SHORTEST PATH TO A SEGMENT AND QUICKEST VISIBILITY QUERIES

Esther M. Arkin,[*] *Alon Efrat,*[†] *Christian Knauer,*[‡] *Joseph S. B. Mitchell,*[*]
Valentin Polishchuk,[§] *Günter Rote,*[¶] *Lena Schlipf,*[¶] *and Topi Talvitie*[‖]

Abstract.We show how to preprocess a polygonal domain with a fixed starting point s in order to answer efficiently the following queries: Given a point q, how should one move from s in order to *see* q as soon as possible? This query resembles the well-known shortest-path-to-a-point query, except that the latter asks for the fastest way to *reach* q, instead of seeing it. Our solution methods include a data structure for a different generalization of shortest-path-to-a-point queries, which may be of independent interest: to report efficiently a shortest path from s to a query *segment* in the domain.

1 Introduction

Finding shortest paths is a classical problem in computational geometry, and efficient algorithms are known for computing the paths both in simple polygons and polygonal domains with holes; see [36, 37] for surveys. In the *query version* of the problem one is given a fixed source point s in the domain, and the goal is to preprocess the domain so that the length of a shortest path from s to a query point q can be reported efficiently. The problem is solved by building the *shortest path map* (SPM) from s – the decomposition of the free space into cells such that for all points q within a cell the shortest s-q path is combinatorially the same, i.e., traverses the same sequence of vertices of the domain. Figure 1 shows an example of SPM.

The query in the shortest path problem can be stated as

> Shortest path query: Given a query point q lying in the free space, how should one move, starting from s, in order to **reach** q as soon as possible?

Queries like this arise in surveillance and security, search and rescue, aid and delivery, and various other applications of the shortest path problem. In this paper we introduce and study a related problem that has a very similar query:

[*] *Department of Applied Mathematics and Statistics, Stony Brook University, USA*, estie,jsbm@ams.sunysb.edu

[†] *Computer Science, the University of Arizona, USA*, alon@cs.arizona.edu

[‡] *Institute of Computer Science, Universität Bayreuth, Germany*, christian.knauer@uni-bayreuth.de

[§] *Communications and Transport Systems, ITN, Linköping University, Sweden*, valentin.polishchuk@liu.se

[¶] *Institute of Computer Science, Freie Universität Berlin, Germany*, rote,schlipfg@mi.fu-berlin.de

[‖] *Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Finland*, totalvit@cs.helsinki.fi

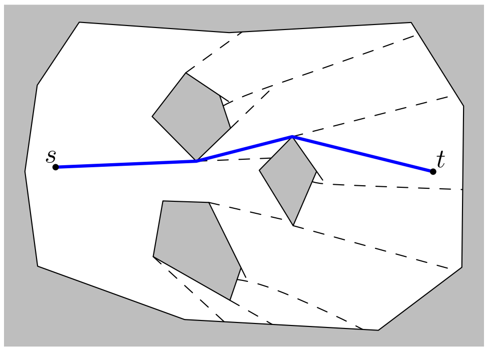

Figure 1: Boundaries between SPM cells are dashed. Shortest path from s to any point in a cell of the map goes through the same vertices of the domain.

> **Quickest visibility query (QVQ)**: *Given a query point q lying in the free space, how should one move, starting from s, in order to* **see** *q as soon as possible?*

Such a query may be natural in applications in which it is important to see (or become seen by) the query point – for inspection purposes, for coming within a shooting range, for establishing communication, etc. In contrast with shortest path queries, such quickest visibility queries have not been studied before, with the single exception of [30] where the problem was considered in simple polygons (in Section 5 we give improved results for this important special case).

The other variant of the shortest path query problem, which we consider in this paper, deals with *segments* instead of points as query objects:

> **Shortest path to a segment query (SPSQ)**: *Given a query* **segment** *ab lying in the free space, how should one move, starting from s, in order to reach ab as soon as possible?*

To our knowledge such queries have not been studied before. We show that in nearly-quadratic time a nearly-quadratic-size data structure can be built to answer SPSQ in poly-logarithmic time (logarithmic-time query can be achieved with nearly-cubic preprocessing time and space). We apply SPSQ as a subroutine in an algorithm for QVQ: given the query point q in an instance of QVQ, build the visibility polygon of q and use SPSQ for each "window" (edge running through the free space) of the polygon to choose the best window through which q can be seen.

1.1 Notation

Let D denote the given polygonal domain; let n, h be the number of vertices and holes of D, respectively. Assume that no two vertices of D have the same x- or y-coordinate. Two points $p, q \in D$ *see* each other if the segment pq fully belongs to the domain (we consider D as a closed set, so that pq may go through a vertex of D or otherwise overlap with the boundary of the domain). Let E be the size of the *visibility graph* of D – the graph on vertices of D with edges between pairs of mutually visible vertices (i.e., pairs of vertices that can be connected with a single link). We also introduce an additional definition related to "3-link visibility" between vertices of D: let Π be the number of pairs of vertices that can be connected by a right-turning 3-link path that makes 90^o turns at both of its bends (refer to Fig. 4).

Let P, S, Q denote the preprocessing time, size of the built data structure and query time, respectively, for an algorithm for answering quickest visibility queries (QVQ) in D. The query point will be generally denoted by q. Let $V(q)$ denote the *visibility polygon* of q (the set of points seen by q); let K denote the complexity (the number of sides) of $V(q)$. We use $\mathsf{P_v}$, $\mathsf{S_v}$, $\mathsf{Q_v}$ to denote the preprocessing time, size of the structure and query time for an algorithm for the problem of building $V(q)$. Finally, we denote by $\mathsf{P_s}$, $\mathsf{S_s}$, $\mathsf{Q_s}$ the corresponding parameters of an algorithm for SPSQ – the problem of reporting length of the shortest path to a query segment lying in D.

Slightly abusing the terminology, we will not differentiate between the two variants of path queries: reporting the *length* of the optimal path and outputting the path itself; similarly to other path query problems, the latter can usually be done straightforwardly (by following back pointers) in additional time proportional to the combinatorial complexity of the path.

1.2 Related work

A shortest path between two points in a simple polygon ($h = 0$) can be found in linear time [8, 32]. The query version (i.e., building the SPM) can be solved within the same time [20]; using the SPM, the length of the (unique) shortest path to a query point can be reported in time $O(\log n)$.

For polygons with holes the *continuous Dijkstra* paradigm [35] leads to an $O(n \log n)$ time algorithm [27] for building the SPM, by propagating a *wave* (which we call the *p-wave*) from s through the free space at unit speed, so that the points reached by the wavefront at any time τ are exactly the points at geodesic distance τ from s (see, e.g., Fig. 6 where gray shows the area covered by the p-wave, and Fig. 12 (left), where the p-wave is blue). At any time during the propagation, the wavefront consists of a sequence of *wavelets* – circular arcs centered on vertices of D, called *generators* of the wavelets; the radius of each arc grows at unit speed. Boundaries between adjacent wavelets trace edges of the SPM (the edges are called *bisectors*, and are further classified in [14] as "walls" and "windows"[1] depending on whether there exist two homotopically distinct shortest paths to points on the bisector); this way the algorithm also builds the SPM which allows one to answer the shortest path queries in $O(\log n)$ time per query. Vertices of the SPM are vertices of D and *triple points*, at which three edges of the map meet (w.l.o.g. four edges of SPM never meet at the same point); the overall complexity of the SPM is linear [27]. Using the continuous Dijkstra method, the quickest way to see a point and the shortest path to a segment (i.e.,

[1]We admit that the term "window" is overused, since it also denotes edges of the visibility polygon $V(q)$. Still, our two different usages of the term are well separated in the text, and are always apparent from the context.

solutions to single-shot, non-query versions of QVQ and SPSQ) can be found in $O(n \log n)$ time by simply declaring $V(q)$ and the segment as obstacles and waiting until the p-wave hits them.

Computing visibility from a point was first studied in simple polygons, for which an $O(n)$-time solution was given already in 1987 [29]. For polygons with holes an optimal $O(n + h \log h)$-time algorithm was presented by Heffernan and Mitchell [24]. The query version of the problem has been well studied too: For simple polygons Guibas, Motwani and Raghavan [21] and Bose, Lubiw and Munro [4] gave algorithms with $\mathsf{P_v} = O(n^3 \log n), \mathsf{S_v} = O(n^3)$ and $\mathsf{Q_v} = O(\log n + K)$; Aronov, Guibas, Teichman and Zhang [3] achieve $\mathsf{P_v} = O(n^2 \log n), \mathsf{S_v} = O(n^2)$ and $\mathsf{Q_v} = O(\log^2 n + K)$. For polygons with holes Zarei and Ghodsi [46] achieve $\mathsf{P_v} = O(n^3 \log n), \mathsf{S_v} = O(n^3), \mathsf{Q_v} = O(K + \min(h, K) \log n)$; Inkulu and Kapoor [28] combine and extend the approaches from [46] and [3] presenting algorithms with several tradeoffs between $\mathsf{P_v}$, $\mathsf{S_v}$ and $\mathsf{Q_v}$, in particular, with $\mathsf{P_v} = O(n^2 \log n), \mathsf{S_v} = O(n^2), \mathsf{Q_v} = O(K \log^2 n)$ (see also [10], as well as [33] giving $\mathsf{P_v} = O(n^2 \log n), \mathsf{S_v} = O(n^2), \mathsf{Q_v} = O(K + \log^2 n + h \log(n/h)))$. A recent paper by Bungiu et al. [5] reports on practical implementation of visibility computation in an upcoming CGAL [7] package.

