Algorithms for Subpath Convex Hull Queries and Ray-Shooting Among Segments

Haitao Wang
Department of Computer Science, Utah State University, Logan, UT 84322, USA
haitao.wang@usu.edu

Abstract

In this paper, we first consider the subpath convex hull query problem: Given a simple path π of n vertices, preprocess it so that the convex hull of any query subpath of π can be quickly obtained. Previously, Guibas, Hershberger, and Snoeyink [SODA 90'] proposed a data structure of $O(n)$ space and $O(\log n \log\log n)$ query time; reducing the query time to $O(\log n)$ increases the space to $O(n \log\log n)$. We present an improved result that uses $O(n)$ space while achieving $O(\log n)$ query time. Like the previous work, our query algorithm returns a compact interval tree representing the convex hull so that standard binary-search-based queries on the hull can be performed in $O(\log n)$ time each. Our new result leads to improvements for several other problems.

In particular, with the help of the above result, we present new algorithms for the ray-shooting problem among segments. Given a set of n (possibly intersecting) line segments in the plane, preprocess it so that the first segment hit by a query ray can be quickly found. We give a data structure of $O(n \log n)$ space that can answer each query in $(\sqrt{n} \log n)$ time. If the segments are nonintersecting or if the segments are lines, then the space can be reduced to $O(n)$. All these are classical problems that have been studied extensively. Previously data structures of $\widetilde{O}(\sqrt{n})$ query time[1] were known in early 1990s; nearly no progress has been made for over two decades. For all problems, our results provide improvements by reducing the space of the data structures by at least a logarithmic factor while the preprocessing and query times are the same as before or even better.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Computational geometry

Keywords and phrases subpath hull queries, convex hulls, compact interval trees, ray-shooting, data structures

Digital Object Identifier 10.4230/LIPIcs.SoCG.2020.69

Related Version A full version of this paper is available at `https://arxiv.org/abs/2002.10672`.

1 Introduction

We first consider the subpath convex hull query problem. Let π be a simple path of n vertices in the plane. A *subpath hull query* specifies two vertices of π and asks for the convex hull of the subpath between the two vertices. The goal is to preprocess π so that the subpath hull queries can be answered quickly. Ideally, the query should return a representation of the convex hull so that standard queries on the hull can be performed in logarithmic time.

The problem has been studied by Guibas, Hershberger, and Snoeyink [18], who proposed a method of using compact interval trees. After $O(n \log n)$ time preprocessing, Guibas et al. [18] built a data structure of $O(n)$ space that can answer each query in $O(\log n \log\log n)$ time. Their query algorithm returns a compact interval tree that represents the convex hull so that all binary-search-based queries on the hull can be performed in $O(\log n)$ time each. The queries on the hull include (but are not limited to) the following: find the most extreme vertex of the convex hull along a query direction; find the intersection between a query

[1] The notation $\widetilde{O}$ suppresses a polylogarithmic factor.

line and the convex hull; find the common tangents from a query point to the convex hull; determine whether a query point is inside the convex hull, etc. Guibas et al. [18] reduced the subpath hull query time to $O(\log n)$ but the space becomes $O(n \log\log n)$. A trade-off was also made with $O(\log n \log^* n)$ query time and $O(n \log^* n)$ space [18].

As compact interval trees are quite amenable, the results of Guibas et al. [18] have found many applications, e.g., [4, 9–13, 25]. Clearly, there is still some room for improvement on the results of Guibas et al. [18]; the ultimate goal might be an $O(n)$ space data structure with $O(\log n)$ query time. We achieve this goal. The preprocessing time of our data structure is $O(n)$, after the vertices of π are sorted by x-coordinate. Like the results of Guibas et al. [18], our query algorithm also returns a compact interval tree that can support logarithmic time queries for all binary-search-based queries on the convex hull of the query subpath; the edges of the convex hull can be retrieved in time linear in the number of vertices of the convex hull. Note that like those in [18] our results are for the random access machine (RAM) model.

With our new result, previous applications that use the results of Guibas et al. [18] can now be improved accordingly. These include the problem of enclosing polygons by two minimum area rectangles [4, 5], computing a guarding set for simple polygons in wireless location [12], computing optimal time-convex hulls [13], L_1 top-k weighted sum aggregate nearest and farthest neighbor searching [25], etc. For all these problems, we reduce the space of their algorithms by a $\log\log n$ factor while the time complexities are the same as before or even better. See the full paper for the details of our improvements.

