Computing Point-to-Point Shortest Paths from External Memory

Andrew V. Goldberg[*] Renato F. Werneck[†]

Abstract

We study the ALT algorithm [19] for the point-to-point shortest path problem in the context of road networks. We suggest improvements to the algorithm itself and to its preprocessing stage. We also develop a memory-efficient implementation of the algorithm that runs on a Pocket PC. It stores graph data in a flash memory card and uses RAM to store information only for the part of the graph visited by the current shortest path computation. The implementation works even on very large graphs, including that of the North America road network, with almost 30 million vertices.

1 Introduction

Finding shortest paths is a fundamental problem with numerous applications. There are several variations, including single-source, point-to-point, and all-pairs shortest paths. The single-source problem with nonnegative arc lengths has been studied most extensively [4, 6, 7, 8, 15, 16, 17, 18, 20, 25, 30, 38, 41]. For this problem, near-optimal algorithms are known both in theory, with near-linear time bounds, and in practice, where running times are within a small constant factor of the breadth-first search time.

In this paper we study another common variant, the point-to-point shortest path problem on directed graphs with nonnegative arc lengths (the *P2P problem*). We are interested in exact shortest paths only. Unlike the single-source case, where every vertex of the graph must be visited, the P2P problem can often be solved while visiting a small subgraph. Therefore, we measure the algorithm performance in an output-sensitive way, in terms of its *efficiency*: the ratio between the number of vertices scanned by the algorithm and the number of vertices on the shortest path. We allow preprocessing, but limit the size of the data computed during this phase to a (moderate) constant times the input graph size.

The P2P problem with no preprocessing has been

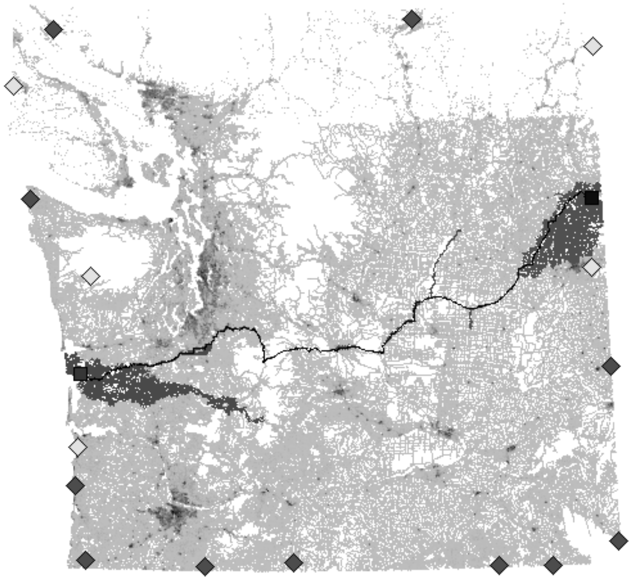
Figure 1: An example of ALT algorithm running on the road network of the Washington state and surrounding area. Grayscale levels indicate road network density. Highlighted area represents vertices actually visited. Landmarks are represented by diamonds, and the five clear ones are active. This is a relatively hard instance.

addressed, for example, in [24, 34, 36, 42]. 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 [11]) is superlinear in the size of the output path unless the path is very long. Algorithms for approximate shortest paths that use preprocessing have been studied; see e.g. [5, 26, 39]. Previous work on exact algorithms with preprocessing includes those using geometric information [21, 28, 40], hierarchical decomposition [35], and landmark distances [19]. We take the latter approach, which is simple, applies to a potentially wider range of problems (as it needs no geometric information), and allows quick preprocessing (and therefore can handle very large graphs). Figure 1 illustrates our version of this algorithm.

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

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

Visiting a small portion of the graph not only improves the running time of the algorithm, but suggests an external memory implementation. One can keep the graph and preprocessing data in secondary storage (e.g., disk or flash memory) and the data required for the visited portion of the graph in main memory (RAM). This approach is interesting because some applications work on large graphs, run on small devices, or both. Mobile or handheld GPS devices with automatic routing capability are good examples of such applications.

The shortest path problem has also been studied in the external memory context, where the goal is to minimize the number of blocks read from secondary storage. Hutchinson et al. [23] present a data structure that supports queries on planar graphs with $O(\sqrt{n})$ block reads per query, but the preprocessing data requires $O(n^{1.5})$ space. The single-source [3, 31] and all-pairs [2] versions of the problem have also been studied, but these variants require looking at the whole input. In our case, we need to look only at part of the input, and assume that the relevant data for this part will fit in the primary memory. Therefore, we must make sure this part is as small as possible.

The goal of the work presented in this paper is to test and advance the state of the art in P2P algorithms. We engineered an external memory implementation of the ALT (A^* search, landmarks, and triangle inequality) algorithm of [19] for a Pocket PC with graph and landmark data stored in flash memory. We tested it on several road networks, including one of North America (NA) with almost 30 million vertices. We are not aware of any previous implementation that solves the P2P problem on small devices on instances of this size

The contributions of our paper that make this possible fall into two categories: general improvements and a memory-efficient implementation.

In the first category, we suggest some modifications to the ALT algorithm that improve its performance in general, and not only in the external memory case. During preprocessing, the ALT algorithm selects a set of landmarks and precomputes distances between each landmark and all vertices. Then it uses these distances to compute lower bounds for an A^* search-based shortest path algorithm. Our most interesting contribution allows us to dynamically adjust the set of *active landmarks*, those that are actually used for the current computation. We also suggest new overall landmark selection strategies and introduce pruning techniques that further reduce the search space. This improves both the average- and worst-case performance of the algorithm. Note that the worst case is especially important because if a larger portion of the graph is visited, the primary memory capacity may be exceeded. Section 6 describes these improvements in detail.

Our second major contribution is a memory-efficient implementation of the ALT algorithm. Careful engineering is required to obtain practical results due to memory limitations and to the low speed of the flash card interface. We use space-efficient data structures in combination with caching, data compression, and hashing, as detailed in Section 7.

The experimental results, presented in Section 8, show that our Pocket PC implementation can answer random P2P queries on graphs with about a million vertices in less than 10 seconds. For local queries, the time is one or two seconds, and does not depend much on the input graph size. The implementation also solves random problems on the road graph of North America in minutes. The current bottleneck is reading the data; any improvement in this area will have a significant impact on the overall performance.

2 Preliminaries

The input to the preprocessing stage of the P2P problem is a directed graph $G = (V, E)$ with n vertices and m arcs, and nonnegative lengths $\ell(a)$ for every arc a. The main algorithm also has as inputs a source s and a sink t. The goal is to find a 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. In general, $\mathrm{dist}(v, w) \neq \mathrm{dist}(w, v)$.

A *potential function* is a function from vertices to reals. Given a potential function π, the *reduced cost* of an arc is defined as $\ell_\pi(v, w) = \ell(v, w) - \pi(v) + \pi(w)$. Suppose we replace ℓ by ℓ_π. Then for any two vertices x and y, the length of every x-y path (including the shortest) changes by the same amount, $\pi(y) - \pi(x)$. Thus the two problems are equivalent.

