Finding shortest paths in the presence of orthogonal obstacles using a combined L_1 and link metric

M. de Berg, M. van Kreveld, B.J. Nilsson, M.H. Overmars

RUU-CS-90-20
April 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Finding shortest paths in the presence of orthogonal obstacles using a combined L_1 and link metric

M. de Berg, M. van Kreveld, B.J. Nilsson, M.H. Overmars

Technical Report RUU-CS-90-20
April 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

Finding Shortest Paths in the Presence of Orthogonal Obstacles Using a Combined L_1 and Link Metric*

Mark de Berg†
markdb@cs.ruu.nl

Marc van Kreveld†
marc@cs.ruu.nl

Bengt J. Nilsson‡
bengt@dna.lu.se

Mark H. Overmars†
markov@cs.ruu.nl

Abstract

The problem of computing shortest paths in obstacle environments has received considerable attention recently. We study this problem for a new metric that generalizes the L_1 metric and the link metric. In this combined metric, the length of a path is defined as its L_1 length plus some non-negative constant C times the number of turns the path makes.

Given an environment of n axis parallel line segments and a target point, we present a data structure in which an obstacle free shortest rectilinear path from a query point to the target can be computed efficiently. The data structure uses $O(n \log n)$ storage and its construction takes $O(n^2)$ time. Queries can be performed in $O(\log n)$ time, and the shortest path can be reported in additional time proportional to its size.

1 Introduction

In this paper we discuss a variation on the problem of finding the shortest path between two points in an environment consisting of obstacles that are not allowed to be crossed by the path. This problem has numerous applications, for example in robotics, circuit design and Geographical Information Systems (GIS).

*This research was supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM). Work of the first author was supported by the Dutch Organization for Scientific Research (N.W.O.). The research was done while the third author was visiting Utrecht University.A preliminary version of this paper will be presented at SWAT '90.

†Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

‡Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden.

Several researchers have studied instances of this problem. In [Sha78] the shortest path problem between two points inside a simple polygon is considered. In [WPW74, LPW79, SS86] the shortest path problem in the more general environment of polyhedral obstacles is discussed. In [LL81, CKV87] the problem is studied in the L_1 metric.

Most solutions to these problems first find a 'critical graph' which contains the shortest path between the source and the target. Dijkstra's algorithm [Dij59] or some other shortest path-finding algorithm is then applied to the graph.

In some cases more efficient solutions have been developed. Such methods solve the problem more directly instead of transforming it to a graph problem. In [LP84, dRLW89] it is shown that the shortest paths in environments of vertical line segment barriers or non-intersecting axis parallel rectangles are monotone with respect to one of the axes. This is used to give an efficient solution based on a plane sweep approach.

Mitchell [Mit89] has devised an optimal $O(n \log n)$ algorithm for the shortest path problem in the L_1 metric when the obstacles are disjoint polygons and the paths are rectilinear. Moreover, a data structure is given in which a shortest path from a query point to a fixed target point can be found in $O(\log n)$ time and the complete path can be reported in additional time proportional to its size.

Recently, shortest path problems in the *link metric* have gained considerable attention [dB89, DLS89, Ke89, LPS*87, Su86, MRW90]. In this metric, the length of a path is equal to the number of turns in the path.

In this paper we study the shortest path problem for rectilinear paths in a new metric that generalizes the L_1 metric and the (rectilinear) link metric. In this *combined metric*, the length of a path is its length in the L_1 metric plus some non-negative constant C times the number of turns in the path. Hence, for $C = 0$ the combined metric is equal to the L_1 metric, whereas for $C = \infty$ the combined metric is equal to the link metric. Such a combined metric is realistic in many applications. For example, when a mobile robot that travels along a straight line wants to make a turn, it has to slow down. Hence, the time it takes to travel the path depends not only on the length of the path, but also on the number of turns the robot has to make.

We provide a data structure such that the shortest path from a query point to some fixed target point can be computed in $O(\log n + k)$ time. Here k is the size of the resulting path, i.e., the number of links of the path. The structure uses $O(n \log n)$ storage and can be built in $O(n^2)$ time. The obstacle environment in our problem consists of n axis parallel line segments. Unlike [CKV87, LP84, LL81, Mit89, dRLW89], we allow the segments to intersect.

The basic idea to solve the problem is the following. First we construct a weighted rectilinear visibility graph on the $2n$ end points of the segments and the target point. (Here a point p 'sees' another point q if there is a rectilinear path from p to q, that does not cross an obstacle, consisting of at most two links.) The length of a path in this graph corresponds to the L_1 length of a path. Then the graph is altered such

that it accounts for our combined metric. Dijkstra's algorithm [Dij59] is used to find the shortest path from every node in the graph that corresponds to an end point of a segment to the target node t.

