Point-to-Point Shortest Path Algorithms with Preprocessing

Andrew V. Goldberg

Microsoft Research – Silicon Valley
1065 La Avenida, Mountain View, CA 94062, USA goldberg@microsoft.com
URL: http://www.research.microsoft.com/~goldberg/

Abstract. This is a survey of some recent results on point-to-point shortest path algorithms. This classical optimization problem received a lot of attention lately and significant progress has been made. After an overview of classical results, we study recent heuristics that solve the problem while examining only a small portion of the input graph; the graph can be very big. Note that the algorithms we discuss find exact shortest paths. These algorithms are heuristic because they perform well only on some graph classes. While their performance has been good in experimental studies, no theoretical bounds are known to support the experimental observations. Most of these algorithms have been motivated by finding paths in large road networks.
We start by reviewing the classical Dijkstra's algorithm and its bidirectional variant, developed in 1950's and 1960's. Then we review A* search, an AI technique developed in 1970's. Next we turn our attention to modern results which are based on preprocessing the graph. To be practical, preprocessing needs to be reasonably fast and not use too much space. We discuss landmark- and reach-based algorithms as well as their combination.

1 Introduction

We study the classical *point-to-point shortest path problem* (P2P): given a directed graph $G = (V, A)$ with non-negative arc lengths and two vertices, the source s and the destination t, find a shortest path from s to t. We are interested in exact shortest paths. We allow preprocessing, but limit the size of the precomputed data to a constant times the input graph size. Preprocessing time is limited by practical considerations. For example, for computing driving directions on large road networks, quadratic-time preprocessing is impractical.

Note that preprocessing-based algorithms have two parts: *preprocessing algorithm* and *query algorithm*. The former may take longer and run on a bigger machine or a cluster of machines. The latter need to be fast and may run on a small device.

Finding shortest paths is a fundamental and widely studied problem. The single-source problem with non-negative arc lengths has been studied most extensively; see e.g. [1, 3–5, 9–12, 17, 22, 28, 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. The P2P problem with no preprocessing has been addressed, for example, in [21, 31, 36, 42]. With preprocessing, no nontrivial theoretical bound is known for the general P2P problem, but there are non-trivial results for the special case of undirected planar graphs with slightly super-linear preprocessing space. The best bound in this context appears in [8]. Algorithms for approximate shortest paths that use preprocessing have been studied; see e.g. [2, 23, 39].

Some of the work on exact algorithms with preprocessing includes [14, 16, 15, 18, 19, 24, 27, 29, 32–35, 40]. Here we address only the approaches based on A^* search with landmark-based lower bounds [14, 18], the notion of reach [19], and their combination [16, 15]. For a related highway hierarchy approach, see [32, 33].

The paper is organized as follows. We give definitions and background in Section 2. Section 3 reviews A^* search and describes its landmark-based implementation, ALT. Section 4 introduces the concept of reach and discusses how one can use it to speed up shortest path computations. The section also discusses the corresponding preprocessing and query algorithms. In Section 5 we discuss how ALT can be combined with reaches. Section 6 gives an example of experimental performance of these algorithms. Concluding remarks appear in Section 7.

2 Preliminaries

The input to the preprocessing stage of the P2P algorithm is a directed graph $G = (V, A)$ with n vertices and m arcs, and non-negative length function ℓ on arcs. The input to each query is a source s and a destination t. The goal is to find a 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.

The labeling method for the shortest path problem [25, 26] finds shortest paths from the source to all vertices in the graph. The method works as follows (see e.g. [37]). For every vertex v it maintains a distance label $d(v)$, a parent $p(v)$, and a 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, *scans* all arcs out of v, and sets $S(v) = \texttt{scanned}$. To scan 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 non-negative, the graph has no negative cycles and the labeling method terminates with correct shortest path distances and a shortest path tree defined by the parent pointers. Its efficiency depends on the rule to choose a labeled vertex to scan next. Define $d(v)$ to be *exact* if it is equal to the distance from s to v. 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 [5] and independently Dantzig [3] observed that if ℓ is non-negative and v is a labeled vertex with the smallest distance label, then $d(v)$ is exact. The labeling method with the minimum label selection rule is known as *Dijkstra's algorithm*. If ℓ is

non-negative 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 destination t, $d(t)$ is exact and the s-t path defined by the parent pointers is a shortest path. We can terminate the algorithm at this point. This termination rule gives Dijkstra's algorithm for the P2P problem. This algorithm searches a ball with s in the center and t on the boundary.

