An Optimal Algorithm for Minimum-Link Rectilinear Paths in Rectilinear Domains

Joseph S.B. Mitchell[1], Valentin Polishchuk[2], Mikko Sysikaski[3], and Haitao Wang[4]

[1] Stony Brook University, Stony Brook, NY 11794, USA, jsbm@ams.stonybrook.edu
[2] Linköping University, Linköping, Sweden, valentin.polishchuk@liu.se
[3] University of Helsinki, Helsinki, Finland, mikko.sysikaski@helsinki.fi
[4] Utah State University, Logan, UT 84322, USA, haitao.wang@usu.edu

Abstract. We present a new algorithm for finding minimum-link rectilinear paths among rectilinear obstacles in the plane. The algorithm runs in $O(n + h \log h)$ time (after triangulation) and $O(n)$ space, where h is the number of obstacles and n is the total number of their vertices. Further, for any point s, with the same time and space preprocessing as above, given any query point t, we can compute a minimum-link rectilinear path from s to t in $O(\log n + k)$ time, where k is the number of edges of the path. The previously best algorithm solves the problem in $O(n \log n)$ time. Our algorithm improves the previous work when $h < n$.

1 Introduction

Let $\mathcal{P}$ be a set of h disjoint polygonal obstacles with a total of n vertices in the plane. The plane minus the interior of all obstacles is called the *free space*. The *link distance* of a path is defined to be the number of edges (also called *links*) in the path. A *minimum-link path* (or *min-link* path) between two points s and t is a path from s to t in the free space with the minimum link distance. The *min-link path query* problem is to construct a data structure (called *link distance map*) with respect to a given *source* point s, such that for any query point t, a min-link path from s to t can be quickly computed. In the following, we say a link distance map has the *standard query performance* if given any t, the link distance of a min-link s-t path can be computed in $O(\log n)$ time and the actual path can be output in additional time linear in the link distance of the path.

A polygon (or path) is *rectilinear* if all its edges are axis-parallel. $\mathcal{P}$ is a *rectilinear polygonal domain* if every obstacle of $\mathcal{P}$ is rectilinear (e.g., see Fig. 5 in Appendix). The *rectilinear version* of the min-link path/query problem is to find min-link *rectilinear* paths in rectilinear polygonal domains.

Previous Work. Linear-time algorithms have been given for finding min-link paths in simple polygons [9, 10, 21–23]. The link distance map can also be built in linear time [21–23] for simple polygons, with the standard query performance. For polygonal domains, the problem becomes much more difficult. Mitchell, Rote, and Woeginger [17] gave an $O(n^2 \alpha(n) \log^2 n)$ time algorithm for finding min-link paths, where $\alpha(n)$ is the inverse Ackermann function; a link distance map with slightly larger construction time is also given in [17]. As shown in [16], finding min-link paths in polygonal domains is 3SUM-hard.

The rectilinear min-link path problems have also been studied. For simple rectilinear polygons, de Berg [7] presented an algorithm that can build an $O(n)$-size link distance map in $O(n \log n)$ time and $O(n)$ space, with the standard query performance. The construction time was later reduced to $O(n)$ time by Lingas, Maheshwari, and Sack [14], and by Schuierer [20].

For rectilinear polygonal domains, Imai and Asano [11] presented an $O(n \log n)$ time and space algorithm for finding min-link rectilinear paths. Later, Das and Narasimhan [6] described an improved algorithm of $O(n \log n)$ time and $O(n)$ space; Sato, Sakanaka, and Ohtsuki [19] gave a similar algorithm with the same performance. Recently, Mitchell, Polishchuk, and Sysikaski [16] presented a simpler algorithm of $O(n \log n)$ time and $O(n)$ space. Link distance maps of $O(n)$-size can also be built in $O(n \log n)$ time and $O(n)$ space [6, 16]. As shown in [6, 15], the problem has an $\Omega(n + h \log h)$ time lower bound. Thus, the algorithms in [6, 16, 19] are optimal only when $h = \Theta(n)$. However, since the value h can be substantially smaller than n, it is desirable to have an algorithm whose running time is bounded by $O(n + f(h))$, where $f(h)$ is a function of h.

Our Results. After the free space of $\mathcal{P}$ is triangulated, our algorithm builds a link distance map in $O(n + h \log h)$ time and $O(n)$ space, with the standard query performance. The triangulation can be done in $O(n \log n)$ time or $O(n + h \log^{1+\epsilon} h)$ time for any $\epsilon > 0$ [1]. Hence, our result improves the previous $O(n \log n)$ time algorithms [6, 16, 19], especially when h is substantially smaller than n, e.g., if $h = O(n^{1-\delta})$ for any $\delta > 0$, our algorithm runs in $O(n)$ time.

Algorithm Overview. Our idea is to combine Das and Narasimhan's algorithmic scheme [6] and a *corridor structure* of polygonal domains. The corridor structure and its extensions have been used to solve shortest path problems, e.g., [3, 4, 12, 18]; however, to the best of our knowledge, it was not used to tackle min-link path problems. The corridor structure partitions the free space of $\mathcal{P}$ into $O(h)$ corridors and $O(h)$ "junction" rectangles that connect all corridors.

The algorithm in [6] (which we call the DN algorithm) sweeps the free space, from the source point s, to build the map. The sweep is controlled in a global way so that the time is bounded by $O(n \log n)$. This global sweeping on the entire free space restricts the DN algorithm from being implemented in $O(n + h \log h)$ time because each operation takes $O(\log n)$ time and there are $O(n)$ operations. Using the corridor structure, our algorithm avoids the global sweeping on the entire free space. When the sweep is in junction rectangles, we control the sweep in a global way as in the DN algorithm. However, when the sweep enters a corridor, we process the corridor independently and "locally" without considering the space outside the corridor. Since a corridor is a simple polygon, we are able to design a faster algorithm for processing the sweep in it. When we finish processing a corridor, we arrive at a junction rectangle. Next, we pick an unprocessed junction rectangle that currently has the smallest link distance to s to "resume" the sweep. This is somewhat similar to Dijkstra's shortest path algorithm. In this way, there are only $O(h)$ operations that need to be performed in logarithmic time each.

To achieve the $O(n + h \log h)$ time, we have to implement the algorithm in a very careful manner. For example, we give an efficient algorithm to compute link

distance maps in corridors (i.e., simple rectilinear polygons), which may be of independent interest. Although efficient algorithms are available for simple polygons, e.g., [7, 10, 14, 20], they are not suitable for our main algorithmic scheme, which requires some special properties (see the details in Section 3.2).

We first define notation and review the DN algorithm [6] in Section 2. Due to the space limit, many details are omitted but can be found in the appendix.

2 Preliminaries

For simplicity of discussion, let $\mathcal{R}$ be a large rectangle that contains all obstacles of $\mathcal{P}$ and let $\mathcal{F}$ denote the free space of $\mathcal{P}$ in $\mathcal{R}$. We assume $\mathcal{F}$ has been triangulated. Let s be any point in $\mathcal{F}$. For ease of exposition, we make a general position assumption that no three vertices of $\mathcal{P} \cup \{s\}$ have the same x- or y-coordinate. In the following, paths always refer to rectilinear paths in $\mathcal{F}$.

Consider any point $t \in \mathcal{F}$. An s-t path (i.e., a path from s to t) π is called a *horizontal-start-vertical-end* path (or *h-v-path* for short) if the first link of π (i.e., the edge incident to s) is horizontal and the last link of (i.e., the edge incident to t) is vertical. The *h-h-paths*, *v-h-paths*, and *v-v-paths* are defined analogously. To make it consistent, if π is an h-h-path of k links, we also consider it to be an h-v-path of $k+1$ links (i.e., we enforce an additional edge of zero length at the end of the path), and similarly, it is also considered to be an v-h-path of $k+1$ links and a v-v-path of $k+2$ links. A *min-link h-v-path* from s to t is an h-v-path from s to t with the minimum number of links. A *min-link h-h-paths*, *v-h-paths*, and *v-v-paths* is defined similarly. To find a min-link s-t path, we will find the four s-t paths: a min-link h-v-path, a min-link h-h-path, a min-link v-v-path, and a min-link v-h-path, and then return the one with minimum link distance.

We will compute four link distance maps of $O(n)$ size each: an *h-h-map*, an *h-v-map*, a *v-h-map*, and a *v-v-map*, defined as follows. The h-h-map is a decomposition of $\mathcal{F}$ into regions such that for any region R, the link distances of the min-link h-h-paths from s to all points in R are the same. The other three maps are defined analogously. Using point location data structures [8, 13], for any query point t, we determine the region containing t in each map and the one with the smallest link distance gives our sought min-link s-t path distance.

The *vertical visibility decomposition* of $\mathcal{F}$, denoted by $VD(\mathcal{F})$, is obtained by extending each vertical edge of the obstacles in $\mathcal{P}$ until it hits either another obstacle or the boundary of $\mathcal{R}$ (e.g., see Fig. 6 in Appendix). We call the above edge extensions the *diagonals*. We consider the point s as a special obstacle and extend a vertical diagonal through s. Since $\mathcal{F}$ has been triangulated, $VD(\mathcal{F})$ can be obtained in $O(n)$ time [1, 2]. In $VD(\mathcal{F})$, $\mathcal{F}$ is decomposed into rectangles, also called *cells*. Due to our general position assumption, each vertical side of a cell can contain at most two diagonals. The horizontal visibility decomposition of $\mathcal{F}$, denoted by $HD(\mathcal{F})$, is defined similarly by extending the horizontal edges of $\mathcal{P}$.

Our v-v-map is on $VD(\mathcal{F})$, i.e., for each diagonal d (resp., cell C) of $VD(\mathcal{F})$, the link distances of the min-link v-v-paths from s to all points in d (resp., in C that are not on diagonals) are the same and we denote this distance by $dis_{vv}(d)$ (resp., $dis_{vv}(C)$). In fact, $dis_{vv}(d) = \min\{dis_{vv}(C_l), dis_{vv}(C_r)\}$, where C_l and

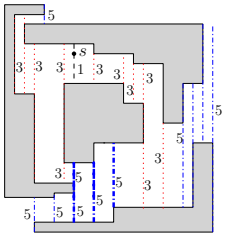

Fig. 1. Illustrating the algorithm: all diagonals are labeled. Note that the three thick dash-dotted diagonals (labeled 5) are swept twice in the 2nd phase.

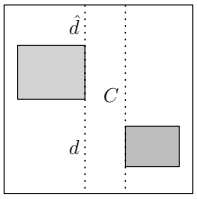

Fig. 2. Illustrating a cell C where d is on the left side of C.

C_r are the two cells on the left and right of d, respectively. Our goal is to compute $dis_{vv}(C)$ for each cell C and $dis(d)$ for each diagonal d of $VD(\mathcal{F})$. We also need to maintain some path information to retrieve an actual path for each query.

Similarly, our h-v-map is also on $VD(\mathcal{F})$, but the h-h-map and the v-h-map are both on $HD(\mathcal{F})$. Next in Section 2.1, we give an $O(n \log n)$ time algorithm, which is similar to the DN algorithm [6] but with slightly different descriptions. Later in Section 3 we will improve the algorithm to $O(n + h \log h)$ time.

