Minimum-Link Paths Among Obstacles in the Plane

Joseph S. B. Mitchell *
Günter Rote †
Gerhard Woeginger †

December 1990

This paper appeared in *Algorithmica* **8** (1992), 431–459.

Abstract

Given a set of nonintersecting polygonal obstacles in the plane, the *link distance* between two points s and t is the minimum number of edges required to form a polygonal path connecting s to t that avoids all obstacles. We present an algorithm that computes the link distance (and a corresponding minimum-link path) between two points in time $O(E\alpha(n)\log^2 n)$ (and space $O(E)$), where n is the total number of edges of the obstacles, E is the size of the visibility graph, and $\alpha(n)$ denotes the extremely slowly growing inverse of Ackermann's function. We show how to extend our method to allow computation of a tree (rooted at s) of minimum-link paths from s to all obstacle vertices. This leads to a method of solving the query version of our problem (for query points t).

1 Introduction

Motivation. The study of link distance problems is partially motivated by a robot motion-planning problem. Consider a point-size robot that wants to move in a collision-free way from a source position s to some target position t in the plane. Suppose the robot can perform

*SORIE, Cornell University, Ithaca, NY 14853. Email: jsbm@ams.sunysb.edu. Partially supported by NSF grants IRI-8710858 and ECSE-8857642, and by a grant from Hughes Research Laboratories.

†Institut für Mathematik, Technische Universität Graz, Steyrergasse 30, A-8010 Graz, Austria. Email: rote@ftug.dnet.tu-graz.ac.at or woeginger@fmatbds01.tu-graz.ac.at, respectively. This work was begun while Günter Rote and Gerhard Woeginger were at the Freie Universität Berlin, Fachbereich Mathematik, Institut für Informatik, and it was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM). Gerhard Woeginger acknowledges the support by the Fonds zur Förderung der Wissenschaftlichen Forschung, Projekt S32/01.

only two types of movements — straight line motion and pure rotation — and suppose that straight line motion is "cheap" and rotation is "expensive" (but the cost is independent of the amount of rotation). Then a reasonable objective for the robot is to minimize the number of turns that it must make, i. e., it should use a *minimum-link path* from s to t.

Related Work. Suri [19] has studied the problem of finding minimum-link paths in a simple polygon (without holes), obtaining a linear-time algorithm to build a shortest path map in a triangulated polygon. His algorithm is based on the fact that the dual graph of the triangulation is a tree and that in a tree there is a unique path between each pair of vertices. Suri's method does not immediately generalize to polygons with holes; indeed, it is easily seen that there may be an exponential number of paths between two nodes in the dual graph of the triangulation of a polygon with holes. Computing link distances in a polygon with holes has been an open problem.

Several other link distance problems within a simple polygon (without holes) have been studied by Djidjev, Lingas, and Sack [6], by Ke [14], and by Lenhart et al. [16], including the computation of the link radius, link center, and link diameter.

Our Problem. Let P be a polygon (with holes), and let n be the total number of vertices describing P. Let $s \in P$ be a given source point, and let $t \in P$ be a given target point. Our problem is to find a polygonal path from s to t such that the path stays within P and the number of bend points of the path is minimized. We call such a path a *minimum-link path.*

Our Results. We solve the problem of finding a minimum-link path from s to t in time $O(E\alpha(n)\log^2 n)$ (and space $O(E)$), where E is the size of the visibility graph of P and $\alpha(n)$ is the extremely slowly growing inverse of Ackermann's function. We show a lower bound of $\Omega(n \log n)$ for the decision problem that asks if there exists a path from s to t with a specified number of links.

We also solve the problem of finding a tree (rooted at s) of minimum-link paths from s to every other vertex in time $O((E + \ell n)^{2/3} n^{2/3} \ell^{1/3} \log^{\delta} n + E \log^3 n)$ (and space $O(E)$), where ℓ is the maximum link distance from s to any vertex of P, and δ is a constant less than 3.11. We refer to this version of the problem as the SPT-problem (for *Shortest Path Tree* problem).

In all of the above bounds, ℓ is bounded above by n, and E is bounded above by $O(n^2)$, but frequently we may expect that both ℓ and E are significantly smaller than their upper bounds. In the worst case, $\ell = \Theta(n)$ and $E = \Theta(n^2)$, we get a bound of $O(n^2\alpha(n)\log^2 n)$ for the problem of finding an optimal path from s to t and a bound of $O(n^{7/3}\log^{\delta} n)$ for the SPT-problem. In the best case, ℓ may be very small (effectively constant) and E may be $O(n)$, in which case our bounds are $O(n\alpha(n)\log^2 n)$ for the shortest-path problem and $O(n^{4/3})$ (times a polylog term) for the SPT-problem. Note that our algorithm for computing the shortest path to a *given* destination is asymptotically faster in the worst case than the algorithm to compute a shortest path tree to *every* destination. This situation is in contrast with other problems, like shortest Euclidean-length paths in polygons or shortest paths in general graphs, where the single-destination problem is not easier than computing a shortest path tree.

Our Approach. Our algorithm follows the basic methodology of Suri [19], being careful to be able to do illuminations efficiently. We iteratively consider the sets of points at link distance k from s, but we do not try to describe these sets fully (as their boundary descriptions may have complexity $\Omega(n^4)$ [20]). Rather, we describe the boundary only of the cell relevant to finding a minimum-link path to t (or, in the case of the SPT-problem, we describe only those cells that contain portions of obstacle boundaries). The key to the efficiency of our method is the application of recent techniques (e.g., Edelsbrunner, Guibas, and Sharir [8]) to compute a single cell in an arrangement of segments, *without* computing the entire arrangement. We then employ one further trick: we "pull taut" the portions of the cell boundary that are non-obstacle edges, thereby reducing its combinatorial complexity without decreasing its usefulness in the next stage of illumination.

The ideas behind our algorithm are conceptually simple and admit a straightforward implementation, applying directly several results from the existing literature.

Overview of the Paper. The paper is organized as follows. Section 2 describes some notation and basic structural results. In Section 3, we give an outline of the algorithm. Section 4 considers the combinatorial complexities of the occurring configurations. Section 5 gives the algorithmic details and analyzes the running time. Section 6 proves the lower bound result. Section 7 describes the algorithm for the shortest path tree problem, while Section 8 describes its application to the query version of the problem. Finally, Section 9 concludes with remarks about various extensions and open problems.

An earlier draft of this paper appeared as an extended abstract in [18].

2 Preliminaries

A *polygon* (or more clearly, a *polygon with holes*) is a subset of the plane whose boundary is the union of finitely many line segments or half-rays. A polygon that is simply connected or whose complement is simply connected is called a *simple polygon*. Note that this definition allows a polygon (with or without holes) to be unbounded.

Problem Definition. We are given a polygon P (with holes), which we call the *free space*, and two points s (the *source*) and t (the *target*) inside it. We are to find a polygonal path from s to t that lies in the free space, and that consists of as few edges ("links") as possible. This number of edges is called the *link distance* between s and t. For example, in Figure 1 we show an instance in which the link distance from s to t is three.

For simplicity of exposition, we assume that the free space P is bounded. The complement of the free space is called the set of *obstacles* (or holes). It consists of a finite number of simple polygons, one of which is unbounded and surrounds the whole scene. We let n denote the total number of edges bounding P. We also assume that the free space is closed, so that paths are allowed to touch the obstacles or to run along an obstacle edge. We may assume that the free space is connected, since otherwise we can restrict our attention to the component containing s and t (if they lie in the same component). This kind of preprocessing can be carried out in $O(n)$ time. Thus, the obstacles are simple polygons, which are allowed to touch, but must be disjoint. Finally, without loss of generality, we assume that s and t are vertices of P. (We can always make a trivial point-obstacle at s or t.)

We say that a point y is *visible* from another point x if the open line segment (x, y) between x and y lies in the free space, i. e., it does not intersect (the interior of) any obstacle. The *visibility region* V_S for a set S of line segments consists of all points in free space that are visible from any point x on any segment in S. For a fixed source point s, the *k-visibility region* VIS_k contains all points in free space whose link distance from s is at most k.

Note that $\mathrm{VIS}_0 = \{s\}$, and that VIS_1 is the visibility polygon within P with respect to s. (Thus, VIS_1 can be found in time $O(n \log n)$; cf. Suri and O'Rourke [20].) Clearly, each k-visibility region is a polygon (with holes). Imagine that VIS_k is a set of light sources. Then the region VIS_{k+1} is that part of the plane that is illuminated by VIS_k. Moreover, everything that can be illuminated from some region and does not belong to this region can be illuminated from its boundary and vice versa.

When we consider various polygonal regions and cells of the plane (such as VIS_k above), we will generally distinguish between two types of boundary edges: *An illumination edge* runs through free space and is a possible candidate to be used for illuminating the dark area in later stages. This is opposed to parts of *obstacle boundary edges*, which are portions of the boundaries of the original obstacles.

Assume that a polygon Q can be separated from a set of obstacles by a non-intersecting closed polygonal curve. Then, there exists a (unique) shortest such curve γ that encloses Q. The *relative convex hull* of Q (relative to a set of obstacles) is defined to be the region bounded by γ that contains Q.

3 Outline of the Algorithm

The most natural means of solving the link-distance problem is to compute iteratively the k-visibility regions VIS_k from the source point s, for $k = 1, 2, \ldots$, until the target point t is reached. VIS_{k+1} consists of VIS_k plus all points in free space that are visible from the boundary of VIS_k. Hence the $(k+1)$-visibility region can be computed from the k-visibility region.

However, following this approach in a straightforward way may lead to difficulties: Even when the boundary of VIS_k consists of a single edge, the complexity of the VIS_{k+1} may be as high as $\Omega(n^4)$, as shown by Suri and O'Rourke [20]. Computing VIS_{k+2} in turn involves determining the set of points that are visible from any of the $\Omega(n^4)$ boundary segments of VIS_{k+1}, a task which seems a priori to take even longer than $\Omega(n^4)$. Hence, our goal is to avoid actually constructing a full boundary representation of VIS_k.

We use two ideas to avoid these problems:

(1) It may happen that the region that is not yet illuminated at stage k consists of several connected components (cells). We restrict our attention to the cell containing t. (In the SPT-problem, we restrict our attention to cells containing portions of obstacle edges.)

In particular, we avoid computing the $O(n^4)$ convex "shadow" cells that contain no obstacle vertices.

(2) Before we compute what is visible from the boundary of VIS_k, we simplify this boundary by computing its relative convex hull $\overline{\mathrm{VIS}}_k$ with respect to the obstacles. This is allowed because a point is visible from the boundary of VIS_k if and only if it is visible from the boundary of $\overline{\mathrm{VIS}}_k$ (Lemma 2). Thus, for the purpose of computing VIS_{k+1}, the simpler boundary of $\overline{\mathrm{VIS}}_k$ is as good as VIS_k. For the boundaries of the simplified visibility regions $\overline{\mathrm{VIS}}_k$, we will be able to give a good bound on their total complexity during the course of the algorithm (Lemma 4).

More specifically, our algorithm goes as follows. It goes through a number of rounds; in round k we start from a simple polygon G_{k-1}, which is related to $\overline{\mathrm{VIS}}_{k-1}$, and we compute G_k for the next round. G_0 consists of the single point s. Each round consists of four steps:

1. **Cover (part of) VIS_k by triangles.** Given G_{k-1}, we compute a set of triangles T_k that covers the relevant portion of VIS_k.

2. **Compute the cell containing t.** We compute the (single) cell containing t in the arrangement consisting of the triangles T_k and all obstacles. We take the complement of the computed face and and remove from it all isolated obstacles (that are completely surrounded by the face containing t). Let the resulting simple polygon be denoted by F_k. Thus, in addition to the portion of P that has already been illuminated, F_k contains all obstacles whose boundary is at least partially illuminated, together with the unimportant components of the unilluminated plane (those that do not contain t), including all obstacles contained in these components.

   In the case of the SPT-problem we compute *all* cells that contain portions of obstacle edges. By dividing the region into cells, we have split our problem into separate subproblems, and we continue with each cell as in the single-destination case.

3. **Simplify F_k.** We enlarge F_k by adding the relative convex hull of $F_k - \mathcal{O}$ with respect to the obstacles $\mathcal{O}$ that are not contained in the interior of F_k. We also add all obstacles that are touched by the relative convex hull and denote the resulting polygon by $\bar{F}_k$. ($\bar{F}_k$ is related to the simplified visibility region $\overline{\mathrm{VIS}}_k$.)

4. **Clean-up.** The complement of $\bar{F}_k$ may have several cells. We find the cell containing t and let G_k denote its complement. G_k is the input to the next round.