More generally, both visibility and shortest paths computations are textbook subjects in computational geometry – see, e.g., the respective chapters in the handbook [18] and the books [16, 40]. Visibility meets path planning in a variety of geometric computing tasks. Historically, the first approach to finding shortest paths was based on searching the visibility graph of the domain. Visibility is vital also in computing *minimum-link* paths, i.e., paths with fewest edges [34, 39, 43]. Last but not least, "visibility-driven" route planning is the subject in *watchman route* problems [6, 12, 13, 38, 41] where the goal is to find the shortest path (or a closed loop) from which every point of the domain is seen. Apart from the above-mentioned theoretical considerations, visibility and motion planning are closely coupled in practice: computer vision and robot navigation go hand-in-hand in many courses and real-world applications.

Reporting optimal paths to *non-point* query objects has not received much attention; we are aware of work only for simple polygons. For efficient (logarithmic-time) queries between two convex polygons within a simple polygon, preprocessing can be done in linear time for Euclidean distances [11] and cubic time (and space) for link distance [2, 11].

On the specific problem of quickest visibility queries addressed in this paper, Khosravi and Ghodsi [30] considered QVQs in *simple* polygons. They gave an algorithm for quickest visibility with logarithmic-time queries after quadratic-time preprocessing for building a quadratic-size structure: $\mathsf{P} = O(n^2), \mathsf{S} = O(n^2), \mathsf{Q} = O(\log n)$. We improve the preprocessing and storage to linear, achieving $\mathsf{P} = O(n), \mathsf{S} = O(n), \mathsf{Q} = O(\log n)$ for simple polygons (Section 5).

1.3 Overview of the results

- We start by giving a conditional lower bound connecting P and Q: Section 2 shows that 3SUM on n numbers can be solved in time $O(\mathsf{P} + n\mathsf{Q})$. For instance subquadratic preprocessing time ($\mathsf{P} = o(n^2)$) and sublinear query time ($\mathsf{Q} = o(n)$) would lead to a subquadratic-time algorithm for 3SUM (see [19] for a recent major breakthrough on the 3SUM problem). The lower bound provides us with some justification for not obtaining sub-quadratic preprocessing time P for the QVQ. (Also more broadly, solutions to visibility and/or closely related link-distance query problems often use cubic-time preprocessing [2, 11, 46].)

- Section 3 employs the following natural approach to quickest visibility query.
  (1) Build the visibility polygon $V(q)$ of the query point q; $V(q)$ is a star-shaped polygon any side of which is either a piece of a boundary edge of D, or is a *window* – extension of the segment qv for some vertex v of D.
  (2) For each window find the shortest path from s to the window, and choose the best window to go to.
  The approach leads to an algorithm for QVQ with $\mathsf{P} = \mathsf{P_v} + \mathsf{P_s}, \mathsf{S} = \mathsf{S_v} + \mathsf{S_s}, \mathsf{Q} = \mathsf{Q_v} + K\mathsf{Q_s}$ (refer to Section 1.1 for the notation). Problem (1)—building $V(q)$—has been well studied (refer to Section 1.2 for the known bounds on $\mathsf{P_v}$, $\mathsf{S_v}$ and $\mathsf{Q_v}$). On the contrary, problem (2)—building a shortest path map for *segments*—has not been studied before. In Section 3.2 we give the first results for shortest path to a segment query (which we abbreviated SPSQ above) achieving $\mathsf{P_s} = O(n^3 \log n), \mathsf{S_s} = O(n^3 \log n), \mathsf{Q_s} = O(\log n)$. Our solution is based on first designing a data structure for *horizontal* segments (Section 3.1) with $\mathsf{P_s} = O(n \log n), \mathsf{S_s} = O(n \log n), \mathsf{Q_s} = O(\log n)$ – a result which may be interesting in its own right. The data structure for SPSQ for arbitrary segments is then built straightforwardly since there are $O(n^2)$ combinatorially different orientations: the data structure for arbitrarily oriented segments is thus just an $O(n^2)$-fold replication of the structure for horizontal ones (we also give bounds in terms of sizes, E and Π, of visibility structures in D). Alternatively, in Section 3.3 we give an algorithm with $\mathsf{P_s} = O(n^2 2^{\alpha(n)} \log n), \mathsf{S_s} = O(n^2 2^{\alpha(n)} \log n), \mathsf{Q_s} = O(\log^2 n)$ where $\alpha(n)$ is the inverse Ackermann function, based on storing "snapshots" of the p-wave propagation in the continuous Dijkstra. (In the conference version of this paper [1], we erroneously claimed $\mathsf{P_s} = O(n^2 \log n), \mathsf{S_s} = O(n^2 \log n)$ for this algorithm.)

- In Section 4 we introduce the full *Quickest Visibility Map* (QVM) – the decomposition of D into cells such that within each cell the quickest visibility query has combinatorially the same answer: the shortest path to see any point q within a cell traverses the same sequence of vertices of D and goes to the same window of $V(q)$. Our algorithm for building the map has $\mathsf{P} = O(n^8 \log n), \mathsf{S} = O(n^7), \mathsf{Q} = O(\log n)$. We also observe that the QVM has $\Omega(n^4)$ complexity.

- In Section 5 we consider the case when D is a simple polygon. We give linear-size data structures that can be constructed in linear time, for answering QVQs and SPSQs in logarithmic time: $\mathsf{P} = O(n)$, $\mathsf{S} = O(n)$, $\mathsf{Q} = O(\log n)$, $\mathsf{P_s} = O(n)$, $\mathsf{S_s} = O(n)$, $\mathsf{Q_s} = O(\log n)$.[2]

We invite the reader to play with our applet [45] demonstrating QVM.

[2]Some results from this section were reported in EuroCG [31].

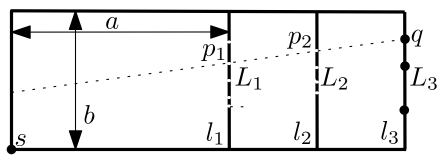

Figure 2: D is long: $a \gg b$. The ray qp_2 (dotted) can reach all the way to the left, provided there exists a gap (p_1) on l_1 collinear with q and p_2.

2 A lower bound

In the *3SUM* problem the input is a set of numbers and the goal is to determine whether there are three numbers whose sum is 0. We connect P and Q (see Section 1.1 for the notation) with the 3SUM problem.

Theorem 1. *A 3SUM instance of size n can be solved in $O(\mathsf{P} + n\mathsf{Q})$ time.*

Proof. We use a construction similar to the one in the proof of 3SUM-hardness of finding minimum-link paths [39]. Start from an instance of the GeomBase problem: Given a set $S = L_1 \cup L_2 \cup L_3$ of n points lying on 3 vertical lines l_1, l_2, l_3 respectively, do there exist collinear points $p_1 \in L_1$, $p_2 \in L_2$, $p_3 \in L_3$? It was shown in [15] that solving GeomBase is as hard as solving 3SUM with n numbers. Construct the domain D for quickest visibility queries as follows (Fig. 2): The lines l_1, l_2, l_3 are obstacles; turn each point from $L_1 \cup L_2$ into a gap punched in the obstacle. Squish vertically the whole construction, i.e., make the distances between the lines much larger than the vertical extent of S; this way all the rays p_2p_1 with $p_2 \in L_2$, $p_1 \in L_1$ are confined to a narrow beam. Put the whole construction in a long box so that the beam shines onto its left side. Put s in the lower left corner of the box.

Now do quickest visibility queries to points in L_3. If some point $q \in L_3$ is collinear with some points $p_1 \in L_1$, $p_2 \in L_2$, then q can be seen by traveling at most b from s; otherwise, one needs to travel at least a to L_1. Thus by making at most n queries we can solve the GeomBase. □

The above proof can be extended in several ways. E.g., since a can be arbitrarily large in comparison with b, even approximate answers to queries would solve the 3SUM problem.

