Computing the Shortest Path: A^* Search Meets Graph Theory

Andrew V. Goldberg[1] Chris Harrelson[2]

March 19, 2003

Technical Report
MSR-TR-2004-24

We study the problem of finding a shortest path between two vertices in a directed graph. This is an important problem with many applications, including that of computing driving directions. We allow preprocessing the graph using a linear amount of extra space to store auxiliary information, and using this information to answer shortest path queries quickly. Our approach uses A^* search in combination with a new graph-theoretic lower-bounding technique based on landmarks and the triangle inequality. We also develop new bidirectional variants of A^* search and investigate several variants of the new algorithms to find those that are most efficient in practice. Our algorithms compute optimal shortest paths and work on any directed graph. We give experimental results showing that the most efficient of our new algorithms outperforms previous algorithms, in particular A^* search with Euclidean bounds, by a wide margin on road networks. We also experiment with several synthetic graph families.

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

[1]Microsoft Research, 1065 La Avenida, Mountain View, CA 94062. Email: goldberg@microsoft.com; URL: http://www.avglab.com/andrew/index.html.

[2]Computer Science Division, UC Berkeley. Part of this work was done while the author was visiting Microsoft Research. Email: chrishtr@cs.berkeley.edu.

1 Introduction

The shortest path problem is a fundamental problem with numerous applications. In this paper we study one of the most common variants of the problem, where the goal is to find a point-to-point shortest path in a directed graph. We refer to this problem as *the P2P problem*. We assume that for the same underlying network, the problem will be solved repeatedly. Thus, we allow preprocessing, with the only restriction that the additional *space* used to store precomputed data is limited: linear in the graph size with a small constant factor.[1] Our goal is a fast algorithm for answering point-to-point shortest path queries. A natural application of the P2P problem is providing driving directions, for example services like Mapquest, Yahoo! Maps and Microsoft MapPoint, and some GPS devices. One can spend time preprocessing maps for these applications, but the underlying graphs are very large, so memory usage much exceeding the graph size is prohibitive. This motivates the linear-space assumption.

Shortest path problems have been extensively studied. The P2P problem with no preprocessing has been addressed, for example, in [17, 25, 27, 33]. 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 superlinear preprocessing space. The best bound in this context (see [9]) is superlinear in the output path size unless the path is very long. Preprocessing using geometric information and hierarchical decomposition is discussed in [15, 26, 31]. Other related work includes algorithms for the single-source shortest path problem, such as [1, 4, 5, 6, 10, 11, 12, 13, 14, 18, 22, 29, 32], and algorithms for approximate shortest paths that use preprocessing [3, 19, 30].

Usually one can solve the P2P problem while searching only a small portion of the graph; the algorithm's running time then depends only on the number of visited vertices. This motivates an *output-sensitive* complexity measure that we adopt. We measure algorithm performance as a function of the number of vertices on the output path. Note that this measure has the additional benefit of being machine-independent.

In Artificial Intelligence settings, one often needs to find a solution in a huge search space. The classical A^* *search*[2] [7, 16] technique often finds a solution while searching a small subspace. A^* search uses estimates on distances to the destination to guide vertex selection in a search from the source. Pohl [25] studied the relationship between A^* search and Dijkstra's algorithm in the context of the P2P problem. He observed that if the bounds used in A^* search are *feasible* (as defined in section 2), A^* search is equivalent to Dijkstra's algorithm on a graph with nonnegative arc lengths and therefore finds the optimal path. In classical applications of A^* search to the P2P problem, distance bounds are implicit in the domain description, with no preprocessing required. For example, for Euclidean graphs, the Euclidean distance between two vertices (which is a part of the domain knowledge) gives a lower bound on the distance between them.

Our first contribution is a new preprocessing-based technique for computing distance bounds. The preprocessing entails carefully choosing a small (constant) number of landmarks, then computing and storing shortest path distances between all vertices and each of these landmarks. Lower bounds are computed in constant time using these distances in combination with the triangle inequality. These lower bounds yield a new class of algorithms, which we call *ALT algorithms* since they are based on A^* search, landmarks, and the triangle inequality. Here we are talking about the triangle inequality with respect to the shortest path distances in the graph, not an embedding in Euclidean space or some other metric, which need not be present. Our experimental results show that ALT algorithms are very efficient on several important graph classes.

To illustrate just how effective our approach can be, consider a square grid with integral arc lengths selected uniformly at random from the interval $\{100, \ldots, 150\}$. Figure 1 shows the area searched by three different algorithms. Dijkstra's algorithm searches a large "Manhattan ball" around the source. Note that

[1] During preprocessing, we either determine that the graph has a negative cycle and the problem is infeasible, or replace the input length function by an equivalent nonnegative one. Thus we assume, without loss of generality, that the input length function is nonnegative.

[2] Also known as *heuristic search*.

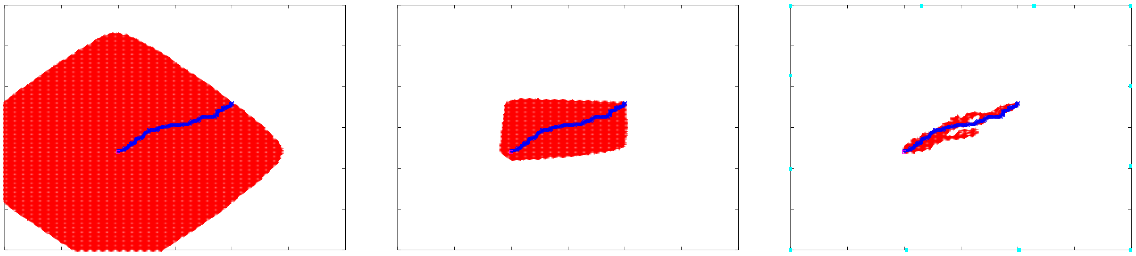

Figure 1: Vertices visited by Dijkstra's algorithm (left), A^* search with Manhattan lower bounds (middle), and an ALT algorithm (right) on the same input.

for a pair of points, the Manhattan distance between them times 100 is a decent lower bound on the true distance, and that for these points the bounding rectangle contains many near-shortest paths. With these observations in mind, one would expect that A^* search based on Manhattan distance bounds will be able to prune the search by an area slightly larger than the bounding box, and it fact it does. However, in spite of many near-optimal paths, our ALT algorithm is able to prune the search to an area much smaller than the bounding box.

The intuitive reason for good performance is in tune with our use of AI techniques. During preprocessing, our algorithm learns about useful paths and encodes the resulting knowledge in the landmark distances. During shortest path computations, this knowledge is used to direct the search. For example, suppose we want to find a shortest path from our office in Mountain View, California, to the Squaw Valley ski resort in the Lake Tahoe area (also in California). Suppose further that we have a landmark in New York City. The best way to get to New York, which we precomputed, is to get to highway 80 east and follow it to New York. The best way to get to Squaw Valley is also to get to highway 80 east and then exit in the Truckee area and follow a local highway. These two routes share a common initial segment that takes us very close to our destination ski resort. Our algorithm takes advantage of the knowledge of the precomputed route in an implicit, but very efficient, way by using the stored distances in lower bound computations. In effect, the edges in the shared segment appear to the algorithm to have *zero* length. Note that a landmark in New York helps us even though it is over ten times further away than our destination. Because of this phenomenon, our ALT algorithms get very good performance with only a small number of landmarks.

Although landmark-based bounds have been previously used for some variants of shortest path computation (see e.g. [3]), the previous bounds are not feasible and cannot be used to obtain an exact A^* search algorithm. Our bounds are feasible.

Proper landmark selection is important to the quality of the bounds. As our second contribution, we give several algorithms for selecting landmarks. While some of our landmark selection methods work on general graphs, others take advantage of additional information, such as geometric embeddings, to obtain better domain-specific landmarks. Note that landmark selection is the only part of the algorithm that may use domain knowledge. For a given set of landmarks, no domain knowledge is required.

As we shall see, making bidirectional A^* search work correctly is nontrivial. Pohl [25] and Ikeda et al. [17] give two ways of combining A^* search with the bidirectional version of Dijkstra's method [4, 8, 23] to get provably optimal algorithms. Our third contribution is an improvement on Pohl's algorithm and an alternative to the Ikeda et al. algorithm. Our bidirectional ALT algorithms outperform those of Pohl and Ikeda et al. (which use Euclidean bounds).

Our fourth contribution is an experimental study comparing the new and the previously known algorithms on synthetic graphs and on real-life road graphs taken from Microsoft's MapPoint database. We study which variants of ALT algorithms perform best in practice, and show that they compare very well to previous algorithms. Our experiments give insight into how ALT algorithm efficiency depends on the number of landmarks, graph size, and graph structure. Some of the methodology we use is new and may prove helpful in future work in the area.

Our output-sensitive way of measuring performance emphasizes the efficiency of our algorithms and shows how much room there is for improvement, assuming that any P2P algorithm examines at least the vertices on the shortest path. For our best algorithm running on road graphs, the average number of vertices scanned varies between 4 and 30 times the number of vertices on the shortest path, over different types of origin-destination pair distributions (for most graphs in our test set, it is closer to 4 than to 30). For example, to find a shortest path with $1,000$ vertices on a graph with $3,000,000$ vertices, our algorithm typically scans only $10,000$ vertices (10 scanned vertices for every shortest path vertex) which is a tiny fraction of the total number of vertices. Furthermore, the algorithm almost never performs much worse than the average efficiency, and occasionally performs much better. Good ALT algorithms are one or more orders of magnitude more efficient than the bidirectional variant of Dijkstra's algorithm and Euclidean distance-based algorithms.

2 Preliminaries

The input to the P2P problem is a directed graph with n vertices, m arcs, a source vertex s, a sink vertex t, and nonnegative lengths $\ell(a)$ for each arc a. Our goal in the P2P problem is to find the shortest path from s to t.

Let $\mathrm{dist}(v, w)$ denote the shortest-path distance from vertex v to vertex w with respect to ℓ. We will often use edge lengths other than ℓ, but $\mathrm{dist}(\cdot, \cdot)$ will always refer to the original arc lengths. Note that in general $\mathrm{dist}(v, w) \neq \mathrm{dist}(w, v)$. In this paper we assume that arc lengths are real-valued, unless mentioned otherwise.