We say that π is *feasible* if ℓ_π is nonnegative for all arcs. The following facts are well-known:

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

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

The first lemma implies that we can often think of $\pi(v)$ as a lower bound on the distance from v to t. The second allows us to combine feasible lower bound functions into a function that is also feasible, and whose value at any point is at least as high as any original one.

3 Labeling Method and Dijkstra's Algorithm

The labeling method for the shortest path problem [12, 13] finds shortest paths from the source to all vertices in the graph. The method works as follows (see for example [37]). It maintains for every vertex v its distance label $d(v)$, parent $p(v)$, and status $S(v) \in \{\texttt{unreached}, \texttt{labeled}, \texttt{scanned}\}$. Initially $d(v) = \infty$, $p(v) = nil$, and $S(v) = \texttt{unreached}$ for every vertex v. The method starts by setting $d(s) = 0$ and $S(s) = \texttt{labeled}$. While there are labeled vertices, the method picks a labeled vertex v, *relaxes* all arcs out of v, and sets $S(v) = \texttt{scanned}$. To relax an arc (v, w), one checks if $d(w) > d(v) + \ell(v, w)$ and, if true, sets $d(w) = d(v) + \ell(v, w)$, $p(w) = v$, and $S(w) = \texttt{labeled}$.

If the length function is nonnegative, the labeling method terminates with correct shortest path distances and a shortest path tree. Its efficiency depends on the rule to choose a vertex to scan next. We say that $d(v)$ is *exact* if it is equal to the distance from s to v. It is easy to see that if one always selects a vertex v such that, at the selection time, $d(v)$ is exact, then each vertex is scanned at most once. Dijkstra [8] (and independently Dantzig [6]) observed that if ℓ is nonnegative and v is a labeled vertex with the smallest distance label, then $d(v)$ is exact. We refer to the labeling method with the minimum label selection rule as *Dijkstra's algorithm.*

Theorem 3.1. [8] *If ℓ is nonnegative then Dijkstra's algorithm scans vertices in nondecreasing order of distance from s and scans each vertex at most once.*

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

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

The *bidirectional algorithm* [6, 10, 33] alternates between running the forward and reverse versions of Dijkstra's algorithm, each maintaining its own set of distance labels (d_f and d_r, respectively). During initialization, the forward search scans s and the reverse search scans t. The algorithm also maintains the length of the shortest path seen so far, μ, and the corresponding path. Initially, $\mu = \infty$. When an arc (v, w) is scanned by the forward search and w has already been scanned in the reverse direction, we know the shortest s-v and w-t paths have lengths $d_f(v)$ and $d_r(w)$, respectively. If $\mu > d_f(v) + \ell(v, w) + d_r(w)$, we have found a path shorter than those seen before, so we update μ and its path accordingly. We perform similar updates during the reverse search. The algorithm terminates when the search in one direction selects a vertex already scanned in the other. Intuitively, the bidirectional algorithm searches two touching balls centered at s and t.

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

Theorem 3.2. *[34] If the sink is reachable from the source, the bidirectional algorithm finds a shortest 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 $\pi_f : V \to \mathbb{R}$ such that $\pi_f(v)$ gives an estimate on the distance from v to t. In the context of this paper, *A^* search* [9, 22] is an algorithm that works like Dijkstra's algorithm, except that at each step it selects a labeled vertex v with the smallest *key*, defined as $k_f(v) = d_f(v) + \pi_f(v)$, to scan next. It is easy to see that A^* search is equivalent to Dijkstra's algorithm on the graph with length function ℓ_{π_f}. If π_f is feasible, ℓ_{π_f} is nonnegative and Theorem 3.1 holds. We refer to the class of A^* search algorithms that use a feasible function π_f with $\pi_f(t) = 0$ as *lower-bounding algorithms.*

Note that the selection rule used by A^* search is a natural one: always choose a vertex on an s-t path with the shortest estimated length. In particular, if π_f gives exact distances to t, the algorithm scans only vertices on shortest paths from s to t. If the shortest path is unique, the algorithm scans exactly the vertices on the shortest path except t.

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

Theorem 4.1. [19] *The set of vertices scanned by A^* search using π'_f is contained in the set of vertices scanned by A^* search using π_f.*

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

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

If π_f and π_r are not consistent, the forward and reverse searches use different length functions. When the searches meet, we have no guarantee that the shortest path has been found. To overcome this difficulty, we can work with consistent potential functions or develop a new termination condition.

We use the former approach. Assume π_f and π_r give lower bounds to the sink and from the source, respectively. Ikeda et al. [24] suggest using an *average function*, defined as $p_f(v) = \frac{\pi_f(v) - \pi_r(v)}{2}$ for the forward computation and $p_r(v) = \frac{\pi_r(v) - \pi_f(v)}{2} = -p_f(v)$ for the reverse one. Although p_f and p_r usually do not give lower bounds as good as the original ones, they are feasible and consistent. To make the algorithm more intuitive, we add $\pi_r(t)/2$ to the forward function (thus making $p_f(t) = 0$) and $\pi_f(s)/2$ to the reverse function (making it zero at s). Because these terms are constant, the functions remain consistent.

5 ALT Algorithms

The main idea behind ALT algorithms is to use landmarks and triangle inequality to compute feasible lower bounds. We select a small subset of vertices as *landmarks* and, for each vertex in the graph, precompute distances to and from every landmark. Consider a landmark L: if $d(\cdot)$ is the distance *to* L, then, by the triangle inequality, $d(v) - d(w) \leq \text{dist}(v, w)$; if $d(\cdot)$ is the distance *from* L, $d(w) - d(v) \leq \text{dist}(v, w)$. To get the tightest lower bound, one can take the maximum of these bounds, over all landmarks. Intuitively, the best lower bounds on $\text{dist}(v, w)$ are given by landmarks that appear "before" v or "after" w.

During an s-t shortest path computation, Goldberg and Harrelson [19] suggest using only a subset of the available landmarks: those that give the highest lower bounds on the s-t distance. This tends to improve performance because most remaining landmarks are unlikely to help in this computation. We suggest a further improvement of this idea in the next section.

Finding good landmarks is critical for the overall performance of ALT algorithms. The simplest way of selecting landmarks is to pick them at random. This works reasonably well, but one can do better. In Section 6.3, we revisit some of the landmark selection algorithms suggested in [19] and propose new ones.

6 Improvements to the ALT Algorithm

In this section we discuss improvements to the main and preprocessing stages of the ALT algorithm. We describe them in the more general context of its bidirectional version.

6.1 Restarting.

As we shall see below, we sometimes need to change the way lower bounds are calculated in the middle of an ALT computation, e.g., by replacing the current potential functions p_f and p_r by another consistent pair. Let S be the set of vertices scanned by the forward search and V_s the corresponding set of labeled vertices (those in its priority queue). Define T and V_t similarly for the reverse search. Note that S and T are disjoint but V_s and V_t may intersect. At this point we know the distances from s to all vertices in S and from all vertices in T to t. The bidirectional algorithm, as described above, stops when one of the searches is about to scan a vertex already scanned by the other search (i.e., when the searches meet). If we replace p and do not want to scan vertices in S and T again, it is unclear when to stop the algorithm. In particular, one cannot always stop if the forward search is about to scan a vertex in T. We need another stopping criterion.

