CORE Metadata, citation and similar papers at core.ac.uk
Provided by Dagstuhl Research Online Publication Server

Quickest Visibility Queries in Polygonal Domains[*][†]

Haitao Wang

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

Abstract

Let s be a point in a polygonal domain $\mathcal{P}$ of $h-1$ holes and n vertices. We consider the following *quickest visibility query* problem. Given a query point q in $\mathcal{P}$, the goal is to find a shortest path in $\mathcal{P}$ to move from s to *see* q as quickly as possible. Previously, Arkin et al. (SoCG 2015) built a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(K \log^2 n)$ time, where $\alpha(n)$ is the inverse Ackermann function and K is the size of the visibility polygon of q in $\mathcal{P}$ (and K can be $\Theta(n)$ in the worst case). In this paper, we present a new data structure of size $O(n \log h + h^2)$ that can answer each query in $O(h \log h \log n)$ time. Our result improves the previous work when h is relatively small. In particular, if h is a constant, then our result even matches the best result for the simple polygon case (i.e., $h = 1$), which is optimal. As a by-product, we also have a new algorithm for the following *shortest-path-to-segment query* problem. Given a query line segment τ in $\mathcal{P}$, the query seeks a shortest path from s to all points of τ. Previously, Arkin et al. gave a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(\log^2 n)$ time, and another data structure of size $O(n^3 \log n)$ with $O(\log n)$ query time. We present a data structure of size $O(n)$ with query time $O(h \log \frac{n}{h})$, which favors small values of h and is optimal when $h = O(1)$.

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

Keywords and phrases shortest paths, visibility, quickest visibility queries, shortest path to segments, polygons with holes

Digital Object Identifier 10.4230/LIPIcs.SoCG.2017.61

1 Introduction

Let $\mathcal{P}$ be a polygonal domain with $h-1$ holes and a total of n vertices, i.e., there is an outer simple polygon containing $h-1$ pairwise disjoint holes and each hole itself is a simple polygon. If $h = 1$, then $\mathcal{P}$ becomes a simple polygon. For any two points s and t in $\mathcal{P}$, a *shortest path* from s to t is a path in $\mathcal{P}$ connecting s and t with the minimum Euclidean length. Two points p and q are *visible* to each other if the line segment $\overline{pq}$ is in $\mathcal{P}$. For any point q in $\mathcal{P}$, its *visibility polygon* consists of all points of $\mathcal{P}$ visible to q, denoted by $\mathit{Vis}(q)$.

We consider the following *quickest visibility query* problem. Let s be a source point in $\mathcal{P}$. Given any point q in $\mathcal{P}$, the query asks for a path to move from s to *see* q as quickly as possible. Such a "quickest path" is actually a shortest path from s to all points of $\mathit{Vis}(q)$. The problem has been recently studied by Arkin et al. [1], who built a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(K \log^2 n)$ time, where K is the size of $\mathit{Vis}(q)$. In this paper, we present a new data structure of $O(n \log h + h^2)$ size with $O(h \log h \log n)$ query time. Our result improves the previous work when h is relatively small. Interesting is that the query time is independent of K, which can be $\Theta(n)$ in the worst case. Our result is

[*] A full version of the paper is available at https://arxiv.org/abs/1703.03048.
[†] This research was supported in part by NSF under Grant CCF-1317143.

© Haitao Wang;
licensed under Creative Commons License CC-BY

also interesting in that when $h = O(1)$, the data structure has $O(n)$ size and $O(\log n)$ query time, which matches the result for the simple polygon case [1] and is optimal.

As in [1], in order to solve the quickest visibility queries, we also solve a *shortest-path-to-segment query* problem (or *segment query* for short), which may have independent interest. Given any line segment τ in $\mathcal{P}$, the segment query asks for a shortest path from s to all points of τ. Arkin et al. [1] gave a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(\log^2 n)$ time, and another data structure of size $O(n^3 \log n)$ with $O(\log n)$ query time. We present a new data structure of $O(n)$ size with $O(h \log \frac{n}{h})$ query time. Our result again favors small values of h and attains optimality when $h = O(1)$, which also matches the best result for the simple polygon case [1, 7].

Given the shortest path map of s, our quickest visibility query data structure can be built in $O(n \log h + h^2 \log h)$ time and our segment query data structure can be built in $O(n)$ time. Arkin et al.'s quickest visibility query data structure and their first segment query data structure can both be built in $O(n^2 2^{\alpha(n)} \log n)$ time, and their second segment query data structure can be built in $O(n^3 \log n)$ time [1].

Throughout the paper, whenever we talk about a query related to paths in $\mathcal{P}$, the query time always refers to the time for computing the path length, and to output the actual path, it needs additional time linear in the number of edges of the path by standard techniques.

1.1 Related Work

The traditional shortest path problem is to compute a shortest path to move from s to "reach" a query point. Each shortest path query can be answered in $O(\log n)$ time by using the *shortest path map* of s, denoted by $SPM(s)$, which is of $O(n)$ size. To build $SPM(s)$, Mitchell [14] gave an algorithm of $O(n^{3/2+\epsilon})$ time for any $\epsilon > 0$ and $O(n)$ space, and later Hershberger and Suri [10] presented an algorithm of $O(n \log n)$ time and space. If $\mathcal{P}$ is a simple polygon (i.e., $h = 1$), $SPM(s)$ can be built in $O(n)$ time, e.g., see [8].

For the quickest visibility queries, Arkin et al. [1] also built a "quickest visibility map" of $O(n^7)$ size in $O(n^8 \log n)$ time, which can answer each query in $O(\log n)$ time. In addition, Arkin et al. [1] gave a conditional lower bound on the problem by showing that the 3SUM problem on n numbers can be solved in $O(\tau_1 + n \cdot \tau_2)$ time, where τ_1 is the preprocessing time and τ_2 is the query time. Therefore, a data structure of $o(n^2)$ preprocessing time and $o(n)$ query time would lead to an $o(n^2)$ time algorithm for 3SUM.

In the simple polygon case (i.e., $h = 1$), better results are known. For the quickest visibility queries, Khosravi and Ghodsi [11] first proposed a data structure of $O(n^2)$ size that can answer each query in $O(\log n)$ time. Arkin et al. [1] gave an improved result and they built a data structure of $O(n)$ size in $O(n)$ time, with $O(\log n)$ query time. For the segment queries, Arkin et al. [1] built a data structure of $O(n)$ size in $O(n)$ time, with $O(\log n)$ query time. Chiang and Tamassia [7] achieved the same result for the segment queries and they also gave some more general results (e.g., when the query is a convex polygon).

Similar in spirit to the "point-to-segment" shortest path problem, Cheung and Daescu [6] considered a "point-to-face" shortest path problem in 3D and approximation algorithms were given for the problem.

1.2 Our Techniques

We first propose a decomposition $\mathcal{D}$ of $\mathcal{P}$ by $O(h)$ shortest paths from s to certain vertices of $SPM(s)$. The decomposition $\mathcal{D}$, whose size is $O(n)$, has $O(n)$ cells with the following three key properties. First, any segment τ in $\mathcal{P}$ can intersect at most $O(h)$ cells of $\mathcal{D}$. Second, for each cell Δ of $\mathcal{D}$, $\tau \cap \Delta$ consists of at most two sub-segments of τ. Third, after $O(n)$ time preprocessing, for each sub-segment τ' of τ in any cell of $\mathcal{D}$, the shortest path from s to τ' can be computed in $O(\log n)$ time. With $\mathcal{D}$, we can easily answer each segment query in $O(h \log \frac{n}{h})$ time by a "pedestrian" algorithm.

To solve the quickest visibility queries, an observation is that the shortest path from s to see q is a shortest path from s to a *window* of $\mathit{Vis}(q)$, i.e., an extension of the segment $\overline{qu}$ for some reflex vertex u of $\mathcal{P}$. Hence, the query can be answered by calling segment queries on all $O(K)$ windows of $\mathit{Vis}(s)$. This leads to the $O(K \log^2 n)$ time query algorithm in [1].

If we follow the same algorithmic scheme and using our new segment query algorithm, then we would obtain an algorithm of $O(K \cdot h \cdot \log \frac{n}{h})$ time for the quickest visibility queries. We instead present a "smarter" algorithm that prunes some "unnecessary" portions of the windows such that it suffices to consider the remaining parts of the windows. Further, with the help of the decomposition $\mathcal{D}$, we show that a shortest path from s to the remaining windows can be found in $O((K + h) \log h \log n)$ time. We refer to it as *the preliminary result*. To achieve this result, we solve many other problems, which may be of independent interest. For example, we build a data structure of $O(n \log h)$ size such that given any query point t and line segment τ in $\mathcal{P}$, we can compute in $O(\log h \log n)$ time the intersection between τ and the shortest path from s to t in $\mathcal{P}$.

