xy-Monotone Path Existence Queries in a Rectilinear Environment*

Gregory Bint† Anil Maheshwari† Michiel Smid†

Abstract

Given a planar environment consisting of n disjoint axis-aligned rectangles, we want to query on any two points and find whether there is a north-east monotone path between them. We present preprocessing and query algorithms which translate the geometric problem into a tree traversal problem and present a corresponding tree structure that gives us $O(n \log n)$ construction time, $O(n)$ space, and $O(\log n)$ query time.

1 Introduction

Consider a closed planar environment which consists of n disjoint axis-aligned rectangular obstacles. We want to query this environment on any two points s and t and determine whether a rectilinear north-east monotone path exists between them. In this paper, we mean rectilinear in the sense that all segments of a path are axis-aligned and that adjacent segments of a path meet at right angles. By north-east monotone, we mean that each path segment extends either above or to the right of the end of the previous segment (Figure 1).

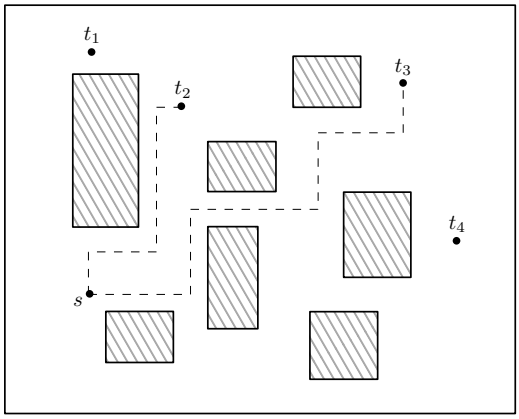

Figure 1: An environment with 7 obstacles. From s, there are north-east monotone paths to t_2 and t_3, but not to t_1 or t_4.

This work is inspired by a single point query algorithm for finding shortest paths in a similar environment by Rezende, Lee, and Wu [7]. Given a fixed point s, they preprocess the environment to allow for queries on any t. Our main contribution is to allow both s and t to be specified at query time, extracting only the information about north-east monotone path existence. We summarize our work with the following theorem.

Theorem 1 *Given a planar environment consisting of n disjoint axis-aligned rectangular obstacles, we can construct, in $O(n \log n)$ time, a data structure with size $O(n)$ with which we can query for the existence of a north-east monotone path between any two query points in $O(\log n)$ time.*

The remainder of this paper details a data structure and query method to satisfy this theorem. Section 2 covers some preliminary terminology and path construction techniques. Section 3 contains the main contribution of this work, showing how we can precalculate so-called *shared paths* and use them to answer monotone path existence queries. As our overall query requires the ability to perform a planar point location query, Section 4 reviews a particularly suitable method. Section 5 brings together the complete query algorithm with some discussion on possible extensions. Finally, we conclude in Section 6.

2 Preliminaries

In this section, we will review some construction techniques and lemmas presented in Rezende, Lee, and Wu. The proofs appear in their original paper.

Definition 1 *Let a path π be defined by a sequence of points $p_1, p_2, ...p_k$, then π is an xy-*path *if, for every adjacent pair of points $p_i, p_{(i+1)}$, either $p_{(i+1)}$ is directly above p_i (i.e. $p_{(i+1)_x} = p_{i_x}$ and $p_{(i+1)_y} > p_{i_y}$) or $p_{(i+1)}$ is directly to the right of p_i (i.e. $p_{(i+1)_x} > p_{i_x}$ and $p_{(i+1)_y} = p_{i_y}$). Following similar rules, we can define $x(-y)$-paths, $(-x)y$-paths, and $(-x)(-y)$-paths. Observe that any such path is rectilinear.*

Definition 2 *A rectilinear north-east monotone path is an xy-path.*

If we assume that no two adjacent segments of a path are co-linear, we observe that any vertical (horizontal)

*This research was supported in part by the NSERC-USRA program

†School of Computer Science, Carleton University {gbint, anil, michiel}@scs.carleton.ca

line can intersect such a path through at most a single point of one horizontal (vertical) segment or lie colinearly with at most one vertical (horizontal) segment.

When constructing xy-paths, we can *prefer* a direction of path extension by always travelling in a particular direction unless we need to route around an obstacle.

