Faster algorithms for minimum-link paths with restricted orientations

Valentin Polishchuk and Mikko Sysikaski

Helsinki Institute for Information Technology
Department of Computer Science, University of Helsinki
firstname.lastname@cs.helsinki.fi

Abstract. We give an $O(n^2 \log^2 n)$-time algorithm for computing a minimum-link rectilinear path in an n-vertex rectilinear domain in three dimensions; the fastest previously known algorithm of [Wagner, Drysdale and Stein, 2009] has running time $O(n^{2.5} \log n)$. We also present an algorithm to find a minimum-link C-oriented path in a C-oriented domain in the plane; our algorithm is simpler and more time-space efficient than the earlier one of [Adegeest, Overmars and Snoeyink, 1994].

1 Introduction

Computing minimum-link rectilinear paths amidst rectilinear obstacles in the plane is one of the oldest and best studied problems in computational geometry [2, 3, 12–16, 19, 22, 25–27]. For a domain with n vertices, an optimal, $O(n \log n)$-time linear-space algorithm was presented at WADS 1991 by Das and Narasimhan [2]. A similar algorithm was claimed in the lesser known work of Sato, Sakanaka and Ohtsuki [22].

1.1 3D rectilinear paths

In three dimensions, the problem is somewhat farther from being solved. De Berg et al. [4] give an $O(n^d \log n)$-time algorithm to find shortest path in combined metric (a linear combination of length and number of links) amidst axis-parallel boxes in $\mathbb{R}^d$. An $O(n^3)$-time algorithm for minimum-link rectilinear path in a 3D rectilinear domain is implied by results of Mikami and Tabuchi [17]. Fitch, Butler and Rus [6] presented an algorithm for the 3D case with good practical performance; in many cases, their algorithm runs in $O(n^2 \log n)$ time. Still, the worst-case running time of the algorithm in [6] is cubic.

The fastest current solution for the minimum-link rectilinear path problem in $\mathbb{R}^3$ is due to Wagner, Drysdale and Stein [23]. Their algorithm, based on searching binary space partition (BSP) of the domain, runs in $O(n^{2.5} \log n)$ time and $O(n^{2.5})$ space.

1.2 C-oriented paths in the plane

Another generalization of the 2D rectilinear setting is the *C-oriented* version [1,8–10,18,21,24] in which orientations of path edges and the domain sides come from a fixed set C of directions. (Abusing the notation, we use C to denote also the cardinality of the set C.) Minimum-link C-oriented paths in the plane were studied by Adegeest, Overmars and Snoyeink [1]. Two algorithms are presented in [1]: one running in $O(C^2 n \log n)$ time and space, the other—in $O(C^2 n \log^2 n)$ time and $O(C^2 n)$ space.

1.3 Contributions

We revisit the two generalizations of the 2D minimum-link rectilinear path problem, giving more efficient algorithms for both versions. Specifically,

- In Section 2 we give an $O(n^2 \log^2 n)$-time $O(n^2)$-space algorithm for the 3D minimum-link rectilinear path problem.
- In Section 3 we present an $O(C^2 n \log n)$-time $O(Cn)$-space algorithm to find a minimum-link C-oriented path in a C-oriented domain in the plane.

Similarly to the earlier works, our algorithms actually build shortest path *maps* – data structures to answer efficiently link distance queries to a fixed source (in the case of rectilinear paths in 3D, constructing the map with our algorithm requires $O(n^2 \log^2 n)$ space – slightly more than just finding one shortest path).

2 Rectilinear paths amidst rectilinear obstacles in $\mathbb{R}^3$

Let P be an n-vertex rectilinear domain in 3D, and let $s, t \in P$ be given points; the goal is to find a minimum-link rectilinear s-t path in P. We follow the "staged illumination" paradigm [7, Sections 26.4, 27.3], dominant in the minimum-link literature: on first step illuminate (and label with link distance 1) the set reachable with a single link from s, on second step illuminate and label with 2 what is reachable with two links, and so on, until t is lit.

At any step k, the illumination is done by sweeping a plane in each of the six directions $\pm x, \pm y, \pm z$. We will describe the sweep of a horizontal plane in $+z$ (vertical) direction; the other sweeps are analogous. The goal of the sweep is two-fold: (1) to discover (and label with k) the volume illuminated when shining light upwards from the volume lit at step $k-1$, and (2) to generate events for (all six) sweeps on step $k+1$.