To further reduce the query time to $O(h \log h \log n)$, by using the extended corridor structure of $\mathcal{P}$ [3, 5], we show that there exists a set $\mathcal{S}(q)$ of $O(h)$ *candidate windows* such that a shortest path from s to see the query point q must be a shortest path from s to a window in $\mathcal{S}(q)$. This is actually quite consistent with the result in the simple polygon case, where only one window is needed for answering each quickest visibility query [1]. Once the set $\mathcal{S}(q)$ is computed, we can apply our pruning algorithm discussed above on $\mathcal{S}(q)$ to answer the quickest visibility query in additional $O(h \log h \log n)$ time. To compute $\mathcal{S}(q)$, we give an algorithm of $O(h \log n)$ time, without having to explicitly compute $\mathit{Vis}(s)$. The algorithm is based on a modification of the algorithm given in [4] that can compute $\mathit{Vis}(q)$ in $O(K \log n)$ time for any point q, after $O(n + h^2)$ space and $O(n + h^2 \log h)$ time preprocessing.

The rest of the paper is organized as follows. In Section 2, we define notation and review some concepts. In Section 3, we introduce the decomposition $\mathcal{D}$ of $\mathcal{P}$, and discuss the segment queries. We present our preliminary result for quickest visibility queries in Section 4, and the improved result is discussed in Section 5. Due to the space limit, we only sketch the main idea and all details can be found in the full paper [15].

2 Preliminaries

For any subset A of $\mathcal{P}$, we say that a point p is *(weakly) visible* to A if p is visible to at least one point of A. For any point $t \in \mathcal{P}$, we use $\pi(s,t)$ to denote a shortest path from s to t in $\mathcal{P}$, and in the case where the shortest path is not unique, $\pi(s,t)$ may refer to an arbitrary such path. With a little abuse of notation, for any subset A of $\mathcal{P}$, we use $\pi(s,A)$ to denote a shortest path from s to all points of A; we use $d(s,A)$ to denote the length of $\pi(s,A)$, i.e., $d(s,A) = \min_{t \in A} d(s,t)$. Let $\mathcal{V}$ denote the set of all vertices of $\mathcal{P}$.

The shortest path map. The shortest path map $\mathit{SPM}(s)$ is a decomposition of $\mathcal{P}$ into regions (or cells) such that in each cell σ, the sequence of obstacle vertices along $\pi(s,t)$ is fixed for all t in σ [10, 14]. Further, the *root* of σ, denoted by $r(\sigma)$, is the last vertex of $\mathcal{V} \cup \{s\}$ in $\pi(s,t)$ for any point $t \in \sigma$ (hence $\pi(s,t) = \pi(s,r(\sigma)) \cup \overline{r(\sigma)t}$; note that $r(\sigma)$ is s if s is visible to t). We classify each edge of a cell σ into three types: a portion of an edge of $\mathcal{P}$, *an extension segment*, which is a line segment extended from $r(\sigma)$ along the opposite direction from $r(\sigma)$ to the vertex of $\pi(s,t)$ preceding $r(\sigma)$, and *a bisector curve/edge* that is a hyperbolic arc. For each point t on a bisector edge of $\mathit{SPM}(s)$, t is on the common boundary of two cells and there are two different shortest paths from s to t through the roots of the two cells, respectively. The *vertices* of $\mathit{SPM}(s)$ include $\mathcal{V} \cup \{s\}$ and all intersections of edges of $\mathit{SPM}(s)$. The intersection of two bisector edges is called a *triple point*, which has more than two shortest paths from s. The map $\mathit{SPM}(s)$ has $O(n)$ vertices, edges, and cells [10, 14].

For differentiation, we call the vertices and edges of $\mathcal{P}$ the *obstacle vertices* and the *obstacle edges*, respectively. The holes and the outer polygon of $\mathcal{P}$ are also called *obstacles*.

The *shortest path tree* $\mathit{SPT}(s)$ is the union of shortest paths from s to all obstacle vertices of $\mathcal{P}$. $\mathit{SPT}(s)$ has $O(n)$ edges [10, 14]. Given $\mathit{SPM}(s)$, $\mathit{SPT}(s)$ can be obtained in $O(n)$ time.

For ease of exposition, we make a general position assumption that no obstacle vertex has more than one shortest path from s and no point of $\mathcal{P}$ has more than three shortest paths from s. Hence, no bisector edge of $\mathit{SPM}(s)$ intersects an obstacle vertex and no three bisector edges intersect at the same point.

For any polygon P, we use $|P|$ to denote the number of vertices of P and use ∂P to denote the boundary of P.

Ray-shooting queries in simple polygons. Let P be a simple polygon. With $O(|P|)$ time and space preprocessing, each ray-shooting query in P (i.e., given a ray in P, find the first point on ∂P hit by the ray) can be answered in $O(\log |P|)$ time [2, 9]. The result can be extended to curved simple polygons or splinegons [12].

The canonical lists and cycles of planar trees. We will often talk about certain planar trees in $\mathcal{P}$ (e.g., $\mathit{SPT}(s)$). Consider a tree T with root r. A leaf v is called a *base leaf* if it is the leftmost leaf of a subtree rooted at a child of r. Denote by $\mathcal{L}(T,v)$ the post-order traversal list of T starting from such a base leaf v, and we call it a *canonical list* of T. The root r must be the last node in $\mathcal{L}(T,v)$. We remove r from $\mathcal{L}(T,v)$ and make the remaining list a cycle by connecting its rear to its front, and let $\mathcal{C}(T)$ denote the circular list. Although T may have multiple base leaves, $\mathcal{C}(T)$ is unique and we call $\mathcal{C}(T)$ the *canonical cycle* of T. We further use $\mathcal{L}_l(T,v)$ to denote the list of the leaves of T following their relative order in $\mathcal{L}(T,v)$ and use $\mathcal{C}_l(T)$ to denote the circular list of $\mathcal{L}_l(T,v)$. One reason we introduce these notation is the following. Let e be any edge of T. All nodes of T whose paths to r in T contain e are consecutive in $\mathcal{L}(T,v)$ and $\mathcal{C}(T)$. Similarly, all leaves of T whose paths to r in T contain e must be consecutive in $\mathcal{L}_l(T,v)$ and $\mathcal{C}_l(T)$.

The following observation on shortest paths will be frequently referred to in the paper.

▶ **Observation 1.**
1. *Suppose π_1 and π_2 are two shortest paths from s to two points in $\mathcal{P}$, respectively; then π_1 and π_2 do not cross each other.*
2. *Suppose π_1 is a shortest path from s to a point in $\mathcal{P}$ and τ is a line segment in $\mathcal{P}$; then the intersection of π_1 and τ is a sub-segment of τ (which may be a single point or empty).*

3 The Decomposition $\mathcal{D}$ and the Segment Queries

In this section, we introduce a decomposition $\mathcal{D}$ of $\mathcal{P}$ and use it to solve the segment query problem. The decomposition $\mathcal{D}$ will also be useful for solving the quickest visibility queries.

We first define a set V of points. Let p be an intersection between a bisector edge of $\mathit{SPM}(s)$ and an obstacle edge. Since p is on a bisector edge, it is in two cells of $\mathit{SPM}(s)$ and has two shortest paths from s. We make two copies of p in the way that each copy belongs to only one cell (and thus corresponds to only one shortest path from s). We add the two copies of p to V. We do this for all intersections between bisector edges and obstacle edges. Consider a triple point p, which is in three cells of $\mathit{SPM}(s)$ and has three shortest paths from s. Similarly, we make three copies of p that belong to the three cells, respectively. We add the three copies of p to V. We do this for all triple points. This finishes the definition of V.

By definition, each point of V has exactly one shortest path from s. Let Π_V denote the set of shortest paths from s to all points of V. Let T_V be the union of all shortest paths of Π_V. We consider points of V distinct although some of them are copies of the same physical point. In this way, we can consider T_V as a "physical" tree rooted at s.

▶ **Definition 1.** Define $\mathcal{D}$ to be the decomposition of $\mathcal{P}$ by the edges of T_V.