One can also run Dijkstra's algorithm from t on the *reverse graph*, the graph with every arc reversed and obtain a shortest path as the reversal of the path found. One can combine the forward and the reverse algorithms. The *bidirectional algorithm* [3, 7, 30] alternates between running the two algorithms, each maintaining its own set of distance labels. Let $d_s(v)$ and $d_t(v)$ be the distance labels of v maintained by the forward and the reverse algorithms, respectively. (We will still use $d(v)$ when the direction would not matter or is clear from the context.) During initialization, the forward search scans s and the reverse search scans t. The algorithm also maintains the length of the shortest path seen so far, μ, and the corresponding path. Initially, $\mu = \infty$. When an arc (v, w) is scanned by the forward search and w has already been scanned by the reverse search, we know the shortest s-v and w-t paths have lengths $d_s(v)$ and $d_t(w)$, respectively. If $\mu > d_s(v) + \ell(v, w) + d_t(w)$, we have found a path shorter than μ, so we update μ and its path accordingly. We perform similar updates during the reverse search. When the search in one direction selects a vertex x already scanned in the other direction, the algorithm terminates. Note that x does not need to be on a shortest path from s to t. The bidirectional algorithm searches two touching balls centered at s and t.

A better stopping criterion (see [18]) is as follows:

> 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.

Note that any alternation strategy works correctly. Balancing the work of the forward and reverse searches is a strategy guaranteed to be within factor of two of the optimal off-line strategy.

A *potential function* is a real-valued function on vertices. Given a potential function π, the *reduced cost* of an arc is defined 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 every x-y path changes by the same amount, $\pi(y) - \pi(x)$. Thus the relative x-y path lengths are preserved, and the two problems are equivalent.

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

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

The first lemma, which is a special case of linear-programming duality, implies that we can think of $\pi(v)$ as a lower bound on the distance from v to t. The second lemma allows us to combine feasible lower bound functions to obtain a better one.

3 A^* Search

The A^* *search* method [6, 20] was originally designed to speed up search in large, sometimes implicitly represented, graphs, such as game graphs. This algorithm is also known as heuristic search or goal-directed search. The idea is that, instead of searching a ball around s, one biases the search towards t. The algorithm uses a (perhaps domain-specific) function $\pi_t : V \to R$ such that $\pi_t(v)$ gives an estimate on the distance from v to t. Define a (forward search) *key* of v $k_s(v) = d_s(v) + \pi_t(v)$. The only difference between A^* search and Dijkstra's algorithm is that at each step the former selects a labeled vertex v with the smallest key to scan next instead of the one with the smallest d_s value. 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 non-negative, so the algorithm is correct.

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_t 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. The following theorem makes this more precise. Consider an instance of the P2P problem and let π_t and π'_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., π'_t dominates π_t). If ties are broken consistently when selecting the next vertex to scan, the following holds.

Theorem 1. [13] *The set of vertices scanned by A^* search using π'_t is contained in the set of vertices scanned by A^* search using π_t.*

3.1 Bidirectional A^* search

We combine A^* search and bidirectional search as follows. Let π_t be the potential function used in the forward search and let π_s be the one used in the reverse search. Define reverse search key by $k_t(v) = d_t(v) + \pi_s(v)$.

Each original arc (v, w) appears in the reverse arc 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 $\pi_t + \pi_s = \text{const}$.

If π_t and π_s 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. In this paper we consider the former approach. For the latter approach, see e.g. [14].