We introduce the new stopping criterion in the context of the bidirectional Dijkstra's algorithm first (for simplicity, we assume the path does exist):

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

One can easily show that the new stopping condition is correct. Since the minimum label for each search is monotone in time, so is their sum. Then, after the condition is met, every vertex x removed from a priority queue will be such that the sum of the distances from s to x and from x to t will be at least μ, which implies that no path shorter than μ exists.

The new stopping condition is at least as strong as the standard one. When we are about to scan a vertex v that has been scanned in the opposite direction, the sum of the s-v and v-t distances is at least μ and the sum of the minimum labels is at least the sum of the distances. After the new stopping condition is satisfied and until the searches meet, the length of the shortest s-t path through any vertex v we scan is at least μ, so no shorter path is discovered.

Since A^* search is equivalent to Dijkstra's algorithm on the graph with arc lengths replaced by reduced costs, we can use the new stopping condition with the modified length function. Next we derive the exact stopping condition. Let k_f^* and k_r^* be the smallest keys of labeled vertices in the forward and reverse directions, respectively (these are the first keys in each priority queue). Let μ_p be the reduced cost of the shortest path seen so far with respect to p_f (or p_r, as the two functions are consistent). The stopping condition is $k_f^* + k_r^* \geq \mu_p$. To get the condition in terms of μ, note that $\mu_p = \mu + p_f(s) - p_f(t)$ and $p_f(t) = 0$. Thus the stopping condition is $k_f^* + k_r^* \geq \mu + p_f(s)$.

Now, if we want to replace p_f and p_r, all we need to do is change the keys of the labeled vertices for the two searches appropriately and update the priority queues containing these vertices. This takes time proportional to the number of labeled vertices at this point.

After restarting, the algorithm proceeds as a regular ALT algorithm with the following modification. If a search (forward or reverse) scans an arc (v, w) and w has been scanned in the same direction, even before the algorithm has been restarted, nothing is done for w.

To prove correctness of the modified algorithm, consider the following transformation. Suppose we have an instance of the P2P problem and two disjoint sets, A and B, such that we know the distances $d_f(v)$ from s to all vertices $v \in A$ and the distances $d_r(w)$ from all vertices $w \in B$ to t. Suppose we contract all vertices in A with s and for every arc (v, w) with $v \in A$, we change the length of the corresponding arc (s, w) in the contracted graph to $\ell(v, w) + d_f(v)$. We make the corresponding transformation for t and B. Let G and G' be the original and the transformed graphs, respectively.

Lemma 6.1. *The P2P problems on G and G' are equivalent.*

Proof. Every path P in G' corresponds to a path of the same length in G as follows. Suppose the first arc of P corresponds to an arc (a, b) of G and the last arc corresponds to (c, d). Then the desired path in G is obtained by concatenating the shortest path from s to a, P, and the shortest path from d to t.

Now consider a path P in G. Let a be the last vertex of $\{s\} \cup A$ on P and let d be the first vertex of $\{t\} \cup B$ on the suffix of P that starts at a. Then the arcs corresponding to those on the segment of P between a and d form a path in G' with the same length as P. $\square$

Theorem 6.1. *The modified algorithm is correct.*

Proof. Consider a restart of the algorithm. Define A as the set of vertices scanned by the forward search up to this point and B as the set of vertices scanned by the reverse search. If we transform the graph as described above and scan s forward and t backward, we get exactly the same configurations of the forward and the reverse priority queues as we do in the actual algorithm, and from this point on the computation of the original algorithm corresponds to a bidirectional Dijkstra's algorithm computation on the transformed graph. An induction on the number of restarts of the algorithm completes the proof. $\square$

6.2 Active landmarks. The original implementation of ALT uses, for each shortest path computation, only a subset of h *active* landmarks, those that give the best lower bounds on the s-t distance. With this approach, the total number of landmarks is limited mostly by the amount of secondary storage available. The choice of h depends on the tradeoff between the search efficiency and the number of landmarks that have to be examined to compute a lower bound.

We improve this idea by updating the set of active landmarks dynamically. We start with only two active landmarks: one that gives the best bound using distances *to* landmarks and another using distances *from* landmarks. Then we periodically try to update the active set by adding new landmarks until the algorithm terminates or the total number of active landmarks reaches an upper bound (six, in our experiments).

Update attempts happen whenever a search (forward or reverse) scans a vertex v whose distance estimate to the destination, as determined by the current lower bound function, is smaller than a certain value (a *checkpoint*—more on that later). At this point, the algorithm verifies if the best lower bound on the distance from v to the destination (using all landmarks) is at least a factor $1 + \epsilon$ better than the current lower bound (using only active landmarks). If so, the landmark yielding the improved bound is activated. In our experiments, we used $\epsilon = 0.01$.

Intuitively, the computation starts using the initially best landmarks. As it progresses and the location of vertices for which lower bounds are needed changes, other landmarks may give better bounds, and should be brought to the active set. After every landmark update, the potential function changes, and we must update the priority queues as described in the previous section.

Checkpoints are determined by the original lower bound b on the distance between s and t, calculated before the computation starts. The i-th checkpoint for each search has value $b(10-i)/10$: we first try to update the landmarks when estimated lower bounds reach 90% of the original value, then when they reach 80%, and so on. This rule works well when s and t are reasonably far apart; when they are close, update attempts would happen too often, thus dominating the running time of the algorithm. Therefore, we also require the algorithm to scan at least 100 vertices between two consecutive update attempts in the same direction.

The dynamic selection of active landmarks improves efficiency, especially for hard instances. It also reduces the running time, as the final number of active landmarks is usually very small (close to three).

6.3 Landmark selection. This section describes several landmark selection strategies. Some have been introduced in [19], and some are new or improved. The new strategies do not use graph layout information, yet they work better than the best method of [19] that does.

The NA graph is quite large and a single-source shortest path computation on it takes around 10 seconds on the machine we used. Any practical preprocessing method cannot perform too many of those; n shortest path computations would take years. We restricted ourselves to more efficient preprocessing.

Note that one can define "optimal" landmark selection in many ways depending on how landmark quality is measured. For example, one can aim to minimize the total number of vertices visited for all $O(n^2)$ possible shortest path computations. Alternatively, one can minimize the maximum number of vertices visited. The related optimization problems are probably hard. Even an exact comparison of two sets of landmarks, although polynomial-time, is impractical except for small graphs.

6.3.1 Random. The simplest way to select landmarks is at random. One can generate several sets of random landmarks and pick the set that works the best in practice. Not surprisingly, one can do better.

6.3.2 Farthest. Originally proposed in [19], *farthest selection* works as follows. Pick a start vertex at random and find a vertex v that is farthest away from it. Add v to the set of landmarks. Proceed in iterations, always adding to the set the vertex that is farthest from it. In this paper, we introduce a slightly modified version: distances are measured in terms of number of hops instead of using the actual length function $\ell(\cdot)$; in other words, we use BFS instead of Dijkstra's algorithm to compute distances. This biases the algorithm towards dense regions. We shall refer to the original version as *farD* and the new one as *farB*.

