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

Andrew V. Goldberg* Chris Harrelson†

Abstract

We propose shortest path algorithms that use A^* search in combination with a new graph-theoretic lower-bounding technique based on landmarks and the triangle inequality. 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 and on some synthetic problem families.

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 weighted, 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. 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 [21, 27, 29, 36]. 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 [10]) is superlinear in the output path size unless the path is very long. Preprocessing using geometric information and hierarchical decomposition is discussed in [19, 28, 34]. Other related work includes algorithms for the single-source shortest path problem, such as [1, 4, 6, 7, 13, 14, 16, 17, 18, 22, 25, 32, 35], and algorithms for approximate shortest paths that use preprocessing [3, 23, 33].

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* (also known as *heuristic search*) [8, 20] 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 [27] 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 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

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

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

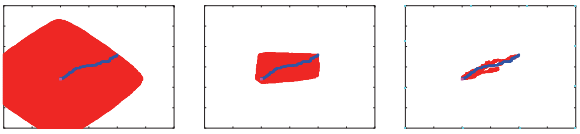

Figure 1: Vertices visited by Dijkstra's algorithm (left), A^* search with Manhattan lower bounds (middle), and an ALT algorithm (right) on the same input.

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 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.

Our algorithm is the first exact shortest path algorithm with preprocessing that can be applied to arbitrary directed graphs. The exact algorithms mentioned above [10, 19, 28, 34] apply to restricted graph classes. Landmark-based bounds have been previously used for approximate shortest path computation (see e.g. [3]), but these bounds are different from ours. In particular, these bounds are not feasible and cannot be used in an exact A^* search algorithm.

We also study bidirectional variants of A^* search. For these, one has to be careful to preserve optimality (for example, see [5, 30]). Pohl [27] (see also [24]) and Ikeda et al. [21] give two ways of combining A^* search with the bidirectional version of Dijkstra's method [4, 9, 26] to get provably optimal algorithms. We describe and investigate an alternative to the Ikeda et al. algorithm, and show that in practice our best implementation of the bidirectional ALT algorithm is more robust than the regular ALT implementation.

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.

Our third contribution is an experimental study comparing the new and 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. We also run experiments showing why ALT algorithms are efficient. In particular, we show that when our algorithms work well, the lower bounds are within a few percent of the true distances for most vertices. Some of the experimental methodology we use is new and may prove helpful in future work in this 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.

Due to the space restriction, we omit some results and details. See the full paper [15] for details.

2 Preliminaries

The input to the preprocessing stage of the P2P problem is a directed graph with n vertices, m arcs, and nonnegative lengths $\ell(a)$ for each arc a. After preprocessing, we get queries specified by a source vertex s and a sink vertex t. The answer to a query is the shortest path from s to t.

Let $\text{dist}(v, w)$ denote the shortest-path distance from vertex v to vertex w with respect to ℓ. We will often use edge lengths other than ℓ, but $\text{dist}(\cdot, \cdot)$ will always refer to the original arc lengths. Note that in general $\text{dist}(v, w) \neq \text{dist}(w, v)$.

A *potential function* is a function from vertices to reals. Given a potential function π, we define the

reduced cost of an edge by $\ell_\pi(v,w) = \ell(v,w) - \pi(v) + \pi(w)$. 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.

We say that π is *feasible* if ℓ_π is nonnegative for all arcs. It is well-known that if $\pi(t) \leq 0$ and π is feasible, then for any v, $\pi(v)$ is a lower bound on the distance from v to t. A related and simple-to-prove fact is:

LEMMA 2.1. *If π_1 and π_2 are feasible potential functions, then $p = \max(\pi_1, \pi_2)$ is feasible.*

One can also combine feasible potential functions by taking the minimum, or, as observed in [21], the average of feasible potential functions. We use the maximum in particular to combine several feasible lower bound functions in order to get one that at any vertex is at least as high as each original function.

3 Labeling Method and Dijkstra's Algorithm

The labeling method for the shortest path problem [11, 12] finds shortest paths from the source to all vertices in the graph. The method works as follows (see for example [31]). 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 [7] (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. *[7] 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*. 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, 9, 26] 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. We use an alternation strategy that balances the work of the forward and reverse searches.

THEOREM 3.2. *[27] If the sink is reachable from the source, the bidirectional algorithm finds an optimal path, and it is the path stored along with μ.*

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.

