Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks

Ron Gutman*

January 6, 2004

Abstract

Past work has explored two strategies for high volume shortest path searching on large graphs such as road networks. One strategy, extensively researched by the academic community, pre-computes paths and avoids a too expensive all-pairs computation by computing and storing only enough paths that a path for an arbitrary origin and destination can be formed by joining a small number of the pre-computed paths. This approach, in practice, has been unwieldy for large graphs - both preprocessing time and the size of the resulting database are excessively burdensome. The implementations can be unusually complex. The other strategy, often used in industry for routes on road networks, exploits the natural hierarchy in a road network to prune a Dijkstra search. This is much less unwieldy, but less reliable as well. The pruning is based on a heuristic in such a way that no guarantee about the optimality of the computed path can be made. In the worst cases, the algorithm will fail to find a path even though one exists. Both of these strategies are inflexible or inefficient in the face of complex requirements such as changes to the network or queries involving multiple destinations or origins.

We introduce a new concept called reach which allows shortest path computation speed on par with the industry approach but computes provably optimum paths as do the theoretical approaches. It is more versatile than both; for example, it easily handles multiple origins and destinations. The versatility makes available a wide range of strategies for dealing with complex routing problems. In a test on a graph of 400,000 vertices, the new algorithm computed paths between randomly chosen origins and destinations 10 times faster than Dijkstra. It also combines naturally with the A* algorithm for an additional reduction in path query processing time.

*WaveMarket and Yahoo, e-mail: gutman@sbcglobal.net

1 Introduction

Some form of hierarchy is inevitably harnessed to improve the speed of shortest path computations on large graphs. Many of the well-researched strategies impose a hierarchy on the graph, usually by partitioning the graph (see [7, 13, 14, 15]). The partitioning permits a trade-off between pre-computation and query time processing. Results stored from the preprocessing represent paths from arbitrary vertices in the graph to vertices at the partition boundaries and paths between vertices on the partitions. Queries are serviced by joining stored paths (usually 2 or 3 paths). Some variations of this approach assume planarity of the graph or that the graph is undirected (e.g., [7, 14, 15]). Road networks cannot be reconciled with either assumption.

It is common in industry to rely on the natural hierarchy in road networks to improve query speed. This approach involves no preprocessing. Typically, roads are classified according to their importance for longer routes, e.g., freeways are very important and residential streets are unimportant. Either vertices or edges in the graph representation may carry the importance attribute. A modified Dijkstra algorithm disregards vertices or edges of low importance when they are far from both origin and destination. A heuristic rule determines whether a vertex or edge can be ignored. Effective implementations require a "bidirectional" search, that is two Dijkstra algorithms running in tandem, one searching from the origin toward the destination and the other searching from the destination in reverse, that is, on a graph in which every edge (u, v) has been replaced by (v, u). The heuristic does not guarantee a shortest path, and reliably good results require tuning. Empirically, the performance is sub-linear as a function of the "distance" between the origin and destination.

Both approaches suffer from inflexibility. Neither can efficiently address problems that involve multiple destinations and origins; each possible pair of origin and destination must be handled as a separate problem. Neither is easily adapted to a dynam-

ically changing graph. For example, if a traffic jam increases the weight on the edges representing the involved roads, a frontage road might become very important. How can that change in importance be discovered?

Our approach was inspired by the latter question. Instead of relying on road classifications, we define a formal attribute of a vertex that reflects the importance of the vertex and can be computed from the graph. The attribute, which we call "reach", makes possible a new variation on the Dijkstra algorithm ([4, 5]) that preserves the optimality of the result while improving computation time significantly. We show that the modified algorithm preserves optimality. We also offer efficient algorithms for computing the attribute.

Our approach offers these advantages:

- Guarantees optimality of computed paths.
- The shortest path computation time is comparable to that of the industry approach.
- Can be combined with other optimizations such as the A* algorithm (see [10, 11]).
- Storage requirements are not significantly increased by the pre-computed data.
- Preprocessing may be fast enough to handle dynamically changing graphs in some applications (e.g, a metropolitan road network on a parallel machine with 10-20 CPUs).
- Greatly reduces computation time of shortest paths for multiple origins or destinations.

We feel that the last of these is the most important because of the importance of multiple origin and destinations in fields such as transporation logistics and the lack of a satisfactory alternative for road networks.

The computational effectiveness of our approach depends on properties of the graph. We have not formalized those properties, so we provide empirical results on 3 different data sets instead of theoretical time bounds. Loosely, our approach depends on the presence of a natural hierarchy in the network. We believe that most large networks used for transport will possess some hierarchy due to the motivations of designers to optimize the network for transport.

Section 2 of this paper introduces the concept of reach and gives its formal definition and explains notation and terminology used in this paper. Section 3 presents our shortest path algorithm.