Ikeda et al. [21] suggest using $p_t(v) = \frac{\pi_t(v) - \pi_s(v)}{2}$ for the forward computation and $p_s(v) = \frac{\pi_s(v) - \pi_t(v)}{2} = -p_t(v)$ for the reverse one. Although p_t and p_s do not give lower bounds as good as the original ones, they are feasible and consistent. To make the algorithm more intuitive, we add $\pi_t(t)/2$ to the forward function (thus making $p_s(t) = 0$) and $\pi_t(s)/2$ to the reverse function (making it zero at s). Note that adding a constant to a potential function does not change reduced costs.

3.2 ALT Algorithms

Introduced in [14], the ALT family of algorithms uses landmarks and triangle inequality to compute feasible lower bounds. We select a small subset of vertices as *landmarks*. For each vertex in the graph, precompute distances to and from every landmark. This information can be used to compute lower bounds as follows. 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. Note that triangle inequality is the property of graph distances, and the ALT method applies even if graph arc lengths do not satisfy the inequality.

The preprocessing algorithm selects the landmarks and computes distances to and from the landmarks. Finding good landmarks is important for the performance of ALT queries. The simplest way of selecting landmarks is to pick them at random. One can do better, however, using heuristics and optimization techniques. Several heuristics have been suggested in [14, 18]. We discuss a fast heuristic and one of the best in terms of landmark quality next.

The *farthest* selection heuristic picks a vertex r and selects a vertex furthest from r as the first landmark. Given a set of current landmarks, the vertex furthest away form this set is selected to be the next landmark. We repeat this process until the desired number of landmarks is reached.

The *avoid* heuristic is more sophisticated. Let S be a set of already selected landmarks (initially an empty set). We choose the next landmark as follows. First, pick a vertex r and compute a shortest path tree T_r rooted at 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. For every vertex v, now compute its *size* $s(v)$ as follows. If the subtree T_v rooted at 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 in a greedy way: Start from w and always go to the child with the largest size, until a leaf is reached. Make this leaf a new landmark. A natural way of picking r (the root vertex) is uniformly at random. Better results are obtained by picking with higher probability vertices that are far from the existing landmarks; see [18].

One can use optimization techniques to generate better landmarks. For example, one can use a heuristic to generate a set of candidate landmarks and then use local search to select a good subset to use as landmarks [18]. On road networks, this significantly increases preprocessing time while only modestly increasing landmark quality.

Now we discuss optimizations of the query algorithm.

Increasing the number of landmarks makes the algorithm more robust and tends to decrease the search space, but it has two disadvantages. First, it increases memory requirements. Second, it increases the overhead of computing the lower bounds. To address the second shortcoming, note that for a particular s-t computation, some landmarks are better than others. *Static landmark selection* [13] uses only a subset of the available landmarks, those that give the highest lower bounds on the s-t distance. With appropriate subset size, this has little effect on the search space and improves the running time.

Dynamic landmark selection has been proposed in [18]. The idea is to dynamically maintain a set of active landmarks to be used for computing lower bounds. Experimental results show that this selection is very effective. The average number of landmarks active during a query is less than 3, and the search space is comparable with that for using all landmarks. In fact, the search space is sometimes smaller because bad landmarks can misdirect the search.

Dynamic selection works as follows. Two landmarks are selected initially, one that gives the best bound using distances to landmarks and the other using distances from landmarks. Let b be the best bound computed during this initial landmark selection. When enough work has been done since the last update to amortize a re-evaluation of the current set of landmarks, we consider making more landmarks active. 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 value that depends on b and the update attempt number i. (In [18], the value of $b(10-i)/10$ is used.) At this point, the algorithm checks if the best lower bound on the distance from v to the destination (using all landmarks) is at least a constant factor (e.g. 1.01) better than the current lower bound (using only active landmarks). If so, the landmark yielding the improved bound is activated.

