How MapQuest Works

Michael O'Brien

February 21, 2006

Abstract

MapQuest is a free online service that calculates the optimal route for driving between two locations. It uses some variant of the bidirectional version of Dijkstra's algorithm with some "heuristic tricks to minimize the size of the graph that must be searched" [1]. Apart from this, AOL (MapQuest's parent company) is reluctant to release information on the specifics of their algorithm as this would jeopardise their position as a market leader.

In this project we consider 5 point-to-point shortest path algorithms (Dijkstra, A^*, `ALT`, `RE` and `REAL`) and consider their applicability to the MapQuest problem.

Contents

1 Introduction **3**

2 Formulation of the Problem **4**

3 Variants of Dijkstra's Algorithm **5**

3.1 The Basic Dijkstra Algorithm 5

3.2 Bidirectional Version of Dijkstra's Algorithm 9

4 Variants of the A^* Algorithm **12**

4.1 The Original A^* Algorithm 12

4.2 Bidirectional Lower Bounding Algorithms 16

5 The `ALT` Algorithm **19**

6 The Reach Algorithm **22**

7 Combining the Reach and `ALT` Algorithms **24**

8 Performance of Algorithms **25**

1 Introduction

The problem of finding your way through the back roads and byways of Ireland is familiar to most, but fumbling with cumbersome fold-out maps and struggling with inadequate road signs could soon become a thing of the past due to services like MapQuest. MapQuest produces detailed driving directions and customised maps within a few seconds for tens of millions of people a day [1]. These driving directions can go from one specific street address to another and also give an estimate of travel time. Such a service obviously takes a lot of the hassle out of journey planning.

The problem of finding the shortest path between two different places can essentially be seen as the problem of finding the shortest path between two vertices on a graph, where vertices are locations and edges are roads. Thus we consider algorithms for finding the shortest path between two points on a graph. Starting with the most well known, Dijkstra's algorithm, we gradually add more and more heuristic tricks until we construct an algorithm capable of finding the shortest path between any two places in North America in under 4ms [6].

While MapQuest is not yet available in Ireland, services like Yahoo! Maps and the AA do provide driving directions between most places. And while they are not yet as specific as MapQuest is in the US, it is really only a matter of time before they catch up.

2 Formulation of the Problem

To begin with, we define a graph. A graph G is a finite set of vertices, V, together with a collection of pairs of vertices, E, called edges. This is written as $G = (V, E)$. The vertices are represented by points and the edges by lines joining pairs of points. If an edge, e, joins a pair of vertices x and y then x and y are said to be *adjacent*.

A graph G is said to be weighted if each edge, e, is assigned a nonnegative number, $w(e)$. We exclude multiple (i.e. parallel) edges. If e joins the vertices x and y we define $l(x, y) = w(e)$. If a path from a to z is defined by $a \to b \to \ldots \to z$, then the length of this path is $l(a, b) + l(b, c) + \ldots + l(y, z)$. The length of a shortest path between two vertices a and z (there may be more than one such path) is defined as $dist(a, z)$.

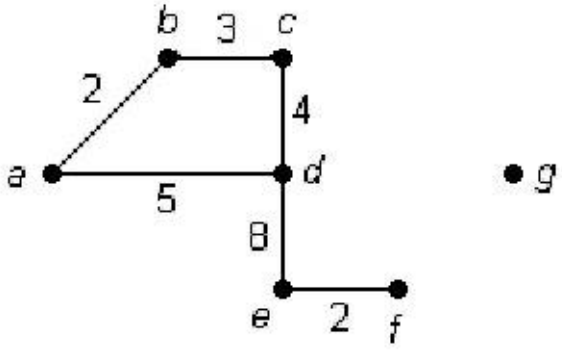

Figure 1: An example of a graph.

Note that some vertices may not be adjacent to any other (e.g. vertex g in Figure 1). The numbers beside the edges represent the weights of those edges. In Figure 1, the path $a \to d \to e \to f$ (written a–d–e–f) has length 15 and $dist(a, f) = 15$ also.

In the context of this project, the vertices are locations and the edges are

the roads between them. The weights are generally travel times or distances between locations but do not have to be given a physical interpretation.

The MapQuest problem is defined as follows: given a starting point s and a terminal point t, find a shortest path between s and t. We always assume that t is reachable from s (i.e. that it is possible to travel along edges in the graph from s to t).

3 Variants of Dijkstra's Algorithm

Dijkstra's is the most intuitive shortest path algorithm that we will present.

3.1 The Basic Dijkstra Algorithm

The original version of the algorithm was developed by E.W. Dijkstra in 1959 [2]. It is best illustrated by use of an example. Consider the graph of Figure 2:

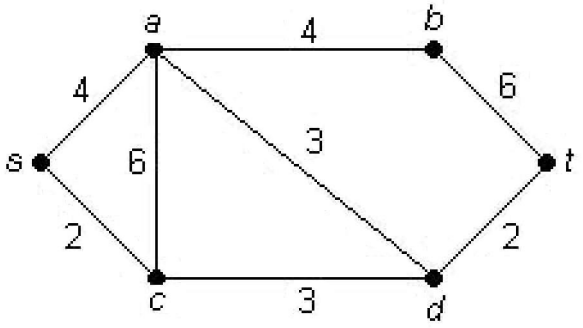

Figure 2:

What is the shortest path from s to t? Although the solution is obvious in this case, we will proceed by finding the closest vertex to s, then the second closest vertex to s and so on until t is reached.