Next we show that for any point p in the plane there is a shortest path to t which, with at most one turn, connects p to some end point of an obstacle segment. This induces a subdivision of the plane such that if two points are in the same region, then they have to be connected to the same end point. Because this subdivision can have quadratic size, we have to represent it implicitly. This is done by considering different starting directions separately. For a fixed starting direction, an efficient implicit representation of the corresponding subdivision can be given.

Once we have the structure, queries are performed as follows. Perform an (implicit) point location to locate the region in which the query point lies. Connect the query point to the end point corresponding to this region and follow the path in the shortest path graph.

The rest of this paper is organized in five sections. In section 2 we give a number of useful results in connection with shortest paths. In section 3 the construction of our modified visibility graph is described. Section 4 discusses the problem of finding the correct end point to go to from the source point. In section 5 we state our main result, and in the concluding section we give a brief overview of our results and discuss some open problems.

2 The Geometry of Shortest Paths

In this section we will do some preliminary work on which our solution is based.

We are given a set S of n axis parallel line segments (the *obstacles*), a *target point* t, and a non-negative real C, that is the cost of making one turn.

DEFINITION 2.1 A polygonal chain $s_1 s_2 \ldots s_m$ between two points p and q is called a *rectilinear path* between p and q if and only if each link s_i of the path is axis parallel.
A rectilinear path between p and q is called *legal* if and only if it does not intersect the interior of any of the line segments in S.
A rectilinear path between p and q is called a *simple step* between p and q if and only if it is legal, and consists of at most two links.
p and q are said to *see* each other if and only if there is a simple step between p and q.

In the following we only consider legal rectilinear paths, so we omit the words 'legal' and 'rectilinear'.

DEFINITION 2.2 A path $P = s_1 \ldots s_m$ between two points p and q has *length* $C \cdot (m-1) + \sum_{i=1}^{m} |s_i|$, where $|s_i|$ denotes the length of s_i. The *distance* between p and q is the minimum of the lengths over all paths between p and q.

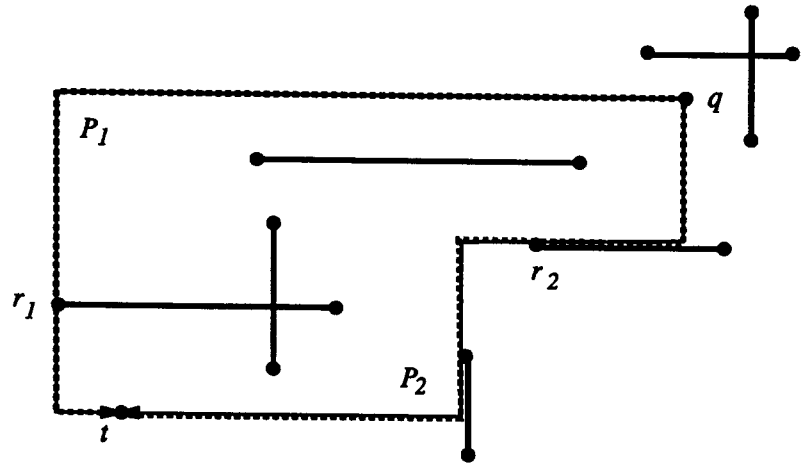

Figure 1: For $C = \infty$, the shortest path from q to t is P_1, for $C = 0$ it is P_2. P_1 first takes a simple step to r_1 and P_2 takes a simple step to r_2.

Observe that the shortest path between two points is not necessarily unique. To see this, consider an obstacle-free environment. A shortest path is traced along either boundary segment pair of the rectangle defined by the two points.

Also notice that if $C = \infty$, then the shortest path in the combined metric will be the shortest path in the link metric, and if $C = 0$, then it is the shortest path in the L_1 metric. See Figure 1 for an illustration of these definitions.

Now we can give the basic lemma on which our solution is based.

LEMMA 2.1 *For any point p in the plane, there exists a shortest path $P = s_1 \dots s_m$ between p and t such that any link s_i, with $1 < i < m$, of the path touches some end point of a line segment of S.*

PROOF: Let P' be a shortest path from p to t. We will transform P' into P, such that the cost will not increase, and without intersecting a line segment of S. Let $s_i = \overline{p_i p_{i+1}}$ $(1 < i < m)$ be the first segment of P' that does not contain any end point of a line segment in S. Assume without loss of generality that s_{i-1} is directed upward and s_i rightward.