2.1 An $O(n \log n)$ time Algorithm (the DN Algorithm)

We compute the v-v-map on $VD(\mathcal{F})$ first. The goal is to compute $dis_{vv}(d)$ for every diagonal d of $VD(\mathcal{F})$ (we will show later that the algorithm can be used to label cells as well). To simplify the notation, we use $dis(\cdot)$ to refer to $dis_{vv}(\cdot)$. Initially, all diagonals have distance value ∞ except $dis(d_s) = 1$, where d_s is the diagonal through s. Note that if a diagonal d is on a side e of a cell, then whenever $dis(d)$ is updated, $dis(e)$ is automatically set to $dis(d)$.

The algorithm has many phases. In the i-th phase for $i \geq 0$, the algorithm determines the set V_i of diagonals d whose distances $dis(d)$ are equal to $2i + 1$, and these diagonals are then "labeled" with distance $2i + 1$ (e.g., see Fig. 1). Initially, $i = 0$, and V_0 consists of the diagonal d_s only. As discussed in [6], if we put light sources on the diagonals in V_{i-1}, then V_i consists of all new diagonals that will get illuminated with light emanating horizontally from the light sources.

Consider a general i-th phase for $i \geq 1$. We assume V_{i-1} has been determined. There are two procedures: *right-sweep* and *left-sweep*. In the right-sweep (resp., left-sweep), we illuminate the diagonals in the rightwards (resp., leftwards) direction. The right-sweep procedure starts from the *locally-rightmost* diagonals of V_{i-1}, defined as follows. Consider any diagonal d in V_{i-1}. Let C be the cell of $VD(\mathcal{F})$ on the right of d, i.e., d is on the left side of C. Let e_r be the right side of C. If $dis(e_r) \neq 2i - 1$ then d is a *locally-rightmost* diagonal of V_{i-1}. Similarly, the left-sweep starts from the *locally-leftmost* diagonals of V_{i-1}. Both locally-leftmost and locally-rightmost diagonals are referred to as *locally-outmost* diagonals. Below, we first discuss the right-sweep.

For each locally-rightmost diagonal d, we put a rightward "light beam" on d, and let $B(d)$ denote the beam; we also call d the *generator* of the beam. Formally, we may define the beam as the segment d associated with the rightward

direction. Initially we insert all locally-rightmost diagonals into a min-heap H_R whose "keys" are the x-coordinates of the diagonals (i.e., the leftmost diagonal is at the root). By using H_R, the diagonals involved in the right-sweep will be processed from left to right. Although initially each diagonal of H_R has only one beam, later we will insert more diagonals into H_R and a diagonal d may have more than one beam, in which case $B(d)$ denotes the set of beams on d.

As long as H_R is not empty, we repeatedly do the following.

We obtain the leftmost diagonal d of H_R and remove it from H_R. Let C be the cell on the right of d (e.g., see Fig. 2). We *process* d in the following way. Intuitively we want to propagate the beams of $B(d)$ to other diagonals in C. Let e_l and e_r denote the left and right sides of C, respectively. Note that d is on e_l. Recall that each cell side has at most two diagonals.

If e_l has another diagonal $\hat{d}$ (e.g., see Fig. 2) and $\hat{d}$ has not been labeled (i.e., $dis(\hat{d}) = \infty$), we set $dis(\hat{d}) = 2i + 1$. The beam set $B(\hat{d})$ of $\hat{d}$ is set to $\emptyset$ since no beam from $B(d)$ illuminates $\hat{d}$. Further, although $B(\hat{d}) = \emptyset$, we associate the *leftward direction* with it, because if later the left-sweep procedure does not illuminate $\hat{d}$, $\hat{d}$ will be a locally-leftmost diagonal of V_i and generate a leftward beam in the next phase. We "temporarily" mark $\hat{d}$ as a locally-leftmost diagonal. We say "temporarily" since the left-sweep may clear the mark, as discussed later.

If $\hat{d}$ has been labeled, then it can be shown that $dis(\hat{d})$ must be $2i + 1$ (see Lemma 2 in Appendix for the proof). In this case, we do nothing on $\hat{d}$.

Next, we consider the diagonals on the right side e_r of C. Depending on the values of $dis(e_r)$, there are several cases. Since we are at the i-th phase, either $dis(e_r) = \infty$ or $dis(e_r) \leq 2i + 1$. If $dis(e_r) < 2i + 1$, then we do nothing on e_r.

If $dis(e_r) = \infty$, we set $dis(e_r) = 2i + 1$. If e_r does not have any diagonals, we are done. Otherwise, for each diagonal d' on e_r, we determine the portions of beams of $B(d)$ that can illuminate d', which are the rightward projections of $B(d)$ on d' (beams of $B(d)$ may be "narrowed" or "split"; see Fig. 7 in Appendix). We use $B(d) \cap d'$ to denote the above portions of $B(d)$. If $B(d) \cap d' = \emptyset$, we temporarily mark d' as a locally-rightmost diagonal and set $B(d') = \emptyset$ with the *rightward direction*; otherwise, we set $B(d') = B(d) \cap d'$ and insert d' to H_R (later we will propagate the beams of $B(d')$ further to the right).

If $dis(e_r) = 2i + 1$, this case happens because e_r was illuminated by beams from another diagonal $\hat{d}$ on e_l. Hence, each diagonal d' on e_r may already have a non-empty $B(d')$. But d' may receive more beams from $B(d)$. We first determine $B(d) \cap d'$ (as defined above) and then do a "merge" operation by merging $B(d) \cap d'$ with $B(d')$ (e.g., see Fig. 7 in Appendix). Finally, we set $B(d')$ to the above merged set of beams (with the rightward direction). If $B(d')$ was empty before the merge and becomes non-empty after the merge, then we insert d' into H_R. If $B(d')$ is still empty after the merge, then we temporarily mark d' as a locally-rightmost diagonal. If $B(d')$ was non-empty before the merge, then d' is already in H_R, so we do not need to insert it into H_R again.

The above finishes processing d. The right-sweep is done once H_R becomes empty. We use balanced binary search trees to maintain the beams in $B(d)$ such that 'merge" and "split" operations can be performed in logarithmic time each.

The left-sweep procedure is similar. There is one subtle thing. If the sweep illuminates a diagonal d that has been labeled by the right-sweep, then this is ignored and we proceed as if d were not labeled. As discussed in [6], the reason for this is that the left-sweep may reach more cells than the right-sweep (e.g., see Fig. 1). In this way, each diagonal can be processed at most twice in a phase. But no diagonal can be processed in more than one phase. Also, suppose a diagonal d was temporarily marked as a locally-outmost diagonal during the right-sweep; if d is illuminated again in the left-sweep but d is not marked locally-outmost in the left-sweep, then we clear the previous mark on d (i.e., d is not considered locally-outmost any more). After the left-sweep, the remaining locally-outmost diagonals are exactly those that will be used in the next phase.

The above describes the i-th phase of the algorithm. The algorithm is done after all diagonals are labeled. The running time is $O(n \log n)$ because each diagonal is processed at most twice in the entire algorithm (once by a left-sweep procedure and once by a right-sweep procedure).

For computing the h-v-map, the algorithm is similar except that the initial set V_0 now consists of all diagonals that intersect d'_s, where d'_s is the horizontal line segment extended from s until it hits the obstacles, and in each i-th phase for $i \geq 0$, the value $dis_{hv}(d)$ is set to $2(i+1)$ for any diagonal d in V_i. The algorithms for computing h-h-map and v-h-map on $HD(\mathcal{F})$ are symmetric.

The above algorithm only labels diagonals. We also need to label cells as discussed before, which can be done by an easy adaption of the above algorithm. In addition, we also need to maintain path information to retrieve an actual path for each query. All these details can be found in Appendix A.

3 Our Improved Algorithm

We first introduce the corridor structure in rectilinear domains, which is similar to that in general polygonal domains [12]. Let G_{vtd} be the dual graph of the vertical visibility decomposition $VD(\mathcal{F})$ (see Fig. 3), i.e., each node of G_{vtd} corresponds to a cell of $VD(\mathcal{F})$, and each edge of G_{vtd} connects two nodes corresponding to two cells sharing the same diagonal. Based on G_{vtd}, we obtain a *corridor graph* G as follows (see Fig. 4). First, we remove every degree-one node from G_{vtd} along with its incident edge; repeat this process until no degree-one node exists. Second, remove every degree-two node from G_{vtd} and replace its two incident edges by a single edge; repeat this process until no degree-two node exists. The remaining graph is G. The cells in $VD(\mathcal{F})$ corresponding to the nodes in G are called *junction cells* (see Fig. 4). We consider the diagonal through s as a degenerate junction cell. As in the general polygonal domains [12], the graph G has $O(h)$ nodes and $O(h)$ edges. The removal of all junction cells from $VD(\mathcal{F})$ results in $O(h)$ *corridors*, each of which corresponds to an edge of G.

The boundary of any corridor $\mathcal{C}$ consists of four parts (see Fig. 4): (1) The boundary portion of an obstacle P_i, from a point a to a point b; (2) a diagonal $\overline{bc}$; (3) the boundary portion of an obstacle P_j from c to a point d; (4) a diagonal $\overline{da}$. The two diagonals $\overline{bc}$ and $\overline{ad}$ are called the *doors* of the corridor $\mathcal{C}$. The corridor $\mathcal{C}$ is a simple rectilinear polygon.

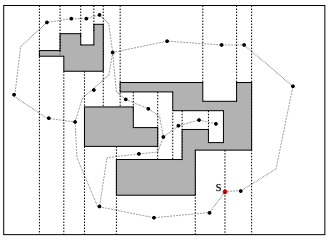

Fig. 3. Illustrating the vertical visibility decomposition (the dashed segments are diagonals) and its dual graph G_{vtd}.

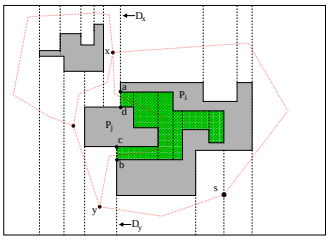

Fig. 4. Illustrating the graph G, and the corridor (shaded by slashes) bounded by P_i and P_j.

3.1 The Main Idea

We focus on computing the v-v-map on $VD(\mathcal{F})$; other three maps can be computed similarly. Our goal is to label d for each diagonal d, i.e., compute the distance value $dis(d) = dis_{vv}(d)$. As with the DN algorithm described in Section 2.1 and discussed in Appendix A, the same algorithm can be used to label cells as well. As before, each diagonal d will maintain a beam set $B(d)$.

Here is some intuition. In the DN algorithm, a sweep procedure will enter each corridor though one of its two doors, and the procedure will either sweep the entire corridor and leave the corridor through the other door, or terminate inside the corridor (in which case the sweep "hits" another sweep that entered the corridor through the other door and both sweeps terminate after "collision"). This means that if we can determine the beams and the distance values at the doors of a corridor, then we can process the corridor independently in a more efficient way since the corridor is a simple rectilinear polygon.

In our algorithm, the sweep in the junction cells is still processed and controlled in a global manner in each phrase. However, whenever the sweep enters a corridor through one of its doors, the corridor will be processed independently by using our more efficient *corridor-processing* algorithm given in Appendix C (i.e., the sweep "jumps" from one junction cell to another through the corridor connecting them). Note that since in the DN algorithm a diagonal may be processed twice in the two sweep procedures in the same phase, here correspondingly an entire corridor may be processed twice in the same phase (this happens *only if* the beams on a door can illuminate the other door directly, and vice versa).