3 Querying shortest paths to windows

The quickest way to see the query point q from s is the quickest way to reach (the boundary of) $V(q)$, or equivalently, to reach a window of $V(q)$. Assuming the visibility polygon of q had been built by existing methods (see Section 1.2), answering QVQ boils down to determining the window closest to s. We do not have a better way of accomplishing this than to do shortest path queries to each window in succession, which leads to the problem of building a data structure to answer efficiently shortest-path-to-a-segment query (abbreviated SPSQ above) – the subject of this section.[3]

3.1 Horizontal segments

In this subsection we present a data structure for SPSQ for fixed-orientation (w.l.o.g. horizontal) segments; in the next subsection we extend the structure to handle arbitrary segments (and in Section 3.3 we present a structure for arbitrary segments, based on different techniques). The shortest path to a segment ab touches it at a, at b, or in the interior; we will focus on shortest paths to the interior, since shortest paths to a or b are answered with the SPM. Such a path follows the shortest path to some vertex v of D (recall that s is also treated as a vertex) and then uses the perpendicular from v onto ab; i.e., the last link of the path is vertical. We describe our data structure only for the case of paths arriving at ab from above, for which this last link is going *down*; an analogous structure is built for the paths arriving to the query from below.

The data structure is the horizontal trapezoidation of D augmented with some extra information for each trapezoid T; specifically – the set of vertices that see the trapezoid from above (i.e., vertices from which downward rays intersect T). Of course, the information is not stored explicitly with each trapezoid (for this may require $\Omega(n)$ information in each of $\Omega(n)$ trapezoids); instead, the information is stored in persistent balanced binary trees. The vertices in the trees are sorted by x-coordinate. To enable $O(\log n)$-time range minimum queries, each internal node stores the minimum of $d(v) + v_y$ values over all vertices v in the subtree of the node, where $d(v)$ is the geodesic distance from s to v (which can be read from the SPM) and v_y is the y-coordinate of v. Knowing the minimum of these values over the range of a segment is our ultimate goal, because the length of the shortest path that arrives to the segment at ordinate y with last link dropped from v is $d(v) + v_y - y$.

We build the trees as follows. Let $\prec$ be the "aboveness" relation on the trapezoids (i.e., $T \prec T'$ if and only if T' is incident to T from above). We traverse the trapezoids using a topological order of the DAG for $\prec$ (e.g., in the order of the y-coordinates of trapezoid top sides) and compute the trees, from top to bottom, for the trapezoids as follows (Fig. 3): If a trapezoid T does not have a successor in $\prec$, then T is a triangle (due to the non-degeneracy assumption on D), and the tree $\tau(T)$ for T simply stores the top vertex of T if the downward ray from the vertex goes inside T; if the ray does not enter T (i.e., T has an obtuse angle at the base), then $\tau(T)$ is empty. If T has successors, then for each trapezoid T' that succeeds T in $\prec$, we take a persistent copy of the tree $\tau(T')$ and remove from it all vertices that do not see the boundary $T \cap T'$ between the trapezoids (the removal is a split operation on the copy). After the removal has been done for all successors of T, we merge the copies of the trees into the tree $\tau(T)$. Additionally, if T has a vertex of D on its top edge, then the vertex is added to $\tau(T)$.

To answer SPSQ, find the trapezoid T containing the query segment ab (recall our

[3]We do not know how to take advantage of the fact that windows are quite special – maximal free-space segments anchored at vertices of D. On one hand this makes our solution more general, as it applies to arbitrary segments; on the other hand, it leaves open the possibility of designing a more efficient algorithm tailored to the special case of windows.

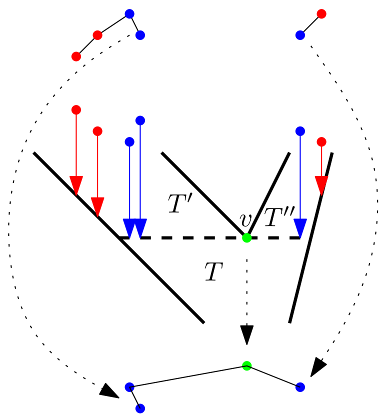

Figure 3: Trees for the trapezoids. Red vertices are removed from persistent copies of $\tau(T')$ and $\tau(T'')$; the other vertices (blue) remain in the copies. Then the copies are merged to form $\tau(T)$. Finally, v is added to $\tau(T)$.

assumption that ab lies in the free space, and hence – in a single trapezoid) and choose the right history snapshot. Then perform the range minimum query $[a, b]$ to obtain the vertex $v \in \tau(T)$ of D with the smallest $d(v) + v_y$ (since $v \in \tau(T)$, the vertex sees ab when looking down and $a \le v_x \le b, v_y \ge y$); this will be the vertex from which the interior of the segment is reached in the quickest way. The shortest path via v is compared with the shortest paths to a and b, altogether in $O(\log n)$ query time. Thus our data structure provides $\mathsf{P_s} = O(n \log n), \mathsf{S_s} = O(n \log n), \mathsf{Q_s} = O(\log n)$ for horizontal segments.

Theorem 2. *A data structure of size $O(n \log n)$ can be built in $O(n \log n)$ time, such that the shortest path length from a given source s to a horizontal query segment can be reported in $O(\log n)$ time.*

3.2 Arbitrary segments

To support all directions of query segments, we build our structure from previous subsection for all rotations of D at which the data structure changes. The data structure changes at three types of events: (1) when two visible vertices get the same x-coordinates, (2) when two visible vertices get the same y-coordinates, and (3) when some query segment can be reached equally fast from two vertices, i.e., when the two vertices get the same $d(v) + v_y$ values (Fig. 4). The number of the first two events is bounded by the size E of the visibility graph of D, and the number of the third-type events is bounded by the number Π of pairs of vertices that can be connected by a right-turning 3-link path that turns by 90 degrees at its both bends. Thus we need to replicate our data structure only $O(E + \Pi)$ times (which may be much smaller than the naive upper bound of $O(n^2)$; note, however, that Π can be $\Omega(n^2)$ even when $E = O(n)$ – see, e.g., Fig. 5).

To find the rotation angles for the first two types of events, we precompute the visibility graph of D (takes $O(E + n \log n)$ time [17]). We can discover the third-type events "on-the-fly", while actually rotating the domain. For that we make our trees "kinetic" by

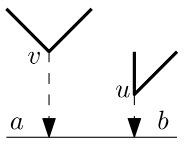

Figure 4: u and v can be connected by a 3-link path making only right turns. ab is seen by u and v, and $d(u)+u_y = d(v)+v_y$.

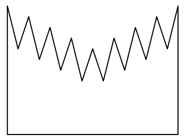

Figure 5: $E = O(n)$ but $\Pi = \Theta(n^2)$.

assigning to each internal node u of the trees the "expiration time" (rotation angle) when the vertex with lowest value of $d(v) + v_y$ in the subtree of u changes; the time for u can be computed when u is constructed, using the lowest $d(v) + v_y$ values in the subtrees of children of u. Computing the expiration time is done once per node instance of the trees.

Overall we obtain $\mathsf{P_s} = O((E + \Pi)n \log n), \mathsf{S_s} = O((E + \Pi)n \log n), \mathsf{Q_s} = O(\log n)$.

Theorem 3. *A data structure of size $O((E + \Pi)n \log n)$ can be built in $O((E + \Pi)n \log n)$ time, such that the shortest path length from a given source s to a query segment can be reported in $O(\log n)$ time.*

Remark. We could reuse the information between the rotations and get a persistent data structure with $\mathsf{P_s} = O(n^2 \log^3 n), \mathsf{S_s} = O(n^2 \log^3 n), \mathsf{Q_s} = O(\log^2 n)$, but this is inferior to the performance of our data structure in the next section. Potentially one could also get a persistent data structure with $\mathsf{P_s} = \mathsf{S_s} = O((E + \Pi)\text{polylog } n), \mathsf{Q_s} = O(\text{polylog } n)$; we, however, were not able to do this.

3.3 Continuous Dijkstra-based algorithm

We now give another data structure for SPSQ, based on storing "snapshots" of p-wave propagation (recall that p-wave is the wave propagated during the continuous Dijkstra algorithm for building the SPM) at times when a wavelet appears or disappears from the wavefront (this includes events when a wavelet is split into two after hitting an obstacle). Say that time t_i is *critical* if the wavefront changes combinatorially at t_i. As shown in [27], there are $O(n)$ critical times. For each critical time t_i we store the *geodesic disk* D_i of radius t_i, i.e., the set of points in D whose geodesic distance to s is at most t_i; the disk is an $O(n)$-complexity region bounded by circular arcs (wavelets) and straight-line segments (obstacle edges). We construct data structures for two types of queries: "Given a segment ab, lying in the free space, does it intersect D_i?" and "Given a segment ab lying outside D_i, where will the segment hit the disk if dragged perpendicularly to itself?".