A round of the algorithm is shown in Figures 2–6.

The single-destination algorithm differs from the SPT-algorithm only in Steps 2 and 4, where we need to restrict ourselves to a single cell only. The underlying ideas, however, are the same for both problems.

Now let us look at the successive steps in more detail. In Step 1, we are given the polygon G_{k-1}. When going from stage $k-1$ to stage k, we are interested in representing the set of all points that are visible from the illumination edges of G_{k-1} in such a way that we can find the component of the unilluminated region that contains point t.

We accomplish this by computing a "description" of the visibility region VIS_k of the illumination edges in the relevant portion of the plane. This is done by adapting a technique of Suri and O'Rourke [20]. Suri and O'Rourke showed how to compute for some fixed edge e, a set of $O(n^2)$ triangles that cover the complete visibility region V_e. Analogously, our "description" of the visibility region VIS_k is a set T_k of a quadratic number of triangles that cover the visibility region. The main idea is to do a rotational line sweep around each illuminated vertex and to output each empty triangle over which the line sweeps. The details are given in Section 5. The union of these triangles forms a set of polygons, possibly with holes.

If one of the triangles contains the target point t (which can be checked as the triangles are constructed), we stop with the output *"The link distance from s to t is k."* Otherwise, we notice that the link distance is at least $k+1$ and move on to Step 2.

In Step 2 we get rid of the empty cells that do not contain the point t. Each triangle in T_k is composed of three sides. Moreover, we must consider a number o_k of obstacle edges of all obstacles whose boundary is partially but not completely illuminated. (Recall that obstacles that are not yet touched by light do not play a role in defining F_k.) The goal is to compute in the arrangement of these $s_k = 3|T_k| + o_k$ segments the face that contains the target t. (It is easy to see that there are triangle sides that cannot contribute to the boundary of the illuminated region. For example, every triangle contains one side that also bounds an obstacle; the common side of two adjacent triangles need also not be considered. However, these considerations do not reduce the asymptotic running times of our algorithms.) By directly applying the results of Edelsbrunner, Guibas, and Sharir [8], this can be done in time $O(s_k \alpha(s_k) \log^2 s_k)$. However, since the sum of the terms o_k over all k may be $\Omega(n^2)$, and our goal is to have a bound dependent on E rather than n^2, we must use some additional tricks to avoid dealing with the same (unilluminated) obstacle segments again and again. We let F_k denote the complement of the computed face.

In the case of the SPT-problem Step 2 computes *all* cells that contain portions of obstacle edges. We apply results on computing many faces in an arrangement of segments (e.g., Agarwal [1]). Each of the cells that we keep can then be treated as a separate subproblem, so we can continue exploring them independently. In this case, F_k denotes the complement of all computed cells.

By a *reflex* vertex of F_k we mean a vertex for which the interior angle (the angle pointing towards the interior of F_k) is greater than π. If the angle is less than π, the vertex is a *convex* vertex.

Our first lemma characterizes the boundary of F_k:

Lemma 1

(i) *For all edges e of F_k, there is either some edge s of an obstacle such that $e \subseteq s$ or there exists some point p on the boundary of VIS_{k-1} such that e is part of a light ray emerging from p.*

(ii) *For each convex vertex of F_k, the two incident edges are (parts of) obstacle edges.*

(iii). *The boundary of F_k consists alternately of two types of chains of edges:*

1. *chains that are part of an obstacle boundary, and*
2. *chains of edges in free space consisting only of reflex vertices.*

(In case of the SPT-algorithm, this statement holds for each component of the boundary of F_k.)

Proof. To see (i), assume that e is surrounded by free space. If parts of it were illuminated by a light ray not containing the whole edge, the light ray has to cross e and pass on into the free space on the other side of e. This is a contradiction. For (ii), consider some convex vertex. If one of its two incident edges does not derive from some obstacle, we get from (i) that there is a light ray containing this edge. Since we assumed that a ray is allowed to touch an obstacle, this ray would continue into free space and hence there would be no convex vertex — a contradiction. Finally, the first part of statement (iii) follows from the fact that the illuminated region is connected, and thus there cannot be two disconnected pieces that have illuminated boundaries. The second part is a straightforward consequence of (i) and (ii). □

We will call the reflex chains of illumination edges mentioned above in part (iii).2 of Lemma 1 *illumination chains*. Note that the boundary of F_k may even consist of a single closed illumination chain that encloses a set of isolated obstacles and the point t.

Now we proceed to Step 3. The shape of the region F_k computed in Step 2 is subject to the restrictions of the previous lemma, but it may still be too complicated for our purposes. We simplify it by "pulling taut" the chains of illumination edges that bound F_k. More precisely, let $\mathcal{O}$ denote all obstacles that are either not contained in F_k or that belong to F_k and share some piece of non-zero length with the boundary of F_k. Note that $F_k - \mathcal{O}$ is a simple polygon. Thus, we can form the relative convex hull (as defined in Section 2) of $F_k - \mathcal{O}$ with respect to $\mathcal{O}$. $\bar{F}_k$ is defined to be the union of F_k with this relative convex hull and with all further obstacles that are touched by the relative convex hull. The following lemma justifies this step of the algorithm.

Lemma 2

(i). *All points of free space in $\bar{F}_k - F_k$ are illuminated from the illuminated boundary of F_k.*

(ii). *A point p in free space that does not lie in $\bar{F}_k$ is illuminated from the illuminated boundary of F_k if and only if it is illuminated from the illuminated boundary of $\bar{F}_k$.*

(iii). *Each component of the boundary of $\bar{F}_k$ consists alternately of chains of edges that are parts of obstacle boundaries and of single straight edges through free space (see Figure 5).*

Proof. It follows from Lemma 1(iii) that the relative convex hull just shortcuts the chains of edges in free space that connect reflex vertices, replacing them by "taut-string" paths. Thus, $\bar{F}_k - F_k$ consists of "half-moon"-like pieces that are bounded by two chains: a convex chain and a concave "taut-string" chain. Clearly, every point inside can be seen directly from the convex chain, which is statement (i). Now consider some ray that emerges from some point on the boundary of $\bar{F}_k$. It is clear that if we elongate this ray in the backward direction, it will hit the (illuminated) boundary of F_k. This proves one direction of (ii). The other direction follows by similar arguments.

Statement (iii) follows from the fact that all edges connecting reflex vertices in free space are shortcut by the relative hull. □

Step 4 does some "clean-up". First, we check whether t lies inside the relative convex hull $\bar{F}_k$. If this is the case, it must lie in $\bar{F}_k - F_k$ and we stop with the message *"The link distance from s to t is $k+1$."* Otherwise, we observe that taking the relative convex hull may have disconnected the cell containing t. (For an example, see Figure 6). Therefore, we once again determine the cell that contains t (this time by a simple linear-time test). The complement of this cell is the simple polygon G_k, the output of round k. The illumination edges are exactly the single straight edges traversing free space mentioned in Lemma 2(iii).

In the first round, we start with $G_0 = \mathrm{VIS}_0 = \{s\}$. Since the light source in the first round is simply a point, $\bar{F}_1 = F_1 - \mathcal{O}$, and thus Steps 3–4 are unnecessary for this round.

4 Combinatorial Complexities

In this section and the following, we deal primarily with the single-destination problem; necessary modifications for the SPT-problem are deferred to Section 7. We summarize below some notation that we need in this section.

- The underlying obstacle set contains n edges and n vertices. (Since we assume that the free space is bounded, there can be no infinite edges.)
- The polygon G_k is the output of the k-th round. n_k of its edges are illumination edges. Let $\bar{n}_k$ denote the number of illumination edges of $\bar{F}_k$.
- T_k is the triangle set constructed in Step 1.
- E is the number of edges in the visibility graph of P; in other words, E is the number of visible pairs of obstacle vertices.

Lemma 4 will relate the magnitudes of these numbers to each other and give some upper bounds. First, we need one more property of the polygons $\bar{F}_k$.

Lemma 3 *Consider some obstacle edge e, and let k be the first round in which (part of) this edge belongs to the boundary of G_k. Then*

(i). *As we walk along the boundary of $\bar{F}_k$, e is adjacent to at most four illumination edges of $\bar{F}_k$, and at most two of them can touch e in its interior;*

(ii). *e is adjacent to no edge of $\bar{F}_{k+1}$, $\bar{F}_{k+2}$, etc.*

Proof. (i) Let us first consider those illumination edges that terminate in the interior of e: Those endpoints must already be endpoints of illumination edges in F_k. But since F_k is a simple polygon, even a whole obstacle is adjacent to at most two illumination edges of F_k: If it were adjacent to more than two, this would mean that as we go around the boundary of F_k, we encounter the obstacle twice. We could connect these two touching points by a curve through the (dark) cell containing t and we could close the curve using a path through the obstacle (see Figure 7). The resulting closed curve does not pass through the illuminated region and has illumination edges both inside and outside, contradicting the fact that the illuminated region is connected. Those illumination edges that terminate on an endpoint of e are handled by noting that at each of the two endpoints of e, at most one illumination edge of $\bar{F}_k$ can be adjacent to e when going along a piece of the boundary of $\bar{F}_k$. This concludes the proof of (i).

To see (ii), note that when some point of e is illuminated in round k, in the next round all points on e are illuminated, and thus no portion of e will belong to the boundary of $\bar{F}_{k+1}$. □

Lemma 4 *The following bounds hold, where the summation index k ranges over the number of rounds of the algorithm.*

(a) $\sum n_k \leq \sum \bar{n}_k \leq 2n$

(b) $\sum |T_k| = O(E)$

Proof.

(a) The first inequality is trivial, since the boundary of G_k is essentially one "component" of the boundary of $\bar{F}_k$. By part (iii) of Lemma 2, every illumination edge must be adjacent to two obstacle edges. Together with Lemma 3 this gives the second inequality. (Actually, the union of all polygons $\bar{F}_k$ forms a planar map, since the illumination edges of all polygons $\bar{F}_k$ taken together do not cross. However, we do not need planarity to prove our bound.)

(b) will be shown in Section 5, Step 1. □

The *shortest path map* with respect to a given source point s is the planar map whose boundaries are given by the boundaries of the k-visibility regions VIS_k ($k = 1, \ldots, \ell$). Cells that do not border any obstacle or that border only part a single obstacle edge will be called *trivial* cells. They are necessarily convex (by Lemma 1(iii)), surround no obstacles, and their boundary contains a single, possibly closed, illumination chain. Such an illumination chain will be called a *trivial illumination chain*. For example, the unshaded triangular cell to the right of the pentagon and the one below the top quadrilateral in Figure 4 are be trivial cells of the shortest path map. While the shortest path map can have $\Omega(n^4)$ cells, we prove below a linear bound on the number of *non*trivial cells. A consequence of this is that the total number of illumination chains that do *not* "disappear" in Step 3 when we pull them taut is at most linear in n.

Lemma 5 *There are $O(n)$ nontrivial illumination chains, and hence the number of nontrivial cells in the shortest path map is also $O(n)$.*

Proof. Let us first consider those illumination chains that are either closed or that start and end at the same obstacle edge. They are trivial unless the cell C that they enclose contains further obstacles. In that case at least one obstacle vertex interior to C will be illuminated in the next round, and this vertex will then never be contained in such a cell again. Thus, the number of non-trivial illumination chains that contain at most one obstacle edge in their boundary is at most n. (In fact, it is bounded by the number of obstacles.)

To count the remaining nontrivial illumination chains, consider the graph whose nodes are the n obstacle edges and whose edges are the nontrivial illumination chains that are adjacent to two different obstacle edges. An edge of the graph connects the nodes corresponding to the two obstacle edges that are adjacent to the illumination chain, when viewed from the side of the chain that is more distant from s. Since these chains do not cross each other, this graph is clearly planar. By construction, the graph has no loops. Thus, our proof will be complete once we show that there are at most two parallel edges between any pair of nodes.

Parallel edges correspond to illumination chains that connect the same pair of obstacle edges. The proof that there cannot be three such "parallel" chains is similar to the reasoning in the proof of Lemma 3. In fact, all illumination chains that are adjacent to the same obstacle edge are generated in the same round, and they alternately have the illuminated side on their left and on their right. Thus, if there were more than two such chains, we would find two of them that contain some part of the illuminated region of round k between them, and these two chains together with the two obstacle edges would separate this interior illuminated part from the rest, contradicting the fact that the illuminated region is connected. See Figure 8.
□

5 Details of the Algorithm

Constructing visibility polygons. In Steps 1 and 3, we perform rotational sweeps around certain points. Thus, we need the *visibility polygons* with respect to these points. Given a point p in free space (not necessarily at a vertex) the *visibility polygon* of p is the circular sequence of obstacle vertices and edges that are seen by p. The visibility polygons of all vertices can be computed in $O(n \log n + E)$ time by the visibility graph algorithm of [11], where E is the number of visible pairs of vertices (the "size" of the visibility graph). Given the visibility polygon of p and a point x that is visible from p, we can easily determine the point w where the ray from p to x first hits an obstacle when it is extended past x. In the case that w is different from x we call w an *extension point* and the segment (x, w) an *extension edge*.