A landmark activation changes reduced costs and makes it necessary to restart the algorithm because labels of vertices in the priority queue have changed. The priority queue is reset and the vertices are inserted into it with the new keys. Although the restarting is natural, its proof of correctness is long and technical. See [18] for details.

Other, more minor, improvements and optimizations of ALT have been discussed in [16, 18].

4 Reach-Based Pruning

In this section we discuss the notion of *reach*, originally proposed by Gutman [19] and further developed in [16, 18]. The definition of reach may seem odd at first,

but becomes natural when one sees how it is used to prune Dijkstra's search. To simplify the presentation, we assume that shortest paths between any pair of vertices are unique. See the above references for the way to deal with ties.

Given a path P from s to t and a vertex v on P, the *reach of v with respect to P* is the minimum of the length of the subpath from s to v and the length of the subpath from v to t. The *reach* of v, $r(v)$, is the maximum, over all **shortest** paths P through v, of the reach of v with respect to P.

In the context of reach-based algorithms, we also use the following definitions. We denote an upper bound on $r(v)$ by $\overline{r}(v)$ and denote a lower bound on the distance from v to w by $\underline{\text{dist}}(v, w)$.

Reach can be used to prune Dijkstra's search:

> Suppose $\overline{r}(v) < \underline{\text{dist}}(s, v)$ and $\overline{r}(v) < \underline{\text{dist}}(v, t)$. Then v is not on a shortest path from s to t, and therefore Dijkstra's algorithm does not need to label or scan v.

Note that this also holds for the bidirectional algorithm.

A straightforward way to compute the reaches of all vertices is to compute all shortest paths and apply the definition. A more efficient algorithm that adds $O(n^2)$ overhead to the cost of constructing shortest path trees for every vertex is as follows. Initialize $r(v) = 0$ for all vertices v. For each vertex x, grow a complete shortest path tree T_x rooted at x. For every vertex v, determine its reach $r_x(v)$ within the tree, given by the minimum between its *depth* (the distance from the root) and its *height* (the distance to its farthest descendant). If $r_x(v) > r(v)$, update $r(v)$.

Implicit in this algorithm is an efficiently verifiable "certificate" that proves a lower bound on reach, i.e., proves that $r(v) > L$. The certificate is a shortest path tree such that with respect to this tree $r(v) > L$. If L is in fact a lower bound on $r(v)$, such a certificate always exists, and we can verify the validity of a certificate in linear time (check if the tree is a shortest path tree, then compute v's depth and height in the tree). For vertices with large reaches, one can find a good lower bound on their reach by growing a moderate number of shortest path trees from random roots and using the resulting values. Unfortunately, there does not seem to be a similar certificate for upper bounds on reaches.

Upper bounds, however, can be used for pruning if the exact reach values are not available. On some classes of graphs, such as road networks, there are heuristics for efficiently computing reach upper bounds when the exact computation takes too long. We discuss this after we give details on how to use the upper bounds in queries.

4.1 Queries Using Upper Bounds on Reaches

As described in Section 4, to prune the search based on the reach of some vertex v, we need a lower bound on the distance of v to the source and a lower bound on the distance of v to the sink. Sometimes the lower bounds are available: For example, if the graph is embedded in a metric space or if landmarks and

landmark distances are available. The bidirectional Dijkstra's algorithm, on the other hand, does not require any additional information to prune by reach. Next we describe two ways in which it can be done.

The first one, the *bidirectional bound* algorithm, works as follows. During the bidirectional search, consider the search in the forward direction, and let γ be the smallest distance label of a labeled vertex in the reverse direction (i.e., the top element of the reverse heap). If a vertex v has not been scanned in the reverse direction, then γ is a lower bound on the distance from v to t. (The same applies to the reverse search.) When we are about to scan v we know that $d_s(v)$ is the distance from the source to v. So we can prune the search at v if v has not been scanned in the reverse direction, $\bar{r}(v) < d_s(v)$, and $\bar{r}(v) < \gamma$. The stopping condition is the same as for the standard bidirectional algorithm without pruning.