If s_{i+1} is directed downward, then the segment s_i can be moved downward, thus shortening the segments s_{i-1} and s_{i+1}, until either s_i touches an end point of a line segment in S, or one of the segments s_{i-1} or s_{i+1} becomes redundant. In the latter case we have reduced the cost of the path, a contradiction with P' being a shortest path.

If s_{i+1} is directed upward, then the segment s_i can be moved upward, thus making s_{i-1} longer and s_{i+1} shorter with the same amount, until either s_i touches

an end point, or s_{i+1} has length zero. In the latter case, s_{i+1} becomes redundant, contradicting the optimality of P'.

Thus we can move s_i such that it contains an end point without increasing the length of the path. This can be repeated with the rest of the path, until every segment s_i $(1 < i < m)$ contains an end point. □

The lemma has the following useful consequence. Suppose we know for every end point of the line segments in S which is the best next end point to go to (including going to the target) on its way to the target. Suppose we also know for each end point the distance to the target. Then a shortest path from an arbitrary query point q to the target can be found in the following way. First take a simple step to that end point p of a line segment in S, such that the sum of the distance from q to p and the distance from p to t is minimized. Then follow the path from p to t.

There is one difficulty with this approach. It only works when the length function is additive in the sense that the length of a path from q to t via p is equal to the sum of the length of the path from q to p and the length of the path from p to t. Unfortunately, the link distance, and thus our length function, does not have this property. The problem is that when the length of the path between q and p is added to the length of the path between p and t, then the turn that might be made at p is not counted, whereas it is counted in the length of the path between q and t. This problem is circumvented as follows. Duplicate every end point p of the line segments in S, such that there is a horizontal version p_h, and a vertical version p_v. If a path arrives at p with a horizontal link, then it arrives at p_h, otherwise it arrives at p_v. It is not allowed to leave p_h vertically or p_v horizontally. But it is allowed to go from p_v to p_h and vice versa, at the cost of one turn. With this extension, the length function becomes additive.

So now we want to find for a query point q the end point p such that the sum of the distance from q to p plus the distance from p to t is minimized. The approach we use to solve this is the *locus approach*: subdivide the plane into regions such that if two points are in the same region then they have to be connected to the same end point. This subdivision, however, can have quadratic size (see Figure 2), so we have to store it implicitly. In order to be able to do this, we consider the eight different possibilities of leaving p with a simple step seperately. Suppose we are only allowed to leave p by a 'left–down' simple step, i.e., the path moves from the query point to the left in the plane, makes a turn and continues downward until it reaches an end point of an obstacle or the target. (For example, the simple step from q to r_1 in Figure 1 is a left–down simple step, whereas the step to r_2 is down–left.) Thus p can only see points that are to its left and below it. The crucial observation that makes an efficient implicit representation of the subdivision possible for a fixed starting step is the following:

LEMMA 2.2 *Suppose that only the left–down simple step is allowed, that p can see two end points e_1 and e_2, and that the path via e_1 has lower cost. Then for any point p' that can see e_1 and e_2, the path via e_1 has lower cost.*

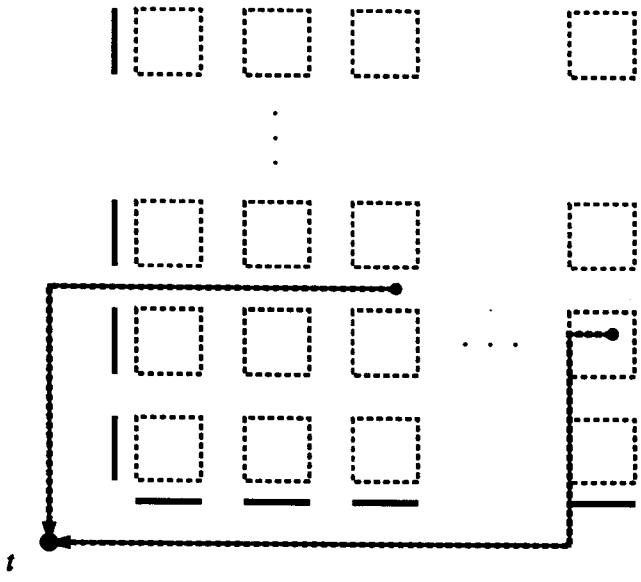

Figure 2: Example showing the quadratic worst case complexity of the subdivision. The bold segments are the obstacles and the dotted squares are regions of the subdivision.

PROOF: Follows from the additivity of the length function for paths. □

This lemma does not imply that the 'left–down subdivision' has subquadratic size. In fact, it can still be quadratic. However, using this lemma we can give an implicit representation of the left–down subdivision that requires subquadratic storage. How this is done is described in section 4.