In the following, we assume the shortest path map $\mathit{SPM}(s)$ has already been computed. We have the following lemma about the decomposition $\mathcal{D}$.

▶ **Lemma 2.**
1. *The size of the set V is $O(h)$.*
2. *The combinatorial size of $\mathcal{D}$ is $O(n)$.*
3. *Each cell of $\mathcal{D}$ is simply connected.*
4. *For any segment τ in $\mathcal{P}$, τ can intersect at most $O(h)$ cells of $\mathcal{D}$. Further, for each cell Δ of $\mathcal{D}$, the intersection τ and Δ consists of at most two (maximal) sub-segments of τ.*
5. *After $O(n)$ time preprocessing, for any segment τ' in a cell Δ of $\mathcal{D}$, the shortest path from s to τ' can be computed in $O(\log|\Delta|)$ time, where $|\Delta|$ is the combinatorial size of Δ.*
6. *For each cell Δ of $\mathcal{D}$, Δ has at most two vertices r_1 and r_2 (both in $\mathcal{V}\cup\{s\}$), called "super-roots", such that for any point $t\in\Delta$, $\pi(s,t)$ is the concatenation of $\pi(s,r)$ and the shortest path from r to t in Δ, for a super-root r in $\{r_1,r_2\}$.*
7. *Given the shortest path map $\mathit{SPM}(s)$, $\mathcal{D}$ can be computed in $O(n)$ time.*

Using $\mathcal{D}$, we can easily answer each segment query in $O(h\log\frac{n}{h})$ time by a "pedestrian" algorithm, similar in spirit to the ray-shooting algorithm of Hershberger and Suri [9].

▶ **Theorem 3.** *Given the shortest path map $\mathit{SPM}(s)$, we can build a data structure of $O(n)$ size in $O(n)$ time, such that each segment query can be answered in $O(h\log\frac{n}{h})$ time.*

4 The Quickest Visibility Queries: The Preliminary Result

In this section, we give our preliminary result on quickest visibility queries, which sets the stage for our improved result.

For any subset A of $\mathcal{P}$, a point $p\in A$ is called a *closest point* of A (with respect to s) if $d(s,A)=d(s,p)$. Given any query point q in $\mathcal{P}$, our goal is to find a shortest path from s to $\mathit{Vis}(q)$. Let q^* be a closest point of $\mathit{Vis}(q)$. To answer the query, it is sufficient to determine q^*. Thus we will focus on finding q^*. Note that if q is visible to s, then $q^*=s$. We can determine whether s is visible to q in $O(\log n)$ time by checking whether q is in the cell of $\mathit{SPM}(s)$ whose root is s. In the following, we assume that s is not visible to q.

We define the *windows* of q and $\mathit{Vis}(q)$. Consider an obstacle vertex u that is visible to q such that the two incident obstacle edges of u are on the same side of the line through q and u (e.g., see Fig. 1). Let $q(u)$ denote the first point on $\partial\mathcal{P}$ hit by the ray from u along the direction from q to u. Then $\overline{uq(u)}$ is called a *window* of q; we say that the window is *defined by* u. Further, we call $\overline{qq(u)}$ the *extended window* of $\overline{uq(u)}$.

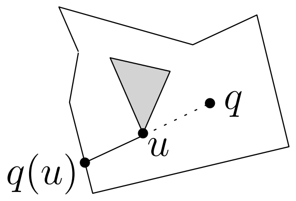

Figure 1 Illustrating a window $\overline{uq(u)}$ of q.

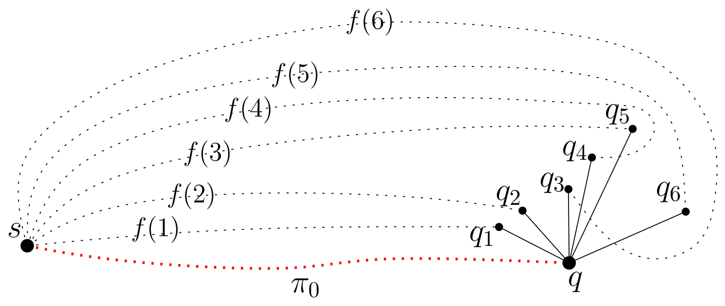

Figure 2 Illustrating the map $f(\cdot)$: $f(1) = 1$, $f(2) = 2$, $f(3) = 5$, $f(4) = 4$, $f(5) = 6$, and $f(6) = 3$. Note that the paths could be "below" π_0, but for ease of exposition, we "flip" them above π_0, and this flip operation does not change the topology of these paths.

Each window of q is an edge of $Vis(q)$, and thus the number of windows of q is $O(K)$, where $K = |Vis(q)|$. Further, there must be a closest point q^* that is on a window of q [1]. Hence, as in [1], a straightforward algorithm to compute q^* is to compute shortest paths from s to all windows of s and the path of minimum length determines q^*. To compute shortest paths from s to all windows, if we apply our segment queries on all windows using Theorem 3, then the total time would be $O(K \cdot h \cdot \log \frac{n}{h})$. In the rest of this section, we present an algorithm that can compute q^* in $O((K + h) \log h \log n)$ time, without having to compute shortest paths to all windows.

4.1 The Algorithm Overview

As the first step, we compute $Vis(q)$, which can be done in $O(K \log n)$ time after $O(n+h^2 \log h)$ time and $O(n + h^2)$ space preprocessing [4]. Then, we can find all windows and extended-windows in $O(K)$ time. For ease of exposition, we make a general position assumption for q that q is not collinear with any two obstacle vertices. The assumption implies that q is in the interior of $\mathcal{P}$ and no two windows are collinear.

Let u_0 be the root of the cell of $SPM(s)$ containing q (if q is on the boundary of multiple cells, then we take an arbitrary such cell). Hence, $\pi(s, u_0) \cup \overline{u_0 q}$ is a shortest path $\pi(s, q)$ from s to q. Note that u_0 must define a window $\overline{u_0 q(u_0)}$ of q [13]. Let $\overline{u_0 q(u_0)}, \overline{u_1 q(u_1)}, \ldots, \overline{u_k q(u_k)}$ be all windows of q ordered *clockwise* around q. Clearly, $k = O(K)$. For each $0 \leq i \leq k$, let $q_i = q(u_i)$. Note that the window $\overline{u_0 q_0}$ is special in the sense that u_0 is in $\pi(s, q)$. So we first apply our algorithm in Theorem 3 on $\overline{u_0 q_0}$ to compute a closest point q_0^* of $\overline{u_0 q_0}$. Clearly, if $q^* \in \overline{u_0 q_0}$, then $q^* = q_0^*$. In the following, we assume $q^* \notin \overline{u_0 q_0}$. Let $Q = \{q, q_1, q_2, \ldots, q_k\}$. Note that Q does not contain q_0 but q. If $q^* \in Q$, then we can find q^* by computing $d(s, p)$ for all $p \in Q$, in $O(k \log n)$ time using $SPM(s)$. Below, we assume $q^* \notin Q$. Note that the above assumption that $q^* \notin \overline{u_0 q_0} \cup Q$ is only for arguing the correctness of our following algorithm, which actually proceeds without knowing whether the assumption is true or not.

For each $0 \leq i \leq k$, let $w_i = \overline{q q_i}$, i.e., the extended window of $\overline{u_i q_i}$. Let $W = \{w_i \mid 1 \leq i \leq k\}$. For convenience of discussion, we assume that each w_i of W does not contain its two endpoints q and q_i (but the endpoints of w_i still refer to q and q_i). Since $q^* \notin \overline{u_0 q_0} \cup Q$, q^* must be on an extended window in W. Clearly, q^* is also a closest point of W. Since no two windows of q are collinear, no extended-window of W contains another. We assign each window $w_i \in W$ a direction from q to q_i, so that we can talk about its left or right side.

Suppose q^* is on $w_i \in W$. Since w_i is an open segment, by the definition of q^*, the shortest path $\pi(s, q^*)$ must reach q^* from either the left side or the right side of w_i. Formally, we say that $\pi(s, q^*)$ reaches q^* from the left side (resp., right side) of w_i if there is a small neighborhood of q^* such that all points of $\pi(s, q^*)$ in the neighborhood are on the left side (resp., right side) of w_i. Let w_i^l (resp., w_i^r) denote the set of points p on w_i whose shortest path from s to p is from the left (resp., right) side of w_i. Hence, q^* is either on w_i^l or on w_i^r.