Having the visibility polygons about vertices alone is not sufficient for our purposes, since we have to sweep also around the *non*-vertex endpoints of illumination edges that lie in the middle of obstacle edges. We will call these endpoints *illumination endpoints*. We now describe how we get the visibility polygons of such points, given the visibility graph.

Clearly, there are at most $2E$ extension points in all the visibility polygons about the vertices; thus, we can sort all extension points of visibility graph edges that lie on a common obstacle edge in a total time of $O(E \log n)$.

Consider (continuously) sliding a point p along some obstacle edge (x, y) starting from vertex x. Our goal is to maintain the visibility polygon of p during this sliding motion, since this will allow us to determine the visibility polygons about each illumination endpoint along (x, y). Note that the visibility polygon about p changes only when p passes an extension point on (x, y). Since we have sorted the extension points along every obstacle edge, and since we know for each extension point its corresponding pair of vertices, we can update the visibility polygon of p in constant time per extension point over which p slides.

Thus, the total time to maintain visibility polygons while sliding p along every obstacle edge is only $O(E)$. Also, the size of the visibility polygon about a point p on some obstacle edge (x, y) is bounded above by the size of the visibility polygon about x plus the number of extension points along the subedge (x, p). Now, by Lemma 3, we know that there are at most two illumination endpoints along each obstacle edge about which we will require visibility. Thus, within overall time $O(E)$ (after $O(E \log n)$ preprocessing) we can compute the visibility polygons about every point that we require. For each visibility between an illumination endpoint and a vertex that we find in this way, we create a visibility edge in an angularly ordered list about the illumination endpoint, we insert this visibility edge into the list of visibility edges about the vertex, and we also insert an extension edge from the vertex to correspond to the newly created visibility edge.

Note that the visibility polygon that we compute about a non-vertex point p does not include visibility information between p and other extension points or illumination endpoints. (If we were to include also the visible pairs *among* extension points and illumination endpoints, we could not maintain our $O(E)$ bound.)

Step 1: Constructing the k-visibility region. We want to compute the region of points visible from a set of illumination edges S on the boundary of G_k. We do this by finding a set of triangles that cover the region illuminated (in the same spirit as the method of [20]). Our main concern will be to identify those visibility graph edges and extension edges that bound these triangles, i.e., that will become the illumination edges of the next round.

Given the $O(E \log n)$ preprocessing as described above, we can assume that we know the (extended) visibility graph, and we have the extension points in sorted order along each obstacle edge. The illumination edges $e \in S$ will be line segments e that either join a (visible) pair of obstacle vertices or that join an obstacle vertex to a (visible) extension point. Now, a vertex will be illuminated by S if and only if it is either visible from an endpoint of some $e \in S$ or there is a visibility edge or extension edge incident to it that crosses some illumination edge $e \in S$. We claim that we can identify all of the visibility edges and extension edges that cross segment e in time proportional to the number of such edges.

The main idea is to use the method of Mitchell and Welzl [17], which allows one to update a visibility graph when a new obstacle is inserted in time $O(n + K)$, where K is the size of the change. We outline here the method as it applies to our problem.

We define a graph $\mathcal{G}$ as follows. $\mathcal{G}$ has a node associated with (a) each visibility graph edge, (b) each extension edge of each visibility graph edge, and (c) each of the visibility edges joining an illumination endpoint to the vertices seen by it. The discussion above showed how we compute the visibility edges of type (c), and we argued that there can be only $O(E)$ such edges in total. Thus, the total number of nodes that can ever appear in $\mathcal{G}$ is $O(E)$.

We now define the edges of graph $\mathcal{G}$ (see Figure 9):

(1) If two nodes a and b of $\mathcal{G}$ correspond to two edges that are incident to a common vertex v and are consecutive in the angular order about v, then we join a and b by an edge in $\mathcal{G}$.

(2) If visibility edges $a = (u, v)$ and $b = (z, v)$ are consecutive in the angular order about vertex v, and $b = (z, v)$ is a visibility graph edge that has an extension edge $c = (w, z)$ at z, then we join a and c by an edge in $\mathcal{G}$.

We now fix any one illumination edge $e = (u, v) \in S$. Color a node of $\mathcal{G}$ "blue" if its corresponding segment intersects e (either by crossing e or by sharing an endpoint with e). The blue nodes correspond to those segments that we wish to identify. We already know many blue nodes: all of those visibility graph edges and extensions that are incident to an endpoint of e are known and are blue. We want to identify the remaining blue nodes of $\mathcal{G}$, which we do by a simple linear-time depth-first search starting from the known blue nodes.

The two crucial facts that allow us to achieve the claimed time bound are:

(i) the degree of any node of $\mathcal{G}$ is bounded (at most 8); and

(ii) the blue nodes of $\mathcal{G}$ define a connected subgraph.

Observation (i) is straightforward from the definition of edges of graph $\mathcal{G}$, and is important in assuring that the search can be accomplished in time proportional to the number of blue nodes found.

For observation (ii), we refer to Figure 10. Consider a blue node a that corresponds to a segment (x, y) that crosses $e = (u, v)$ at point z. Note that either x or y is a vertex; assume without loss of generality that x is a vertex. Similarly, assume without loss of generality that u is a vertex. Now, the path from x to z to u is in free space. When this path is pulled taut, we either get the segment (u, x) (in which case we are done, since (u, x) is blue if it is visible), or we get a concave path from x to u that bends only at vertices $(p_1, p_2, \ldots, p_m)$ of obstacles. Our goal is to exhibit a path within the graph $\mathcal{G}$ from blue node $a = (x, y)$ to the blue node (u, p_m), such that every node along the path is blue.

We march through the (clockwise) list of visibility edges about x, starting at (x, y) and stopping at the edge (x, q_1) just before edge (x, p_1). Each edge of this march must be blue, since the region R bounded by the concave chain and the edges (x, z) and (u, z) must be obstacle-free. From the edge (x, q_1) (which is blue) there is a connection to the extension edge of the (visibility graph) edge (x, p_1), and this extension edge must cross e (and therefore be blue). Starting with this extension edge (which is incident on vertex p_1), we march through the (clockwise) list of visibility edges about p_1, stopping just before reaching (p_1, p_2). This process continues until we reach the visibility graph edge (u, p_m), which we know to be blue since it is incident to an endpoint of e. We have exhibited a path of blue nodes from (u, p_m) to (x, y), so we have shown that the set of all blue nodes can be discovered by a depth-first search in time linear in the number of blue nodes.

When carrying out the depth-first search, we need not construct the graph $\mathcal{G}$ explicitly. We simply start with the type (c) edges and explore for each blue edge all adjacent edges, testing each to see if it is blue. (Here, adjacency is taken with respect to $\mathcal{G}$.) The adjacent edges of an edge can be determined in constant time from the visibility graph information that we have stored.

The illumination edges e are now considered one by one; for each e, we apply the above method to obtain the set of all visibility edges that cross e in time proportional to the number of crossings found, and we delete these (blue) nodes of $\mathcal{G}$ from further consideration. Then, when we consider the next illumination edge, we are working with the new (modified) graph $\mathcal{G}$ that corresponds to having considered e as a non-transparent "obstacle". Once we have identified the set of visibility edges that cross illumination edges S, we know which obstacle vertices are illuminated, and it is then straightforward to output a set of triangles that cover the region illuminated by S. The number of triangles in the covering will be proportional to the number of visibility edges that cross S. Since we delete all edges as soon as they give rise to a triangle, and since the total number of visibility edges considered is $O(E)$, we have concluded a proof of Lemma 4(b).

If we consider the $O(E \log n)$ preprocessing overhead, we get for the total time spent in Step 1 during all rounds,

$$O(E \log n) + O(E) = O(E \log n) \tag{1}$$

As we generate an edge that may contribute to the boundary of VIS_k, we know which side of it is illuminated and which side may be dark. This knowledge will be used in the next step, when we compute a dark face in the arrangement of these edges.

Step 2: Computing a face. Edelsbrunner, Guibas, and Sharir [8] show how to compute a single face in an arrangement of N line segments in time $O(N\alpha(N)\log^2 N)$. However, applying this directly would yield a complexity of $O(n^2\alpha(n)\log^2 n)$, because it may happen that the same set of $\Omega(n)$ obstacle edges is part of the boundary again and again, for $\Omega(n)$ rounds.

Thus, we find the cell F_k only implicitly as far as its "dark" (obstacle) boundary is concerned. First, we look at the arrangement of the $O(|T_k|)$ segments given by the triangles from Step 1, and we also include the $O(|T_k|)$ obstacle edges that are touched by these illumination edges. Now we determine the cell containing t in this arrangement by the algorithm of [8], in time $O(|T_k|\alpha(|T_k|)\log^2|T_k|)$. We know that all chains of illumination edges of F_k are contained in the boundary of this cell. By Lemma 1, we can throw away all parts of the boundary that do not belong to chains of reflex angles between points on an obstacle, and we can also ignore pieces that border the face from the "wrong" side, i. e., from the side where the light should be, or from inside an obstacle. Now we try to trace out the boundary of the "true" cells in the complete arrangement with all obstacle segments included. To do this, we have to solve two problems: When we "fall off" the boundary because we should move from an obstacle edge to an adjacent obstacle edge that isn't there, we should know in which place we would come back to an edge of our restricted arrangement if we were to follow the correct cell boundary along the obstacle. The second problem is to determine which of the cells that we trace contain the point t.

In order to solve the first problem, we must jump over chains of obstacle edges that are not touched by an illumination edge without looking at each edge individually. In other words, we have to find the next "marked" edge around the obstacle following a given one, where "marked" means being illuminated, i. e. being touched by an illumination edge. If we store the edges of each obstacle as the leaves of a binary tree, with each node knowing whether any of the leaves in its subtree are marked, we can easily accomplish such queries in $O(\log n)$ time. Marking an edge as being touched (illuminated) takes also at most $O(\log n)$ time. (Note that we can leave an edge marked once we have marked it, since an edge will remain illuminated after it has been illuminated for the first time.) We query for the next (or preceding) marked edge at most once for each obstacle edge, because we do it only when we come from the left-most or right-most touching illumination edge, and in the following round, there won't be any more illumination edges touching this obstacle edge. Thus, the total time for all such queries is $O(n \log n)$.

The second problem amounts to testing whether t is contained in a cell that was traced out only implicitly, as described above. The standard test for point-in-polygon containment determines whether a given half-ray starting from t intersects the boundary of the polygon an even or an odd number of times. We can precompute this number for a given fixed half-ray through t, for all obstacle edges. (For a single edge this number can only be 0 or 1.) By storing the partial sums sequence of each obstacle in a table, we can retrieve the number of intersections with the half-ray for any connected part of the obstacle boundary in constant time. Thus, the time for testing containment of t in a cell will not exceed the time that we take to trace out the cell.

Note that isolated obstacles (within the unilluminated region) are implicitly ignored in the above method, and therefore the final cell that we obtain is indeed the true boundary of F_k as we defined it.

For the running time of Step 2, we obtain

$$\begin{aligned} O(n \log n) + \sum_k O(|T_k|\alpha(|T_k|)\log^2|T_k|) \\ = O(E\alpha(n)\log^2 n), \end{aligned} \tag{2}$$

by Lemma 4(b).

Step 3: The relative convex hull. In order to shortcut an illumination chain in free space, we start walking along the chain at one endpoint, extending a rubber band whose end is fixed at this endpoint. Computationally we do this by a rotational sweep around the endpoint of the chain and a concurrent scan of the chain. Whenever we sweep over an obstacle vertex we simply test whether it is in front of or behind the chain. In the latter case, we simply ignore the obstacle; in the former case, we have found an edge of the relative convex hull, and we start sweeping around the new obstacle vertex (cf. Figure 11).