Wagener [24] proposed a parallel algorithm for computing a data structure, called *bridge tree*, for representing the convex hull of a simple path π. If using one processor, for any query subpath of π, Wagener [24] showed that the bridge tree can be used to answer decomposable queries on the convex hull of the query subpath in logarithmic time each. Wagener [24] claimed that some non-decomposable queries can also be handled; however no details were provided. In contrast, our approach returns a compact interval tree that is more amenable (indeed, the bridge trees [24] were mainly designed for parallel processing) and can support both decomposable and non-decomposable queries. In addition, if one wants to output the convex hull of the query subpath, our approach can do so in time linear in the number of the vertices of the convex hull while the method of Wagener [24] needs $O(n)$ time.

1.1 Ray-shooting

With the help of our above result and other new techniques, we present improved results for several classical ray-shooting problems. Previously, data structures of $\widetilde{O}(\sqrt{n})$ query time and near-linear space were known in early 1990s; nearly no progress has been made for over two decades. Our results reduce the space by at least a logarithmic factor while achieving the same or better preprocessing and query times. In the following, we use $O(T(n), S(n), Q(n))$ to represent the complexity of a data structure, where $T(n)$ is the preprocessing time, $S(n)$ is the space, and $Q(n)$ is the query time. We will confine the discussion of the previous work to data structures of linear or near-linear space. Refer to Table 1 for a summary. Throughout the paper, we use δ to refer to an arbitrarily small positive constant.

Ray-shooting among lines. Given a set of n lines in the plane, the problem is to build a data structure so that the first line hit by a query ray can be quickly found.

Bar-Yehuda and Fogel [3] gave a data structure of complexity $O(n^{1.5}, n\log^2 n, \sqrt{n}\log n)$. Cheng and Janardan [11] gave a data structure of complexity $O(n^{1.5}\log^2 n, n\log n, \sqrt{n}\log n)$. Agarwal and Sharir [2] developed a data structure of complexity $O(n\log n, n\log n, n^{1/2+\delta})$.

Table 1 Summary of the results. The big-O notation is omitted. δ can be any small positive constant. The results marked with * hold with high probability (except that the result of Chan [6] is expected).

	Preprocessing time	Space	Query time	Source
Ray-shooting among lines	$n^{1.5}$	$n\log^2 n$	$\sqrt{n}\log n$	BF [3]
	$n^{1.5}\log^2 n$	$n\log n$	$\sqrt{n}\log n$	CJ [11]
	$n\log n$	$n\log n$	$n^{0.5+\delta}$	AS [2]
	$n^{1.5}$	n	$\sqrt{n}\log n$	this paper
	$n\log n$	n	$\sqrt{n}\log n$ *	this paper
Intersection detection	$n^{1.5}\log^2 n$	$n\log n$	$\sqrt{n}\log n$	CJ [11]
	$n^{1.5}$	n	$\sqrt{n}\log n$	this paper
	$n\log n$	n	$\sqrt{n}\log n$ *	this paper
Ray-shooting among intersecting segments	$n\alpha(n)\log^3 n$	$n\log^2 n$	$n^{0.695}\log n$	OSS [23]
	$n\alpha(n)\log^3 n$	$n\alpha(n)$	$n^{2/3+\delta}$	GOS [19]
	$n^{1.5}\log^{4.33} n$	$n\alpha(n)\log^4 n$	$\sqrt{n\alpha(n)}\log^2 n$	A [1]
	$(n\alpha(n))^{1.5}$	$n\alpha(n)\log^2 n$	$\sqrt{n\alpha(n)}\log n$	BF [3]
	$n^{1.5}\log^2 n$	$n\log^2 n$	$\sqrt{n}\log n$	CJ [11]
	$n\log^2 n$	$n\log^2 n$	$n^{0.5+\delta}$	AS [2]
	$n\log^3 n$	$n\log^2 n$	$\sqrt{n}\log^2 n$ *	C [6]
	$n^{1.5}$	$n\log n$	$\sqrt{n}\log n$	this paper
	$n\log^2 n$	$n\log n$	$\sqrt{n}\log n$ *	this paper
Ray-shooting among nonintersecting segments	$n\log n$	n	$n^{0.695}\log n$	OSS [23]
	$n^{1.5}\log^{4.33} n$	$n\alpha(n)\log^3 n$	$\sqrt{n}\log^2 n$	A [1]
	$n^{1.5}$	$n\log n$	$\sqrt{n}\log n$	BF [3]
	$n^{1.5}$	n	$\sqrt{n}\log n$	this paper
	$n\log n$	n	$\sqrt{n}\log n$ *	this paper

By using our subpath hull query data structure and a result from Chazelle and Guibas [7], we present a new data structure of complexity $O(n^{1.5}, n, \sqrt{n}\log n)$.