Because the computation of reach for each vertex potentially requires an all-pairs shortest path computation, we present an alternative in sections 4 and 5. The alternative method computes an upper bound on the reach of each vertex at much less cost than an all-pairs computation. An upper bound can be used in our shortest path algorithms without affecting correctness. Section 4 gives an intuitive description of the approach while section 5 presents theorems that lead to the actual algorithm for computing upper bounds on reach. Though we have proofs for all of the theorems, space did not permit their inclusion. Section 6 describes the algorithm for computing the upper bounds and its implementation.

Section 7 presents experimental results that characterize the performance of our reach bound computation and shortest path algorithms.

2 The Concept of Reach

Intuitively, the reach of a vertex encodes the lengths of shortest paths on which it lies. To have a high value of reach, a vertex must lie on a shortest path that extends a long distance in both directions from the vertex.

The notion of "length" or "distance" here is not necessarily the same as the notion of "weight" or "cost", that is, the length of a path is not necessarily the same as the weight, or cost, of the path. In a road network, for example, travel time is commonly used as the cost metric and the weights on the edges reflect travel time not travel distance. The definition of reach will also depend on some metric. The reach metric might be the same as the cost metric. However, our experimental results showed that, on road networks, using travel distance as the reach metric more effectively captured the hierarchy in the network. In addition, a metric based on geometric distance permits a more straight forward implementation of the shortest path algorithm and one which can be readily adapted to a variety of problems and strategies for solving them.

For this reason, we use terminology and notation that distinguishes the weight function of a graph from a reach metric for the graph. The definition of a reach metric is identical to the definition of a weight function: a reach metric for graph $G = (V, E)$ is a function $m : E \longrightarrow R$ mapping any edge, e, in G to a real number, $m(e)$. For a path P, we use the notation $m(P)$ to represent the sum of $m(e)$ over all edges e of P or zero if P has only 1 vertex. The notation, $m(u, v, P)$, represents $m(Q)$ where Q is the subpath of P from u to v. (In this paper, the term "path" always means "simple path").

It's possible for the reach metric to be the same

as the weight function, or cost metric. However, we found that a reach metric based on distance is more effective and has other advantages, so the shortest path algorithm we present uses a reach metric based on distance. We assume that the graph is "projected" into a plane. We use the word "projected" to distinguish this notion from the notion of a planar graph. None of our work assumes planarity of the graph. We only assume that each vertex has been assigned coordinates in a some Euclidean space without requiring that edges do not intersect. We do assume that the assignment of coordinates is consistent with the reach metric in this way:

> Given an edge, (u, v), $m(u, v) \geq d(u, v)$ where d gives the Euclidean distance between two vertices.

The projection into a Euclidean space is not strictly needed as long as there is a distance function consistent with the reach metric, but most networks with a distance function are projected into some space. Note that given a path P from u to v, $m(P) = m(u, v, P) \geq d(u, v)$

Because the term "shortest path" too easily brings length or distance to mind, we, instead, use the term "least-cost path" for the remainder of the paper except for the "Experimental Results" section (as much for our own sanity as the reader's). A "least-cost path tree" is the same as a "shortest path tree".

Definition: Given,

- a directed graph $G = (V, E)$ with positive weights
- a non-negative reach metric $m : E \longrightarrow R$
- a path P in G starting at vertex s and ending at vertex t
- a vertex v on path P

then the **reach of v on P,** $r(v, P)$, is $\min\{m(s, v, P), m(v, t, P)\}$ and the **reach of v in G,** $r(v, G)$, is the maximum value of $r(v, Q)$ over all least-cost paths Q in G containing v.

3 A Reach-based Shortest Path Algorithm

The algorithm we present here assumes a reach metric that is consistent with the Euclidean distance function in the manner described above. Let G be a directed graph $G = (V, E)$ with positive weights, and reach metric, m consistent with distance function $d : (V \times V) \longrightarrow R$.

The algorithm is a modification of Dijkstra's algorithm in which a function, $test(v)$, is called immediately prior to inserting a vertex, v, into the priority queue. If $test$ returns true, the vertex is inserted into the priority queue; otherwise the vertex is not inserted into the priority queue. We describe the algorithm for $test(v)$ and prove that the modified Dijkstra algorithm finds the least cost path. Our tests show that the algorithm reduces the number of insertions into the priority queue by an order of magnitude or more on moderately long paths (more than 25km) in road networks.

The function $test(v)$ uses the following information:

- $r(v, G)$ as defined above
- $m(P)$ where P is the computed path from the origin, s, to v at the time v is to be inserted into the priority queue.
- $d(v, t)$ where t is the destination

The value of $test(v)$ is:

- true if $r(v, G) \geq m(P)$ or $r(v, G) \geq d(v, t)$
- false otherwise

In other words, the value returned by $test(v)$ is only false if the reach of v is too small for it to lie on a least-cost path a distance $m(P)$ from the origin and at a straight-line distance $d(v, t)$ from the destination.