The running time of our algorithm is $O(n + h \log h)$. More specifically, since there are $O(h)$ junction cells, the time spent on processing the diagonals in all junction cells is $O(h \log h)$, and the processing on all corridors takes $O(n+h \log h)$ time because the number of vertices of all corridors is $O(n)$, in addition to another $O(h \log h)$ time spent on maintaining the beams on all diagonals.

3.2 The Algorithm (a Sketch)

We only sketch our algorithm here; all details can be found in Appendix B.

Initially, we set $dis(d)$ to ∞ and $B(d) = \emptyset$ for each diagonal d except that $dis(d_s) = 1$ and $B(d_s) = \{d_s\}$, where d_s is the diagonal through s.

We use a min-heap H to store the diagonals in all junction cells, where the "keys" are the distance values the diagonals currently have (and these values may not be set correctly), with the *smallest* key at the root of H. Since there are $O(h)$ junction cells, the size of H is $O(h)$. Each diagonal d in H is also associated with its beam set $B(d)$ (along with its direction). As in the DN algorithm, it is possible that $B(d)$ is empty, in which case d might be a locally-outmost diagonal, but it is also associated with a direction for generating a beam towards that direction in the next phase. The algorithm is similar in spirit to Dijkstra's algorithm and the heap H plays the same rule as the priority queue in Dijkstra's algorithm.

If some diagonals of H have the same keys, we break the ties by the following rules. Consider two diagonals d_1 and d_2 in H with $dis(d_1) = dis(d_2)$. If $B(d_1)$ and $B(d_2)$ are both empty or both non-empty, then we break ties arbitrarily. Otherwise, assume $B(d_1) \neq \emptyset$ but $B(d_2) = \emptyset$. Then, we consider the key of d_1 is *smaller* than that of d_2. The reason is as follows. Since $B(d_1) \neq \emptyset$ and $B(d_2) = \emptyset$, the current sweep procedure should be over *before* processing d_2 while the sweep should continue *after* processing d_1, and thus, we should process d_1 before d_2. Therefore, our way of resolving ties in H is crucial and consistent with the DN algorithm. In the following, for any diagonal d, even if d is not in H, we consider $dis(d)$ along with $B(d)$ as the *global-key* of d, and whenever we compare the global-keys of diagonals, we follow the above rules to break ties.

Consider any corridor $\mathcal{C}$ with two doors d and d'. Suppose the beams of $B(d)$ are going inside $\mathcal{C}$, and we want to *process* $\mathcal{C}$ (i.e., compute the v-v-map in $\mathcal{C}$) using the beams of $B(d)$. We say the above way of processing $\mathcal{C}$ is in the *direction* from d to d'. As will be seen later, a corridor may be processed twice: once from d to d' and the other from d' to d. Due to the special geometric structure of the corridor, we have the following observation (see Appendix B for the proof).

Observation 1 *Suppose d and d' are the two doors of a corridor $\mathcal{C}$, and the direction of the beams of $B(d)$ is towards the inside of $\mathcal{C}$. Then after $\mathcal{C}$ is processed by using $B(d)$, the beam set of d' is not empty.*

Our algorithm is consistent with the DN algorithm in the sense that after the algorithm finishes, each diagonal in any junction cell is *correctly labeled*, i.e., both its distance value and its beam set are the same as those in the DN algorithm. Let d^* be the diagonal in the root of H. Our algorithm will maintain the following three invariants.

1. The diagonal d^* is correctly labeled. Further, for any other diagonal d in a junction cell, if the global-key of d is no larger than that of d^*, then d has been correctly labeled.
2. For any diagonal d in a junction cell, if $dis(d) \neq \infty$ and the global-key of d is larger than that of d^*, then d is in H.
3. For any corridor $\mathcal{C}$ with two doors d and d', if $\mathcal{C}$ is processed in the direction from d to d', then $\mathcal{C}$ will never be processed from d to d' again in the algorithm (although $\mathcal{C}$ may be processed later in the other direction from d' to d).

Initially $H = \emptyset$. Recall that $dis(d_s) = 1$ and $B(d) = \{d_s\}$. We consider the diagonal d_s through s as a degenerate junction cell. Specifically, we consider d_s

as two duplicate diagonals with one generating a rightward beam and the other generating a leftward beam from the entire d_s. We insert these two diagonals into H. As long as H is not empty, we repeatedly do the following.

Let d^* be the diagonal of H with the smallest global-key (i.e., it is in the root of H). We assume $dis(d^*) = 2i + 1$ for some integer i. If we were running the DN algorithm, we are currently working on the i-th phase. Let S be the set of all diagonals in H that have the same global-key as d^*. The diagonals of S can be found by continuing the extract-min operations on H in $O(|S| \log |H|)$ time, and after that, diagonals of S are removed from H.

There are two cases depending on whether $B(d^*) = \emptyset$.

$B(d^*) \neq \emptyset$. In this case, all diagonals of S have non-empty beam sets. Due to our corridor structure, the sweeps of the i-th phase "paused" at the diagonals in S. To continue the i-th phase, we "resume" the sweeps from these diagonals. Unlike the DN algorithm where we complete the right-sweep before we start the left-sweep, here, before the pause, we may have already done some left-sweep and right-sweep. Hence, the two sweeps may be somehow "interleaved" and our algorithm will need to take care of this situation.

Let S_R (resp., S_L) be the subset of the diagonals of S whose beams are rightward (resp., leftward). Intuitively, the right-sweep (resp., left-sweep) paused at the diagonals in S_R (resp., S_L), and thus, we resume it from the diagonals in S_R (resp., S_L). Below we focus on the right-sweep.

We build another min-heap H_R by inserting the diagonals of S_R, and the "keys" of diagonals in H_R are their x-coordinates such that the leftmost diagonal is at the root. (Similarly, we build a min-heap H_L on S_L for the left-sweep.)

The algorithm essentially performs the i-th phase as the DN algorithm. But since here the right-sweep and left-sweep may be interleaved, some diagonals may have two sets of beams with opposite directions. However, our algorithm makes sure that if a diagonal d has two sets of beams with opposite directions, it will not be in H (i.e., it has been removed from H), but in both H_R and H_L if d has not been processed yet. To differentiate the two sets of beams, we use $B_r(d)$ (resp., $B_l(d)$) to denote the beam set of any diagonal d in H_R (resp., H_L), meaning that the direction of the beams is rightward (resp., leftward).

During the right-sweep, if we find a new diagonal d that has the same global-key as d^*, then d will be inserted to H_R and d will be removed from H if it is already in H. Hence, all diagonals of H_R have the same global-key as d^*.

As long as H_R is not empty, we repeatedly do the following.

We obtain the leftmost diagonal d of H_R (which is at the root of H_R) and remove it from H_R. The beams of $B_r(d)$ may enter a junction cell or a corridor. If it is the former case, our way of processing d is similar to the DN algorithm, although we need to take care of the situation that the left and right sweeps are interleaved; we skip the details, which can be found in Appendix B.

Next, we consider the case where beams of $B_r(d)$ enter a corridor $\mathcal{C}$. We process $\mathcal{C}$ using the beams of $B_r(d)$. One may assume we still use the DN algorithm to process $\mathcal{C}$, and later we will replace it by our corridor-processing algorithm in Appendix C. Let δ be the distance value labeled on the other door d' of $\mathcal{C}$ by the

above processing and let B' denote the corresponding beam set on d'. Let $dis(d')$ and $B(d')$ be the original distance value and beam set at d' before processing $\mathcal{C}$. By the third algorithm invariant, this is the first time $\mathcal{C}$ is processed in the direction from d to d'. Hence, if $dis(d') \neq \infty$, the value $dis(d')$ must be obtained by the sweep from outside $\mathcal{C}$, i.e., beams in $B(d')$ are towards the inside of $\mathcal{C}$.

Due to the above processing of $\mathcal{C}$, we have obtained another distance value δ and beam set B' for d'. We need to update the label of d' and possibly insert d' to some heap. Depending on the value of $dis(d')$, there are several cases.

1. If $dis(d')$ is ∞, then we set $dis(d') = \delta$ and $B(d') = B'$.
   If $\delta > 2i + 1$, then we insert d' into H.
   If $\delta = 2i + 1$, since $dis(d) = 2i + 1$, d' must be illuminated directly by the beams in $B_r(d)$ and the beams of $B(d')$ are still towards right. By Observation 1, $B' \neq \emptyset$. Hence we obtain $B_r(d') = B' \neq \emptyset$ (we set $B_r(d')$ to B' because the beams of B' are rightward). Finally, we insert d' into H_R.
2. If $dis(d') < 2i+1$, then the global-key of d' is smaller than that of d^* because $dis(d^*) = 2i + 1$. By the first algorithm invariant, d' has been correctly labeled. Recall that the direction of $B(d')$ is towards the inside of $\mathcal{C}$. Also by the first algorithm invariant, d is correctly labeled, and the direction of the beams of $B_r(d)$ is towards the inside of $\mathcal{C}$. This means we have computed complete information on the two doors of $\mathcal{C}$ for the min-link v-v-paths from s to the points inside $\mathcal{C}$. Hence, we can do a "post-processing" step to compute the v-v-map in $\mathcal{C}$ by using the beams of $B_r(d)$ and $B(d')$. We will give a *corridor-post-processing* algorithm for this step in Appendix C.
3. If $dis(d') = 2i + 1$, then d' has been labeled in the current phase.
   If $B(d') \neq \emptyset$, then the global-key of d' is the same as that of d^*. By the first algorithm invariant, d' has been correctly labeled. As above, since both d and d' have been correctly labeled, we do a "post-processing" to compute the v-v-map in $\mathcal{C}$ using $B_r(d)$ and $B(d')$.
   If $B(d') = \emptyset$, then the global-key of d' is strictly larger than that of d^*. By the second algorithm invariant, d' is already in H. If $\delta > 2i + 1$, we do nothing. If $\delta = 2i+1$, then as in the above first case, we set $B_r(d') = B' \neq \emptyset$. Finally, we insert d' into H_R and remove d' from H.
4. The remaining case is when $dis(d') \neq \infty$ and $dis(d') > 2i + 1$. It can be shown that this case cannot happen (see Appendix B for the details).

The above finishes the processing of the diagonal d. The right-sweep procedure is done after the heap H_R becomes empty. Afterwards we do the left-sweep from the diagonals of S_L using the heap H_L in the completely symmetric way.

This finishes our algorithm in the case in which $B(d^*)$ is not empty.

$B(d^*) = \emptyset$. In this case, according to our way of comparing global-keys, all diagonals of S have empty beam sets. If we were running the DN algorithm, the diagonals of S would be locally-outmost and we would be about to start the $(i + 1)$-th phase (not the i-th phase). As in the previous case, we run the two sweep procedures starting from the diagonals of S. Let S_R (resp., S_L) be the subset of diagonals of S whose beam directions are rightward (resp., leftward).

We build a min-heap H_R (resp., H_L) on the diagonals of S_R (resp., S_L). Below, we only discuss the right-sweep since the left-sweep is similar.

Since we are doing the right-sweep in the $(i+1)$-th phase, each diagonal of S_R will generate a beam from the entire diagonal, and all new diagonals illuminated in the right-sweep will get distance $2i+3$ instead of $2i+1$. From now on, we associate each diagonal of S_R with the beam, i.e., for each $d \in S_R$, $B_r(d) = \{d\}$. Since each diagonal of S_R originally got an empty beam set (at the end of the i-th phase), by Observation 1, the beam of $B_r(d)$ cannot be towards a junction cell and thus it must be towards the inside of a corridor.