3.3.1 Determining i

Assume that D_i has been preprocessed for point location, to test in $O(\log n)$ time whether a or b is inside D_i (in which case, obviously ab intersects D_i). To answer the intersection query when neither a nor b lies inside D_i, we look at the complement, C_i, of D_i in D; obviously, a segment intersects the nonobstacle boundary of D_i if and only if it intersects

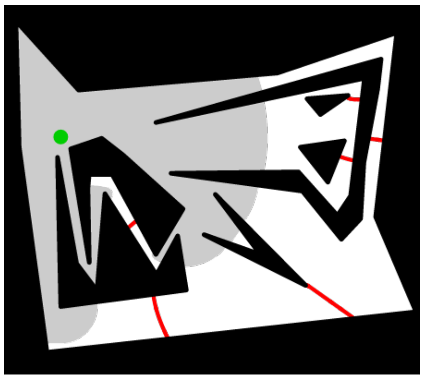
Figure 6: s is green, D_i is gray, and C_i (the part of free space not reached by the wave) is white; it has four connected components, one of which has two holes inside it. Red curves are the walls (bisectors with more than one homotopy type of the shortest path) of the SPM.

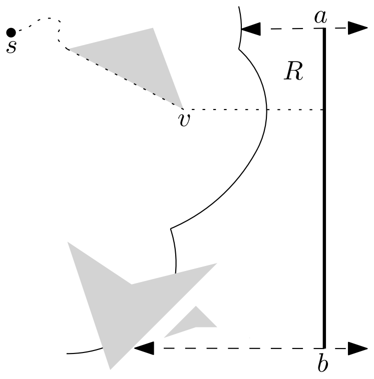

Figure 7: D_i is bounded by circular-arc wavelets (solid curves) and edges of obstacles (gray); the rays orthogonal to ab are dashed. The shortest path to ab ends with the perpendicular from v onto ab (dotted).

the (nonobstacle) boundary of C_i. The set C_i may have several connected components (Fig. 6), at most one of which surrounds D_i (by *surrounds* we mean that any path from D_i to infinity must intersect the component). Each connected component C of C_i is preprocessed separately as follows: Let $\mathcal{H}$ be the set of holes lying inside C. Let $\hat{C} = C \cup \bigcup_{H \in \mathcal{H}} H$ be C together with the holes $\mathcal{H}$; the set $\hat{C}$ either has no holes (i.e., is simply connected) or has one hole (D_i, if C is the component that surrounds D_i). In any case $\hat{C}$ can be preprocessed in $O(|C| \log n)$ time to answer ray shooting queries in $O(\log n)$ time [9], where $|C|$ is the complexity of C (the geodesic triangulations framework of [9] extends to regions with circular arcs on the boundary). To answer the intersection query we first determine the connected component C_a of C_i that contains a (assume that all connected components have been preprocessed for point location) and use the ray shooting data structure on $\hat{C}_a$ to determine where the ray r from a through b exits $\hat{C}_a$; ab intersects D_i if and only if r exits into D_i and does so before b. Note that here we crucially use the assumption that the query segment lies in the free space: we do not care if r intersects holes on the way to D_i.

With the above data structures built for all disks D_i, we can do binary search on the critical times to determine the index i such that the query segment ab intersects D_{i+1} but does not intersect D_i, which means that ab is reached by the wavefront at some time between t_i and t_{i+1}. We spend $O(\log n)$ time for ray shooting per choice of i, yielding $O(\log^2 n)$ time overall to determine i. Now the goal is to determine which wavelet of D_i hit ab first.

3.3.2 Determining the wavelet

Using the point location data structure on C_i we find the component C of C_i that contains ab (the segment must fully lie inside a single connected component, for otherwise it intersects

D_i). Next, using the ray shooting data structure on C, we shoot rays within C, with sources at a and at b, firing orthogonal to ab, in both directions. This yields one region on each side of ab, and we consider the two regions separately; let R be the region on one of the sides (Fig. 7).

The boundary of R consists of ab, a ray shot from a to the boundary of C, a portion of the (outer) boundary of C (which may include circular-arc wavelets alternating with sequences of straight-line segments on the boundary of obstacles), then a ray shot from b. Within R, we translate ab parallel to itself to discover the first wavelet on the boundary of R that is hit – the generator v of the wavelet is the last vertex on the shortest path to ab, with the last link of the path being the perpendicular dropped from v onto ab. This can be done by computing and storing convex hulls of pairs of consecutive wavelets on the boundary of C, pairs of pairs, pairs of pairs of pairs, etc., up to the convex hull of the whole component C. The next paragraph gives the details on building the convex hulls, and the subsequent paragraphs describe how to account for the possibility that the dragged segment hits an obstacle before hitting a wavelet.

In this paragraph we imagine that the dragged query segment ab can freely move over obstacles (and we want to discover which wavelet on the boundary of R will be encountered first as ab is moved perpendicularly to itself, possibly going over obstacles). Assume that the wavelets on the boundary of C are numbered in the order as they appear on the boundary. Compute convex hulls of wavelets 1 and 2, of wavelets 3 and 4, wavelets 5 and 6, etc.; then compute convex hulls of wavelets 1 through 4, wavelets 5 through 8, etc.; ...; finally, compute convex hull of all the wavelets (Fig. 8). We thus obtain a hierarchy of convex hulls. Each convex hull of this hierarchy can be built by drawing bitangents to wavelets on the corresponding convex hulls of the preceding level, in $O(\log n)$ time per bitangent; since the complexity of each level is $O(|C|)$ and there are $O(\log n)$ levels, the whole hierarchy, for all connected components of C_i, can be stored in $O(n \log n)$ space and computed in $O(n \log^2 n)$ time. We preprocess each convex hull to answer extreme-wavelet queries—"Which wavelet is first hit by a query line moving in from infinity parallel to itself towards the convex hull?"– in $O(\log n)$ time (such preprocessing involves simply storing the bitangents to the consecutive wavelets along the convex hull in a search tree, sorted by the slope). Now, the rays shot from a and b (the ones that define the region R) hit the boundary of D_i at two wavelets, whose numbers are, say, w_1 and w_2. The interval $[w_1, w_2]$ is covered by $O(\log n)$ *canonical* intervals, for which we precomputed and stored the convex hulls; by doing the extreme-wavelet query in each of the intervals we determine the first wavelet between w_1 and w_2 hit by the sliding ab in overall $O(\log^2 n)$ time.

Of course it may happen that letting the dragged segment ab pass over obstacles leads to a wrong answer to the query: the first hit wavelet may actually be "hidden" behind an obstacle (Fig. 9). To fix this, we start from turning the wavelets into "constant-aperture" arcs as follows. Let c be an endpoint of a wavelet w centered on a vertex v; we trim or expand w depending on whether c belongs to an obstacle or a bisector in the SPM (Fig. 10):

- If c is on an obstacle edge, then the aperture of w can only shrink as the wavefront expands. Let c' be the position of c at the next critical time. We pretend that

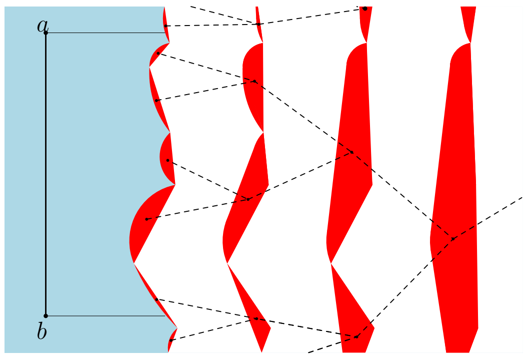

Figure 8: Wavelets on a wavefront and the hierarchy of their convex hulls (red). Also shown is a query segment ab and the rays shot from its endpoints towards the wavefront (of course, the hierarchy is created during the preprocessing, i.e., *prior to* knowing ab). Lightblue is the area that has not been reached by the p-wave.

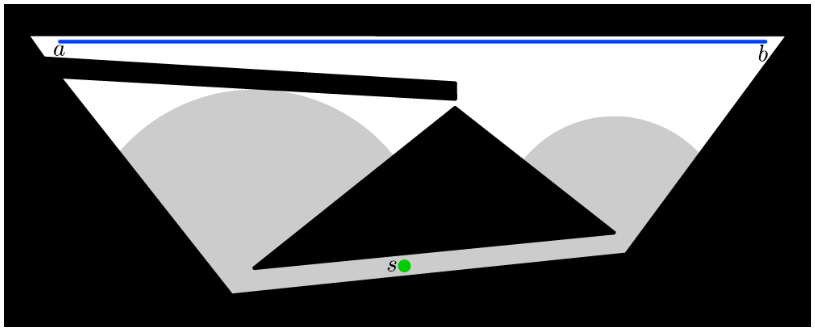

Figure 9: Obstacles are black, D_i is gray. Since ab can move over obstacles, it will hit the leftmost wavelet before the rightmost wavelet – the latter giving the correct answer to the SPSQ.