3 Constructing the Shortest Path Graph

The (rectilinear) visibility graph of the set $S \cup \{t\}$ is defined as the graph $G = (V, E)$ where V is a set of $2n + 1$ nodes each corresponding to an end point of a segment in S or to t. The set E of edges is defined by the visibility property, that is, there exists an edge between nodes of the visibility graph if and only if the points corresponding to the nodes see each other. (Recall that two points see each other if and only if there is a simple step connecting the two points that does not cross an obstacle.) Furthermore, we assign to each edge a weight that is the length of the corresponding simple step in the L_1 metric. Thus, by Lemma 2.1, a shortest path between two nodes in G corresponds to a shortest path in the L_1 metric between the two points corresponding to these nodes.

We wish to modify the visibility graph G to take into account the cost for making turns. The modified visibility graph G^* is constructed from G by duplicating the

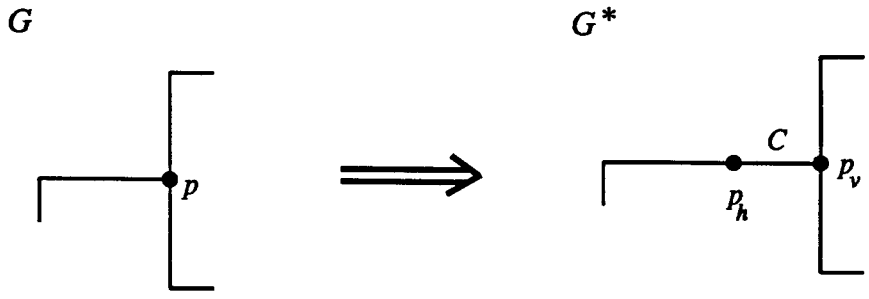

Figure 3: Modification of the Visibility Graph.

nodes that correspond to the end point of an obstacle. We now have two nodes for each end point p of an obstacle, an *h-node* p_h and a *v-node* p_v. There can never be edges between two h-nodes or between two v-nodes, only between an h-node and a v-node (or between an h- or v-node and the target node t). Edges incident on an h-node represent horizontal path links to the corresponding end point. Edges incident on a v-node represent vertical path links. There is an edge between an h-node p_h and a v-node r_v if and only if there is a simple step between p and r that leaves p horizontally and r vertically. Observe that it is possible that there is an edge between p_h and r_v and also between p_v and r_h. We let the weight of the edge be the L_1 distance between the points plus C (the cost of making one turn). Between two nodes p_h and p_v corresponding to the same end point p of a segment there is an edge of weight C. This edge accounts for the turn that has to be made on a path that reaches p horizontally and leaves p vertically. See Figure 3 for an illustration of this modification of the visibility graph.

From the modified visibility graph we can easily construct the single target shortest path graph from the target t using Dijkstra's algorithm [Dij59]. This graph enables us to answer shortest path queries where the source point is some end point of a segment. Such a query can be answered in time proportional to the number of links in the resulting path, and the graph uses linear storage in the number of nodes.

Next it is shown how the modified visibility graph can be constructed in time $O(n \log n + |E|)$. This is done by two plane sweeps. In the first sweep, in which a vertical scan line is moved from left to right, the edges are computed between h-nodes and v-nodes such that the end point corresponding to the h-node is to the left of the end point corresponding to the v-node. The second sweep, which is identical except that it moves from right to left, computes the other edges, between h-nodes and v-nodes such that the h-node is to the right of the v-node.

In the first sweep, a vertical scan line is moved from left to right over the plane, halting at every end point and at the target point. The algorithm maintains two data structures. The first structure is a one-dimensional range tree T_R on the y–

coordinates of the points to the left of the scan line that can still be connected by a right–down or right–up simple step to a point to the right of the scan line. The second structure is a binary search tree T_S on the y–coordinates of the (horizontal) obstacle segments currently intersecting the scan line. When an end point p is encountered the following actions are performed. First, we search with p_y in T_S for the y–coordinates y_a and y_b of the segment above and the segment below p. Now for points q to the left of the scan line, there is en edge from q_h to p_v if and only if q is stored in T_R and $y_b \leq q_y \leq y_a$. By searching with the range $[q_b : q_a]$ in T_R, these points q can be found in $O(\log n + k)$ time, where k is the number of points in the range. After reporting the edges to p_v we have to update the two trees. If p is the left end point of a horizontal segment s, then we insert s into T_S; if p is the right end point of a horizontal segment s, then we delete s from T_S. In both cases we also insert p into T_R. If p is the end point of a vertical segment s, then we first report the edges to the other end point p' of s in the same way as we reported the edges to p. Note that no point to the left of s can reach a point to the right of s by a right-down or right-up simple step. Hence, T_R has to be updated: all the points with y–coordinate in the range $[p_y : p'_y]$ have to be removed from the tree. Finally, p and p' are inserted in T_R. It is easily seen that this algorithm runs in time $O(n \log n + k)$, where k is the number of reported edges. Notice that the weights of the edges are easily computed in constant time per edge.