As long as H_R is not empty, we repeatedly do the following.

We obtain the leftmost diagonal d of H_R (which is at the root) and remove it from H_R. Let $\mathcal{C}$ denote the corridor that the beam of $B_r(d)$ enters. We process the corridor $\mathcal{C}$ using the beams of $B_r(d)$. The rest of the algorithm is very similar as in the case $B(d^*) \neq \emptyset$ and the details can be found in Appendix B.

This finishes our discussion in the case in which $B(d^*)$ is empty.

The algorithm finishes after all three heaps H, H_L, and H_R become empty. After that, for each diagonal d in a junction cell, $dis(d)$ and $B(d)$ have been correctly computed. During the algorithm some corridors have been labeled correctly while others are left for post-processing. Specifically, consider any corridor $\mathcal{C}$ and let d_1 and d_2 be its two doors with their beam sets $B(d_1)$ and $B(d_2)$. If $\mathcal{C}$ is not left for a post-processing, then $\mathcal{C}$ has been processed either from d_1 to d_2 or from d_2 to d_1 and the v-v-map in $\mathcal{C}$ has been computed after the processing. Suppose the above processing is from d_1 and d_2. Then, $\mathcal{C}$ is processed using the beams of $B(d_1)$, and $B(d_2)$ is obtained after the processing. We show in Appendix C that our corridor-processing algorithm on $\mathcal{C}$ runs in $O(m+(h_1-h_2+1)\log h_1)$ time, where m is the number of vertices of $\mathcal{C}$, $h_1 = |B(d_1)|$, and $h_2 = |B(d_2)|$. If $\mathcal{C}$ is left for a post-processing, i.e., to compute the v-v-map in $\mathcal{C}$ by using $B(d_1)$ and $B(d_2)$, we show in Appendix C that our corridor-post-processing algorithm runs in $O(m + h_1 \log h_1 + h_2 \log h_2)$ time. These two algorithms in Appendix C may have independent interest. This leads to the following lemma.

Lemma 1. *Given $VD(\mathcal{F})$, our algorithm computes the v-v-map on $VD(\mathcal{F})$ in $O(n + h \log h)$ time.*

The other three maps can be computed similarly. For computing the h-v-map on $VD(\mathcal{F})$, one difference is on the initial steps. Initially, we let s generate two beams that are two horizontal rays towards right and left, repectively. We set the distance value of d_s to 2, where d_s is the vertical diagonal through s. Then, we consider d_s as two duplicate diagonals associated with the above two beams respectively, and insert the two duplicate diagonals into the heap H. The remaining algorithm is the same as before except that we replace the distance values $2i+1$ and $2i+3$ in the algorithm description with $2i+2$ and $2i+4$, respectively. The h-h-map and v-h-map on $HD(\mathcal{F})$ can be computed similarly.

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. B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.
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 (SoCG)*, pages 406–415, 2014.
4. 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 (ESA)*, pages 481–492, 2011.
5. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3nd edition, 2009.
6. G. Das and G. Narasimhan. Geometric searching and link distance. In *Proc. of the 2nd Workshop of Algorithms and Data Structures (WADS)*, pages 261–272, 1991.
7. M. de Berg. On rectilinear link distance. *Computational Geometry: Theory and Applications*, 1:13–34, 1991.
8. H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.
9. S.K. Ghosh. Computing the visibility polygon from a convex set and related problems. *Journal of Algorithms*, 12:75–95, 1991.
10. J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry: Theory and Applications*, 4:63–97, 1994.
11. H. Imai and T. Asano. Efficient algorithms for geometric graph search problems. *SIAM Journal on Computing*, 15(2):478–494, 1986.
12. S. Kapoor, S.N. Maheshwari, and J.S.B. Mitchell. An efficient algorithm for Euclidean shortest paths among polygonal obstacles in the plane. *Discrete and Computational Geometry*, 18(4):377–383, 1997.
13. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.
14. A. Lingas, A. Maheshwari, and J.-R. Sack. Parallel algorithms for rectilinear link distance problems. *Algorithmica*, 14:261–289, 1995.
15. A. Maheshwari, J.-R Sack, and H.N. Djidjev. Link distance problems, in *Handbook of Computational Geometry,* J.-R Sack and J. Urrutia (eds.), pages 519–558. Elsevier, Amsterdam, the Netherlands, 2000.
16. J.S.B. Mitchell, V. Polishchuk, and M. Sysikaski. Minimum-link paths revisited. *Computational Geometry: Theory and Applications*, 47:651–667, 2014.
17. J.S.B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.
18. J.S.B. Mitchell and S. Suri. Separation and approximation of polyhedral objects. *Computational Geometry: Theory and Applications*, 5:95–114, 1995.
19. 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.
20. S. Schuierer. An optimal data structure for shortest rectilinear path queries in a simple rectilinear polygon. *International Journal of Compututational Geometry and Applications*, 6:205–226, 1996.
21. S. Suri. A linear time algorithm with minimum link paths inside a simple polygon. *Computer Vision, Graphics, and Image Processing*, 35(1):99–110, 1986.
22. S. Suri. *Minimum link paths in polygons and related problems.* PhD thesis, Johns Hopkins University, Baltimore, MD, 1987.
23. S. Suri. On some link distance problems in a simple polygon. *IEEE Transactions on Robotics and Automation*, 6:108–113, 1990.

APPENDIX

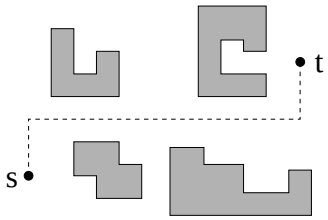

Fig. 5. Illustrating a rectilinear min-link path in a rectilinear polygonal domain.

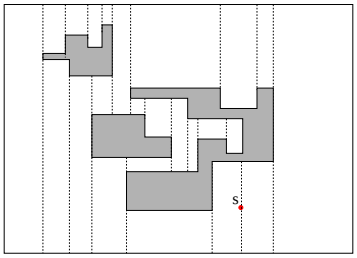

Fig. 6. Illustrating the vertical visibility decomposition of $\mathcal{F}$.

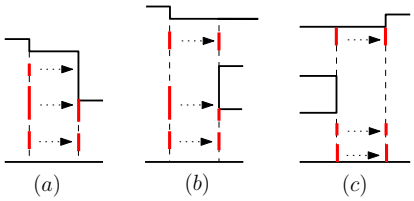

Fig. 7. Illustrating some beam operations in a right-sweep procedure: In (a) and (b), beams are *split*, and some beams are "narrowed" and some beams "terminate" at obstacle edges; in (c), beams are *merged*.

Lemma 2. *If $\hat{d}$ has been labeled, then $dis(\hat{d})$ must be $2i+1$.*

Proof. Note that since we are currently at the i-th phase, any diagonal that has been labeled must have a distance value at most $2i+1$. Assume to the contrary that $dis(\hat{d}) < 2i+1$. Then $\hat{d}$ has already been processed, and since $\hat{d}$ and d are on the same cell side, we must have already obtained $dis(d) = dis(\hat{d}) < 2i+1$, contradicting with $dis(d) = 2i+1$. Hence, $dis(\hat{d}) = 2i+1$. □

A Labeling Cells and Retrieving Paths

In fact, the DN algorithm in [6] only labels the diagonals. Das and Narasimhan [6] claimed that queries can be answered by using only the distance values on the diagonals. However, this is not clear to us. We, instead, also label the cells as discussed before. It turns out that this can be done by a slight modification of the DN algorithm (described in Section 2.1), as follows.

We only discuss the v-v-map since the other maps are similar.

Recall that for each cell C of $VD(\mathcal{F})$, all the points in C that are not on the diagonals have the same distance values. Hence, to compute $dis_{vv}(C)$, it is sufficient to know the distance value for any arbitrary point in C that is not on any diagonal. To this end, for each cell C, we add a vertical segment in C with its upper endpoint on the upper side of C and its lower endpoint on the lower side of C such that the segment is not overlapped with the left side or the right side of C; hence, no point of the segment is on any diagonal, and we call the segment a "fake diagonal". Let $VD'(\mathcal{F})$ denote $VD(\mathcal{F})$ with all fake diagonals. We run the DN algorithm on $VD'(\mathcal{F})$ and treat all fake diagonals as "ordinary diagonals" to label all diagonals and fake diagonals of $VD'(\mathcal{F})$. Finally, we label each cell of $VD(\mathcal{F})$ with the same distance value as its fake diagonal. Since the total number of fake diagonals are $O(n)$, the running time is still $O(n \log n)$ and the space is $O(n)$.

To obtain an actual min-link v-v-path from s to t, we need to maintain additional information on our v-v-map. No details on this are given in [6]. We briefly discuss it below for the completeness of this paper. Essentially, when we label the cell sides (and the fake diagonals), we need to record how we reach there. Specifically, suppose we label a diagonal d on the right side of a cell in a right-sweep procedure due to a beam from a diagonal on the left side of the cell; then we record any such beam at d (it is sufficient to record any point on d in the beam) along with its generator, and in the case where $B(d)$ is empty, d is a locally-outmost diagonal and the path should make a turn there. Further, for each locally-outmost diagonal d, it is reached by a beam that illuminates the cell side containing d and we record that beam for d so that we know the turn on d is for that beam. With this path information, for any query point t, we can easily retrieve an actual min-link v-v-path from t back to s in time we claimed before. We omit the details.

B The Algorithm

To help the reader understand the algorithm, we first describe the first few steps of the algorithm and then delve into the details of the general steps.

Initially, we set $dis(d)$ to ∞ and $B(d) = \emptyset$ for each diagonal d except that $dis(d_s) = 1$ and $B(d_s) = \{d_s\}$, where d_s is the diagonal through s. In our discussion of the corridor structure, d_s is considered as a degenerate junction cell. Here, for ease of discussion, we assume d_s is in the interior of a corridor $\mathcal{C}(s)$ as shown in Fig. 4. We first process $\mathcal{C}(s)$, i.e., label all diagonals in $\mathcal{C}(s)$.

For the purpose of describing our algorithm, one may assume we still use the DN algorithm to process $\mathcal{C}(s)$, and later we will replace the DN algorithm by our new and more efficient corridor-processing algorithm in Appendix C.

Denote by D_x and D_y the two doors of $\mathcal{C}(s)$ (e.g., see Fig. 4). Suppose after the above processing of $\mathcal{C}(s)$, $dis(D_x) = 2i + 1$ and $dis(D_y) = 2j + 1$ for some integers i and j. Since $dis(D_x)$ is obtained "locally" in $\mathcal{C}(x)$, i.e., $dis(D_x)$ is the link distance of the *local* min-link v-v-path from s to D_x in $\mathcal{C}(s)$, it may not be "set correctly", i.e., it may not be the link distance of the *global* min-link v-v-path from s to D_x in $\mathcal{F}$. The value $dis(D_y)$ has the same issue. However, we show below the at least one of $dis(D_x)$ and $dis(D_y)$ must have been set correctly.

Without loss of generality (WLOG), assume $j \leq i$.