An alternative, the *self-bounding algorithm*, uses the distance label itself for pruning. Assume we are about to scan a vertex v in the forward direction (the procedure in the reverse direction is similar). If $\bar{r}(v) < d_s(v)$, we prune the vertex. Note that if the distance from v to t is at most $\bar{r}(v)$, the vertex will still be scanned in the reverse direction, given the appropriate stopping condition. It is easy to see that the following condition works correctly:

> Stop the search in a given direction when either there are no labeled vertices or the minimum distance label of labeled vertices for the corresponding search is at least half the length of the shortest path seen so far.

The reason why this algorithm can safely ignore the lower bound to the destination is that it leaves to the other search to visit vertices that are closer to it. Note, however, that when scanning an arc (v, w), even if we end up pruning w, we must check if w had been scanned in the opposite direction and, if so, check if the candidate path using (v, w) is the shortest path seen so far.

Remark. The self-bounding method has an interesting property. If a search in a certain direction prunes a vertex v, all vertices scanned by the search afterwards have reaches greater than v. This defines a "continuous hierarchy" of reaches: Once the search leaves a reach level, it never comes back to it.

The following natural algorithm falls into both of the above categories. We call this algorithm *distance-balanced*. It balances the radius of the forward and reverse search regions by picking the labeled vertex with minimum distance label, considering both directions. Note that the distance label of this vertex is also a lower bound on the distance to the target, as the search in the opposite direction has not selected the vertex yet. The algorithm can be implemented with only one priority queue.

Note that both bidirectional bound and self-bounding algorithms can use explicit lower bounds, if available, with the implicit bounds. This may result in more pruning.

4.2 Reach Upper Bounds

The preprocessing algorithm computes upper bounds on vertex reaches and also adds arcs to accelerate both preprocessing and queries. We only briefly mention the main ideas behind the algorithm. For details, see [16, 18, 19, 32, 33].

Instead of building shortest path trees as the exact algorithm, we use *partial trees* [19]. The idea is to bound reaches of vertices with reach less then a threshold ϵ. When building shortest path trees, the algorithm stops early and uses the resulting partial trees. For example, if all arc lengths are less then ϵ and we stop building trees when their radius exceeds 3ϵ, we are guaranteed to compute exact reaches of all vertices with reach below ϵ. In general, we want to use ϵ that is much smaller than the maximum arc length, and the partial tree algorithm gets more complicated. In particular, for efficiency reasons it computes upper bounds on reaches and may fail to bound reach of some vertices with reach below ϵ. We omit details.

Once we bound reaches of some vertices, we delete these vertices from the graph, replacing them by *penalties* (defined in the next paragraph). We proceed by recursively bounding reaches of the remaining vertices. In addition, at each iteration we add *shortcut arcs* [32] to reduce the reach of some vertices. This speeds up both the preprocessing (the graph shrinks faster) and the queries (more vertices are pruned). We discuss the shortcuts later in this section.

We define *penalties* as follows. Let $G_i = (V_i, A_i)$ be the subgraph processed by the partial-trees algorithm at iteration i. We define the *in-penalty* of a vertex $v \in V_i$ as

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

if v has at least one eliminated incoming arc, and zero otherwise. The *out-penalty* of v is defined similarly, considering outgoing arcs instead of incoming arcs:

$$\textit{out-penalty}(v) = \max_{A \setminus A_i} \{\bar{r}(w) + \ell(v, w)\}.$$

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

Penalty at a vertex v can be interpreted as follows. Let v' and v'' be new vertices. Add arcs (v', v) and (v, v'') with arcs of length equal to the in- and out-penalty of v, respectively. The arcs model paths deleted from the original graph in a conservative way, where intersecting paths may be modeled as non-intersecting paths. Thus the reach of v in the transformed graph will be at least as high as in the original graph.

