Bicriteria Rectilinear Shortest Paths among Rectilinear Obstacles in the Plane*†

Haitao Wang

Department of Computer Science, Utah State University, Logan, UT, USA
haitao.wang@usu.edu

Abstract

Given a rectilinear domain $\mathcal{P}$ of h pairwise-disjoint rectilinear obstacles with a total of n vertices in the plane, we study the problem of computing bicriteria rectilinear shortest paths between two points s and t in $\mathcal{P}$. Three types of bicriteria rectilinear paths are considered: minimum-link shortest paths, shortest minimum-link paths, and minimum-cost paths where the cost of a path is a non-decreasing function of both the number of edges and the length of the path. The one-point and two-point path queries are also considered. Algorithms for these problems have been given previously. Our contributions are threefold. First, we find a critical error in all previous algorithms. Second, we correct the error in a not-so-trivial way. Third, we further improve the algorithms so that they are even faster than the previous (incorrect) algorithms when h is relatively small. For example, for computing a minimum-link shortest s-t path, the previous algorithm runs in $O(n \log^{3/2} n)$ time while the time of our new algorithm is $O(n + h \log^{3/2} h)$.

1998 ACM Subject Classification I.3.5 Computational Geometry and Object Modeling, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases rectilinear paths, shortest paths, minimum-link paths, bicriteria paths, rectilinear polygons

Digital Object Identifier 10.4230/LIPIcs.SoCG.2017.60

1 Introduction

Let $\mathcal{P}$ be a *rectilinear domain* with a total of h holes and n vertices in the plane, i.e., $\mathcal{P}$ is a multiply-connected region whose boundary is a union of n axis-parallel line segments, forming $h + 1$ closed polygonal cycles (i.e., h holes plus an outer boundary). A simple rectilinear polygon is a special case of a rectilinear domain with $h = 0$. A *rectilinear* path is a path consisting of only horizontal and vertical line segments.

For a rectilinear path π, we define its *length* as the total sum of the lengths of the segments of π, and we define its *link distance* as the number of edges of π (each edge is also called a *link*). We use the *measure* of π to refer to both its length and its link distance. For any two points s and t in $\mathcal{P}$, a *shortest rectilinear path* from s to t is a rectilinear path connecting s to t in $\mathcal{P}$ with the minimum length, and a *minimum-link rectilinear path* is a rectilinear s-t path with the minimum link distance. Among all shortest rectilinear s-t paths, one with the minimum link distance is called a *minimum-link shortest s-t path*; among all minimum-link s-t paths, one with the minimum length is called a *shortest minimum-link s-t path*. We define the *cost* of π as a non-decreasing function f of both the length and the link distance of π. We assume that given the number of links of π and the length of π, its cost can be computed

* A full version of the paper is available at https://arxiv.org/abs/1703.04466.
† This research was supported in part by NSF under Grant CCF-1317143.

in constant time. Depending on the context, the measure of π may also refer to its cost. A *minimum-cost* path from s to t is a rectilinear s-t path in $\mathcal{P}$ with the minimum cost.

All the three types of paths discussed above (i.e., minimum-link shortest paths, shortest minimum-link paths, and minimum-cost paths) are called *bicriteria shortest paths*. In order to differentiate between "bicriteria shortest paths" and "shortest paths", we will use *optimal paths* to refer to these bicriteria shortest paths. Since some observations and algorithmic schemes may be applicable to all three types of optimal paths, unless otherwise stated, a statement made to "optimal paths" should be applicable to all three types of optimal paths.

In this paper, we study the problem of computing all three types of optimal paths between two points s and t in $\mathcal{P}$. Their one-point and two-point queries are also considered.

Previous Work. The following results are applicable to all three types of optimal paths.

Yang et al. [24] first presented an $O(nr + n \log n)$ time algorithm, where r is the number of extreme edges of $\mathcal{P}$ (an edge e is *extreme* if its two adjacent edges lie on the same side of the line containing e; $r = \Omega(n)$ in the worst case). Later, Yang et al. [25] proposed an algorithm of $O(n \log^2 n)$ time and $O(n \log n)$ space and another algorithm of $O(n \log^{3/2} n)$ time and space; Chen et. al. [2] improved it to $O(n \log^{3/2} n)$ time and $O(n \log n)$ space.

The *one-point optimal path query problem*, where s is the source and t is a query point, was also studied. Based on the algorithm of Yang et al. [25], Chen et. al. [2] built a data structure of $O(n \log n)$ size in $O(n \log^{3/2} n)$ time such that for each query point t, the measure of the optimal s-t path can be computed in $O(\log n)$ time and an actual path can be output in additional time linear in the number of edges of the path. For simplicity, in the following, when we say that the query time of a data structure for finding a path is $O(g(n))$, we mean that the measure of the path can be computed in $O(g(n))$ time and an actual path can be output in additional time linear in the number of edges of the path.

The *two-point optimal path query problem*, i.e., both s and t are query points, was also studied by Chen et. al. [2], where a data structure of $O(n^2 \log^2 n)$ size was built in $O(n^2 \log^2 n)$ time such that each two-point query can be answered in $O(\log^2 n)$ time.

Our Results. We provide a comprehensive study on these problems and our contributions are threefold.

First, we show that all above algorithms in the previous work are incorrect. More specifically, we find a critical error in the algorithm of Yang et al. [25]. Since the results of Chen et. al. [2] are all based on the method of Yang et al. [25], they are not correct either. A similar error also appears in [24]. Note that the technique of Chen et. al. [2], which follows the similar idea in [4] for computing L_1 shortest paths in arbitrary polygonal domains, would work if it was based on a correct algorithm (for example, it still works in our new algorithm).

Second, we fix the error of Yang et al. [25] in a not-so-trivial way. However, the complexities are not the same as before for all three types of optimal paths. Specifically, for computing a minimum-link shortest path, our corrected algorithm runs in $O(n \log^{3/2} n)$ time and $O(n \log n)$ space (with the help of the technique of Chen et. al. [2] to reduce a factor of $\log^{1/2} n$). For the other two types of optimal paths, however, the complexities have one more $O(n)$ factor, i.e., $O(n^2 \log^{3/2} n)$ time and $O(n^2 \log n)$ space.

Third, we further improve the algorithms in the way that the complexities only depend on h, in addition to $O(n)$. For computing a minimum-link shortest path, our algorithm runs in $O(n + h \log^{3/2} h)$ time and $O(n + h \log h)$ space. For computing other two types of optimal paths, our algorithm runs in $O(n + h^2 \log^2 h)$ time and $O(n + h^2 \log h)$ space. We also obtain data structures for one-point and two-point queries, and the results are summarized

Table 1 Summary of our data structures on one-point and two-point optimal path queries. Note that $\log^2 h \cdot 4^{\sqrt{\log h}} = O(h^\epsilon)$ for any $\epsilon > 0$. The "Prep. Time" stands for "Preprocessing Time".

Types of Paths		One-Point Queries	Two-Point Queries	
Min-Link Shortest	Prep. Time	$O(n + h\log^{3/2} h)$	$O(n + h^2\log^2 h)$	$O(n + h^2\log^2 h 4^{\sqrt{\log h}})$
	Space	$O(n + h\log h)$	$O(n + h^2\log^2 h)$	$O(n + h^2\log h 4^{\sqrt{\log h}})$
	Query Time	$O(\log n)$	$O(\log n + \log^2 h)$	$O(\log n)$
Shortest Min-Link	Prep. Time	$O(n + h^2\log^{3/2} h)$	$O(n + h^3\log^2 h)$	$O(n + h^3\log^2 h 4^{\sqrt{\log h}})$
	Space	$O(n + h^2\log h)$	$O(n + h^3\log^2 h)$	$O(n + h^3\log h 4^{\sqrt{\log h}})$
	Query Time	$O(\log n + \log^2 h)$	$O(\log n + \log^3 h)$	$O(\log n + \log^2 h)$
Minimum-Cost	Prep. Time	$O(n + h^2\log^{3/2} h)$	$O(n + h^3\log^2 h)$	$O(n + h^3\log^2 h 4^{\sqrt{\log h}})$
	Space	$O(n + h^2\log h)$	$O(n + h^3\log^2 h)$	$O(n + h^3\log h 4^{\sqrt{\log h}})$
	Query Time	$O(\log n + h\log h)$	$O(\log n + h\log^2 h)$	$O(\log n + h\log h)$
Minimum-Link	Prep. Time		$O(n + h^2\log^2 h)$	$O(n + h^2\log^2 h 4^{\sqrt{\log h}})$
	Space		$O(n + h^2\log^2 h)$	$O(n + h^2\log h 4^{\sqrt{\log h}})$
	Query Time		$O(\log n + \log^2 h)$	$O(\log n)$

in Table 1. Note that for two-point queries, we give two data structures for each problem with tradeoff between the preprocessing and the query time. We also consider the two-point query problem for minimum-link paths (without considering the lengths) since the problem was not studied before (but its one-point query problem has been solved, as discussed below).