As in [23], we assume that obstacle faces are decomposed into rectangles; we call them *obstacle rectangles*. The sweep is guided by a decomposition of the free space into axis-aligned boxes (cells), with each cell storing links to the neighbor boxes and the obstacle rectangles touching the box.

The status of the sweep plane is the subset of the plane that gets illuminated if light is shone upwards from the points that were illuminated at step $k-1$; the status is maintained as a set of rectangles in a 2D segment tree. With each

rectangle R we store a variable z_R which is the smallest z-coordinate of the origin of an upward-shining light ray that hits R; in other words, if one moves down from a point $r \in R$, then z_R is the height below which there is no point that was illuminated at step $k-1$ and can see r.

The event queue contains events of three types:

- CellEvent occurs when the sweep plane reaches the upper boundary of a cell. Processing the event involves the following operations: (1) each upper neighbor of the cell is inserted, as a CellEvent, into the event queue; (2) each obstacle rectangle (if any) touching the cell from above is inserted, as an ObstacleEvent, into the event queue; (3) the cell is inserted as a CellEvent into the event queues for the sweeps in the other directions at the next step, $k+1$; (4) if the cell is not labeled (i.e., has not been illuminated before), the cell is labeled with k.
- ObstacleEvent occurs when the sweep plane reaches an obstacle rectangle R_o; let z_o be the time (height) of the event. For each rectangle R from the status, intersected by R_o, let $[x_R^{\min}, x_R^{\max}] \times [y_R^{\min}, y_R^{\max}]$ be the intersection $R \cap R_o$. We generate an AddRectangleEvent into the event queue for the $+x$ sweep at step $k+1$: the generated rectangle is $[y_R^{\min}, y_R^{\max}] \times [z_R, z_o]$ and the time of the event is $x_R^{\max}$. Analogous rectangles are inserted to other sweeps. R_o is removed from the sweep plane status.
- AddRectangleEvent, on which a rectangle is added to the segment tree.

This sweep-label-generate strategy is a common one. Clearly, the running time of the algorithm employing the strategy, depends heavily on the number of cells in the decomposition (all cells must be labeled, after all): using the $O(n^{1.5})$-size BSP [5,11,20] instead of the $O(n^3)$ "full grid" allowed [23] to bring the time complexity from cubic down to $O(n^{2.5} \log n)$. Following this direction, one might improve the running time by using a yet smaller-size decomposition; we, however were not able to do this. Instead, we resort to a seemingly worse, *quadratic*-size decomposition. We compensate the increase in the number of cells by a better (albeit still quadratic) bound on the number of neighbor relations between the cells; the number enters the algorithm's running time via the need to perform Operation (1) in CellEvents. While [23] spend $O(n^{2.5})$ time determining the neighbor relations, we get them for free. *Any* quadratic-size decomposition with quadratic number of cell-to-cell pointers works for us; one simple decomposition is obtained by sweeping a horizontal plane in $+z$ direction stopping at every horizontal obstacle rectangle, decomposing the cross-section of P with the stopped plane into rectangles by extending maximal free horizontal segments through vertices of the cross-section, and pulling the rectangles up in $+z$ direction until the height of the next stop of the sweep plane.

The above description is very generic, and many technical details have to be filled in, both by [23] and by us (below). We remark that also in these details, our algorithm is different from [23] (our analysis is different from [23] as well). This is not surprising: shooting for a smaller running time while working with

a larger-complexity decomposition is challenging, and prevents directly reusing ideas from [23].[1]

In particular, a standard way to avoid reilluminating the same cell at too many steps is to declare labeled cells as obstacles. We cannot afford this because updating the sweep plane status at one ObstacleEvent may take as much as $O(n \log n)$ time — and we might declare all our $O(n^2)$ cells as obstacles. (Note that handling ClearRect events—part of the sweep plane status updates—is the bottleneck in the $O(n^{2.5} \log n)$ algorithm of [23].) So we approach limiting the number of status updates from the other end: instead of trying to decrease directly the number of rectangles *deleted* from the status, we restrict the number of rectangles that are ever *added*. See Section 2.1 for the details.