The only paths leaving s are s–a and s–c. As the lengths of these paths are 4 and 2 respectively, c is the closest vertex to s.

To find the second closest vertex to s, we need only consider vertices adjacent to s and c. The shortest path to a is still s–a and has length 4 whereas the shortest path to d is s–c–d and has length 5. Therefore a is the second closest vertex to s.

Similarly, for the third closest vertex we need vertices adjacent to s, c and a. There is a path of length 8 from s to b (s–a–b) and a path of length 5 from s to d. Hence d is the third closest vertex to s.

Finally, proceeding as before, we see that t is the fourth closest vertex to s and the shortest path is s–c–d–t.

We will now generalise this technique and formulate *Dijkstra's algorithm* for finding a shortest path between a starting point, s, and a terminal point, t, in a simple, connected, undirected, weighted graph.

The algorithm is an iterative one. At each iteration it adds another vertex to a distinguished set S. Throughout the process, the algorithm maintains a distance label $d(v)$ for each vertex v. This is the length of the shortest path from s to v that contains only vertices already in S and the vertex v itself. The parent vertex, $p(v)$, of every vertex v is also recorded. This is the vertex immediately preceding v in the shortest path to v.

Initially $d(v) = \infty$ and $p(v)$ is not yet defined for every vertex v. The algorithm starts by setting $d(s) = 0$ and adding s to the distinguished set, S. While there are vertices in S, the algorithm 'relaxes' all edges from the last vertex added to S to vertices outside S. To relax an edge (v, w) one checks if $d(w) > d(v) + l(v, w)$ and, if true, sets $d(w) = d(v) + l(v, w)$. We then say that the vertex w has been *scanned*. When w has been scanned, $d(w)$ will be equal to the length of the shortest path from s to w seen so far. We say w has been *labelled* with $d(w)$. At each iteration the algorithm adds to S the vertex outside S with the smallest label. If multiple vertices have the same smallest label, any such vertex may be chosen. We say that the

vertex added to S has been *selected*. When a vertex v is selected, $p(v)$ is set equal to the vertex immediately preceding it in the shortest path from s to v. Each iteration of the algorithm ends with the selection of a vertex. The algorithm terminates when t has been selected. Note that the algorithm must terminate as there are a finite number of vertices in the graph and one vertex is added to S at each iteration.

Theorem 3.1. *[3] Consider a weighted graph $G = (V, E)$. When a vertex v is selected by Dijkstra's algorithm it has been labelled with $dist(s, v)$, the length of the shortest path from s to v.*

Proof. The proof of this theorem is by induction on k, the iteration counter.

Let S_k be the set S after the k^{th} iteration.

The inductive hypothesis is as follows: at the k^{th} iteration

(i) the label of a vertex v in S_k is the length of the shortest path from s to v, and

(ii) the label of a scanned vertex w not in S_k is the length of the shortest path from s to w that contains only vertices in S_k (apart from w itself). If no such path exists (i.e. w is not adjacent to a vertex in S and so has not been scanned) then its label is ∞.

The first step of the algorithm sets $d(s) = 0$, $d(v) = \infty$ for $v \neq s$ and then selects s. The shortest path from s to itself is of length 0 so (i) is true for the case $k = 1$. As no other vertices have been scanned yet, (ii) is also true for the case $k = 1$.

Assume that the inductive hypothesis holds true for the k^{th} iteration. Let v be the vertex added to S_k at the $(k + 1)^{st}$ iteration to form S_{k+1}.

By the inductive hypothesis, every vertex x in S_k is labelled with the length of the shortest path from s to x. Also, v must be labelled with the length of the shortest path from s to v. If this were not the case, at the end

of the k^{th} iteration there would be a path of length less than $d(v)$ containing at least one vertex not in S_k (because $d(v)$ is the length of the shortest path from s to v containing only vertices in S_k). Let u be the first vertex not in S_k in such a path. Then there is a path of length less than $d(v)$ from s to u containing only vertices in S_k. This contradicts the choice of v (because v is the vertex with the smallest label outside S_k). Thus (i) is true at the $(k+1)^{st}$ iteration.

Let w be a scanned vertex not in S_{k+1}. Then w must be adjacent to some vertex in S_k (otherwise it would not have been scanned). By part (i) we know the length of the shortest path to all vertices in S_k to which w is adjacent. When all edges from these vertices are relaxed w must be labelled with the length of the shortest path from s to w containing only vertices in S_k and w itself by the way the labelling process is defined. If a vertex has not been scanned, then its label remains ∞. Thus (ii) holds true at the $(k+1)^{st}$ iteration.