Farthest selection is very efficient. Selecting landmarks takes less time than computing distances to and from landmarks. It requires little space, since landmark distances can be output as soon as computed and never used in preprocessing again. However, with more time and space one can find better landmarks.

6.3.3 Planar. Also proposed in [19], *planar* landmark selection is an example of a method that uses graph layout information. It picks a point close to the center of the graph and divides the map into sectors originating from this point (this can be done efficiently by sorting the vertices in polar coordinates). The farthest point in each sector is selected as a landmark. In addition, if a vertex near a sector boundary is selected, adjacent vertices of the neighboring sector are skipped to ensure no two landmarks are close. In our experiments, we obtained slightly better results by picking as the reference a point close to the median (instead of center) of the graph, and using BFS (instead of Dijkstra's algorithm) to compute distances while computing the median. We report only results with the modified version in this paper.

6.3.4 Avoid. We now introduce a new landmark selection method. Assume there is a set S of landmarks already selected and that we want an additional landmark. First, compute a shortest-path tree T_r rooted at some vertex r. Then calculate, for every vertex v, its *weight*, defined as the difference between $\text{dist}(r, v)$ and the lower bound for $\text{dist}(r, v)$ given by S. This is a measure of how bad the current distance estimates are.

For every vertex v, now compute its *size* $s(v)$, which depends on T_v, the subtree of T_r rooted at v. If T_v contains a landmark, $s(v) = 0$; otherwise, $s(v)$ is the sum of the weights of all vertices in T_v. Let w be the vertex of maximum size. Traverse T_w, starting from w and always following the child with the largest size, until a leaf is reached. Make this leaf a new landmark.

We call this method *avoid*. It tries to identify regions of the graph that are not "well-covered" by avoiding existing landmarks. No path from w to a vertex in its subtree has a landmark "behind" it. By adding a leaf of this tree to the set of landmarks, *avoid* tries to improve the coverage.

A natural way of picking r (the root vertex) is uniformly at random. We obtained better results by picking with higher probability vertices that are far from the existing landmarks.

6.3.5 Optimization. A downside of constructive heuristics, such as the ones described above, is that some landmarks selected earlier on might be of limited usefulness once others are selected. It seems reasonable to try to replace them with better ones. We use local search [1] for this purpose.

To implement the search, we need a way to measure how good a solution (set of landmarks) is. Ultimately, the goal is to find a solution that makes all point-to-point searches more efficient, but that is prohibitively expensive, as already mentioned. In practice, we can only estimate the quality of a given set of landmarks.

We use reduced costs for the estimation. In this context, we define the reduced cost of an arc with respect to landmark L as $\ell(v, w) - d(L, w) + d(L, v)$. If the reduced cost is zero, we say that the landmark *covers* the arc. The best case for the point-to-point shortest path algorithm happens when a landmark covers every arc on the path. With that in mind, we define the *cost* of a given solution as the number of arcs that have zero reduced cost with respect to at least one landmark. Less costly solutions are better: for a fixed k, we are interested in finding a set of k landmarks that covers as many arcs as possible.

Determining which arcs a given vertex covers requires performing a single-source shortest path computation. For large graphs, it is impractical to do this for all vertices. Therefore, we work only with a small set of candidate landmarks, selected using *avoid*.

More precisely, let C be the set of candidates, initially empty. We start by running *avoid* to find a solution with k landmarks, all of which are added to C. We then remove each landmark from the current solution with probability 1/2. Once they are removed, we generate more landmarks (also using *avoid*) until the solution has size k again. Each new landmark that is not already in C is added to it. We repeat this process until either C has size $4k$ or *avoid* is executed $5k$ times (whichever happens first). The second condition is important to limit the running time of the algorithm; not every execution of *avoid* will generate a new landmark.

Eventually, we have a set C with between k and $4k$ landmarks. Interpreting each landmark as the set of arcs that it covers, we need to solve an instance of the maximum cover problem. Since it is NP-hard, we resort to a multistart heuristic. Each iteration starts with a random subset S of C with k landmarks and applies a local search procedure to it. In the end, we pick the best solution obtained across all iterations. We set the number of iterations to $\lfloor \log_2 k + 1 \rfloor$.

The local search procedure is based on swapping landmarks. It tries to replace one landmark that belongs to the current solution with another that does not (but belongs to the candidate set). It works by computing the profit associated with each of the $O(k^2)$ possible swaps. It discards those whose profit is negative or zero. Among the ones that remain, it picks a swap at random with probability proportional to the profit. The same procedure is then applied to the new solution. The local search stops when it reaches a local optimum, i.e., a solution on which no improving swap can be made. Each iteration of the local search takes $O(km)$ time.

We call this method of landmark generation *maxcover*. The optimization phase is quite fast. The running time is dominated by calls to *avoid* used to generate the set of candidate landmarks.

6.3.6 Other approaches. We experimented with many other selection strategies. Devising reasonably-sounding strategies is easy and combinations of such strategies are numerous. Some ideas work well, others do not. While better selection strategies probably do exist, one cannot expect an improvement of an order of magnitude on the average efficiency. It is already above 10% for the largest graph we tested, and even higher for smaller graphs. The worst case can be improved, but one must keep in mind that preprocessing times are an issue.

6.4 Pruning. Pruning may reduce the time and memory requirements of the algorithm. We describe it for the forward search. The reverse is symmetric.

Suppose we are scanning an arc (v, w). Normally, we check if $d_f(v) + \ell(v, w) < d_f(w)$; if so, we update $d_f(w)$ and the forward priority queue. With pruning, we also check if $d_f(v) + \ell(v, w) + \pi_f(w) < \mu$. When this inequality is not true, the shortest s-t path through (v, w) does not improve upon the current shortest path. Therefore, there is no need to store an updated value of $d_f(w)$. Note that the lower bound function used for pruning does not need to be consistent.

7 External Memory Implementation

Our implementation stores graph and landmark data on a flash memory card. System constraints dictate that the minimum amount one can read from the card is a 512-byte sector. We read data in pages, with a page containing one or more sectors. (One seems best for large graph data with limited locality.) As Section 8 shows, reading is slow. This, together with the fact that not all data in a block is actually used, makes reading the bottleneck of our implementation. This motivates some of our choices; others are motivated by the limited amount of primary memory on the Pocket PC.

7.1 Graph representation. Our graph is stored in the flash card in the following format. Arcs are represented as an array of records sorted by the arc tail. Each record has a 16-bit arc length (in our case, transit time in seconds) and the 32-bit ID of the head vertex. Another array represents vertex records, each consisting of the 32-bit index of the record representing the first outgoing arc. The reverse graph is also stored (in the same format).

Additional information needed for each vertex visited by a search is kept in main memory in a record we call a *mutable node*. Each vertex may need two mutable nodes, one for the forward and another for the reverse search. A mutable node contains four 32-bit fields: an ID, a distance label, a parent pointer, and a heap position. Some fields are bigger than needed even for our largest graph, but we chose to make the records word-aligned to keep the implementation clean and flexible. The user specifies M, the maximum number of mutable nodes allowed. The total amount of RAM used is proportional to M.