Definition 3 *An* x-preferred xy-path, π, *is an* xy*-path which extends east, in the* $+x$*, direction whenever possible. Let* o *be an obstacle with sides* $left(o)$*,* $top(o)$*,* $right(o)$ *and* $bottom(o)$*. If* π *encounters* $left(o)$*, a vertical segment is added which continues north (in the* $+y$ *direction) to the vertex located at the incidence of* $left(o)$ *and* $top(o)$*. From there, a horizontal segment is added, which resumes travelling east along* $top(o)$*, and beyond, until the next obstacle is encountered (see Figure 2).*

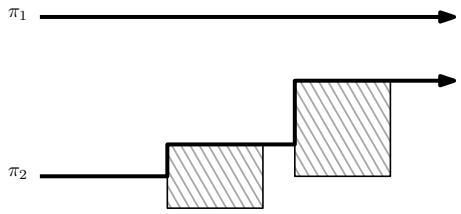

Figure 2: Two x-preferred xy-paths, one without, one with obstacles.

In a similar way, π may be a y-preferred xy-path, which travels north whenever possible, only travelling east when moving around an obstacle. $x(-y)$-paths, $(-x)y$-paths, and $(-x)(-y)$-paths can also be constructed with a preference for either of their two component directions. For example, an $x(-y)$-path can be x-preferred or $(-y)$-preferred.

Definition 4 *Given a point* s*, if we extend both an* x*-preferred* xy*-path and a* $(-x)$*-preferred* $(-x)y$*-path from* s*, the area above the union of these paths is called the* y-region *of* s*. In a similar fashion, the* x-region of s *is the area to the right of the union of the* y*-preferred* xy*-path and the* $(-y)$*-preferred* $x(-y)$*-path rooted at* s*.*

Definition 5 *The* xy-region of s *is the intersection of the* x*-region of* s *and the* y*-region of* s *(Figure 3).*

Using these definitions, Rezende, Lee, and Wu give the following lemma which is of particular interest.

Lemma 2 *There is a rectilinear north-east monotone path from* s *to* t *if and only if* t *lies within the* xy*-region of* s*.*

From their lemma, and from the construction of the xy-region of s, we derive the following additional lemma.

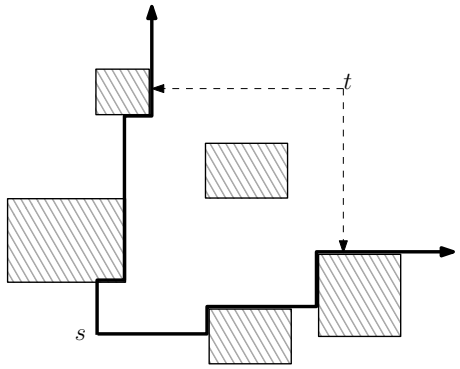

Figure 3: The complete xy-region of s with point t contained within it.

Lemma 3 *If* t *is in the* xy*-region of* s*, then, disregarding all obstacles, a vertical line through* t *must intersect the* x*-preferred* xy*-path rooted at* s *somewhere below* t*, and a horizontal line through* t *must intersect the* y*-preferred* xy*-path rooted at* s *somewhere left of* t*.*

Rezende, Lee, and Wu's algorithm constructs the xy-region of s, which is then further refined into a rectangular subdivision. When a query point t is given, they perform a point location on t within the rectangular subdivision. Assuming t lies within the xy-region of s, their subdivision stores enough information to allow them to construct a rectilinear north-east monotone path between the two points. By Lemma 2, we know that such a path is possible. Such a path is also a shortest path, which was the goal of their work, however we are only interested in the existence of it.

3 Shared Paths

Having seen how xy-paths are constructed around obstacles, we now turn our attention to how multiple paths will route around the same obstacle.

Lemma 4 *Given an* x*-preferred* xy*-path* π *which meets* $left(o)$ *for some obstacle* o*, any other* x*-preferred* xy*-path* π' *which meets* $left(o)$ *will have the same structure as* π *from the corner of* $left(o)$ *and* $top(o)$ *and beyond.*

The proof is apparent directly from construction. It should be clear that this lemma holds for other $(\pm x)(\pm y)$-paths, with either component direction as the preferred direction.