If the inductive hypothesis holds true at the k^{th} iteration, we have shown that it holds true at the $(k+1)^{st}$ iteration. By the priciple of induction, the theorem has been proved. □

Theorem 3.2. *[4, Theorem 3.1] In a weighted graph $G = (V, E)$, Dijkstra's algorithm selects vertices in nondecreasing order of their distances from the starting vertex s.*

Proof. The proof of this theorem is by induction on k, the iteration counter.

The inductive hypothesis is that $d(v) \geq d(x)$ where v is the vertex selected at the k^{th} iteration of the algorithm and x is any vertex already in S.

The first step of the algorithm selects s. As no other vertex can be closer to s than s itself, the case $k = 1$ is true.

Assume the hypothesis holds true for the k^{th} iteration and that v is the

vertex selected at this iteration. Let w be the vertex selected at the $(k+1)^{st}$ iteration.

The shortest path to w either contains v or it does not. If it does contain v, then $d(w) = d(v) + l(v, w) \geq d(v)$ as $l(v, w)$ is nonnegative. By the inductive hypothesis, $d(v) \geq d(x)$ for all vertices x in S prior to the k^{th} iteration. Thus $d(w) \geq d(x)$ for all vertices x in S prior to the $(k+1)^{st}$ iteration.

If the shortest path from s to w does not contain v then w must be adjacent to some vertex selected earlier (as only vertices adjacent to selected vertices can possibly be selected). As v was selected before w, so $d(w) \geq d(v)$, otherwise w would have been selected before v. Thus, as before, $d(w) \geq d(x)$ for all vertices x in S prior to the $(k+1)^{st}$ iteration.

Thus the hypothesis is true for the $(k+1)^{st}$ iteration if it is true for the k^{th} iteration.

By the principle of induction, this theorem has been proved. □

The algorithm terminates when t is selected. By Theorem 3.1, the vertex t has been labelled with the length of the shortest path from s to t when it is selected. By Theorem 3.2 all vertices selected before t will be as close or closer to s than t. When the algorithm terminates we find the shortest path from s to t by noting $p(t), p(p(t))$ and so on until s is reached.

3.2 Bidirectional Version of Dijkstra's Algorithm

If we are dealing with a weighted graph each of whose edges has length 1, it can be seen intuitively that Dijkstra's algorithm searches a ball with s at the centre and t on the boundary. If the distance from s to t is D, then Dijkstra searches an area of roughly πD^2. If we run the algorithm forward from s and backwards from t (i.e. with t as the starting point and s as the terminating point) simultaneously, terminating when the two searches meet (i.e. their distinguished sets have a non-empty intersection),

then this algorithm searches an area of roughly $2\pi(D/2)^2$, which is half the area scanned by the original algorithm. This is a very useful improvement for a service such as MapQuest as it would drastically reduce computation time.

Formally, the algorithm works as follows: the forward algorithm begins at s and proceeds to add vertices to its distinguished set based on their label, $d_s(v)$. The reverse algorithm begins at t and adds vertices to its distinguished set by virtue of their labels, $d_t(v)$. At each iteration of the bidirectional algorithm the forward algorithm selects one vertex and then the reverse algorithm selects one vertex. The algorithm also maintains the length μ of the shortest path between s and t seen so far. Initially, $\mu = \infty$. When an arc (v, w) is scanned by the forward algorithm and w has already been scanned by the reverse algorithm, we know the lengths $d_s(v)$ and $d_t(w)$ of the shortest paths from s to v and from t to w. If $\mu > d_s(v) + l(v, w) + d_t(w)$, we have found a shorter path and μ is updated accordingly. The value of μ is similarly updated by the reverse search. The algorithm terminates when the search in one direction selects a vertex that has been selected by the search in the reverse direction.

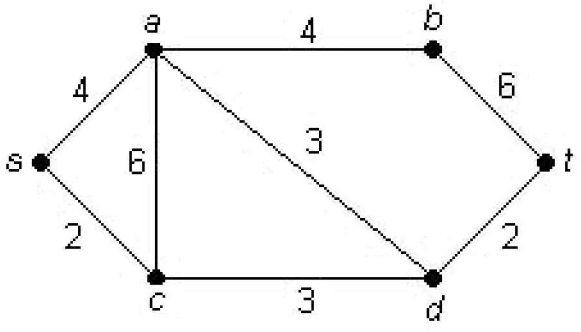

Figure 3:

To illustrate the bidirectional version of Dijkstra's algorithm we will

consider the graph of Figure 3, which was discussed in §3.1. In the first iteration the forward search scans the vertices a and c and labels them with 4 and 2 respectively. Then c, the closest vertex to s, is selected. The reverse search scans the vertices b and d and labels them with 6 and 2 respectively. Then d, the closest vertex to t, is selected. In the second iteration the forward search scans the vertices adjacent to c: a and d. The label on a remains unchanged and d is labelled with 5. The forward search selects a, the second closest vertex to s. The reverse search scans the vertices adjacent to d, that is a and c, and labels them both with 5. As there is a tie for the second closest vertex to t, the search can choose which vertex it selects next. Let's say a is selected. As a has already been selected by the forward search the algorithm terminates here. The shortest path from s to t (s–c–d–t) contains only vertices already selected.