Our results are particularly interesting when h is relatively small. For example if $h = O(n^{1/2-\epsilon})$ for any $\epsilon > 0$, then for finding a single optimal path of any type, our algorithm runs in $O(n)$ time, and our data structures for the minimum-link shortest path and minimum-link path queries are also optimal.

It is easy to see that the minimum-link shortest paths and the shortest minimum-link paths are special cases of minimum-cost paths, and we discuss them separately mainly because our results for the two special cases are generally better that those for the minimum-cost paths. In fact, as the cost function f is quite general, our algorithm for computing minimum-cost paths may find many applications. We give two examples below.

Polishchuk and Mitchell [19] gave an $O(kn\log^2 n)$ time algorithm for computing a shortest s-t path with at most k links for a given integer k, which improves the $O(kn^2)$ time algorithm in [24]. As indicated in [19], the problem can be solved using any algorithm that can find a minimum-cost path with the cost function defined as $f(a, b) = a$ if $b \leq k$ and $f(a, b) = \infty$ otherwise, where a and b are the length and the link distance of the path, respectively. Partially due to this reason, Polishchuk and Mitchell [19] already suspected that there is a misunderstanding on the algorithms of [2, 25] for computing minimum-cost paths. We thus confirm their suspicion. On the other hand, applying our new (and correct) algorithm for minimum-cost paths can solve the problem in $O(n + h^2\log^{3/2} h)$ time, which is faster than the algorithm in [19] when h is sufficiently small or when k is relatively large.

As a dual problem, finding a minimum-link s-t path with length at most a given value l was also studied in [24], where a worst-case $O(n^2(r + \log n))$ time algorithm was given with r as the number of extreme edges of $\mathcal{P}$. Note that $r \geq h$. The problem can also be solved using any minimum-cost path algorithm by defining the cost function as $f(a, b) = b$ if $a \leq l$ and $f(a, b) = \infty$ otherwise. Hence, applying our algorithm for minimum-cost paths can solve the problem in $O(n + h^2\log^{3/2} h)$, which clearly improves the algorithm of [24].

Other Related Work. If $\mathcal{P}$ is a simple rectilinear polygon (i.e., $h = 0$), then there always exists a rectilinear s-t path that has both the minimum length and the minimum link distance for any s and t in $\mathcal{P}$ [10, 11]. de Berg [10] built a data structure of $O(n \log n)$ size in $O(n \log n)$ time that can find such a path in $O(\log n)$ time for any two-point query. The preprocessing time and space were both reduced to $O(n)$ by Schuierer [21] (with $O(\log n)$ query time).

If $\mathcal{P}$ is a general rectilinear domain with $h \neq 0$, then there may not exist a rectilinear path that is both a minimum-link path and a shortest path [24]. The problems of finding only minimum-link paths or only shortest paths have been studied extensively. Imai and Asano [12] presented an $O(n \log n)$ time and space algorithm for finding a minimum-link s-t path in $\mathcal{P}$, and the space was reduced to $O(n)$ [9, 16, 20]. Recently, Mitchell et al. [17] proposed an $O(n + h \log h)$ time and $O(n)$ space algorithm for the problem, after $\mathcal{P}$ is triangulated (which can be done in $O(n \log n)$ time or $O(n + h \log^{1+\epsilon} h)$ time for any $\epsilon > 0$ [1]). The algorithms in [9, 16, 17] also construct an $O(n)$ size data structure that can answer each one-point minimum-link path query in $O(\log n)$ time.

For computing shortest s-t paths in $\mathcal{P}$, Clarkson et al. [7] gave an algorithm of $O(n \log^2 n)$ time and $O(n \log n)$ space, and as a tradeoff between time and space, they modified their algorithm so that it runs in $O(n \log^{3/2} n)$ time and space [8]. Wu et al. [23] proposed an $O(n \log r + r^2 \log r)$ time algorithm, where r is the number of extreme edges of $\mathcal{P}$, and the algorithm was later improved to $O(n \log r + r \log^{3/2} r)$ time [25]. Mitchell [14, 15] solved the problem in $O(n \log n)$ time and $O(n)$ space, and Chen and Wang [5, 6] reduced the time to $O(n + h \log h)$ after $\mathcal{P}$ is triangulated.

If $\mathcal{P}$ is an arbitrary polygonal domain (i.e., not rectilinear), then the results from [5, 6, 7, 8, 14, 15] are also applicable to finding arbitrary shortest paths under L_1 metric. In addition, the algorithms in [5, 6, 14, 15] can be used to compute an $O(n)$ size data structure so that each one-point L_1 shortest path query can be answered in $O(\log n)$ time. For two-point L_1 shortest path queries, Chen et al. [4] constructed a data structure of size $O(n^2 \log n)$ in $O(n^2 \log^2 n)$ time that can answer each query in $O(\log^2 n)$ time. Recently, Chen et al. [3] reduced the query time to $O(\log n)$ by building a data structure of size $O(n + h^2 \cdot \log h \cdot 4^{\sqrt{\log h}})$ in $O(n + h^2 \cdot \log^2 h \cdot 4^{\sqrt{\log h}})$ time.

To find a minimum-link s-t path between two points s and t in an arbitrary polygonal domain $\mathcal{P}$, Mitchell [18] gave an $O(E\alpha(n) \log^2 n)$ time algorithm, where $\alpha(n)$ is the inverse of Ackermann's function and E is the size of the visibility graph of $\mathcal{P}$ and $E = \Theta(n^2)$ in the worst case. The one-point query problem was also studied in [18].

In the following, unless otherwise stated, a path always refers to a rectilinear path.

Our Techniques. Given two points s and t in the rectilinear domain $\mathcal{P}$, to find an optimal s-t path, the algorithm of Yang et al. [25] first built a "path-preserving" graph G of size $O(n \log n)$ by using the idea of Clarkson et al. [7]. Then, it is shown that G contains an s-t path $\pi_G(s,t)$ that is homotopic to an optimal s-t path $\pi(s,t)$ in $\mathcal{P}$ with the same length, and further, $\pi(s,t)$ can be obtained from $\pi_G(s,t)$ by performing certain "dragging" operations. Motivated by this observation, Yang et al. [25] computed an optimal s-t path by applying Dijkstra's algorithm on G and simultaneously performing the dragging operations. We find a critical error in their way of applying Dijkstra's algorithm. We fix the error by using a "path-based" Dijkstra's algorithm and maintaining some additional information, and we prove that our algorithm is correct. Due to that we need to maintain more information on computing shortest minimum-link paths and minimum-cost paths, our algorithm for them runs slower than that for computing minimum-link shortest paths.

To further reduce the running time (for small h), our main idea is to use a reduced graph G_r of size $O(h \log h)$ instead of G. We show that G_r contains an s-t path $\pi_{G_r}(s,t)$ that

is homotopic to an optimal s-t path $\pi(s,t)$ in $\mathcal{P}$ with the same length, and further, $\pi(s,t)$ can be obtained from $\pi_{G_r}(s,t)$ by performing the dragging operations as in [25] and a new kind of operations, called *through-corridor-path generating operations*. The graph G_r is built based on a corridor structure of $\mathcal{P}$, which was used to find minimum-link paths in [17]. More specifically, we decompose $\mathcal{P}$ into $O(h)$ *junction rectangles* and $O(h)$ *corridors*. Each corridor is a simple rectilinear polygon. Although each corridor may have $\Theta(n)$ vertices, we show that we only need to consider at most four points of each corridor to build the graph G_r. To this end, we make use of the histogram partitions of rectilinear simple polygons [21].