7.2 Data structures. To map vertex IDs to the corresponding mutable nodes, we use double hashing with a table of size at least $1.5M$. We maintain two priority queues, one for each search. For shortest path algorithms, the improved multi-level bucket implementation tends to be the fastest [7, 17], and we did use it on the landmark generating routines. For P2P computations, however, we used 4-heaps. Although slower than multi-level buckets, 4-heaps have less space overhead (one heap index per vertex). In addition, the priority queue never contains too many elements in our application, so the overhead associated with heap operations is modest compared to that of data access. The maximum size of each heap was set to $M/8 + 100$ elements.

7.3 Caching. The data we deal with has strong locality. On partial graphs, at least 50% of the arcs are between vertices whose IDs differ by 15 or less; for more than 90% of the arcs the difference is at most 100. On the NA graph, more than 99% of the arcs have endpoints whose IDs differ by at most 10. For this reason, and also because data must be read in 512-byte blocks, our algorithm implements an explicit caching mechanism. A page allocation table maps physical page addresses to virtual page addresses (in RAM), and the replacement strategy is LRU: the least recently used page was evicted whenever necessary. We use separate caches for graphs and landmarks. Each of the six landmark caches (one for each active landmark) has 1 MB, and each of the two graph caches has 2 MB.

7.4 Landmark representation and compression. We store data for each landmark in a separate file. Each distance is represented by a 32-bit integer. To and from distances for the same vertex are adjacent. Although the graph is not completely symmetric, the two distances are usually close. Moreover, since vertices with similar IDs tend to be close to each other, their distances to (or from) the landmark are also similar.

This similarity is important for compression, which allows more data to fit in the flash card and speeds up data read operations. We use the fact that the two most significant bytes of adjacent words (distances) tend to be the same—there are actually long runs in which these bytes coincide. For each run, we represent the common bytes just once, together with the run length; only the two least significant bytes are represented explicitly for each element. The resulting compression ratio is almost 50%, which is a few percentage points better than what is achieved by the standard compression program `gzip`. To allow random access to the file, each page is compressed separately. Since compression rates vary, the file has a directory with page offsets.

8 Experimental Analysis

8.1 Setup. Our preprocessing was done in memory, with landmarks output to disk. Most graphs were preprocessed on a Pentium 4M with 512 MB of RAM running at 2.2 MHz. The only exception is the (much larger) NA graph, for which we used a 900 MHz Itanium 2 workstation with 11.9 GB of RAM.

The Pocket PC we used was a Toshiba 800e. It has a 400 MHz ARM-4 processor and 128 MB of RAM, and runs the Windows Mobile 2003 Second Edition operating system. We set the system file cache size to 2 MB. This is the maximum allowed by the operating system, and big enough to fit the file allocation table. In some experiments, we reduced the clock speed to 100 MHz to check if the computation is CPU bound.

Data for the P2P algorithm was stored on Compact Flash memory cards. For the NA graph, we used a 4 GB Lexar 80x card with FAT32 file system. For smaller graphs, we used a 2 GB Lexar 80x card with FAT file system. To measure their speed, we created a 32 MB file and read a sequence of 512-byte blocks starting at random positions aligned at multiples of 512 bytes. As Table 1 shows, the throughput in this case is quite low, and it depends on CPU frequency.

CARD	THROUGHPUT		
CAPACITY	@100	@400	RATIO
2 GB	234	379	1.62
4 GB	228	366	1.61

Table 1: Throughputs of each flash card (in KB/s) for accesses to random 512-byte blocks with the CPU at 100 MHz and 400 MHz.

Our code was written in C++ and compiled under eMbedded Visual C++ 4.0 (for the Pocket PC) and Visual C++ 7.0 (for PCs). We use the *Mersenne Twister* pseudorandom number generator [29].

We experimented with six road network graphs extracted from `Mappoint.NET` data: San Francisco Bay Area (including San Jose and Sacramento), Dallas area,

GRAPH	DIMENSIONS		FILE SIZES (MB)		
	VERTICES	ARCS	GRAPH	LANDMARK	TOTAL
Bay Area	330 024	793 681	5.80	1.34	42.42
Los Angeles	563 992	1 392 202	10.12	2.21	71.07
St Louis	588 940	1 370 273	10.09	2.31	73.31
Dallas	639 821	1 522 485	11.15	2.51	80.03
Washington	991 848	2 294 870	16.91	3.90	123.52
North America	29 883 886	70 297 895	516.25	117.50	3 735.00

Table 2: Road network problems: graph dimensions and file sizes in MB (of a single graph file, of a single landmark file, and the total considering two graph files and 23 landmark files).

Los Angeles area, St Louis area, Washington State and vicinity (including Vancouver Island and Portland), and the Continental North America (Canada, the United States, and parts of Mexico). We refer to the first five graphs as *partial*; although by no means small, these graphs are much smaller than the last one. We use road segment transit times as arc lengths (using road segment lengths instead has little effect on the performance of ALT, as shown in [19]). Table 2 reports the problem sizes, including the average size of each of the two graph files (forward and reverse) and of a typical landmark file. It also reports the total space the algorithm requires when working with 23 landmarks, the number we used on the Pocket PC.

Following [19], we use two kinds of (s,t) pair distributions. One selects the endpoints uniformly at random (RAND), which tends to produce pairs of far away vertices. Another is BFS, which selects s at random, does breadth-first search to find all vertices that are 50 hops away from s, and chooses t at random from these; this produces a local pair. Our design choices were made with the random distribution in mind, since the searches it induces cost much more. Therefore, most of our experiments will focus on the random distribution; only those in Section 8.3 use BFS.

We designed experiments to test several variants of the algorithm. For each of the partial graphs, we picked a random set of 1000 s-t pairs and ran the P2P algorithm on it for all variants tested. For consistency, each partial graph is tested with the same set of 1000 pairs on all experiments. Most experiments on NA, which take longer to run, used 100 pairs only.

To compare two variants of the algorithm, the obvious measure of performance is running time. However, with many design choices to evaluate, most of the experiments were done on a PC, which is much faster, mainly due to disk caching: for partial graphs, all data fits in the cache. The flip side of this speedup is that the running times are unsteady: different runs of the same algorithm may have completely different running times depending on the initial cache state. For this reason, and to get a better understanding of the algorithm, we use five machine-independent measures of quality.

The first three measures refer to the number of mutable nodes (i.e., vertices visited): the average, the 99th percentile, and the maximum. We are interested in optimizing the worst case of the algorithm. Since we test only a fraction of the $O(n^2)$ possible pairs, the variance on the maximum can be quite high. The 99th percentile tends to be a more stable measure of the relative performance of two different landmark selection methods. For NA, on which only 100 pairs were tested, we often report the 97th percentile instead.

The fourth measure is the average *efficiency*, defined as the ratio between the number of vertices scanned and the actual number of vertices on the path. Finally, the fifth measure is the average amount of data read from secondary storage per search. This correlates well with the running time of the algorithm.

All experiments comparing different design choices were made on the PC, and for them only the machine-independent measures are presented. Running times are shown only for the runs made on the Pocket PC (see Section 8.3).

8.2 Design choices. Our two main improvements to the original ALT algorithm are the new landmark selection schemes (*avoid* and *maxcover*) and the dynamic selection of active landmarks. There are also some minor improvements, such as a new pruning strategy.