This example illustrates an important point: the shortest path found by the bidirectional version of Dijkstra's algorithm does not necessarily include the vertex on which the algorithm terminates (in this example, vertex a).

Theorem 3.3. *The bidirectional version of Dijkstra's algorithm finds the length of a shortest path between two vertices.*

Proof. Let v be the vertex on which the algorithm terminates. Then v has been selected by both the forward and reverse searches and Theorem 3.1 implies that $d_s(v) = dist(s, v)$ and $d_t(v) = dist(v, t)$ (i.e. the distance labels are exact). Therefore the length of the shortest path from s to t through v is $d_s(v) + d_t(v)$. We want to show that the length of any path from s to t containing vertices that have not been selected by either search must be greater than $d_s(v) + d_t(v)$. Assume there is a path going from s to t through an unselected vertex w whose length is shorter than any path through vertices that have already been selected. If this were true, then the length of the path through w would be less than the length of the path

through v, i.e.

$$dist(s, w) + dist(w, t) < dist(s, v) + dist(v, t)$$

This must mean that $dist(s, w) < dist(s, v)$ or $dist(w, t) < dist(v, t)$ or both. If this were true then w would already have been selected by one of the searches as Dijkstra's algorithm selects vertices in nondecreasing order of their distance from the starting vertex (Theorem 3.2). We assumed w had not been selected and hence we have a contradiction. Thus the shortest path from s to t must contain only vertices that have already been selected. □

To find the shortest path from s to t we consider the set S of vertices selected by the forward search and the set T of vertices selected by the reverse search. Let $U = S \cup T$. To find the shortest path between s and t we run Dijkstra's algorithm on U starting at s. We already know the shortest path from s to all vertices in S and from t to all vertices in T and we also note that for large graphs, U will be considerably smaller than the entire graph. Thus Dijkstra's algorithm will converge rapidly and by Theorem 3.3 will find the shortest path between s and t.

4 Variants of the A^* Algorithm

Although the bidirectional version of Dijkstra's algorithm is a significant improvement, it is still not efficient enough on very large graphs of the type used by MapQuest.

4.1 The Original A^* Algorithm

One way to improve the efficiency of Dijkstra's algorithm is to estimate the distance from every vertex, v, to the terminal point, t. These estimates are made using an easily computed *potential function* π_t, with $\pi_t(v)$ estimating $dist(v, t)$. For example, the Euclidean distance between each vertex and t

could be used as the potential function. Then the vertex with the smallest value of $k_s(v) = d_s(v) + \pi_t(v)$ (with $d_s(v)$ defined as in §3.2) is added to the distinguished set at each step in the algorithm. This is known as the *best first search algorithm*. Unfortunately, it does not always yield a shortest path.

We define a new length function $l_{\pi_t}(v, w)$ for adjacent vertices v and w as follows:

$$l_{\pi_t}(v, w) = l(v, w) - \pi_t(v) + \pi_t(w)$$

The potential function π_t is said to be *feasible* if $l_{\pi_t}(v, w) \geq 0$ for all v, w.

The best first search algorithm combined with a feasible potential function is known as the A^* algorithm. The algorithm runs as follows: initially $k_s(v) = \infty$ and $p(v)$ is not defined for every vertex v. The vertex s is then added to the distinguished set S. While there are vertices in S, the algorithm 'relaxes' all edges from the last vertex added to S to vertices outside S. The A^* algorithm relaxes an edge (v, w) by checking if $k_s(w) > k_s(v) + l_{\pi_t}(v, w)$ and, if true, setting $k_s(w) = k_s(v) + l_{\pi_t}(v, w)$. Note that

$$\begin{aligned} k_s(v) + l_{\pi_t}(v, w) &= d_s(v) + \pi_s(v) + l(v, w) - \pi_s(v) + \pi_s(w) \\ &= d_s(v) + l(v, w) + \pi(w) \end{aligned}$$

and is therefore a reasonable estimate on the length of the path from s to t through w. At each iteration the algorithm selects the vertex with the smallest label. If multiple vertices have the same smallest label, any such vertex may be chosen. When a vertex v is selected its parent vertex $p(v)$ is updated as in Dijkstra's algorithm. Each iteration of the algorithm ends with the selection of a vertex. The algorithm terminates when t is selected. Note that the algorithm must terminate as there are a finite number of vertices in the graph and one vertex is selected at each iteration.