A *potential function* is a function from vertices to reals. Given a potential function π, we define the reduced cost of an edge by

$$\ell_\pi(v, w) = \ell(v) - \pi(v) + \pi(w).$$

Suppose we replace ℓ by ℓ_π. Then for any two vertices x and y, the length of any x-y path changes by the same amount, $\pi(y) - \pi(x)$ (the other potentials telescope). Thus a path is a shortest path with respect to ℓ iff it is a shortest path with respect to ℓ_π, and the two problems are equivalent.

For a constant c, we define a *shift by* c to be a transformation that replaces $\pi(v)$ by $\pi(v) - c$ for all vertices. Shifts do not change reduced costs.

We say that π is *feasible* if ℓ_π is nonnegative for all vertices. The following fact is well-known:

Lemma 2.1 *Suppose π is feasible and for a vertex $t \in V$ we have $\pi(t) \leq 0$. Then for any $v \in V$, $\pi(v) \leq dist(v, t)$.*

Thus in this case we think of $\pi(v)$ as a lower bound on the distance from v to t.

Also observe that the maximum of two feasible potential functions is feasible.

Lemma 2.2 *If π_1 and π_2 are feasible potential functions, then $p = \max(\pi_1, \pi_2)$ is a feasible potential function.*

Proof. Consider $(v, w) \in E$. Feasibility of π_1 and π_2 implies that $\ell(v, w) - \pi_1(v) + \pi_1(w) \geq 0$ and $\ell(v, w) - \pi_2(v) + \pi_2(w) \geq 0$. Suppose $\pi_1(v) \geq \pi_2(v)$; the other case is symmetric. If $\pi_1(w) \geq \pi_2(w)$, then $\ell(v, w) - p(v) + p(w) = \ell(v, w) - \pi_1(v) + \pi_1(w) \geq 0$. Otherwise

$$\ell(v, w) - p(v) + p(w) = \ell(v, w) - \pi_1(v) + \pi_2(w) \geq \ell(v, w) - \pi_1(v) + \pi_1(w) \geq 0.$$

■

One can also combine feasible potential functions by taking the minimum, or, as observed in [17], the average or any convex linear combination of feasible potential functions. We use the maximum in the following context. Given several feasible lower bound functions, we take the maximum of these to get a feasible lower bound function that at any vertex is at least as high as each original function.

3 The labeling method and Dijkstra's algorithm

The labeling method for the shortest path problem [20, 21] finds shortest paths from the source to all vertices in the graph. The method works as follows (see for example [28]). It maintains for every vertex v its distance label $d_s(v)$, parent $p(v)$, and status $S(v) \in \{\texttt{unreached}, \texttt{labeled}, \texttt{scanned}\}$. Initially $d_s(v) = \infty$, $p(v) = nil$, and $S(v) = \texttt{unreached}$ for every vertex v. The method starts by setting $d_s(s) = 0$ 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 $d_s(w) > d_s(v) + \ell(v, w)$ and, if true, sets $d_s(w) = d_s(v) + \ell(v, w)$, $p(w) = v$, and $S(w) = \texttt{labeled}$.

If the length function is nonnegative, the labeling method always terminates with correct shortest path distances and a shortest path tree. The efficiency of the method depends on the rule to chose a vertex to scan next. We say that $d_s(v)$ is *exact* if the distance from s to v is equal to $d_s(v)$. It is easy to see that if the method always selects a vertex v such that, at the selection time, $d_s(v)$ is exact, then each vertex is scanned at most once. Dijkstra [6] (and independently Dantzig [4]) observed that if ℓ is nonnegative and v is a labeled vertex with the smallest distance label, then $d_s(v)$ is exact. We refer to the scanning method with the minimum labeled vertex selection rule as Dijkstra's algorithm for the single-source problem.

Theorem 3.1 *[6] If ℓ is nonnegative then Dijkstra's algorithm scans vertices in nondecreasing order of their distances from s and scans each vertex at most once.*

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

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

The *bidirectional algorithm* [4, 8, 23] works as follows. It alternates between running the forward and reverse version of Dijkstra's algorithm. We refer to these as the forward and the reverse search, respectively. During initialization, the forward search scans s and the reverse search scans t. In addition, the algorithm maintains the length of the shortest path seen so far, μ, and the corresponding path as follows. Initially $\mu = \infty$. When an arc (v, w) is scanned by the forward search and w has already been scanned in the reversed direction, we know the shortest s-v and w-t paths of lengths $d_s(v)$ and $d_t(w)$, respectively. If $\mu > d_s(v) + \ell(v, w) + d_t(w)$, we have found a shorter path than those seen before, so we update μ and its path accordingly. We do similar updates during the reverse search. The algorithm terminates when the search in one directing selects a vertex that has been scanned in the other direction.

Note that any alternation strategy works correctly. We use the one that balances the work of the forward and reverse searches. One can show that this strategy is within a factor of two of the optimal off-line strategy. Also note that a common mistake in defining the bidirectional algorithm is to assume that if the algorithm stops at vertex v, then the shortest path goes through v. This is not necessarily the case. However, upon termination the path yielding μ is optimal.

Theorem 3.2 *[25] If the sink is reachable from the source, the bidirectional algorithm finds an optimal path, and it is the path stored along with μ.*

Intuitively, the bidirectional algorithm searches two touching balls centered at s and t. To understand why this algorithm usually outperforms Dijkstra's algorithm, consider an infinite k dimensional grid with each vertex connected to its neighbors by an arc of length one. If the s-t distance is D, Dijkstra's algorithm visits about $(2D)^k$ vertices versus $2 \cdot D^k$ for the bidirectional algorithm. In this case, the bidirectional algorithm gives a factor 2^{k-1} speedup.

4 A^* Search

Consider the problem of looking for a path from s to t and suppose we have a (perhaps domain-specific) function $\pi_t : V \to \mathbb{R}$ such that $\pi_t(v)$ gives an estimate on the distance from v to t. In the context of this paper, A^* *search* is an algorithm that works like Dijkstra's algorithm, except that at each step it selects a labeled vertex v with the smallest value of $k(v) = d_s(v) + \pi_t(v)$ 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_t}. If π_t is feasible, ℓ_{π_t} is nonnegative and Theorem 3.1 holds.

Note that the selection rule used by A^* search is a natural one: the chosen vertex is on an s-t path with the shortest estimated length. In particular, if π_t gives exact distances to t, the algorithm scans only vertices on shortest paths from s to t, and if the shortest path is unique, the algorithm terminates after scanning exactly the vertices on the shortest path except t. Intuitively, the better the estimates, the fewer vertices are scanned.

We refer to the class of A^* search algorithms that use a feasible function π_t with $\pi_t(t) \leq 0$ as *lower-bounding algorithms*.

5 Bidirectional Lower-Bounding Algorithms

In this section we show how to combine the ideas of bidirectional search and A^* search. This seems trivial: just run the forward and the reverse searches and stop as soon as they meet. This does not work, however.

Let π_t be a potential function used in the forward search and let π_s be one used in the reverse search. Since the latter works in the reversed graph, each arc $(v, w) \in E$ appears as (w, v), and its reduced cost w.r.t. π_s is $\ell_{\pi_s}(w, v) = \ell(v, w) - \pi_s(w) + \pi_s(v)$, where $\ell(v, w)$ is in the original graph.

We say that π_t and π_s are *consistent* if for all arcs (v, w), $\ell_{\pi_t}(v, w)$ in the original graph is equal to $\ell_{\pi_s}(w, v)$ in the reverse graph. This is equivalent to $\pi_t + \pi_s = \text{const}$.

It is easy to come up with lower-bounding schemes for which π_t and π_s are not consistent. If they are not, the forward and the reverse searches use different length functions. Therefore when the searches meet, we have no guarantee that the shortest path has been found.

One can overcome this difficulty in two ways: develop a new termination condition, or use consistent potential functions. We call the algorithms based on the former and the latter approaches *symmetric* and *consistent*, respectively. Each of these has strengths and weaknesses. The symmetric approach can use the best available potential functions but cannot terminate as soon as the two searches meet. The consistent approach can stop as soon as the searches meet, but the consistency requirement restricts the potential function choice.

5.1 Symmetric Approach

The following symmetric algorithm is due to Pohl [25]. Run the forward and the reverse searches, alternating in some way. Each time a forward search scans an arc (v, w) such that w has been scanned by the reverse search, see if the concatenation of the s-t path formed by concatenating the shortest s-v path found by the forward search, (v, w), and the shortest w-t path found by the reverse search, is shorter than best s-t path found so far, and update the best path and its length, μ, if needed. Also do the corresponding updates during the reverse search. Stop when one of the searches is about to scan a vertex v with $k(v) \geq \mu$ or when both searches have no labeled vertices. The algorithm is correct because the search must have found the shortest path by then.

Our symmetric algorithm is an improvement on Pohl's algorithm. When the forward search scans an arc (v, w) such that w has been scanned by the reverse search, we do nothing to w. This is because we already know the shortest path from w to t. This prunes the forward search. We prune the reverse search similarly. We call this algorithm the *symmetric lower-bounding algorithm*.

Theorem 5.1 *If the sink is reachable from the source, the symmetric lower-bounding algorithm finds an optimal path.*

Proof. If the sink is reachable from the source, the set of labeled vertices is nonempty while $\mu = \infty$, and therefore the algorithm stops with a finite value of μ and finds some path. Let P be the path found and let μ', the final value of μ, be its length.

Suppose for contradiction that a shortest path, Q, is shorter that P. Then for every vertex v on Q, we have $\text{dist}(s,v)+\pi_t(v) \leq \ell(Q) < \mu'$ and $\text{dist}(v,t)+\pi_s(v) \leq \ell(Q) < \mu'$. Therefore each vertex on Q has been scanned by one of the searches. Since s is scanned by the forward search and t by the backward search, there must be an arc (v,w) on Q such that v was scanned by the forward search and w by the backward search. Then the arc (v,w) has been scanned, and during this scan μ was updated to $\ell(Q)$. Since μ never increases in value, $\mu' \leq \ell(Q)$; this is a contradiction. ∎

5.2 Consistent Approach

Given a potential function p, a consistent algorithm uses p for the forward computation and $-p$ (or its shift by a constant, which is equivalent correctness-wise) for the reverse one. These two potential functions are consistent; the difficulty is to select a function p that works well.