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

Observe that better lower bounds give better performance, in the following sense. Consider an instance of the P2P problem and let π_t and $\pi_t{}'$ be two feasible potential functions such that $\pi_t(t) = \pi_t{}'(t) = 0$ and for any vertex v, $\pi_t{}'(v) \geq \pi_t(v)$ (i.e., $\pi_t{}'$ dominates π_t). As-

sume that when selecting the next vertex for scan, we break ties based on vertex IDs, and when we say $v > w$ we mean that v's ID is greater than w's.

THEOREM 4.1. *The set of vertices scanned by A^* search search using π_t' is a subset of the set of vertices scanned by A^* search search using π_t.*

This theorem implies that any lower-bounding algorithm with a nonnegative potential function visits no more vertices than Dijkstra's algorithm.

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 [27]. 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 $k(v) \geq \mu$ (see the first paragraph of Section 4 for a definition of $k(\cdot)$) or when both searches have no labeled vertices. The algorithm is correct because the search must have found the shortest path by then.

Kwa [24] suggests several potential improvements to Pohl's algorithm, one of which we use. The improvement is as follows. 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*.

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. [21] 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*.

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 2.1.

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*.

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

We use the following optimization. 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. Although Theorem 4.1 suggests that using a subset of landmarks may lead to more vertex scans, for a moderate number of landmarks (e.g. 16) and selecting a small subset (e.g. 4), this increase is small relative to the improved efficiency of the lower bound computations. Note that a natural analog of the theorem holds for the symmetric bidirectional algorithm, but there does not seem to be an obvious analog for the consistent 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. One can do better, however.

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. The graph does not need to be planar; for example, road networks are non-planar. A simple landmark selection algorithm in this case works as follows. 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.

Name	# of vert.	# of arcs	Lat./long. range
M_1	267,403	631,964	[34,37]/[-107,-103]
M_2	330,024	793,681	[37,39]/[-123,-121]
M_3	563,992	1,392,202	[33,35]/[-120,-115]
M_4	588,940	1,370,273	[37,40]/[-92,-88]
M_5	639,821	1,522,485	[31,34]/[-98,-94]
M_6	1,235,735	2,856,831	[33,45]/[-130,-120]
M_7	2,219,925	5,244,506	[33,45]/[-110,-100]
M_8	2,263,758	5,300,035	[33,45]/[-120,-110]
M_9	4,130,777	9,802,953	[33,45]/[-100,-90]
M_{10}	4,469,462	10,549,756	[33,45]/[-80,-70]
M_{11}	6,687,940	15,561,631	[33,45]/[-90,-80]

Table 1: Road network problems, sorted by size.

Another optimization technique for a given set of landmarks is to remove a landmark and replace it by the best landmark in a set of *candidate landmarks*. To select the best candidate, we compute a score for each one 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. To obtain the sample of vertex pairs, for each vertex we choose 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, the 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.

The optimized planar selection, although somewhat computationally expensive, is superior to regular planar selection, and in fact is our best landmark selection rule for graphs with a given planar layout.

8 Experimental Setup

8.1 Problem Families We ran experiments on road graphs and on several classes of synthetic problems. The road graphs are subgraphs of the graph used in MapPoint that includes all of the roads 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 GRID and RANDOM. We omit the former due to the space limit. A RANDOM graph with n vertices and m arcs is 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 $65536 \cdot 4^{i-1}$ vertices, $4 \cdot 65536 \cdot 4^{i-1}$ arcs, and edge weights chosen uniformly at random from $\{1, \ldots, 10 \cdot 100^{j-1}\}$.

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., [35]). 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 c arcs away from s, and chose one of these vertices uniformly at random. On road and grid graphs, we use $c = 50$. 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 100 road segments. In this sense it is a more "typical" distribution. On random graphs we use $c = 6$ 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 [27, 29].

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 [21].

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 We leave to the full paper a comparison of different landmark selection algorithms.

When comparing algorithms, we set the number of landmarks to 16 with the optimized planar (P2) landmark selection algorithm when it is applicable, and the farthest (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, and 16 because is the maximum number that fits in memory for our biggest test problem.

8.3 Implementation Choices 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 [21]. Such scaling may result in non-optimal 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.