To bound the number of AddRectangleEvents processed by the algorithm, we would like to have "nice" rectangles; unfortunately, the rectangles generated as described above are somewhat arbitrary. One nice set of rectangles is as follows: take the cross-section of P by the horizontal plane supporting an obstacle rectangle, and decompose the cross-section (which is a 2D rectilinear domain) into $O(n)$ rectangles by extending maximal free-space segments parallel to x trough vertices of the domain (this is the standard trapezoidal decomposition of a 2D rectilinear domain; we use it e.g., in Section 3). To enforce that the AddRectangleEvents deal only with such nice rectangles, at every ObstacleEvent we filter out the added rectangles locally, keeping for each sweep direction only those rectangles that are not "dominated" by larger rectangles going in the same direction and are not "annihilated" by rectangles going in the opposite direction. Section 2.2 presents the details.

2.1 Avoiding reillumination

We only describe how we add xz- and yz-rectangles, generated during the $+z$ sweep; the other sweeps are identical. Call such rectangles *displays*.

The purpose of displays is to emit light into *dark* volumes; thus it makes sense to keep only those displays that have a potential to shine into previously unilluminated space. Any cell σ with label $k-3$ or less gets *fully* illuminated by step $k-1$. Hence after step k, everything visible from σ will be illuminated. Therefore any display R' that touches only cells with labels $k-3$ or less can be safely discarded (i.e., not added as an AddRectangleEvent): nothing new gets illuminated by shining light from R'.

To determine whether a given display R' intersects any "new" cell (cell with label $k-2, k-1$ or k) we maintain an auxiliary 2D segment tree storing projection of the new cells onto the xy plane; the tree is updated on every CellEvent and ObstacleEvent, and also cleared of old cells when k increases. (Note that directly testing R' for intersection with every new cell would be too inefficient because

[1] For completeness, let us mention that we slightly differ from [23] already in the generic description of the events: [23] inserts AddRectangleEvents into the other sweeps when processing CellEvents; because we have too many cells, we cannot do it, and instead insert AddRectangleEvents when processing ObstacleEvents.

R' may in principle intersect $\Omega(n)$ cells). The query for R' reduces then to the simple test of whether the projection of R' on the horizontal plane intersects rectangles stored in the auxiliary tree.

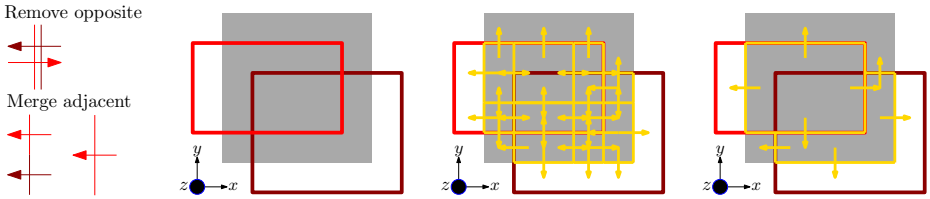
2.2 Filtering

Again, we only describe what we do during the $+z$ sweep when generating displays (which are the potential AddRectangleEvents for the sweeps in $\pm x$ and $\pm y$ directions). Recall that on an ObstacleEvent we remove an obstacle rectangle R_o from the sweep plane status 2D segment tree, i.e., we clear all subtrees rooted at canonical nodes of R_o; to clear a subtree, we visit all canonical nodes of the rectangles in the subtree, and at every canonical node R generate 4 displays as specified in the description of the ObstacleEvent.

For our analysis of the number of AddRectangleEvents (Lemma 3) it will be important that each rectangle in the decomposition of a cross-section of P by maximal free-space horizontal segments is charged to exactly one display. To enforce this, we do the following operations after generating the displays, before moving to the next event in the $+z$ sweep:

- Find all pairs of displays with the same positions but opposite directions (Fig. 1(a), top). If the z-ranges of the displays are the same, remove both of them. Otherwise remove the one with smaller z-range, i.e., the one with the higher z_R.
- Merge aligned displays with the same z-range into bigger displays (Fig. 1(a), bottom).

Figure 1 illustrates the events that are generated initially and the remaining events after the filtering.



(a) Elementary filtering operations

(b) Original rectangles of the segment tree (red) and the obstacle face to be removed (grey)

(c) AddRectangleEvents generated for each canonical node to be cleared

(d) AddRectangleEvents remaining after filtering

Fig. 1. Filtering the displays during an ObstacleEvent. We generate displays in all 4 directions for every canonical node cleared from the segment tree (Fig. 1(c)). After the filtering, the displays are maximal in x- and y-directions (Fig. 1(d)).