3.1 Shared Path Tree

We can use these *shared paths* to aid us in identifying the xy-region of any s during query time. As the construction of x-regions and y-regions are similar, we will consider only the y-region case. To do so, we will

pre-calculate paths from the top-left vertex of each obstacle. Then, during query time, all that remains is to find the first obstacle that an x-preferred xy-path from s would hit. From that obstacle, we can follow the pre-calculated xy-path from its top-left vertex. Specifically, these shared paths will be stored in a tree, which is constructed in the following way.

Imagine a bounding box that contains all of the obstacles and all of the valid query range for s and t. Along the right-hand side of this box is a vertical line segment obstacle, labeled o_0, whose top-left vertex is v_0. All x-preferred xy-paths must eventually meet this obstacle.

We sort the remaining obstacles from right to left, top to bottom, according to their top-left vertices so that we have a sequence of obstacles $o_1, ..., o_n$ with corresponding top-left vertices $v_1, ..., v_n$. Notice that o_n is the leftmost obstacle, and that v_1 is the rightmost vertex to the left of v_0.

Our tree, $\mathcal{T}$, will store $v_0, ..., v_n$, with v_0 at the root. Then, processing $v_1, ..., v_n$ in order starting with v_1, we process each v_i in the following way. Let $seg(v_i)$ be a horizontal line segment starting at v_i, traveling east in the $+x$-direction, and let o_j be the obstacle impacted on the left side by $seg(v_i)$. Note that $j < i$. From o_j, we obtain a pointer to v_j, which must already exist in $\mathcal{T}$. We insert v_i into $\mathcal{T}$ as a child of v_j. Notice that $seg(v_i)$ is the line segment $(v_{i_x}, v_{i_y}) - (v_{j_x}, v_{i_y})$ and that $v_{i_y} \leq v_{j_y}$. As a result of the insertion method, any path of vertices in the tree will be ordered with respect to their x components. See Figure 4 for an example environment and corresponding $\mathcal{T}$.

Reconstructing an xy-path based on $\mathcal{T}$ is simple: given a pointer to a particular vertex v_{p_1} in the tree, if $v_{p_1}, v_{p_2}, ..., v_{p_k}$ is the sequence of vertices in the path from v_{p_1} to the root of the tree, then $seg(v_{p_1}), seg(v_{p_2}), ..., seg(v_{p_k})$ is the sequence of horizontal line segments that make up the xy-path of v_{p_1}[1].

Construction of $\mathcal{T}$ takes $O(n \log n)$ time. We first sort the vertices in the order given above. Next, for each of the n obstacles, we maintain a line segment intersection sweepline to find, in $O(\log n)$ time, the obstacle which will be hit by a horizontal line segment leaving the top-left vertex. Insertion into $\mathcal{T}$ takes only $O(1)$ as we acquire the parent pointer directly from the sweepline structure. $\mathcal{T}$ has size $O(n)$ as each top-left obstacle vertex is inserted exactly once.

3.2 Querying the Shared Path Tree

From Lemma 3, we see that it is sufficient to show that t is in the y-region of s by testing that t is above the x-preferred xy-path rooted at s. Here, we refer only to that portion of the y-region which is to the east of s, since no other part of the y-region can contribute to the xy-region of s. To that end, we will further assume that t is also east of s.

The first step towards identifying the y-region of s is to identify the obstacle which a horizontal ray leaving s in the $+x$ direction would hit. We label that obstacle as o_s. We can use a point location data structure to find this obstacle and return the pointer v_s, corresponding to an entry in $\mathcal{T}$. Section 4 explores this in more detail, but for now it suffices to assume that we can acquire v_s.

With v_s, we know that the first segment of the x-preferred xy-path rooted at s is the horizontal segment defined by $(s_x, s_y) - (v_{s_x}, s_y)$. The remaining segments of the path are already stored in the tree, and so a simple query method would be as follows.

Assume that t is north-east of s, otherwise the query result is 'no'. Let $v(t)$ be the vertical line through t. If $v(t)_x$ is within the x-interval of the first horizontal line segment, then we test and return whether t is above it and we are done.

Otherwise, let $v_{s_1}, v_{s_2}, ..., v_{s_k}$ be the path through $\mathcal{T}$ where $v_{s_1} = v_s$ and v_{s_k} is the root of $\mathcal{T}$. We test $v(t)$ against each $seg(v_{s_i})$ in order from 1 to k. If $v(t)_x$ is not within the x-interval of a line segment, we advance to the next segment by following the parent pointer of v_{s_i}. If it is, we test and return whether t is above that segment, and we are done. Since we construct our environment such that the root node of $\mathcal{T}$ is farther right than any other input, there must be some segment which $v(t)$ intersects. See Figure 4 for an example.