The modified Dijkstra algorithm is equivalent to the unmodified Dijkstra algorithm performed on a graph, G', that results by removing the rejected nodes from G. To prove that the algorithm is correct, we only need to show that G' has the same least-cost path from s to t as G. It is clearly not possible for the reduced graph to have a lower cost path from s to t, so it is sufficient to show that a least-cost path, P, from s to t in G also exists in G'. Induction shows that P exists in G' if for every v on P, $test(v)$ is true. Assuming there is a v on P for which $test(v)$ is not true, leads to a contradiction we now show. Let v be the first vertex on P for which $test(v)$ is false. (Note that $test(s)$ is necessarily true). That v is on P implies that:

$$(1) \quad r(v, G) \geq r(v, P) = \min\{m(s, v, P), m(v, t, P)\}$$

However, if $test(v)$ is false, then both of the following are true:

$$(2) \quad r(v, G) < m(s, v, P)$$

$$(3) \quad r(v, G) < d(v, t) \leq m(v, t, P)$$

Taken together, (3) and (2) contradict (1). We conclude that for any v on P, $test(v)$ is true, that G' includes P, and that the modified algorithm is correct.

Note that, $test(v)$ could use, in place of $r(v, G)$,

some upper bound, b, on $r(v,G)$. In that case, a correctness proof is the same except that (2) and (3) must be justified for the modified $test(v)$. (2) and (3) would follow from these observations:

- $b < m(s,v,P)$ because $test(v)$ failed using b
- $b < d(v,t) \leq m(v,t,P)$ because $test(v)$ failed using b
- $r(v,G) \leq b$

Definition: We call b a **reach bound** for v.

If infinity is employed as the reach bound for all v, then the algorithm becomes the Dijkstra algorithm.

We note, without giving details, that this modification can be easily applied to the A* algorithm. For graphs projected into some space, the A* algorithm's estimation function for vertex v might be based on $d(v,t)$ depending on the cost metric (weight function) for the graph. That A* and $test(v)$ can share the cost of computing $d(v,t)$, which is not completely trivial, is an added benefit of combining the two techniques.

Finally consider a requirement to compute a least-cost path from an origin s to any vertex in a set T of vertices. The modified algorithm, with or without A*, readily performs this as long as there is a function $d(v,T)$ that computes the distance from a vertex v to the nearest vertex in T. For example, we have implemented this for T where T are the points on the network that intersect the boundary of a circle or rectangle (we temporarily add vertices on the edges where the intersections occur). Then the algorithm computes the shortest path to the circle or rectangle boundary.

The performance of our algorithm obviously depends on how many vertices are rejected by the test function. That, in turn, depends on distribution of the values of reach for the vertices in the graph. For road networks and other networks with a high degree of hierarchy, most vertices have low reach values (short reach) and only a few have high reach values (long reach). In fact, the distribution approximates an exponentially decreasing function of reach.

There is a more general algorithm that uses a bidirectional search and requires no Euclidean distance function, but that approach lacks several advantages compared to the one we present here and is more complex.

4 A Fast Algorithm for Computing Reach Bounds

Computing $r(v,G)$ for every v in G is expensive. The only method we know is to perform an all-pairs shortest path computation on G. But as noted at the end of section 3, an upper bound on $r(v,G)$ can be used in place of $r(v,G)$. We describe an algorithm to compute reach bounds that are close enough to the actual reach values that our reach-based least-cost path algorithm, using those reach bounds, yields the performance results stated in section 7.

Our strategy computes small reach bounds first, then uses that information to compute larger reach bounds. This bootstrapping from low to high reach bounds is repeated until reach bounds for most vertices have been computed. The remaining vertices are assigned infinite reach bounds which means that they are never rejected by the *test* function described in section 3.

For a network with a high degree of hierarchy, the first iteration computes reach bounds for most vertices. The computation is performed by computing what we call "partial least-cost path trees". A partial least-cost path tree is a tree which is a directed subgraph of a least-cost path tree and has the same root as the least-cost path tree. The partial least-cost path trees that are needed to compute small reach bounds are small trees so their computation time is much shorter than that of the complete least-cost path trees. The theory in the next section shows how to determine the extent of the required partial least-cost path trees.

When some reach bounds have been computed for some vertices, the next iteration is performed on a graph, G', with those vertices omitted. Because G' is smaller than G, partial least-cost path trees computed can be extended further on G', at a reasonable computing cost, than on G. This allows reach bounds to be computed for additional vertices. The bounds are computed using least-cost paths in G' and the previously computed reach bounds of vertices in $G - G'$ adjacent to those least-cost paths. The reach bounds of the adjacent vertices allow the algorithm to infer how far least-cost paths in G can extend from where they leave G'.

Iterations of this process with smaller and smaller G' continue until G' is small enough to assign its vertices infinite reach bounds without badly affecting the performance of path computation.

Implementation details appear in section 6.

5 Theory for Computing Reach Bounds

The theorems in this section answer three key questions about computation of reach bounds:

- How far should the least-cost path computations extend in order to compute reach bounds? (Theorems 5.3 to 5.6)

- Given that extent of computation, for which vertices can a reach bound be computed? (Theorems 5.5 and 5.6)
- For those vertices, what reach bounds can be determined? (Theorems 5.1 and 5.2)

Theorems 5.3 to 5.6 address the first question by characterizing the set of least-cost paths that must be computed. The length of these paths, and hence the length of the computations, are bounded by these theorems.