There is only one case when there is a problem with this method, namely, when the boundary of F_k consists of a single closed illumination chain. In this case, we do not have a starting point for which we know its visibility polygon so that we could sweep around this point. In such a case we take an endpoint of any illumination edge that contributes to the chain, and extend the rubber band from there. Refer to Figure 12. We may need to sweep around some obstacle vertices that are outside the illumination chain before we hit the first obstacle vertex inside the chain, but eventually, after at most two full turns, we obtain the relative convex hull (which, in this case, is simply the convex hull of the obstacles enclosed by the chain).

It is easy to see that we sweep around each point (obstacle vertex or illumination endpoint) at most once; hence, the total effort for looking at other obstacle vertices from the points around which we sweep is $O(E)$. In addition, we have to traverse concurrently the chains. This can be done in time linear in the size of the chains. The total size of all illumination chains is $O(E\alpha(n))$, since, by Lemma 4(b), there are only $O(E)$ segments that participate in the boundary of the single face that we compute. This gives a total time bound of

$$O(E\alpha(n)). \tag{3}$$

(We have already accounted for the preprocessing in Step 1.)

Step 4: Cleaning up. Determining the components of the boundary of $\bar{F}_k$ and selecting the cell in which t lies can certainly be done in linear time. Thus, we get for Step 4:

$$\sum_k O(\bar{n}_k) = O(n). \tag{4}$$

Finding a path; putting the results together. The algorithm described above will stop after t has been reached, and it will correctly report the link distance from s to t; If we also want to output a minimum-link path, we have to store with each illumination edge that is generated during the algorithm a backward pointer that shows where the edge "comes from". A path can then be found by tracing the pointers back from t.

This is no problem for edges that start at boundary edges of F_k, which were generated in Step 2: The backward pointer simply points to the respective "source" edge, which will have is own backward pointer. However, if an illumination edge originates at an edge of $\bar{F}_k$, that was only created in the computation of the relative convex hull in Step 3, the shortest direction to the source is not so clear. For these edges, we have to do more work. The regions that are added to F_k to form the relative convex hull have the shape of half-moons (see the region swept by the rubber band in Figure 12); they are bounded by a convex chain (belonging to the boundary of F_k) and a concave chain (belonging to the boundary of $\bar{F}_k$). (Here, convex and concave are taken with respect to the interior of the region.) Now, all edges that originate at the illumination edges of $\bar{F}_k$ have to be extended backwards until they hit F_k. Their backward pointer has to point to the edge of F_k where they hit. The hitting edge can be determined by binary search in the convex chain in $O(\log n)$ time per backward pointer. Since the total number of illumination edges is $O(E)$, by Lemma 4(b), this adds a total of $O(E \log n)$ to the complexity of Step 1, which does not change the bound (1).

Adding equations (1)–(4) yields then the following theorem:

Theorem 6 *A minimum-link path between two points in a polygon (with holes) P with n vertices can be computed in time $O(E\alpha(n)\log^2 n)$ and space $O(E)$, where E denotes the size of the visibility graph of P and $\alpha(n)$ denotes the extremely slowly growing functional inverse of Ackermann's function (cf. [12]).* □

6 An $\Omega(n \log n)$ Lower Bound

In this section we show a lower bound for the link distance problem in the algebraic computation tree model (see Ben-Or [2]). We do this for the weakest possible formulation of our problem, namely for the decision problem of deciding whether the link distance between two points s and t in a polygon with obstacles is at most a given number L. We reduce the following "separation problem" to the link distance problem:

> **INSTANCE:** A set of n real numbers $x_1, \ldots, x_n$ with $|x_i - x_j| \geq 4$ for all $i \neq j$.
>
> **QUESTION:** Is it also true that $|x_i - x_j| \geq 6$ for all $i \neq j$?

A lower bound of $\Omega(n \log n)$ for this problem has been proved by Ó'Dúnlaing [7]. To reduce this problem to the link distance problem, we first add a constant to all numbers so that we have $x_i \geq 6$ for all i. Then, for each given number x, we construct a box-shaped obstacle in the form of the following polygonal chain (see figure 13): $(0, x-1)$, $(0, x-2)$, $(-x, x-2)$, $(-x, -x)$, $(x, -x)$, (x, x), $(-4x/5 - 24/5, x)$, $(-4x/5 - 24/5, x-1)$. By the separation assumption, these boxes are disjoint. We place point s at the origin and t at, say, $(0, 5 + \max x_i)$, and we set the bound L on the link distance to $2n + 1$.

Now it is easy to see that any minimum-link path from s to t must "wiggle" its way out through the "lids" of the boxes, implying that it will contain at least one left-to-right edge and at least one right-to-left edge per box, plus an initial edge starting from s, adding up to a total of at least $2n + 1$ edges. Thus, to achieve this bound, we may use just one left-to-right edge and one right-to-left edge per box. The "best" point that can be reached in one right-to-left edge through the lid of the box for x is the point $(-x', x-1)$, where x' is the number after x in the sorted order. In order to see from this point directly to the next horizontal right-to-left edge at y-coordinate $x' - 1$, the edge leaving point $(-x', x-1)$ must pass above the obstacle corner at $(-4x/5 - 24/5, x)$ and below the corner at $(0, x'-2)$. An easy calculation shows that this is possible if and only if $x' - x \geq 6$, which proves the following theorem.

Theorem 7 *Computing the link distance between two points in the presence of polygonal obstacles with a total number of n vertices takes $\Omega(n \log n)$ time, in the algebraic computation tree model.*

7 The Shortest Path Tree Problem

In this section, we discuss the extension of our results to the problem of computing a tree of shortest paths from s to every obstacle vertex. We follow the structure of Section 5 in giving the details of the successive steps of the algorithm, indicating what changes are necessary for the SPT-problem.

Constructing visibility polygons. As before, we compute the visibility graph and the set of extension edges in time $O(E + n \log n)$. The new complication lies in the fact that we need visibility information about illumination endpoints in *all* cells, and there may be more than just two per obstacle edge. In fact as many as $\Omega(n)$ illumination endpoints may lie on a single obstacle edge. Storing the full visibility polygon for each of these points may blow up the complexity to at least $\Omega(nE)$ in the worst case. Thus we must find a way to get along with "partial" visibility polygons.

When some part of an obstacle edge is illuminated for the first time, we first identify the illumination endpoints lying on this edge for which we need visibility information. They are those endpoints that actually lie on the boundary between the illuminated region and the dark region. We can ignore those endpoints such that all of free space in some neighborhood of the endpoint is illuminated at the same time.

To be specific, let us look at an obstacle edge (a, b), drawn horizontally, with a on the left, b on the right, and the obstacle above. In the round in which the edge (a, b) is first hit by some illumination edge, we sort all illumination endpoints and all extension points of visibility graph edges that lie on (a, b). The illumination endpoints on this edge can be classified as either "left" or "right", depending on whether the illuminated side is to the left or right of the endpoint. By simply scanning the edge (a, b), keeping track of how many left and right illumination endpoints have been encountered, we can determine those endpoints that lie on the boundary between the illuminated region and the dark region. (The initial count of how many triangles cover the initial part of the edge near a can be determined as the triangles are generated.)

Now we make two scans along (a, b) (one in each direction) in order to determine the visibility information for the illumination endpoints on (a, b). During the right-to-left scan we store visibility polygons for all "left" endpoints, and vice versa for the left-to-right scan. We slide a point p from b towards a, maintaining a list of visible points as we go. We initialize the list to be those obstacle vertices visible from b, and each time we pass an extension point, we make the necessary (constant-size) change. Now, the difference between what we do here and what was done in the single-destination case is the following: When the sliding point p comes to a left endpoint w, we store only some *part* of the current list of visible vertices about p as the visibility information for w, and we *delete* this part from the current list. The part that we store is the part that is directed into the dark sector in the neighborhood of w. It is clear that those visibility edges that are directed to the side that has already been illuminated, are not important for determining future visibility regions. The only other thing that has to be checked is that we do not commit an error by deleting too much. This is formulated in the following lemma:

Lemma 8 *Let w and w' be two "left" illumination endpoints on the obstacle edge (a, b), as above, with w to the right of w' (see Figure 14). Let (w, v) and (w', v) be visible pairs such that the edges (w, v) and (w', v) are directed into the (local) dark sector about w and w', respectively. Then the pair (w', v) will not generate an illumination edge in the next round of the algorithm.*

Proof. In order for a visibility edge (w, v) to generate an illumination edge in the next round of the algorithm, the segment (w, v) must lie entirely within the "dark" region on which w is a boundary point.

Let H be the connected component of the dark region on which w' is a boundary point. Assume that (w, v) and (w', v) leave w and w' to enter the dark sector about these points. We will show that the segment (w', v) cannot lie within H, thereby proving that (w', v) cannot generate an illumination edge at the next round of the algorithm.

There is a k-link path π from s to w; there is also a k-link path π' from s to w'. In the neighborhood of w, the path π lies within the triangle $\triangle vww'$, while in the neighborhood of w' the path π' lies outside $\triangle vww'$. Thus, the path Π from w to w' obtained by appending paths π and π' must either cross the edge (v, w) or cross the edge (v, w'). If the path Π crosses (v, w) at a point z, then z must be at link distance at most $k-1$ from s (see Figure 14). But then the segment (z, w) belongs to the illuminated region and (w, v) does not really enter the dark sector about w — a contradiction to our assumption. Thus, the path Π must cross (exactly once) the boundary of $\triangle vww'$ along the edge (v, w'). This implies that the path Π, together with the segment $(w'w)$, forms a closed Jordan curve separating w' from v, so that v cannot be in the dark region H.□

The following lemma extends Lemma 4 to the SPT algorithm:

Lemma 9 *The total number of illumination edges determined in all rounds of the algorithm is $O(E)$.*

Proof. As we march along (a, b) from b to a, we start with the visibility polygon of b as the current list of visible points, and we add or delete a point whenever we pass an extension point of the original extended visibility graph. Since every visibility edge that we store with a left illumination endpoint is deleted from the current list, the total number of visibility edges that are stored with left illumination endpoints on this edge is therefore bounded by the size of the visibility polygon of b plus the number of extension points on (a, b). Since there is at most one left-to-right sweep and one right-to left sweep for each obstacle edge, the total number of visibility edges that are stored for illumination endpoints in the middle of obstacle edges is at most twice the total size of the visibility polygons of all obstacle vertices plus twice the total number of extension points of the original visibility graph. Since both of these numbers are $O(E)$, we get a bound of $O(E)$ for the total number of visibility edges that we store about illumination endpoints; the number of visibility edges that are stored about obstacle vertices is clearly $O(E)$. Since an illumination edge can arise only as an extension of a visibility edge, the lemma follows. □

Lemma 10 *Finding the necessary visibility edges about all illumination endpoints, as described above, can be carried out in $O(E \log n)$ time.*

Proof. When processing an obstacle edge, we first have to sort all points on that edge, i. e., all extension points of the original visibility graph and all illumination endpoints (whether or not they actually lie on the boundary between the illuminated and the dark regions). By the previous lemma, the total number of illumination endpoints is $O(E)$, and the total number of extension points is clearly $O(E)$. Thus, the sorting can be done in $O(E \log E) = O(E \log n)$ time. The remaining steps can be carried out in linear ($O(E)$) time. □

Step 1: Constructing the k-visibility region. This step of the algorithm is exactly as in the single-destination case.

If we consider the $O(E \log n)$ preprocessing overhead for the visibility calculations, we get for the total time spent in Step 1 during all rounds,

$$O(E \log n) + O(E) = O(E \log n) \quad (5)$$

Step 2: Computing the interesting faces. This step requires the greatest change, since now we do not restrict our attention to just a single cell in the arrangement of the $s_k = 3|T_k| + o_k$ segments of the covering triangles and obstacle edges. (Actually, we would not have to look at obstacle boundaries that are already fully illuminated. However, in the worst case, the number of obstacle edges that we have to consider could still not be as big as $\Omega(n)$.) Instead, we appeal to the results of Agarwal [1] to compute many faces in this arrangement. The faces that we want to compute are those that contain unilluminated vertices (of which there are o_k at stage k). A straightforward application of [1] then yields a (deterministic) time bound of $O(o_k^{2/3} s_k^{2/3} \log s_k \log^{\omega/3+1} \frac{s_k}{\sqrt{o_k}} + s_k \log^3 s_k + o_k \log s_k)$, where ω is a constant less than 3.33. Since $s_k = O(n^2)$ and $o_k = O(n)$, the bound is $O(n^{2/3} s_k^{2/3} \log^{\delta} n + s_k \log^3 n)$, where $\delta = 2 + \frac{\omega}{3}$ is a constant less than 3.11. Since $\sum_k s_k \le E + \ell n$, and the index k runs from 1 to ℓ (the maximum link distance from s to a vertex), we get a total bound of

$$
\begin{aligned}
& O(\ell(\frac{E+\ell n}{\ell})^{2/3} n^{2/3} \log^{\delta} n + E \log^3 n) \\
= & O((E+\ell n)^{2/3} n^{2/3} \ell^{1/3} \log^{3.11} n + E \log^3 n) \qquad (6)
\end{aligned}
$$

on the work involved in doing Step 2 (using Hölder's inequality).