2.3 Analysis

Let σ be a cell of the decomposition, and let R_o be an obstacle rectangle.

Lemma 1. CellEvent*s containing* σ *are generated on at most* 9 *steps of the illumination.*

Proof. Assume that σ is first discovered on step k. Then no later than on step $k+2$ any point inside σ is illuminated, and on step $k+3$ we discover the set S of all points visible from σ. After step $k+6$ we have discovered every point that might share a common AddRectangleEvent with a point in S so after step $k+8$ we can no longer generate AddRectangleEvents that would reach σ. □

Lemma 2. *At most* 9 ObstacleEvent*s containing* R_o *are generated over the course of the illumination.*

Proof. Similarly to Lemma 1, after discovering R_o on step k we discover every point that can see R_o no later than on step $k+3$, so again no sweep can reach R_o after step $k+8$. □

Lemma 3. *The number of* AddRectangleEvent*s is* $O(n^2)$.

Proof. The boundaries of illuminated area are always aligned with obstacle faces. Hence any AddRectangleEvent belongs to the supporting plane of some obstacle face. We will consider only the AddRectangleEvents processed in the $+z$ sweep; the other directions are identical.

Let R_o be some horizontal obstacle rectangle, and let H_o be the supporting plane of R_o. We will bound the number of AddRectangleEvents happening in H_o. We assume that the plane H_o is unique to R_o. That is, even if another obstacle rectangle R'_o has the supporting plane $H'_o = H_o$ (because R_o and R'_o belong to the same obstacle face), we will treat the AddRectangleEvents for H_o and H'_o separately. This way we may only overcount the number of AddRectangleEvents.

Let P_o be the cross-section of P by H_o. Let D_x be the trapezoidal (in fact, rectangular) decomposition of P_o obtained by extending maximal free-space segments parallel to x through vertices of P_o; define D_y similarly. Consider finding a minimum-link path in P_o from some arbitrary set of maximal (either in x or in y or both) rectangles lying in P_o. The staged illumination from the rectangles proceeds in P_o by lighting up rectangles from $D_x \cup D_y$ (this is the classical process central to the study of 2D minimum-link paths).

Now look at our staged illumination of the 3D domain P, and consider the "cross-section of the illumination" by H_o; i.e., look at when and how the light from the 3D illumination intersects P_o. The crucial observation is the following: after the 3D illumination has reached H_o, the cross-section of the illumination is exactly the 2D staged illumination in P_o. In particular, at most 2 AddRectangleEvents will happen in each rectangle from $D_x \cup D_y$.

Indeed, the 3D illumination arriving at H_o can be viewed as "piercing" P_o with rectangular "pillars of light"; the cross-section of the pillars by H_o is a set of rectangles – not necessarily maximal (yet). However, already on the next step,

when the light is shone in x- and y-directions from the rectangles, the lit area of P_o *is* a set of maximal rectangles, and the illumination from the set proceeds along $D_x \cup D_y$.

Thus, the number of AddRectangleEvents happening at H_o is the sum of 2 parts: (1) the total number of rectangles added immediately after the piercing, and (2) the complexity of D_x, D_y. The events in (1) are defined by those rectangles from the xz- and yz- sweep planes segment trees, that are intersected by H_o: the number of the events is the number of the rectangles in the trees that are intersected by H_o. But the number of rectangles in a segment tree intersected by any line is $O(n)$; thus the number of type-(1) events is $O(n)$. The complexity of D_x, D_y is also linear; hence the total number of AddRectangleEvents on H_o is linear.

To finish the proof, recall that H_o is the supporting plane of an obstacle rectangle; the number of such planes is $O(n)$. □

Theorem 1. *Minimum-link path can be computed in time* $O(n^2 \log^2 n)$ *using* $O(n^2)$ *space.*

Proof. The time in the algorithm is spent in creating the decomposition, maintaining event queues and handling events, of which the event handling is the dominating component. Each CellEvent and AddRectangleEvent is handled in $O(\log^2 n)$ time using standard segment tree operations, and the number of such events is $O(n^2)$ by Lemmas 1 and 3. Clearing a rectangle in an ObstacleEvent can be performed in $O(n \log n + \rho)$ time, where ρ is the number of the tree nodes to remove (and to generate AddRectangleEvents for); the $O(\rho)$-part can be charged to the earlier events that created the nodes. Together with Lemma 2 this gives an $O(n^2 \log n)$ time bound for handling the ObstacleEvents. Thus in total handling the events takes $O(n^2 \log^2 n)$ time.