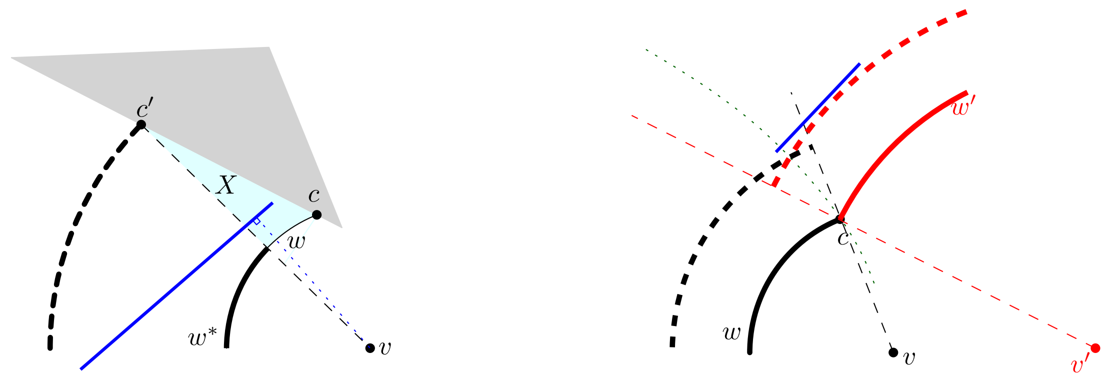

Figure 10: Left: w^* (bold)—the trimmed w (solid)—does not sweep over X (lightgreen); still, detecting when (if ever) the p-wave hits the query segment (blue) in X is easy: if the wavefront does hit the segment in X, then an endpoint of the segment must lie in the SPM cell of v. The thick dashed arc shows where w^* will be at the next critical time. Right: w^*—the widened w—reaches onto the other side of the bisector (dotted darkgreen) between SPM cells of v and v', where it may erroneously hit the query segment (blue); still, w'^*—the widened w'—will hit the segment earlier. The thick dashed arcs show where w^* and w'^* will be at some future moment before the next critical time.

as the wavefront expands, c slides along the ray vc' (instead of sliding along the obstacle edge). This erroneously kills the part of w in the wedge $c'vc$ (and deprives w of sweeping the wedge), but this is no problem because if it is this part of w that actually hits ab, then either a or b (or both) belongs to the wedge (otherwise ab would have intersected either the obstacle or w), and so v must be the root of the SPM cell containing the endpoint of ab; hence we can discover that w hits ab simply by drawing the shortest segment from v to ab (which may be either the perpendicular from v onto ab or the segment from v to ab's endpoint) – all this in $O(\log n)$ time. See Fig. 10, left.

- If c is on a bisector between wavelets w and w', we pretend that as the wavefront expands, c slides along the ray vc (instead of sliding along the bisector). This erroneously keeps alive the part of w on the other side of the bisector, and we might erroneously discover that w hits the query segment first, but this mistake will be corrected when we discover that w' hits ab even earlier. See Fig. 10, right.

We thus transform each wavelet w into a *modified arc* w^*; as the wavefront expands (at unit speed) between the consecutive critical times, the radius of the arc increases at the unit speed (just as the radius of w did), but the aperture of w^* stays constant (unlike the aperture of w, which could change due to w's endpoints sliding along bisector(s) and/or obstacle(s)). Our goal now is to discover which modified arc will be the first that the dragged segment will hit *properly*, i.e., so that the hit happens in the interior of the segment and the interior of the arc (with the segment tangent to the arc).

Note that, thanks to switching from wavelets to modified arcs (and proper hits), we no longer have to worry about ab hitting a wavelet hidden behind an obstacle – (even

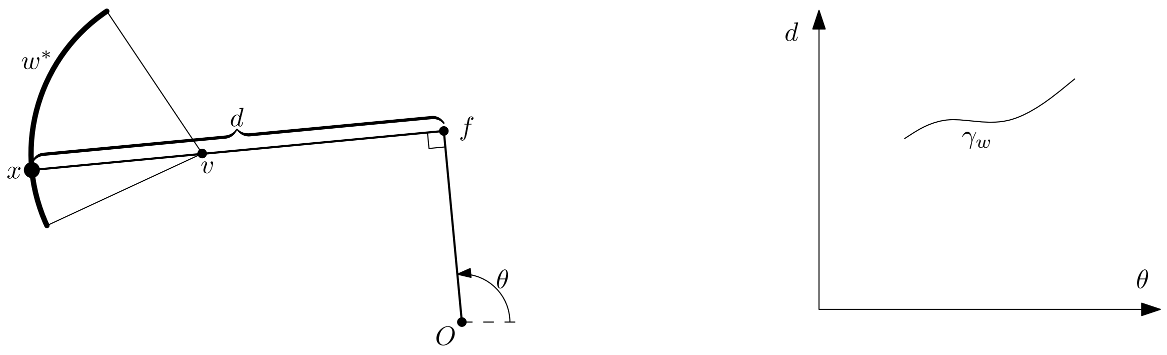

Figure 11: Any point x on w^* defines θ and d; the modified arc is thus represented in the (θ, d)-space by the curve $\gamma_w = (\theta(x), d(x))$ for $x \in w^*$.

though the dragged segment might hit an obstacle before hitting an arc properly, like, e.g., in Fig. 9) if the dragged ab properly hits an arc w^* at a point p^*, then the ray vp^* reaches the (original, non-dragged) ab without encountering an obstacle; indeed, otherwise, i.e., if the ray were to intersect an obstacle at a point p before reaching ab, there would have been another critical time when the p-wave engulfs p. (This property was, in fact, our reason for introducing the constant-aperture arcs: with a changing-aperture wavelet w, the dragged ab could have hit w at a point p such that the ray vp would hit an obstacle before reaching the original location of ab; see, e.g., Fig. 9.) Also note that if we did not care about *proper hits only*, we could have discovered the first hit modified arc with the help of the arcs' convex hulls hierarchy, just as we did with the wavelets; however, the dragged segment might hit a convex hull "improperly", at a "corner" (endpoint) of an arc. Therefore we need to work out a data structure capable of "turning off" the "incorrect" directions of the dragged segment, i.e., capable of taking into account that an arc w^* can be properly hit only by those segments whose direction belongs to the aperture of w^*. Below we describe such a data structure, supporting efficient discovery of the first modified arc properly hit by the dragged query segment.

We view each modified arc w^* as a θ-monotone curve in the (θ, d)-plane where θ is a direction and d is the "radial" distance. Specifically, we arbitrarily choose the origin $O \in \mathbb{R}^2$. Let v be the generator of w^* (inherited from w) and let x be a point on w^*; let f be the foot of the perpendicular dropped from O onto the ray xv. Then $\theta = \theta(x)$ is the angle between the horizontal direction and Of, and $d = d(x) = |xf|$ (Fig. 11). As x moves along w^*, the point $(\theta(x), d(x))$ traces the curve γ_w in the (θ, d)-space, corresponding to w^* (note that we index the curve with the original wavelet w, not with w^*).

Now, how does the curve γ_w change as the wavefront expands until the next critical time? For any fixed θ, the distance d increases at unit speed (assuming the wavefront expands at unit speed), implying that γ_w moves up, in the (θ, d)-plane, at the unit speed. Therefore, if the query segment ab has direction θ, then the first modified arc hit by the translated ab is the arc w^* whose curve γ_w lies on the upper envelope, at the abscissa θ, of the curves $\gamma_{w_1}, \ldots, \gamma_{w_2}$ corresponding to the wavelets $w_1, \ldots, w_2$ that bound the region R (refer to Fig. 7). To find w^* efficiently, we create a hierarchy of sets of modified arcs similarly to the hierarchy of convex hulls (refer to Fig. 8); this time, for each set of the arcs we compute the upper envelope of their representative curves in the (θ, d)-space. Because any two curves intersect at most twice, the complexity of the upper envelope of $O(n)$ arcs is $O(\lambda_4(n))$ and it can be built in $O(\lambda_3(n)\log n)$ time [25], where $\lambda_s(n)$ is the maximum length of an n-element order-s Davenport-Schinzel sequence [42]. In fact, we can build the full hierarchy in overall $O(\lambda_4(n)\log n)$ time: we simply build the structure for each node in the hierarchy by merging the structures for the child nodes. After the hierarchy has been built, we can discover the first modified arc hit by the dragged query segment in $O(\log^2 n)$ time, using the hierarchy similarly to that of the wavelets' convex hulls: the interval $[w_1, w_2]$ is covered by $O(\log n)$ *canonical* intervals for which the upper envelopes are built, and finding whose curve is on the upper envelope at θ takes $O(\log n)$ time per envelope.

Remark. Note that one could skip building the convex hulls hierarchy, and proceed directly to computing the envelopes (the envelopes alone are sufficient to answer the closest-wavelet queries). We nevertheless kept the description of the convex hulls since they are more tangible than the envelopes and serve as a natural first step before explaining the more complicated envelopes structure.

3.3.3 Putting everything together

Our data structure achieves $\mathsf{P_s} = O(n\lambda_4(n)\log n), \mathsf{S_s} = O(n\lambda_4(n)\log n), \mathsf{Q_s} = O(\log^2 n)$: For each of the $O(n)$ critical times, the ray shooting data structures require $O(n \log n)$ preprocessing time and storage (Section 3.3.1), and the upper envelopes hierarchy require $O(\lambda_4(n)\log n)$ preprocessing time and $O(\lambda_4(n)\log n) = O(n2^{\alpha(n)}\log n)$ storage (Section 3.3.2). A query involves finding the relevant D_i ($O(\log^2 n)$ time, Section 3.3.1) and then finding the first wavelet hit by the sliding ab (also $O(\log^2 n)$, Section 3.3.2).