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

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 2: RAND *s-t* distribution on road networks with DISTANCE lengths. Efficiency (%); *time (ms)*.

data structure for all algorithms. As the result, performance may suffer somewhat. However, the loss is probably less than a factor of two, and 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.[1] 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. Despite their shortcomings, running times are important for sanity-checking and complement efficiency to provide a better understanding of practical performance of algorithms under consideration. However, efficiency is closely correlated with running time. To save space, we report running times only in the first data table (Table 2)). From this table one can get a good idea of the overhead for different 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. All reported data points are the average of 128 trials.

For most algorithms, we used priority queues based on multi-level buckets [6, 17, 18]. 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 Road Networks Tables 2 and 3 give data for road networks with distance arc lengths. The results for transit time lengths are similar (see the full paper). 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. 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 improve-

[1] This does not include vertices that were labeled but not scanned, which is an alternative measure.

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
M_2	0.82	1.10	9.48	1.58	1.78	9.86	9.84	12.55
M_3	0.69	0.78	8.36	1.35	1.57	8.48	8.18	10.84
M_4	1.58	1.72	22.43	3.19	3.32	22.50	20.52	26.40
M_5	1.46	1.89	17.96	3.02	3.28	22.83	22.67	26.47
M_6	1.40	1.99	12.43	2.80	3.21	13.53	11.77	16.89
M_7	1.63	2.42	17.63	3.57	4.19	17.72	16.62	19.82
M_8	1.27	1.38	10.68	2.53	2.79	11.57	9.29	14.46
M_9	1.37	1.83	19.55	3.05	3.43	22.42	19.98	24.93
M_{10}	1.03	1.34	12.79	2.37	2.59	13.96	11.48	17.74
M_{11}	1.59	1.93	18.45	3.58	3.83	19.76	17.46	22.38

Table 3: Eff. for BFS source-destination distribution on road networks with DISTANCE lengths.

ment 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.

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.

9.2 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 4 and 5 give data for these graphs. We report efficiency only.

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 only slightly below that of B. Performance of BLM is worse, but within a factor of two of that of BLA. Performance of BLS is significantly worse, suggesting that the symmetric algorithm is less robust.

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.

9.3 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 6-7 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.

9.4 Lower-Bound Quality In this section, we study the quality of our lower bounds. For this test, we picked two problems: the Bay Area road network (M_2) with the distance-based length function, and a large random problem R_{41}. For each of the problems, we generated 16 landmarks, using the P2 heuristic for the first two and the F heuristic for the last one.

For each of the problems, we ran two experiments, one with RAND vertex pair selection and the other with BFS selection. For each pair of selected vertices, we computed the ratio of the our lower bound on the distance between them and the true distance. Each experiment was repeated 128 times.

Table 8 gives a summary of the results. Note that there is a clear correlation between lower bound quality and algorithm performance. First consider the road network and the grid problems where ALT algorithms work well. For RAND selection, the bounds are very good, within a few percent of the true distances, leading to good performance. For BFS selection, the bounds are somewhat worse but still good. On the other hand, for

Name	D	AL	B	BLS	BLM	BLA
R_{11}	0.035	0.322	1.947	0.329	1.095	1.618
R_{12}	0.040	0.385	1.926	0.318	1.165	1.759
R_{13}	0.040	0.385	1.924	0.317	1.163	1.764
R_{21}	0.009	0.075	1.054	0.083	0.551	0.840
R_{22}	0.010	0.087	1.036	0.075	0.545	0.867
R_{23}	0.010	0.083	1.035	0.076	0.535	0.886
R_{31}	0.003	0.025	0.600	0.025	0.317	0.464
R_{32}	0.003	0.029	0.577	0.024	0.287	0.484
R_{33}	0.003	0.029	0.577	0.024	0.285	0.485
R_{41}	0.001	0.006	0.343	0.008	0.158	0.261
R_{42}	0.001	0.008	0.340	0.008	0.154	0.268
R_{43}	0.001	0.008	0.340	0.008	0.153	0.268

Table 4: Algorithm comparison for the RAND source-destination distribution on RANDOM networks.