The size of the decomposition as well as the number of events and the size of a segment tree are all $O(n^2)$ so the space used is $O(n^2)$. □

To build the shortest path map we use a persistent version of the segment tree and store with each cell the snapshots of the sweepline status from the times when a sweep reached the cell.

3 *C*-oriented paths in the plane

In this section P is a C-oriented n-vertex polygonal domain in the plane, and we want to build the C-oriented link distance map from a given point $s \in P$.

3.1 Overview

As in [1], we build C trapezoidations of P; the trapezoids in the trapezoidation $c \in C$ are obtained by extending maximal free c-oriented segments through vertices of P. We label the trapezoids with link distance from s; the labeling proceeds in n *steps*, with label-k trapezoids receiving their label at step k. Any

label-k trapezoid must be intersected by a label-$(k-1)$ trapezoid of a different orientation; hence, step k boils down to detecting all unlabeled trapezoids intersected by label-$(k-1)$ trapezoids.

Some trapezoids may get labeled only partially during a step k; in the final link distance map, such trapezoids are split into subtrapezoids. The partial labeling and splitting are due to the possibility that two different-orientation trapezoids do not "straddle" each other; instead they both may be "flush" with an obstacle edge whose orientation is different from the orientations of both trapezoids. Such flushness can be read off easily from lists of incident trapezoids stored with every edge of P.

After the partially labeled trapezoids are processed, we are left with discovering unlabeled trapezoids "fully straddled" by trapezoids labeled at the previous step. We clip the latter trapezoids into parallelograms, with the new sides parallel to the sweepline, and finish the step with $C(C-1)$ sweeps—one per pair of orientations.

The main difference of our algorithm from that of [1] is the separate treatment of flush and straddling trapezoids. It allows us to use only elementary data structures and improve the time-space bounds to $O(C^2 n \log n)$ and $O(Cn)$.

3.2 Definitions and notation

Any trapezoid T will have two opposite edges belonging to the boundary of P; these edges are *sides* of T. The other two edges are T's *bases*; the bases are parallel segments whose orientation belongs to C. For an orientation $c \in C$, a c-segment is a segment with orientation c. A *c-trapezoid* has c-segments as bases.

A *c-path* is a path (starting from s) whose last link is a c-segment. A point $p \in P$ is at *c-distance* k from s if p can be reached by a k-link c-path (but not faster). The c-distance equivalence decomposition of P (the *c-map*) is the partition of P into c-trapezoids such that the c-distance to any point within a cell is the same. If the c-distance to points in a c-trapezoid of the c-map is k, then the c-trapezoid has *label* k. Using the illumination analogy we also say that the trapezoid is *lit* at step k; unlit trapezoids are *dark*. We denote the set of c-trapezoids lit at step k by S_c^k.

In our algorithm, finding S_c^k is completely identical to (and independent from) finding $S_{c^*}^k$ for any $c^* \in C \setminus c$; in what follows we focus on finding S_c^k. Where it creates no confusion, we omit the subscript c and the prefix c-. E.g., "path to trapezoid in S^{k-1}" means "c-path to c-trapezoid in S_c^{k-1}", etc. We let C^* denote $C \setminus c$, and use c^* for a generic orientation from C^*. Assume w.l.o.g. that c is horizontal.

Denote by D^k the "at-most-k-links map", i.e., the trapezoidation whose trapezoids are of $k+1$ types—dark trapezoids, and trapezoids lit at steps $1 \ldots k$; in particular, D^n is the c-map. Let T' be a dark trapezoid from D^{k-1}. The crucial (albeit obvious) observation about minlink paths is that there exists a k-link path to a point $p \in T'$ if and only if there exists a $(k-1)$-link c^*-path π^* to some point $q \in T'$ that has the same y-coordinate as p. Restated in our terms, this means that a dark trapezoid $T' \in D^{k-1}$ gets fully or partially lit at step k

if and only if it is intersected by some trapezoid $T^* \in S_{c^*}^{k-1}$ lit at step $k-1$. We distinguish between two types of trapezoids intersection (Fig. 2):

Definition 1. *T', T^* are* flush *if a side of T' overlaps with a side of T^*; we say that T' is (fully or partially)* flush-lit *by T^*. T', T^** straddle *each other if both bases of T' intersect both bases of T^* (in particular, if a side of T^* overlaps with a base of T' or vice versa, then T', T^* are counted as straddling, not as flush); we say that T' is (fully)* straddle-lit *by T^*.*

Note that flushness and straddling are the only possible ways for two trapezoids from $D_c^{k-1}, D_{c^*}^{k-1}$ to intersect.