Let π_t and π_s be feasible potential functions giving lower bounds to the source and from the sink, respectively. Ikeda et al. [17] use $p_t(v) = \frac{\pi_t(v)-\pi_s(v)}{2}$ as the potential function for the forward computation and $p_s(v) = \frac{\pi_s(v)-\pi_t(v)}{2} = -p_t(v)$ for the reverse one. We refer to this function as the *average function.* (They also observed that any convex combination of p_t and p_s can be used.) They show that the consistent algorithm that uses this function with Euclidean lower bound functions π_t and π_s outperforms the standard bidirectional algorithm on certain graphs. However, the improvement is relatively modest.

Notice that each of p_t and $-p_s$ is feasible in the forward direction. Thus $p_s(t)-p_s(v)$ gives lower bounds on the distance from v to t, although not necessarily good ones. Feasibility of the average of p_t and $-p_s$ is obvious. Slightly less intuitive is the feasibility of the maximum, as shown in Lemma 2.2.

We define an alternative potential function p_t by $p_t(v) = \max(\pi_t(v), \pi_s(t)-\pi_s(v)+\beta)$, where for a fixed problem β is a constant that depends on $\pi_t(s)$ and/or $\pi_s(t)$ (our implementation uses a constant fraction of $\pi_t(s)$). It is easy to see that p_t is a feasible potential function. We refer to this function as the *max function.*

The intuition for why the max function is a good choice is as follows. Both $\pi_t(v)$ and $\pi_s(t) - \pi_s(v)$ are lower bounds on the distance from v to t. Since π_t is specifically designed to be a lower bound on distances to t and π_s is a lower bound on distances from s converted into a lower bound on distances to t, $\pi_t(v)$ will be significantly bigger than $\pi_s(t) - \pi_s(v)$ for v far away from t, in particular for v near s. Therefore for v around s, $\pi_t(v)$ will tend to determine the value of p and for an initial period, the forward search will behave like the one that uses π_t. Since $\pi_t(t) = 0$ and $\pi_t(t) - \pi_t(t) + \beta = \beta > 0$, in the vicinity of t the second term will dominate π_t and therefore determine the value of $-p_t$. Thus for v around t, the reverse search will be directed by a shift of π_s and will behave like the one that uses π_s. Choosing β properly will balance the two sides so that as few vertices total as possible are scanned.

In our experiments, the average function had a somewhat better performance than the max function. However, its performance was close, and it may perform better with some landmark selection heuristics.

6 Computing Lower Bounds

Previous implementations of the lower bounding algorithm used information implicit in the domain, like Euclidean distances for Euclidean graphs, to compute lower bounds. We take a different approach. We select a small set of *landmarks* and, for each vertex, precompute distances to and from every landmark. Consider a landmark L and let $d(\cdot)$ be the distance *to* L. Then by the triangle inequality, $d(v) - d(w) \leq$

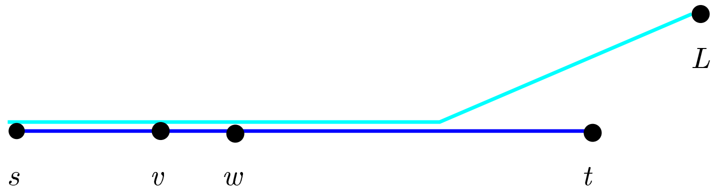

Figure 2: Why our ALT algorithms work well.

dist(v, w). Similarly, if $d(\cdot)$ is the distance *from* L, $d(w) - d(v) \leq \text{dist}(v, w)$. To compute the tightest lower bound, one can take the maximum, for all landmarks, over these bounds.

Usually it is better to use only some of the landmarks. First, this is more efficient. Second, a tighter bound is not necessarily better for the search. Intuitively, a landmark away from the shortest s-t path may attract the search towards itself at some point, which may not be desirable. For a given s and t, we select a fixed-size subset of landmarks that give the highest lower bounds on the s-t distance. During the s-t shortest path computation, we limit ourselves to this subset when computing lower bounds.

To get an understanding of why the ALT algorithms often work well, suppose we have a map with s and t far from each other, and a landmark L so that t is approximately between s and L. It is likely that the shortest route from s to L consists of a segment from s to a highway, a segment that uses highways only, and a segment from a highway to L. Furthermore, the shortest route to t follows the same path to the highway and goes on the same highway path for a while, but exits earlier and takes local roads to t. In other words, for a good choice of L, the shortest paths from s to L and t share an initial segment. See Figure 2. Consider an arc (v, w) on this segment. It is easy to see that the lower bound π given by the distances to L and the triangle inequality has the following property: $\ell_\pi(v, w) = 0$. Thus for the shortest path from s, the reduced costs of arcs on the shared path segment are zero, so these arcs will be the first ones the ALT algorithm will scan.

This argument gives intuition for why the bidirectional algorithms work so well when the landmarks are well-chosen. Both backward and forward searches follow zero reduced costs path for a while. In the consistent algorithm case, the paths sometimes meet and the resulting path is the shortest path. However, the paths need not meet, but they usually come close to each other, and the two searches expand the paths and meet quickly. For the symmetric algorithm, the searches cannot stop even if the paths meet, but still the searches are biased towards each other and the algorithm terminates faster than the bidirectional algorithm.

7 Landmark Selection

Finding good landmarks is critical for the overall performance of lower-bounding algorithms. Let k denote the number of landmarks we would like to choose. The simplest way of selecting landmarks is to select k landmark vertices at random. This works reasonably well, but one can do better.

One greedy landmark selection algorithm works as follows. Pick a start vertex and find a vertex v_1 that is farthest away from it. Add v_1 to the set of landmarks. Proceed in iterations, at each iteration finding a vertex that is farthest away from the current set of landmarks and adding the vertex to the set. This algorithm can be viewed as a quick approximation to the problem of selecting a set of k vertices so that the minimum distance between a pair of selected vertices is maximized. Call this method the *farthest* landmark selection.

For road graphs and other geometric graphs, having a landmark geometrically lying behind the destination tends to give good bounds. Consider a map or a graph drawing on the plane where graph and geometric distances are strongly correlated.[3] A simple planar landmark selection algorithm works as fol-

[3]The graph does not need to be planar; for example, road networks are nonplanar.

lows. First, find a vertex c closest to the center of the embedding. Divide the embedding into k pie-slice sectors centered at c, each containing approximately the same number of vertices. For each sector, pick a vertex farthest away from the center. To avoid having two landmarks close to each other, if we processed sector A and are processing sector B such that the landmark for A is close to the border of A and B, we skip the vertices of B close to the border. We refer to this as *planar* landmark selection.

The above three selection rules are relatively fast, and one can optimize them in various ways. In the *optimized farthest landmark selection* algorithm, for example, we repeatedly remove a landmark and replace it with the farthest one from the remaining set of landmarks.

Another optimization technique for a given set of landmarks is to remove a landmark and replace it by the best landmark in the set of *candidate landmarks*. To select the best candidate, we compute a score for each landmark and select one with the highest score. We use a fixed sample of vertex pairs to compute scores. For each pair in the sample, we compute the distance lower bound b as the maximum over the lower bounds given by the current landmarks. Then for each candidate, we compute the lower bound b' given by it. If the $b' > b$, we add $b' - b$ to the candidate's score.[4] To obtain the sample of vertex pairs, for each vertex we chose a random one and add the pair to the sample.

We use this technique to get *optimized random* and *optimized planar* landmark selection. In both cases, we make passes over landmarks, trying to improve a landmark at each step. For the former, a set of candidates for a given landmark replacement contains the landmark and several other randomly chosen candidates. For the latter, we use a fixed set of candidates for each sector. We divide each sector into subsectors and choose the farthest vertex in each subsector to be a candidate landmark for the sector. In our implementation the total number of candidates (over all sectors) is 64.

Optimized landmark selection strategies can be computationally expensive. The optimized planar selection is especially expensive and takes hours to compute for the biggest problems in our tests. This selection rule, however, is superior to regular planar selection, and in fact is our best landmark selection rule for graphs with a given planar layout.

Optimized farthest selection, however, does not seem to improve on the regular one. Regular farthest selection is relatively efficient, and would be our choice for road networks if the layout information were not available. Optimized random selection is usually superior to regular random selection, but has lower overall performance than farthest selection and takes longer to produce landmarks.

8 Experimental Setup

8.1 Problem Families

Short name	# of vertices	# of arcs	Description	Latitude/longitude range
M_1	267,403	631,964	New Mexico	[34,37]/[-107,-103]
M_2	330,024	793,681	San Francisco Bay Area	[37,39]/[-123,-121]
M_3	563,992	1,392,202	Los Angeles area	[33,35]/[-120,-115]
M_4	588,940	1,370,273	St. Louis area	[37,40]/[-92,-88]
M_5	639,821	1,522,485	Dallas area	[31,34]/[-98,-94]
M_6	1,235,735	2,856,831	US west coast	[33,45]/[-130,-120]
M_7	2,219,925	5,244,506	Rocky mountains/plains	[33,45]/[-110,-100]
M_8	2,263,758	5,300,035	Western US	[33,45]/[-120,-110]
M_9	4,130,777	9,802,953	Central US	[33,45]/[-100,-90]
M_{10}	4,469,462	10,549,756	US east coast	[33,45]/[-80,-70]
M_{11}	6,687,940	15,561,631	Central-eastern US	[33,45]/[-90,-80]

Table 1: Road network problem descriptions, sorted by size.

[4] We also tried adding 1 instead, which seems to produce worse landmarks.

We ran experiments on road graphs and on several classes of synthetic problems. The road graphs are subgraphs of the graph used in MapPoint. The full MapPoint graph includes all of the roads, local and highway, in North America. There is one vertex for each intersection of two roads and one directed arc for each road segment. There are also degree two vertices in the middle of some road segments, for example where the segments intersect the map grid. Each vertex has a latitude and a longitude, and each road segment has a speed limit and a length. The full graph is too big for the computer used in our experiments, so we ran experiments on smaller subgraphs. Our subgraphs are created by choosing only the vertices inside a given rectangular range of latitudes and longitudes, then reducing to the largest strongly connected component of the corresponding induced subgraph. For bigger graphs, we took vertices between 33 and 45 degrees of Northern longitude and partitioned them into regions between 130–120, 120–110, 110–100, 100–90, 90–80, and 80–70 degrees Western latitude. This corresponds roughly to the dimensions of the United States. Smaller graphs correspond to the New Mexico, San Francisco, Los Angeles, St. Louis and Dallas metropolitan areas.