For the one-point queries, Chen et al. [2] "insert" the query point t to the graph G to obtain a set $V_g(t)$ of $O(\log n)$ vertices (called "gateways") of G such that an optimal path can be obtained by performing the dragging operations from the gateways. We follow the similar scheme but on our reduced graph G_r, where only $O(\log h)$ gateways are necessary. Further, we also need to utilize the techniques of Schuierer [21] for simple rectilinear polygons.

For the two-point queries, the approach of [2] inserts both query points s and t to the graph G to obtain a set $V_g(s)$ of $O(\log n)$ gateways for s and a set $V_g(t)$ of $O(\log n)$ gateways for t, so that an optimal s-t path can be obtained by performing dragging operations from these gateways. The query time becomes $O(\log^2 n)$ because every pair of points (p,q) with $p \in V_g(s)$ and $q \in V_g(t)$ needs to be considered. We again use the same scheme but on the graph G_r with only $O(\log h)$ gateways for both s and t, and the query time is $O(\log n + \log^2 h)$. To reduce the query time to $O(\log n)$, we follow the scheme in [3] for two-point L_1 shortest path queries in arbitrary polygonal domains. The main idea is to build a larger graph by adding more vertices to G_r so that $O(\sqrt{\log h})$ gateways are sufficient for each query point.

The rest of the paper is organized as follows. We define some notation in Section 2. In Section 3, we review the algorithm given by Yang, Lee, and Wong [25] (we refer to it as the YLW algorithm), point out the error, and correct it. In Section 4, we further improve the algorithm for finding a single optimal s-t path. Due to the space limit, some details are omitted but can be found in the full paper [22]. Our data structures for the one-point and two-point path queries are also omitted and in the full paper.

2 Preliminaries

In this section, we define some concepts and notation. For any two points p and q of $\mathcal{P}$, if the line segment $\overline{pq}$ is in $\mathcal{P}$, then we say that p is *visible* to q. Consider a vertical line l and a point $p \in \mathcal{P}$. Let p' be the point on l whose y-coordinate is the same as that of p. We call p' the *horizontal projection* of p on l. If p is visible to p', then p is *horizontally visible* to l.

For any two points p and q, we use R_{pq} to denote the rectangle with $\overline{pq}$ as a diagonal. A path in $\mathcal{P}$ is *L-shaped* if it consists of a horizontal segment and a vertical segment (each of them may be empty). A path is *U-shaped* if it consists of three segments s_1, s_2, and s_3 such that s_1 and s_3 are on the same side of the line containing s_2. A path is called a *staircase path* if it does not contain a U-shaped subpath. Note that a staircase path is a shortest path.

Let $\mathcal{V}$ denote the set of all vertices of $\mathcal{P}$. We let $\mathcal{V}$ also include the two points s and t. We review a "path-preserving" graph $G(\mathcal{V})$ on $\mathcal{V}$ [7]. The vertex set of $G(\mathcal{V})$ consists of the points of $\mathcal{V}$ and *Steiner points* on some vertical lines, called *cut-lines*. The cut-lines and the Steiner points are defined as follows. Let v_m be the point of $\mathcal{V}$ with the median x-coordinate. The vertical line l_m through v_m is a *cut-line*. For each point $v \in \mathcal{V}$, if v is horizontally visible to l_m, then the horizontal projection of v on l_m is a *Steiner point*. Let $\mathcal{V}_l$ (resp., $\mathcal{V}_r$) be the points of $\mathcal{V}$ on the left (resp., right) side of l_m. The cut-lines and Steiner points on the left and right sides of l_m are defined on $\mathcal{V}_l$ and $\mathcal{V}_r$, recursively. We use a

binary tree $T(\mathcal{V})$ to represent the above recursive procedure, called *cut-line tree*. Each node $u \in T(\mathcal{V})$ corresponds to a cut-line $l(u)$ and a subset $V(u) \subseteq \mathcal{V}$. If u is the root, then $l(u)$ is l_m and $V(u) = \mathcal{V}$. The left and right subtrees of the root are defined recursively on $\mathcal{V}_l$ and $\mathcal{V}_r$. Hence, $T(\mathcal{V})$ has $O(n)$ nodes and each point of $\mathcal{V}$ can define a Steiner point on at most $O(\log n)$ cut-lines. Therefore, there are $O(n \log n)$ Steiner points in total.

The vertex set of $G(\mathcal{V})$ consists of all points of $\mathcal{V}$ and all Steiner points defined above. The edges of the graph are defined as follows. First, if a point $v \in \mathcal{V}$ defines a Steiner point v' on a cut-line, then $G(\mathcal{V})$ has an edge $\overline{vv'}$. Second, for any two adjacent Steiner points p_1 and p_2 on each cut-line, if they are visible to each other, then $G(\mathcal{V})$ has an edge $\overline{p_1p_2}$.

Clearly, $G(\mathcal{V})$ has $O(n \log n)$ nodes and edges. Each edge of $G(\mathcal{V})$ is either horizontal or vertical, whose weight is the length of the corresponding line segment. The graph $G(\mathcal{V})$ can be built in $O(n \log^2 n)$ time [7, 13, 25]. The following lemma was proved before [7, 13, 25].

▶ **Lemma 1** ([7, 13, 25]). *For any two points p and q in $\mathcal{V}$, if R_{pq} is empty (i.e., R_{pq} is in $\mathcal{P}$), then $G(\mathcal{V})$ contains a staircase path from p to q.*

For any path π in $\mathcal{P}$, let $L_1(\pi)$ denote its length and let $L_d(\pi)$ denote its link distance. For any two points a and b on π, if the context is clear, we often use $\pi(a, b)$ to denote the subpath of π between a and b. For any two points p and q in the plane, we say that q is to the *northeast* of p if q is in the first quadrant (including its boundary) with respect to p.

3 The YLW Algorithm and Our Correction

In this section, we first review the YLW algorithm [25] and then point out the error. Finally, we will fix the error. The YLW algorithm is essentially based on the following observation.

▶ **Lemma 2** (Yang et al. [25]). *For any optimal path π from s to t in $\mathcal{P}$, there is path π_G in $G(\mathcal{V})$ such that $L_1(\pi_G) = L_1(\pi)$ and π_G is homotopic to π (i.e., π_G can be continuously dragged to π without going outside of $\mathcal{P}$).*

We briefly review the proof of Lemma 2 because it will help to understand the algorithm.

Let π be any optimal path from s to t. It is shown (Lemma 2.1 [25]) that π can be divided into a sequence of staircase subpaths, and the two endpoints of each such subpath are in $\mathcal{V}$. Hence, it is sufficient to prove the lemma for any staircase subpath of π. Consider a staircase subpath $\pi(p, q)$ of π with p and q as the two endpoints. We further obtain a *pushed staircase path* as follows. Without loss of generality, we assume q is to the northeast of p and the segment of $\pi(p, q)$ incident to p is horizontal. We push the first vertical segment of $\pi(p, q)$ rightwards until either it hits a vertex of $\mathcal{V}$ or it becomes collinear with the second vertical segment of $\pi(p, q)$. In the latter case, we merge the two vertical segments and keep pushing the merged segment rightwards. In the former case, we push the next horizontal segment upwards in a similar way. The procedure stops until we arrive at the segment incident to q. Let π' denote the resulting path. Observe that $L_1(\pi') = L_1(\pi(p, q))$, π' is homotopic to $\pi(p, q)$, and π' is also a staircase path. π' is called a *pushed staircase path* [25]. Also note that each segment of π' contains at least one vertex of $\mathcal{V}$. There are eight types of pushed staircase paths from p to q depending on which quadrant of p the point q lies in and also depending on whether the first segment of the path incident to p is horizontal or vertical.