Our algorithm will find two points q_l^* and q_r^* such that if q^* is on w_i^l for some $i \in [1, k]$, then $q^* = q_l^*$, and otherwise (i.e.,q^* is in w_i^r for some $i \in [1, k]$), $q^* = q_r^*$.

In the following, we will only present our algorithm for finding q_l^* since the case for q_r^* is symmetric. In the following discussion, we assume q^* is on w_i^l for some $i \in [1, k]$.

The rest of this section is organized as follows. In Section 4.2, we discuss some observations, based on which we describe our pruning algorithm in Section 4.3 to prune some (portions of) segments of W such that q^* $(= q_l^*)$ is still in the remaining segments of W. In Section 4.4, we will finally compute q_l^* on the remaining segments of W. As will be clear later, our algorithm uses extended windows instead of windows because extended windows can help us with the pruning.

4.2 Observations

For any point $t \in \mathcal{P}$ with $s \neq t$, and its shortest path $\pi(s, t)$, we use t^+ to denote a point on $\pi(s, t)$ arbitrarily close to t (but $t^+ \neq t$). If t is on w_i^l for some $i \in [1, k]$, then t^+ must be on the left side of w_i. For any segment w of W, we say that w or a sub-segment of w can be *pruned* if it does not contain q^*. Our pruning algorithm, albeit somewhat involved, is based on the following simple observation.

▶ **Observation 2.** *For any point $t \in w_i^l$ for some $i \in [1, k]$, if $\pi(s, t^+)$ intersects any segment $w \in W$ or an endpoint of it, then t can be pruned (i.e., t cannot be q^*).*

Proof. Let t' be a point on $\pi(s, t^+)$ that is a point on any segment $w \in W$ or an endpoint of it. Clearly, $t' \in \mathit{Vis}(s)$ and $d(s, t') < d(s, t)$. Thus, t cannot be q^*. ◀

Consider the shortest paths $\pi(s, q_i)$ for $i = 1, 2, \ldots, k$. To simplify the notation, let $\pi_i = \pi(s, q_i)$ for each $i \in [1, k]$. In particular, let $\pi_0 = \pi(s, q)$ (not $\pi(s, q_0)$). Recall that $Q = \{q, q_1, \ldots, q_k\}$. The union of all paths π_i for $0 \leq i \leq k$ forms a planar tree, denoted by T_Q, with root at s. Consider the canonical cycle $\mathcal{C}(T_Q)$ as defined in Section 2. Let $\mathcal{C}_Q$ be the circular list of the points of Q following their relative order in $\mathcal{C}(T_Q)$. We further break $\mathcal{C}_Q$ into a list $\mathcal{L}_Q$ at q, such that $\mathcal{L}_Q$ starts from q and all other points of $\mathcal{L}_Q$ follow the counterclockwise order in $\mathcal{C}_Q$. Assume $\mathcal{L}_Q$ is $\{q, q_{f(1)}, q_{f(2)}, \ldots, q_{f(k)}\}$, i.e., the $(i+1)$-th point of the list is $q_{f(i)}$ (e.g., see Fig. 2). So $f(\cdot)$ essentially maps each point of $Q \setminus \{q\}$ from its position in $\mathcal{L}_Q$ to its position in the list $\{q_1, q_2, \ldots, q_k\}$. Hence, $f(1) \ldots, f(k)$ is a permutation of $1, \ldots, k$, and $f(i) \neq f(j)$ if $i \neq j$. The reason we introduce the list $\mathcal{L}_Q$ is that intuitively, for any $1 \leq i < j \leq k$, the path $\pi_{f(j)}$ is *counterclockwise* from $\pi_{f(i)}$ with respect to π_0 around s. For convenience, we let $f(0) = 0$.

Given $\mathit{SPM}(s)$, after $O(n)$ time preprocessing, we can compute the list $\mathcal{L}_Q$ and thus determine the map $f(\cdot)$ in $O(k \log n)$ time. The details are omitted.

We can show that for any $i \in [1, k]$, π_0 does not contain q_i and π_i does not contain q. The following lemma is proved based on Observation 2.

▶ **Lemma 4.** *Suppose π_j contains q_i with $i \neq j$ and $i, j \in [1, k]$. If $i < j$, then w_j can be pruned; otherwise, w_i can be pruned.*

In $O(k \log n)$ time, we can remove all extended-windows of W that can be pruned by Lemma 4. The details are omitted. But to simplify the notation, we assume that none of the

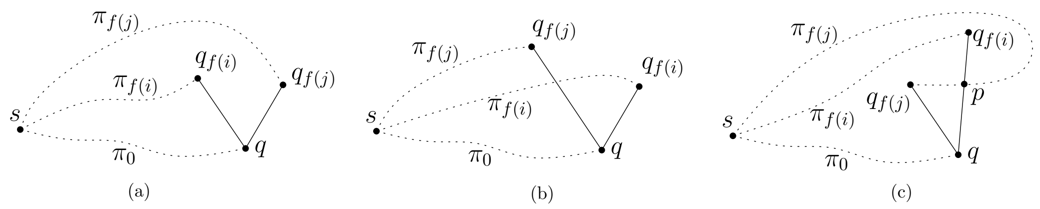

Figure 3 Illustrating Lemma 5.

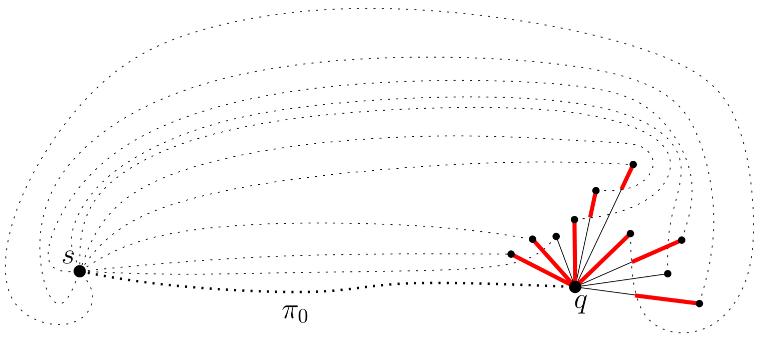

Figure 4 The thick (red) segments are the remaining parts of the segments of W after the pruning algorithms (so that q_l^* must be on the left side of a red segment). Again, we "flip" all paths above π_0.

segments of W is pruned since otherwise we could re-index all segments of W. So now W has the following property: For any $i \in [1, k]$, q_i is not contained in any π_j with $j \in [0, k]$ and $j \neq i$.

For each $i \in [1, k]$, since π_0 does not cross π_i, $\pi_0 \cup \pi_i \cup w_i$ forms a closed curve that separates the plane into two regions, one locally on the left of w_i and the other locally on the right w_i. We let D_i denote the region locally on the left side of w_i including $\pi_0 \cup \pi_i \cup w_i$ as its boundary (it is possible that D_i is unbounded). If $\pi_0 \cap \pi_i$ is a sub-path including at least one edge, then it is also considered to be in D_i. We can show that if $q^* \in w_i^l$, then $\pi(s, q^*)$ must be in D_i.

Our pruning algorithm mainly relies on Lemma 5, which is based on Observation 2.

▶ **Lemma 5.** *Suppose i and j are two indices with $1 \leq i < j \leq k$.*

1. *If $f(i) < f(j)$, then $\pi_{f(i)}$ does not cross $w_{f(j)}$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, and further, $D_{f(i)}$ is contained in $D_{f(j)}$ (e.g., see Fig. 3(a)).*
2. *If $f(i) > f(j)$, then either $\pi_{f(i)}$ crosses $w_{f(j)}$ or $\pi_{f(j)}$ crosses $w_{f(i)}$. Further, in the former case (see Fig. 3(b)), $w_{f(i)}$ can be pruned, and in the latter case (see Fig. 3(c)), the sub-segment $\overline{qp}$ of $w_{f(i)}$ can be pruned, where p is the point at which $\pi_{f(j)}$ crosses $w_{f(i)}$.*

For any $1 \leq i < j \leq k$, we say π_i and π_j are *consistent* if $f(i) < f(j)$. By Lemma 5, if π_i and π_j are not consistent, then we can do some pruning, based on which we present our pruning algorithm in Section 4.3. Figure 4 gives an example showing the remaining parts of the segments of W after the pruning.

