—	0.8
	HH-mem	55	692	1976	—	1.4
	D	—	393	8984289	—	4365.81
USA	ALT	18.6	2563	187968	2183718	400.51
(times)	RE	44.3	890	2317	4735	1.81
	REAL-(16,1)	63.9	3028	675	3011	1.14
	REAL-(64,16)	121.0	1575	540	1937	1.05
	RE-OLD	366	830	3851	8722	4.50
	REAL-OLD	459	2392	891	3667	1.84
	HH	18	1686	1076	—	0.88
	HH-mem	65	919	2217	—	1.60
	D	—	536	11808864	—	5440.49
Europe	ALT	10.1	1622	240750	3306755	430.02
(distances)	RE	49.3	664	7045	12958	5.53
	REAL-(16,1)	60.3	1913	882	5973	1.52
	REAL-(64,16)	89.8	1066	583	2774	1.16
	D	—	393	8991955	—	2934.24
USA	ALT	14.5	2417	276195	2910133	530.35
(distances)	RE	70.8	928	7104	13706	5.97
	REAL-(16,1)	87.8	2932	892	4894	1.80
	REAL-(64,16)	138.1	1585	628	4076	1.48
	D	—	536	11782104	—	4576.02

the average query time (in milliseconds), the average number of scanned vertices and, when available, the maximum number of scanned vertices. Also shown are the total preprocessing time and the total space on disk used by the preprocessed data. For D, this is the graph itself; for ALT, this includes the graph and landmark data; for RE, it includes the graph with shortcuts and an array of vertex reaches; the data for REAL includes the data for RE plus landmark data.

For travel times, the table also reports the performance of other algorithms available at the time of writing. We give the data for our previous implementations, RE-OLD and REAL-OLD from [10] (run on the same machine). REAL-OLD uses 16 landmarks selected with the *maxcover* method (which finds slightly better landmarks than *avoid*, but is slower). In addition, we present results for the highway hierarchy-based algorithm of Sanders and Schultes from [20], run sequentially on a dual-core 2.0 GHz AMD Opteron machine (which is about 20% faster than our machine on the DIMACS benchmark due to a different memory

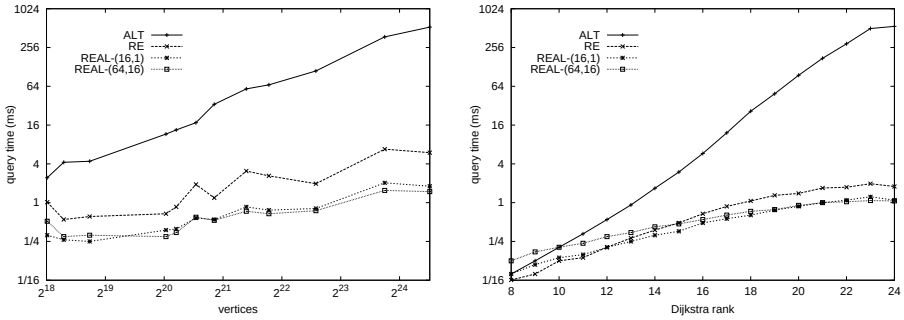

Fig. 1. USA queries: random on subgraphs (left) and local on the full graph (right).

architecture). There are two versions of their algorithm: HH-mem, entirely based on highway hierarchies, and HH, which replaces high levels of the hierarchy by a table with distances between all pairs of vertices in the corresponding graph.

The table shows that RE is considerably faster than ALT for queries, and that REAL-(16,1) yields an additional speedup. Comparing REAL-(16,1) to REAL-(64,16), we see they have almost identical query performance with transit times, and that REAL-(64,16) is slightly better with travel distances. Given that REAL-(64,16) requires about half as much disk space, it has the edge for these queries. With travel distances, REAL-(64,16) wins both in time and in space, but preprocessing takes roughly twice as long. RE is less robust than REAL.

With travel times, RE and REAL substantially improve on their old counterparts, especially in terms of preprocessing time. RE and HH-mem have similar performance on USA, and HH-mem is slightly better on Europe. For queries, REAL-(64,16) performs similarly to HH: REAL-(64,16) uses less space, while HH is slightly faster. Preprocessing for HH, however, is faster, especially for Europe.

Graph size dependence. To test the performance on smaller graphs, we performed random queries on the subgraphs of USA that are part of the DIMACS data set; their sizes range from 264 thousand (NYC) to 14 million (CTR) vertices. Figure 1 (left) shows how query times scale with graph size when travel times are used as the length function. Although times tend to increase with graph size, they are not strictly monotone: graph structure clearly plays a part. Reach-based algorithms have better asymptotic performance than ALT. Regarding preprocessing (not shown in the figure), with a fixed number of landmarks ALT is roughly linear in the graph size. With 16 landmarks, ALT preprocessing is faster than RE, and the ratio between the two remains roughly constant as the graph size increases; with 64, reach computation and landmark selection take roughly the same time.