Name	D	AL	B	BLS	BLM	BLA
R_{11}	0.024	0.128	2.022	0.182	0.951	1.636
R_{12}	0.026	0.210	2.111	0.249	1.241	2.248
R_{13}	0.026	0.211	2.111	0.250	1.239	2.255
R_{21}	0.007	0.051	1.318	0.079	0.652	1.125
R_{22}	0.007	0.070	1.391	0.078	0.632	1.207
R_{23}	0.007	0.064	1.390	0.079	0.614	1.147
R_{31}	0.002	0.019	0.761	0.029	0.395	0.711
R_{32}	0.002	0.025	0.799	0.027	0.385	0.721
R_{33}	0.002	0.025	0.799	0.027	0.385	0.720
R_{41}	0.001	0.003	0.379	0.005	0.155	0.298
R_{42}	0.001	0.006	0.421	0.006	0.202	0.432
R_{43}	0.001	0.006	0.422	0.006	0.201	0.432

Table 5: Algorithm comparison for the BFS source-destination distribution on RANDOM networks.

the random graph, the lower bounds for the RAND and BFS selection are, respectively, almost a factor of four and two below the true distances. This explains why the bounds do not help much.

10 Concluding Remarks

When our experiments were near completion, we learned about the work of Gutman [19], 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. In particular, 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.

Name	AL-1	AL-2	AL-4	AL-8	AL-16
M_1	1.05	1.47	3.14	4.49	5.34
M_2	0.73	1.02	2.03	2.55	3.02
M_3	0.51	0.56	1.43	2.24	2.90
M_4	0.55	0.76	2.07	3.06	3.82
M_5	0.55	0.71	1.96	3.13	4.21
M_6	0.77	1.06	1.53	2.11	2.39
M_7	0.36	0.51	1.34	1.95	3.13
M_8	0.42	0.61	1.31	2.00	2.69
M_9	0.25	0.34	0.89	1.27	1.87
M_{10}	0.27	0.38	0.73	1.18	1.56
M_{11}	0.20	0.24	0.67	1.17	1.81

Table 6: Landmark quantity comparision for the RAND source-destination distribution on road networks with DISTANCE lengths.

Name	AL-1	AL-2	AL-4	AL-8	AL-16
M_1	4.84	9.50	14.06	14.75	16.20
M_2	2.60	3.89	7.18	8.94	9.48
M_3	2.44	3.74	6.87	8.47	8.36
M_4	5.01	7.82	16.73	21.46	22.43
M_5	4.95	7.37	15.52	18.36	17.96
M_6	4.09	6.92	9.60	11.55	12.43
M_7	4.90	7.70	14.80	17.46	17.63
M_8	3.76	6.88	8.50	10.04	10.68
M_9	4.66	8.25	15.93	19.01	19.55
M_{10}	3.68	5.64	9.68	12.14	12.79
M_{11}	5.60	8.71	14.19	16.72	18.45

Table 7: Landmark quantity comparision for the BFS source-destination distribution on road networks with DISTANCE lengths.

Acknowledgments We are very grateful to Boris Cherkassky for many discussions and for his help with the design and implementation of landmark selection algorithms. We would like to thank Jeff Couckuyt for help with the MapPoint data, Gary Miller, Guy Blelloch, and Stefan Lewandowski 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.

Name	RAND	BFS
M_2	96.0 (4.5)	89.4 (9.97)
R_{41}	28.5 (41.2)	60.0 (46.5)

Table 8: Ratio of the lower bound and the true distance in percent (standard deviation).

[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] D. de Champeaux. Bidirectional Heuristic Search Again. *J. ACM*, 30(1):22–32, 1983.

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

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

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

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

[10] 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.

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

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

[13] 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.

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

[15] A. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2004.

[16] A. V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. In *Proc. 9th Annual European Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 230–241. Springer-Verlag, 2001.

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

[18] 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.

[19] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. 6th International Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[20] 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.

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

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

[23] P. Klein. Preprocessing an Undirected Planar Network to Enable Fast Approximate Distance Queries. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 820–827, 2002.

[24] J. Kwa. BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. *Artif. Intell.*, 38(1):95–109, 1989.

[25] U. Meyer. Single-Source Shortest Paths on Arbitrary Directed Graphs in Linear Average Time. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 797–806, 2001.

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

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

[28] F. Schulz, D. Wagner, and K. Weihe. Using Multi-Level Graphs for Timetable Information. In *Proc. 4th International Workshop on Algorithm Engineering and Experiments*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer-Verlag, 2002.

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

[30] L. Sint and D. de Champeaux. An Improved Bidirectional Heuristic Search Algorithm. *J. ACM*, 24(2):177–191, 1977.

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

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

[33] 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.

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

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

[36] 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.