Definition: Given a directed graph G with positive weights and a path P in G, we say that "P is minimal with respect to the reach of v in G", if P is a least-cost path including v, $r(v,P) = r(v,G)$, and, for any proper subpath of P containing v, Q', $r(v,Q') < r(v,P)$.

Intuitively this means that removing an edge from either end of a path reduces the reach of v on the path. Obviously, for every vertex v in graph G, there exists a path P which is minimal with respect to the reach of v in G.

To illustrate the application of the theorems, suppose we wish to compute reach bounds for all vertices v in G for which $r(v,G) < b$. A special case of Theorem 5.6 tells us that it is sufficient to compute all least-cost paths, P, in G satisfying

$m(P) < 2b + \max\{m(f) + m(l)\}$
where f and l are, respectively, the first and last edges of P

Longer paths are not needed. For a given v in G, the maximum value of $r(v,P)$ for all such P is computed. If that maximum is less than b, then this special case of Theorem 5.6 further tells us that among those paths is one which is minimal with respect to the reach of v in G. Therefore that maximum value is $r(v,G)$.

This special case ($G = G'$) of Theorem 5.6 is all that is needed for the first iteration of the algorithm. For subsequent iterations on successively smaller subgraphs of G, the more general theorem is needed to determine the set of least-cost paths to be computed and which vertices' reach bounds are determined by those paths. Theorem 5.2 is needed to compute the reach bounds.

Theorem 5.1 provides a bound on the reach of a node on a path in terms of the reach, in the graph, of the path's endpoints.

Theorem 5.1 (Simple Reach Bounding Theorem): Given,

- a directed graph $G = (V,E)$ with positive weights
- a non-negative reach metric $m : E \longrightarrow R$
- a vertex v in G
- a least-cost path P in G from s to t including v
- a subpath of P, P', from s' to t' and including v

then, $r(v,P) \leq \min\{r(s',G) + m(s',v,P'), r(t',G) + m(v,t',P')\}$

For Theorems 5.2, 5.3, and 5.4, the following are given:

- a directed graph $G = (V,E)$ with positive weights
- a non-negative reach metric $m : E \longrightarrow R$
- a subgraph of G, $G' = (V',E')$, such that $E' = (u,v)|u,v \in V', (u,v) \in E$
- a vertex v in G' such that $r(v,G) > 0$
- a path P which is minimal with respect to the reach of v in G

In addition, for those theorems, we let

- path P' = the longest subpath of P containing v and included in G'
- s and s' be the first vertices of P and P', respectively
- t and t' be the last vertices of P and P', respectively
- f be the first edge of P after s' (the first edge of P' if P' has more than one vertex)
- l be the last edge of P before t' (the last edge of P' if P' has more than one vertex) (note: f and l exist because $r(v,P) = r(v,G) > 0$, so P must have at least one edge prior to v and one edge following v)

Theorem 5.1 cannot be readily applied when the reach of s' and t' are unknown. Theorem 5.2, which follows from Theorem 5.1, can be applied when the reach of vertices in $G - G'$ only are known. Theorem 5.2 will show that computing P' is a key to computing a reach bound for v.

Theorem 5.2 (Practical Reach Bounding Theorem): Let

- $g = \max\{r(x,G) + m(x,s')|(x,s') \in E - E'\}$ if there is a $(x,s') \in E - E'$, otherwise $g = 0$,
- $h = \max\{r(y,G) + m(t',y)|(t',y) \in E - E'\}$ if there is a $(t',y) \in E - E'$, otherwise $h = 0$,

then, $r(v,G) = r(v,P) \leq \min\{g + m(s',v,P'), h + m(v,t',P')\}$.

With Theorem 5.3, we begin to characterize the least-cost paths that must be computed to compute reach bounds by placing bounds on the lengths of the paths consistent with ensuring that P' is among those paths.

Theorem 5.3 (Simple Minimal Path Bounding Theorem):
$m(P') \leq 2r(v,G') + \max\{m(l) + m(s,s',P), m(f) + m(t',t,P)\}$

The right-hand side of this inequality contains terms that are unknown during the computation: $m(s,s',P)$, $m(t',t,P)$, and $r(v,G')$. Theorems that follow will replace the unknown terms with larger but known values.

Theorem 5.4 (Practical Minimal Path Bounding Theorem): Let

- $c = \max\{r(x,G) | x \in V - V'\}$ if $V \neq V'$, otherwise 0,
- d be defined such that if there exists a vertex u immediately preceding s' on P, then $d = m(u,s')$, and if not, then $d = 0$,
- e be defined such that if there exists a vertex u immediately following t' on P, then $e = m(t',u)$, and if not, then $e = 0$,

then, $m(P') \leq 2r(v,G') + c + \max\{m(l)+d, m(f)+e\}$.