Next we discuss shortcuts. Sanders and Schultes suggest shortcutting small degree vertices To *shortcut* v, we examine all pairs of arcs $((u, v), (v, w))$ with $u \neq w$. For each pair, if the arc (u, w) is not in the graph, we add an arc (u, w) of length $\ell(u, v) + \ell(v, w)$. Otherwise, we set $\ell(u, w) = \min(\ell(u, w), \ell(u, v) + \ell(v, w))$. We delete v and all arcs adjacent to it. Finally, we adjust penalties of the neighbors of v. See [15] for details. Note that if we shortcut vertices with degree bounded by a constant, the number of arcs added is linear in n.

Shortcuts speed up preprocessing because they allow us to delete more vertices at each iteration. Shortcuts also speed up queries. If one break ties on path

lengths by the number of arcs in the path, then a path that includes a shortcut arc will be favored over the corresponding (same-length) path that uses original arcs. This reduces reach of the bypassed vertex and allows it to be pruned: the search will use the shortcut arc instead of going through the vertex.

We are now ready to give a high-level description of the algorithm. It uses two subroutines: one adds shortcuts to the graph (*shortcut step*), and the other runs the partial tree algorithm and eliminates low-reach vertices (*partial-tree step*). Each iteration $i = 0, 1, \ldots$ has a threshold ϵ_i (an increasing function of i). By the end of the i-th iteration, the algorithm will have eliminated every vertex whose reach it can prove is less than ϵ_i. Each iteration applies a shortcut step followed by a partial-tree step. If there are still vertices left in the graph after iteration i, we set $\epsilon_{i+1} = \alpha\,\epsilon_i$ (for some $\alpha > 1$) and proceed to the next iteration. Our implementation uses $\alpha = 3$.

Note that the high-reach vertices appear on more shortest paths than the low-reach ones, and it is more important to have tight bounds on the reaches of these vertices. After reaches of all vertices have been bounded, one can delete low-reach vertices, replaces them by penalties, and run the exact reach algorithm on the remaining graph. The threshold between low- and high-reach vertices is chosen to make the exact computation not too expensive.

Below we refer to the reach-based algorithm with shortcuts and reach upper bounds computed using partial trees as RE.

5 Reach for A*

Reach-based pruning can be easily combined with A^* search, as originally observed by Gutman [19]. The general approach is to run A^* search and prune vertices based on reach conditions. Specifically, when A^* search is about to scan a vertex v we extract the length of the shortest path from the source to v from the key of v. Furthermore, $\pi_t(v)$ is a lower bound on the distance from v to the destination. If the reach of v is smaller than both $d_s(v)$ and $\pi_t(v)$, we prune the search at v.

The reason why reach-based pruning works is that, although A^* search uses transformed lengths, the shortest paths remain invariant. This applies to bidirectional search as well. In this case, we use $d_s(v)$ and $\pi_t(v)$ to prune in the forward direction, and $d_t(v)$ and $\pi_s(v)$ to prune in the reverse direction. Using pruning by reach does not affect the stopping condition of the algorithm. We still use the usual condition for A^* search, which is similar to that of the standard bidirectional Dijkstra, but with respect to reduced costs (see [18]). We call our implementation of the bidirectional A^* search algorithm with landmarks and reach-based pruning REAL.

However, one cannot use implicit bounds with A^* search. The implicit bound based on the radius of the ball searched in the opposite direction does not apply because the ball is in the transformed space. The self-bounding algorithm cannot be combined with A^* search in a useful way, because it assumes that the two searches will process balls of radius equal to half of the s-t distance. However,

processing these balls defeats the purpose of A^* search, which aims at processing a smaller set.

The main gain in the performance of A^* search comes from the fact that it directs the two searches towards their goals, reducing the search space. Reach-based pruning sparsifies search regions, and this sparsification is effective for regions searched by both Dijkstra's algorithm and A^* search.