We start by defining a reference version of our algorithm, with the best choice of each of the parameters tested. It uses landmarks obtained with the *maxcover* method. To solve the P2P problem, we used dynamic selection of active landmarks, starting with two and increasing this number as necessary (up to six). Pruning was also used.

Table 3 presents data for this reference version, with 16 landmarks. For this experiment only, we used 1000 pairs to test NA. The average efficiency was above 26% over the five partial graphs and above 10% for the NA graph. Comparing this with the results

GRAPH	MUTABLE NODES			EFF.	DATA	GENER.
	AVG	99TH	MAX	(%)	(KB)	TIME (s)
Bay Area	4 258	21 567	31 065	29.11	542	94
Los Angeles	7 242	46 759	93 801	26.52	700	168
St Louis	6 575	31 135	53 761	29.22	564	207
Dallas	8 145	49 913	86 784	26.86	682	173
Washington	10 159	48 407	111 205	33.16	808	248
North America	303 148	1 700 827	3 608 315	10.65	51 461	12 488

Table 3: Results with 16 landmarks generated with *maxcover* on 1000 random pairs: number of mutable nodes (average, 99th percentile, and worst), average efficiency, and average amount of data read from disk. The last column presents the total time required to generate the landmarks, in seconds.

reported in [19], the improvements on the original ALT implementation become clear. The original ALT efficiency (with 16 landmarks) was 7.8% on Bay Area, 5.6% on Los Angeles, 10.6% on St. Louis, and 7.1% on Dallas (these were the only problems tested in both studies).

Regarding the number of mutable nodes visited by the algorithm, the table shows that, on average, they are just over 1% of the total number of vertices on all graphs, including NA. Moreover, on at least 99% of the pairs, at most 8% of the vertices were visited. The worst cases observed ranged from 9% (St Louis) to 17% (Los Angeles) of the vertices in the graph.

Note that the 99th percentile is often closer to the average than to the worst case, which suggests that the distribution is far from uniform, and increases sharply towards the end. Figure 2, which shows the full histogram for Bay Area and Washington, confirms this. The curves for the other three partial graphs, omitted for clarity, fall between those two. The curve for NA has a similar shape.

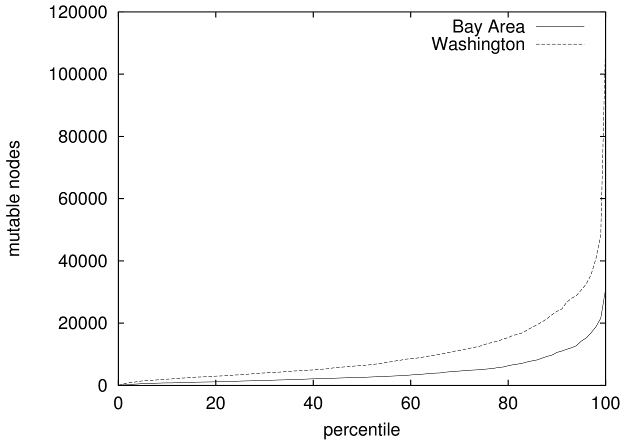

Figure 2: Distribution of the number of mutable nodes for each graph, with 1000 random pairs.

In absolute terms, the performance of the algorithm on NA was worse than on the smaller graphs. However, the fraction of vertices that were visited (mutable nodes) was quite similar, and the average efficiency was not much worse. The amount of data read per mutable node, however, increased from less than 100 bytes in most partial graphs to almost 170 bytes. This is because larger maps have less locality, since the search spreads over a larger portion of the graph.

In the remaining experiments in this section, we vary one or more of the parameters used to build Table 3. All parameters that are modified are explicitly mentioned; those that are not remain the same as above.

8.2.1 Landmark selection. We start by analyzing the effect of the landmark selection scheme on solution quality. We compare *maxcover* with five alternative methods. For each graph and each method, we generated three sets of landmarks (with different random seeds), and picked the best (the one with the smallest 99th percentile when processing 1000 random pairs). For NA, we used the 97th percentile and 100 pairs.

Table 4 shows the results for partial graphs. Note that all measures are relative to *maxcover*. For each graph, we compute the ratio between the value obtained by the method and what was obtained by *maxcover*, then report the geometric mean of these ratios over all five graphs.[1]

Observe that *random* is the worst method on all measures except the time to generate the landmarks. Even in this case, it is followed closely by the original farthest selection (*farD*), which is better on all other accounts. Farthest selection with BFS (*farB*) provides even better results and is still relatively fast. Note that *farB* is slower than *farD* because BFS does not

[1]We use geometric instead of arithmetic means because it makes it easy to change the reference method. To see how all methods fare with respect to *random*, for example, just divide every entry in a column by the value in *random*.

SELECTION SCHEME	MUTABLE AVG	MUTABLE 99TH	AVG EFF.	DATA READ	GEN. TIME
random	2.11	2.64	0.66	1.55	0.07
farD	1.55	1.77	0.80	1.29	0.08
farB	1.39	1.48	0.84	1.21	0.13
planar	1.33	1.32	0.83	1.15	0.13
avoid	1.20	1.22	0.90	1.10	0.20
maxcover	1.00	1.00	1.00	1.00	1.00

Table 4: Effect of the landmark selection scheme on the P2P algorithm. All values are relative to *maxcover*. Only the five partial graphs are considered.

replace the shortest path computation: *farB* still has to compute the actual shortest path trees in order to output the landmark files.

Planar landmark selection takes roughly the same time as *farB* and has slightly better performance. *Avoid*, the first completely new method proposed here, is even better, despite not using any geometric information. Being only three times slower than *random*, it can still be considered fairly quick.

The reference method, *maxcover*, is roughly five times slower than *avoid*, but it is clearly superior on all other measures. It should be the method of choice among those tested if one can afford the extra time.

Results for NA are presented in Table 5. They are similar to Table 4, but there are some differences. The performance of *farD* is much worse—even worse than *random*. This happens because several landmarks are placed in areas that have very few vertices and are far from denser regions (and from each other), such as Alaska, Northern Canada, and Mexico. *FarB* is less susceptible to problems of this kind, and performs better than *planar*.

LAND.	MUTABLE AVG	MUTABLE 97TH	AVG. EFF.	DATA READ	GEN. TIME
farD	3.38	4.71	0.40	6.56	0.10
random	2.34	2.59	0.67	3.40	0.09
planar	1.86	2.46	0.68	2.46	0.15
farB	1.63	2.12	0.74	2.12	0.15
avoid	1.51	1.84	0.86	1.96	0.21
maxcover	1.00	1.00	1.00	1.00	1.00

Table 5: Effect of the landmark selection scheme on the P2P algorithm for NA. All values are relative to *maxcover*.

Another important difference is that on NA the choice of landmarks has much more impact on the performance of the P2P algorithm. On the partial graphs, doubling the average number of mutable nodes (using *random* instead of *maxcover*) increases the amount of data read by 55%. On NA, the difference is greater than 200%.