The vertices of $\mathcal{V}$ partition π' into subpaths. To prove the lemma, it is sufficient to show the following *claim*: for any subpath $\pi'(p', q')$ of π' between any two adjacent vertices p' and q' of $\mathcal{V}$ on π', there is a path $\pi_G(p', q')$ connecting p' and q' in $G(\mathcal{V})$ with the same length and the two paths are homotopic. Because every segment of π' contains at least one vertex

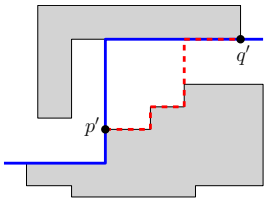

Figure 1 Converting $\pi_G(p', q')$ (the dashed red path) to $\pi'(p', q')$ (the solid blue path between p' and q').

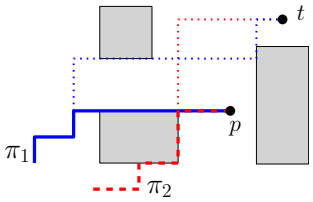

Figure 2 Illustrating a counter example for the YLW algorithm.

of $\mathcal{V}$, $\pi'(p', q')$ must be an L-shaped path. Without loss of generality, we assume q' is to the northwest of p'. If the rectangle $R_{p'q'}$ is empty, then by Lemma 1, the above claim is true. Otherwise, as shown in [25] (Lemma 4.5), there are some points of $\mathcal{V}$ in $R_{p'q'}$ that can be ordered as $p' = v_0, v_1, \ldots, v_t = q'$ with $R_{v_{i-1}v_i}$ being empty and v_i to the northwest of v_{i-1} for each $1 \leq i \leq t$, and further, $\pi'(p', q')$ is homotopic to the concatenation of $\overline{v_{i-1}v_i}$ for all $1 \leq i \leq t$. By Lemma 1, for each $1 \leq i \leq t$, $G(\mathcal{V})$ contains a staircase path connecting v_{i-1} and v_i and the path is in $R_{v_{i-1}v_i}$ (and thus is homotopic to $\overline{v_{i-1}v_i}$). Hence, by concatenating the staircase paths from v_{i-1} to v_i for all $i = 1, 2, \ldots, t$, we obtain a staircase path from p' to q' and the path is homotopic to $\pi'(p', q')$. Note that the staircase path has the same length as $\pi'(p', q')$ since $\pi'(p', q')$ is an L-shaped path. The above claim thus follows.

This proves Lemma 2. The proof actually constructs the path π_G in $G(\mathcal{V})$ corresponding to the optimal path π, and π_G is called a *target path*. Yang et al. [25] also showed that π can be obtained from π_G by applying certain *dragging* operations during searching the graph $G(\mathcal{V})$. Instead of describing the details of the operation (refer to our full paper or [25]), we give some intuition on how π can be obtained from π_G by using the dragging operations. Based on the above constructive proof for Lemma 2, we only need to show that for each L-shaped path $\pi'(p', q')$, it can be obtained from the corresponding staircase path $\pi_G(p', q')$ in $G(\mathcal{V})$. Without loss of generality, we assume that q' is to the northeast of p' and the segment incident to p' in $\pi'(p', q')$ is vertical. Because $\pi_G(p', q')$ is homotopic to $\pi'(p', q')$, we can convert $\pi_G(p', q')$ to $\pi'(p', q')$ as follows (e.g., see Fig. 1). Starting from p', for each horizontal segment of $\pi_G(p', q')$, drag it upwards until either it hits the horizontal segment of $\pi'(p', q')$ or it becomes collinear with the next horizontal segment of $\pi_G(p', q')$. In the former case, we have obtained $\pi'(p', q')$. In the latter case, we continue to drag the new horizontal segment upwards in the same way as before.

The YLW algorithm applies Dijkstra's algorithm using the measure vector $(L_1(\pi), L_d(\pi))$ for a path π. Initially, all vertices of $G(\mathcal{V})$ are in a priority queue Q with measure vectors (∞, ∞) except that the measure vector for s is $(0, 0)$. While Q is not empty, the algorithm removes from Q the vertex p with the smallest measure vector (lexicographically) and advance the paths stored at p to each of p's neighbor q by the dragging operations. Let $\pi(s, q)$ be a path obtained for q. There may be other paths already stored at q and the types of the last staircase subpaths of these paths are also stored (recall that there are eight types of pushed staircase subpaths). The YLW algorithm relies on the following two rules to determine whether the new path $\pi(s, q)$ should be stored at q, and if yes, whether some paths stored at q should be removed. Let $\pi'(s, q)$ be any path that has already been stored at q.

Rule(a) If the measure vectors of $\pi(s, q)$ and $\pi'(s, q)$ are not the same, then discard the one whose measure vector is strictly larger.

Rule(b) If $\pi(s, q)$ and $\pi'(s, q)$ have the same measure vector and of the same type, compare their last segments. If they overlap, discard the path whose last segment is longer.

It is claimed in [25] that once the point t is processed, among all paths stored at t, the one with the smallest measure vector is an optimal s-t path.

The Error. We find that the algorithm is not correct, mainly due to Rule(a). Figure 2 illustrates a counterexample. Assume that both π_1 and π_2 are paths from s to p with $L_1(\pi_1) = L_1(\pi_2)$ and $L_d(\pi_1) + 1 = L_d(\pi_2)$. Thus, the measure vector of π_1 is strictly smaller than that of π_2. According to Rule(a), we should discard π_2. Observe that we can obtain an s-t path from s to t using π_2 without any extra link. However, to obtain an s-t path using π_1, we need at least two more links. Therefore, π_2 can lead to a better s-t path than π_1, and thus, we should not discard π_2. Notice that the reason this happens is that although the measure vector of π_1 is strictly smaller than that of π_2, the last segment of π_2 is shorter than that of π_1 (and thus it may be "freely" dragged upwards higher than that of π_1).

In fact, the most essential reason for this error to happen might be the following. If π is a shortest s-t path, then for any two points p and q in π, the subpath $\pi(p, q)$ of π between p and q is also a shortest path from p to q. However, this may not be the case for minimum-link paths. Namely, if π is a minimum-link s-t path, then it is possible that for two points p and q in π, $\pi(p, q)$ is not a minimum-link path from p to q. Due to this reason, one can verify that the $O(nt + n \log n)$ time algorithm given by Yang et al. [24] for computing optimal paths is not correct either. Indeed, the approach in [24] also applies Dijkstra's algorithm on a graph to search the optimal paths using the measure vectors like $(L_1(\pi), L_d(\pi))$.

3.1 Our New Algorithm

To fix the error, we need to fix Rule(a). We first consider the minimum-link shortest paths. We replace Rule(a) by the following Rule(a_1), but still keep Rule(b). (Recall that $\pi'(s, q)$ denotes any path that has already been stored at q.)

Rule(a_1) Let π_1 be one of $\pi'(s, q)$ and $\pi(s, q)$, and π_2 the other. If $L_1(\pi_1) < L_1(\pi_2)$, or $L_1(\pi_1) = L_1(\pi_2)$ but $L_d(\pi_1) \leq L_d(\pi_2) - 2$, then we discard π_2.

By Rule(a_1), we may need to store two paths π_1 and π_2 at q even if the measure vector of one path is strictly smaller than that of the other, in which case $L_1(\pi_1) = L_1(\pi_2)$ and $L_d(\pi_1) = L_d(\pi_2) \pm 1$. Hence, unlike the YLW algorithm, each vertex q of $G(\mathcal{V})$ may store paths with different measure vectors. Therefore, we cannot apply the same "vertex-based" Dijkstra's algorithm as before. Instead, we propose a "path-based" Dijkstra's algorithm. Roughly speaking, we will process individual paths instead of vertices. Specifically, in the beginning there is only one path from s to s itself in the priority queue Q. In general, as long as Q is not empty, we remove from Q the path π with the smallest measure vector. Assume that the endpoint of π is p. Then, we advance π from p to each of p's neighbors q. If $\pi(s, q)$ is stored at q by our rules (i.e., both Rule(a_1) and Rule(b)), then we (implicitly) insert $\pi(s, q)$ to Q. The algorithm stops once Q is empty. Since we process paths following the increasing measure order, the algorithm will eventually stop. Finally, among all paths stored at t, we return the one with the smallest measure as the optimal solution. The correctness of the algorithm is proved in the full paper.