Note that REAL has two preprocessing algorithms: the one used by RE (which computes shortcuts and reach bounds) and the one used by ALT (which chooses landmarks and computes landmark distances). These two procedures are independent from each other: since shortcuts do not change distances, landmarks can be generated regardless of what shortcuts are added. Furthermore, the query is still independent of the preprocessing algorithm: the query only takes as input the graph with shortcuts, the reach values, and the distances to and from landmarks.

A combination of ALT and RE allows additional optimizations, in particular *reach-aware landmarks* [15]. The idea is to keep landmark distances only for a fraction (e.g., 1/16) of high-reach vertices. This saves space while not increasing query time much. Furthermore, one can increase the number of landmarks and sometimes win both in space and query time. For example, for large road networks, going from 16 landmarks for all vertices to 64 landmarks for 1/16 high-reach vertices has this effect. See [15].

6 Experimental Results

To illustrate practical implications of the above techniques, we use two graphs of roughly comparable size but with different structure. One is a 400×400 planar grid with adjacent vertices connected by arcs with lengths chosen uniformly and independently from the interval $[1, 16\,000]$. The grid is directed: length of arcs (v, w) and (w, v) are chosen independently. It has 160 000 vertices and 638 400 arcs. The other graph is a road network of the San Francisco Bay area. This graph has 330 024 vertices and 793 681 arcs.

The algorithms we compare is B, the bidirectional Dijkstra's code, ALT, an implementation of the landmark-based A^* search (using 16 "avoid" landmarks), RE, an implementation of the reach-based method using shortcuts and landmark upper bounds, and REAL, which combines ALT and RE techniques. We also consider two variants of RE, one without shortcuts, and another without shortcuts using exact reach values. The latter algorithm uses an optimized version of the exact reach algorithm [15].

Algorithms were run on a Toshiba Tecra 5 laptop with 2GB of RAM and dual-core 2GHz processor (but the programs are single-threaded). Query data is for 10 000 random vertex pairs. We give the average running time, the average number of scans, and the maximum number of scans. These examples gives some idea of possible speed-ups and memory overheads.

Table 1 gives running times and operation counts for the grid graph. First consider ALT and RE. These algorithms have similar average running times which

method	preprocessing seconds	preprocessing MB	query avgscan	query maxscan	query ms
B	—	5.7	52 514	128 399	41.0
ALT	6.3	30.1	1 915	31 159	3.1
RE	361.6	8.8	3 360	5 502	3.3
REAL	367.9	30.8	326	2 361	0.7
RE (no shortcuts)	7 658.2	6.4	28 691	56 194	23.7
RE (no shortcuts, exact reaches)	14 117.5	6.4	21 326	39 248	17.2

Table 1. Data for a random grid.

are an order of magnitude better than that of B. There are differences, however. The former algorithm scans fewer vertices, but scans are more expensive. RE is more robust: The worst case differs little from the average. Preprocessing for ALT is much faster but it takes more space to store the results.

REAL has combined overhead of ALT and RE for preprocessing time and memory, but leads to significantly faster query times. The queries are almost two orders of magnitude faster than those for B.

method	preprocessing seconds	preprocessing MB	query avgscan	query maxscan	query ms
B	—	6.1	116 588	293 152	30.49
ALT	5.7	27.8	4 430	54 194	2.91
RE	45.4	12.3	668	1 697	0.55
REAL	51.1	34.0	172	982	0.28
RE (no shortcuts)	753.3	7.4	13 419	28 670	5.24
RE (no shortcuts, exact reaches)	8 629.9	7.4	11 140	24 565	4.41

Table 2. Data for the Bay Area road network.

The last two rows of the table show the benefit of shortcuts and that of exact reaches. Shortcuts significantly improve performance of both preprocessing and queries. Exact reaches are more expensive to compute, and they improve queries only modestly. In large graphs, current algorithms can compute only approximate reaches with shortcuts.

Table 2 gives data for the road network. Performance of B and ALT is similar to that on the grid graph; the latter algorithm is significantly faster than the former. In contrast, both preprocessing and queries for RE are significantly faster on the road network, which has a better defined highway hierarchy (i.e., there is a small number of vertices with relatively large reach values). As RE is already very efficient, the relative improvement of REAL over it is smaller than for the grid graph case, but still non-trivial.