Observation 2 *If $j \leq i$, then the value $dis(D_y)$ has been set correctly.*

Proof. Let π be any global min-link v-v-path from s to D_y in $\mathcal{F}$. For any subpath π' of π, we use $dis(\pi')$ to denote the link distance of π'. If π is in $\mathcal{C}(s)$, then our observation follows. Otherwise, π must cross one of the doors of $\mathcal{C}(s)$. Let p be the first point on a door of the corridor if we go from s to D_y along π, and let $\pi(s, p)$ denote the sub-path of π from s to p. If p is on D_y, then $\pi(s, p)$ is a min-link v-v-path from s to D_y. Since $\pi(s, p) \in \mathcal{C}(s)$ and $dis(D_d)$ is the link distance of the local min-link v-v-path from s to D_y in $\mathcal{C}(s)$, we have $dis(\pi) \geq dis(\pi(s, p)) \geq dis(D_y)$, and thus our observation follows. If p is on D_x, then $dis(\pi(s, p)) = 2i + 1 \geq 2j + 1$. Therefore, $dis(\pi) \geq dis(\pi(s, p)) \geq 2j + 1 = dis(D_y)$ and our observation also follows. □

Another way to see the observation is that if we were running the DN algorithm, after the $(j - 1)$-th phase, neither D_x nor D_y is labeled, and no diagonal outside the corridor is labeled either. In the j-th phase, D_y will be labeled and thus $dis(D_y) = 2j + 1$ is set correctly.

After the processing of $\mathcal{C}(s)$, the beams on D_y, i.e., $B(D_y)$, have also been obtained. The next step is to process the diagonal D_y by propagating the beams of D_y outside the corridor.

Next, we describe the details of the general steps of our algorithm.

We use a min-heap H to maintain the diagonals in all junction cells where the "keys" are the distance values the diagonals currently have (and these values may not be set correctly), with the *smallest* key at the root of H. Since there are $O(h)$ junction cells, the size of H is $O(h)$. Each diagonal d in H is also associated with its beam set $B(d)$ (along with its direction). As in the DN algorithm, it is possible that $B(d)$ is empty, in which case d might be a locally-outmost diagonal, but it is also associated with a direction for generating a beam towards that direction in the next phase. The algorithm is similar in spirit to Dijkstra's algorithm and the heap H plays the same rule as the priority queue in Dijkstra's algorithm.

If some diagonals of H have the same keys, we break the ties by applying the following rules. Consider two diagonals d_1 and d_2 in H with $dis(d_1) = dis(d_2)$. If $B(d_1)$ and $B(d_2)$ are both empty or both non-empty, then we break ties arbitrarily. Otherwise, WLOG, assume $B(d_1) \neq \emptyset$ but $B(d_2) = \emptyset$. Then, we consider the key of d_1 to be *smaller* than that of d_2. The reason is as follows.

Since $B(d_1) \neq \emptyset$ and $B(d_2) = \emptyset$, the current sweep procedure should be over *before* processing d_2 while the sweep should continue *after* processing d_1, and thus, we should process d_1 before d_2. Therefore, our way of resolving ties in H is crucial and consistent with the DN algorithm. In the following, to differentiate from keys of other heaps, for any diagonal d, even if d is not in H, we consider $dis(d)$ along with $B(d)$ as the *global-key* of d, and whenever we compare the global-keys of diagonals, we follow the above rules to break ties.

Consider any corridor $\mathcal{C}$ with two doors d and d'. Suppose the beams of $B(d)$ are going inside $\mathcal{C}$, and we want to *process* $\mathcal{C}$ (i.e., compute the v-v-map in $\mathcal{C}$) using the beams of $B(d)$. We say the above way of processing $\mathcal{C}$ is in the *direction* from d to d'. As will be seen later, a corridor may be processed twice: once from d to d' and the other from d' to d. Due to the special geometric structure of the corridor, we have the following observation that will be useful later.

Observation 1 *Suppose d and d' are the two doors of a corridor $\mathcal{C}$, and the direction of the beams of $B(d)$ is towards the inside of $\mathcal{C}$. Then after $\mathcal{C}$ is processed by using $B(d)$, the beam set of d' is not empty.*

Proof. Let $VD(\mathcal{C})$ denote the vertical visibility decomposition of the corridor $\mathcal{C}$. Consider the cell C of $VD(\mathcal{C})$ that contains d'. Without loss of generality, assume d' is on the right side C. Denote by e_r the right side of C.

We claim that d' is the entire right side of C, i.e., d' is e_r. Indeed, according to our corridor structure, d' is an extension of a vertical obstacle edge e and one endpoint p of e is also an endpoint of d' and the other endpoint of e is outside the corridor $\mathcal{C}$. This means p is also an endpoint of e_r and the other endpoint q of d' than p is not on e but on another obstacle edge e'. Due to our general position assumption that no two vertical edges are collinear, q must be in the interior of e', implying that q is also an endpoint of e_r. Hence, we obtain $d' = e_r = \overline{pq}$.

Note that we obtain the beam set of d' from the rightward beams of the diagonals on the left side of C. Now that d' is the entire right side of C, d' will receive all beams of any diagonal on the left side of C. Hence, the beam set of d' cannot be empty. □

To show the correctness of our algorithm, we will argue that our algorithm is consistent with the DN algorithm. We will show that after the algorithm finishes, each diagonal in any junction cell is *correctly labeled*, i.e., both its distance value and its beam set are the same as those in the DN algorithm. Let d^* be the diagonal in the root of H. Our algorithm will maintain the following three invariants.

1. The diagonal d^* is correctly labeled. Further, for any other diagonal d in a junction cell, if the global-key of d is no larger than that of d^*, then d has been correctly labeled.
2. For any diagonal d in a junction cell, if $dis(d) \neq \infty$ and the global-key of d is larger than that of d^*, then d is in H.
3. For any corridor $\mathcal{C}$ with two doors d and d', if $\mathcal{C}$ is processed in the direction from d to d', then $\mathcal{C}$ will never be processed from d to d' again in the entire

algorithm (although $\mathcal{C}$ may be processed later in the other direction from d' to d).

Initially $H = \emptyset$. Recall that $dis(d_s) = 1$ and $B(d) = \{d_s\}$. We consider the diagonal d_s through s as a degenerate junction cell. Specifically, we consider d_s as two duplicate diagonals with one generating a rightward beam and the other generating a leftward beam from the entire d_s. We insert these two diagonals into H. Clearly all algorithm invariants hold. In the following we will describe the details of our algorithm. To avoid the tedious discussion, we will not explicitly explain that the algorithm invariants are maintained after each step, but rather discuss it briefly later after the algorithm description.

As long as H is not empty, we repeatedly do the following.

By an extract-min operation on H, we obtain the diagonal d^* of H with the smallest global-key and remove it from H. We assume $dis(d^*) = 2i + 1$ for some integer i. If we were running the DN algorithm, we are currently working on the i-th phase. There are two cases depending on whether $B(d^*) = \emptyset$.

$B(d^*) \neq \emptyset$ We first discuss our algorithm for the case where $B(d^*) \neq \emptyset$. If we were running the DN algorithm, we would be in the "middle" of the i-th phase (because $B(d^*) \neq \emptyset$ implies that all locally-outmost diagonals have already been processed). So we should continue the left-sweep and right-sweep of the i-th phase. The first question is where we should start the sweep. Let S be the set of all diagonals in H that have the same global-keys as d^* (according to our way of comparing global-keys, for each diagonal $d \in S$, it holds that $dis(d) = dis(d^*)$ and $B(d) \neq \emptyset$). By the first algorithm invariant, all diagonals in S have been correctly labeled. Due to our corridor structure, the sweeps of the i-th phase "paused" at the diagonals in S. To continue the i-th phase, we "resume" the sweeps from these diagonals. Unlike the DN algorithm where we complete the right-sweep before we start the left-sweep, here, before the pause, we may have already done some left-sweep and right-sweep. Hence, the two sweeps may be somehow "interleaved" and our algorithm will need to take care of this situation.

Due to our way of breaking ties in H, the diagonals of S can be found by keeping doing the extract-min operations on H in $O(|S| \log |H|)$ time (i.e., all diagonals of S are removed from H). We also let S contain d^*. Let S_R (resp., S_L) be the subset of the diagonals of S whose beams are rightward (resp., leftward). Intuitively, the right-sweep (resp., left-sweep) paused at the diagonals in S_R (resp., S_L), and thus, we resume it from the diagonals in S_R (resp., S_L). Below we focus on the right-sweep, and the left-sweep is very similar.

Recall that in the right-sweep of the DN algorithm we use a heap H_R to guide the procedure. Here we do the same thing and process the diagonals of H_R from left to right. We build a min-heap H_R by inserting the diagonals of S_R, and the "keys" of diagonals in H_R are their x-coordinates such that the leftmost diagonal is at the root. (Similarly, we build a min-heap H_L on S_L for the left-sweep.)

The algorithm essentially performs the i-th phase as the DN algorithm. But since here the right-sweep and left-sweep may be interleaved, in the following

discussion, some diagonals may have two sets of beams with opposite directions. However, our algorithm makes sure that if a diagonal d has two sets of beams with opposite directions, it will not be in H (i.e., it has been removed from H), but in both H_R and H_L if d has not been processed yet. To differentiate the two sets of beams, we use $B_r(d)$ (resp., $B_l(d)$) to denote the beam set of any diagonal d in H_R (resp., H_L), meaning that the direction of the beams is rightward (resp., leftward).

During the right-sweep, if we find a new diagonal d that has the same global-key as d^*, then d will be inserted to H_R and d will be removed from H if it is already in H. Hence, all diagonals of H_R have the same global-key as d^*. In fact, H_R maintains all diagonals in junction cells that will be processed in the following right-sweep of the i-th phase.

As long as H_R is not empty, we repeatedly do the following.

We obtain the leftmost diagonal d of H_R (which is at the root of H_R) and remove it from H_R. The beams of $B_r(d)$ may enter a junction cell or a corridor. In the sequel, we discuss how to process d in these two cases.

The beams of $B_r(d)$ entering a junction cell. Let C denote the junction cell that the beams of $B_r(d)$ enter. In this case, our way of processing d is similar to the DN algorithm. Let e_l and e_r denote the left and right sides of C, respectively. Note that d is on the left side e_l. Recall that each cell side may have two diagonals. To process d, we update the labels of all other diagonals of C, as follows.

Suppose there is another diagonal $\hat{d}$ on e_l (e.g., see Fig. 2).

If $dis(\hat{d}) = \infty$, we set $dis(\hat{d}) = dis(d)$ and $B(\hat{d}) = \emptyset$ with the leftward direction. Then, we insert $\hat{d}$ into H.

If $dis(\hat{d}) \neq \infty$ but $dis(\hat{d}) > dis(d)$, then we set $dis(\hat{d}) = dis(d)$ and $B(\hat{d}) = \emptyset$ (with the leftward direction). Since $dis(\hat{d})$ was greater than $dis(d)$ but not ∞, by the second algorithm invariant, $\hat{d}$ is already in the heap H. Hence, after $dis(\hat{d})$ and $B(\hat{d})$ are reset as above, we do a "decrease-key" operation on $\hat{d}$ in H since the global-key of $\hat{d}$ has been decreased.

If $dis(\hat{d}) \leq dis(d)$, we do nothing.

Next, we consider the diagonals on the right side e_r of C. Depending on the distance value $dis(e_r)$, there are several cases. Note that if $dis(e_r) \neq \infty$, then e_r automatically got the value $dis(e_r)$ because a diagonal d' on e_r was labeled with the same distance value.