Local queries. Up to this point, we have considered only random queries; we now test what happens when queries are more local. If v is k-th vertex scanned when Dijkstra's algorithm is run from s, then the *Dijkstra rank* of v with respect to s is $\lfloor \log_2 k \rfloor$ (our definition differs slightly from [19]). To generate a local query with rank k, we pick s uniformly at random from V and pick t uniformly at random from all vertices with Dijkstra rank k with respect to s. Figure 1 (right) shows

Table 2. Effect of applying (•) or discarding (○) each of the improvements to RE on USA with travel times: generalized shortcuts (S), penalty-aware partial trees (P), faster exact reach computation (E) and improved locality (L).

FEATURES				PREP. TIME	DISK SPACE	QUERY		
S	P	E	L	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
•	•	•	•	44.3	890	2317	4735	1.81
○	•	•	•	63.8	817	3861	7679	2.64
•	○	•	•	76.2	888	2272	4633	1.75
•	•	○	•	58.7	890	2317	4735	1.81
•	•	•	○	44.3	890	2317	4735	3.20
○	○	○	○	156.1	816	3741	7388	3.89

the average query times as a function of Dijkstra rank (1 000 pairs were tested in each case). For high Dijkstra ranks, the results are similar to those for random queries: ALT is the slowest algorithm, and the REAL variants are the fastest. For small Dijkstra ranks, REAL-(16,1) scans the fewest vertices, but due to higher overhead its running time is slightly worse than that of RE. REAL-(64,16) mostly visits low-reach vertices and thus fails to take advantage of the landmark data. It scans about the same number of vertices as RE, but is slower due to higher overhead. Although ALT has the worst asymptotic performance, for small ranks it scans only slightly more vertices than RE. As the rank grows, REAL-(64,16) eventually catches up with REAL-(16,1).

Improvement breakdown. Table 1 has shown that the new version of RE is significantly more efficient than RE-OLD. Table 2 shows how each of the four major improvements affect the performance of RE on USA with travel times. Starting with all improvements, we turn them off one at a time, and then all at once. Preprocessing is accelerated by generalized shortcuts (Section 3.4), penalty-aware partial trees (Section 3.3), and faster exact reach computation (Section 3.5). Sorting by reach to improve locality (Section 3.1) actually slows preprocessing, but by a negligible amount. When these improvements are combined, the overall speedup is more than 3.5. On Europe with travel times (not shown), the combined speedup is more than 6. Queries benefit from generalized shortcuts and sorting by reach, and are largely unaffected by the other improvements.

Refinement step. During preprocessing, the refinement step recomputes the $\lceil 5\sqrt{n} \rceil$ highest reaches (24 469 vertices in the USA graph) with an exact algorithm. If we quadruple this value, RE query times decrease from 1.86 ms to 1.66 ms (with travel times as lengths); however, preprocessing time increases from under 45 minutes to 2.5 hours. With no refinement step, the preprocessing time decreases to 32 minutes, but query times increase to 2.00 ms. Recomputing all reaches is too expensive. Even on Bay Area, which has only 321 270 vertices, exact reach computation takes almost 2.5 hours with the new algorithm (10 hours with the standard one). Computing upper bounds takes less than a minute, but queries are 40% faster with exact reaches.

Table 3. Average results for 1 000 random queries on 2-dimensional grids.

		PREP. TIME	DISK SPACE	QUERY		
VERTICES	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
65536	ALT	0.05	11.5	851	6563	1.19
	RE	1.83	3.4	2195	3706	1.50
	REAL-(16,1)	1.87	12.7	214	1108	0.30
	REAL-(64,16)	2.00	5.9	1999	3134	1.69
	D	—	2.2	33752	—	14.83
131044	ALT	0.11	23.8	1404	11535	3.62
	RE	4.72	6.8	3045	5025	2.33
	REAL-(16,1)	4.82	26.1	266	1261	0.61
	REAL-(64,16)	5.09	12.1	2370	3513	2.19
	D	—	4.5	66865	—	30.72
262144	ALT	0.22	48.3	2439	27936	5.72
	RE	16.61	13.2	4384	6933	3.52
	REAL-(16,1)	16.83	52.5	410	2024	0.88
	REAL-(64,16)	17.42	23.9	1869	2453	2.28
	D	—	9.0	134492	—	63.58
524176	ALT	0.27	96.6	6057	65664	8.11
	RE	15.45	25.9	6334	9843	4.23
	REAL-(16,1)	15.76	104.5	524	2663	1.08
	REAL-(64,16)	16.68	47.5	2001	3113	1.97
	D	—	18.0	275589	—	112.80

Retrieving the shortest path. The query times reported so far for RE and REAL consider only finding the shortest path on the graph with shortcuts. This path has much fewer arcs then the corresponding path in the original graph. On USA with travel times, for example, the shortest path between a random pair of vertices has around 5 000 vertices in the original graph, but only about 30 in the graph with shortcuts. We can retrieve the original path in time proportional to its length, which means about 1 ms on the USA graph. This is comparable to the time it takes for REAL to compute the distances. For applications that require a full description of the path, our algorithms are therefore close to optimal.