In terms of the running time, the YLW algorithm maintains at most eight paths at each vertex p of $G(\mathcal{V})$. To see this, due to Rule(b), for each type of staircase paths, p maintains at most one path. In our new algorithm, the paths maintained at p always have the same length but their link distances differ by at most one. Hence, again due to Rule(b), there are at most sixteen paths maintained at p. Clearly, this does not affect both the time and

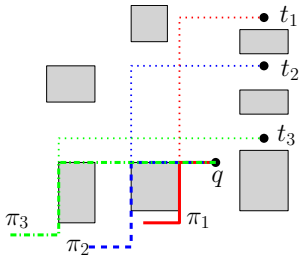

Figure 3 Illustrating an example on why we need Rule(a_2).

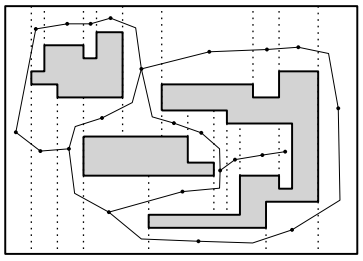

Figure 4 Illustrating the vertical visibility decomposition $VD(\mathcal{P})$ and its dual graph G_{vd}.

the space complexities of the algorithm asymptotically. Thus, the algorithm still runs in $O(n \log^2 n)$ time and $O(n \log n)$ space, as the YLW algorithm.

In addition, using another path-preserving graph $G^*(\mathcal{V})$ of $O(n \log^{1/2} n)$ vertices and $O(n \log^{3/2} n)$ edges [8], Yang et al. [25] proposed another $O(n \log^{3/2} n)$ time and space algorithm (see Section 4.2 of [25]). Further, Chen et al. [2] reduced the space of the algorithm to $O(n \log n)$ with the same $O(n \log^{3/2} n)$ time (similar technique was also used in [4]). By applying the techniques of both [25] and [2] to our new method, we can also obtain an algorithm of $O(n \log^{3/2} n)$ time and $O(n \log n)$ space. We omit the details.

We proceed on the problem of finding a minimum-cost s-t path. Recall that we have a cost function f. For any path π, we use $f(\pi)$ to denote the cost of the path. Our algorithm is the same as above with the following changes. First, the paths π in the priority Q are prioritized by $f(\pi)$. Second, we replace both Rule(a_1) and Rule(b) by the following rule.

Rule(a_2) Let π_1 be one of $\pi'(s,q)$ and $\pi(s,q)$, and π_2 the other. If the last segments of π_1 and π_2 are exactly the same and $f(\pi_1) \leq f(\pi_2)$, then we discard π_2.

We give some intuition on why we use the above rule. Refer to Fig. 3, where there are three paths π_1, π_2, and π_3 from s to q. Let s_i be the last segment of π_i for each $1 \leq i \leq 3$, and we assume that they overlap with $|s_1| < |s_2| < |s_3|$, where $|s_i|$ is the length of each s_i. We also assume that $L_d(\pi_1) = L_d(\pi_2) = L_d(\pi_3)$ and $L_1(\pi_1) > L_1(\pi_2) > L_1(\pi_3)$. In this case, we have to keep all three paths because any of them may lead to the best path from s to t. For example, for each $1 \leq i \leq 3$, the path π_i may lead to the best path from s to t_i. One can generalize the example so that a total of $\Omega(n)$ paths may need to be stored at p. However, $O(n)$ is the upper bound since the last segment of each such path starts from a different vertex of $G(\mathcal{V})$ in the horizontal line through q and there are $O(n)$ such vertices. For this reason, their are $O(n^2 \log n)$ paths stored in all $O(n \log n)$ vertices of $G(\mathcal{V})$. Hence, the running time of the algorithm becomes $O(n^2 \log^2 n)$ and the space becomes $O(n^2 \log n)$. As for the minimum-link shortest paths, by using the graph $G^*(\mathcal{V})$ and the techniques in [2, 25], we can reduce the running time by a factor of $\sqrt{\log n}$. We omit the details.

For computing a shortest minimum-link s-t path, we use a similar algorithm as above but with the following changes. First, we use the measure vector $(L_d(\pi), L_1(\pi))$ instead. Second, we use the following rule, which is similar to Rule(a_2). The complexities are the same as the above for minimum-cost paths.

Rule(a_3) Let π_1 be one of $\pi'(s,q)$ and $\pi(s,q)$, and π_2 the other. If the last segments of π_1 and π_2 are exactly the same and the measure vector of π_1 is no larger than that of π_2, then we discard π_2.

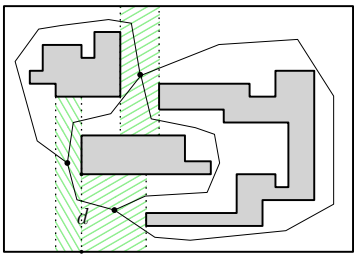

Figure 5 Illustrating the corridor structure and the corridor graph G_{cor} of three vertices. There are three junction rectangles, which are highlighted. Each connected white region is a corridor, which corresponds to an edge of G_{cor}. The diagonal d forms a degenerated corridor.

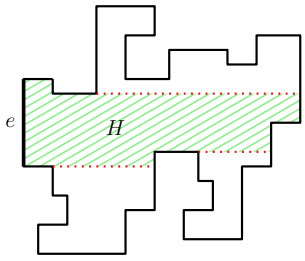

Figure 6 Illustrating the maximal histogram H, which has three windows shown with (red) dotted segments.

4 The Improved Algorithm

We further improve our algorithm, so that in addition to $O(n)$, the complexities of our improved algorithm only depend on h, i.e., the number of holes of $\mathcal{P}$. We first review the corridor structure of $\mathcal{P}$ [17] and the histogram partitions of rectilinear simple polygons [21].

The Corridor Structure of $\mathcal{P}$. For ease of exposition, we make a general position assumption that no two edges of $\mathcal{P}$ are collinear. The *vertical visibility decomposition* of $\mathcal{P}$, denoted by $VD(\mathcal{P})$, is obtained by extending each vertical edge of $\mathcal{P}$ until it hits the boundary of $\mathcal{P}$. Each cell of $VD(\mathcal{P})$ is a rectangle. Each extension segment is called a *diagonal* of $VD(\mathcal{P})$.

Let G_{vd} be the dual graph of $VD(\mathcal{P})$ (see Fig. 4), i.e., each node of G_{vd} corresponds to a cell of $VD(\mathcal{P})$ and two nodes have an edge if the corresponding cells share an edge. Based on G_{vd}, we obtain a *corridor graph* G_{cor} as follows. First, we keep removing every degree-one node from G_{vd} along with its incident edge until no such nodes remain. Second, we keep contracting every degree-two node from G_{vd} (i.e., remove the node and replace its two incident edges by a single edge) until no such nodes remain. The graph thus obtained is G_{cor}, which has $O(h)$ nodes and $O(h)$ edges [17]. See Fig. 5. The cells of $VD(\mathcal{P})$ corresponding to the nodes of G_{cor} are called *junction rectangles*. If we remove all junction rectangles from $\mathcal{P}$, each connected region is a simple rectilinear polygon, which is called a *corridor*. Each corridor has two diagonals each of which is on a vertical side of a junction rectangle, and we call them the *doors* of the corridor. For convenience, if a diagonal d bounds two junction rectangles (see Fig. 5), then we consider d itself as a "degenerate" corridor whose two doors are both d. With the degenerated corridors, each vertex of $\mathcal{P}$ lies in a unique corridor.

The decomposition $VD(\mathcal{P})$ can be computed in $O(n + h\log^{1+\epsilon} h)$ time for any $\epsilon > 0$ [1]. After $VD(\mathcal{P})$ is known, the corridor structure of $\mathcal{P}$ can be obtained in $O(n)$ time.

The Histogram Partitions. The histogram partition is a decomposition of a simple rectilinear polygon [21]. We will need to build the histogram partitions on the corridors of $\mathcal{P}$.

A simple rectilinear polygon H is called a *histogram* if its boundary can be divided into an x- or y-monotone chain and a single line segment, which is called the base of H.