The next row shows that, as in the grid graph case, the shortcuts significantly speed up preprocessing and queries. The last row shows that computing exact reaches is much more expensive than computing the upper bounds, while the query speed-up is modest. Note that the difference between exact and approximate reach computation times is much greater than for grid graphs.

7 Concluding Remarks

Many P2P shortest path algorithms with preprocessing have been developed recently. These algorithms are very efficient in practice on road networks and some other kinds of graphs. Many open questions remain, however, in particular theoretical questions.

One would like to have a theoretical justification for these algorithms. Two possible directions are proving good worst-case bounds for the algorithms on special graph types or proving average-case bounds on graph distributions. For the latter, random grids like the one used in Section 6 are interesting candidates.

Another set of questions has to do with computing reaches. One can modify a standard all-pairs shortest path algorithm to compute reaches in the same time bound, which is $O^*(n^2)$ for sparse graphs. Since the size of the output for the all-pairs problem is $\Omega(n^2)$, there is limited room for improvement. As reaches need only one value per vertex, this argument does not apply to the problem of computing reaches. An interesting open question is the existence of an algorithm that computes reaches – or provably good upper bounds on reaches – in $o(n^2)$ time.

Acknowledgment

I would like to thank Renato Werneck for his comments on this paper.

References

1. B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.
2. L. J. Cowen and C. G. Wagner. Compact Roundtrip Routing in Directed Networks. In *Proc. Symp. on Principles of Distributed Computation*, pages 51–59, 2000.
3. G. B. Dantzig. *Linear Programming and Extensions*. Princeton Univ. Press, Princeton, NJ, 1962.
4. E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.
5. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.
6. J. Doran. An Approach to Automatic Problem-Solving. *Machine Intelligence*, 1:105–127, 1967.
7. D. Dreyfus. An Appraisal of Some Shortest Path Algorithms. Technical Report RM-5433, Rand Corporation, Santa Monica, CA, 1967.

8. J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. 42nd IEEE Annual Symposium on Foundations of Computer Science*, pages 232–241, 2001.
9. M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.
10. G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.
11. A. V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. In *Proc. 9th ESA, Lecture Notes in Computer Science LNCS 2161*, pages 230–241. Springer-Verlag, 2001.
12. A. V. Goldberg. Shortest Path Algorithms: Engineering Aspects. In *Proc. ESAAC '01, Lecture Notes in Computer Science*. Springer-Verlag, 2001.
13. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2004.
14. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.
15. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better Landmarks within Reach. In *The 9th DIMACS Implementation Challenge: Shortest Paths*, 2006.
16. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proc. 7th International Workshop on Algorithm Engineering and Experiments*. SIAM, 2006.
17. 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.
18. A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proc. 7th International Workshop on Algorithm Engineering and Experiments*, pages 26–40. SIAM, 2005.
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, 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.
22. R. Jacob, M.V. 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. Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of shortest path and constrained shortest path computation. In *WEA*, pages 126–138, 2005.
25. Jr. L. R. Ford. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956.
26. Jr. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

27. 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.
28. 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.
29. Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *WEA*, pages 189–202, 2005.
30. T. A. J. Nicholson. Finding the Shortest Route Between Two Points in a Network. *Computer J.*, 9:275–280, 1966.
31. I. Pohl. Bi-directional Search. In *Machine Intelligence*, volume 6, pages 124–140. Edinburgh Univ. Press, Edinburgh, 1971.
32. P. Sanders and D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. In *Proc. 13th Annual European Symposium Algorithms*, 2005.
33. P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proc. 14th Annual European Symposium Algorithms*, 2006.
34. D. Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. Master's thesis, Department of Computer Science, Universitt des Saarlandes, Germany, 2005.
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*, pages 43–59. LNCS, Springer, 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 *European Symposium on Algorithms*, 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.