In addition, we also consider a more general *first-k-hits* query, i.e., given a query ray and an integer k, report the first k lines hit by the ray. This problem was studied by Bar-Yehuda and Fogel [3], who gave a data structure of complexity $O(n^{1.5}, n\log^2 n, \sqrt{n}\log n + k\log^2 n)$. Our new result is a data structure of complexity $O(n^{1.5}, n, \sqrt{n}\log n + k\log n)$.

Intersection detection. Given a set of n line segments in the plane, the problem is to build a data structure to determine whether a query line intersects at least one segment. Cheng and Janardan [11] gave a data structure of complexity $O(n^{1.5}\log^2 n, n\log n, \sqrt{n}\log n)$. By adapting the interval partition trees of Overmars et al. [23] to the partition trees of Matoušek [20, 21], we obtain a data structure of complexity $O(n^{1.5}, n, \sqrt{n}\log n)$.

Ray-shooting among segments. Given a set of n (possibly intersecting) line segments in the plane, we want to build a data structure to find the first segment hit by a query ray.

The result of Overmars et al. [23] has complexity $O(n\alpha(n)\log^3 n, n\log^2 n, n^{0.695}\log n)$, where $\alpha(n)$ is the inverse Ackermann's function. Guibas et al. [19] presented a data structure of complexity $O(n\alpha(n)\log^3 n, n\alpha(n), n^{2/3+\delta})$. Agarwal [1] gave a data structure of complexity $O(n^{1.5}\log^{4.33} n, n\alpha(n)\log^4 n, \sqrt{n\alpha(n)}\log^2 n)$. Bar-Yehuda and Fogel [3] gave a data structure

of complexity $O((n\alpha(n))^{1.5}, n\alpha(n)\log^2 n, \sqrt{n\alpha(n)}\log n)$. Cheng and Janardan [11] developed a data structure of complexity $O(n^{1.5}\log^2 n, n\log^2 n, \sqrt{n}\log n)$. Agarwal and Sharir [2] proposed a data structure of complexity $O(n\log^2 n, n\log^2 n, n^{0.5+\delta})$. Chan's randomized result [6] has complexity $O(n\log^3 n, n\log^2 n, \sqrt{n}\log^2 n)$, where the query time is expected.

Cheng and Janardan's algorithm [11] relies on their results for the ray-shooting problem among lines and the intersection detection problem. Following their algorithmic scheme and using our above new results for these two problems, we obtain a data structure for the ray-shooting problem among segments with complexity $O(n^{1.5}, n\log n, \sqrt{n}\log n)$. This is the first data structure of $\widetilde{O}(\sqrt{n})$ query time that uses only $O(n\log n)$ space.

If the segments are nonintersecting, Overmars et al. [23] gave a data structure of complexity $O(n\log n, n, n^{0.695}\log n)$. Agarwal [1] presented a data structure of complexity $O(n^{1.5}\log^{4.33} n, n\alpha(n)\log^3 n, \sqrt{n}\log^2 n)$. Bar-Yehuda and Fogel [3] proposed a data structure of complexity $O(n^{1.5}, n\log n, \sqrt{n}\log n)$. Our result has complexity $O(n^{1.5}, n, \sqrt{n}\log n)$.

Randomized results. Using Chan's randomized techniques [6], the preprocessing time of all our above results can be reduced to $O(n\log n)$ (except $O(n\log^2 n)$ time for the ray-shooting problem among intersecting segments), while the same query time complexities hold with high probability (i.e., probability at least $1 - 1/n^c$ for any large constant c).

Outline. Section 2 reviews some previous work of the subpath hull queries; Section 3 presents our new data structure for the problem. Section 4 solves the ray-shooting problem.

2 Preliminaries

Let $p_1, \ldots, p_n$ be the vertices of a simple path π ordered along π. For any two indices i and j with $1 \le i \le j \le n$, we use $\pi(i, j)$ to refer to the subpath of π from p_i to p_j. Given a pair (i, j) of indices with $1 \le i \le j \le n$, the *subpath hull query* asks for the convex hull of $\pi(i, j)$.

The convex hull of a simple path can be found in linear time, e.g., [17, 22]. Note that the convex hull of a simple path is the same as the convex hull of its vertices. For this reason, in our discussion a subpath π' of π actually refers to its vertex set. For each subpath π' of π, we use $|\pi'|$ to denote the number of vertices of π'.