Consider a simple rectilinear polygon Q (e.g., a corridor $\mathcal{C}$ of the corridor structure of $\mathcal{P}$) and let e be an edge of Q (e.g., a door of $\mathcal{C}$). A histogram partition of Q with respect to e, denoted by $\mathcal{H}(Q, e)$, is defined as follows. Let H be the *maximal histogram* with base

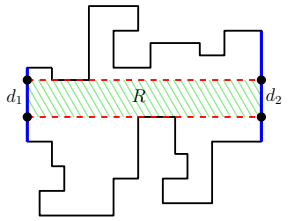

Figure 7 Illustrating an open corridor: the canal R and the two bridges are highlighted. The four points on the two doors are backbone points.

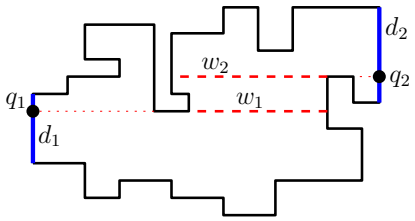

Figure 8 Illustrating a closed corridor. The points q_1 and q_2 are backbone points on d_1 and d_2, respectively.

e in Q, i.e., there is no other histogram in Q with base e that can properly contain it (see Fig. 6). A *window* of H is a maximal segment on the boundary of H that is contained in the interior of Q except its two endpoints. For each window w of H, it divides H into two subpolygons, and we let $Q(w)$ be the one that does not contain e. If H does not have a window, we are done with the histogram partition of Q. Otherwise, for each window w, we perform the above partition on $Q(w)$ recursively with respect to w.

For any points p and q in Q, it is known that there exists a path from p to q in Q that is both a shortest path and a minimum-link path [10, 11, 21], and we call it a *smallest path*.

4.1 A Reduced Path Preserving Graph

Recall that our algorithm in Section 3 use a graph $G(\mathcal{V})$, which is built on the vertices of $\mathcal{V}$ and has $O(n \log n)$ nodes and edges. In this section, as a major tool for reducing the complexities of our algorithm, we propose a *reduced graph* of $O(h \log h)$ nodes and edges. We first introduce a set $\mathcal{B}$ of $O(h)$ *backbone* points on the doors of the corridors of $\mathcal{P}$.

The Backbone Points. Consider a corridor $\mathcal{C}$ of the corridor structure of $\mathcal{P}$. Let d_1 and d_2 be the two doors of $\mathcal{C}$, which are both vertical. The region of $\mathcal{C}$ excluding the two doors is called the *interior* of $\mathcal{C}$. If there exist a point $p_1 \in d_1$ and a point $p_2 \in d_2$ such that $\overline{p_1p_2}$ is horizontal and $\overline{p_1p_2}$ in $\mathcal{C}$ then we say that $\mathcal{C}$ is an *open corridor*; otherwise, it is *closed*.

Consider an open corridor $\mathcal{C}$ (see Fig. 7). Let p_1 and p_2 be the points defined above. Imagine that we drag $\overline{p_1p_2}$ vertically upwards (resp., downwards) until we hit a vertex of $\mathcal{C}$, then the current locations of p_1 and p_2 are two *backbone points*. In this way, each door of $\mathcal{C}$ has two backbone points. Clearly, the rectangle R with the four backbone points as the vertices is in $\mathcal{C}$ and we call R the *canal* of $\mathcal{C}$. The two horizontal edges of R are called *bridges* of $\mathcal{C}$. Further, the top edge of R is the *upper* bridge and the bottom edge is the *lower* bridge.

If $\mathcal{C}$ is a degenerate corridor, which is a single diagonal d, then $\mathcal{C}$ is also an open corridor and the upper (resp., lower) bridge is degenerated to the upper (resp., lower) endpoint of d.

Next, we consider the case where $\mathcal{C}$ is closed (see Fig. 8). Let H_1 be the maximal histogram in $\mathcal{C}$ with base d_1. As $\mathcal{C}$ is closed, H_1 has a window w_1 that *separates* d_1 from d_2, that is, w_1 divides $\mathcal{C}$ into two sub-polygons that contain d_1 and d_2, respectively. By the definition of windows, if we extend w_1 to d_1, the extension will hit d_1 at a point, denoted by q_1, before it goes out of $\mathcal{C}$. Similarly, we define H_2, w_2, and q_2, with respect to the other door d_2. The two points q_1 and q_2 are *backbone points* of $\mathcal{C}$.

The above defines two backbone points on each door of every open corridor and one backbone point on each door of every closed corridor. Let $\mathcal{B}$ denote the set of all such backbone points. Since there are $O(h)$ corridors, the size of $\mathcal{B}$ is $O(h)$.

The Reduced Graph $G(\mathcal{B})$. In the sequel, we introduce the reduced graph, denoted by $G(\mathcal{B})$. We first consider the case where both s and t are in junction rectangles. With a little abuse of notation, we let $\mathcal{B}$ also contain both s and t.

We build the graph $G(\mathcal{B})$ with respect to the points of $\mathcal{B}$ in the same way as $G(\mathcal{V})$ with respect to $\mathcal{V}$ in Section 3. Hence, $G(\mathcal{B})$ has $O(h \log h)$ vertices and $O(h \log h)$ edges. In addition, we add the following $O(h)$ edges to $G(\mathcal{B})$. Consider a closed corridor $\mathcal{C}$ with the two backbone points q_1 and q_2 on its two doors. Note that q_1 and q_2 are also two vertices in $G(\mathcal{B})$. We add to $G(\mathcal{B})$ an edge $e(q_1, q_2)$ to connect q_1 and q_2 with length equal to $L_1(\pi(\mathcal{C}, q_1, q_2))$, where $\pi(\mathcal{C}, q_1, q_2)$ is a shortest path from q_1 to q_2 in $\mathcal{C}$. We call $e(q_1, q_2)$ a *corridor edge* of $G(\mathcal{B})$, and call $\pi(\mathcal{C}, q_1, q_2)$ a *corridor path* of $\mathcal{C}$. We do this for all closed corridors. This completes the construction of $G(\mathcal{B})$. Since there are $O(h)$ corridors, $G(\mathcal{B})$ has $O(h)$ corridor edges. For differentiation, other edges of $G(\mathcal{B})$ that are not corridor edges are called *ordinary edges*. Hence, $G(\mathcal{B})$ has $O(h \log h)$ edges in total. Note that every path $\pi_{G(\mathcal{B})}$ in $G(\mathcal{B})$ corresponds to a path π in $\mathcal{P}$ with the same length in the sense that if the path $\pi_{G(\mathcal{B})}$ contains a corridor edge, then π contains the corresponding corridor path.

The following lemma is analogous to Lemma 2, but on the reduced graph $G(\mathcal{B})$. It explains why the graph $G(\mathcal{B})$ can help to find optimal paths.

▶ **Lemma 3.** *There exists a path $\pi_{G(\mathcal{B})}$ in $G(\mathcal{B})$ from s to t that is homotopic to an optimal s-t path and the two paths have the same length; we call $\pi_{G(\mathcal{B})}$ a* target path*.*

Lemma 3 implies that a shortest s-t path in $G(\mathcal{B})$ is a shortest s-t path in $\mathcal{P}$. Hence, $G(\mathcal{B})$ is indeed a "path-preserving" graph. We can compute $G(\mathcal{B})$ in $O(n + h \log^2 h)$ time using the previous algorithm [7, 13, 25] as well as a so-called *reduced domain* $\mathcal{P}_r$, which consists of all junction rectangles and the canals of all open corridors of $\mathcal{P}$. The details are omitted.

4.2 Computing an Optimal Path

In this section, we compute an optimal s-t path using $G(\mathcal{B})$. We will show that an optimal s-t path can be computed by applying the dragging operations as in [25] on the ordinary edges of $\pi_{G(\mathcal{B})}$ and applying a new kind of operations, called *through-corridor-path generating operations*, on corridor edges of $\pi_{G(\mathcal{B})}$, where $\pi_{G(\mathcal{B})}$ is a target path defined in Lemma 3.

The algorithmic scheme is similar to that in Section 3.1. When we advance the searching process through an ordinary edge, we perform a dragging operation as in [25]. If we are advancing along a corridor edge, then we apply a through-corridor-path generating operation, which is introduced in the following. We first review some results from Schuierer [21].