1. If $dis(e_r) = \infty$, then we set $dis(e_r) = 2i+1$. If e_r does not have any diagonals, then we are done with processing d. Otherwise, for each diagonal d' on e_r, we set $dis(d') = 2i + 1$ and determine $B_r(d) \cap d'$, i.e., the portion of $B_r(d)$ that can illuminate d'.

   If $B_r(d) \cap d' \neq \emptyset$, then we set $B_r(d') = B_r(d) \cap d'$ and insert d' to H_R (note that d' has the same global-key as d^*). Otherwise, we set $B(d') = \emptyset$ with the rightward direction and insert d' into H, because the global-key of d' is larger than that of d^* and the right-sweep should stop at d'. Note that before the insertion, d' was not in H as $dis(d')$ was ∞.

2. If $dis(e_r) \neq \infty$ but $dis(e_r) > 2i+1$, then the algorithm is similar as above. For each diagonal d' on e_r, $dis(d')$ was equal to $dis(e_r)$, and now we set $dis(d') = 2i+1$ and determine $B_r(d) \cap d'$.
   If $B_r(d) \cap d' \neq \emptyset$, we set $B_r(d') = B_r(d) \cap d'$ and insert d' into H_R. Since $dis(d')$ was equal to $dis(e_r)$, the global-key of d' was larger than that of d^* and $dis(d')$ was not ∞. By the second algorithm invariant, d' was already in H. Hence, we remove d' from H. Note that we can perform a remove operation on d' in H by doing a decrease-key operation followed by an extract-min operation [5].
   If $B_r(d) \cap d' = \emptyset$, we set $B(d') = \emptyset$. Since d' was already in H, after setting $dis(d') = 2i+1$ and $B(d') = \emptyset$, we need to do a decrease-key operation on d' in H.
3. If $dis(e_r) < 2i+1$, we do nothing.
4. If $dis(e_r) = 2i+1$, due to that the left and right sweeps are interleaved, e_r may have got labeled from the right-sweep, the left-sweep, or both. We discuss the three cases below. The algorithm for this case is also simple, but we need a few more words to explain why the algorithm works in that way.

   (a) If e_r got the value $dis(e_r)$ by the right-sweep only, as in our discussions in the right-sweep of the DN algorithm, this case happens because e_r was illuminated by beams from another diagonal $\hat{d}$ on the left side e_l. Hence, for each diagonal d' on e_r, $dis(d')$ and $B_r(d')$ have been set. Further, if $B_r(d') = \emptyset$, then d' is in H; otherwise d' is in H_R.
   For each diagonal d' on e_r, we first determine $B_r(d) \cap d'$, and then set $B_r(d')$ by merging the original $B_r(d')$ with $B_r(d) \cap d'$. If $B_r(d')$ was non-empty before the merge, we do nothing since it is already in H_R. If $B_r(d')$ was empty before the merge and becomes non-empty after the merge, then we insert d' into H_R and remove it from H. If $B_r(d')$ is still empty after the merge, then we do nothing since it is already in H.

   (b) If e_r got the value $dis(e_r)$ by the left-sweep only, then e_r was illuminated by beams from its right. As discussed in the DN algorithm, if the right-sweep procedure sweeps a diagonal d' on e_r, the fact that d' was swept already by the left-sweep procedure should be ignored in the sense that we should keep propagating the beams of $B_r(d)$ to the right of d'. The details are given below.
   For each diagonal d' on e_r, we first determine $B_r(d) \cap d'$.
   - If $B_r(d) \cap d' \neq \emptyset$, then we set $B_r(d') = B_r(d) \cap d'$ and insert d' into H_R. If d' was already in H, then we remove it from H.
   - If $B_r(d) \cap d' = \emptyset$, no beam from $B_r(d)$ illuminates d'. At first sight, it seems that we should insert d' into H with an empty beam set and the rightward direction. Below we elaborate on whether we should do so.
     Note that d' is a door of a corridor $\mathcal{C}'$ that is locally on the right of d'. It is possible that e_r got labeled because the left-sweep was from $\mathcal{C}'$, in which case $B_l(d')$ is not empty by Observation 1, and thus, d' must be already in H_L (since $dis(d') = 2i+1$) and we do not need

to insert d' to H because d' will be processed in the left-sweep of the current phase.

On the other hand, if e_r has another diagonal d'', then it is possible that e_r got labeled because of the processing of the corridor on the right of d'' during the left-sweep. In this case, d' may have got labeled because of the processing of d'', in which case as in the DN algorithm the beam set of d' must be empty and rightward, and thus d' is already in H with $B(d') = \emptyset$ and the rightward direction and we do not insert d' into H again. But if d' has not been labeled yet, then we insert d' into H with $B(d') = \emptyset$ and the rightward direction. Therefore, for the case where e_r has another diagonal d'', if d' is already in H, we do nothing; otherwise we insert d' to H.

It is also possible that e_r got labeled "simultaneously" because of the processing of the two corridors on the right of d' and d'', in which case by Observation 1 we again have $B_l(d') \neq \emptyset$, and thus d' is already in H_L. Hence, we do not need to insert d' to H.

In summary, for the case $B_r(d) \cap d' = \emptyset$, if d' is in neither H_L nor H, then we insert d' into H with $B(d') = \emptyset$ and the rightward direction; otherwise we do nothing.

(c) If e_r got the value $dis(e_r)$ by both the right-sweep and the left-sweep, this is a combination case of the above two cases.

For each diagonal d' on e_r, we determine $B_r(d) \cap d'$, and then set $B_r(d')$ by merging the original $B_r(d')$ with $B_r(d) \cap d'$.

If $B_r(d')$ was non-empty before the merge, then we do nothing since d' is already in H_R.

If $B_r(d')$ was empty before the merge and becomes non-empty after the merge, then we insert d' into H_R. Further, if d' is in H, then we remove it from H.

If $B_r(d')$ is still empty after the merge, as in the above second case, we do the following. If d' is in neither H_L nor H, then we insert d' into H with $B(d') = \emptyset$ and the rightward direction; otherwise we do nothing.

We are done with processing d when the beams $B_r(d)$ of d enter a junction cell.

The beams of $B_r(d)$ entering a corridor. Let $\mathcal{C}$ denote the corridor that the beams enter. We process $\mathcal{C}$ using the beams of $B_r(d)$. Again, one may assume we still use the DN algorithm to process $\mathcal{C}$, and later we will replace it by our corridor-processing algorithm in Section C.

Let δ be the distance value labeled on the other door d' of $\mathcal{C}$ by the above processing and let B' denote the corresponding beam set on d'. Let $dis(d')$ and $B(d')$ be the original distance value and beam set at d' before the processing of $\mathcal{C}$. By the third algorithm invariant, this is the first time $\mathcal{C}$ is processed in the direction from d to d'. Hence, if $dis(d') \neq \infty$, then the value $dis(d')$ must be obtained by the sweep from outside $\mathcal{C}$, i.e., beams in $B(d')$ are towards the inside of $\mathcal{C}$.

Due to the above processing of $\mathcal{C}$, we have obtained another distance value δ and beam set B' for d'. Hence, we need to update the label of d' and possibly insert d' to some heap. Depending on the value of $dis(d')$, there are several cases.

1. If $dis(d')$ is ∞, then we set $dis(d') = \delta$ and $B(d') = B'$. If $\delta > 2i+1$, then we insert d' into H. If $\delta = 2i+1$, since $dis(d) = 2i+1$, d' must be illuminated directly by the beams in $B_r(d)$ and the beams of $B(d')$ are still towards right. By Observation 1, $B' \neq \emptyset$. Hence we obtain $B_r(d') = B' \neq \emptyset$ (we set $B_r(d')$ to B' because the beams of B' are rightward). Finally, we insert d' into H_R.
2. If $dis(d') < 2i+1$, then the global-key of d' is smaller than that of d^* because $dis(d^*) = 2i+1$. By the first algorithm invariant, d' has been correctly labeled. Recall that the direction of $B(d')$ is towards the inside of $\mathcal{C}$. Also by the first algorithm invariant, d is correctly labeled, and the direction of the beam set of $B_r(d)$ is towards the inside of $\mathcal{C}$. This means we have computed complete information on the two doors of $\mathcal{C}$ for the min-link v-v-paths from s to the points inside $\mathcal{C}$. Hence, we can do a "post-processing" step to compute the v-v-map in $\mathcal{C}$ by using the beams of $B_r(d)$ and $B(d')$. We will give a *corridor-post-processing* algorithm for this step later in Section C.
3. If $dis(d') = 2i+1$, then d' has been labeled in the current phase.
   If $B(d') \neq \emptyset$, then the global-key of d' is the same as that of d^*. By the first algorithm invariant, d' has been correctly labeled. Then, as above, since both d and d' have been correctly labeled, we do a "post-processing" to compute the v-v-map in $\mathcal{C}$ using $B_r(d)$ and $B(d')$.
   If $B(d') = \emptyset$, then the global-key of d' is strictly larger than that of d^*. By the second algorithm invariant, d' is already in H. If $\delta > 2i+1$, then we do nothing. If $\delta = 2i+1$, then as in the above first case, we set $B_r(d') = B' \neq \emptyset$. Finally, we insert d' into H_R and remove d' from H.
4. The remaining case is when $dis(d') \neq \infty$ and $dis(d') > 2i+1$. We claim that this case can never happen.
   Indeed, assume to the contrary that this case happens. Recall that the beams of $B(d')$ are towards the inside of the corridor $\mathcal{C}$. Let C be the junction cell that contains d'. Without loss of generality, assume d' is on the left side of C. Clearly, d' got labeled after some diagonal d'' in C was processed. Since the beams of d'' that illuminate d' must be towards the cell C, they are from the corridor that is bounded by d''. By Observation 1, the beam set of d'' is not empty. Hence, it must be the case that $dis(d'') = dis(d') > 2i+1$. Since we use the heap H to guide the main algorithm and we are currently processing the diagonal d with $dis(d) = 2i+1$, all diagonals of junction cells that have been processed must have distance values at most $2i+1$. However, that the diagonal d'' has been processed with $dis(d'') > 2i+1$, incurring contradiction. Therefore, the case where $dis(d') \neq \infty$ and $dis(d') > 2i+1$ cannot happen.

The above finishes the processing of the diagonal d. The right-sweep procedure is done after the heap H_R becomes empty. Afterwards we do the left-sweep

from the diagonals of S_L using the heap H_L in the completely symmetric way. We omit the details.

This finishes our algorithm in the case in which $B(d^*)$ is not empty for the diagonal d^* at the root of H. We briefly discuss that after the above processing our algorithm invariants still hold.

Indeed, before the diagonals of S_R are processed, if we were running the DN algorithm, at some moment during the i-th phase, the diagonals of S_R would be labeled the same as in our algorithm. Our algorithm processes the diagonals of S_R in a way consistent with the DN algorithm. Based on our previous discussion on how the sweep procedures of the DN algorithm can sweep the corridors, after all diagonals of S_R are processed, the diagonals in H with the smallest global-key must have been correctly labeled because if we were running the DN algorithm, at some moment the algorithm would label them in the same way. Also, since we process diagonals in the order of their global-keys, any diagonal that has global-key smaller than that of the root of H must have been processed and labeled correctly. Hence, the first algorithm invariant follows. For the second invariant, whenever a diagonal has its distance value set to non-infinity for the first time, it is always inserted into H. For the third invariant, as discussed in the algorithm description, a corridor $\mathcal{C}$ is processed only if a door d of $\mathcal{C}$ is processed and beams of $B(d)$ are towards $\mathcal{C}$. Although d may be processed in the algorithm twice, it is processed only once in either the right-sweep or the left-sweep. Hence, with beams towards the inside of $\mathcal{C}$, d is only processed once in the entire algorithm, implying that $\mathcal{C}$ is processed only once in the entire algorithm in the direction from d to the other door of $\mathcal{C}$.