Table 1 gives more details of the graphs used, as well as the shorthand names we use to report data. This leaves open the notion of distance used. For each graph, we used two natural distance notions:

TRANSIT TIME: Distances are calculated in terms of the time needed to traverse each road, assuming that one always travels at the speed limit.

DISTANCE: Distances are calculated according to the actual Euclidean length of the road segments.

The synthetic classes of graphs used are as follows:

GRID: For n total vertices, this is a directed $\sqrt{n} \times \sqrt{n}$ grid graph, with each vertex connected to its neighbor above, below, to the left, and to the right (except the border vertices, which have fewer connections). Each edge weight is an integer chosen uniformly at random from the set $\{1, \ldots, M\}$ for $M \in \{10, 1000, 100000\}$. Note that this graph is directed, since for adjacent vertices v and w, $\ell(v, w)$ is not necessarily the same as $\ell(w, v)$.

We tested on square grids of side-length 256, 512, 1024 and 2048. Let G_{ij} denote the grid graph which is $256 \cdot 2^{i-1}$ on a side, and has edge weights randomly chosen from $\{1, \ldots, 10 \cdot 10^j\}$. For example, G_{23} is a 512×512 grid with edge weights chosen uniformly at random from $\{1, \ldots, 100000\}$.

RANDOM: For n vertices and m arcs, this is a random directed multigraph $G(n, m)$ with exactly m arcs, where each edge is chosen independently and uniformly at random. Each edge weight is an integer chosen uniformly at random from the set $\{1, \ldots, M\}$, for $M \in \{10, 1000, 100000\}$.

We tested on average degree four random graphs with 65536, 262144, 1048576 and 4194304 vertices. Let R_{ij} denote a random directed graph with $65536 \cdot 2^{i-1}$ vertices, $4 \cdot 65536 \cdot 2^{i-1}$ arcs, and edge weights chosen uniformly at random from $\{1, \ldots, 10 \cdot 10^j\}$.

Each of these is a natural family of graphs.

For a given graph, we study two distributions of s, t pairs:

RAND: In this distribution, we select s and t uniformly at random among all vertices. This natural distribution has been used previously (e.g., [32]). It produces "hard" problems for the following reason: s and t tend to be far apart when chosen this way, thus forcing Dijkstra's algorithm to visit most of the graph.

BFS: This distribution is more local. In this distribution, we chose s at random, run breadth-first search from s to find all vertices that are k arcs away from s, and chose one of these vertices uniformly at random. On road and grid graphs, we use $k = 50$. Note that the corresponding shortest paths tend to have between 50 and 100 arcs. On road networks, this corresponds to trips on the order of an hour, where one passes through 50 to a 100 road segments. In this sense it is a more "typical" distribution. On random graphs we use $k = 6$ because these graphs have small diameters.

Although we compared all variants of regular and bidirectional search, we report only on the most promising or representative algorithms.

D: Dijkstra's algorithm, to compare with the bidirectional algorithm.

AE: A^* search with Euclidean lower bounds. This was previously studied in [25, 27].

AL: Regular ALT algorithm.

B: The bidirectional variant of Dijkstra's algorithm, to provide a basis for comparison.

BEA: The bidirectional algorithm with a consistent potential function based on average Euclidean bounds. This was previously studied in [17].

BLS: The symmetric bidirectional ALT algorithm.

BLA: The consistent bidirectional ALT algorithm with the average potential function.

BLM: The consistent bidirectional ALT algorithm with the max potential function.

8.2 Landmark Selection Algorithms Tested

In our work we compared six different landmark selection algorithms from three general approaches. The algorithms are described in section 7 and are as follows:

R : random;

R2 : optimized random;

F : farthest;

P : planar;

P2 : optimized planar.

8.3 Implementation Choices

Euclidean bounds. For road networks, exact Euclidean bounds offer virtually no help, even for the distance-based length function. To get noticeable improvement, one needs to scale these bounds up. This is consistent with comments in [17]. Such scaling may result in nonoptimal paths being found. Although we are interested in exact algorithms, we use aggressive scaling parameters, different for distance- and time-based road networks. Even though the resulting codes sometimes find paths that are longer than the shortest paths (on the average by over 10% on some graphs), the resulting algorithms are not competitive with landmark-based ones.

Landmark selection. When comparing algorithms, we set the number of landmarks to 16 with the P2 landmark selection algorithm when it is applicable, and the F algorithm otherwise (for the case of random graphs). We use the P2 algorithm because it has the best efficiency on almost all test cases (see section 9.6), and 16 because is the maximum number that fits in memory for our biggest test problem.

Name	D	AE	AL	B	BEA	BLS	BLM	BLA
M_1	0.5 (116.4)	0.5 (126.7)	6.5 (202.7)	0.7 (194.8)	0.8 (176.1)	8.2 (148.3)	14.9 (117.0)	16.9 (146.5)
	62.6	*110.2*	*7.2*	*43.1*	*120.4*	*5.7*	*5.6*	*5.2*
M_2	0.3 (127.9)	0.3 (162.0)	3.0 (544.0)	0.4 (173.7)	0.4 (173.7)	3.5 (311.0)	6.1 (253.2)	7.8 (305.3)
	82.4	*135.2*	*12.8*	*59.8*	*163.6*	*13.5*	*13.5*	*11.6*
M_3	0.2 (90.5)	0.2 (86.7)	2.5 (418.6)	0.3 (95.4)	0.3 (140.1)	2.8 (326.1)	4.3 (350.0)	5.6 (365.1)
	173.1	*324.9*	*19.9*	*111.9*	*290.1*	*20.2*	*22.7*	*18.3*
M_4	0.3 (93.7)	0.3 (102.5)	4.1 (319.2)	0.4 (109.9)	0.4 (113.8)	5.0 (211.5)	8.4 (180.6)	10.6 (206.1)
	162.2	*353.1*	*14.2*	*108.5*	*341.4*	*12.3*	*13.3*	*11.5*
M_5	0.2 (101.6)	0.2 (229.5)	3.3 (449.4)	0.4 (134.6)	0.4 (161.3)	3.8 (343.4)	5.2 (235.0)	7.1 (285.5)
	180.5	*321.1*	*18.4*	*119.6*	*310.6*	*17.7*	*24.7*	*18.1*
M_6	0.3 (160.2)	0.3 (301.2)	2.3 (411.3)	0.3 (285.4)	0.3 (224.4)	3.0 (381.5)	6.6 (301.2)	8.6 (334.0)
	338.3	*613.7*	*63.5*	*349.1*	*922.0*	*51.5*	*44.3*	*36.3*
M_7	0.1 (65.1)	0.2 (91.8)	2.7 (542.1)	0.2 (84.5)	0.2 (96.8)	3.0 (411.7)	4.3 (402.3)	5.6 (401.1)
	692.3	*1197.0*	*50.4*	*537.0*	*1328.0*	*51.9*	*64.9*	*54.2*
M_8	0.2 (88.3)	0.2 (108.9)	2.8 (521.7)	0.2 (141.5)	0.2 (213.4)	3.1 (344.7)	4.9 (322.9)	6.0 (403.5)
	679.9	*1303.1*	*62.1*	*557.1*	*1456.6*	*68.7*	*72.3*	*62.5*
M_9	0.1 (109.8)	0.1 (164.1)	2.1 (393.5)	0.1 (169.8)	0.2 (196.9)	2.2 (265.8)	3.9 (316.2)	5.5 (300.1)
	1362.4	*2489.0*	*87.0*	*1074.1*	*2631.4*	*92.4*	*90.4*	*69.6*
M_{10}	0.1 (78.0)	0.1 (214.0)	1.7 (762.0)	0.1 (83.4)	0.1 (103.5)	2.0 (462.6)	3.7 (356.0)	4.7 (491.6)
	1474.6	*2606.1*	*138.6*	*1240.3*	*2912.3*	*122.3*	*108.5*	*97.6*
M_{11}	0.1 (110.8)	0.1 (147.5)	1.7 (299.4)	0.1 (174.6)	0.1 (193.7)	1.7 (266.3)	2.5 (511.9)	3.6 (531.6)
	2450.2	*4431.4*	*152.9*	*1999.3*	*4813.6*	*188.2*	*213.4*	*161.0*

Table 2: Algorithm comparison for the RAND source-destination distribution on road networks with TRANSIT TIME distances and 16 landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics. Standard deviations (% of mean) are indicated in parentheses.

Name	D	AE	AL	B	BEA	BLS	BLM	BLA
M_1	0.44	0.46	5.34	0.67	0.69	7.43	13.13	13.51
	57.14	*112.42*	*8.01*	*41.49*	*121.18*	*5.91*	*6.12*	*6.25*
M_2	0.26	0.28	3.02	0.37	0.38	3.74	5.93	6.45
	66.38	*140.90*	*13.05*	*53.93*	*228.17*	*18.18*	*22.08*	*19.19*
M_3	0.17	0.18	2.90	0.29	0.29	3.16	5.77	7.22
	137.46	*326.12*	*15.50*	*94.14*	*290.00*	*15.58*	*14.03*	*11.88*
M_4	0.24	0.24	3.82	0.38	0.39	4.48	6.90	10.71
	139.34	*353.95*	*14.20*	*96.91*	*339.57*	*13.56*	*15.76*	*10.42*
M_5	0.22	0.23	4.21	0.35	0.36	4.29	5.23	7.70
	240.52	*521.61*	*13.04*	*175.21*	*319.05*	*14.39*	*21.93*	*14.98*
M_6	0.25	0.26	2.39	0.29	0.30	3.29	8.20	8.82
	281.19	*641.15*	*62.33*	*300.25*	*906.57*	*49.61*	*36.09*	*35.61*
M_7	0.14	0.15	3.13	0.20	0.21	3.61	6.58	7.56
	605.04	*1252.61*	*40.67*	*482.99*	*1332.02*	*38.77*	*38.75*	*36.19*
M_8	0.15	0.16	2.69	0.21	0.21	3.47	5.63	7.21
	579.59	*1325.01*	*59.78*	*492.49*	*1464.67*	*52.27*	*55.06*	*43.91*
M_9	0.09	0.10	1.87	0.14	0.14	2.02	3.27	3.87
	1208.30	*2565.61*	*92.88*	*954.08*	*2620.47*	*97.69*	*100.80*	*91.68*
M_{10}	0.10	0.10	1.56	0.14	0.14	1.91	3.31	4.69
	1249.86	*2740.81*	*147.54*	*1085.57*	*2958.20*	*146.22*	*132.64*	*102.53*
M_{11}	0.08	0.08	1.81	0.11	0.11	2.01	2.82	4.01
	2113.80	*4693.30*	*132.83*	*1736.52*	*4775.70*	*145.12*	*176.84*	*133.29*