Consider a closed corridor $\mathcal{C}$. Let d be a door of $\mathcal{C}$ and let q be the backbone point on d. Recall that q is an extension of a window w of the maximal histogram H in $\mathcal{C}$ with base d.

Let p be a point in $\mathcal{C}$. Following the terminology in [21], a rectilinear path from p to a point on d is called an *admissible path* if the last link is orthogonal to d. A *minimum-link admissible path* from p to d is an admissible path from p to any point of d with the smallest number of links, and we use $L_d(p, d)$ to denote the number of links in the path. Let $I_1(p, d)$ (resp., $I_2(p, d)$) denote the set of points on d that can be reached by p with an admissible path of at most $L_d(p, d)$ (resp., $L_d(p, d) + 1$) links (e.g., see Fig. 9). It is known that each of $I_1(p, d)$ and $I_2(p, d)$ is an interval of d, and $I_1(p, d) \subseteq I_2(p, d)$ [21]. Further, if p is not horizontally visible to d, then both intervals have q as one of their endpoints. By using the histogram partition $\mathcal{H}(\mathcal{C}, d)$, Schuierer [21] built a data structure in $O(|\mathcal{C}|)$ time such that given any point $p \in \mathcal{C}$, the two intervals $I_1(p, d)$ and $I_2(p, d)$ can be determined in $O(\log |\mathcal{C}|)$ time. With a little abuse of notation, we also use $\mathcal{H}(\mathcal{C}, d)$ to refer to the above data structure.

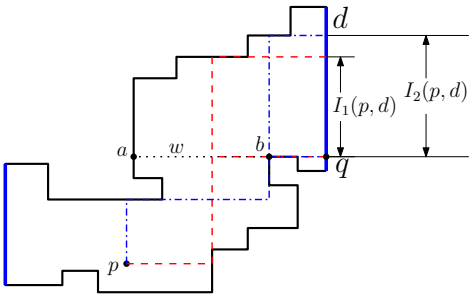

Figure 9 Illustrating the two intervals $I_1(p,d)$ and $I_2(p,d)$, where $L_d(p,d) = 3$ and $\overline{ab}$ is the window w. The two blue segments are doors of the corridor.

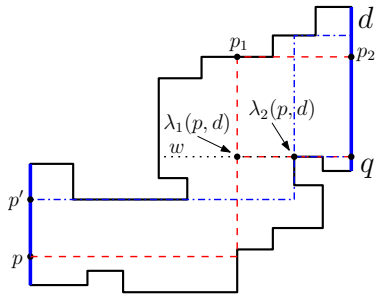

Figure 10 Illustrating the two points $\lambda_1(p,d)$ and $\lambda_2(p,d)$ on the window w. p' is also a backbone point.

Suppose p is a point on the other door of $\mathcal{C}$ than d (so p is not horizontally visible to d). Then, $I_1(p,d)$ is uniquely determined by a point, denoted by $\lambda_1(p,d)$, on the window w in the following way [21] (e.g., see Fig. 10). Recall that d is vertical and thus w is horizontal. Without loss of generality, assume that the histogram H is locally above w and locally on the left of d. We shoot a ray from $\lambda_1(p,d)$ upwards until a point p_1 on the boundary of $\mathcal{C}$ and then we project p_1 perpendicular to d and let p_2 be the projection point. The point p_2 is the other endpoint of the interval $I_1(p,d)$, i.e., $I_1(p,d) = \overline{qp_2}$. Note that p_2 is above q. Let $I_1'(p,d)$ denote the segment $\overline{\lambda_1(p,d)q}$, which is on the extension of the window w. We can also understand the two intervals $I_1(p,d)$ and $I_1'(p,d)$ in the following way. There exists an admissible path of $L_d(p,d)$ links from p to q, denoted by $\pi_1(\mathcal{C},p,q)$, which is actually a *smallest* path from p to q, and its last link is $I_1'(p,d)$; for any point $q' \in I_1(p,d)$, by dragging the last segment of $\pi_1(\mathcal{C},p,q)$ upwards until q', we can obtain an admissible path of $L_d(p,d)$ links from p to q'. The data structure $\mathcal{H}(\mathcal{C},d)$ can also report $\lambda_1(p,d)$ in $O(\log n)$ time and the path $\pi_1(\mathcal{C},p,q)$ can be output in additional time linear in the link distance of the path.

The interval $I_2(p,d)$ is uniquely determined by a point $\lambda_2(p,d)$ on the window w in the similar way as above. Similarly, we define $I_2'(p,d)$ and the corresponding admissible path of $L_d(p,d)+1$ links from p to q whose last link is $I_2'(p,d)$, denoted by $\pi_2(\mathcal{C},p,q)$, which is a *shortest* path (but not necessarily a smallest path) from p to q in $\mathcal{C}$ [21]. Similarly, the data structure $\mathcal{H}(\mathcal{C},d)$ can also report $\lambda_2(p,d)$ in $O(\log n)$ time and the path $\pi_2(\mathcal{C},p,q)$ can be output in additional time linear in the link distance of the path.

In the following, we introduce our through-corridor-path generating operations for advancing paths along corridor edges in our algorithm for searching the graph $G(\mathcal{B})$.

Consider a corridor edge $e(q_1,q_2)$ connecting two vertices q_1 and q_2 of $G(\mathcal{B})$. Note that q_1 and q_2 are two backbone points that are on the two doors d_1 and d_2 of a closed corridor $\mathcal{C}$, respectively. Consider a path $\pi(s,q_1)$ from s to q_1 maintained by our algorithm. Suppose we want to advance $\pi(s,q_1)$ from q_1 to q_2 along the corridor edge $e(q_1,q_2)$. We perform the following through-corridor-path generating operation that will extend $\pi(s,q_1)$ from q_1 to q_2 to obtain a path $\pi(s,q_2)$ from s to q_2.

Recall that q_1 is an extension of a window w_1 of the maximal histogram H_1 in $\mathcal{C}$ with base d_1. Hence, w_1 divides $\mathcal{C}$ into two sub-polygons that contain d_1 and d_2, respectively. Without loss of generality, we assume that the sub-polygon containing d_2 is locally above w_1. We also assume that $\mathcal{C}$ is locally on the right of d_1 (e.g., see Fig. 11).

Let α be the last segment of $\pi(s,q_1)$ (i.e., the one incident to q_1) and let p be the other endpoint of α than q_1. Suppose we have already built the data structure $\mathcal{H}(\mathcal{C},d_2)$ for $\mathcal{C}$ with respect to the door d_2. Depending on whether α is horizontal or vertical, there are two cases.

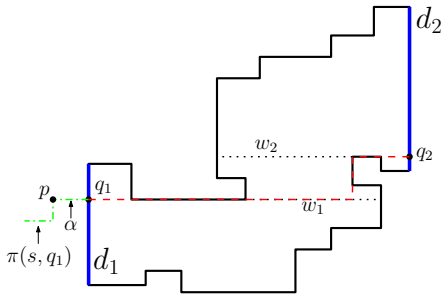

Figure 11 Illustrating the through-corridor-path generating operation for the case where α is horizontal. The path $\pi_1(\mathcal{C}, q_1, q_2)$ are shown with red dashed segments.

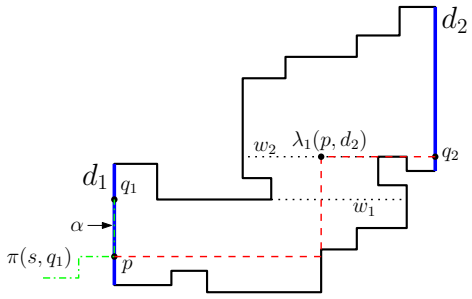

Figure 12 Illustrating the through-corridor-path generating operation for the case where α is vertical and p is on d_1 below q_1. The smallest path $\pi_{opt}(\mathcal{C}, q_1, q_2)$ are shown with red dashed segments. Note that $I_1'(p, d_2) = \overline{\lambda_1(p, d_2)q_2}$.