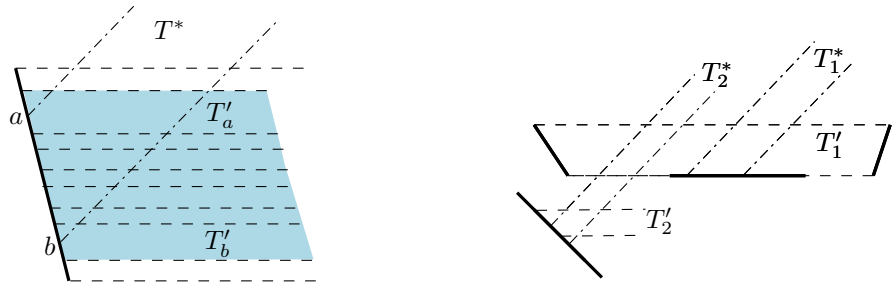


Fig. 2. Intersection types. Left: T^* is flush with the light blue trapezoids. T'_b will be split in D^k unless more of it is lit by another trapezoid. T_a will be the pot for the parallelogram cut out of T^* by the c-segment through a. Right: T_1^*, T_2^* straddle T'_1. The parallelogram cut out of T_1^* is planted into T'_1; the pot for the parallelogram cut out of T_2^* is a trapezoid T'_2 flush with T_2^*.

With Definition 1, step k of the algorithm can be completed as follows: Find dark c-trapezoids flush with trapezoids from $S_{c^*}^{k-1}$, and (fully or partially) light them. After this has been done for all $c^* \in C^*$, i.e., after all flush trapezoids are processed, any dark trapezoid will either fully remain dark in D^k or will be fully straddle-lit (i.e., there will be no more partial lighting and splitting). To straddle-light c-trapezoids, we clip each c^*-trapezoid $T^* \in S_{c^*}^{k-1}$ to a (c, c^*)-parallelogram P^*, using c-segments going through vertices of T^*. We then do a sweep in the direction perpendicular to c. The clipping and sweeping is repeated for each $c^* \in C^*$, i.e., overall, to straddle-light the c-trapezoids in S_c^k, we perform $C-1$ sweeps—one per $c^* \in C^*$.

We proceed to a detailed description of the flush- and straddle-lighting.

3.3 Flush-lighting

Sides of trapezoids belong to edges of P; we say that an edge e *supports* a trapezoid if one of its sides belongs to e. We maintain the ordered list $L_c(e)$ of c-trapezoids supported by e. The flush-lighting is done as follows: For every c^*-trapezoid $T^* \in S_{c^*}^{k-1}$ and each edge e that supports T^*, locate the vertices

a, b of T^* (lying on e) in the list $L_c(e)$. All (dark) trapezoids lying between a and b are labeled k. One of the trapezoids T'_a, T'_b containing a, b in the interior of the side is marked to be split, at the end of flush-lighting, by horizontal cut through a or b—unless more of the trapezoid is flush-lit by another trapezoid. Refer to Fig. 2, left.