Table 3: Algorithm comparison for the RAND source-destination distribution on road networks with DISTANCE distances and 16 landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	D	AE	AL	B	BEA	BLS	BLM	BLA
M_1	1.75	2.43	15.28	3.97	4.60	15.87	15.04	19.19
	1.64	*2.09*	*0.35*	*0.71*	*2.07*	*0.34*	*0.59*	*0.45*
M_2	0.93	1.33	8.03	1.77	2.04	8.76	8.73	11.83
	3.69	*4.53*	*0.79*	*1.89*	*5.37*	*0.78*	*1.29*	*0.86*
M_3	0.76	0.81	7.27	1.56	1.92	7.34	7.21	10.35
	4.35	*8.72*	*0.84*	*2.11*	*6.13*	*0.80*	*1.40*	*0.93*
M_4	1.65	1.90	18.89	3.37	3.56	19.62	18.13	24.23
	1.74	*3.21*	*0.32*	*0.88*	*2.74*	*0.29*	*0.50*	*0.38*
M_5	1.52	2.19	16.99	3.26	3.69	18.80	18.87	22.68
	1.87	*2.51*	*0.33*	*0.89*	*2.71*	*0.30*	*0.47*	*0.39*
M_6	1.38	2.51	12.35	2.72	3.48	14.61	12.17	17.70
	2.01	*2.08*	*0.44*	*1.10*	*2.84*	*0.37*	*0.73*	*0.51*
M_7	1.74	3.00	18.83	3.87	4.90	21.04	19.51	24.66
	1.60	*1.80*	*0.29*	*0.73*	*1.98*	*0.25*	*0.43*	*0.35*
M_8	1.36	1.23	10.01	2.77	3.36	11.39	9.28	14.02
	2.22	*5.66*	*0.57*	*1.08*	*3.28*	*0.51*	*0.99*	*0.63*
M_9	1.42	2.22	16.81	3.25	3.87	21.38	18.27	25.30
	2.00	*2.63*	*0.35*	*0.93*	*2.53*	*0.26*	*0.50*	*0.35*
M_{10}	1.15	1.98	9.78	2.65	3.33	10.73	8.80	13.86
	2.75	*2.89*	*0.63*	*1.17*	*3.27*	*0.56*	*1.14*	*0.68*
M_{11}	1.48	2.40	14.37	3.35	4.07	15.99	13.21	20.31
	1.97	*2.23*	*0.40*	*0.84*	*2.39*	*0.34*	*0.69*	*0.43*

Table 4: Algorithm comparison for the BFS source-destination distribution on road networks with TRANSIT TIME distances and 16 landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	D	AE	AL	B	BEA	BLS	BLM	BLA
M_1	1.74	2.21	16.20	3.73	4.12	19.22	16.97	22.54
	1.45	*2.29*	*0.32*	*0.70*	*2.13*	*0.26*	*0.47*	*0.37*
M_2	0.82	1.10	9.48	1.58	1.78	9.86	9.84	12.55
	3.17	*5.03*	*0.64*	*1.77*	*8.30*	*1.07*	*1.78*	*0.79*
M_3	0.69	0.78	8.36	1.35	1.57	8.48	8.18	10.84
	3.82	*8.55*	*0.67*	*2.07*	*6.83*	*0.63*	*1.09*	*0.81*
M_4	1.58	1.72	22.43	3.19	3.32	22.50	20.52	26.40
	1.61	*3.32*	*0.25*	*0.84*	*2.70*	*0.24*	*0.41*	*0.32*
M_5	1.46	1.89	17.96	3.02	3.28	22.83	22.67	26.47
	1.84	*4.70*	*0.30*	*0.99*	*2.88*	*0.23*	*0.38*	*0.33*
M_6	1.40	1.99	12.43	2.80	3.21	13.53	11.77	16.89
	1.70	*2.60*	*0.44*	*0.89*	*2.84*	*0.38*	*0.70*	*0.47*
M_7	1.63	2.42	17.63	3.57	4.19	17.72	16.62	19.82
	1.52	*2.21*	*0.29*	*0.70*	*2.20*	*0.30*	*0.51*	*0.43*
M_8	1.27	1.38	10.68	2.53	2.79	11.57	9.29	14.46
	1.99	*4.51*	*0.50*	*1.05*	*3.46*	*0.46*	*0.91*	*0.57*
M_9	1.37	1.83	19.55	3.05	3.43	22.42	19.98	24.93
	1.85	*3.16*	*0.29*	*0.85*	*2.71*	*0.24*	*0.42*	*0.33*
M_{10}	1.03	1.34	12.79	2.37	2.59	13.96	11.48	17.74
	2.49	*4.32*	*0.46*	*1.16*	*3.87*	*0.41*	*0.86*	*0.51*
M_{11}	1.59	1.93	18.45	3.58	3.83	19.76	17.46	22.38
	1.50	*2.71*	*0.33*	*0.71*	*2.32*	*0.30*	*0.54*	*0.40*

Table 5: Algorithm comparison for the BFS source-destination distribution on road networks with DISTANCE distances and 16 landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Data Structures. In implementing graph data structure, we used a standard cache-efficient representation of arc lists where for each vertex, its outgoing arcs are adjacent in memory.

Although in general we attempted to write efficient code, to facilitate flexibility we used the same graph data structure for all algorithms. For a given algorithm, some of the vertex-related data may be unused. For example, we store geographic information for each vertex, even though only the algorithms based on Euclidean bounds need it. The resulting loss of locality hurts algorithm performance somewhat, but probably not by more than 50%. Note, however, that we use running times as a supplement to a machine-independent measure of performance which is not affected by these issues.

9 Experimental Results

In this section we present experimental results. As a primary measure of algorithm performance, we use an output-sensitive measure we call *efficiency*. The efficiency of a run of a P2P algorithm is defined as the number of vertices on the shortest path divided by the number of vertices scanned by the algorithm.[5] We report efficiency in percent. An optimal algorithm that scans only the shortest path vertices has 100% efficiency. Note that efficiency is a machine-independent measure of performance.

We also report the average running times of our algorithms in milliseconds. Running times are machine- and implementation-dependent. In all our experiments, all data fits in main memory. When the graph fits in memory, factors like the lower bound computation time have more influence on the running time. In particular, Euclidean bound computation is somewhat expensive because of the floating point operations involved. Despite their shortcomings, running times are important and complement efficiency to provide a better understanding of practical performance of algorithms under consideration. Note that more complicated bidirectional search algorithms have somewhat higher overhead and need higher efficiency to compete with the corresponding regular search algorithms.

All experiments were run under Redhat Linux 9.0 on an HP XW-8000 workstation, which has 4GB of RAM and a 3.06 Ghz Pentium-4 processor. Due to limitations of the Linux kernel, however, only a little over 3GB was accessible to an individual process. Finally, all reported data points are the average of 128 trials.

For most algorithms, we used priority queues based on multi-level buckets [5, 13, 14]. For algorithms that use Euclidean bounds, we used a standard heap implementation of priority queues, as described in, for example, [2]. This is because these algorithms use aggressive bounds which can lead to negative reduced costs, making the use of monotone priority queues, such as multi-level buckets, impossible.

9.1 Algorithms Tested

Deviation bounds. The efficiency (and hence also running time) data presented in our tables has very high deviation relative to the mean; see Table 2. However, efficiency is almost never significantly below the mean, but sometimes is much greater than the mean. This is good because surprises in efficiency are almost entirely positive. Examples of this phenomenon are given in Figure 3. In order to avoid clutter, we omit standard deviations from other tables.

9.2 Road Networks

Tables 2, 3, 4 and 5 give data for road networks with both types of arc lengths and input distributions. All algorithms perform better under the BFS s-t distribution than under the RAND distribution. As expected, efficiency for RAND problems generally goes down with the problem size while for BFS problems, the efficiency depends mostly on the problem structure. Road networks on the west coast map, the Rocky mountain map, and the east coast map are less uniform than the other maps, and Dijkstra's algorithm

[5]This does not include vertices that were labeled but not scanned.

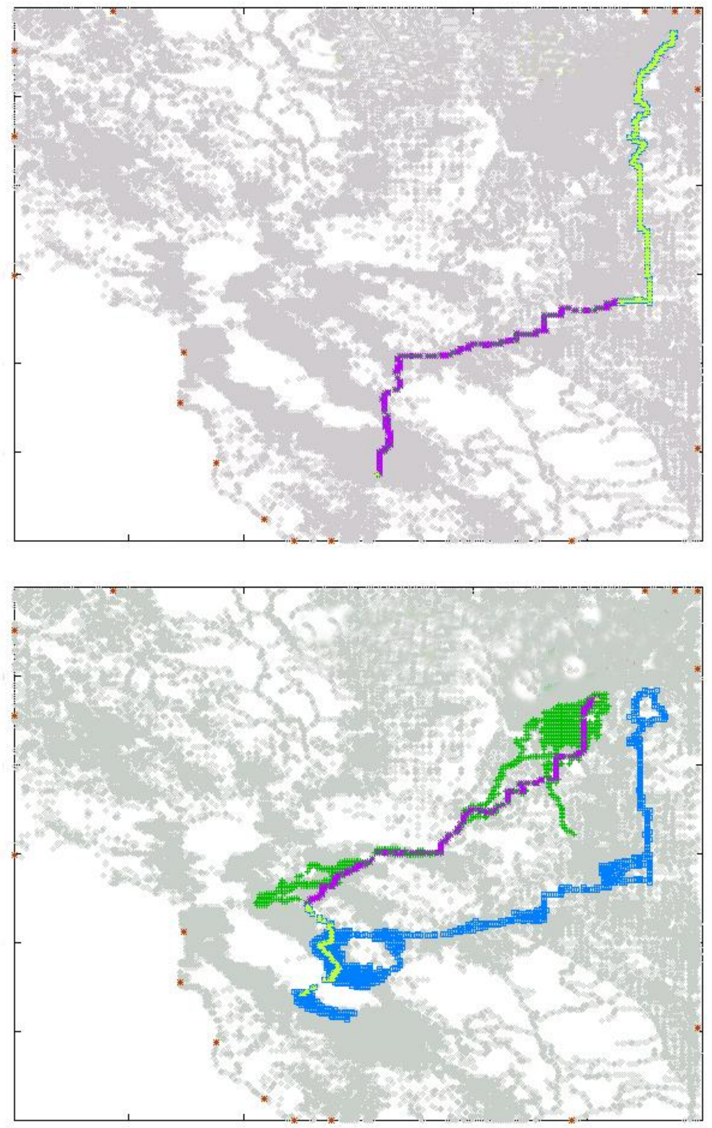

Figure 3: Two example runs of a bidirectional ALT algorithm. The input graph represents the road network in the San Francisco Bay Area. The first example (top) is an input for which our algorithm runs extremely fast; the second (bottom) is one which is a bit worse than the "average" query. Dark areas represent scanned vertices and light areas represent the rest of the underlying graph.

efficiency on BFS problems for these graphs is worse. With minor exceptions, this observation applies to the other algorithms as well.