The value of c can be computed by iterating over $V - V'$, but there remains one term whose value is unknown during computation of the least-cost paths: $r(v,G')$. This problem can be addressed by limiting v to those vertices of G' for which $r(v,G') < b$ for some suitably chosen value b. Then b can substituted into the inequality of Theorem 5.4 in place of $r(v,G')$, and all of the values on the right-hand side of the inequality become readily available during the computation for such v. However, we must be able to distinguish vertices v that satisfy $r(v,G') < b$ from those that don't. Theorem 5 allows the computation to make this distinction.

Theorem 5.5 (Reach Bound Validation Theorem): Given,

- some $b > 0$,
- a directed graph $G' = (V', E')$ with positive weights,
- a non-negative reach metric $m : E' \longrightarrow R'$,
- a vertex $v \in G'$ such that $r(v,G') \geq b$,
- a path P' in G' which includes v and is minimal for $r(v,G') \geq b$ (which we define to mean that, for any proper subpath of P', Q, including v, $r(v,Q) < b$),

if f and l are, respectively, the first and last edges of P' (f and l exist because $r(v,P') = r(v,G') > 0$), then $m(P') < 2b + m(f) + m(l)$.

Theorem 5.6 puts Theorems 5.4 and 5.5 together to describe a set of least-cost paths from which the reach bounds for some vertices in G' can be computed.

Theorem 5.6 (Pruning Theorem for Reach Bound Computation): Given,

- G, G', m, v, and P as defined for Theorems 5.2, 5.3, and 5.4,
- some $b > 0$,
- the set S of all least-cost paths in G' such that each path $P' \in S$ satisfies

  P' has at least one edge, and
  $m(P') < 2b + c + \max\{m(l)+d, m(f)+e, m(f)+m(l)\}$

  where

  s' and t' are, respectively, the first and last vertices of P',
  f and l are, respectively, the first and last edges of P',
  $c = \max\{r(x,G) | x \in V - V'\}$ if $V \neq V'$, otherwise 0,
  $d = \max\{m(u,s') | (u,s') \in E - E'\}$ if such a u exists, otherwise 0,
  $e = \max\{m(t',u) | (t',u) \in E - E'\}$ if such a u exists, otherwise 0,

if $\max\{r(v,P') | P' \in S\} < b$, then $r(v,G') < b$, and S includes the longest subpath of P containing v and included in G' provided the subpath has at least one edge.

In other words, for a v meeting the conditions of Theorem 5.6, the set S includes a path P' with which we can apply the reach bound formula of Theorem 5.2. We don't know which P' it is, but the maximum value for all such P' in S is clearly a safe bound.

6 Algorithm and Implementation for Computing Reach Bounds

The iterative process that computes reach bounds, as previously explained, computes partial least-cost path trees on progressively smaller subgraphs of the

input graph G. As the subgraphs become smaller the partial least cost path trees extend greater distances from their roots in order to compute reach bounds for more vertices. Theorem 5.6 helps determines how far each partial least-cost path trees must extend while Theorem 5.2 provides a formula for the reach bounds.

There are three differences between the theory described in the preceding section and our actual implementation. We describe those differences, meant to simplify the implementation and enhance its performance, before describing the algorithm we used.

First, Theorem 5.6 does not actually call for the computation of least-cost path trees. In general, the set of paths prescribed by Theorem 5.6 might not form a tree, but a directed acyclic subgraph of G because, for given source and target vertices, there can be multiple least-cost paths. Though computing least-cost path dags is feasible, it's much simpler to compute least-cost path trees using the standard Dijkstra algorithm. Each iteration computes one partial least-cost path tree for each vertex in G' applying Theorem 5.6 to determine their extent. If all least-cost paths in G' are unique, that is, no two least-cost paths have the same origin and destination, then those trees contain all of the paths in the set S described by Theorem 5.6. If two least-cost paths have the same origin and destination, then the trees do not contain all of the paths in S. As a result, there might be some v for which the reach bound computed is incorrect. Such a v lies on a least-cost path, P, which is not unique to its origin and destination. However, an alternate path with the same origin and destination in G' is included in the tree rooted at that origin. So although the reach bound for v is underestimated it never prevents the reach-based Dijkstra algorithm from finding a least-cost path.

The other two differences improve performance by helping to limit the size of the trees computed. A partial least-cost path tree can be computed by running a Dijkstra algorithm that terminates when the tree is sufficiently large to contain the desired paths. However, some branches of the tree might be extended much further than necessary, because the cost-metric, or weights, of the graph determine how the tree grows while the reach metric determines whether a particular branch of the tree has been extended far enough. An optimization stops exploration on branches that satisfy Theorem 5.6 before other branches do. A side effect is that the tree produced contains some paths which are not least-cost paths. We found that this optimization only slightly increases the reach bounds computed, and has no other ill effects, if implemented conservatively.