The second sweep is performed in a similar fashion. It only remains to add the edges between the h- and v-nodes of the same end point, which is trivially done in linear time. Once the modified visibility graph has been computed, the single target shortest path graph by Dijkstra's algorithm takes $O(n^2)$ time. This leads to:

LEMMA 3.1 *The Single Target Shortest Path graph for the combined metric of a set of n axis parallel obstacle segments and a fixed target point can be computed in $O(n^2)$ time. The graph has size $O(n)$.*

4 Finding Initial Steps

Our next goal is to build a subdivision of the plane where each region is associated with an end point of a segment, namely the end point which yields the shortest path to the target for each query point q that lies in that region. As was noted before (refer to Figure 2), such a subdivision may require quadratic storage. However, by considering different ways of leaving q separately, we will be able to give an efficient implicit representation of this subdivision. Thus we have eight 'initial step finding' structures, one for each possible simple step with which we can leave q. Each structure again consists of two structures, one for finding the region in which to look for the appropriate connecting end point and the other for choosing the correct end point among a set of possible ones.

So, let us fix a simple step with which we will leave the query point, say the 'left–down' simple step. In the following subsections, we describe how the subdivision

corresponding to a left–down initial step can be stored implicitly. The (seven) other initial steps can be handled similarly.

4.1 The Interval Finding Structure

When we want to start from the query point q with a left–down step, we need to be able to tell how far to the left the first link can go. In other words, we want to know the first obstacle that we hit when we move to the left from q. For this problem, a structure exists that uses linear storage and in which the first obstacle that is hit by a horizontal ray can be found in $O(\log n)$ time (see [EOS84]). This structure can be built in $O(n \log n)$ time.

LEMMA 4.1 ([EOS84]) *The first obstacle to the left of the query point can be found in $O(\log n)$ with a structure that uses $O(n)$ storage. This structure can be built in $O(n \log n)$ time.*

4.2 The Slab Tree

The query with the source point in the structure of Lemma 4.1 results in a horizontal interval, which defines the region in which end points should be considered. This interval is used as the query object in the structure to be described next. This structure, which we call the *slab tree*, is an implicit representation of the subdivision corresponding to an initial left–down step. To simplify the description, we assign colours to each end point of an obstacle and we give the region from which we should go to a certain end point the corresponding colour. The special colour *white* is used when no end point can be reached at all.

The idea of the implicit colouring is as follows. Consider the intersection of a vertical line with the (coloured) subdivision. The colouring of this line tells us were to go when the source point lies on that line. But by Lemma 2.2, the colouring of this line gives us even more information: it tells us what the best end point to the left of the line is for points to the right of it. In other words, if the interval (that is found with the 'interval finding' structure) of a source point intersects this line in a red region, then the best end point to go to that lies to the left of the line is the red end point. Of course, the best end point need not lie to the left of this line. Therefore we do the following. We divide the plane in vertical *slabs* and we colour the right boundaries of these slabs with the colours of end points in that slab. (The fact that we only use colours of end points in the slab is crucial for the reduction in storage.) Then, for a query interval, we have to select a number of slabs that together cover the interval. The right boundaries of these slabs are all intersected by the interval and thus give us a number of colours. Of the corresponding end points, we have to select the best one.

We next give a more precise description of the slab tree.

Draw vertical lines through all end points of the line segments. This results in a

number of *elementary slabs*. Associate the elementary slabs from left to right with the leaves of a balanced binary tree. Every internal node δ corresponds to a slab which is the union of the elementary slabs that are associated with the leaves of the subtree rooted at δ. We will paint the right bounding line of the slab of δ with the colours of end points that lie in the slab. (Points lying on the boundary between two slabs belong to the leftmost of these two slabs.) This results in a list ordered on y-coordinate, such that for any query interval that crosses the slab completely, the 'best' end point in this slab can be found by searching in this list. This list is called the *associated list* of δ. Although one colour can appear more than once, the size of an associated list is linear in the number of end points in the corresponding slab. This can be proved by showing that the enumeration of the m colours on a line forms an $(m, 2)$ Davenport-Schinzel sequence. It is well known that such a sequence has linear size (see e.g. [HS86]). Hence, the slab tree uses $O(n)$ space at every level of the tree which sums up to $O(n \log n)$ space in total. In Figure 4, an example of a slab tree is given.