Note that we are computing (explicitly) the exact cells containing unilluminated vertices, rather than using the method of implicitly finding a cell (as we did in the single-target case). It should be possible to use the implicit method of finding the cells of interest, in which case the term $(E + \ell n)^{2/3}$ can be replaced by $E^{2/3}$. However, we do not know of a way to avoid the term $O(E + \ell n)$ in the time complexity of Step 3, so making this slight improvement here would yield only a minimal improvement in the overall time bound.

To remain consistent with our definition of F_k, we would have to scan the boundary of F_k and remove each boundary component that is just the boundary of a single "isolated" obstacle, in a total time of $O(n\ell)$. However, in contrast to the single-destination algorithm, it has no impact on the time complexity whether we define F_k "without isolated obstacles" or with them. Thus we can also omit this scan for simplicity.

Step 3: The relative convex hull. The set F_k will in general have many holes, with each hole corresponding to a connected component of unilluminated space that contains one or more vertices. In this step, we want to "pull taut" each component of the boundary of F_k.

For a boundary component that consists of a single (closed) illumination chain, the taut version of the chain is simply the convex hull of the obstacles inside this chain. We can find this convex hull easily once we know one vertex that is on the hull: we simply use the visibility graph information to trace out the hull. The problem is to obtain a single vertex on the convex hull. This can be done in time linear in the number of vertices inside the chain (e.g., by finding the lowest vertex within each such chain). The overall cost is then $O(\ell n)$, since this must be done at each of the ℓ rounds of the algorithm.

For a boundary component of F_k that includes at least some portion of some obstacle boundary, we can proceed as in the single-destination case to pull taut each illumination chain, knowing that the work can be charged off to edges of the visibility graph and the turn points of the illumination chains. Thus, for this step of the algorithm, we get a total time bound of

$$O(E + \ell n) \tag{7}$$

plus the total boundary complexity of all illumination chains, which is dominated by the time necessary to compute the interesting faces (Step 2).

Step 4: Cleaning up. This step is not relevant for the SPT-problem.

Finding a path; putting the results together. If we want to output minimum-link paths, not just link distances, we simply store a backward pointer with each illumination edge that is generated during the algorithm, in a manner similar to that of the single-destination case. With this information, we can trace back to s from any vertex to output a path. Adding equations (5)–(7) yields then the following theorem:

Theorem 11 *Let P be a polygon (with holes) with a total of n vertices. A tree of minimum-link paths from a given point s to every other vertex of P can be computed in time $O((E + \ell n)^{2/3} n^{2/3} \ell^{1/3} \log^{\delta} n + E \log^3 n)$ and space $O(E)$, where E denotes the size of the visibility graph of P, ℓ denotes the link length of the longest path from s to a vertex of P, and δ is a constant less than 3.11.* □

Note that the "tree of minimum-link paths" is not a tree when it is laid out geometrically in the plane, since different branches may cross each other. It is a tree when viewed topologically, or as a data structure with backward pointers.

8 Link Distance Queries

When we solve the SPT-problem we end up knowing the link distance from s to every point on every visibility graph edge and every illumination edge. Thus, if we compute the arrangement $\mathcal{A}$ of obstacle boundary segments together with all the illumination edges produced during the solution to the SPT-problem, we will end up with an arrangement of segments such that if we locate a query point t within a cell of $\mathcal{A}$, we will be able to report the link distance to t (and follow back pointers to retrieve a minimum-link path). Using an output-sensitive algorithm (e. g., Chazelle and Edelsbrunner [5]) to build $\mathcal{A}$, then, we can construct the full shortest path map for our problem in the same complexity as our solution to SPT-problem plus the size of the map, which is no worse than $O(E^2)$, since $O(E)$ is an upper bound on the number of illumination edges. The map can be preprocessed in time $O(|\mathcal{A}|) = O(E^2)$ to support $O(\log(|\mathcal{A}|))$ time point location queries, using the results of Edelsbrunner, Guibas, and Stolfi [8], together with the recent triangulation result of Chazelle [4] to perform the regularization of the map. (It is not hard to see that the regularization can also be done within the $O(E^2)$ bound by methods other than applying [4].) The worst-case running time of this method is $O(n^4)$. We have thus proved the following theorem:

Theorem 12 *Given a fixed source point s, one can compute the shortest path map (with respect to s) in time $O(K + (E + \ell n)^{2/3} n^{2/3} \ell^{1/3} \log^{3.11} n + E \log^3 n)$ and space $O(K)$, where $K = O(E^2)$ is the size of the map. Given the shortest path map with respect to s, the link distance from s to any query point $t \in P$ can be found in time $O(\log n)$, and a minimum-link path from s to t can be reported in time $O(\log n + k)$, where k is the length of the path.*

If we are willing to give up the logarithmic query time, we can get an algorithm for the query version of the problem that uses, in the worst case, much less preprocessing time and much less space than the method outlined above. In particular, we construct the map $\mathcal{M}$ consisting of obstacle edges and the taut versions of nontrivial illumination chains. By Lemma 5, we know that the size of $\mathcal{M}$ is only $O(n)$. Furthermore, $\mathcal{M}$ can be found by running the algorithm for the SPT-problem, in time $O((E + \ell n)^{2/3} n^{2/3} \ell^{1/3} \log^{3.11} n + E \log^3 n)$. This time bound includes the time necessary to triangulate $\mathcal{M}$ and preprocess it for point location. Given $\mathcal{M}$, we can determine the link distance to a query point t in time $O(n)$ as follows. We first locate t within a cell C of the map (in time $O(\log n)$). Knowing C already tells us the link distance to t within an accuracy of 1: The link distance from s to any point t in C is either k or $k+1$, where k is the round in which C was generated, depending on whether t belonged to the k-visibility region VIS_k originally or whether it got "across the boundary" because it was contained in the relative convex hull $\overline{\mathrm{VIS}}_k$. The boundary of C consists of parts of the boundary of $\overline{\mathrm{VIS}}_k$, parts of the boundary of $\overline{\mathrm{VIS}}_{k-1}$, and parts of obstacle edges. In order to tell if the true link distance of t is k or $k+1$, we must know whether or not t can see any part of that portion of the boundary of C coming from $\overline{\mathrm{VIS}}_{k-1}$ (by Lemma 2(ii)). Thus, we just have to compute the visibility polygon of t within C, which can be done in time linear in the size of C. (See El Gindy and Avis [10] and Lee [15]; see also Joe and Simpson [13], for a correction to [10, 15].)

Theorem 13 *Given a fixed source point s, one can compute a planar map of size $O(n)$ in time $O((E + \ell n)^{2/3} n^{2/3} \ell^{1/3} \log^{3.11} n + E \log^3 n)$, such that, given the map, the link distance from s to any query point $t \in P$ can be found to within accuracy 1 in time $O(\log n)$, and can be found exactly in time $O(n)$ (after which a minimum-link path from s to t can be reported in time $O(k)$, where k is the length of the path).*

9 Conclusion

We have shown that minimum-link paths in multiply-connected domains can be found in nearly quadratic time. Our algorithms are conceptually simple and apply several recent techniques from the literature. The time bottleneck in our algorithms is the computation of cells in an arrangement (Step 2), both for the single-destination case and for the SPT-problem. An improved time complexity for the computation of a single cell or of many cells in an arrangement of line segments would immediately translate into an improvement for computing the link distance. Also, it may be that through more careful analysis or use of data structures our bounds could be improved (e.g., to $O(E \log n)$ or $O(E+n \log n)$). In particular, if we could perform the illumination step (Step 1) for a set of *crossing* illumination edges in the same output-sensitive time bound as we currently do for non-crossing illumination edges, the algorithms would simplify greatly, since we would not need to compute cells and relative convex hulls at all.

Our algorithm relies very much on the structural information that is provided by the visibility graph of the given polygon. The visibility graph or some related structure is also used in most algorithms for computing Euclidean shortest paths in the presence of obstacles. The visibility graph, as well as the link distance, is invariant under arbitrary affine transformations and even under some projective transformations (assuming that they do not turn the bounding polygon inside out). Such transformations, however, may drastically change the metric (Euclidean) structure of a problem. In this sense, the visibility graph is a more naturally suited tool for computing the link distance than for computing the Euclidean distance.

Our methods should permit efficient solutions to various other problems (e.g., link radius, link diameter, and link center) involving link distances in multiply-connected regions.

We are also interested in the three-dimensional problem of computing minimum-link paths. While Canny and Reif [3] have shown that Euclidean shortest path problems in three dimensions is NP-hard, it is open whether or not methods similar to ours may yield a polynomial-time solution for the link-distance problem in higher dimensions. In higher dimensions, however, the visibility regions (VIS_k) will be bounded by curved surfaces, which may make it difficult to extend our method in an easy way.

Acknowledgements. J. Mitchell thanks E. Arkin, R. Freimer, C. Piatko, S. Khuller, and E. Wynters for helpful discussions on the content of this paper. G. Rote and G. Woeginger thank Mark Overmars for fruitful discussions in the early stage of this work.

References

[1] P.K. Agarwal, Intersection and decomposition algorithms for arrangements of curves in the plane, Ph.D. Dissertation, Robotics Report No. 207, New York University, Dept. of Computer Science, August, 1989.

[2] M. Ben-Or, Lower bounds for algebraic computation trees, *Proc. 15th ACM Symposium on Theory of Computing*, 1983, pp. 80–86.

[3] J. Canny, J. Reif, New lower bound techniques for robot motion planning and real geometry, *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 39–48.

[4] B. Chazelle, Triangulating a simple polygon in linear time, *Discrete and Computational Geometry* **6** (1991), 485–524.

[5] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, *J. Assoc. Comput. Mach.* **39** (1992), 1–54.

[6] H.N. Djidjev, A. Lingas, and J. Sack, An $O(n \log n)$ algorithm for computing a link center of a simple polygon, in *STACS 89*, Proc. 6th Ann. Symp. on Theoretical Aspects of Computer Science, Paderborn, FRG, February 1989, eds. B. Monien and R. Cori, Lecture Notes in Computer Science **349**, Springer-Verlag, 1989, pp. 96–107.

[7] C. Ó'Dúnlaing, A tight lower bound for the complexity of path-planning for a disc, *Information Processing Letters* **28** (1988), 165–170.

[8] H. Edelsbrunner, L.J. Guibas, and Micha Sharir, The complexity and construction of many faces in arrangements of lines and of segments, *Discrete and Computational Geometry* **5** (1990), 161–196.

[9] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, Optimal point location in a monotone subdivision, *SIAM J. Computing* **15** (1986), 317–340.

[10] H.A. El Gindy and D. Avis, A linear algorithm for computing the visibility polygon from a point, *Journal of Algorithms* **2** (1981), pp. 186–197.

[11] S.K. Ghosh and D.M. Mount, An output sensitive algorithm for computing visibility graphs, *SIAM J. Computing* **20** (1991), 888–910.

[12] S. Hart and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, *Combinatorica* **6** (1986), 151–177.

[13] B. Joe and R.B. Simpson, Correction to Lee's visibility polygon algorithm, *BIT* **27** (1987), 458–473.

[14] Y. Ke, An efficient algorithm for link distance problems, *Proc. 5th Annual ACM Symposium on Computational Geometry*, 1989, pp. 69–78.

[15] D.T. Lee, Visibility of a simple polygon, *Computer Vision, Graphics, and Image Processing* **22** (1983), pp. 207–221.

[16] W. Lenhart, R. Pollack, J. Sack, M. Sharir, R. Seidel, S. Suri, G. Toussaint, S. Whitesides and C. Yap, Computing the link center of a simple polygon, *Discrete and Computational Geometry* **3** (1988), 281–293.

[17] J.S.B. Mitchell and E. Welzl, Dynamically maintaining a visibility graph under insertions of new obstacles, Manuscript, Cornell University, 1989.

[18] J.S.B. Mitchell, G. Rote, and G. Woeginger, "Minimum-link paths among obstacles in the plane (extended abstract)", *Proc. 6th Annual ACM Symposium on Computational Geometry*, 1990, pp. 63–72.

[19] S. Suri, A linear time algorithm for minimum link paths inside a simple polygon, *Computer Vision, Graphics, and Image Processing* **35** (1986), 99–110.