$B(d^*) = \emptyset$ In the sequel, we discuss the case where $B(d^*) = \emptyset$. The algorithm is simpler in this case. First, we find the set S of diagonals in H whose distances are $2i + 1$, in $O(|S| \log |H|)$ time by keeping doing the extract-min operations (i.e., all diagonals of S are removed from H). We also let S contain d^*. According to our way of comparing keys and global-keys, all diagonals of S have empty beam sets. If we were running the DN algorithm, the diagonals of S would be locally-outmost and we would be about to start the $(i + 1)$-th phase (not the i-th phase). As in the previous case, we run the two sweep procedures starting from the diagonals of S. Let S_R (resp., S_L) be the subset of diagonals of S whose beam directions are rightward (resp., leftward). We build a min-heap H_R (resp., H_L) on the diagonals of S_R (resp., S_L). Below, we only discuss the right-sweep since the left-sweep is similar.

Since we are doing the right-sweep in the $(i+1)$-th phase, each diagonal of S_R will generate a beam from the entire diagonal, and all new diagonals illuminated in the right-sweep will get distance $2i + 3$ instead of $2i + 1$. From now on, we associate each diagonal of S_R with the beam, i.e., for each $d \in S_R$, $B_r(d)$ now consists of the only beam on d although it was empty in the i-th phase. Since each diagonal of S_R originally got an empty beam set (at the end of the i-th phase), by Observation 1, the beam of $B_r(d)$ cannot be towards a junction cell and thus it must be towards the inside of a corridor.

As long as H_R is not empty, we repeatedly do the following.

We obtain the leftmost diagonal d of H_R (which is at the root) and remove it from H_R. Let $\mathcal{C}$ denote the corridor that the beam of $B_r(d)$ enters. We process the corridor $\mathcal{C}$ using the beams of $B_r(d)$. Again, one may assume we still use the DN algorithm to process $\mathcal{C}$, and later we will replace it by our corridor-processing algorithm in Section C.

Let δ be the distance on the other door d' obtained by the above processing and let B' be the corresponding beam set. Let $dis(d')$ and $B(d')$ be the original distance value and beam set at d'. Again, by the third algorithm invariant, this is the the first time $\mathcal{C}$ is processed in the direction from d to d'; hence, if $dis(d') \neq \infty$, then d' must be labeled by a sweep from outside $\mathcal{C}$ and the beams of $B(d')$ must enter $\mathcal{C}$. Depending on the value of $dis(d')$, we may need to update the label of d' in several cases.

1. If $dis(d') = \infty$, we set $dis(d') = \delta$ and $B(d') = B'$. Note that $\delta \geq 2i + 3$. Hence, the global-key of d' is strictly larger than that of d^*, which has distance value $2i + 1$. We insert d' into H (not H_R).
2. If $dis(d') \leq 2i + 1$, then since $B(d^*) = \emptyset$, the global-key of d' is no larger than that of d^* regardless of whether $B(d')$ is empty or not. By the first algorithm invariant, d' has been correctly labeled. We do a "post-processing" to compute the v-v-map in $\mathcal{C}$ by using the beams of $B_r(d)$ and $B(d')$. Again, we will give a *corridor-post-processing* algorithm for this step later in Section C.
3. The remaining case is when $dis(d') \neq \infty$ and $dis(d') > 2i + 1$. By the same argument as in the previous case where $B(d^*) \neq \emptyset$, this case cannot happen.

The above describes the right-sweep procedure. The left-sweep is similar.

This finishes our discussion on the case in which $B(d^*)$ is empty for the diagonal d^* at the root of H. As in the previous case, all algorithm invariants hold.

The algorithm finishes if all three heaps H, H_L, and H_R become empty. After that, for each diagonal d in a junction cell, $dis(d)$ and $B(d)$ have been correctly computed. During the algorithm some corridors have been labeled correctly while others are left for post-processing.

Specifically, consider any corridor $\mathcal{C}$ and let d_1 and d_2 be its two doors with their beam sets $B(d_1)$ and $B(d_2)$. If $\mathcal{C}$ is not left for a post-processing, then $\mathcal{C}$ has been processed either from d_1 to d_2 or from d_2 to d_1 and the v-v-map in $\mathcal{C}$ has been computed after the processing. Suppose the above processing is from d_1 and d_2. Then, $\mathcal{C}$ is processed using the beams of $B(d_1)$ and $B(d_2)$ is obtained after the processing. We will show in Section C that our corridor-processing algorithm on $\mathcal{C}$ runs in $O(m + (h_1 - h_2 + 1) \log h_1)$ time, where m is the number of vertices of $\mathcal{C}$, $h_1 = |B(d_1)|$, and $h_2 = |B(d_2)|$. If $\mathcal{C}$ is left for a post-processing, i.e., to compute the v-v-map in $\mathcal{C}$ by using $B(d_1)$ and $B(d_2)$, we will show in Section C that our corridor-post-processing algorithm runs in $O(m + h_1 \log h_1 + h_2 \log h_2)$ time.

Lemma 1 *Given $VD(\mathcal{F})$, our algorithm computes the v-v-map on $VD(\mathcal{F})$ in $O(n + h \log h)$ time.*

Proof. Recall that the beams in our algorithm are generated by locally-outmost diagonals. We say that two beams are *different* if they do not have the same generator. We say a diagonal is *generated* by a corridor if the generator of the diagonal is in the corridor.

We first prove a *claim* that the number of different beams at the diagonals in all junction cells is $O(h)$. To see this, a key observation is that since a corridor is a simple rectilinear polygon, it can only generate at most two new beams that can go out of the corridor in the entire algorithm. Specifically, consider a corridor $\mathcal{C}$ with doors d_1 and d_2. Suppose the algorithm processes $\mathcal{C}$ in the direction from d_1 to d_2. Then, as will be seen later in Section C, if some beams of $B(d_1)$ can directly illuminate d_2, then all beams of $B(d_2)$ are from $B(d_1)$ (although some beams may become narrowed) and there is no new beam generated by $\mathcal{C}$; otherwise all beams of $B(d_1)$ terminate inside $\mathcal{C}$ and $B(d_2)$ will only have one beam coming out of the corridor through d_2 (this is because $\mathcal{C}$ is a simple rectilinear polygon). Since any corridor can be processed at most twice, it can generate at most two new beams. Since there are $O(h)$ corridors, the total number of beams on the diagonals of junction cells is $O(h)$.

To obtain the running time of the algorithm, we analyze the time we spent on junction cells and the corridors separately. Due to the above claim, the size of $B(d)$ for each diagonal d in any junction cell is $O(h)$. Since there are $O(h)$ diagonals in all junction cells, the total time we spent on processing them is $O(h \log h)$.

For the time we spent on all corridors, it is the sum of the time of our corridor-processing algorithm and corridor-post-processing algorithm on all corridors.

Let S denote the set of the diagonals in all junction cells. Define $\mathcal{C}$, d_1, d_2, $B(d_1)$, $B(d_2)$, m, h_1, and h_2 the same as before. Suppose $\mathcal{C}$ has been post-processed. Then the corridor-post-processing algorithm on $\mathcal{C}$ takes $O(m+h_1 \log h_1+ h_2 \log h_2)$ time. The sum of the term m overall all corridors is $O(n)$. All beams of $B(d_1)$ and $B(d_2)$ are terminated inside $\mathcal{C}$ after the post-processing. Since the number of beams that are terminated inside corridors is no more than the number of different beams at the diagonals in all junction cells, by the above claim, the number of beams on the diagonals of S that are terminated inside corridors is $O(h)$. Hence, the sum of h_1 (resp., h_2) over all corridors that have been post-processed is $O(h)$, and the sum of $h_1 \log h_1 + h_2 \log h_2$ over all corridors that have been post-processed is $O(h \log h)$. Note that each corridor can be post-processed at most once. This proves that the total time of the corridor-post-processing algorithm in the entire algorithm is $O(n + h \log h)$.

We can use the similar approach to analyze the total time of the corridor-processing algorithm. Suppose $\mathcal{C}$ has been processed by the corridor-processing algorithm in the direction from d_1 to d_2. Then, the running time of the algorithm on $\mathcal{C}$ is $O(m+(h_1-h_2+1) \log h_1)$ time. Similarly, the sum of m over all corridors is $O(n)$. After the processing of $\mathcal{C}$, the number of beams of $B(d_1)$ that have been terminated in $\mathcal{C}$ is at least h_1-h_2. Since the the number of beams on the diagonals

of S that are terminated inside corridors is $O(h)$, the sum of the term $h_1 - h_2$ over all corridors that have been processed is $O(h)$. The sum of the additional term 1 over all corridors is clearly $O(h)$. Therefore, although a corridor may be processed twice, the total time of the corridor-processing algorithm in the entire algorithm is $O(n + h \log h)$.

The lemma thus follows. □

The above computes the v-v-map on $VD(\mathcal{F})$. Again, the above algorithm only labels diagonals. Using the similar approaches as discussed in Section A, we can also label cells and maintain path information within the same running time asymptotically. We omit the details.

Other three maps can be computed similarly. For computing the h-v-map on $VD(\mathcal{F})$, one difference is on the initial steps, as follows. Initially, we let s generate two beams that are two horizontal rays towards right and left, repectively. We set the distance value of d_s to 2, where d_s is the vertical diagonal through s. Then, we consider d_s as two duplicate diagonals associated with the above two beams respectively, and insert the two duplicate diagonals into the heap H. The remaining algorithm is the same as before except that we replace the distance values $2i + 1$ and $2i + 3$ in the algorithm description by $2i + 2$ and $2i + 4$, respectively. The h-h-map and v-h-map on $HD(\mathcal{F})$ can be computed in symmetric ways.

We thus obtain the main result of this paper.

Theorem 1. *Given a set of h pairwise-disjoint rectilinear polygonal obstacles with a total of n vertices in the plane, after a triangulation of the free space is computed in $O(\min\{n \log n, n + h \log^{1+\epsilon} h\})$ time for any $\epsilon > 0$, we can construct a link distance map with respect to any given source point s in $O(n + h \log h)$ time, such that for any query point t, the link distance of a min-link rectilinear s-t path π can be computed in $O(\log n)$ time and π can be output in additional time linear in its link distance.*

C The Algorithms for Processing Corridors

In this section, we present our corridor-processing algorithm and corridor-post-processing algorithm. In fact, as shown later, the second algorithm is simply to run the first one twice from the two doors. Below, we discuss the corridor-processing algorithm first.

Let $\mathcal{C}$ be any corridor with two doors d_1 and d_2 and their beam sets $B(d_1)$ and $B(d_2)$. Let m be the number of vertices of $\mathcal{C}$, $h_1 = |B(d_1)|$, and $h_2 = |B(d_2)|$. Suppose we want to process $\mathcal{C}$ to compute the labels of all diagonals, by using the beams in $B(d_1)$ and the distance value $dis(d_1)$ of d_1. In particular, we want to obtain the beam set $B(d_2)$ and the distance value $dis(d_2)$ for d_2. Our algorithm is conceptually similar to the DN algorithm if we apply it on $\mathcal{C}$, but our approach is more efficient due to a better implementation by making use of the simplicity of the corridor (one crucial property is that there will be no merge operations on the beam sets).