Next we discuss performance. A^* search based on Euclidean lower bounds offers little efficiency improvement over the corresponding variant of Dijkstra's algorithm but hurts the running time, both in its regular and bidirectional forms. On the other hand, combining A^* search with our landmark-based lower bounds yields a major performance improvement.

BLA is the algorithm with the highest efficiency. Its efficiency is higher than that of B by roughly a factor of 30 on the RAND problems and about a factor of 6 on the BFS problems. The three fastest algorithms are AL, BLS, and BLA, with none dominating the other two. Of the codes that do not use landmarks, B is the fastest, although its efficiency is usually a little lower than that of BEA. As noted in the previous section, BEA and AE are slower for the same efficiency because the time to compute lower bounds is much greater.

Comparing AL with BLA, we note that on RAND problems, bidirectional search usually outperforms the regular one by more than a factor of two in efficiency, while for BFS problems, the improvement is usually less that a factor of 1.5.

9.3 Grids

The main goal of our grid graph experiments is to see how algorithm performance depends on the grid graph size and the arc length range. Tables 6 and 7 give data for the two input distributions and three arc length ranges. However, even for the $\{1, \ldots, 10\}$ and $\{1, \ldots, 100000\}$ ranges, algorithm efficiency is very similar, so we discuss the results together. On grid networks, geometric (Euclidean or Manhattan) bounds are very weak. We do not run AE and BEA on grid problems.

Qualitatively, the results are similar to those for road networks, although more well-behaved as the problem structure is very uniform. In particular, efficiency of D and B on RAND problems is inversely proportional to the grid length. Every time the grid length doubles (and the size quadruples), the efficiency halves. On BFS problems, efficiency of these algorithms shows much less dependence on problem size. This is to be expected because the area of a two-dimensional circular region increases as the square of the area of a thin region. Hence when dimension length is doubled, the former increases by a factor of four, while the latter by a factor of two, and the efficiency (the ratio of these two) is halved.

On grids, as on road networks, landmark based codes outperform AL and B by an order of magnitude. Qualitative results for the former are similar to those for the latter, although the numbers are not quite as well-behaved. We note that on these problems, BLS seems to have slightly lower overall efficiency than AL. As usual, BLA has the highest efficiency. On RAND problems, the efficiency starts in the mid-twenty percent range and drops to about 3% for the biggest problem. On BFS problems, the efficiency is roughly between 1/3 and 1/4, meaning that three to four vertices are scanned for every output vertex.

9.4 Random Graphs

For random graphs, B outperforms D by orders of magnitude, both in terms of efficiency and running time. This is to be expected, as a ball of twice the radius in an expander graph contains orders of magnitude more vertices. Tables 8 and 9 give data for these graphs.

Using landmark-based A^* search significantly improves regular search performance: AL is over an order of magnitude faster and more efficient than D. However, it is still worse by a large margin than B.

Performance of BLA is slightly below that of B. Interestingly, BLS performance is significantly below that of B and, overall, is slightly below that of AL. Recall that on the other classes, BLS was somewhat less efficient, but often a little faster than BLA. Our random graph experiments suggest that BLS is less robust than BLA.

For random graphs, our techniques do not improve the previous state of the art: B is the best algorithm among those we tested. This shows that ALT algorithms do not offer a performance improvement on all graph classes.

Name	D	AL	B	BLS	BLM	BLA
G_{11}	0.56	11.51	0.84	12.54	14.58	25.10
	11.75	*1.91*	*8.38*	*1.71*	*2.91*	*1.69*
G_{12}	0.58	12.49	0.89	12.86	13.98	26.22
	14.90	*1.90*	*10.72*	*1.80*	*3.13*	*1.74*
G_{13}	0.58	12.51	0.89	12.87	14.00	26.47
	18.39	*1.92*	*12.93*	*1.90*	*3.19*	*1.76*
G_{21}	0.28	7.31	0.42	7.26	8.00	14.32
	53.59	*7.59*	*41.79*	*7.67*	*12.31*	*6.96*
G_{22}	0.29	7.46	0.44	7.56	7.60	14.12
	65.63	*7.94*	*50.83*	*7.90*	*13.63*	*7.55*
G_{23}	0.29	7.47	0.44	7.56	7.60	14.11
	79.58	*8.09*	*58.88*	*8.05*	*14.14*	*7.76*
G_{31}	0.14	3.94	0.21	3.86	4.25	7.48
	313.81	*33.69*	*249.64*	*35.82*	*53.33*	*31.68*
G_{32}	0.14	4.31	0.22	4.23	4.45	8.10
	360.81	*29.09*	*281.73*	*32.96*	*49.30*	*28.40*
G_{33}	0.14	4.24	0.22	4.18	4.45	8.11
	429.57	*32.01*	*322.97*	*33.37*	*49.18*	*28.40*
G_{41}	0.07	1.98	0.11	1.74	1.84	2.86
	2080.46	*147.73*	*1509.83*	*187.26*	*274.11*	*193.83*
G_{42}	0.07	2.08	0.11	1.78	2.07	3.24
	2260.67	*143.30*	*1680.36*	*181.06*	*237.14*	*163.74*
G_{43}	0.07	2.08	0.11	1.78	2.07	3.25
	2536.80	*146.06*	*1878.31*	*181.73*	*236.13*	*166.86*

Table 6: Algorithm comparison for the RAND source-destination distribution on GRID networks and 16 landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	D	AL	B	BLS	BLM	BLA
G_{11}	1.27	26.61	2.53	27.11	24.92	33.27
	1.48	*0.27*	*0.79*	*0.27*	*0.48*	*0.39*
G_{12}	1.33	28.09	2.67	27.47	26.40	34.44
	1.78	*0.28*	*1.02*	*0.29*	*0.48*	*0.39*
G_{13}	1.33	28.15	2.67	27.47	26.45	34.49
	2.09	*0.28*	*1.10*	*0.28*	*0.50*	*0.41*
G_{21}	1.14	22.73	2.39	23.10	21.72	28.47
	1.74	*0.37*	*0.87*	*0.34*	*0.65*	*0.49*
G_{22}	1.18	23.54	2.49	22.95	23.29	28.45
	2.22	*0.39*	*1.10*	*0.38*	*0.64*	*0.53*
G_{23}	1.18	23.46	2.49	22.92	23.30	28.49
	2.58	*0.40*	*1.29*	*0.38*	*0.65*	*0.53*
G_{31}	1.14	24.20	2.37	24.00	23.56	30.24
	1.81	*0.44*	*0.93*	*0.43*	*0.78*	*0.63*
G_{32}	1.18	25.71	2.49	24.64	24.03	29.92
	2.23	*0.38*	*1.14*	*0.42*	*0.74*	*0.61*
G_{33}	1.18	25.72	2.49	24.62	23.97	30.04
	2.61	*0.43*	*1.33*	*0.43*	*0.73*	*0.57*
G_{41}	1.10	22.52	2.31	23.87	22.73	28.88
	1.94	*0.47*	*0.96*	*0.44*	*0.82*	*0.66*
G_{42}	1.14	22.66	2.43	22.52	22.35	27.69
	2.41	*0.51*	*1.22*	*0.46*	*0.92*	*0.68*
G_{43}	1.14	22.73	2.44	22.56	22.44	27.69
	2.77	*0.56*	*1.40*	*0.51*	*0.92*	*0.73*

Table 7: Algorithm comparison for the BFS source-destination distribution on GRID networks and 16 landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	D	AL	B	BLS	BLM	BLA
R_{11}	0.035	0.322	1.947	0.329	1.095	1.618
	41.850	*19.977*	*0.887*	*17.063*	*6.780*	*4.726*
R_{12}	0.040	0.385	1.926	0.318	1.165	1.759
	47.915	*19.119*	*0.824*	*19.709*	*7.454*	*5.167*
R_{13}	0.040	0.385	1.924	0.317	1.163	1.764
	53.206	*19.427*	*0.925*	*19.891*	*7.363*	*5.020*
R_{21}	0.009	0.075	1.054	0.083	0.551	0.840
	219.529	*119.427*	*1.684*	*95.273*	*19.112*	*12.702*
R_{22}	0.010	0.087	1.036	0.075	0.545	0.867
	274.250	*114.541*	*2.058*	*113.695*	*20.886*	*13.686*
R_{23}	0.010	0.083	1.035	0.076	0.535	0.886
	301.165	*123.411*	*2.257*	*116.649*	*22.293*	*14.106*
R_{31}	0.003	0.025	0.600	0.025	0.317	0.464
	912.840	*468.785*	*3.898*	*412.553*	*42.523*	*29.304*
R_{32}	0.003	0.029	0.577	0.024	0.287	0.484
	1155.378	*454.630*	*4.889*	*448.477*	*49.789*	*30.722*
R_{33}	0.003	0.029	0.577	0.024	0.285	0.485
	1298.041	*454.845*	*5.199*	*469.109*	*51.038*	*31.902*
R_{41}	0.001	0.006	0.343	0.008	0.158	0.261
	4192.350	*2259.739*	*8.556*	*1756.248*	*108.516*	*65.308*
R_{42}	0.001	0.008	0.340	0.008	0.154	0.268
	5007.551	*2155.511*	*10.307*	*2003.515*	*116.236*	*71.369*
R_{43}	0.001	0.008	0.340	0.008	0.153	0.268
	5884.373	*2065.892*	*10.913*	*1922.259*	*116.174*	*70.873*