[20] S. Suri and J. O'Rourke, Worst-case optimal algorithms for constructing visibility polygons with holes, *Proc. 2nd Annual Symposium on Computational Geometry*, 1986, pp. 14–23.

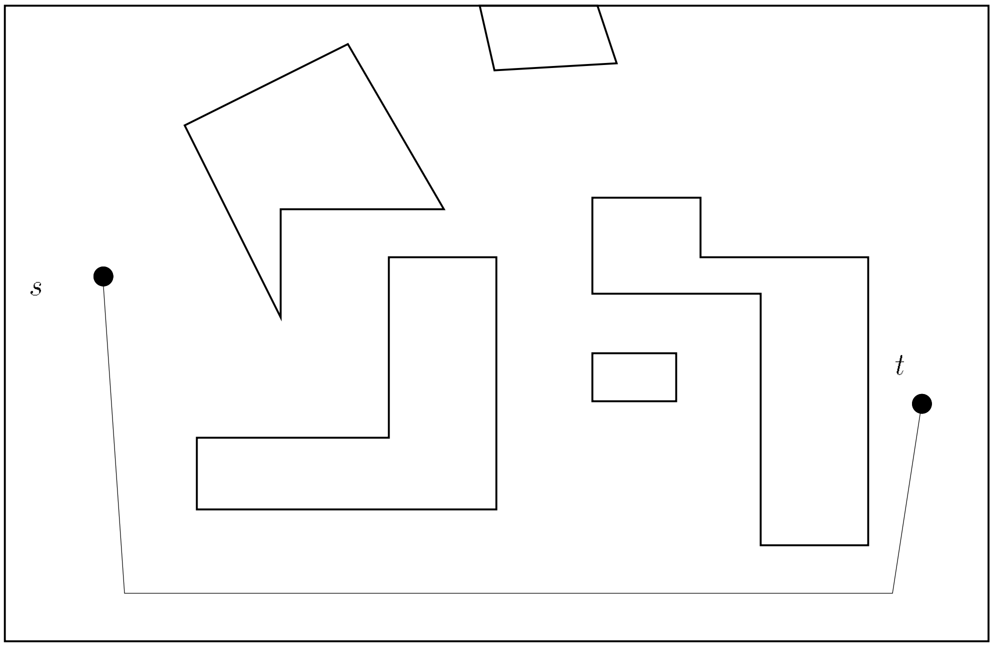

Figure 1: A polygon with holes, and a path from s to t with three links.

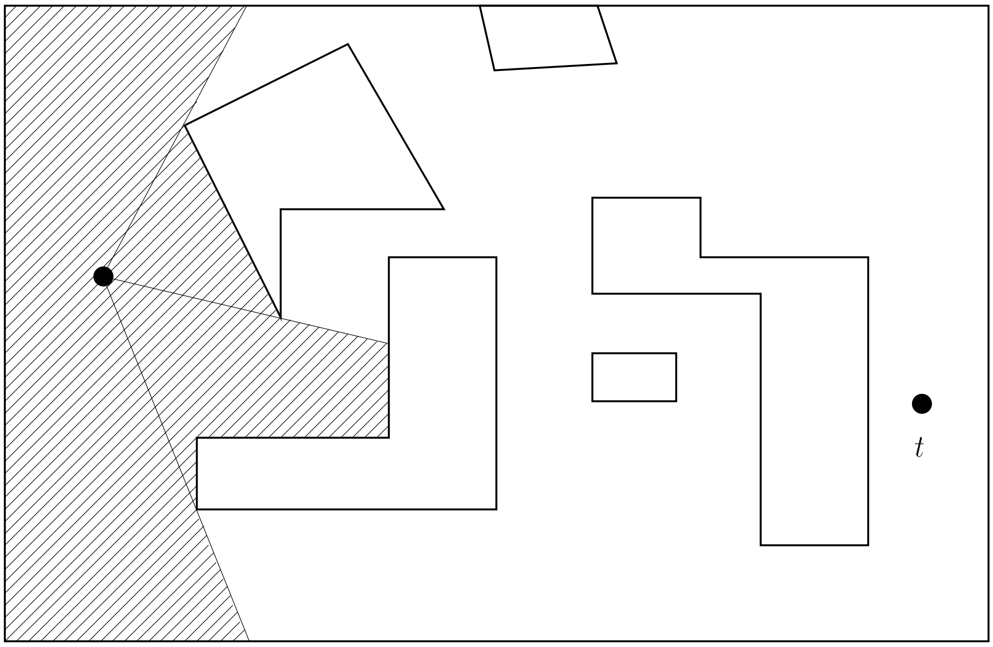

Figure 2: The 1-visibility region VIS_1.

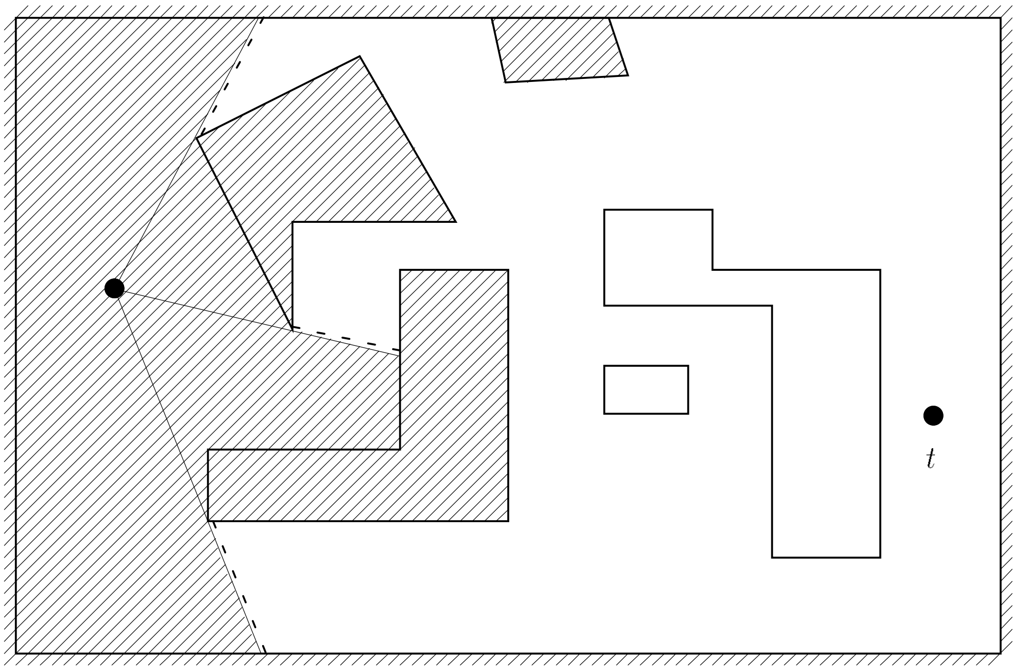

Figure 3: The polygon G_1. Illumination edges are drawn with fat broken lines. (The outside of the room is also included in G_1.)

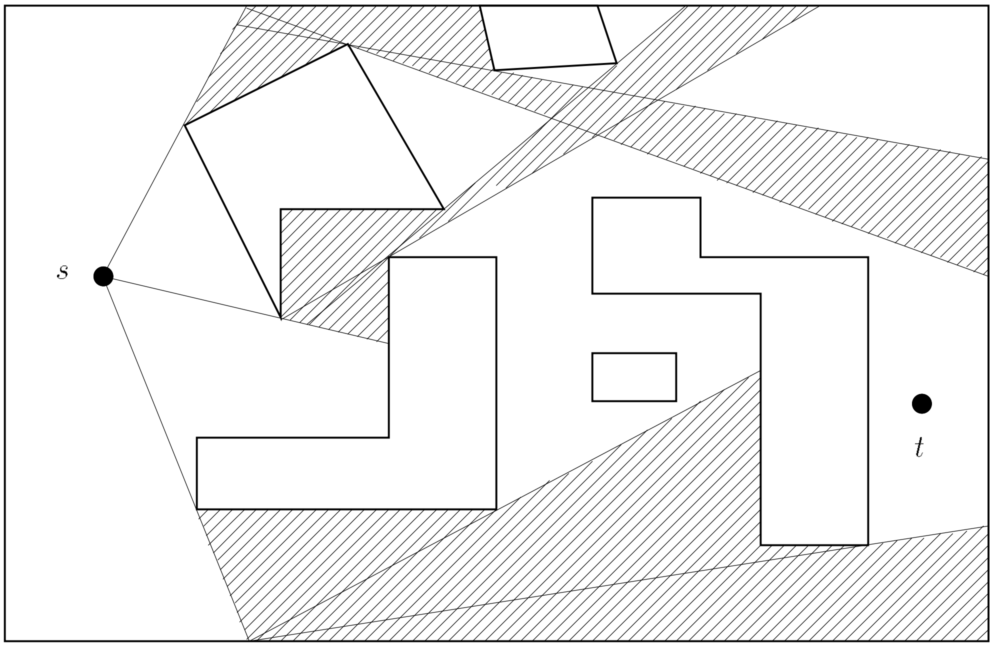

Figure 4: Step 1: Constructing VIS_2. The shaded area is the newly illuminated region $\mathrm{VIS}_2 - \mathrm{VIS}_1$.

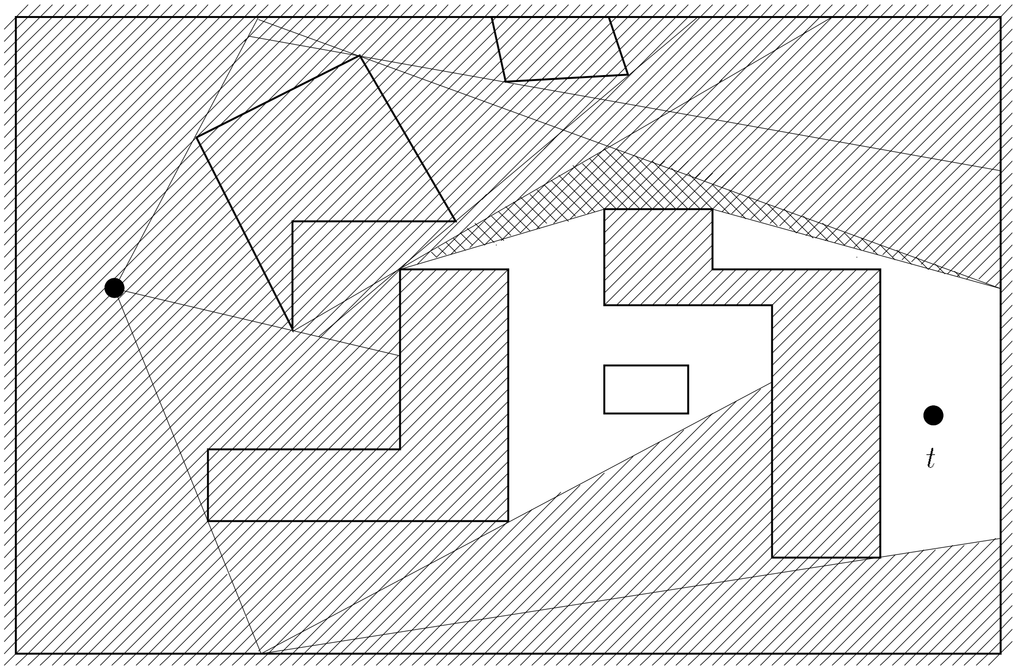

Figure 5: Steps 2 and 3: The set F_2 (hatched area) is the complement of the cell containing t. The relative convex hull $\bar{F}_2$ (hatched area plus cross-hatched area) and F_2 include also the outside of the room.