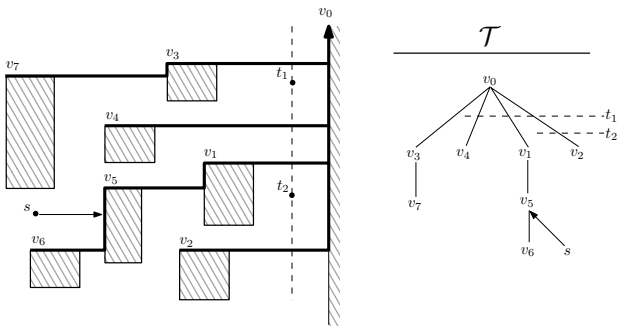

Figure 4: An environment and corresponding shared path tree showing how s, t_1, and t_2 interact. Notice that there is a north-east monotone path from s to t_1, but not from s to t_2.

Performance of this query method is dependent on the height of $\mathcal{T}$. If we consider the case where all obstacles are arranged in an ascending staircase such that a path starting with the leftmost obstacle hits every remaining obstacle on its way east, we see that $\mathcal{T}$ has a height and query time of $O(n)$, which is not sufficient for our theorem.

[1] We assume that the vertical segments of an x-preferred xy-path have no width, and as a result, any vertical line through such a path must impact some horizontal segment. As we will see, we are only interested in performing vertical line tests on these paths, so we can disregard the vertical path segments.

3.3 Augmenting the Shared Path Tree

In order to achieve $O(\log n)$ query time on $\mathcal{T}$, we will need to augment it. The augmentation we will use is a method first discussed by Cole and Vishkin [3]. This method was later illustrated by Narasimhan and Smid [6] in a manner very similar to our own use.

In brief, the vertices in the tree are processed into *groups* which have the property that we can follow a path from any vertex to the root by looking at only $O(\log n)$ groups.

The augmentation adds the following information to $\mathcal{T}$. For every vertex v, define m to be the number of vertices in the subtree of v. We define *l-value(v)* to be $\lfloor \log m \rfloor$. We define a *group* to be a path of vertices that share the same *l-value*. The head of this group is the vertex closest to the root and is called the *group parent* for all vertices in the group, a pointer to which is stored at every v as *gpar(v)*.

From the definitions given, observe the following properties about groups. Every leaf will have an *l-value* of 0 and will belong to a group consisting only of itself, as its parent's subtree size and thus its parent's *l-value* must be ≥ 2 and ≥ 1, respectively. We also see that the root of the tree will have an *l-value* of $\lfloor \log n \rfloor$. An example is given in Figure 5.

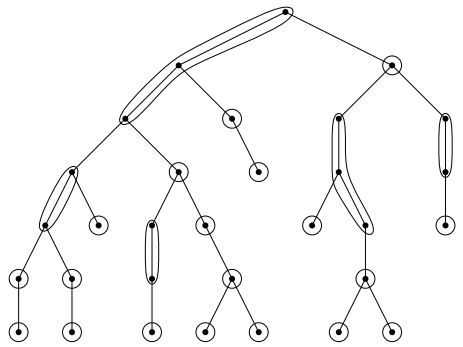

Figure 5: A binary tree with grouped nodes indicated.

Note that *l-values* can only increase as we consider vertices closer to the root, and that groups must be paths. Thus, we can traverse from any v to the root of the tree by following $O(\log n)$ *gpar* pointers.

With the groups configured, we will further augment $\mathcal{T}$ at each group parent by creating an ordered *group array* containing all the vertices of its corresponding group. This array permits us to perform a binary search on the group while examining the group parent. By copying the children pointers in the appropriate order, according to their orientation along the group path, we can create this array in $O(n)$ time over all groups.

The group number and group parent pointers of each vertex, and the group arrays associated with each group parent, can be calculated with a simple post-order traversal of the tree in $O(n)$ time and requires only $O(1)$ extra space per vertex. Note that since each vertex appears in exactly one group array, the total size of all group arrays is $O(n)$.