Theorem 4. *A data structure of size $O(n^2 2^{\alpha(n)}\log n)$ can be built in $O(n^2 2^{\alpha(n)}\log n)$ time, such that the shortest path length from a given source s to a query segment can be reported in $O(\log^2 n)$ time.*

3.4 Quickest visibility queries

Applying a data structure for SPSQ to QVQ, we obtain a solution for the latter with $\mathsf{P} = \mathsf{P_v} + \mathsf{P_s}, \mathsf{S} = \mathsf{S_v} + \mathsf{S_s}, \mathsf{Q} = \mathsf{Q_v} + K\mathsf{Q_s}$. For instance, using [28] (which provides $\mathsf{P_v} = O(n^2\log n), \mathsf{S_v} = O(n^2), \mathsf{Q_v} = O(K\log^2 n)$) and the structure from Section 3.3, we obtain $\mathsf{P} = O(n^2 2^{\alpha(n)}\log n), \mathsf{S} = O(n^2 2^{\alpha(n)}\log n), \mathsf{Q} = O(K\log^2 n)$. See Section 1.2 for other bounds on $\mathsf{P_v}, \mathsf{S_v}, \mathsf{Q_v}$.

Theorem 5. *A data structure of size $O(n^2 2^{\alpha(n)}\log n)$ can be built in $O(n^2 2^{\alpha(n)}\log n)$ time, such that the shortest path length from a given source s to see a query point q can be reported in $O(K\log^2 n)$ time, where K is the complexity of the visibility polygon $V(q)$ of q.*

4 Quickest visibility map

Assuming the SPM has been built, the quickest way to see a query point q becomes evident as soon as the following information is specified: the window W of $V(q)$ through which to see q and the vertex g of D that is the last vertex on the shortest path to W. Let r be the vertex of D that defines W (i.e., W is part of the ray from q through r); we say that r is the *root* and g is the *generator* for q. We define the *quickest visibility map* (QVM) as the decomposition of D into cells such that all points within a cell have the same root and generator. That is, within a cell of QVM the answer to QVQ is combinatorially the same: draw the ray from q through the root r and drop the shortest segment from the generator g onto the window (this segment may be perpendicular to the window, or the segment to a window endpoint). In this section we describe an algorithm to build QVM. After the map is preprocessed for point location, QVQs can be answered in $O(\log n)$ time just by identifying the cell containing the query.

Reusing the idea of continuous Dijkstra algorithm for constructing the SPM we propagate "visibility wave" (v-wave) from s (Fig. 12, left). Similarly to the geodesic disk (the set of points that can be *reached* from s, by a certain time, starting from s and moving with unit speed), we define the *visibility disk of radius* t as the set of points that can be *seen* before time t by an observer starting from s and moving with unit speed. The ball is bounded by extensions of tangents from vertices of D to circles centered at vertices of the domain; intersections between tangents trace *bisectors* of QVM – a point q on a bisector can be seen equally fast by going to more than one side of $V(q)$ (Fig. 12, right).

To bound the complexity of QVM, we first introduce some notation. Let r, g be the root-generator pair for some cell of QVM. Let T be the line through r tangent to the wavelet centered at g at some time during the p- and v-waves propagation; let l be the point of contact of T with the wavelet. The part of the ray lr after r running through the free space (if such part exists) is called a *sweeper* – as the wavelet radius grows, T rotates around r and (parts of) the sweeper claim the cell(s) of QVM that have (r, g) as the root-generator pair. We call the segment rl the *leg* of the sweeper, and the segment gl (the radius of the wavelet) its *foot* (refer to Fig. 12, right).

Our argument below benefits from the assumption that all angles of the obstacles in D are larger than 90^o; to satisfy the assumption we can (symbolically) file the domain by replacing each acute vertex with a short edge (see the *corner arc algorithm* [22, Ch. 4] for similar ideas). The reason to make the assumption is that the speed of rotation of a sweeper depends on the (inverse of) the length of its leg; in particular, if the length is 0, the sweeper rotates at infinite speed, leading to a discontinuity in v-wave propagation.[4] The filing ensures that the v-wave propagation is continuous, which implies that QVM features (vertices and edges) are due only to intersections of sweepers, or (dis)appearance of sweepers, or possible sweeper extension/contraction as it passes over a vertex of D.

Consider now the subdivision S of D into maximal regions such that for any point inside a region, the set of sweepers that pass over the point is the same (i.e., if $\aleph(p)$ denotes the set of sweepers that ever pass over p, then S is the subdivision into the regions where $\aleph$

[4]See http://www.cs.helsinki.fi/group/compgeom/qvm/infinitespeed.gif for an animation.

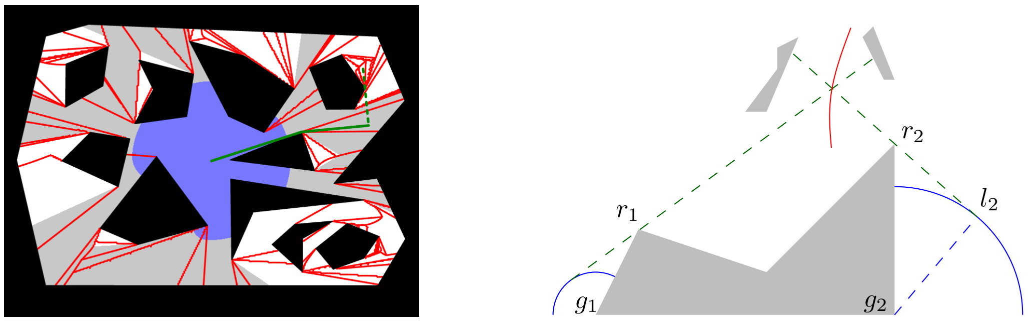

Figure 12: Left: The v-wave is gray, the p-wave is blue (s is in the center of the rectangle). Red curves are bisectors in the QVM. Solid green shows the shortest path to see a query point; the path ends with a perpendicular dropped from D's vertex (the generator) onto the ray (dashed green) from the query point through another vertex of D (the root). Right: Gray is an obstacle. As p-wave propagates, the geodesic disk grows by expanding the wavelets (blue arcs) at unit speed (wavelets are centered at generators g_1, g_2 and their radii grow at unit speed). Wavelets growth rotates tangents (dashed green) to the wavelets dropped from vertices r_1, r_2 – roots of the QVM cells. The tangents define "shadows" – the boundaries of the visibility disk; the tangents intersection traces the bisector (red) in the QVM. The QVM cell to the left of the bisector has (r_1, g_1) as the root-generator pair, while the cell on the right has (r_2, g_2) as the pair; points on the bisector have both (r_1, g_1) and (r_2, g_2), and can be seen equally fast using paths via g_1 and via g_2. g_2l_2 is the foot of the sweeper hinged at r_2; l_2r_2 is its leg.

stays the same); the QVM complexity equals to the number of vertices inside the regions of S plus on the edges of S. The vertices of QVM in the interiors of the regions are the *triple points* where three sweepers (and three bisectors) meet; since a sweeper is defined by two vertices of D (the root and the generator), there are $O(n^6)$ triple points.

What remains is to bound the number of vertices of QVM that lie on the edges of S; to do that we define a superset $\bar{S}$ of the edges. Specifically, disappearance of a sweeper may be due to one of the three events (Fig. 13): sweeper becoming aligned with an edge of D incident to the sweeper's root, the leg's rotation becoming blocked, or the foot's rotation becoming blocked; appearance of a sweeper is due to the reverse of the events. To account for the first-type events we add the supporting lines of edges of D to $\bar{S}$. The second-type events happen on supporting lines of edges of the visibility graph of D; we add the lines to $\bar{S}$. Third-type events happen on lines through vertices of D perpendicular to supporting lines of the visibility graph edges; we add these perpendicular lines to $\bar{S}$. Finally, extension/contraction of a sweeper happens along the extension of the visibility graph edge. Overall $\bar{S}$ consists of $O(nE)$ lines, and all $O(n^2E^2)$ of their intersections could potentially be vertices of QVM. The only remaining vertices of QVM are intersections of bisectors with the lines in $\bar{S}$ (all the other vertices are in the interior of the cells of S); since any bisector is defined by 4 vertices of D (2 root-generator pairs for the sweepers defining the bisectors) there are $O(n^4)$ bisectors. Thus, the total number of vertices of QVM on edges of $\bar{S}$ (and hence on the edges of S) is $O(n^2E^2 + n^4En)$.

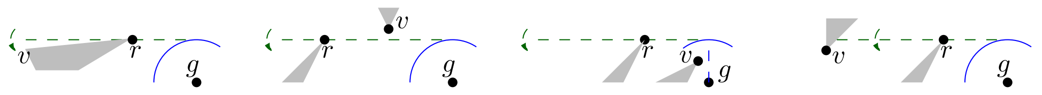

Figure 13: From left to right: Sweeper aligns with rv; leg gets blocked by v; foot gets blocked by v; sweeper extends at v.