A query in the slab tree is performed in the following way. The query object is a horizontal interval that defines the region in which the end point via which we have to go to the target must be contained. (This interval has been found with the 'interval finding' structure of Lemma 4.1.)

Move with the interval down the slab tree to the node δ where the path to its left end point and right end point split. From here follow the path to the left end point of the interval and for every node where we go left, search in the associated structure of its right son to find a colour. Similarly, follow the path from node δ to the right end point of the interval and for every node on the path where we go right, search in the associated structure of its left son to find a colour. Searching in the associated structures is done with the y-coordinate of the interval. In this way we search in the associated structures of $O(\log n)$ nodes whose slabs together cover the horizontal query interval. Thus the query time is $O(\log^2 n)$, which can be reduced to $O(\log n)$ by applying fractional cascading ([CG86]) to the slab tree. (It is straightforward to apply this technique, because all associated lists are ordered on y-coordinate, and when we search, we always search with the same y-coordinate in the associated structures.) Of the $O(\log n)$ colours found, choose the one that gives the shortest path to the target. This colour represents the end point of an obstacle segment which will yield the shortest path to the target if the path must start with a left–down step. It can easily be decided which end point results in the shortest path, because for each end point we have the distance to the target, and we can calculate the distance from the source to any end point by adding C to the L_1 distance.

Next we show how the slab tree can be built efficiently. Let the *mask* of a node δ be defined as the projection of the union of all vertical line segments in the slab of δ. Thus, the mask of a slab represents the 'inaccessible area' to the left of the slab. It is stored in a linear list. Masks are only used for the construction of the slab

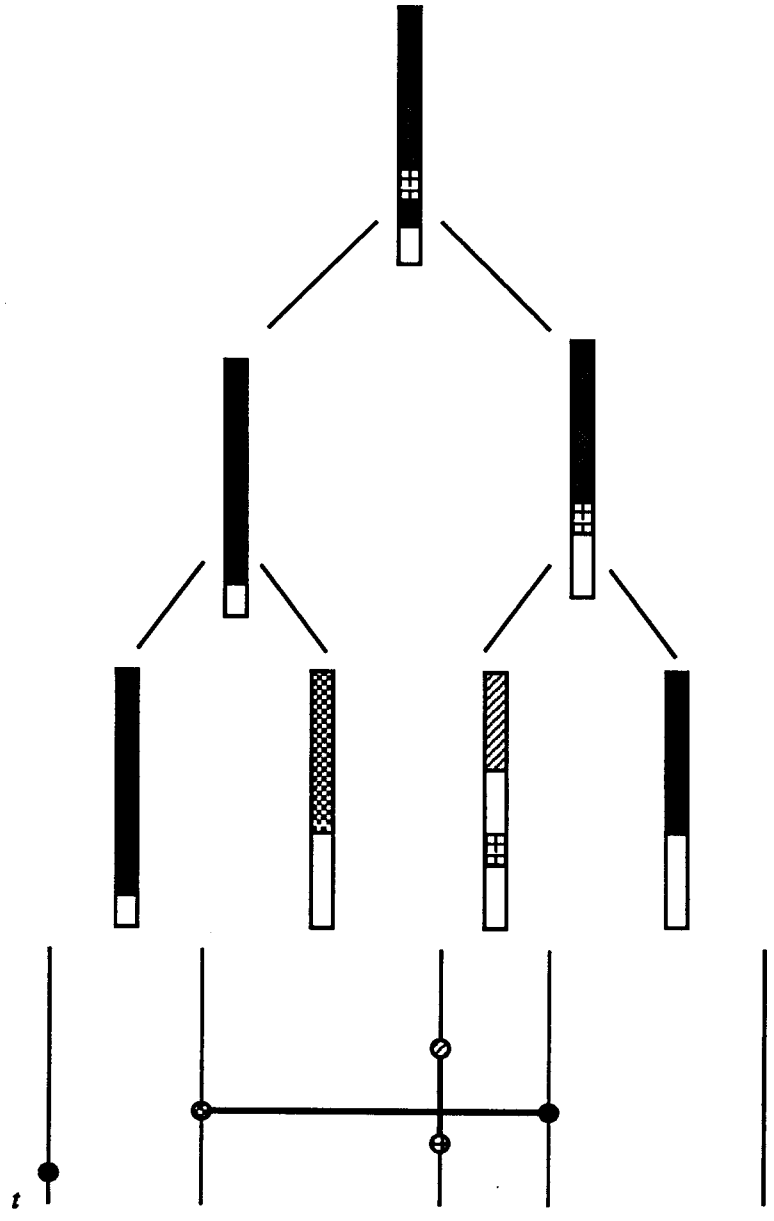

Figure 4: An example of a slab tree for the left–down simple step. The colour white corresponds to areas where no end point can be reached (with a left–down simple step).

tree. Hence, they can be removed when the construction of the slab tree has been completed.