The third difference is the most important, calls for some restatement of theorems, and is reflected in the psuedocode given at the end of this section. The test, from Theorem 5.6, to determine whether a particular path, P', is needed in the tree,

$$(4)\quad m(P') < 2b + \max\{r(x,G)|x \in V - V'\} + \max\{m(l) + d, m(f) + e, m(f) + m(l)\}$$

is difficult to apply efficiently, that is, in such a way that keeps each partial least-cost path trees small. To be applied efficiently an inclusion test, $I(P)$, which is true if path P is be included in the tree, should have this property:

> Given P and Q, both paths starting at the root of the least-cost path tree, such that P is a subpath of Q, and Q has one more vertex that P, then $I(Q) \Rightarrow I(P)$. Conversely, $\text{not}(I(P)) \Rightarrow \text{not}(I(Q))$.

This permits the computation to stop at P if $I(P)$ fails and avoid evaluating $I(Q)$. If $I(P)$ fails, no least-cost path that extends through and beyond the endpoint of P need be included in the least-cost path tree. We call this a "monotonic inclusion test". The term e in (4) prevents it from being monotonic. All of the other terms in (4), except $m(l)$ and e, are fixed for a given least-cost path tree, so without those two terms, (4) would be monotonic, but only e prevents monotonicity.

One way to form a monotonic inclusion test is to replace e by some constant. Recall that e is the maximum length of edges in $E - E'$ leaving the last vertex of P'. If we replace e by k where k is the maximum length of any edge in $E - E'$, we get a monotonic test,

$$(5)\quad m(P') < 2b + \max\{r(x,G)|x \in V - V'\} + \max\{m(l) + d, m(f) + k, m(f) + m(l)\}$$

Theorem 6.1: (5) is a monotonic inclusion test.

The disadvantage of (5) is that k can be large. One remedy is to replace long edges in the graph with several smaller ones reducing k to some maximum allowed edge length. This might work well in some road networks.

However, we implemented a simpler approach though one that requires restatement of some theorems. In this approach, the least-cost path trees are generated, not from G' alone, but from H whose edges and vertices, respectively, are

$$E(H) = \{(x,y)|x \in V', y \in V\}$$
$$V(H) = V' \cup \{v|\exists(x,v) \in E(H)\}$$

So H includes vertices in G' and vertices in G adjacent to vertices in G':

The following theorems are slight restatements of earlier theorems to support this refined approach.

Theorem 6.2 (Practical Reach Bounding Theorem): Given the conditions of Theorem 5.2, $r(v,G) = r(v,P) \leq \min\{g + m(s',v,P'), r(t',G) + m(v,t',P')\}$

Theorem 6.4 (Monotonic Minimal Path Bounding Theorem): Given, all of the conditions and defintions of Theorem 5.4 and H as defined above with the modification that path P' be the longest subpath of P containing v and included in H (instead of G'), then $m(P') \leq 2r(v,H) + \max\{r(x,G)|x \in V - V'\} + \max\{m(l) + d, m(f)\}$

From Theorem 6.4, we can derive Theorem 6.6 in the same way that Theorem 5.6 is derived from Theorem 5.4.

Theorem 6.6 (Pruning Theorem for Reach Bound Computation): Given,

- G, G', m, v, P, and b as defined for Theorem 5.6,
- H as defined above,
- the set S of all least-cost paths in H such that each path P' in S satisfies

  P' has at least one edge, and
  $m(P') < 2b + c + \max\{m(l) + d, m(f), m(f) + m(l)\}$

  where

  s' and t' are, respectively, the first and last vertices of P',
  f and l are, respectively, the first and last edges of P',
  $c = \max\{r(x,G)|x \in V - V'\}$ if $V \neq V'$, otherwise 0,
  $d = \max\{m(u,s')|(u,s') \in E\}$ if such a u exists, otherwise 0,

if $\max\{r(v,P')|P' \in S\} < b$, then, $r(v,H) < b$, and S includes a longest subpath of P containing v and included in H provided the subpath has at least one edge.

In comparison with Theorem 5.6, Theorem 6.6 eliminates the e term in the inequality that each path P' in S satisfies.

In the pseudocode of Figure 1, *ReachBoundComputation* computes reach bounds on graph G by calling *Iterate* repeatedly passing it graph G and subgraph G'. *Iterate* attempts to compute reach bounds for vertices of G' given reach bounds for vertices of $G - G'$. Initially $G' = G$ and reach bounds, held in array *bounds*, are set to infinity. Vertices for which finite reach bounds are computed are removed from G' after each iteration. After the last iteration, G' is expected to be small enough that assigning each of its vertices an infinite reach bound, which means those vertices are never pruned by the routing algorithm, does not significantly affect query performance. The b parameter of *Iterate* (corresponding to b in Theorem 6.6) controls the trade-off between the amount of computation performed by the iteration and the amount of reduction in the size of G'.

In each iteration, Theorem 6.6 is applied to compute least-cost path trees needed to assign reach bounds to some vertices in V'. Each computed tree is traversed and Theorem 6.2 is applied to each vertex, v, in the tree to produce a reach bound on the assumption that the tree contains, P', the longest subpath in G' of a path which is minimal with respect to the reach of v in G. The maximum of the bounds over all of the trees is computed for each v. At the same time, the reach of each v over all of the trees is computed and used with Theorem 6.6, to identify some v for which one of trees contains the actual P'. For those v, the maximum bound computed over the trees is a valid reach bound for v in G.