For any set P of points in the plane, let $H(P)$ denote the convex hull of P. Denote by $H_U(P)$ and $H_L(P)$ the upper and lower hulls, respectively.

Interval trees. Let S be a set of n points in the plane. The *interval tree* $T(S)$ is a complete binary tree whose leaves from left to right correspond to the points of S sorted from left to right. Each internal node corresponds to the interval between the rightmost leaf in its left subtree and the leftmost leaf in its right subtree. We say that a segment joining two points of S *spans* an internal node v if the projection of the interval of v on the x-axis is contained in the projection of the segment on the x-axis.

We store each edge e of the upper hull $H_U(S)$ at the highest node of $T(S)$ that e spans (e.g., see Fig. 1). By also storing the edges of the lower hull $H_L(S)$ in $T(S)$ in the same way, we can answer all standard binary-search-based queries on the convex hull $H(S)$ in $O(\log n)$ time, by following a path from the root of $T(S)$ to a leaf (see Lemma 4.1 of [18] for details).

Compact interval trees. As the size of $T(S)$ is $\Theta(n)$ while $|H(S)|$ may be much smaller than n, where $|H(S)|$ is the number of edges of $H(S)$, using $T(S)$ to store $H(S)$ may not be space-efficient. Guibas et al. [18] proposed to use a *compact interval tree* $T_U(S)$ of $O(|H_U(S)|)$

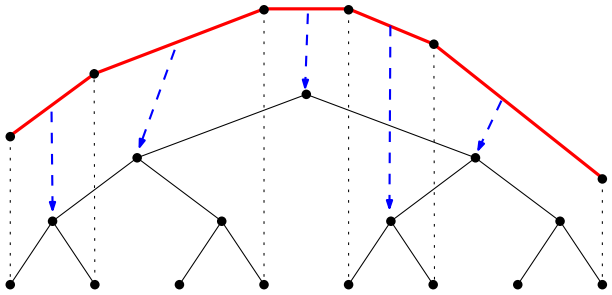

Figure 1 Illustrating an interval tree that stores upper hull edges.

size to store $H_U(S)$, as follows. In $T(S)$, a node v is *empty* if it does not store an edge of $H_U(S)$; otherwise it is *full*. It was shown in [18] that if two nodes of $T(S)$ are full, then their lowest common ancestor is also full. We remove empty nodes from $T(S)$ by relinking the tree to make each full node the child of its nearest full ancestor. Let $T_U(S)$ be the new tree and we still use $T(S)$ to refer to the original interval tree without storing any hull edges. Each node of $T_U(S)$ stores exactly one edge of $H_U(S)$, and thus $T_U(S)$ has $|H_U(S)|$ nodes. After $O(n)$ time preprocessing on $T(S)$, $T_U(S)$ can be computed from $H_U(S)$ in $O(|H_U(S)|)$ time (see Lemma 4.4 in [18]). Similarly, we use a compact interval tree $T_L(S)$ of $|H_L(S)|$ nodes to store $H_L(S)$. Then, using the three trees $T_U(S)$, $T_L(S)$, and $T(S)$, all standard binary-search-based queries on $H(S)$ can be answered in $O(\log n)$ time. The main idea is that the algorithm walks down through the compact interval trees while keeping track of the corresponding position in $T(S)$ (see Lemma 4.3 [18] for details). We call $T(S)$ a *reference tree*. In addition, using $T_U(S)$ and $T_L(S)$, $H(S)$ can be output in $O(|H(S)|)$ time.

As discussed above, to represent $H(S)$, we need two compact interval trees, one for $H_U(S)$ and the other for $H_L(S)$. To make our discussion more concise, we will simply say "the compact interval tree" for S and use $T^+(S)$ to refer to it, which actually includes two trees.

Compact interval trees for π. Consider two *consecutive* subpaths π_1 and π_2 of π. Suppose their compact interval trees $T^+(\pi_1)$ and $T^+(\pi_2)$ and the interval tree $T(\pi)$ of π are available. We know that $H(\pi_1)$ and $H(\pi_2)$ have at most two common tangents [7]. Using the path-copying method of persistent data structures [14], Guibas et al. [18] obtained the following.