To illustrate the A^* algorithm we consider the graph of Figure 4, which was previously considered in Sections 3.1 and 3.2. The potential function

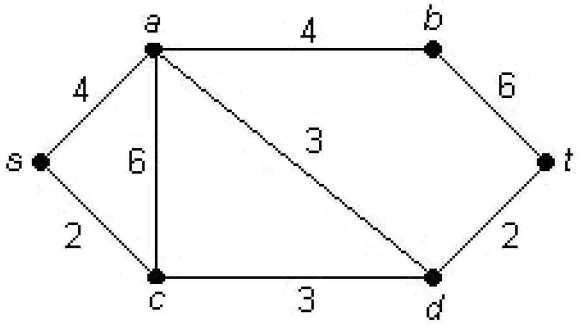

Figure 4:

we use is the Euclidean distance between each vertex and t. This distance is measured by hand and in cm. The potential function is defined as follows: $\pi_t(a) = \pi_t(c) = 5$ and $\pi_t(b) = \pi_t(d) = 2$. The algorithm begins by scanning a and c and labelling them with $k_s(a) = d_s(a) + \pi_t(a) = 9$ and $k_s(c) = d_s(c) + \pi_t(c) = 7$ respectively. The first vertex selected is c. Relaxing all edges from c results in no change in the label of a and d is labelled with $k_s(d) = d_s(c) + l(c,d) + \pi_t(d) = 7$. The second vertex selected is d. Relaxing all edges from d results in no change in the label of a and t is labelled with 7. The third vertex selected is t. The algorithm terminates and the shortest path (s–c–d–t) has been found. Note that only vertices on the shortest path were selected. This is because the estimates given by the potential function were almost exact. In general, the better the estimates given by the potential function, the fewer vertices selected outside the shortest path.

Theorem 4.1. *In a weighted graph $G = (V, E)$, the A^* algorithm finds a shortest path between s and t.*

Proof. At each iteration, the A^* algorithm checks whether

$$k_s(w) > k_s(v) + l_{\pi_t}(v, w)$$

This is equivalent to Dijkstra's algorithm with distance label $k_s(v)$ and

length function $l_{\pi_t}(\cdot,\cdot)$. Observe that $l_{\pi_t}(\cdot,\cdot) \geq 0$, because π_t is feasible. Thus the proof of Dijkstra's algorithm implies that the A^* algorithm finds a shortest path from s to t with respect to the length function $l_{\pi_t}(\cdot,\cdot)$.

To show that this path is the same shortest path as that found by Dijkstra's algorithm applied to the original problem, observe that for any 2 vertices a and z, the length of any path a–b–c–…–z with length function l is

$$l(a,b) + l(b,c) + \ldots + l(y,z) \tag{4.1}$$

whereas with the length function l_{π_t} it's

$$\begin{aligned} & l_{\pi_t}(a,b) + l_{\pi_t}(b,c) + \ldots + l_{\pi_t}(y,z) \\ & = l(a,b) - \pi_t(a) + \pi_t(b) + l(b,c) - \pi_t(b) \\ & \quad + \pi_t(c) + \ldots + l(y,z) - \pi_t(y) + \pi_t(z) \\ & = l(a,b) + l(b,c) + \ldots + l(y,z) + \pi_t(z) - \pi_t(a), \end{aligned} \tag{4.2}$$

a shift by a fixed amount $\pi_t(z) - \pi_t(a)$ (that is independent of the precise path chosen from a to z) of the original length.

Thus the length of all paths between a given pair of vertices is shifted by a constant when l is replaced by l_{π_t}. Consequently a shortest path from s to t in the graph with length function l is also a shortest path in the graph with length function l_{π_t}. The proof is complete. □

We refer to the class of A^* algorithms that use a feasible potential function π_t with $\pi_t(t) \leq 0$ and $\pi_t(v) \geq 0$ for all $v \neq t$ as *lower-bounding algorithms*. To see why, consider a path s–a_1–a_2–…–a_n–t. The length of this path with the length function l is

$$l(s,a_1) + l(a_1,a_2) + \ldots + l(a_n,v)$$

whereas with the length function l_{π_t}, by (4.2) it is

$$\begin{aligned} & l(s,a_1) + l(a_1,a_2) + \ldots + l(a_n,t) + \pi_t(t) - \pi_t(s) \\ & \leq l(s,a_1) + l(a_1,a_2) + \ldots + l(a_n,t) \end{aligned}$$

because $\pi_t(t) \leq 0$ and $\pi_t(s) \geq 0$. That is l_{π_t} gives a lower bound on the length of any path from any vertex s to t.

For the remainder of §4 we assume that π_t is a feasible potential function with $\pi_t(t) \leq 0$ and $\pi_t(v) \geq 0$ for all $v \neq t$. We also assume that $\pi_t(v)$ is a lower bound for $dist(v,t)$ for all vertices v.

4.2 Bidirectional Lower Bounding Algorithms

A bidirectional version of the A^* algorithm seems trivial: just run the algorithm forwards and backwards and terminate when the two searches meet. Unfortunately, there is no guarantee that this will work. It is quite possible, for example, that the potential function used π_t by the forward search and the potential function π_s used by the reverse search give rise to different length functions l_{π_t} and l_{π_s}. We therefore have no guarantee that the shortest path between s and t has been found when the algorithm terminates.