4.3 A Pruning Algorithm for Pruning the Segments of W

We process the paths $\pi_{f(1)}, \pi_{f(2)}, \ldots, \pi_{f(k)}$ in this order. Assume that $\pi_{f(i-1)}$ has just been processed and we are about to process $\pi_{f(i)}$. Our algorithm maintains a sequence of *bundles*,

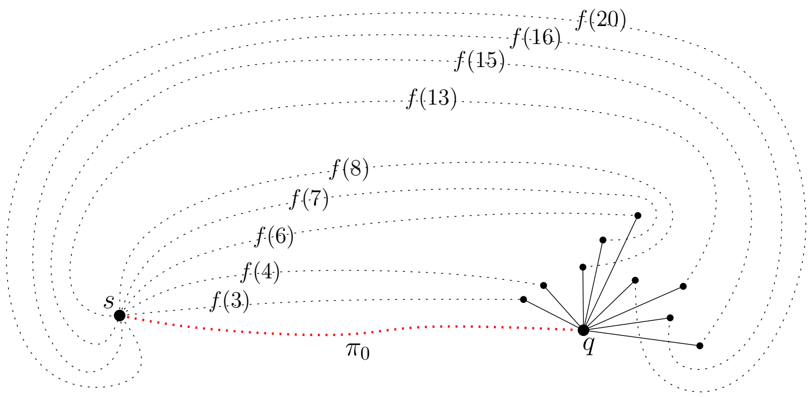

Figure 5 Illustrating the shortest paths corresponding to the indices in the current bundle sequence $\mathbb{B} = \{\underline{\{3\}}, \underline{\{4\}}, \underline{\{\{\{6\},\{7\}\},\{8\}\}}, \underline{\{\{13\},\{\{15\},\{16\}\},\{20\}\}}\}$, where each underline indicates a bundle of $\mathbb{B}$. For example, the last bundle is a composite bundle consisting of three children bundles with 20 as its wrap index. In the figure, the indices of the paths are labeled. Again, we "flip" all paths above π_0.

denoted by $\mathbb{B} = \{B_1, B_2, \ldots B_g\}$. Each *bundle* $B \in \mathbb{B}$ is defined recursively as follows. Essentially B is a list of sorted indices of a subset of $\{1, 2, \ldots, i-1\}$, but the indices are grouped in a special and systematic way.

There are two types of bundles: *atomic* and *composite*. If B has only one index, then it is an atomic bundle. Otherwise, B is a composite bundle consisting of a sequence of at least two bundles $B'_1, \ldots, B'_{g'}$ (with $g' \geq 2$) such that the last bundle $B'_{g'}$ must be atomic (others can be either atomic or composite), and we call the index contained in $B'_{g'}$ the *wrap index* of B. We consider the bundles $B'_1, \ldots, B'_{g'}$ as the *children bundles* of B.

Let $f_{\min}(B)$ and $f_{\max}(B)$ denote the smallest and largest $f(j)$ of all indices j of B, respectively. If B is composite, then B further has the following three *bundle-properties*. (1) The indices of B are distinct and sorted increasingly by their order in B. (2) For any $1 \leq b < g' - 1$, $f_{\max}(B'_b) < f_{\min}(B'_{b+1})$. (3) If j is the wrap index of B, then $f_{\min}(B) = f(j)$ and $\pi_{f(j)}$ crosses $w_{f(j')}$ for every $j' \in B \setminus \{j\}$ (intuitively, $\pi_{f(j)}$ "wraps" the point $q_{f(j')}$, and this is why we call j a "wrap" index). Refer to Fig. 5 for an example.

For convenience, if the context is clear, we also consider a bundle B as a set of sorted indices. So if an index j is in B, we can write "$j \in B$". We use the word "bundle" because each index j of B refers to the path $\pi_{f(j)}$. Therefore, B is a "bundle" of shortest paths.

In addition, the bundle sequence $\mathbb{B} = \{B_1, B_2, \ldots, B_g\}$ maintained by our algorithm has two $\mathbb{B}$-*properties*. (1) The indices in all bundles are distinct in $[1, i-1]$ and are sorted increasingly by their order in the sequence. (2) For any $1 \leq b < g$, $f_{\max}(B_b) < f_{\min}(B_{b+1})$.

▶ **Observation 3.**

1. *For any $1 \leq b < b' \leq g$ and any indices $j \in B_b$ and $j' \in B_{b'}$ (both B_b and $B_{b'}$ are from $\mathbb{B}$), the two shortest paths $\pi_{f(j)}$ and $\pi_{f(j')}$ are consistent (see Fig. 5).*
2. *For any composite bundle $B = \{B'_1, \ldots, B'_{g'}\}$, for any $1 \leq b < b' \leq g' - 1$ and any indices $j \in B'_b$ and $j' \in B'_{b'}$, the two shortest paths $\pi_{f(j)}$ and $\pi_{f(j')}$ are consistent (see Fig. 5).*

In the following, we describe our algorithm for processing the shortest path $\pi_{f(i)}$, during which $\mathbb{B}$ will be updated. Initially when $i = 1$, $\mathbb{B}$ contains the only atomic bundle $B = \{1\}$ and this finishes our processing for $\pi_{f(1)}$. In general when $i > 1$, we do the following.

We first find the index β such that $f_{\max}(B_\beta) < f(i) < f_{\max}(B_{\beta+1})$. We can maintain the bundle sequence $\mathbb{B}$ in a data structure so that β can be found in $O(\log n)$ time. The details are omitted. If $\beta = g$ (so $B_{\beta+1}$ does not exist in this case), then we add a new atomic bundle $B_{g+1} = \{i\}$ to the rear of $\mathbb{B}$ and this finishes the processing of $\pi_{f(i)}$.

If $\beta \neq g$, we check whether $f_{\min}(B_{\beta+1}) < f(i)$. If $f_{\min}(B_{\beta+1}) < f(i)$, we can show that $w_{f(i)}$ can be pruned. Hence, in this case, we simply ignore $\pi_{f(i)}$ and finish the processing of $\pi_{f(i)}$. In the following, we assume $f(i) < f_{\min}(B_{\beta+1})$ (note that $f(i) \neq f_{\min}(B_{\beta+1})$ since $i \notin \mathbb{B}$). Next, we are going to find all such indices j of $\mathbb{B}$ that $\pi_{f(j)}$ crosses $w_{f(i)}$. To this end, the following two lemmas are crucial.

▶ **Lemma 6.**
1. *For any index j in B_b for any $b \in [1, \beta]$, $\pi_{f(j)}$ does not cross $w_{f(i)}$.*
2. *For any index j in B_b for any $b \in [\beta+1, g]$, if $\pi_{f(j)}$ crosses $w_{f(i)}$, then $w_{f(j)}$ can be pruned; otherwise, $\pi_{f(i)}$ must cross $w_{f(j)}$.*
3. *If j is in B_b for some $b \in [\beta+2, g]$ and $\pi_{f(j)}$ crosses $w_{f(i)}$, then $\pi_{f(j')}$ crosses $w_{f(i)}$ for any $j' \in B_{b'}$ and any $b' \in [\beta+1, b-1]$.*
4. *If j is in B_b for some $b \in [\beta+1, g-1]$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, then $\pi_{f(j')}$ does not cross $w_{f(i)}$ for any $j' \in B_{b'}$ and any $b' \in [b+1, g]$.*

For any bundle B in $\{B_{\beta+1}, B_{\beta+2}, \ldots, B_g\}$, if B has two indices j and j' such that $w_{f(i)}$ crosses $\pi_{f(j)}$ but does not cross $\pi_{f(j')}$, then we say that B is a *mixed* bundle, which is necessarily a composite bundle.

▶ **Lemma 7.** *For any mixed bundle $B = \{B'_1, B'_2, \ldots, B'_{g'}\}$, the following holds.*
1. *The path $\pi_{f(r)}$ must cross $w_{f(i)}$, where r is the wrap index of B, i.e., $B'_{g'} = \{r\}$.*
2. *If an index j is in B'_b for some $b \in [2, g'-1]$ and $\pi_{f(j)}$ crosses $w_{f(i)}$, then $\pi_{f(j')}$ crosses $w_{f(i)}$ for any $j' \in B'_{b'}$ and any $b' \in [1, b-1]$.*
3. *If an index j is in B'_b for some $b \in [1, g'-2]$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, then $\pi_{f(j')}$ does not cross $w_{f(i)}$ for any $j' \in B'_{b'}$ and any $b' \in [b+1, g'-1]$.*
4. *If a bundle B' of B has two indices j and j' such that $w_{f(i)}$ crosses $\pi_{f(j)}$ but does not cross $\pi_{f(j')}$, then B' is also a* mixed *bundle. This lemma applies to B' recursively.*

In light of the preceding two lemmas, in the following we will find the indices j of $\mathbb{B}$ such that $\pi_{f(j)}$ crosses $w_{f(i)}$ and then prune $w_{f(j)}$ by Lemma 6(2) (i.e., remove j from $\mathbb{B}$); we say that such an index j is *prunable*.