3.4 Querying the Augmented Shared Path Tree

Querying the augmented tree has the same goal as in Section 3.2: to identify the eastern y-region of s and determine if t is within it.

The initial parts of the query are performed identically. Again we assume that we have a point location data structure that allows us to identify the first obstacle to the right of s, labeled o_s, and which gives us the corresponding pointer v_s into $\mathcal{T}$. From this, we can define and test the first horizontal line segment of the x-preferred xy-path at s.

Recall that $v(t)$ is the vertical line through t. If $v(t)$ does not intersect that first horizontal segment, then we follow v_s into $\mathcal{T}$. We immediately jump to the group parent of v_s, labeled $gpar(v_s)$. If $v(t)$ is to the left of (or at) $gpar(v_s)$, then we perform a binary search on the array of vertices stored there, each of which correspond to a horizontal line segment in the environment. $v(t)$ must intersect one of these segments, and we test and return whether $v(t)$ is above that segment.

If $v(t)$ is to the right of $gpar(v_s)$, then we follow $gpar(v_s)$'s parent pointer, which brings us to some vertex in the next group towards the root of $\mathcal{T}$ and repeat the same procedure, jumping to the group parent, testing the array contents there, and so on. Since we construct our environment such that the root vertex of $\mathcal{T}$ must be farther right than any other input, there must be a group such that $v(t)$ intersects one of its constituent vertices.

In performing this query, we need to test at most $O(\log n)$ group parent pointers. In one group, we will also need to perform a binary search on the array stored there, for a total query time of $O(\log n)$, as required by our theorem.

4 Finding s

Before we can use $\mathcal{T}$ to identify the y-region of s, we need to identify the first obstacle that an x-preferred xy-path leaving s will impact.

4.1 Planar Subdivision

Step 1 is to create a horizontal subdivision of the environment into planar rectangles. Conceptually, this is accomplished by extending a horizontal ray from each obstacle vertex away from the obstacle until it strikes another obstacle or the environment boundary. Every such ray bisects the space it travels through into two regions resulting in a subdivision of size $O(n)$ (Figure

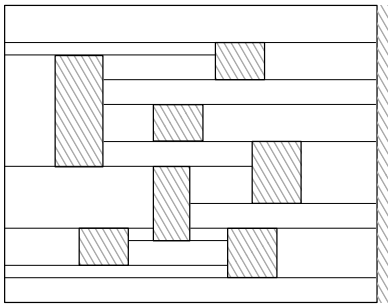

Figure 6: example of a planar rectangular subdivision of a set of obstacles in enclosing environment. note the imaginary obstacle along the right-hand side.

6). We call each non-obstacle face of the subdivision a *cell*.

Lemma 5 *The right-hand boundary of a cell is defined by a single obstacle.*

The subdivision can be constructed using the same horizontal sweepline used to build $\mathcal{T}$. With each cell, we store the appropriate pointer into $\mathcal{T}$ based on the obstacle that the cell sees to its right.

4.2 Point Location Query

To find the cell that a particular point falls into, we will use a *Skewer Tree* [4], a data structure by Edelsbrunner, Haring, and Hilbert specifically for point location in a collection of axis-aligned, non-overlapping rectangles.

Construction of a skewer tree follows a divide and conquer approach. For a set, S, of rectangles, we place a vertical line l through the median x-value. We divide S into three subsets: S_1 is the set of rectangles that lie strictly to the left of l, S_2 is the set of rectangles intersected by l, and S_3 is the set of rectangles that lie strictly to the right of l, so that $|S_1| + |S_2| + |S_3| = |S|$. We create a node n_S in the skewer tree which contains the definition of l, the size of S_2, and a balanced tree containing the rectangles of S_2, sorted by y-value.

If S_1 is non-empty, we recurse on S_1, attaching the resulting subtree as the left child of n_S. Similarly, if S_3 is non-empty, we recurse on S_3, attaching the resulting subtree as the right child of n_S.

Every rectangle appears exactly once in the skewer tree, as it is attributed only to the first node whose l intersected it and not passed down to deeper levels of recursion. The skewer tree requires $O(n)$ space and $O(n \log n)$ construction time.