7 Experimental Results

We present performance results for both the computation of reach bounds (for all but 3% to 5% of vertices) and the reach-based variant of Dijkstra's algorithm. All tests were performed by C++ programs without compiler optimization on a 2GHz PC running Linux with sufficient memory to hold all of the graph representation, reach bounds, and data structures required by the algorithms.

All of the tests were performed on graphs representing real road networks. For all tests, the cost metric was travel time and the reach metric was travel distance.

Our priority queue was a bucket priority queue (see [2, 3, 8, 9]).

Table 1 shows a somewhat greater than linear increase in computation time of reach bounds as the size of the dataset increases. This is what we expected. Generally, for hierarchical networks, we expect each iteration to require computation proportional to the size of the network, but that the number of iterations needed will increase as a sublinear function of size.

With the increase in data size, computation of ex-

ReachBoundComputation$(G, B, bounds)$

// G is a graph, (E, V), with weight function, w, and reach metric, m.
// B is an array of increasing positive integers.
// *bounds* is an array indexed by $v \in V$ and into which reach bounds are placed
$G' := G$
For each $v \in V$:
    $bounds[v] := \infty$
For each index, i, of B, in ascending order:
    Call $Iterate(G, G', B[i], bounds)$ // attempts to set finite values in *bounds*
    $V' := \{v | v \in V, bounds[v] = \infty\}$ // should become smaller in succesive iterations
    $E' := \{(u, v) | (u, v) \in E, u \in V', v \in V'\}$
    $G' := \{V', E'\}$

Iterate$(G, G', b, bounds)$

If $V = V'$ then $c := \max\{bounds[x] | x \in V - V'\}$ else $c := 0$
For each $v \in V'$:
    $bounds[v] := 0$ // will set back to ∞ if needed
    $r[v] := 0$ // reach of v in least-cost path trees
Form the graph H:
    $EH := \{(x, y) | x \in V', y \in V\}$
    $VH := V' \cup \{v | \exists (x, v) \in EH\}$
    $H := \{VH, EH\}$
For each vertex $s' \in V'$:
    If there exists $(x, s') \in E - E'$
        $g := \max\{bounds[x] + m(x, s') | (x, s') \in E - E'\}$
        $d := max\{m(x, s') | (x, s') \in E - E'\}$
    else
        $g := d := 0$
    $T :=$ partial least-cost path tree of H rooted at s' containing all P' such that
        $m(P') < 2b + c + d + m(f) + m(l)$, where f and l are the first and last edges of P'
        // this inequality is simpler but slightly looser than Theorem 6.6 requires
    Traverse T (once) to do the following for each vertex, v, in T:
        Compute $r(v, T)$
        Over all paths, P', in T that begin at s', include v, and end at a leaf, t', of T:
            If $t' \in V - V'$
                $rt := bounds[t']$
            else
                $rt := 0$ // in this case, Theorem 6.6 guarantees that P terminates at t'
            $rb := \min\{g + m(s', v, P'), rt + m(v, t', P')\}$ // application of Theorem 6.2
            if $rb > bounds[v]$
                $bounds[v] := rb$
        if $r(v, T) > r[v]$
            $r[v] := r(v, T)$
For each $v \in V'$:
    If $r[v] \geq b$ // apply Theorem 6.6
        $bounds[v] := \infty$ // reach bound not validated

Figure 1: **Pseudocode for Reach Bound Computation**

region	number of vertices	exact reach computation	reach bound computation
Alameda County	97240	233 minutes	28 minutes
San Francisco Bay Area	393368	4415 minutes	161 minutes

Table 1: **Exact Reach and Reach Bound Computations**

algorithm	avg route length	cpu time (1000 routes)	priority queue insertions per path
Dijkstra	26 kilometers	62 seconds	44122
A*	26 kilometers	60 seconds	27395
Reach	26 kilometers	14 seconds	5058
Reach + A*	26 kilometers	12 seconds	3711
Exact Reach	26 kilometers	9 seconds	3199

Table 2: **Shortest Path Computation for Alameda**

algorithm	avg route length	cpu time (1000 routes)	priority queue insertions per path
Dijkstra	56 kilometers	289 seconds	179263
A*	56 kilometers	194 seconds	79293
Reach	56 kilometers	28 seconds	10043
Reach + A*	56 kilometers	17 seconds	5314
Exact Reach	56 kilometers	15 seconds	5797

Table 3: **Shortest Path Computation for the Bay Area**

algorithm	avg route length	cpu time (1000 routes)	priority queue insertions per path
Dijkstra	52 kilometers	334 seconds	141464
A*	52 kilometers	140 seconds	50692
Reach	52 kilometers	27 seconds	7910
Reach + A*	52 kilometers	13 seconds	3473

Table 4: **Shortest Path to Box Computation for the Bay Area**

act reach values increased 19 times, roughly in proportion to the square of the size of the dataset as we'd expect from an all-pairs shortest path computation.