Before describing our algorithm, we discuss an operation that will be used in the algorithm. Consider a composite bundle $B = \{B'_1, B'_2, \ldots, B'_{g'}\}$ of $\mathbb{B}$. Let r be a wrap index of B, i.e., $B'_{g'} = \{r\}$. Suppose $w_{f(i)}$ crosses $\pi_{f(r)}$. Our algorithm will remove r from B and thus from $\mathbb{B}$. This is done by a *wrap-index-removal* operation. Further, suppose B is the j-th bundle of $\mathbb{B}$, i.e., $B = B_j$. After r is removed, the operation will implicitly insert the bundles $B'_1, B'_2, \ldots, B'_{g'-1}$ into the position of B in $\mathbb{B}$, i.e., after the operation, $\mathbb{B}$ becomes $B_1, \ldots, B_{j-1}, B'_1, \ldots, B'_{g'-1}, B_{j+1}, \ldots, B_g$. Note that this new bundle list still has the two $\mathbb{B}$-properties. Indeed, $f_{\max}(B_{j-1}) < f_{\min}(B) = f(r) < f_{\min}(B'_1)$ and $f_{\max}(B'_{g'-1}) \leq f_{\max}(B) < f_{\min}(B_{j+1})$. We can maintain the bundles of $\mathbb{B}$ in a data structure so that each wrap-index-removal operation can be performed in $O(\log n)$ time. The details are omitted.

Another operation that is often used in the algorithm is the following. Given any $i, j \in [1, k]$, we want to determine whether $w_{f(i)}$ crosses $\pi_{f(j)}$. We call it the *shortest path segment intersection* (or *SP-segment-intersection*) query. Our full paper presents an algorithm that can answer each such query in $O(\log h \log n)$ time, after $O(n \log h)$ time and space preprocessing.

We are ready to describe our algorithm for removing all prunable indices from $\mathbb{B}$. By Lemma 6(1), each bundle B_b of $\mathbb{B}$ for $1 \leq b \leq \beta$ does not contain any prunable index.

For each bundle B of $B_{\beta+1}, B_{\beta+2}, \ldots, B_g$ in order, we call a procedure *prune(B)* until the procedure returns “false”.

If all indices of B are prunable, then *prune(B)* will return “true” and the entire bundle B will be removed from $\mathbb{B}$. Otherwise, the procedure will return false. Further, if B is a mixed bundle, then all prunable indices of B will be removed (and the procedure returns false).

The procedure $\mathit{prune}(B)$ works as follows. It is a recursive procedure. As a base case, if B is an atomic bundle $\{j\}$, then we call an SP-segment-intersection query to check whether $\pi_{f(j)}$ crosses $w_{f(i)}$. If yes, we remove B and return true; otherwise, return false. If B is a composite bundle $\{B'_1, B'_2, \ldots, B'_{g'}\}$ with r as the wrap index (i.e., $B'_{g'} = \{r\}$), then we first call an SP-segment-intersection to check whether $\pi_{f(r)}$ crosses $w_{f(i)}$. If not, by Lemma 7(1), B does not have any prunable index and thus we simply return false. If yes, then we call a wrap-index-removal operation to remove $B'_{g'}$. Afterwards, for each $b' = 1, 2, \ldots, g' - 1$ in order, we call $\mathit{prune}(B'_{b'})$ recursively. If $\mathit{prune}(B'_{b'})$ returns false, then we return false (without calling $\mathit{prune}(B'_{b'+1})$). If it returns true, we remove $B'_{b'}$ (in fact all children bundles of $B'_{b'}$ have been removed by $\mathit{prune}(B'_{b'})$). If $b' = g' - 1$, then we return true (since all bundles of B have been removed); otherwise, we proceed on calling $\mathit{prune}(B'_{b'+1})$.

If *prune(*B_b*)* returns true for every b with $\beta + 1 \le b \le g$, then we add a new atomic bundle $\{i\}$ at the end of $\mathbb{B}$, which now becomes $\{B_1, B_2, \ldots, B_\beta, \{i\}\}$. This also finishes our preprocessing for $\pi_{f(i)}$. Otherwise, *prune(*B_b*)* returns false for some b with $\beta + 1 \le b \le g$. In this case, as a final step, we create a new composite bundle B, consisting of all bundles of $\mathbb{B}$ after B_β (not including B_β) and the atomic bundle $\{i\}$ as the last child bundle of B. This is done by a *bundle-creation* operation, which can be implemented in $O(\log n)$ time (the details are omitted). Afterwards, the new bundle sequence $\mathbb{B}$ becomes $\{B_1, B_2, \ldots, B_\beta, B\}$. It can be shown that the new bundle B is a “valid” composite bundle and the updated $\mathbb{B}$ maintains the two $\mathbb{B}$-properties.

To analyze the running time of the above algorithm, let m be the number of indices that have been removed from $\mathbb{B}$. Then, the algorithm makes at most $m+1$ SP-segment-intersection queries. To see this, once the query discovers an index j that is not prunable, the algorithm will stop without making any more such queries. On the other hand, each wrap-index-removal operation removes an index, and thus the number of such operations is at most m. Further, observe that for each bundle B, whenever we make a recursive call on a child bundle of B, the wrap index of B is guaranteed to be removed. Therefore, the number of total recursive calls is at most m as well. Hence, the running time of the algorithm is $O((m+1)\log h \log n)$.

This finishes our algorithm for processing the path $\pi_{f(i)}$. The total time for processing $\pi_{f(i)}$ is $O((m+1)\log h \log n)$. Since once an index is removed from $\mathbb{B}$, it will never be inserted into $\mathbb{B}$ again, the sum of all such m in the entire algorithm for processing all paths $\pi_{f(i)}$ for $i = 1, 2, \ldots, k$ is at most k. Hence, the total time of the entire algorithm is $O(k \log h \log n)$.

4.4 Computing the Closest Point q^*

Recall that we have assumed that q^* is on w_i^l for some $i \in [1, k]$, i.e., $q^* = q_l^*$. According to our pruning algorithm for computing the bundle sequence $\mathbb{B}$, q^* must be on $w_{f(j)}^l$ for some $j \in \mathbb{B}$. In this section, we will compute q^* by using the bundle sequence $\mathbb{B}$. For example, in Fig 4, our goal is to compute q^* on the left sides of those (red) thick segments.

4.4.1 The Set of Regions $\mathcal{R}$

Our algorithm for computing q^* uses a set $\mathcal{R}$ of regions of $\mathcal{P}$, which is introduced below.

Let $\mathcal{O}$ denote the obstacle space, which is the complement of the free space of $\mathcal{P}$. More specifically, $\mathcal{O}$ consists of the $h-1$ simple polygonal holes of $\mathcal{P}$ and the (unbounded) region outside the outer boundary of $\mathcal{P}$. Let $\mathcal{B}$ denote the union of all bisector edges of $\mathit{SPM}(s)$. Mitchell [13] proved that $\mathcal{O} \cup \mathcal{B}$ is simply connected and $\mathcal{P} \setminus \mathcal{B}$ is also simply connected (e.g., see Fig.1 in the appendix). We consider $\mathcal{O} \cup \mathcal{B}$ as a planar graph G. Specifically, the vertex set of G consists of all obstacles of $\mathcal{O}$ and all triple points of $\mathit{SPM}(s)$. For any two vertices of G, if they are connected by a chain of bisector edges in $\mathit{SPM}(s)$ such that the chain does not contain any other vertex of G, then G has an edge connecting the two vertices, and further, we call the above chain of bisector edges a *bisector super-curve*. It can be shown that G is a simple graph with $O(h)$ vertices, edges, and faces.

Since $|V| = O(h)$ (by Lemma 2(1)), Π_V is a set of $O(h)$ shortest paths. Recall that T_V is the union of all shortest paths of Π_V and T_V is considered as a “physical” tree rooted at s. Note that each edge of any path of Π_V except the last edge (i.e., the one connecting a point of V) is an edge of $\mathit{SPT}(s)$. Hence, the total number of edges of the tree T_V is $O(n)$. Throughout the paper, let $h^* = |V|$. Thus, $h^* = O(h)$.