The overall complexity of QVM (the number of vertices inside the regions of S plus on the edges of S) is thus $O(n^6 + n^2E^2 + n^5E) = O(n^7)$. The above description leads to an algorithm to compute the potential $O(n^7)$ QVM vertices by brute force; for each of them we can check in $O(n \log n)$ time whether it is indeed a vertex of QVM (see Section 1.2). We then sweep the plane to restore the QVM edges: from each vertex, extend the bisector until it hits another vertex. Putting point location data structure on top of QVM, we obtain $\mathsf{P} = O(n^8 \log n), \mathsf{S} = O(n^7), \mathsf{Q} = O(\log n)$.

Theorem 6. *A data structure of size $O(n^7)$ can be built in $O(n^8 \log n)$ time, such that the shortest path length from a given source s to see a query point q can be reported in $O(\log n)$ time.*

We note that any algorithm for QVM must have $\mathsf{P} = \Omega(n^4), \mathsf{S} = \Omega(n^4)$ because it may need to store explicitly the region weakly visible from a segment, which may have $\Theta(n^4)$ complexity [44].

5 Simple polygons

We now present an optimal ($\mathsf{P_s} = O(n), \mathsf{S_s} = O(n), \mathsf{Q_s} = O(\log n)$) algorithm for SPSQs for the case when D is a simple polygon ($h = 0$); together with the shortest path map of D and a data structure for ray shooting queries (both can be built in $O(n)$ time to support $O(\log n)$-time queries), it leads to an optimal algorithm ($\mathsf{P} = O(n), \mathsf{S} = O(n), \mathsf{Q} = O(\log n)$) for QVQs as well. We start by introducing additional notation for this section.

Assume that the vertices of D are stored in an array $\vec{D}$ sorted in clockwise order along the boundary of D. For points $x, y \in D$, let $\pi(x, y)$ denote the shortest path between x and y; the shortest path from s to a point y is denoted simply by $\pi(y)$. Let the *predecessor* $\text{pred}(y)$ of y be the last vertex of D on $\pi(y)$ before y (or s if y sees s); the predecessor of any point can be read off the shortest path map (SPM) of D in $O(\log n)$ time. Let SPT be the shortest path tree from s in D; the tree is the union of paths $\pi(v)$ for all vertices v of D. Assume that the SPT is preprocessed to support lowest common ancestor (LCA) queries in constant time [23].

Let ab be the query segment. Let r be the last common vertex of the shortest paths $\pi(a), \pi(b)$ from s to the endpoints of the segment; r can be determined from SPM and SPT in $O(\log n)$ time: either $\text{pred}(a) = \text{pred}(b) = r$, or $r = LCA(a, b)$ (Fig. 14, left). The paths $\pi(r, a)$ and $\pi(r, b)$ together with the segment ab form the *funnel* F of ab; the vertex r is the *apex* of F.

Let $a = v_0, v_1, \dots, r = v_m, \dots, v_k, v_{k+1} = b$ be the vertices of the funnel from a to b. Note that the paths $\pi(r, a)$ and $\pi(r, b)$ are outward convex; in particular, F can be

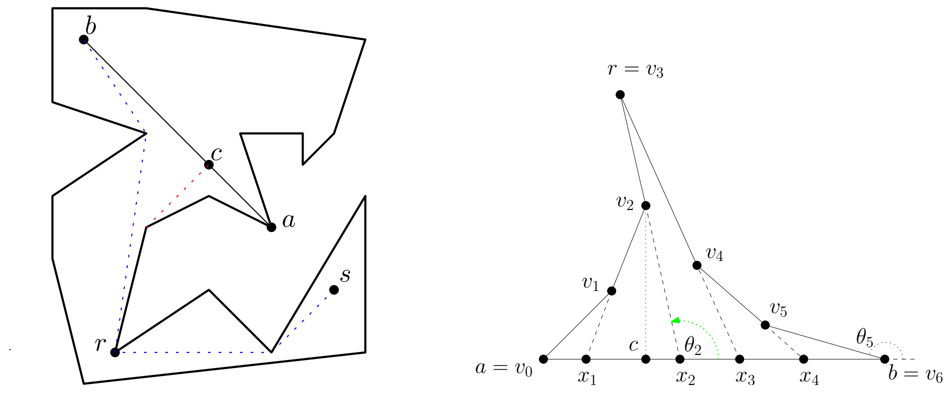

Figure 14: Left: $r = LCA(a, b)$; $\pi(c)$ is the answer to the query. Right: c is the foot of the perpendicular dropped from v_2 to ab.

decomposed into triangles by extending the edges of F until they intersect ab (Fig. 14, right). Let x_i denote the intersection point of the extension of the edge $v_i v_{i+1}$ with ab (in particular, $x_0 = a$ and $x_k = b$). The shortest path from s to points on the segment $x_i x_{i+1}$ passes through v_{i+1} as the last vertex of D: $\forall p \in x_i x_{i+1}, \text{pred}(p) = v_{i+1}$.

Let $\theta_0, \theta_1, \ldots, \theta_k$ denote the angles between the extension edges and ab: $\theta_0 = \angle bav_1$, $\theta_i = \angle bx_iv_i$ for $0 < i < k$ and $\theta_k = \pi - \angle abv_k$. The outward convexity of the paths $\pi(r, a), \pi(r, b)$ implies that the sequence $\theta_0, \theta_1, \ldots, \theta_k$ is increasing. As a consequence the point $c \in ab$ closest to s can be characterized as follows [30]: c is the foot of the perpendicular from v_{i+1} to ab for i such that $\theta_i < \pi/2$ and $\theta_{i+1} \geq \pi/2$. Thus c can be found by a binary search on the angles θ_i: if $\theta_i > \pi/2$ then c lies left of x_i, whereas if $\theta_i < \pi/2$ then c lies right of x_i. We now describe how to implement the search in $O(\log n)$ time.

First, if $\theta_0 > \pi/2$ then $c = a$, and if $\theta_k < \pi/2$ then $c = b$; in both cases we are done. Next, look at the extensions of the edges emanating from the apex $r = v_m$ of the funnel. If $\theta_{m-1} \leq \pi/2 < \theta_m$, c is the foot of the perpendicular from v_m to ab and we are done.

It remains to show what to do if $\theta_{m-1} > \pi/2$ (the case $\theta_{m-1} < \pi/2$ is symmetric). In this case $\theta_i > \pi/2$ for $m \leq i \leq k$ since the angle sequence is increasing; in particular c is the foot of the perpendicular from some vertex v_i to ab, where v_i is an interior vertex of the left side $\pi(r, a)$ of the funnel F, i.e., $1 \leq i < m$. To determine v_i we would like to perform a binary search on the sequence $v_1, \ldots, v_{m-1}$; however this sequence is not directly accessible (we do not compute it during the query since it can have $\Omega(n)$ size). We therefore use the array $\vec{D}$, and perform a binary search on the interval $[v_1, v_{m-1}]$ in $\vec{D}$.

For a vertex u in this interval we find the vertex $LCA(u, v_1)$, which is one of the vertices $v_1, \ldots, v_{m-1}$ on the left edge of the funnel, say v_j. By computing the angle θ_j we can decide if the binary search has to continue to the left or to the right of u. After $O(\log n)$ iterations the binary search is narrowed down to an interval between two successive vertices in $\vec{D}$. This implies that the point v_i from which the perpendicular to c has to be dropped is also determined. (Note that for several successive vertices u_l in $[v_1, v_{m-1}]$ we can get the same vertex v_j as a result of computing $LCA(u_l, v_1)$; still, since the total number of vertices in $[v_1, v_{m-1}]$ is $O(n)$, after $O(\log n)$ iterations the binary search is narrowed down to an interval between two successive vertices in $\vec{D}$.)

Theorem 7. *For a simple polygon with n vertices, a data structure of size $O(n)$ can be built in $O(n)$ time, such that the shortest path length from a given source s to a query segment can be reported in $O(\log n)$ time.*

5.1 Quickest visibility queries

In a simple polygon, s is separated from q by a unique window of $V(q)$ (unless s and q see each other, which can be tested in $O(\log n)$ time by ray shooting). Since the last edge of the shortest path $\pi(q)$ is a straight-line segment, one of the window endpoints is $a = \text{pred}(q)$; the endpoint can be read off the SPM of D in $O(\log n)$ time. To find the other endpoint b of the window, shoot the ray qa until it intersects the boundary of D; this also takes $O(\log n)$ time using the data structure for ray shooting [26]. Once we have the window ab, our data structure described above finds the (unique) shortest path to the window in additional $O(\log n)$ time.