There are two ways to overcome this problem. The first is to ensure that both the forward and reverse search use the same length function (the *consistent approach*) and the second is to develop a new stopping condition (the *symmetric approach*).

We define $k_t(v) = d_t(v) + \pi_s(v)$.

The Consistent Approach

We say π_s and π_t are *consistent* if $l_{\pi_t}(v,w) = l_{\pi_s}(w,v)$ for every edge (v,w) in the graph. Now, $l_{\pi_t}(v,w) = l_{\pi_s}(w,v)$ is equivalent to

$$l(v,w) - \pi_t(v) + \pi_t(w) = l(w,v) - \pi_s(w) + \pi_s(v) \ \forall v,w. \tag{4.3}$$

But $l(v,w) = l(w,v)$ so (4.3) yields

$$\pi_t(w) + \pi_s(w) = \pi_t(v) + \pi_s(v)\ \forall v, w,$$

i.e. $\pi_t + \pi_s$ is a constant function.

Note that if p is a given feasible potential function for the forward search then the use of $-p$ in the reverse search will be consistent because (4.3) is then satisfied.

In general, if we are given any two potential function π_t and π_s we can construct consistent potential functions by using $p_t(v) = [\pi_t(v) - \pi_s(v)]/2$ for the forward search and $p_s(v) = [\pi_s(v) - \pi_t(v)]/2 = -p_t(v)$ for the reverse search. These are known as the *average potential functions*.

The consistent bidirectional algorithm terminates when the search in one direction selects a vertex that has already been selected by the search in the other direction. The shortest path has been found at this point. The proof that this algorithm works is the same as that of Theorem 3.3 upon noting that both the forward and reverse searches are merely modified versions of Dijkstra's algorithm.

The Symmetric Approach

In the symmetric bidirectional algorithm the length μ of the shortest path between s and t seen so far is maintained. Initially $\mu = \infty$. When the forward search scans a vertex w that has already been scanned by the reverse search, the algorithm checks if μ is greater than the length of the path formed by concatenating the shortest s–w path seen so far with the shortest w–t path seen so far. If it is, μ is updated accordingly. Similar updates are also made during the reverse search. The algorithm terminates when the search in one direction selects a vertex with $k_s(v) \geq \mu$ or $k_t(v) \geq \mu$ or when all vertices have been selected. If $k_s(v) = d_s(v) + \pi_t(v) \geq \mu$ then any path from s to t through v will have length greater than or equal to $k_s(v)$ (because

$\pi_t(v)$ is a lower bound for the distance from s to t) which is greater than or equal to μ. For any vertices w selected after v, we have $k_s(w) \geq k_s(v) \geq \mu$, as the algorithm selects vertices in nondecreasing order of the value of k_s. Hence the length of the path from s to t through any vertices selected after v will be greater than or equal to μ. Thus the shortest path must contain only vertices that have already been selected. Similar observations can be made for the reverse search using $k_t(v)$. If all vertices have been selected then we must know the shortest path. Thus the algorithm works.

These approaches both have their advantages. The consistent approach can stop when the two searches meet but may not use the best potential functions available. This means that the potential functions used might give bad estimates and so don't improve the efficiency of the algorithm as much. The symmetric approach, on the other hand, can use the best potential functions available but cannot stop when the searches meet. This may mean the algorithm runs for a long time after the searches meet and so is less efficient. The most efficient algorithm can be chosen in practice (e.g. if one has very good potential functions the symmetric approach would be used).

5 The ALT Algorithm

This is a version of the A^* algorithm that uses *landmarks* and the triangle inequality to improve efficiency. It is the first algorithm we consider that was explicitly constructed to solve the MapQuest problem. For that reason it takes a somewhat different approach to the previous algorithms. It is assumed in the construction of the ALT algorithm that the MapQuest problem would be solved repeatedly on the same graph. Therefore a much greater amount of preprocessing is allowed because it can improve the efficiency of all subsequent shortest path queries. This is also true of all the algorithms we consider later.

Given a graph $G = (V, E)$, we take a subset of the vertices and call them landmarks. Prior to commencing the algorithm, the shortest distance between each vertex v and each landmark L is calculated using one of the previous algorithms. We then define our potential function π_t as $\pi_t(v) = dist(v, L) - dist(t, L)$ for a given L. By the triangle inequality

$$dist(v, L) - dist(t, L) \leq dist(v, t). \tag{5.1}$$

Thus $\pi_t(v)$ gives a lower bound for the distance between v and t.

Note also that if vertices v and w are adjacent, then

$$\begin{aligned} l_{\pi_t}(v, w) &= l(v, w) - \pi_t(v) + \pi_t(w) \\ &= l(v, w) - (dist(v, L) - dist(t, L)) + (dist(w, L) - dist(t, L)) \\ &= l(v, w) - dist(v, L) + dist(w, L) \geq 0, \end{aligned}$$

again by the triangle inequality. Thus l_{π_t} is a feasible potential function.