It is known that $\mathcal{P} \setminus \mathcal{B}$ is simply connected and $\pi(s,t)$ is in $\mathcal{P} \setminus \mathcal{B}$ for any point $t \in \mathcal{P}$ [13]. To simplify the discussion, together with the copies of the points of V, we consider $\mathcal{P}' = \mathcal{P} \setminus \mathcal{B}$ as a simple polygon (with some curved edges) by making two copies for each interior point of every bisector super-curve such that they respectively belong to the two sides of the curve.

Since T_V is a planar tree, we can define its canonical lists as discussed in Section 2. Let v_1 be an arbitrary base leaf of T_V. Let the leaf list $\mathcal{L}_l(T_V, v_1)$ be $v_1, v_2, \ldots, v_{h^*}$, which follow the counterclockwise order along $\partial\mathcal{P}'$.

For each $1 \le i \le h^*$, let α_i denote the portion of $\partial\mathcal{P}'$ counterclockwise from v_i to v_{i+1} (let v_{h^*+1} refer to v_1). Note that α_i is either a bisector super-curve or a chain of obstacle edges. Suppose we move a point t on α_i from v_i to v_{i+1}. The shortest path $\pi(s,t)$ will continuously change with the same topology since $\pi(s,t)$ is always in $\mathcal{P}'$ (which is simply connected). Let R_i be the region of $\mathcal{P}'$ that is “swept” by $\pi(s,t)$ during the above movement of t. More specifically, let p_i be the common point on $\pi(s,v_i) \cap \pi(s,v_{i+1})$ that is farthest to s. Then, R_i is bounded by $\pi(p_i,v_i)$, $\pi(p_i,v_{i+1})$, and α_i. For convenience of discussion, we let R_i also contain the common sub-path $\pi(s,p_i) = \pi(s,v_i) \cap \pi(s,v_{i+1})$ and we call $\pi(s,p_i)$ the *tail* of R_i. We call the region bounded by $\pi(p_i,v_i)$, $\pi(p_i,v_{i+1})$, and α_i the *cell* of R_i. We consider $\pi(s,v_i)$, $\pi(s,v_{i+1})$, and α_i as the three portions of the boundary ∂R_i of R_i. The definition implies that for any point t in R_i, $\pi(s,t)$ is in R_i. In fact, if t is in the cell of R_i, then $\pi(s,t)$ is the concatenation of $\pi(s,p_i)$ and the shortest path from p_i to t in the cell. Clearly, $\mathcal{P}'$ is the union of $R_1, R_2, \ldots, R_{h^*}$. Roughly speaking, the regions $R_1, \ldots, R_{h^*}$ are counterclockwise around s. Define $\mathcal{R} = \{R_1, R_2, \ldots, R_{h^*}\}$.

4.4.2 The Algorithm for Computing q^*

Let τ be any segment in $\mathcal{P}$ such that a region $R_i \in \mathcal{R}$ contains $\pi(s,\tau)$. Suppose R_i is known. With the help of the decomposition $\mathcal{D}$ proposed in Section 3, we give a *region-processing* algorithm in the full paper to compute $\pi(s,\tau)$ in $O(\log h \log n)$ time.

Recall that $\mathcal{R} = \{R_1, R_2, \ldots, R_{h^*}\}$. Due to our general position assumption that q is not collinear with any two obstacle vertices, none of $\{q, q_1, \ldots, q_k\}$ is an obstacle vertex. Then, for each $k' \in [0,k]$, there is a unique region R_i of $\mathcal{R}$ whose cell contains $q_{f(k')}$, such that the shortest path $\pi_{f(k')}$ is contained in R_i, and we let $z(k')$ refer to the index i of R_i. Computing the indices $z(0), z(1), \ldots, z(k)$ can be done in $O(k \log n)$ time by point location queries on the cells of the regions of $\mathcal{R}$.

For any two indices k_1 and k_2 in $[1,h^*]$, if $k_1 \le k_2$, then let $[k_1,k_2]_R$ denote the set of all integers $k' \in [k_1,k_2]$; otherwise, let $[k_1,k_2]_R$ denote the set of all integers $k' \in [k_1,h^*] \cup [1,k_2]$.

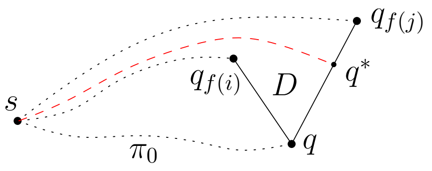

Figure 6 Illustrating Observation 4.

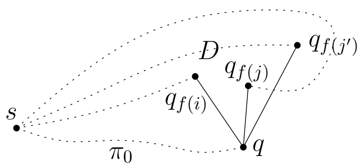

Figure 7 j is the wrap index of B_b and j' is another index of B_b with $j' \neq j$; $\pi_{f(j')}$ is in the region D.

Recall that the regions $R_1, \ldots, R_{h^*}$ are counterclockwise around s. We actually use $[k_1, k_2]_R$ to refer to the set of indices of the regions of $\mathcal{R}$ from R_{k_1} to R_{k_2} counterclockwise around s.

Next we compute q^* on $w^l_{f(j)}$ for $j \in \mathbb{B}$. Consider the bundles of $\mathbb{B} = \{B_1, B_2, \ldots, B_g\}$. For each b with $1 \leq b \leq g$, we call a procedure *path*$(B_b, z(i))$, where i is the last index of B_{b-1} if $b \geq 2$ and $i = 0$ otherwise. Note that $i < j$ for any index $j \in B_b$. The procedure *path*$(B_b, z(i))$ works as follows. Depending on whether B_b is atomic, there are two cases.

The atomic case. If B_b is atomic, let j be the only index of B_b. According to the bundle-properties, $i < j$ and $f(i) < f(j)$. So $\pi_{f(j)}$ and $\pi_{f(i)}$ are consistent. By Lemma 5(1), D_i is contained in D_j. Let D be D_j minus the interior of D_i.

▶ **Observation 4.** *If q^* is on $w^l_{f(j)}$, then $\pi(s, q^*)$ must be in D (see Fig. 6).*

▶ **Lemma 8.** *If q^* is on $w^l_{f(j)}$, then $\pi(s, q^*)$ is in $R_{k'}$ for some index $k' \in [z(i), z(j)]_R$, and further, any shortest path $\pi(s, w_{f(j)})$ from s to $w_{f(j)}$ is $\pi(s, q^*)$.*

For each $k' \in [z(i), z(j)]_R$, we apply our region-processing algorithm on $R_{k'}$ and $w_{f(j)}$ to obtain a path, and we keep the shortest path π among all such paths; let $q^l_{f(j)}$ be the endpoint of π on $w_{f(j)}$. According to Lemma 8, if q^* is on $w^l_{f(j)}$, then q^* must be $q^l_{f(j)}$.

For analyzing the total running time of our algorithm, as will be seen later, for each $k' \in [z(i), z(j)]_R$ with $k' \neq z(i)$ and $k' \neq z(j)$, the region-processing algorithm will not be called on $R_{k'}$ again in the entire algorithm for computing q^*_l. On the other hand, we charge the two algorithm calls on $R_{k'}$ for $k' = z(i)$ and $k' = z(j)$ to the index j of $\mathbb{B}$. In this way, the total number of calls to the region-processing procedure in the entire algorithm is $O(h^* + k)$ since the total number of indices of $\mathbb{B}$ is at most k and the total number of regions $R_{k'}$ is h^*.

The composite case. If B_b is composite, the algorithm is more complicated. Let j be the wrap index of B_b. Observation 4 and Lemma 8 still hold on j. However, since now the region D contains a portion of $w_{f(j')}$ for each $j' \in B_b \setminus \{j\}$ (see Fig. 7), D may also contain the shortest path from s to $w_{f(j')}$. In order to avoid calling the region-processing procedure on the same region of $\mathcal{R}$ too many times, we use the following approach to process $w_{f(j)}$.

For any two different indices of k' and k'' in a range $[k_1, k_2]_R$ of indices of the regions of $\mathcal{R}$, we say that k'' is *ccw-larger* than k' if $[k', k'']_R$ is a subset of $[k_1, k_2]_R$ (e.g., if $k_1 < k_2$, then $k' < k''$). Define z_{ij} to be the ccw-largest index in $[z(i), z(j)]$ such that $w_{f(j)}$ crosses $\partial R_{z_{ij}}$ (if no such index exists, then let $z_{ij} = z(i)$).