3.4 Straddle-lighting

Clip each c^*-trapezoid $T^* \in S_{c^*}^{k-1}$ to the parallelogram P^* using horizontal lines through vertices of T^*; denote the set of the obtained parallelograms by $S^{\succ^\circ}$. Any c-trapezoid straddled by T^* is also straddled by P^*, and thus straddle-lighting with c^*-trapezoids is equivalent to finding dark trapezoids intersected by parallelograms from $S^{\succ^\circ}$. This can be accomplished with a sweep, called (c, c^*)-*sweep*, which discovers the trapezoids from S_c^k in the order of increasing y-coordinate of the lower bases, by sweeping upwards a horizontal line. The sweepline status will be the intersection of the sweepline with the (interiors of) parallelograms from $S_{c^*}^{k-1}$; the status thus is a set of disjoint (open) intervals. The status changes at *parallelogram events* when the sweepline reaches horizontal sides of parallelograms. Because the intervals in the status are disjoint, we can keep them in any ordered structure, e.g., a balanced binary search tree indexed by left endpoints of the intervals. Clearly, the tree handles any of the following three operations in $O(\log n)$ time: (1) adding an interval, (2) removing part of an interval that hits an obstacle edge, and (3) checking whether any of the intervals overlaps with a given query interval. In the last operation, the query interval is a trapezoid lower base; we need it for the trapezoid events, described next.

The main events in the sweep are *trapezoid events* that occur when the sweepline reaches a lower base of a trapezoid (some of the trapezoid events happen simultaneously with parallelogram events; in this case parallelogram events take priority). Suppose that a trapezoid T is the event. We check whether the lower base of T is intersected by the intervals in the sweepline status. If yes, we insert T's upper neighbors into the event queue. In addition, if T is unlabeled, we label it with k.

To initialize the sweep, we "plant" each parallelogram into a "pot" trapezoid; the pots are initially inserted into the event queue. Say that a parallelogram $P^* \in S_{c^*}^{k-1}$ is *planted* into a trapezoid T if the lower side of P^* belongs to T; say also that T is the *pot* of P^* (Fig. 2). Each parallelogram is planted into exactly one pot (even though a pot may have many parallelograms planted side-by-side into it). Now, some c^*-trapezoids from $S_{c^*}^{k-1}$ (such as, e.g., trapezoid T_1^* from Fig. 2, right) have lower bases supported by c-edges of P—the pots for such parallelograms are read off directly from trapezoidation D_c and the lists $L_{c^*}(e)$. The rest of the trapezoids from $S_{c^*}^{k-1}$ (such as, e.g., trapezoid T_2^* from Fig. 2, right, or T^* from Fig. 2, left) are flush with trapezoids from D_c^{k-1}. The pot T' for the parallelogram P^* cut out from such a trapezoid T^* can be determined from the list $L_c(e)$, where e is the edge supporting T^* and T': all that is needed is to locate in which trapezoid from the list the vertex of T^* lands.

We emphasize that clipping by the c-segments is done only to find c-trapezoids straddle-lit by c^*-trapezoids; after the (c, c^*)-sweep completes, the c^*-trapezoids are "unclipped" back to what they were (and in general, during an (a, b)-sweep, b-trapezoids lit at the previous step are only temporarily clipped into (a, b)-parallelograms using a-segments through the vertices).

3.5 Analysis

Flush-lighting takes overall $O(C^2 n \log n)$ time: For every trapezoid T^* that flush-lights c-trapezoids through an edge e, it takes $O(\log n)$ time to locate the vertices a, b of T^* (lying on e) in the list $L_c(e)$. Overall there are $O(Cn)$ trapezoids T^*, and for each we have to locate the vertices a, b in the $C-1$ lists $L_c(e)$; thus the locating takes overall $O(C^2 n \log n)$ time. After the vertices a, b have been located, it takes $O(n_e)$ time to label each (dark) trapezoid T' supported by e, where n_e is the number of the trapezoids that are flush with T^*. Again, overall there are $O(Cn)$ trapezoids T', and each can be flush-lit from at most $C-1$ directions; thus the total time spent in the labeling (not counting the time spent in locating the vertices a, b) is $O(C^2 n)$.

As for straddle-lighting, any trapezoid has $O(1)$ neighbors (assume no two edges of P are supported by the same line); thus, processing an event during any of the sweeps involves a constant number of the priority queue and/or interval tree operations, i.e., $O(\log n)$ time per event. To bound the number of events, observe that any trapezoid inserted in the event queue at step k is either itself intersected by a parallelogram form $S^{\succ^\circ}$, or has a lower neighbor intersected by a parallelogram from $S^{\succ^\circ}$; thus any trapezoid enters the event queue on at most 7 consecutive steps. At any step k, a c-trapezoid may appear in the event queue during each of the $C-1$ (c, c^*)-sweeps. Thus, as there are $O(Cn)$ trapezoids, we have $O(C^2 n)$ events, and the total running time of $O(C^2 n \log n)$.