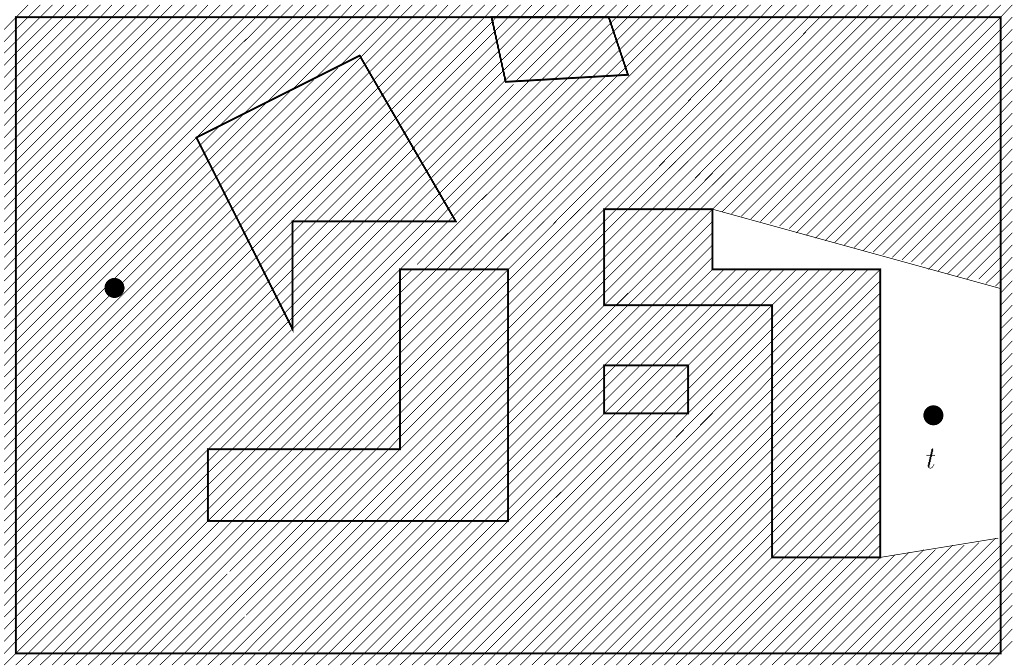

Figure 6: Step 4: The polygon G_2.

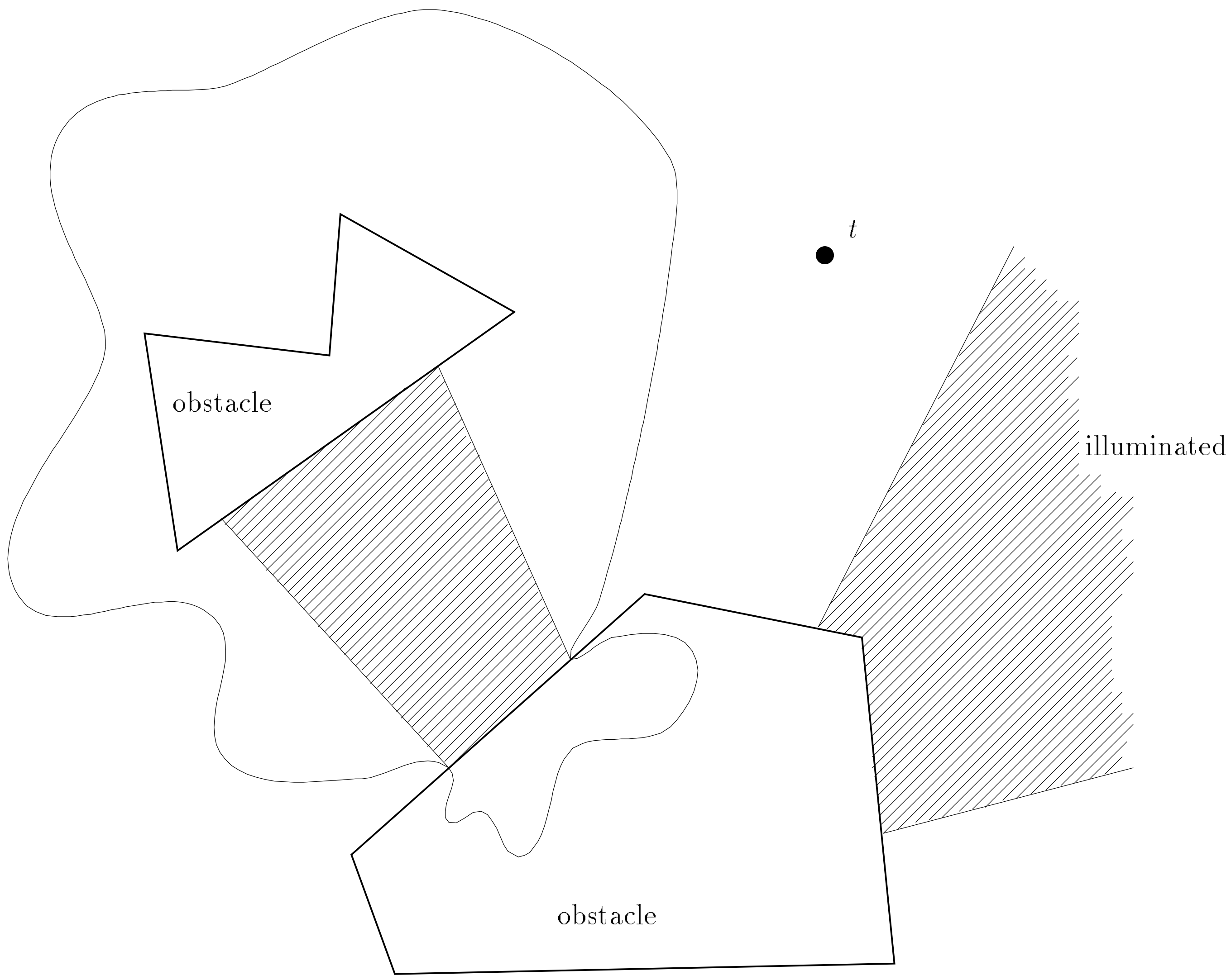

Figure 7: Proof of Lemma 3(i).

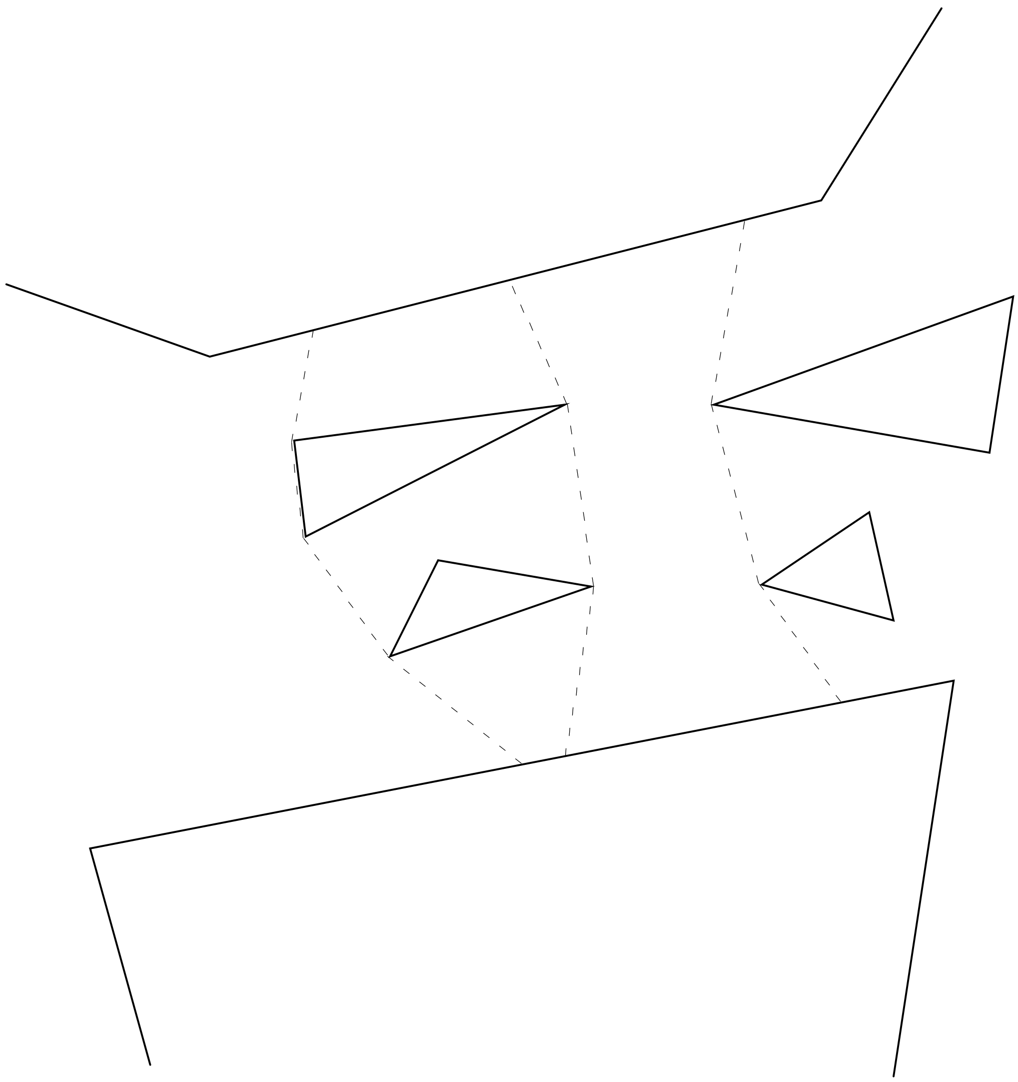

Figure 8: Proof of Lemma 5: There cannot be three illumination chains between the same pair of obstacles.

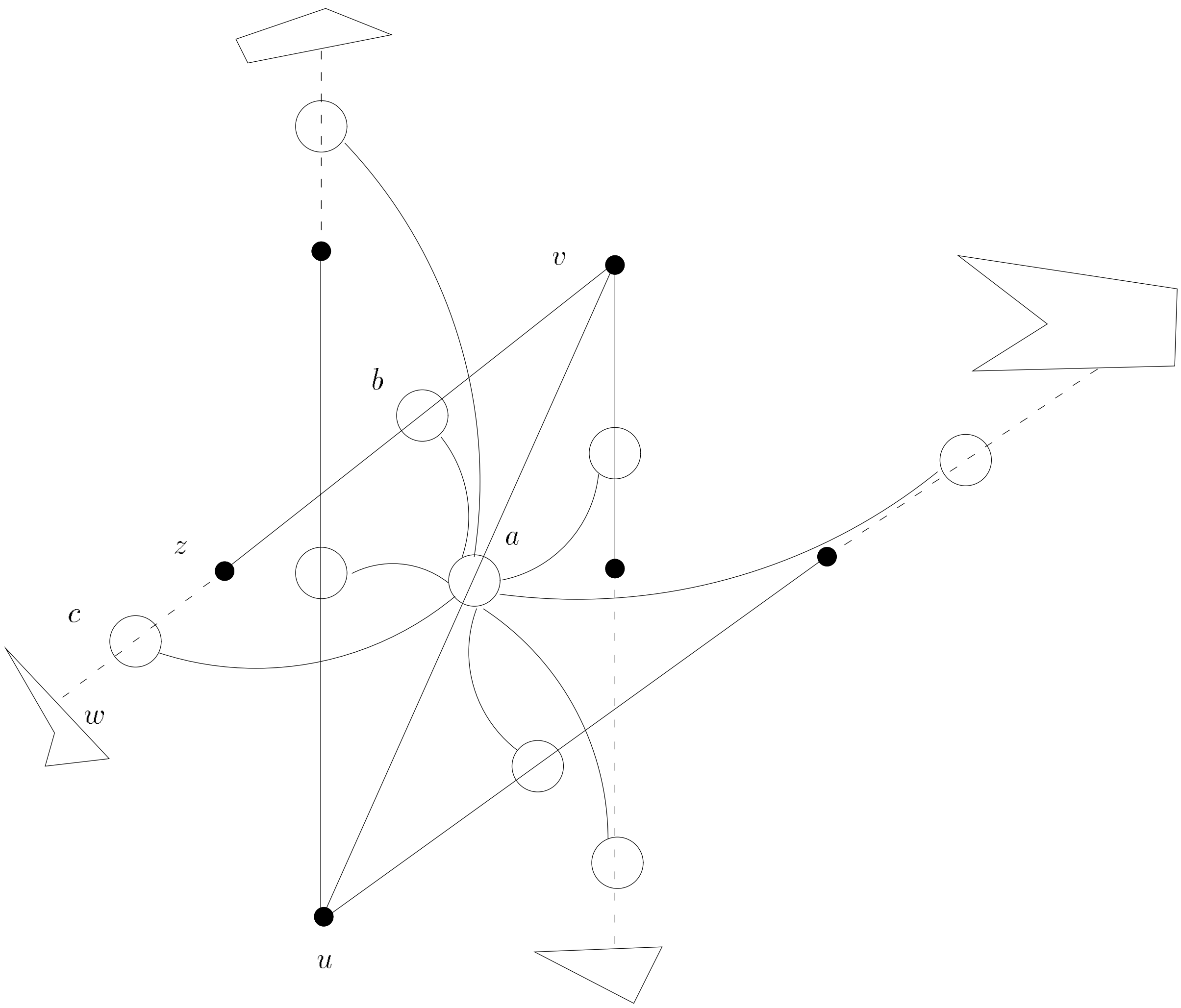

Figure 9: Definition of graph $\mathcal{G}$.