We first compute z_{ij} (to be discussed later). Then, we call the region-processing procedure on $R_{k'}$ for all $k' \in [z(i), z_{ij}]$ and return the shortest path π that is found; let $q^l_{f(j)}$ be the endpoint of π on $w_{f(j)}$. By the following lemma, if q^* is on $w^l_{f(j)}$, then $q^l_{f(j)}$ is q^*.

▶ **Lemma 9.** *If q^* is on $w^l_{f(j)}$, then $\pi(s, q^*)$ is in $R_{k'}$ for some index $k' \in [z(i), z_{ij}]_R$, and further, any shortest path $\pi(s, w_{f(j)})$ from s to $w_{f(j)}$ is $\pi(s, q^*)$.*

The following lemma makes sure that when we process $w_{f(j')}$ for any other index j' of B_b with $j' \neq j$, we do not need to consider the regions $R_{k'}$ for $k' \in [z(i), z_{ij} - 1]$ if $z_{ij} \neq z(i)$.

▶ **Lemma 10.** *Suppose $z_{ij} \neq z(i)$. If q^* is on $w^l_{f(j')}$ for some $j' \in B_b$ and $j' \neq j$, then $\pi(s, q^*)$ is in $R_{k'}$ for some $k' \in [z_{ij}, z(j')]_R$.*

In order to compute the index z_{ij}, we will use a $\mathcal{R}$*-region range* query. Namely, given the index range $[z(i), z(j)]_R$ as well as $w_{f(j)}$, the query can be used to compute z_{ij}. In the full paper, we give a data structure that can answer each such query in $O(\log h \log n)$ time, after $O(n \log h)$ time and space preprocessing.

After $w_{f(j)}$ is processed as above, $q^l_{f(j)}$ is computed. By Lemma 10, to process $w_{f(j')}$ for other indices j' of $B_b \setminus \{j\}$, we only need to consider the indices of the regions of $\mathcal{R}$ after z_{ij}. Let $B'_1, B'_2, \ldots, B'_{g'-1}$ be the bundles in B_b other than the last one. For each $1 \leq b' \leq g' - 1$, if $b' = 1$, we call $path(B'_{b'}, z_{ij})$ recursively; otherwise, we call $path(B'_{b'}, z(i'))$ recursively, where i' is the last index of $B'_{b'-1}$.

After $w_{f(j)}$ is processed for each $j \in \mathbb{B}$, $q^l_{f(j)}$ is computed for every $j \in \mathbb{B}$; among these at most k points, we return the point q' whose value $d(s, q')$ is the smallest as q^*_l, which is q^* based on our above analysis (and also due to our assumption that q^* is on w^l_i for some $i \in [1, k]$). The total number of calls on the region-processing procedures is $O(k + h^*)$. The total number of $\mathcal{R}$-region range queries is $O(k)$ since each such query is for a composite bundle and there are at most k bundles in total. Hence, the total time of the algorithm is $O((h + k) \log h \log n)$. Recall that $k \leq K$.

We summarize our overall algorithm in the following theorem.

▶ **Theorem 11.** *Given $SPM(s)$, we can build a data structure of $O(n \log h + h^2)$ size in $O(n \log h + h^2 \log h)$ time, such that each quickest visibility query can be answered in $O((K + h) \log h \log n)$ time, where K is the size of the visibility polygon of the query point q.*

Proof. In the preprocessing, we compute the visibility polygon query data structure in [4] for computing $Vis(q)$, which is of $O(n + h^2)$ size and can be built in $O(n + h^2 \log h)$ time. The rest of the preprocessing work includes building the decomposition $\mathcal{D}$ and the segment query data structure of Theorem 3, performing the preprocessing for computing the map $f(\cdot)$, for the region-processing algorithms, for answering SP-segment-intersection queries, for answering $\mathcal{R}$-region range queries, etc; these work takes $O(n \log h)$ time and space in total.

Given any query point q, we first compute $Vis(q)$ in $O(K \log n)$ time by the query algorithm in [4]. Then, we obtain the extended window set W. Let $k = |W|$, which is $O(K)$. Next, we compute a closest point q^* on a segment of W in $O(k \log h \log n)$ time. To this end, we compute a set S of $O(k)$ candidate points as follows. We first add $q, q_1, \ldots, q_k$ to S. Then, we compute the closest point q^*_0 of $\overline{u_0 q_0}$ and add q^*_0 to S. Next we compute the point q^*_l in $O((k + h) \log h \log n)$ time by using our pruning algorithm in Sections 4.3 and 4.4. By a symmetric algorithm, we can also compute q^*_r. We add both q^*_l and q^*_r to S. By our analysis, q^* must be one of the points of S. Since $|S| = O(k)$, we can find q^* in S in additional $O(k \log n)$ time by using the shortest path map $SPM(s)$. ◀

In fact, we have the following more general result, which might have independent interest.

▶ **Corollary 12.** *Given $SPM(s)$, we can build a data structure of $O(n \log h)$ size in $O(n \log h)$ time, such that given $k = O(n)$ segments in $\mathcal{P}$ intersecting at the same point, we can compute a shortest path from s to all these segments in $O((k + h) \log h \log n)$ time.*

Proof. The preprocessing step is the same as in Theorem 11 except that the visibility polygon query data structure [4] is not necessary any more. Hence, the total preprocessing time and space is $O(n \log h)$. Given a set S of k segments intersecting at the same point, denoted by p, we break each segment at p to obtain two segments and we still use S to denote the new set of at most $2k$ segments. Next we compute a closest point p^* on the segments of S. To do so, we can apply the same algorithm as in Theorem 11 for computing q^* on the extended-windows of W. Indeed, the only key property of the segments of W we need is that all segments of W have a common endpoint at q. Now that all segments of S have a common endpoint p, the same algorithm still works. ◀

5 The Quickest Visibility Queries: The Improved Result

To further reduce the query time of Theorem 11 to $O(h \log h \log n)$, independent of K, the key idea is the following. First, we show that for any query point q, there exists a subset $\mathcal{S}(q)$ of $O(h)$ windows such that a closest point q^* is on a segment of $\mathcal{S}(q)$. This is done by making use of the extended corridor structure [3, 5]. Second, we give an algorithm that can compute $\mathcal{S}(q)$ in $O(h \log n)$ time, without computing $\mathit{Vis}(q)$, after $O(n \log h + h^2)$ space and $O(n \log h + h^2 \log h)$ time preprocessing. The result is obtained by modifying the query algorithm for computing $\mathit{Vis}(q)$ in [4]. Refer to our full paper [15] for all these details.

▶ **Theorem 13.** *Given $SPM(s)$, we can build a data structure of $O(n \log h + h^2)$ size in $O(n \log h + h^2 \log h)$ time, such that each quickest visibility query can be answered in $O(h \log h \log n)$ time.*

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. *Journal of Computational Geometry*, 7:77–100, 2016.

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

3 D. Z. Chen and H. Wang. L_1 shortest path queries among polygonal obstacles in the plane. In *Proc. of 30th Symposium on Theoretical Aspects of Computer Science*, pages 293–304, 2013.

4 D. Z. Chen and H. Wang. Visibility and ray shooting queries in polygonal domains. *Computational Geometry: Theory and Applications*, 48:31–41, 2015.

5 D. Z. Chen and H. Wang. Computing the visibility polygon of an island in a polygonal domain. *Algorithmica*, 77:40–64, 2017.

6 Y. K. Cheung and O. Daescu. Approximate point-to-face shortest paths in $\mathcal{R}^3$. arXiv:1004.1588, 2010.

7 Y.-J. Chiang and R. Tamassia. Optimal shortest path and minimum-link path queries between two convex polygons in the presence of obstacles. *International Journal of Computational Geometry and Applications*, 7:85–121, 1997.

8 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(1-4):209–233, 1987.

9 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.

10 J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

11 R. Khosravi and M. Ghodsi. The fastest way to view a query point in simple polygons. In *Proc. of the 24th European Workshop on Computational Geometry*, pages 187–190, 2005.

12 E. Melissaratos and D. Souvaine. Shortest paths help solve geometric optimization problems in planar regions. *SIAM Journal on Computing*, 21(4):601–638, 1992.

13 J. S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3(1):83–105, 1991.

14 J. S. B. Mitchell. Shortest paths among obstacles in the plane. *International Journal of Computational Geometry and Applications*, 6(3):309–332, 1996.

15 H. Wang. Quickest visibility queries in polygonal domains. arXiv:1703.03048, 2017.