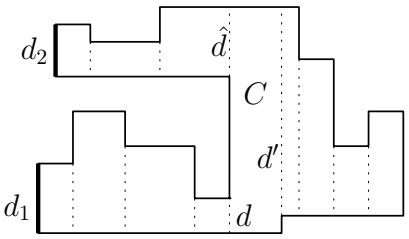

Fig. 8. Illustrating $VD(\mathcal{C})$: the two doors are shown with thick segments.

Denote by $VD(\mathcal{C})$ the vertical decomposition of $\mathcal{C}$ (e.g., see Fig. 8). Note that although d_1 and d_2 are now on the boundary of $\mathcal{C}$, we still consider them as diagonals. Let $C(d_1)$ denote the cell of $VD(\mathcal{C})$ that contains d_1. Starting from $C(d_1)$, we propagate the beams from $B(d_1)$ to all other cells one by one until all diagonals have been labeled.

Consider any diagonal d of $VD(\mathcal{C})$ such that d is not d_1 or d_2. Since $\mathcal{C}$ is simply connected, d divides $\mathcal{C}$ into two polygons, and we use $\mathcal{C}(d)$ to refer to the one that does not contain d_1. Clearly, for any point $t \in \mathcal{C}(d)$, any path from d_1 to t must intersect d. Consider any cell C of $VD(\mathcal{C})$. The cell C may have diagonals on both its two vertical sides. It is not difficult to see that C has one and only one diagonal d such that C is in $\mathcal{C}(d)$, and we call that diagonal the *entrance diagonal* of C. Other diagonals of C are called *exit diagonals* of C. For example, in Fig. 8, d is the entrance diagonal of C, and $\hat{d}$ and d' are exit diagonals. Note that every diagonal is an entrance diagonal of one and only one cell. In particular, we consider d_1 as the entrance diagonal of $C(d_1)$.

A general step of our algorithm works as follows. Consider a cell C and suppose the entrance diagonal d of C has been labeled, i.e., $dis(d)$ and $B(d)$ are available (and the beams of $B(d)$ are stored in a balanced binary search tree $T(d)$). Initially, C is the cell $C(d_1)$ and d is d_1. Our goal for processing d is to label all exit diagonals in C.

Denote by e_l and e_r the left and right sides of C, respectively. Without loss of generality, we assume d is on e_l (e.g., see Fig. 8). Thus, the beams of $B(d)$ are rightward. The distance values of the exit diagonals are easy to compute. If $B(d)$ is empty, then we need to generate a single beam from the entire d and every exit diagonal of C obtains the distance value $dis(d) + 2$; otherwise, every exit diagonal of C obtains the distance value $dis(d)$. Below, we focus on computing the beam sets of the exit diagonals.

Suppose e_l has another diagonal $\hat{d}$. Then, regardless of whether $B(d)$ is empty, we set $B(d) = \emptyset$ (and $T(d) = \emptyset$), which can be done in only constant time.

Now consider any diagonal d' on e_r. First, we want to compute $B(d) \cap d'$, i.e., the portions of the beams of $B(d)$ that can illuminate d', and set $B(d') = B(d) \cap d'$. If $B(d) = \emptyset$, then since we generate a single beam from d, $B(d') = B(d) \cap d'$ has at most one beam and can be computed in constant time.

In the sequel we consider the case where $B(d) \neq \emptyset$. Our main effort is on handling this case. Recall that the beams of $B(d)$ are stored in a balanced bi-

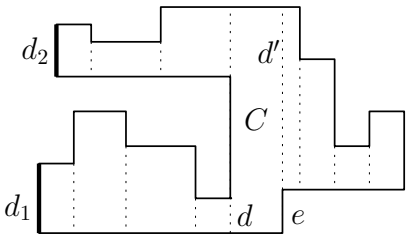

Fig. 9. The right side e_r of C consists of d' and e from top to bottom.

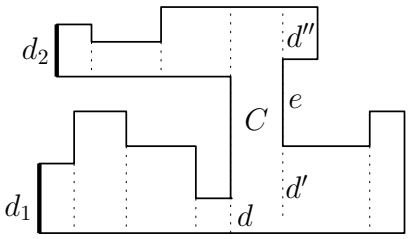

Fig. 10. The right side e_r of C consists of d'', e, and d' from top to bottom.

nary search tree $T(d)$. We add two special pointers to $T(d)$ that point to the lowest beam and the highest beam of $B(d)$ respectively so that we can access these beams in constant time. Note that with these special pointers, we can still perform the previous operations on $T(d)$ each in logarithmic time.

Depending on whether e_r has one or two diagonals, there are two cases.

If e_r has only one diagonal d', let e be the obstacle edge on e_r such that d' is the vertical extension of e (e.g., see Fig. 9). Clearly, e_r is the union of d' and e. Without loss of generality, assume e is lower than d'.

Suppose b is a beam in $B(d)$ and b' is the rightward projection of b on e_r. For any line segment l on e_r, we say that b *intersects* l if b' intersects l properly, and b *fully intersects* l if b' is contained in l.

Below we let b denote the lowest beam of $B(d)$, which can be obtained in constant time by using the special pointers on $T(d)$. We first check whether b intersects e. Depending on how b intersects e, there are three cases. The correctness of our setting in all these cases is based on that e_r consists of d' and e from top to bottom (e.g., see Fig. 9).

1. If b does not intersect e, then every beam of $B(d)$ fully intersects d'. Thus we have $B(d') = B(d)$ and $T(d') = T(d)$. This can be done in constant time.
2. If b intersects e but does not fully intersects e, then we set $B(d') = B(d)$ and $T(d') = T(d)$, but we also change b's length to that of its portion intersecting d'. This can be done in constant time.
3. If b fully intersects e (so b "terminates" at e), then depending on whether $B(d)$ has only one beam, there are further two subcases.
   If $B(d)$ has only one beam, which is b, then we simply set $B(d') = \emptyset$ and $T(d') = \emptyset$ since b terminates at e.
   Otherwise, we have $|B(d)| \geq 2$. In this subcase, as before we use a split operation on $T(d)$ to obtain $T(d')$ and $B(d') = B(d) \cap d'$ in $O(\log |B(d)|)$ time. Note that since b terminates at e, b is not in $B(d')$, and thus $|B(d')| \leq |B(d)| - 1$. This will be useful to our time analysis on the split operations.

If e_r has another diagonal d'', without loss of generality, we assume d' is the lower one. Let e be the obstacle edge on e_r. Then e_r consists of d'', e, and d' from top to bottom (e.g., see Fig. 10).

Again, let b denote the lowest beam of $B(d)$. Depending on how b intersects d', there are three cases. In every case below, we will also obtain a tree T' of beams that will be used later to compute the beams of d''.

1. If b does not intersect d', then we have $B(d') = \emptyset$ and $T(d) = \emptyset$. We set $T' = T(d)$.
2. If b intersects d' but does not fully intersects d', then $B(d')$ consists of a single beam that is the portion of b intersecting d'. It is straightforward to construct $T(d')$. We set $T' = T(d)$ but also change the length of b to that of the portion of b not intersecting d'. All above can be done in constant time.
3. If b fully intersects d', then we further check whether the *highest* beam b^* of $B(d)$ intersects d'.
   If b^* fully intersects d', then we have $B(d') = B(d)$ and $T(d') = T(d)$. Also, $T' = \emptyset$.
   If b^* intersects d' but does not fully intersect d', then we set $B(d') = B(d)$ and $T(d') = T(d)$ but also change b^*'s length to that of its portion intersecting d'. Also, we let T' only include the portion of b^* not intersecting d'. All this can be done in constant time.
   If b^* does not intersect d', then as before we use a split operation that split $T(d)$ into two trees: $T(d')$, which consists of the beams of $B(d') = B(d) \cap d'$, and T', which consists of the rest of the beams of $B(d)$. This can be done in $O(\log |B(d)|)$ time. Let B' be the set of beams in T'. Note that since b fully intersects d', b is not in B', and since b^* does not intersect d', b^* is not in $B(d')$. Hence, we have $|B'| \leq |B(d)| - 1$ and $|B(d')| \leq |B(d)| - 1$, which will be useful to our time analysis on the split operations. Further, since b fully intersects d' but b^* does not intersect d', we have $b' \neq b^*$ in this case, implying that $|B(d)| \geq 2$.

Next, we compute $B(d'')$ and $T(d'')$ from the beams in the tree T', in the same way as the previous case where e_r only contains one diagonal. Namely, we first check how the lowest beam of T' intersects e and then proceed accordingly for the three cases.

The above labels all exit diagonals of the cell C. Based on our above algorithm, an easy observation is that if $|B(d)| = 1$, then $|B(d')| \leq 1$ for each exit diagonal d' of C.

Next, from each exit diagonal of C, we proceed with the same procedure. The algorithm is done once all diagonals have been labeled. The following lemma analyzes the running time of the algorithm.

Lemma 3. *Our corridor-processing algorithm on $\mathcal{C}$ runs in $O(m + (h_1 - h_2 + 1) \log h_1)$ time.*

Proof. We make a few observations on our algorithm. First, for any diagonal d, if computing $B(d)$ does not need a split operation, then it takes only constant time to do so. Second, for any diagonal d of $\mathcal{C}$, $|B(d)| \leq |B(d_1)| = h_1$ always hold, and if $|B(d)| \geq 2$, then the beams of $B(d)$ are from $B(d_1)$. Third, for any

diagonal d, if $B(d)$ is obtained by a split operation on a beam set B, then $|B| \geq 2$ and $|B(d)| \leq |B| - 1$.

The first two observations imply that if there are k split operations in the entire algorithm, then the total running time is $O(m + k \log h_1)$. We claim that $k = O(h_1 - h_2 + 1)$.

Indeed, by the second and the third observation, if a split operation is performed on $B(d)$ for some diagonal d, then all beams of $B(d)$ must be originally from $B(d_1)$, and further, the split operation splits $B(d_1)$ into two sets such that the number of beams in each set is at most $|B(d)| - 1$. Therefore, it can be verified that if $h_2 \geq 2$, then $k = O(h_1 - h_2)$, and otherwise, $k = O(h_1)$. In either case, $k = O(h_1 - h_2 + 1)$.

We conclude that the algorithm runs in $O(m + (h_1 - h_2 + 1) \log h_1)$ time. □

Next, we present our corridor-post-processing algorithm. Again, consider the corridor $\mathcal{C}$ as above, but now $B(d_2)$ and $dis(d_2)$ are also given as input and the beams of $B(d_2)$ are towards the inside of $\mathcal{C}$. Our goal is to compute the distance values for all diagonals of $\mathcal{C}$ by using the beams of $B(d_1)$ and $B(d_2)$. An easy solution is to use our above corridor-processing algorithm to process $\mathcal{C}$ twice, once only using $dis(d_1)$ and $B(d_1)$ and once only using $dis(d_2)$ and $B(d_2)$. Then, each diagonal d has been labeled twice, and we finally set $dis(d)$ to the smaller distance value labeled above. Clearly, the total running time is bounded by $O(m + h_1 \log h_1 + h_2 \log h_2)$.