4.2 Grid Graphs

Reach-based pruning works well on road networks because they have a natural highway hierarchy, so that relatively few vertices have high reach. We also tested the algorithms on graphs without an obvious hierarchy. We created square 2-dimensional with the `spgrid` generator, available at the 9th DIMACS Challenge download page. The graphs are directed and each vertex is connected to its neighbors in the grid with arcs of length chosen uniformly at random from the range $[1, n]$, where n is the number of vertices. Comparing the results in Table 3 to those reported in [10], we see that our new preprocessing algorithm is an order of magnitude faster on these graphs. Query times improve by a factor of about five. This makes RE competitive with ALT; in fact, RE appears to be

asymptotically faster. Performance of REAL-(16,1) improves as well. This shows that reaches help even when a graph does not have an obvious highway hierarchy, and that the applicability of REAL is not restricted to road networks. On the largest grid, it is four times faster than ALT, and two orders of magnitude faster than Dijkstra's algorithm. On such small graphs, extra landmarks do not help much, and REAL-(64,16) does not perform as well as REAL-(16,1).

Similar experiments on cube-shaped three-dimensional grids (not shown) revealed that the ALT algorithm is much less effective than on two-dimensional grids. For a quarter of a million vertices, queries are only five times as fast as bidirectional Dijkstra's algorithm. A combination with pruning by reach does improve the algorithm for large graphs, but only marginally. Moreover, reach computation becomes asymptotically slower (the time roughly triples when the graph size doubles), thus making preprocessing large graphs prohibitively expensive. Results for higher-dimension grids and random graphs were even worse.

5 Final Remarks

Several other papers presented at the 9th DIMACS Implementation Challenge [5] also dealt with the P2P problem. Lauther [17] and Köhler et al. [15] presented algorithms based on arc flags, but their (preprocessing and query) running times are dominated by REAL and HH. Delling, Sanders, et al. [4] presented a variant of the partial landmarks algorithm in the context of highway hierarchies, but with only modest speedups; for technical reasons A^* search cannot be combined naturally with HH. Delling, Holzer, et al. [3] showed how multi-level graphs can support random queries in less than 1 ms, but only after weeks of preprocessing.

The best results were those based on *transit node routing*, introduced by Bast et al. [1] and combined with highway hierarchies by Sanders and Schultes [21] (see also [2]). With travel times, the road networks of both USA and Europe can be processed in about three hours and random queries take 5 μs on average. With travel distances, preprocessing takes about eight hours, and average query times are close to 0.1 ms. Performance would probably be worse on grids.

Queries with transit node routing are significantly faster than with REAL. Our method does appear to be more robust, however, when the length function changes. Moreover, these approaches are not mutually exclusive. As Bast et al. observe [2], reaches could be used instead of highway hierarchies to compute the transit nodes and the corresponding distance tables. An actual implementation of the combined algorithm is an interesting topic for future research.

References

1. H. Bast, S. Funke, and D. Matijevic. TRANSIT: Ultrafast shortest-path queries with linear-time preprocessing. 9th DIMACS Implementation Challenge, 2006.
2. H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proc. 9th ALENEX*. SIAM, 2007. Available at http://www.mpi-inf.mpg.de/ bast/tmp/transit.pdf.

3. D. Delling, M. Holzer, K. Muller, F. Schulz, and D. Wagner. High-performance multi-level graphs. 9th DIMACS Implementation Challenge, 2006.
4. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. 9th DIMACS Implementation Challenge, 2006.
5. C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/∼challenge9/, 2006.
6. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.
7. J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. 42nd FOCS*, pages 232–241, 2001.
8. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proc. 16th SODA*, pages 156–165, 2005.
9. A. V. Goldberg, H.Kaplan, and R. F. Werneck. Better landmarks within reach. 9th DIMACS Implementation Challenge, 2006.
10. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proc. 8th ALENEX*. SIAM, 2006.
11. A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proc. 7th ALENEX*, pages 26–40. SIAM, 2005.
12. R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. 6th ALENEX*, pages 100–111, 2004.
13. 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.
14. E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *WEA*, pages 126–138, 2005.
15. E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. 9th DIMACS Implementation Challenge, 2006.
16. 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.
17. U. Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. 9th DIMACS Implementation Challenge, 2006.
18. R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *WEA*, pages 189–202, 2005.
19. P. Sanders and D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. In *Proc. 13th ESA*, 2005.
20. P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proc. 14th ESA*, 2006.
21. P. Sanders and D. Schultes. Robust, almost constant time shortest-path queries on road networks. 9th DIMACS Implementation Challenge, 2006.
22. D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. Master's thesis, Department of Computer Science, Universitt des Saarlandes, Germany, 2005.
23. F. Schulz, D. Wagner, and K. Weihe. Using Multi-Level Graphs for Timetable Information. In *Proc. 4th ALENEX*, pages 43–59. LNCS, Springer, 2002.
24. R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
25. D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proc. 11th ESA*, 2003.