We call the A^* algorithm with this potential function the ALT algorithm.

The set of landmarks is usually taken to be much smaller than the set of vertices (e.g. in [4] 64 landmarks are selected from graphs of over 6,000,000 vertices). There are a number of methods of selecting the best possible k landmarks for a graph. The simplest is to select the k vertices at random,

which works reasonably well. The method favoured by Goldberg et al. [4] is that of *farthest* landmark selection. The method works as follow: select any starting vertex and find a vertex v_1 that is farthest away from it. Add v_1 to the set of landmarks. Proceed in iterations, adding the vertex farthest from the current set of landmarks to the set of landmarks at each iteration. This process can be viewed as a rough solution to the problem of selecting a set of k vertices so that the minimum distance between each pair of vertices is maximised.

For each given s, t pair, a subset P of the set of landmarks (usually 16 in [4]) that give the highest lower bounds on $dist(s, t)$ by (5.1) are chosen. When the algorithm is scanning a vertex v, the landmark $L \in P$ that gives the highest lower bound on $dist(v, L)$ is used in the potential function.

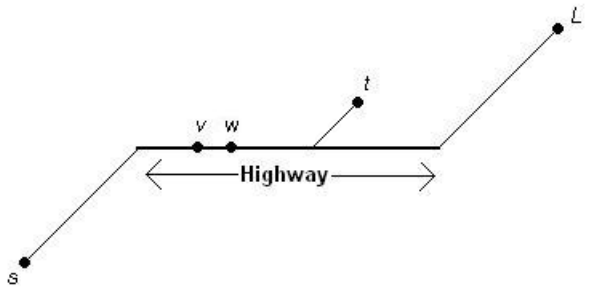

Figure 5: The advantages of the `ALT` algorithm

To see why this algorithm can be much more efficient than previous algorithms, consider the practical driving example in Figure 5 where s and t are locations that are far apart and L is the landmark location. The shortest path from s to L will typically consist of a segment from s to a highway, a segment that uses highways only and a segment from a highway to L. If L is chosen so that t lies approximately between s and L as in Figure 5, the shortest path from s to t will consist of the same segment from s to a highway, a segment using highways only and a segment from the highway

to t.

For adjacent vertices v, w on the segment shared by the two paths (as in Figure 5)

$$\begin{aligned} dist(v, L) &= l(v, w) + dist(w, L) \\ \Rightarrow l_{\pi_t}(v, w) &= l(v, w) - \pi_t(v) + \pi_t(w) \\ &= l(v, w) - dist(v, L) + dist(w, L) \\ &= l(v, w) - (l(v, w) + dist(w, L)) + dist(w, L) \\ &= 0. \end{aligned}$$

It follows that edges on the shared path have a weight of 0 in the A^* algorithm and will be the first selected.

The bidirectional ALT algorithm finds a shortest path much more quickly than previous algorithms because both forward and reverse searches follow paths of weight 0 for a time.

Remark 5.1. The algorithm is attracted towards whatever landmarks you use and, therefore, bad landmark selection can attract the search away from the shortest path. □

6 The Reach Algorithm

Another heuristic trick to minimise the size of the graph that must be searched is considered by R. Gutman [5]. The idea is to discard all vertices that cannot possibly lie on the shortest path between s and t.

It is important to note that the reach algorithm is highly impractical as the preprocessing involves calculating the shortest paths between all pairs of vertices. Nevertheless, the theory is still valid and it gives good motivation to consider a more practical version of the algorithm later.

Consider a vertex v on a path P from a to z. We define the *reach* of v with respect to P, $r_P(v)$, as the minimum of the length of the path from a to v along P and the length of the path from v to z along P. The reach of v, $r(v)$, is defined as the maximum value of $r_P(v)$ over all shortest paths P between all pairs of vertices (s, t) on the graph that pass through v.

The reach algorithm for finding the shortest path between s and t works as follows: run Dijkstra's algorithm and every time a vertex v is scanned check if

$$r(v) < dist(s, v) \text{ and } r(v) < dist(v, t). \tag{6.1}$$

If this is true v cannot possibly lie on the shortest path between s and t because its reach isn't big enough (i.e. the maximum value of its distance from s and from t isn't big enough). Thus Dijkstra's algorithm need not select it. We say the search has been *pruned* at v.

Theorem 6.1. *[5] In a weighted graph, $G = (V, E)$, the reach algorithm finds a shortest path between s and t.*

Proof. Let G' be the graph with all vertices for which (6.1) holds true removed. The reach algorithm on G is equivalent to Dijkstra's algorithm on G'.

Let P be a shortest path from s to t in G. Assume that there is at least one vertex (v, say) that lies on P but was pruned by the reach algorithm.