The query time is a bit more interesting. We start at the root node and check if our query point s is contained within one of the rectangles stored there, which takes $O(\log n)$ time. If it is not, we compare l with s_x and decide whether we will next check the left or right subtree. We repeat these steps at every level of the tree until the rectangle containing s is found. The height of the skewer tree is $O(\log n)$, so we need to make at most that many queries for a total query time of $O(\log^2 n)$ time.

We can improve the query time to $O(\log n)$ with *Fractional Cascading* [1, 2, 5, 8]. To help illustrate how, we will refer to the outer nodes of the Skewer tree as the *line tree*, and the trees of rectangles attached to each node of the line tree as *rectangle trees*.

Every path through the line tree represents a sequence of arrays of sorted values. Each array is queried over the same range of keys, defined by the interval of y-values covered by the bounding box of the environment.

Considering a single path through the line tree for now, we will store extra pointers in each of the rectangle trees so that a query on one tree can return not just the successor to the search key value in that tree, but in the next tree in the path as well. If this mechanism is implemented in every rectangle tree, then we can answer our query in every tree of the path by performing a standard binary tree search on the root rectangle tree in $O(\log n)$ time, and then continuing by walking along $O(\log n)$ pointers through the path, each taking $O(1)$ time.

Because the line tree is a binary tree, we need to store two sets of additional pointers in each rectangle tree: one to use if we follow a line tree node's left child, and one to use if we follow its right child. See Chazelle and Guibas [1, 2] for details on how these extra pointers can be developed in linear time.

5 The Complete Algorithm

We now have all the tools we need to solve our query problem.

5.1 Construction

The construction phase of the algorithm involves building both the shared path data structure and a point location data structure.

When discussing shared path data structures, we have primarily considered obstacles to the east of our query point, and the resulting x-preferred xy-paths. We will relabel that data structure as $\mathcal{T}_E$. We also need to consider obstacles to the north and the associated y-preferred xy-paths, which we do by creating a second data structure labeled $\mathcal{T}_N$. The algorithms as written in Section 3.1 for $\mathcal{T}_E$ can easily be adapted for $\mathcal{T}_N$.

While performing the plane-sweeps needed to construct $\mathcal{T}_E$ and $\mathcal{T}_N$, we can also produce the rectangular planar subdivisions used by the skewer trees.

Total construction time and space for $\mathcal{T}_E$, $\mathcal{T}_N$, and the skewer trees, is $O(n \log n)$ and $O(n)$, respectively, as required by our theorem.

5.2 Query

When a query is made, we are given s and t. Assume that t is actually north-east of s, otherwise the query result is 'no'.

We first perform a point location query on s, identifying the obstacle to its right. We then follow the procedure in Section 3.4 to determine whether t is in the y-region of s.

Using the obstacle above s, we identify the x-region of s, and again follow the procedure in Section 3.4 to determine if it contains t. If both return true, then we know that t is in the xy-region of s, and so by Lemma 2, there is a north-east monotone path from s to t. The total query time is $O(\log n)$, as required by our theorem.

6 Conclusion

In this paper we have discussed a method for deciding whether there exists a north-east monotone path between any two query points in the plane. We developed the concept of shared paths and showed a method for storing them in a tree, and for augmenting that tree to allow for quick query time. We also reviewed a suitable point location query method. Together, these methods require $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time.

One problem which remains open is the following. In the event that a path is found to be possible, we would like to report one such path in $O(\log n + k)$ time, where k is the number of segments in the reported path, and is within a constant factor of the minimum number of segments among all such paths.

References

[1] B. Chazelle and L. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[2] B. Chazelle and L. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1:163–191, 1986.

[3] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[4] H. Edelsbrunner, G. Haring, and D. Hilbert. Rectangular point location in d dimensions with applications. *The Computer Journal*, 29(1):76–82, 1986.

[5] K. Mehlhorn and S. Nher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[6] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge University Press, New York, NY, USA, 2007.

[7] P. Rezende, D. Lee, and Y. Wu. Rectilinear shortest paths in the presence of rectangular barriers. *Discrete & Computational Geometry*, 4:41–53, 1989.

[8] M. Smid. Rectangular point location and the dynamic closest pair problem. In W.-L. Hsu and R. Lee, editors, *ISA'91 Algorithms*, volume 557 of *Lecture Notes in Computer Science*, pages 364–374. Springer Berlin / Heidelberg, 1991.