Table 8: Algorithm comparison for the RAND source-destination distribution on RANDOM networks and 16 landmarks calculated with algorithm F2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	D	AL	B	BLS	BLM	BLA
R_{11}	0.024	0.128	2.022	0.182	0.951	1.636
	76.169	*59.104*	*0.880*	*38.030*	*10.024*	*5.954*
R_{12}	0.026	0.210	2.111	0.249	1.241	2.248
	87.903	*38.512*	*1.013*	*29.447*	*8.213*	*4.815*
R_{13}	0.026	0.211	2.111	0.250	1.239	2.255
	97.224	*39.476*	*1.089*	*29.955*	*8.393*	*4.844*
R_{21}	0.007	0.051	1.318	0.079	0.652	1.125
	368.850	*219.394*	*1.897*	*131.852*	*21.508*	*13.020*
R_{22}	0.007	0.070	1.391	0.078	0.632	1.207
	461.453	*176.563*	*2.071*	*143.200*	*23.836*	*13.379*
R_{23}	0.007	0.064	1.390	0.079	0.614	1.147
	504.651	*193.788*	*2.277*	*142.798*	*25.225*	*14.453*
R_{31}	0.002	0.019	0.761	0.029	0.395	0.711
	1672.077	*761.569*	*4.239*	*457.577*	*44.608*	*25.581*
R_{32}	0.002	0.025	0.799	0.027	0.385	0.721
	2133.890	*636.455*	*4.714*	*498.045*	*47.440*	*26.852*
R_{33}	0.002	0.025	0.799	0.027	0.385	0.720
	2402.068	*617.263*	*5.018*	*515.936*	*48.384*	*27.569*
R_{41}	0.001	0.003	0.379	0.005	0.155	0.298
	7696.567	*6529.415*	*10.260*	*3969.952*	*148.765*	*77.373*
R_{42}	0.001	0.006	0.421	0.006	0.202	0.432
	9670.789	*3619.273*	*10.908*	*2866.274*	*109.631*	*55.217*
R_{43}	0.001	0.006	0.422	0.006	0.201	0.432
	11393.517	*3414.361*	*11.548*	*2704.135*	*108.754*	*54.531*

Table 9: Algorithm comparison for the BFS source-destination distribution on RANDOM networks and 16 landmarks calculated with algorithm F2. Efficiency (%) is in Roman and time (ms) is in italics.

9.5 Number of Landmarks

In this section we study the relationship between algorithm efficiency and the number of landmarks. We ran experiments with 1, 2, 4, 8, and 16 landmarks for the AL and BLA algorithms. Tables 10-13 give results for road networks.

First, note that even with one landmark, AL and BLA outperform all non-landmark-based codes in our study. In particular, this includes BEA on road networks. As the number of landmarks increases, so does algorithm efficiency. The rate of improvement is substantial for RAND selection up to 16 landmarks and somewhat smaller for BFS. For the former, using 32 or more landmarks is likely to give significantly better results.

An interesting observation is that for a small number of landmarks, regular search often has higher efficiency than bidirectional search. This brings us to the following point. Consider the memory tradeoff question for regular vs. bidirectional search. Efficient implementation of the latter requires the list of reverse arcs in addition to the list of forward arcs. Given the same storage limit, one can have a few (2 to 4 for natural graph representations) more landmarks for AL than for BLA. Which code will perform better? The data suggests that if the number of landmarks is large, bidirectional search is more efficient in spite of the landmark deficit. If the number of landmarks is small, regular search with extra landmarks is more efficient.

9.6 Landmark Selection

Tables 14-19 give data comparing our landmark heuristics. We give data only for the choice of 16 landmarks; similar results hold for fewer. Note also that this data does not give information about the tradeoff between precomputation time and efficiency. Keep in mind when reading the data that the algorithms R and F are both quite fast, P is a bit slower, and R2 and P2 are slow, taking hours to compute for the largest graphs.

For all graph types, random (R) landmark selection, while typically the worst efficiency-wise, still does reasonably well. Farthest (F) is a modest improvement on random, while retaining the feature that it works on an arbitrary graph with no geometric information, and optimized planar (P2) is the best on nearly every example.

Next observe that, for road and GRID networks, the difference between algorithms is much less pronounced for the BFS input distribution than for RAND. One intuition for this is that, since a typical input s, t under BFS has endpoints very close to one another, we need only make sure we have landmarks far away, with one "behind" t from the perspective of s, and one "behind" s from the perspective of t. Under the RAND distribution, this is much more difficult to have happen because the endpoints are so often very far apart.

10 Concluding Remarks

We proposed a new lower-bounding technique based on landmarks and triangle inequality, as well as several landmark selection techniques. Our best landmark selection strategies consistently outperform naïve random landmark selection. However, luck still plays a role in landmark selection, and there may be room for improvement in this area.

We would like to comment on the use of ALT algorithms in dynamic settings where arc lengths change (note that additions and deletions can be modeled using infinite arc lengths). First consider a semi-dynamic case when arc lengths can only increase, like on road networks due to traffic congestion and road closures. In this case, our lower bounds remain valid and the algorithms work correctly. One would hope that if the changes are not dramatic, the performance remains good. In the fully dynamic case or with drastic changes, one can keep landmark placement but periodically recompute distances to and from landmarks. Single-source shortest path computation is fairly efficient, and for a reasonable number of landmarks the time to update the distances may be acceptable in practice. For example, for road networks the update can be done on the order of a minute. Reoptimization techniques (see e.g. [24]) may further reduce the update

Name	AL-1	AL-2	AL-4	AL-8	AL-16
M_1	1.16	1.63	3.82	5.63	6.49
	35.71	*25.88*	*12.08*	*8.07*	*7.23*
M_2	0.73	1.01	2.17	2.85	3.00
	50.71	*36.68*	*17.12*	*12.96*	*12.84*
M_3	0.51	0.66	1.48	1.85	2.48
	93.05	*74.62*	*36.87*	*27.28*	*19.92*
M_4	0.58	0.75	2.01	2.92	4.14
	103.49	*79.15*	*31.54*	*21.82*	*14.24*
M_5	0.58	0.82	1.81	2.72	3.34
	104.64	*73.28*	*36.81*	*23.70*	*18.43*
M_6	0.75	1.08	1.47	2.07	2.25
	172.94	*106.32*	*93.83*	*67.26*	*63.47*
M_7	0.36	0.53	1.32	1.82	2.72
	446.21	*270.18*	*124.21*	*82.60*	*50.39*
M_8	0.44	0.67	1.42	2.09	2.76
	353.61	*255.91*	*132.30*	*80.53*	*62.15*
M_9	0.27	0.37	0.88	1.45	2.07
	697.65	*521.85*	*251.52*	*132.98*	*86.99*
M_{10}	0.26	0.36	0.76	1.29	1.67
	813.70	*545.79*	*319.62*	*184.66*	*138.58*
M_{11}	0.21	0.23	0.65	1.12	1.74
	1261.50	*1194.78*	*436.46*	*257.65*	*152.90*

Table 10: Landmark quantity comparision for the RAND source-destination distribution on road networks with TRANSIT TIME distances and landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	AL-1	AL-2	AL-4	AL-8	AL-16
M_1	1.05	1.47	3.14	4.49	5.34
	36.46	*26.85*	*13.49*	*9.30*	*8.01*
M_2	0.73	1.02	2.03	2.55	3.02
	43.22	*33.37*	*17.88*	*14.19*	*13.05*
M_3	0.51	0.56	1.43	2.24	2.90
	80.00	*76.63*	*34.10*	*20.03*	*15.50*
M_4	0.55	0.76	2.07	3.06	3.82
	98.63	*71.00*	*28.43*	*18.09*	*14.20*
M_5	0.55	0.71	1.96	3.13	4.21
	100.65	*81.30*	*32.15*	*18.33*	*13.04*
M_6	0.77	1.06	1.53	2.11	2.39
	162.52	*110.90*	*98.47*	*73.11*	*62.33*
M_7	0.36	0.51	1.34	1.95	3.13
	427.43	*280.55*	*115.33*	*72.54*	*40.67*
M_8	0.42	0.61	1.31	2.00	2.69
	335.42	*247.90*	*132.69*	*83.93*	*59.78*
M_9	0.25	0.34	0.89	1.27	1.87
	681.07	*533.54*	*228.41*	*142.88*	*92.88*
M_{10}	0.27	0.38	0.73	1.18	1.56
	766.34	*514.13*	*342.43*	*198.76*	*147.54*
M_{11}	0.20	0.24	0.67	1.17	1.81
	1251.07	*1088.48*	*423.39*	*222.17*	*132.83*

Table 11: Landmark quantity comparision for the RAND source-destination distribution on road networks with DISTANCE distances and landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	AL-1	AL-2	AL-4	AL-8	AL-16
M_1	4.70	9.74	14.05	14.79	15.28
	1.02	*0.46*	*0.35*	*0.35*	*0.35*
M_2	2.53	4.03	7.41	8.26	8.03
	2.10	*1.31*	*0.79*	*0.75*	*0.79*
M_3	2.45	4.24	6.78	7.31	7.27
	2.05	*1.19*	*0.83*	*0.81*	*0.84*
M_4	5.07	7.51	14.65	17.16	18.89
	0.95	*0.65*	*0.37*	*0.33*	*0.32*
M_5	4.60	7.43	14.64	17.08	16.99
	1.00	*0.63*	*0.35*	*0.32*	*0.33*
M_6	4.01	6.40	7.91	9.05	12.35
	1.15	*0.76*	*0.68*	*0.62*	*0.44*
M_7	5.50	7.82	14.89	15.25	18.83
	0.90	*0.63*	*0.35*	*0.35*	*0.29*
M_8	3.90	6.40	9.35	9.41	10.01
	1.22	*0.75*	*0.56*	*0.59*	*0.57*
M_9	4.33	7.39	15.28	17.69	16.81
	1.05	*0.62*	*0.35*	*0.31*	*0.35*
M_{10}	3.62	5.36	8.38	9.77	9.78
	1.39	*0.95*	*0.67*	*0.60*	*0.63*
M_{11}	4.51	7.59	11.78	12.92	14.37
	1.06	*0.60*	*0.42*	*0.40*	*0.40*