8.2.2 Number of landmarks. Table 6 shows how performance improves as the number of landmarks increases from 2 to 32 (always using *maxcover*). As in the previous experiment, we actually generated three sets of landmarks in each case and picked the best (with respect to the 99th percentile). All measures are relative to what was obtained with 16 landmarks. Note that all measures of solution quality improve as the number of landmarks increase, but the marginal benefits of each additional landmark decrease. Only the five partial graphs are considered in the table.

LAND.	MUTABLE AVG	MUTABLE 99TH	AVG EFF	DATA READ	GEN. TIME
2	12.43	9.45	0.21	4.25	0.08
4	4.11	4.26	0.41	2.10	0.18
8	1.93	2.19	0.68	1.37	0.42
16	1.00	1.00	1.00	1.00	1.00
23	0.81	0.80	1.15	0.95	1.72
32	0.67	0.66	1.29	0.92	3.28

Table 6: Effect of the total number of landmarks available on the P2P algorithm (partial graphs only). All values are relative to 16 landmarks.

Results for NA, presented on Table 7, are similar. Once again, the main difference is that an improvement on the set of landmarks has greater effect on the amount of data read from external memory.

LAND.	MUTABLE AVG	MUTABLE 97TH	AVG. EFF.	DATA READ	GEN. TIME
2	18.51	17.76	0.20	25.07	0.11
4	5.16	4.90	0.22	9.00	0.22
8	2.55	2.66	0.51	3.22	0.44
16	1.00	1.00	1.00	1.00	1.00
23	0.77	0.72	1.15	0.69	2.12
32	0.64	0.65	1.24	0.51	5.44

Table 7: Effect of the total number of landmarks available on the performance of the P2P algorithm (NA). All values are relative to 16 landmarks.

8.2.3 Active landmarks. We now compare the dynamic and static schemes for selecting active landmarks. For each of the five partial graphs, we ran nine versions of the static algorithm (with h varying from 2 to 10) and compared them to the dynamic scheme. Given a fixed instance, we use the same set of 16 *maxcover* landmarks and the same 1000 random *s*-*t* pairs for all runs.

Figure 3 summarizes the experiment. It shows how the measures taken (average number of mutable nodes, 99th percentile of mutable nodes, average efficiency, and average amount of data read) behave as a function of h. The values are normalized: we consider the ratio between the actual value and what was obtained by the dynamic algorithm. Each point in the plot is the geometric mean of five normalized values (one for each instance). As usual, larger values are better for efficiency, and small values are better for all other measures. The closer a point is to 1.0, the more similar to dynamic selection the corresponding method is. The data is for partial graphs only.

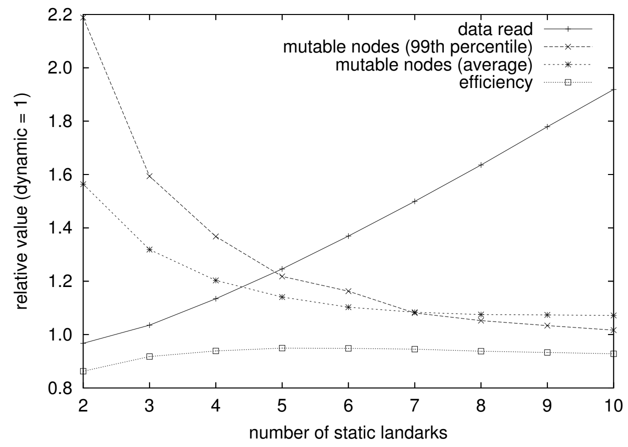

Figure 3: Performance of the P2P algorithm on partial graphs as a function of the number of statically selected landmarks. All values are relative to dynamic selection.

Note that, with dynamic selection, the algorithm reads less than the amount required with three static landmarks, and achieves much better results on average and particularly on hard cases (as the 99th percentile shows).

Results for NA were very similar (we omit the corresponding plot). As with partial graphs, static selection always obtained worse results in terms the average number of mutable nodes and the 99% percentile. Moreover, the average amount of data read was high even for small values of h; for $h = 4$ (the best for static selection), it was 60% larger than with dynamic selection. In terms of efficiency, static selection was better then dynamic for h between 3 and 5, but by no more than 4%.

We interpret these results as follows. Often, one landmark gives very good bounds for the forward search and another one for the reverse search. Both active landmark selection variants choose these landmarks, and the dynamic version never adds new ones. This case is easy for ALT, and the use of more landmarks by the static algorithm only increases the amount of landmark data read. In hard cases, for one or both searches there is no landmark that gives good bounds throughout the computation, and some good landmarks cannot be determined based on their *s*-*t* distance bounds. Static selection must guess which landmarks may become useful, while dynamic selection can wait until is has more information to make better choices.

Furthermore, Theorem 4.1 does not apply to the variant of ALT we study. Bad landmarks can "pull" a search toward themselves and away from from the goal. Dynamic selection only picks landmarks that are better than those currently in use, at least locally. The downside is that a landmark may become useful between checkpoints, and the dynamic selection method will use worse landmarks until the next checkpoint.

8.2.4 Other improvements. We now briefly discuss the effect of pruning and the new stopping criterion on the performance of the algorithm.

On the partial graphs, with 16 *maxcover* landmarks, efficiency decreases by roughly 3% and the average number of mutable nodes increases by more than 6% if pruning is not used. The effect of the new stopping criterion is also modest on average: both the efficiency and the number of mutable nodes get only 3% worse if the old criterion is used.[2] The average results for NA were similar.

However, the effects of both improvements depend heavily on the particular characteristics of each search. Take NA, for instance. On more than half of the searches tested, switching to the old stopping criterion would increase the number of mutable nodes by less than 1%. However, there were searches where the difference was greater than 20%. In 60% of the cases the number of vertices visited would increase by less than 1% if pruning were not used, but we did observe differences as high as 164%.

Interestingly, there was no obvious correlation between the "hardness" of the search (in terms of the number of vertices visited) and the effect of these improvements. Some searches visiting more than a million vertices were largely unaffected, while others benefited greatly. Such diversity was also observed for easier searches.

8.3 Running times. The previous experiments have shown that the best approach (among those tested) is to generate as many landmarks as possible using the

[2]Since the standard stopping criterion cannot be used with dynamically selected landmarks, we compared it with the new stopping criterion using static selection (with $h = 6$).

maxcover method and dynamically choose which ones to use during the actual P2P computation. This section presents running times for this algorithm on a Pocket PC.

The total number of landmarks we could use for NA was limited by the size of the CF Card we had. A formatted four-gigabyte card can only hold 3.74 GB of data. That is enough for 23 landmarks and the graph data; 24 landmarks would require 3.76 GB. For symmetry, we also tested the other graphs with 23 landmarks.

Table 8 reports the performance of the algorithm. Both distributions (BFS and RAND) are considered. For each, we tested the algorithm with 1000 pairs on the partial graphs, and 100 pairs on NA. We ran each experiment twice, with different clock speeds: 100 MHz and 400 MHz. This was done to verify that main bottleneck of our implementation is not CPU, but reading the data from flash.

The results confirm this. Reducing the clock speed by a factor of four increases running times by a factor of 1.68 to 1.85. Compare this with Table 1, which shows that the time to merely read a fixed amount of data already increases by more than 1.6 when the CPU is slowed down. This is evidence that reading the data dominates the running time.