First the skeleton of the slab tree is built. Then the associated list and the masks of the nodes are computed. This is done bottom up, i.e., we start at the leaves and work our way up to the root. The mask of a leaf is trivial to compute and the associated lists of the leaves can be computed in $O(n \log n)$ time in total using a plane sweep.

Now consider two nodes δ_l and δ_r that have the same father δ. Because we are working bottom up, we can assume that we have the associated lists and the masks of δ_l and δ_r. From this information we have to compute the associated list and the mask of δ. The mask of δ is easy to obtain: it is just the union of the masks of its two sons and, hence, it can be computed by a straightforward merge. The associated list of δ is also not hard to compute: consider a point x on the right boundary of the slab corresponding to δ. Assume that this point is coloured green in δ_r. If the mask of δ_r contains x then clearly no end point in δ_l can be reached from x. Hence, point x remains green. Now assume that x is not contained in the mask of δ_r and let the point on the right boundary of δ_l with the same y-coordinate as x be coloured red. Then x may have to be coloured red. Deciding whether the red end point is better than the green one can be done in constant time because we know for each end point its distance to the target. Summarizing, the associated list of δ is computed in the following way. Filter the associated list of δ_l by the mask of δ_r and merge this filtered list with the associated list of δ_r, using the shortest path graph to decide what is the result of merging two colours.

This building algorithm takes time linear in the total size of all associated lists and all masks. We already noted that the size of an associated list of a node is linear in the number of end points in the corresponding slab. The size of a mask is easily seen to be linear in this number as well. Hence, the algorithm takes $O(n)$ time at every level of the tree and $O(n \log n)$ time in total. The application of fractional cascading does not influence the bound on the preprocessing. This leads to:

LEMMA 4.2 *The slab tree can be built in $O(n \log n)$ time, it uses $O(n \log n)$ storage and queries in the slab tree take time $O(\log n)$.*

5 The Main Result

In this section we give an overview of the preprocessing and the query algorithm, and we state our main result.

The preprocessing of our algorithm runs as shown in the following description.

Algorithm	*Preprocess*(S, t, C)
Step 1	Build the modified visibility graph of $S \cup \{t\}$ for the constant C
Step 2	Compute the Single Target Shortest Path graph of this graph
Step 3	**for** each initial simple step **do**
Step 3.1	Build the interval finding structure
Step 3.2	Build the slab tree
	endfor
end *Preprocess*	

By Lemma 3.1, steps 1 and 2 of the preprocessing algorithm take $O(n^2)$ time. Steps 3.1 and 3.2 take $O(n \log n)$ time by Lemmas 4.1 and 4.2 and, because there are eight different simple steps, step 3 takes $O(n \log n)$ time in total. The Single Target Shortest Path Graph, as well as the interval finding structures, use $O(n)$ storage and the slab trees use $O(n \log n)$ storage. Hence, the total amount of storage used is $O(n \log n)$.

Queries are performed according to the following description.

Algorithm	*Query*(s)
Step 1	**for** each initial simple step **do**
Step 1.1	Do an interval finding query with s in the appropriate structure
Step 1.2	Do a query with the interval thus found in the appropriate slab tree
	endfor
Step 2	Of the eight end points found, select the best one
Step 3	Follow the path from s via this end point in the STSP graph to the target t
end *Query*	

Interval finding queries take $O(\log n)$ time, see Lemma 4.1. Queries in the slab tree also take $O(\log n)$ time by Lemma 4.2. Since there are eight different initial steps for which the queries have to be done, steps 1 and 2 of the query algorithm take $O(\log n)$ time. Note that the distance of the query point to t (i.e. the length of a shortest path) can be reported now. Reporting the path itself is done in step 3 and takes time proportional to the number of links of the path.

We can now state the main theorem of our paper.

THEOREM 1 *For a set of n axis parallel obstacle segments, there exists a data structure of size $O(n \log n)$, such that the distance in the combined metric from a query point to a fixed target point can be found in $O(\log n)$ time. A shortest path can be reported in $O(k)$ additional time, where k is the size of the resulting path. The data structure can be built in $O(n^2)$ time and it uses $O(n \log n)$ storage.*

6 Conclusion

We have provided a data structure to compute the shortest path to a fixed target in an obstacle environment consisting of n possibly intersecting axis parallel line segments. The length of a path was measured in a new metric that generalizes the L_1 and the link metric. The query time is $O(\log n + k)$, and the structure uses $O(n \log n)$ storage. Here n is the number of obstacle segments and k is the number of links in a shortest path.