As for the space, the dominating factor is storing the C trapezoidations.

Acknowledgments We thank anonymous reviewers for helpful comments. VP is funded by the Academy of Finland grant 138520.

References

1. J. Adegeest, M. H. Overmars, and J. Snoeyink. Minimum-link c-oriented paths: Single-source queries. *IJCGA*, 4(1):39–51, 1994.
2. G. Das and G. Narasimhan. Geometric searching and link distance. In *WADS'91*.
3. M. de Berg. On rectilinear link distance. *CGTA*, 1:13–34, 1991.
4. M. de Berg, M. J. van Kreveld, B. J. Nilsson, and M. H. Overmars. Shortest path queries in rectilinear worlds. *IJCGA*, 2(3):287–309, 1992.
5. A. Dumitrescu, J. S. B. Mitchell, and M. Sharir. Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. *DCG*, 31(2):207–227, 2004.
6. R. Fitch, Z. Butler, and D. Rus. 3d rectilinear motion planning with minimum bend paths. In *International Conference on Intelligent Robots and Systems*, 2001.

7. J. E. Goodman and J. O'Rourke. *Handbook of disc. and comp. geom.* CRC Press series on discrete mathematics and its applications. Chapman & Hall/CRC, 2004.
8. R. Güting. *Conquering Contours: Efficient Algorithms for Computational Geometry.* PhD thesis, Fachbereich Informatik, Universität Dortmund, 1983.
9. R. H. Güting and T. Ottmann. New algorithms for special cases of the hidden line elimination problem. *Comp. Vis., Graph., Image Proc.*, 40(2):188–204, 1987.
10. J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *CGTA*, 4:63–97, 1994.
11. J. Hershberger and S. Suri. BSP for 3d subdivisions. In *SODA'03*.
12. H. Imai and T. Asano. Efficient algorithms for geometric graph search problems. *SIAM J. Comput.*, 15(2):478–494, 1986.
13. H. Imai and T. Asano. Dynamic orthogonal segment intersection search. *J. Algorithms*, 8(1):1–18, 1987.
14. D. T. Lee, C. D. Yang, and C. K. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Appl. Math.*, 70:185–215, 1996.
15. A. Lingas, A. Maheshwari, and J.-R. Sack. Optimal parallel algorithms for rectilinear link-distance problems. *Algorithmica*, 14(3):261–289, 1995.
16. A. Maheshwari, J.-R. Sack, and D. Djidjev. Link distance problems. In J.-R. Sack and J. Urrutia, editors, *Handbook of Comp. Geom.*, pages 519–558. Elsevier, 2000.
17. K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit conductors. In *Int. Federation Inf. Proc. Congress*, pages 1475–1478, 1968.
18. G. Neyer. Line simplification with restricted orientations. In *WADS'99*.
19. T. Ohtsuki. Gridless routers - new wire routing algorithm based on computational geometry. In *Internat. Conf. on Circuits and Systems, China*, 1985.
20. M. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13(1):99–113, 1992.
21. G. J. E. Rawlins and D. Wood. Optimal computation of finitely oriented convex hulls. *Inf. Comput.*, 72(2):150–166, 1987.
22. M. Sato, J. Sakanaka, and T. Ohtsuki. A fast line-search method based on a tile plane. In *Proc. IEE ISCAS*, pages 588–591, 1987.
23. D. P. Wagner, R. L. S. Drysdale, and C. Stein. An $O(n^{2.5} \log n)$ algorithm for the rectilinear minimum link-distance problem in three dimensions. *CGTA*, 42(5):376–387, 2009.
24. P. Widmayer, Y.-F. Wu, and C. K. Wong. On some distance problems in fixed orientations. *SIAM J. Comput.*, 16(4):728–746, 1987.
25. C. D. Yang, D. T. Lee, and C. K. Wong. On bends and lengths of rectilinear paths: a graph theoretic approach. *IJCGA*, 2(1):61–74, 1992.
26. C. D. Yang, D. T. Lee, and C. K. Wong. On bends and distances of paths among obstacles in 2-layer interconnection model. *IEEE Tran. Comp*, 43(6):711–724, 1994.
27. C. D. Yang, D. T. Lee, and C. K. Wong. Rectilinear paths problems among rectilinear obstacles revisited. *SIAM J. Comput.*, 24:457–472, 1995.