We compared the performance of shortest path computations for four Dijkstra variations. In each case, 1000 random routes were computed. Each route was chosen randomly by randomly choosing two vertices from the graph. The random choices of the two vertices were independent, so the average distances between them were roughly proportionate to the geographic width and length of the road network.

Comparing the tests on the smaller network (Table 2) with the tests on the larger network (Table 3), suggests a linear increase in computation as a function of path length using the reach algorithm and sublinear using the combination of reach and A*. The computation time of the Dijkstra algorithm on road networks is commonly considered to increase with the square of the distance as rule of thumb. The improvement in performance provided by the reach algorithm is consistent with the heuristic approach using road classifications often used in industry.

The numbers of priority queue insertions per path computation give some insight into the factors determining performance. The A* algorithm, for instance, exhibits a reduction in priority queue insertions out of proportion to the reduction in computation time. That suggests that the cost of the estimate function, which involves an expensive square root, is itself significant.

The improvements provided by reach and by A* appear to be independent from each other and complimentary.

The performance of the path computation using exact reach values (without A*) suggests that there is considerable room for trade-off between reach preprocessing time and query performance and/or that the preprocessing can benefit from tuning.

Tests with two road networks in Asia demonstrated similar reductions in computation time compared to Dijkstra ranging from a 5 times reduction to 10 times reduction.

To validate the reach algorithms and their implemetation, we compared the cost of each path computed among the various algorithm implementations. There were no differences.

In a parallel series of tests (Table 4), the destinations were regions bounded by latitude and longitude lines (which we call "boxes"). The origin was a vertex chosen at random. The center of the box and its size were randomly chosen. Combinations in which the origin vertex was inside the box were discarded. This simulates an application in which it is desirable to know how soon a mobile device can enter a region using the road network. These tests demonstrate the versatility of the reach-based algorithms and suggest that reach-based algorithms would provide the fastest possible means of computing paths to multiple destinations or from multiple origins - given the difficulties other approaches face.

8 Further Work

It's desirable to characterize the performance of our algorithms as functions of easily computed network properties. Under some conditions, we think, the computation time of our shortest path algorithm is a linear, or near linear, as a functon of some metric on the path computed. Such a bound would be considered meaningful in road network applications. Considering the distribution of reach values would play a role in the analysis.

We also think that, under some conditions, there is a dynamic, or incremental, approach that allows fast update to the reach bounds when there are localized changes to the graph.

9 Acknowledgements

This research was performed at Wavemarket, Inc in Emeryville, California.

Thanks to Wavemarket, Inc. and Scott Hotes at Wavemarket for making this research and paper possible.

Thanks to Dave Blackston at Wavemarket for his comments on this paper.

Patents related to the ideas in this paper have been filed by Wavemarket, Inc.

References

[1] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni. Incremental Algorithms for Minimal Length Paths. *Journal of Algorithms*, 12(4):615-638, 1991.

[2] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. *Siam Journal of Computing*, 28(4):1326-1346, 1999.

[3] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. In *Proceedings of the Eight Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 83-92, 1997.

[4] T. H. Cormen, C. E. Leiserson, R. E. Rivest, and Clifford Stein. *Introduction to Algorithms.* Second Edition, MIT Press, 2001.

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

[6] Hristo N. Djidjev, Grammati E. Pantziou, Christos D. Zaroliagis. Improved Algorithms for Dynamic Shortest Paths. *Algorithmica*, 28(4):367-389, 2000.

[7] H. N. Djidjev. Efficient Algorithms for Shortest Path Queries in Planar Digraphs. In *Proceedings of the 22nd Workshop on Graph Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, pages 151–165. Springer Verlag, 1996.

[8] A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. Technical Report 95-187, NEC Research Institute, Princeton, NJ 1995.

[9] R. Gutman. Priority Queues for Motorists. *Dr. Dobb's Journal*, 340:89-94, 2002.

[10] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100-107, 1968.

[11] P. Hart, N. Nilsson, and B. Raphael. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Newsletter*, no. 37:28-29, 1994.

[12] R. Jacob, M. Marathe, and K. Nagel. A Computational Study of Routing Algorithms for Realistic Transportion Networks. *ACM Journal of Experimental Algorithms*, 4(6), 1999.

[13] N. Jing, Y. W. Huang, and E. Rundensteiner. Hierarchical Encoded Path Views for Path Query Processing: an Optimal Model and its Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409-431, 1998.

[14] P. Klein, and S. Subramanian. A Fully Dynamic Approximation Scheme for Shortest Path Problems in Planar Graphs. *Algorithmica*, 22(3):235-249, 1998.

[15] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster Shortest-Path Algorithms for Planar Graphs. *Special issue of Journal of Computer and System Sciences on selected papers of STOC 1994*, 55(1):3-23, 1997.

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