A number of open problems do remain. A first interesting problem is the construction of a data structure that allows both the source and the target to be query points. It would also be nice if the cost of a turn could be specified in the query. Finally, in this paper we studied axis parallel obstacles and paths. It would be interesting to consider arbitrary polygonal obstacles and paths. Here the length of a path could be defined as its Euclidean length plus some extra term which is dependent on the number of turns as well as on the angle of the turns. No results for this general model are known to date.

References

[CG86] B. CHAZELLE, L.J. GUIBAS. Fractional Cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[CKV87] K. CLARKSON, S. KAPOOR, P. VAIDYA. Rectilinear Shortest Paths through Polygonal Obstacles in $O(n \log^2 n)$ Time. In *Proc. 3rd ACM Symp. on Computational Geometry*, pages 251–257, 1987.

[dB89] M. DE BERG. *On Rectilinear Link Distance.* Technical Report RUU-CS-89-13, Department of Computer Science, Utrecht University, 1989.

[Dij59] E.W. DIJKSTRA. A Note on Two Problems in Connection with Graphs. *Numer. Math.*, 1:269–271, 1959.

[DLS89] H.N. DJIDJEV, A. LINGAS AND J. SACK. An $O(n \log n)$ time Algorithm for Computing the Link Center in a Simple Polygon. In B. Monien and R. Cori (Eds.) *Proceedings STACS '89*, Lect. Notes in Comp. Science 349, Springer Verlag, pages 96–107, 1989.

[EOS84] H. EDELSBRUNNER, M.H. OVERMARS, R. SEIDEL. Some Methods of Computational geometry Applied to Computer Graphics. *Computer Vision, Graphics, and Image Processing*, 28:92–108, 1984.

[dRLW89] P.J. DE REZENDE, D.T. LEE, Y.F. WU. Rectilinear Shortest Paths with Rectangular Barriers. *Journal of Discrete and Computational Geometry*, 4:41–53, 1989.

[HS86] S. HART AND M. SHARIR. Nonlinearity of Davenport-Schinzel Sequences and of Generalized Path Compression Schemes. *Combinatorica*, 6:151–177, 1986.

[Ke89] Y. KE. An Efficient Algorithm for Link Distance Problems. In *Proc. 5th ACM Symp. on Computational Geometry*, pages 69–78, 1989.

[LL81] R.C. LARSON, V.O. LI. Finding Minimum Rectilinear Distance Paths in the Presence of Barriers. *Networks*, 11:285–304, 1981.

[LP84] D.T. LEE, F.P. PREPARATA. Euclidean Shortest Paths in the Presence of Rectilinear Barriers. *Networks*, 14:393–410, 1984.

[LPS*87] W. LENHART, R. POLLACK, J. SACK, R. SEIDEL, M. SHARIR, S. SURI, G. TOUSSAINT, S. WHITESIDES AND C. YAP. Computing the Link Center of a Simple Polygon. In *Proc. 3rd ACM Symp. on Computational Geometry*, pages 1–10, 1987.

[LPW79] T. LOZANO-PEREZ, M.A. WESLEY. An Algorithm for Planning Collision-free Paths among Polyhedral Obstacles. *Communications of the ACM*, 22:560–570, 1979.

[MRW90] J.S.B. MITCHELL, G. ROTE AND G. WÖGINGER. Computing the Minimum Link Path Among a Set of Obstacles in the Plane. To appear in *Proc. 6th ACM Symp. on Computational Geometry*, 1990.

[Mit89] J.S.B. MITCHELL. An Optimal Algorithm for Shortest Rectilinear Paths Among Obstacles in the Plane. In *Abstracts of the First Canadian Conference on Computational Geometry*, page 22, 1989.

[Sha78] M.I. SHAMOS. *Computational Geometry.* PhD thesis, Yale University, New Haven, CN, 1978.

[SS86] M. SHARIR, A. SCHORR. On Shortest Paths in Polyhedral Spaces. *SIAM Journal of Computing*, 15(1):193–215, 1986.

[Su86] S. SURI. *Minimum Link Paths in Polygons and Related Problems.* Ph. D. thesis, Johns Hopkins University, 1986.

[WPW74] G.E. WANGDAHL, S.M. POLLACK, J.B. WOODWARD. Minimum-Trajectory Pipe Routing. *Journal of Ship Research*, 18:46–49, 1974.