Theorem 8. *For a simple polygon with n vertices, a data structure of size $O(n)$ can be built in $O(n)$ time, such that the shortest path length from a given source s to see a query point q can be reported in $O(\log n)$ time.*

6 Conclusion

We considered Quickest Visibility Queries (QVQ) and Shortest Path to a Segment Queries (SPSQ) – sisters of the well studied shortest path queries. In QVQ the goal is to preprocess a polygonal domain with a given starting point s so as to report efficiently the quickest way to see a query point q starting from s, or in other words, to report the fastest way to reach (a side of) the visibility polygon $V(q)$ of q from s; SPSQ arises naturally in the latter interpretation of QVQ, as it gives the fastest way to reach a side of $V(q)$.

We gave upper and lower bounds on the complexity of the problems. Many open questions remain; the main one is prompted by the (polynomial but) high complexity of the full Quickest Visibility Map (QVM): How much time and space are needed to get, say, sublinear-time answer in a QVQ? For instance, can one avoid querying *all* sides of $V(q)$? Or perhaps there is a more efficient way to query shortest path to segments when the segments are the windows of $V(q)$? It is also possible that our continuous Dijkstra based solution can be improved by a factor of roughly n using persistent data structures (we thank Pankaj Agarwal for this observation).

Another interesting direction for future work is obtaining improved bounds for *approximate* answers to the queries. For instance, instead of storing the $\Omega(n)$ geodesic disks for the critical times in the continuous-Dijkstra wavefront propagation, one could store only the geodesic disks whose radii form a geometric progression with denominator $1+\varepsilon$ and compute the overlay of the (weak) visibility regions of the disks; then a $(1+\varepsilon)$-approximate answer to QVQ could be found by determining the cell of the overlay containing the query point q. Such an approach, however, requires a lower bound on the answer, which we do not see how to obtain (the bound is needed to determine the first term of the progression).

A technical assumption used in our results for SPSQ is that the query segment must lie in the free space; the assumption is used crucially in several places in Section 3. Lifting the assumption, i.e., giving efficient solution for SPSQ for segments that may intersect obstacles is open (the question is whether the problem can be solved more efficiently than by treating each connected component of the segment individually and taking the minimum).

Last but not least, in applications QVQs are interesting also in other environments, such as, e.g., polyhedral terrains.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. E. Arkin and J. Mitchell acknowledge support from the US-Israel Binational Science Foundation (grant 2010074) and the National Science Foundation (CCF-1018388, CCF-1526406). VP is supported by grant 2014-03476 from the Sweden's innovation agency VINNOVA. TT was supported by the University of Helsinki Research Funds.

References

[1] E. M. Arkin, A. Efrat, C. Knauer, J. S. B. Mitchell, V. Polishchuk, G. Rote, L. Schlipf, and T. Talvitie. Shortest path to a segment and quickest visibility queries. In L. Arge and J. Pach, editors, *31st International Symposium on Computational Geometry, SoCG 2015, June 22-25, 2015, Eindhoven, The Netherlands*, volume 34 of *LIPIcs*, pages 658–673. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[2] E. M. Arkin, J. S. B. Mitchell, and S. Suri. Optimal link path queries in a simple polygon. In *Proc. 3rd Ann. ACM-SIAM Symp. Discrete Algorithms (SODA'92)*, pages 269–279, 1992.

[3] B. Aronov, L. J. Guibas, M. Teichmann, and L. Zhang. Visibility queries and maintenance in simple polygons. *Discrete & Computational Geometry*, 27(4):461–483, 2002.

[4] P. Bose, A. Lubiw, and J. I. Munro. Efficient visibility queries in simple polygons. *Comput. Geom. Theory Appl.*, 23(3):313–335, Nov. 2002.

[5] F. Bungiu, M. Hemmer, J. Hershberger, K. Huang, and A. Kröller. Efficient computation of visibility polygons. In *30th Europ. Workshop on Comput. Geom. (EuroCG'14)*, 2014.

[6] S. Carlsson, H. Jonsson, and B. J. Nilsson. Finding the shortest watchman route in a simple polygon. *Discrete & Computational Geometry*, 22(3):377–402, 1999.

[7] CGAL. Computational Geometry Algorithms Library. http://www.cgal.org.

[8] B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. Sympos. Found. Comput. Sci. (FOCS'82)*, pages 339–349. IEEE, 1982.

[9] B. Chazelle, H. Edelsbrunner, M. Grigni, L. J. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, 1994.

[10] D. Z. Chen and H. Wang. Visibility and ray shooting queries in polygonal domains. *Computational Geometry*, 48(2):31 – 41, 2015.

[11] Y.-J. Chiang and R. Tamassia. Optimal shortest path and minimum-link path queries between two convex polygons inside a simple polygonal obstacle. *Int. J. Comput. Geometry Appl.*, 7(1/2):85–121, 1997.

[12] M. Dror, A. Efrat, A. Lubiw, and J. S. B. Mitchell. Touring a sequence of polygons. In *Proc. 35th Symposium on Theory of Computing (STOC'03)*, pages 473–482, 2003.

[13] A. Dumitrescu and C. D. Tóth. Watchman tours for polygons with holes. *Comput. Geom. Theory Appl.*, 45(7):326–333, 2012.

[14] S. Eriksson-Bique, J. Hershberger, V. Polishchuk, B. Speckmann, S. Suri, T. Talvitie, K. Verbeek, and H. Yldz. Geometric k shortest paths. In S. Khanna, editor, *Proc. 26th Ann. ACM-SIAM Symp. Discrete Algorithms, (SODA'15)*, pages 1616–1625. SIAM, 2015.

[15] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry: Theory and Applications*, 5:165–185, 1995.

[16] S. Ghosh. *Visibility Algorithms in the Plane.* Cambridge University Press, 2007.

[17] S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20(5):888–910, 1991.

[18] J. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* Taylor & Francis, 2nd edition, 2010.

[19] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proc. 55th Ann. Sympos. Found. Comput. Sci. (FOCS'14)*, pages 621–630. IEEE, 2014.

[20] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[21] L. J. Guibas, R. Motwani, and P. Raghavan. The robot localization problem. *SIAM J. Comput.*, 26(4):1120–1138, Aug. 1997.

[22] O. A. Hall-Holt. *Kinetic Visibility.* PhD thesis, Stanford University, 2002.

[23] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[24] P. J. Heffernan and J. S. B. Mitchell. An optimal algorithm for computing visibility in the plane. *SIAM J. Comput.*, 24(1):184–201, 1995.

[25] J. Hershberger. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett.*, 33(4):169–174, 1989.

[26] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995.

[27] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Comput.*, 28(6):2215–2256, 1999.

[28] R. Inkulu and S. Kapoor. Visibility queries in a polygonal region. *Comput. Geom. Theory Appl.*, 42(9):852–864, 2009.

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

[30] R. Khosravi and M. Ghodsi. The fastest way to view a query point in simple polygons. In *21st European Workshop on Computational Geometry (EuroCG'05)*, pages 187–190. Eindhoven, 2005.

[31] C. Knauer, G. Rote, and L. Schlipf. Shortest inspection-path queries in simple polygons. In *24th European Workshop on Computational Geometry (EuroCG'08)*, pages 153–156, 2008.

[32] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14:393–410, 1984.

[33] L. Lu, C. Yang, and J. Wang. Point visibility computing in polygons with holes. *Journal of Information & Computational Science*, 8(16):4165–4173, 2011.

[34] J. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8(1):431–459, 1992.

[35] J. S. B. Mitchell. Shortest paths among obstacles in the plane. *Internat. J. Comput. Geom. Appl.*, 6:309–332, 1996.

[36] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 633–701. Elsevier, 2000.

[37] J. S. B. Mitchell. Shortest paths and networks. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 445–466. Elsevier, 2004.

[38] J. S. B. Mitchell. Approximating watchman routes. In S. Khanna, editor, *Proc. 24th Annual ACM-SIAM Symp. on Discrete Algorithms, SODA'13*, pages 844–855. SIAM, 2013.

[39] J. S. B. Mitchell, V. Polishchuk, and M. Sysikaski. Minimum-link paths revisited. *Comput. Geom. Theory Appl.*, 47(6):651–667, 2014.

[40] J. O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, 1987.

[41] E. Packer. Computing multiple watchman routes. In C. C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA, Provincetown, MA, USA*, volume 5038 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2008.

[42] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, NY, USA, 1996.

[43] S. Suri. A linear time algorithm with minimum link paths inside a simple polygon. *Computer Vision, Graphics and Image Processing*, 35(1):99–110, 1986.

[44] S. Suri and J. O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Proc. 2nd Ann. Symp. Computational Geometry*, pages 14–23. ACM, 1986.

[45] T. Talvitie. Visualizing Quickest Visibility Maps. In L. Arge and J. Pach, editors, *31st International Symposium on Computational Geometry (SoCG 2015)*, volume 34 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26–28, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://www.cs.helsinki.fi/group/compgeom/qvm/.

[46] A. Zarei and M. Ghodsi. Query point visibility computation in polygons with holes. *Comput. Geom. Theory Appl.*, 39(2):78–90, 2008.