Hence, $r(v) < dist(s,v)$ and $r(v) < dist(v,t)$. But v lies on P and so $r(v) \geq r_P(v) = min\{dist(s,v), dist(v,t)\}$. This is a contradiction. Thus the reach algorithm retains all vertices lying on each shortest path from s to t and will consequently find a path of the same length as the shortest path found by Dijkstra's algorithm. □

This bidirectional version of this algorithm works by alternating between the forward search from s to t and the reverse search from t to s and using (6.1) to eliminate vertices that cannot lie on the shortest path between s and t. As both the forward and reverse searches are equivalent to Dijkstra's algorithm with these vertices removed, Theorem 3.3 can easily be adapted to prove that the bidirectional reach algorithm finds the shortest path between s and t.

The Bounded Reach Algorithm

Let $\overline{r}(v)$ be an upper bound on $r(v)$ and $\underline{dist}(v,w)$ be a lower bound on $dist(v,w)$. The bounded reach algorithm is the same as the reach algorithm except that at every iteration of the bounded reach algorithm we check if

$$\overline{r}(v) < \underline{dist}(s,v) \text{ and } \overline{r}(v) < \underline{dist}(v,t).$$

If this is true, then, because $r(v) \leq \overline{r}(v)$ and $dist(v,w) \leq \underline{dist}(v,w)$ for all v and w, one has

$$r(v) < dist(s,v) \text{ and } r(v) < dist(v,t),$$

and hence v cannot possibly lie on the shortest path between s and t as before.

The pruning conditions for the bounded reach algorithm don't discard vertices on a shortest path and so the proof that the bounded reach algorithm finds a shortest path between two vertices is the same as Theorem 6.1.

If ∞ is used as the upper bound for the reach of each vertex then the bounded reach algorithm becomes Dijkstra's algorithm.

The bidirectional bounded reach algorithm is known as `RE`.

Calculation of the upper and lower reach bounds required for the algorithm (considered in [5, Section 5] and [6, Section 5]) are quite complicated and will only be briefly outlined here. The Euclidean distance between two vertices can be used as a lower bound for $dist(\cdot, \cdot)$ if $dist(\cdot, \cdot)$ is the travel distance between vertices. Upper bounds for the reach of a vertex are very complicated to compute. The general idea is to run Dijkstra's algorithm starting at each vertex v until some given termination condition is reached. This creates a shortest path 'tree' consisting of the shortest paths from v to its nearest vertices. This is initially done for vertices close to s and t. This information is then used to compute reach bounds for vertices slightly further away. This process is repeated until reach bounds have been computed for the desired number of vertices.

7 Combining the Reach and `ALT` Algorithms

The reach and A^* algorithms can be easily combined. When a vertex v is about to be selected by the A^* algorithm we check if $r(v) < d_s(v)$ and $r(v) < \pi_t(v)$. If true, we prune the search at v. It is possible to do this because $\pi_t(v)$ is a lower bound for $dist(v, t)$ and at this point $d_s(v) = dist(s, v)$. This combined algorithm finds a shortest path because, as we have shown before, the pruning conditions will only disregard vertices not on the shortest path.

The bidirectional version of this algorithm works like the bidirectional A^* algorithm (symmetric or consistent) combined with the pruning conditions. Again it finds a shortest path because the bidirectional A^* algorithm finds a shortest path and the pruning conditions don't effect shortest paths.

We call the combination of the bidirectional `ALT` algorithm and the `RE` algorithm `REAL`. For the graph of the North American road network (con-

taining almost 30 million vertices), `REAL` finds the shortest path between two random vertices in less than four milliseconds while scanning fewer than 2000 vertices on average [6].

8 Performance of Algorithms

To give some idea of the efficiency of each of these algorithms we will quote some results from [4] and [6]. The following table lists each algorithm alongside the average time taken to find the shortest path between two random vertices in the graph of the San Francisco Bay Area with 330,024 vertices.

The A^* algorithm uses the Euclidean distance as its potential function. The `ALT` and `REAL` algorithms each use 16 landmarks. The bidirectional `ALT` uses the consistent approach with average potential functions.

Algorithm	**Time (in ms)**
Dijkstra	82.4
Bidirectional Dijkstra	59.8
A^*	135.2
`ALT`	12.8
Bidirectional `ALT`	11.6
`RE`	1.17
`REAL`	0.45

Table 1: A comparison of algorithms.

Note that the A^* algorithm seems to perform much worse than Dijkstra's algorithm. This is because of the time taken to estimate the distance from each vertex to t using the potential function.

Although we cannot know exactly how MapQuest works, it is clear that the `REAL` algorithm we have derived is very efficient and would provide a suitable solution to the MapQuest problem.

References

[1] S. Robinson, Mapping Magic. *http://www.siam.org/siamnews/09-04/mapping.htm*

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

[3] K.H. Rosen, *Discrete Mathematics and Its Applications, Fourth Edition.*

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

[5] R. Gutman, Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. Algorithm engineering and experimentation: sixth annual international workshop, 2004*

[6] A.V. Goldberg, H. Kaplan and R.F. Werneck, Reach for A^*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the eighth workshop on algorithm, engineering and experiments (ALENEX 06), SIAM, 2006.*