In a related experiment, we tested an in-memory version of our algorithm. Each input file (graphs and landmarks) was preloaded to a separate array in memory, and all calls to `fread` (the C function used to read data from the file) were replaced by calls to `memcpy`. Everything else remained unaltered: the data was still read in blocks, decompressed on the fly (for landmarks), and caching was used. In other words, all the overhead associated with an external memory implementation was still present. Of course, a "pure" in-memory implementation of the ALT algorithm (without block reads, caching, compression, hashing, and other overheads) would be significantly faster.

Still, the version we implemented is already one order of magnitude faster than the external memory algorithm. On the Pocket PC, the average time for Bay Area (with the RAND distribution) was 0.40 seconds at 400 MHz (and 1.21 seconds at 100 MHz).[3] This represents a speedup of roughly 13 when compared with the external memory version, showing that reading the data is indeed the bottleneck of our algorithm.

Even though flash card performance is bad in our application, running times for our code are reasonable. Local (BFS) queries, which would be the most typical in navigation software, took a second or two. For this distribution, performance does not seem to depend much on the graph size: queries take similar amounts of time on small and large maps (they are only slightly larger on NA).

For random pairs, on the other hand, running times clearly depend on graph size. On smaller graphs, the average time was between 5 and 8 seconds. On NA, searches took less than six minutes on average, still respectable considering the size of the data set and the system limitations. The median time (not shown in the table) was only one minute.

On partial graphs, the efficiency of the algorithm is roughly the same for BFS and RAND, which might seem counter-intuitive. This is related to the nature of the ALT algorithm. On a long path, each search (forward and reverse) tends to visit more vertices close to its origin. The middle portion of the path is often perfectly "covered" by a landmark, and therefore is traversed more efficiently. The search depicted in Figure 1 is a good example of this. A search in the BFS distribution can be seen as one with no sizeable middle portion.

Note that we set M, the limit on the maximum number of mutable nodes, to two million. With this value, the program footprint is about 78 MB. This would be reduced to 15 MB if we used 200 000 mutable nodes, more than enough for the partial graphs. For NA, on the other hand, there are cases where two million mutable nodes are not enough, but Table 3 shows that this is rare: at least 99% of the searches visit fewer vertices. None of the 100 pairs tested on the Pocket PC reached this limit.

9 Final Remarks

We improved the ALT algorithm, developed a memory-efficient version of it, and tested it on a Pocket PC. The implementation works even for large graphs, with tens of millions of vertices. Reading data from flash memory is the bottleneck. On graphs corresponding to local area maps, our Pocket PC implementation is fast enough for practical use.

On NA with random pairs of vertices, the implementation is still somewhat slow, but being able to solve problems of this size of the device in minutes is still an achievement. In addition to possible algorithmic improvements, faster flash cards and interfaces are likely in the future. Combined with cheaper and higher capacity flash memory cards becoming available, solving problems of NA size on small devices may become more practical.

Although our implementation is not directly comparable to that of [19], we believe that an in-memory

[3]Note that a four-time increase in clock speed makes the algorithm only three times as fast. We conjecture that this happens because the change in clock frequency does not affect other components—notably RAM.

		MUTABLE NODES			EFF.	DATA	SEARCH TIME		
PAIRS	GRAPH	AVG	99TH	MAX	(%)	(KB)	@100	@400	RATIO
BFS	Bay Area	582	2 040	2 248	28.2	103	2.07	1.20	1.73
	Los Angeles	697	2 737	3 466	26.8	109	2.32	1.36	1.70
	St Louis	341	1 255	1 327	42.9	53	1.33	0.79	1.68
	Dallas	337	921	952	39.1	58	1.38	0.82	1.69
	Washington	400	2 016	3 817	42.9	60	1.47	0.85	1.72
	North America	441	1 785	3 188	33.4	68	3.92	2.15	1.82
RANDOM	Bay Area	3 473	14 974	15 490	32.3	505	9.33	5.13	1.82
	Los Angeles	5 635	35 535	49 926	29.6	650	13.04	7.25	1.80
	St Louis	5 863	36 389	37 299	34.1	561	12.26	6.94	1.77
	Dallas	8 067	46 619	53 863	29.4	726	16.17	8.72	1.85
	Washington	7 154	29 491	39 680	44.9	707	14.59	8.00	1.82
	North America	189 602	974 362	1 804 161	14.7	22 397	552.35	328.78	1.68

Table 8: Pocket PC runs with 23 *maxcover* landmarks: number of mutable nodes, average efficiency, average amount of data read, average running time in seconds at 100 MHz and 400 MHz, and the ratio between them.

variant of our algorithm will be substantially faster. Not only does our algorithm visit fewer vertices (because of higher efficiency), but it also processes each one faster (because the number of active landmarks is reduced with dynamic selection).

We have seen that better landmark selection schemes can significantly improve the performance. We believe there is still room for improvement, particularly for the NA graph. Developing new landmark selection schemes is an obvious path for future research.

Other potential improvements include a more compact representation of landmarks (for example, storing distances only for some vertices and recomputing the rest) and reusing mutable nodes to further reduce the footprint of the algorithm. The implementation would also benefit from further tuning. We have not performed extensive experiments to determine the ideal number of pages to read at a time, the best size of each cache, or the minimum number of vertices visited between checkpoints. Treating the flash card as a raw device may also improve performance.

An open problem is to analyze the performance of ALT on interesting classes of graphs and compare it to (bidirectional) Dijkstra's algorithm. Our approach is related to using beacons to estimate distances in the Internet [14, 32]. The empirical work on beacons received a theoretical justification [27], suggesting a path to explain the good performance of ALT in practice.

References

[1] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization.* Wiley, 1997.

[2] L. Arge, U. Meyer, and L. Toma. External Memory Algorithms for Diameter and All-Pairs Shortest-Paths on Sparse Graphs. In J. Diaz, J. Karhumäki, A. Leistö, and D. Sannella, editors, *Proc. 31st International Colloquim on Automata, Languages, and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 2004.

[3] G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths. In T. Hagerup and J. Katajainen, editors, *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer-Verlag, 2004.

[4] 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.

[5] 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.

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

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

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

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

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

[11] 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.

[12] L. R. Ford, Jr. Network Flow Theory. Technical

Report P-932, The Rand Corporation, 1956.

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

[14] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, and Y. Shavit. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Trans. on Networking*, 2001.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *DAM*, 126:55–82, 2003.

[24] 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.

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

[26] 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.

[27] J. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. In *Proc. 45th IEEE Symposium on Foundations of Computer Science*, pages 444–453, 2004.

[28] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *IfGIprints 22, Institut fuer Geoinformatik, Universitaet Muenster (ISBN 3-936616-22-1)*, pages 219–230, 2004.

[29] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[30] 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.

[31] U. Meyer and N. Zeh. I/O-Efficient Undirected Shortest Paths. In G. Di Battista and U. Zwick, editors, *Proc. 11th European Symp. on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 434–445. Springer-Verlag, 2003.

[32] E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approach. In *IEEE INFOCOM*, 2002.

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

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

[35] 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.

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

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

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

[39] 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.

[40] D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proc. 11th Annual European Symposium on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer-Verlag, 2003.

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

[42] 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.