1. If α is horizontal (e.g., see Fig. 11), then p must be to the left of q_1 since $\mathcal{C}$ is locally on the right side of d_1. In this case, we use $\mathcal{H}(\mathcal{C}, d_2)$ to determine the path $\pi_1(\mathcal{C}, q_1, q_2)$ (whose last link is $I_1'(q_1, d_2)$) and concatenate it with $\pi(s, q_1)$ to obtain $\pi(s, q_2)$. We also compute the number of links of $\pi(s, q_2)$ and its length, and store them at q_2. Note that $L_1(\pi(s, q_1))$ and $L_d(\pi(s, q_1))$ are already stored at q_1.
2. If α is vertical, then depending on whether p is above q_1, there are two subcases.
   a. If p is above q_1, then we use the same approach as above to obtain $\pi(s, q_2)$. Note that in this case the path makes a turn at q_1 while there is no turn at q_1 in the above case.
   b. If p is below q_1, then depending on whether p is on d_1, there are further two subcases.
      i. If p is not on d_1, then we use the same approach as above to obtain $\pi(s, q_2)$.
      ii. If p is on d_1, this is the trickiest case. Let $\pi_{opt}(\mathcal{C}, p, q_2)$ denote the smallest path from p to q_2 in $\mathcal{C}$ whose last link is $I_1'(p, d_2)$ (e.g., see Fig. 12). $\pi_{opt}(\mathcal{C}, p, q_2)$ can also be determined in $O(\log n)$ time by the data structure $\mathcal{H}(\mathcal{C}, d_2)$. We obtain $\pi(s, q_2)$ by concatenating $\pi_{opt}(\mathcal{C}, p, q_2)$ with the subpath of $\pi(s, q_1)$ between s and p.

As a summary, to obtain $\pi(s, q_2)$, if Case 2(b)ii happens, then we connect the subpath of $\pi(s, q_1)$ between s and p with $\pi_{opt}(\mathcal{C}, p, q_2)$; otherwise, we connect $\pi(s, q_1)$ with $\pi_1(\mathcal{C}, q_1, q_2)$. Note that in either case the last link of $\pi(s, q_2)$ is $I_1'(q_1, d_2)$. In either case, let π' be the subpath of $\pi(s, q_2)$ contained in $\mathcal{C}$. With the histogram partition $\mathcal{H}(\mathcal{C}, d_2)$, we can obtain $L_1(\pi')$ and $L_d(\pi')$ as well as the first and last links of π' in $O(\log n)$ time (the path π' can be output in additional $O(L_d(\pi'))$ time). Hence, we can compute $L_1(\pi(s, p_2))$ and $L_d(\pi(s, p_2))$ as well as its last link in $O(\log n)$ time, without explicitly computing the path π'. Therefore, the through-corridor-path generating operation can be performed in $O(\log n)$ time.

As discussed before, our algorithm works in the same way as the one in Section 3 except that we apply through-corridor-path generating operations on corridor edges of $G(\mathcal{B})$ instead of the dragging operations. We can compute the histogram partitions for all closed corridors as the preprocessing for performing the through-corridor-path generating operations, and the total preprocessing time is $O(n)$ since the size of all corridors is $O(n)$. After the algorithm finishes, the path stored at t with the smallest measure is an optimal s-t path.

The above only discussed the case where both s and t are in junction rectangles. We can generalizes the approach to other cases if at least one of them is in a corridor. This is done by introducing *corridor-connection points* and adding a *beginning procedure* and a *concatenation procedure* to the algorithm. The details can be found in the full paper [22].

▶ **Theorem 4.** *We can compute a minimum-link shortest s-t path in $O(n + h\log^{3/2} h)$ time and $O(n + h\log h)$ space, and compute a shortest minimum-link s-t path or a minimum-cost s-t path in $O(n + h^2\log^{3/2} h)$ time and $O(n + h^2\log h)$ space.*

References

1. R. Bar-Yehuda and B. Chazelle. Triangulating disjoint Jordan chains. *International Journal of Computational Geometry and Applications*, 4(4):475–481, 1994.
2. D. Z. Chen, O. Daescu, and K. S. Klenk. On geometric path query problems. *International Journal of Computational Geometry and Applications*, 11(6):617–645, 2001.
3. D. Z. Chen, R. Inkulu, and H. Wang. Two-point L_1 shortest path queries in the plane. In *Proc. of the 30th Annual Symposium on Computational Geometry*, pages 406–415, 2014.
4. D. Z. Chen, K. S. Klenk, and H.-Y. T. Tu. Shortest path queries among weighted obstacles in the rectilinear plane. *SIAM Journal on Computing*, 29(4):1223–1246, 2000.
5. D. Z. Chen and H. Wang. A nearly optimal algorithm for finding L_1 shortest paths among polygonal obstacles in the plane. In *Proc. of the 19th European Symposium on Algorithms*, pages 481–492, 2011.
6. D. Z. Chen and H. Wang. L_1 shortest path queries among polygonal obstacles in the plane. In *Proc. of 30th Symp. on Theoretical Aspects of Computer Science*, pages 293–304, 2013.
7. K. Clarkson, S. Kapoor, and P. Vaidya. Rectilinear shortest paths through polygonal obstacles in $O(n\log^2 n)$ time. In *Proc. of the 3rd Annual Symposium on Computational Geometry*, pages 251–257, 1987.
8. K. Clarkson, S. Kapoor, and P. Vaidya. Rectilinear shortest paths through polygonal obstacles in $O(n\log^{2/3} n)$ time. Manuscript, 1988.
9. G. Das and G. Narasimhan. Geometric searching and link distance. In *Proc. of the 2nd Workshop of Algorithms and Data Structures*, pages 261–272, 1991.
10. M. de Berg. On rectilinear link distance. *Computational Geometry: Theory and Applications*, 1:13–34, 1991.
11. J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry: Theory and Applications*, 4(2):63–97, 1994.
12. H. Imai and T. Asano. Efficient algorithms for geometric graph search problems. *SIAM Journal on Computing*, 15(2):478–494, 1986.
13. D. T. Lee, C. D. Yang, and T. H. Chen. Shortest rectilinear paths among weighted obstacles. *International Journal of Computational Geometry and Applications*, 1(2):109–124, 1991.
14. J. S. B. Mitchell. An optimal algorithm for shortest rectilinear paths among obstacles. Abstracts of the *1st Canadian Conference on Computational Geometry*, 1989.
15. J. S. B. Mitchell. L_1 shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8(1):55–88, 1992.
16. J. S. B. Mitchell, V. Polishchuk, and M. Sysikaski. Minimum-link paths revisited. *CGTA*, 47:651–667, 2014.
17. J. S. B. Mitchell, V. Polishchuk, M. Sysikaski, and H. Wang. An optimal algorithm for minimum-link rectilinear paths in triangulated rectilinear domains. In *Proc. of the 42nd International Colloquium on Automata, Languages and Programming*, pages 947–959, 2015.
18. J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.
19. V. Polishchuk and J. S. B. Mitchell. k-Link rectilinear shortest paths among rectilinear obstacles in the plane. In *Proc. of the 17th Canadian Conference on Computational Geometry (CCCG)*, pages 101–104, 2005.
20. M. Sato, J. Sakanaka, and T. Ohtsuki. A fast line-search method based on a tile plane. In *Proc. of the IEEE International Symposium on Circuits and Systems*, pages 588–597, 1987.

21 S. Schuierer. An optimal data structure for shortest rectilinear path queries in a simple rectilinear polygon. *International Journal of Computational Geometry and Applications*, 6:205–226, 1996.

22 H. Wang. Bicriteria rectilinear shortest paths among rectilinear obstacles in the plane. arXiv:1703.04466, 2017.

23 Y.-F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong. Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles. *IEEE Transactions on Computers*, 36:321–331, 1987.

24 C. D. Yang, D. T. Lee, and C. K. Wong. On bends and lengths of rectilinear paths: A graph-theoretic approach. *Int. J. Comput. Geom. Appl.*, 02:61–74, 1992.

25 C. D. Yang, D. T. Lee, and C. K. Wong. Rectilinear path problems among rectilinear obstacles revisited. *SIAM Journal on Computing*, 24:457–472, 1995.