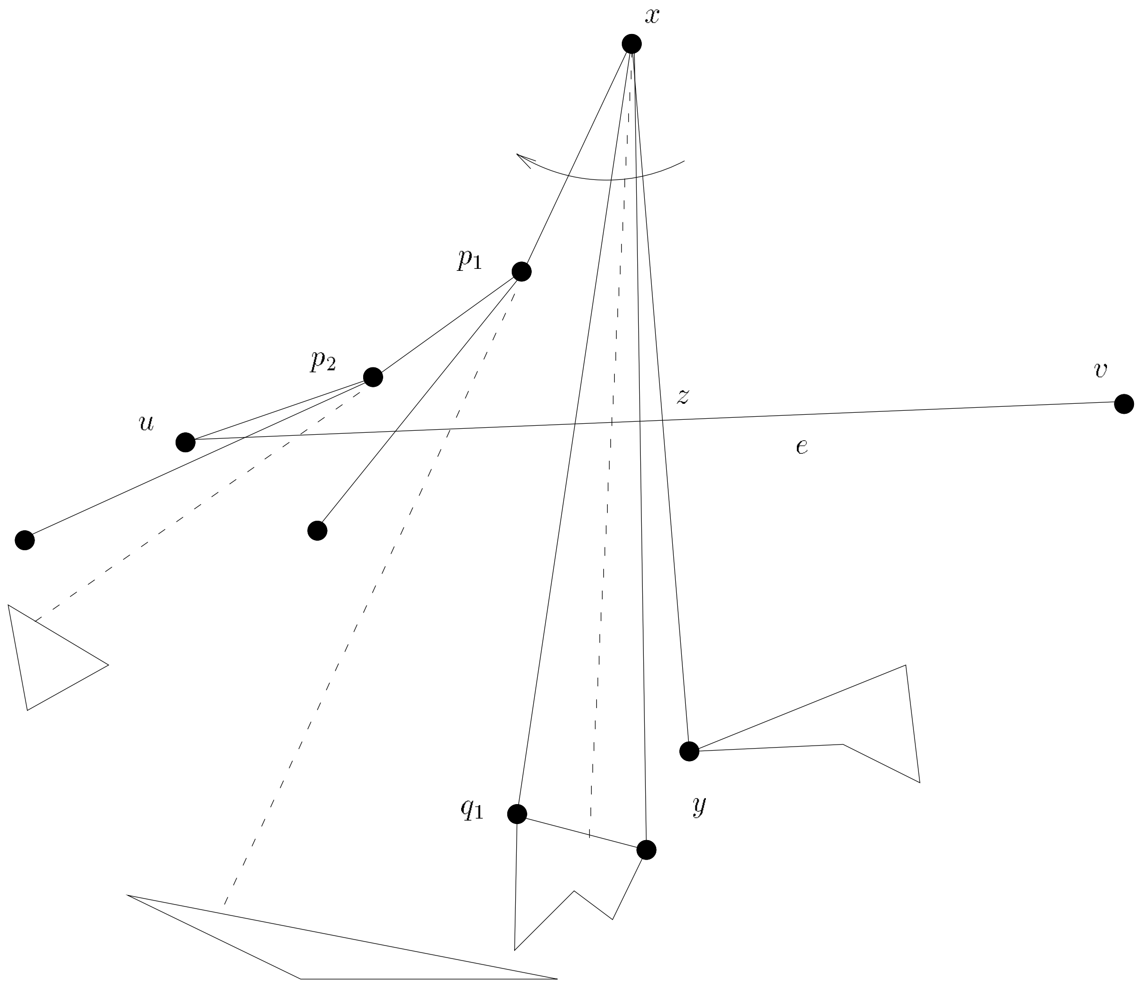

Figure 10: Proving connectivity of the "blue" subgraph.

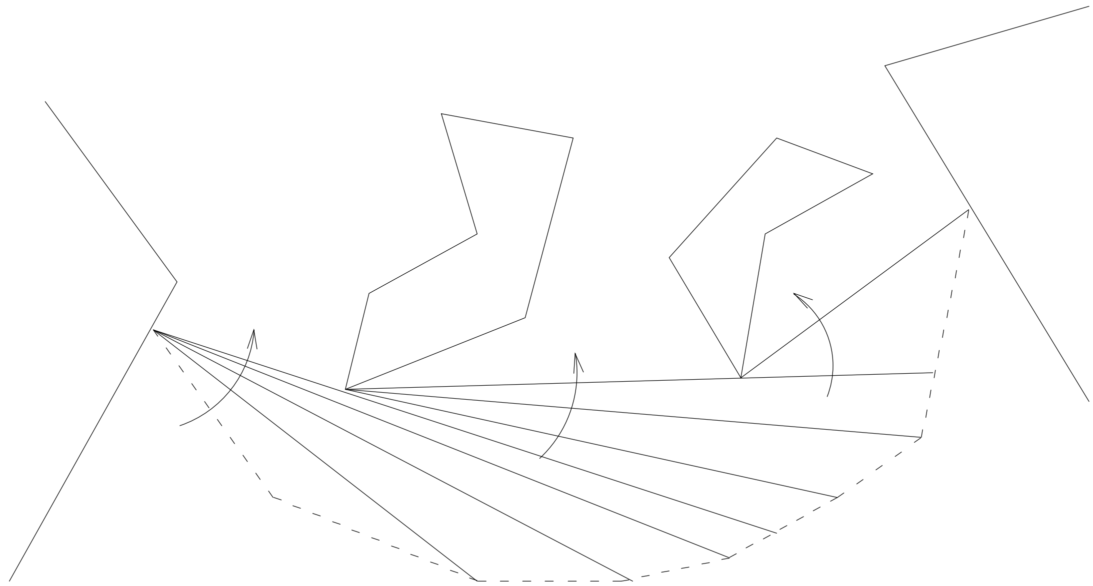

Figure 11: Shortcutting a chain in the construction of the relative convex hull.

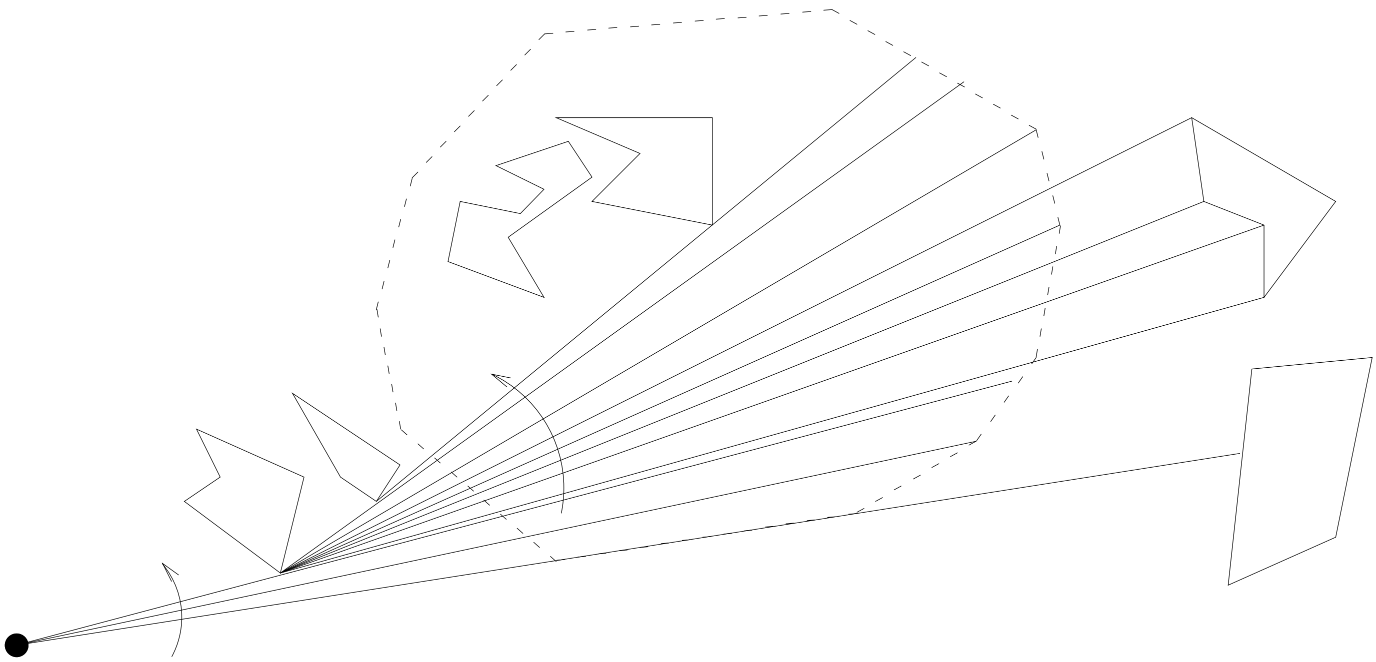

Figure 12: Shortcutting a closed (convex) chain in the construction of the relative convex hull.

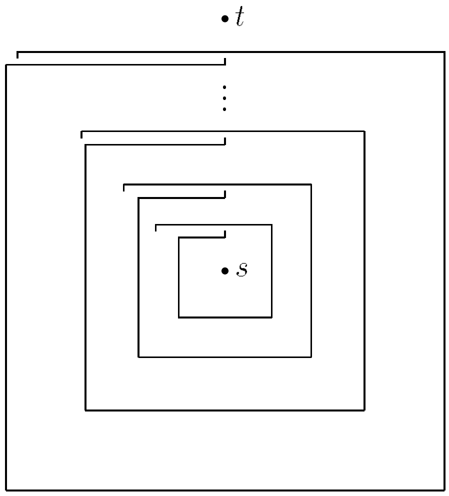

Figure 13: The boxes for the lower bound construction.

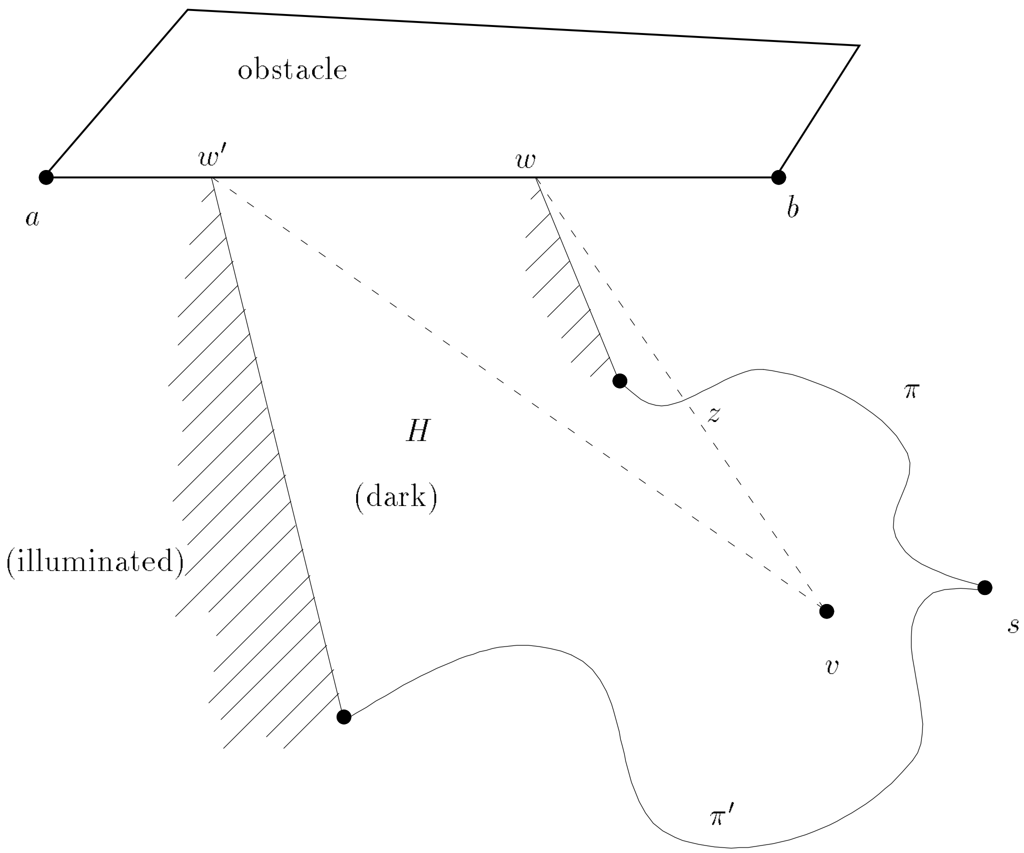

Figure 14: Computing visibilities along edge (a, b).