Table 12: Landmark quantity comparision for the BFS source-destination distribution on road networks with Transit Time distances and landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	AL-1	AL-2	AL-4	AL-8	AL-16
M_1	4.84	9.50	14.06	14.75	16.20
	0.86	*0.44*	*0.33*	*0.33*	*0.32*
M_2	2.60	3.89	7.18	8.94	9.48
	1.78	*1.23*	*0.73*	*0.64*	*0.64*
M_3	2.44	3.74	6.87	8.47	8.36
	1.86	*1.28*	*0.76*	*0.64*	*0.67*
M_4	5.01	7.82	16.73	21.46	22.43
	0.88	*0.56*	*0.31*	*0.25*	*0.25*
M_5	4.95	7.37	15.52	18.36	17.96
	0.95	*0.61*	*0.31*	*0.28*	*0.30*
M_6	4.09	6.92	9.60	11.55	12.43
	1.03	*0.63*	*0.48*	*0.42*	*0.44*
M_7	4.90	7.70	14.80	17.46	17.63
	0.97	*0.60*	*0.35*	*0.28*	*0.29*
M_8	3.76	6.88	8.50	10.04	10.68
	1.17	*0.65*	*0.58*	*0.52*	*0.50*
M_9	4.66	8.25	15.93	19.01	19.55
	0.92	*0.51*	*0.31*	*0.27*	*0.29*
M_{10}	3.68	5.64	9.68	12.14	12.79
	1.23	*0.84*	*0.56*	*0.51*	*0.46*
M_{11}	5.60	8.71	14.19	16.72	18.45
	0.79	*0.49*	*0.34*	*0.30*	*0.33*

Table 13: Landmark quantity comparision for the BFS source-destination distribution on road networks with Distance distances and landmarks calculated with algorithm P2. Efficiency (%) is in Roman and time (ms) is in italics.

Name	AL-R	AL-R2	AL-F	AL-P	AL-P2
M_1	4.47	6.47	5.28	5.46	6.49
M_2	1.88	2.94	2.74	2.36	3.00
M_3	1.80	2.28	2.06	2.04	2.48
M_4	2.30	3.27	3.66	3.25	4.14
M_5	2.02	2.97	3.16	2.74	3.34
M_6	1.49	2.12	2.07	1.93	2.25
M_7	1.19	1.58	2.07	2.26	2.72
M_8	1.52	2.35	2.05	2.13	2.76
M_9	1.15	1.54	1.51	1.73	2.07
M_{10}	0.92	1.10	1.47	1.17	1.67
M_{11}	0.65	1.16	1.28	1.52	1.74

Table 14: Landmark type comparison for the RAND source-destination distribution on road networks with TRANSIT TIME distances and 16 landmarks. Efficiency (%) is in Roman.

Name	AL-R	AL-R2	AL-F	AL-P	AL-P2
G_{11}	6.28	7.75	11.06	11.88	11.51
G_{12}	6.66	8.04	11.11	12.10	12.49
G_{13}	6.67	8.04	11.13	12.12	12.51
G_{21}	3.06	5.21	5.55	5.62	7.31
G_{22}	3.19	5.60	5.83	6.14	7.46
G_{23}	3.20	5.60	5.83	6.15	7.47
G_{31}	1.38	2.60	3.21	3.20	3.94
G_{32}	1.40	2.88	3.13	3.12	4.31
G_{33}	1.40	2.86	3.17	3.17	4.24
G_{41}	0.83	1.59	1.85	1.78	1.98
G_{42}	0.86	1.64	1.77	1.86	2.08
G_{43}	0.87	1.64	1.78	1.86	2.08

Table 15: Landmark type comparison for the RAND source-destination distribution on GRID networks and 16 landmarks. Efficiency (%) is in Roman.

Name	AL-R	AL-R2	AL-F
R_{11}	0.31	0.31	0.33
R_{12}	0.37	0.38	0.36
R_{13}	0.37	0.38	0.36
R_{21}	0.08	0.08	0.07
R_{22}	0.09	0.09	0.09
R_{23}	0.09	0.09	0.08
R_{31}	0.02	0.03	0.02
R_{32}	0.03	0.03	0.03
R_{33}	0.03	0.03	0.03
R_{41}	0.01	0.01	0.01
R_{42}	0.01	0.01	0.01
R_{43}	0.01	0.01	0.01

Table 16: Landmark type comparison for the RAND source-destination distribution on RANDOM networks and 16 landmarks. Efficiency (%) is in Roman.

Name	AL-R	AL-R2	AL-F	AL-P	AL-P2
M_1	12.05	14.08	12.82	14.95	15.28
M_2	6.86	8.01	8.38	7.78	8.03
M_3	6.54	6.87	6.98	6.65	7.27
M_4	15.83	16.20	17.97	16.50	18.89
M_5	12.67	15.66	16.21	15.45	16.99
M_6	10.47	12.47	11.90	10.29	12.35
M_7	13.51	13.96	17.66	16.18	18.83
M_8	8.59	9.67	9.42	9.47	10.01
M_9	13.39	15.47	15.83	15.39	16.81
M_{10}	7.99	8.81	10.06	9.30	9.78
M_{11}	10.28	11.51	13.31	12.47	14.37

Table 17: Landmark type comparison for the BFS source-destination distribution on road networks with TRANSIT TIME distances and 16 landmarks. Efficiency (%) is in Roman.

Name	AL-R	AL-R2	AL-F	AL-P	AL-P2
G_{11}	19.92	22.76	25.77	24.76	26.61
G_{12}	20.95	23.43	26.82	25.35	28.09
G_{13}	20.98	23.48	27.05	25.43	28.15
G_{21}	17.49	20.05	21.26	22.08	22.73
G_{22}	17.39	19.90	21.31	21.04	23.54
G_{23}	17.38	19.85	21.32	21.04	23.46
G_{31}	16.11	20.12	22.24	22.15	24.20
G_{32}	16.70	20.05	22.96	23.43	25.71
G_{33}	16.72	20.06	22.98	23.46	25.72
G_{41}	15.65	18.89	19.87	20.58	22.52
G_{42}	15.60	19.02	20.84	20.61	22.66
G_{43}	15.59	19.05	20.91	20.64	22.73

Table 18: Landmark type comparison for the BFS source-destination distribution on GRID networks and 16 landmarks. Efficiency (%) is in Roman.

Name	AL-R	AL-R2	AL-F
R_{11}	0.124	0.127	0.160
R_{12}	0.154	0.155	0.212
R_{13}	0.155	0.156	0.212
R_{21}	0.039	0.039	0.051
R_{22}	0.052	0.051	0.067
R_{23}	0.052	0.051	0.068
R_{31}	0.012	0.012	0.017
R_{32}	0.016	0.017	0.021
R_{33}	0.016	0.017	0.021
R_{41}	0.003	0.003	0.004
R_{42}	0.004	0.003	0.005
R_{43}	0.004	0.003	0.005

Table 19: Landmark type comparison for the BFS source-destination distribution on RANDOM networks and 16 landmarks. Efficiency (%) is in Roman.

time. Finally, one can recompute landmarks themselves using an efficient landmark selection strategy (observe that farthest landmark selection takes less time than updating landmark distances).

When our experiments were near completion, we learned about the work of Gutman [15], who studies the P2P problem in a similar setting to ours. Gutman's algorithms are based on the concept of *reach* and need to store a single "reach value" and Euclidean coordinates of every vertex. Based on indirect comparison, performance of his fastest algorithm is better than that of ours with one landmark and worse than that of ours with sixteen landmarks. Gutman's approach requires more assumptions about the input domain than ours, his preprocessing is more time-consuming, and his approach does not seem to adapt to dynamic settings as well as ours. However, his results are very interesting. If space is very limited, his algorithm is faster than ours. Furthermore, Gutman observes that his ideas can be combined with A^* search. It would be interesting to see if using Gutman's reach-based pruning in ALT algorithms will noticeably improve their efficiency. Another possibility is to use our lower-bounding technique in place of Euclidean lower bounds in his algorithm. This would have the advantage of making reach-based routing work on an arbitrary graph, and may improve performance for map graphs as well.

Unless the number of landmarks is very small, landmark distances dominate the space required by our algorithm. We would like to note that one can compress these distances by using locality, domain knowledge, and by trading space for time. For example for road networks, compression by a factor of two to four is easily achievable. Such compression techniques are an interesting direction for further research.

Finally, many applications of A^* search (e.g., solving the 15 puzzle) work with implicit representations of huge search spaces. It would be interesting to see if our techniques can be used in such a context.

Acknowledgments

We are very grateful to Boris Cherkassky for many stimulating discussions and for his help with the design and implementation of landmark selection algorithms. We would also like to thank Jeff Couckuyt for help with the MapPoint data, Gary Miller and Guy Blelloch for pointing us to some of the literature, and Bob Tarjan, Satish Rao, Kris Hildrum, and Frank McSherry for useful discussions.

References

[1] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525, 1994.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[3] 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.

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

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

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

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

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

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

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

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

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

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

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

[15] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proc. Algorithm engineering and experimentation: sixth annual international workshop*, 2004.

[16] 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.

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

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

[19] P. Klein. Preprocessing an Undirected Planar Network to Enable Fast Approximate Distance Queries. In *SODA*, pages 820–827, 2002.

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

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

[22] 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.

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

[24] S. Pallottino and M. G. Scutell. A New Algorithm for Reoptimizing Shortest Paths when the Arc Costs Change. *Networks*, 31:149–160, 2003.

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

[26] F. Schulz, D. Wagner, and K. Weihe. Using Multi-Level Graphs for Timetable Information. In *Proc. Algorithm Engineering and Exteriments*, pages 43–59. LNCS, Springer, 2002.

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

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

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

[30] 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.

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

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

[33] 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.