▶ **Lemma 1** (Guibas et al. [18]). *Without altering $T^+(\pi_1)$ and $T^+(\pi_2)$, the compact interval tree $T^+(\pi_1 \cup \pi_2)$ can be produced in $O(\log n)$ time and $O(\log n)$ additional space.*

▶ **Lemma 2** (Guibas et al. [18]). *Given the interval tree $T(\pi)$, with $O(n)$ time preprocessing, we can compute $T^+(\pi')$ for any subpath π' of π in $O(|\pi'|)$ time.*

3 Subpath convex hull queries

We present our new data structure for subpath hull queries. We first sort all vertices of π by x-coordinate. The rest of the preprocessing of our data structure takes $O(n)$ time in total.

3.1 A decomposition tree

After having the interval tree $T(\pi)$, we construct a *decomposition tree* $\Psi(\pi)$, which is a segment tree on the vertices of π following their order along π. Specifically, $\Psi(\pi)$ is a complete binary tree with n leaves corresponding to the vertices of π in order along π. Each internal

node v of $\Psi(\pi)$ corresponds to the subpath $\pi(a_v, b_v)$, where a_v (resp., b_v) is defined to be the index of the vertex of π corresponding to the leftmost (resp., rightmost) leaf of the subtree of $\Psi(\pi)$ rooted at v; we call $\pi(a_v, b_v)$ a *canonical subpath* of π and use $\pi(v)$ to denote it.

Next, we remove some nodes in the lower part of $\Psi(\pi)$, as follows. For each node v whose canonical path has at most $\log^2 n$ vertices and whose parent canonical subpath has more than $\log^2 n$ vertices, we remove both the left and the right subtrees of v from $\Psi(\pi)$ but explicitly store $\pi(v)$ at v, after which v becomes a leaf of the new tree. From now on we use $\Psi(\pi)$ to refer to the new tree. It is not difficult to see that $\Psi(\pi)$ now has $O(n/\log^2 n)$ nodes.

We then compute compact interval trees $T^+(\pi(v))$ for all nodes v of $\Psi(\pi)$ in a bottom-up manner. Specifically, if v is a leaf, then $\pi(v)$ has at most $\log^2 n$ vertices, and we compute $T^+(\pi(v))$ from scratch, which takes $O(\log^2 n)$ time by Lemma 2. If v is not a leaf, then $T^+(\pi(v))$ can be obtained by merging the two compact interval trees of its children, which takes $O(\log n)$ time by Lemma 1. In this way, computing compact interval trees for all nodes of $\Psi(\pi)$ takes $O(n)$ time in total, for $\Psi(\pi)$ has $O(n/\log^2 n)$ nodes.

3.2 A preliminary query algorithm

Consider a subpath hull query (i, j). We first present an $O(\log^2 n)$ time query algorithm using $\Psi(\pi)$ and then reduce the time to $O(\log n)$. Depending on whether the two vertices p_i and p_j are in the same canonical subpath of a leaf of $\Psi(\pi)$, there are two cases.

Case 1. If yes, let v be the leaf. Then, $\pi(i, j)$ is a subpath of $\pi(v)$ and thus has at most $\log^2 n$ vertices. We compute $T^+(\pi(i, j))$ from scratch in $O(\log^2 n)$ time by Lemma 2.

Case 2. Otherwise, let v be the leaf of $\Psi(\pi)$ whose canonical subpath contains p_i and u the leaf whose canonical subpath contains p_j. Let w be the lowest common ancestor of u and v. As in [18], we partition $\pi(i, j)$ into two subpaths $\pi(i, k)$ and $\pi(k + 1, j)$, where $k = b_{w'}$ with w' being the left child of w (recall the definition of $b_{w'}$ given before). We will compute the compact interval trees for the two subpaths separately, and then merge them to obtain $T^+(\pi(i, j))$ in additional $O(\log n)$ time by Lemma 1. We only discuss how to compute $T^+(\pi(i, k))$, for the other tree can be computed likewise.

We partition $\pi(i, k)$ into two subpaths $\pi(i, b_v)$ and $\pi(b_v + 1, k)$. We will compute the trees for them separately and then merge the two trees to obtain $T^+(\pi(i, k))$.

For computing $T^+(\pi(i, b_v))$, as $\pi(i, b_v)$ is a subpath of $\pi(v)$, it has at most $\log^2 n$ vertices. Hence, we can compute $T^+(\pi(i, b_v))$ from scratch in $O(\log^2 n)$ time.

For $T^+(\pi(b_v + 1, k))$, observe that $\pi(b_v + 1, k)$ is the concatenation of canonical subpaths of $O(\log n)$ nodes of $\Psi(\pi)$; precisely, these nodes are the right children of their parents that are in the path of $\Psi(\pi)$ from v's parent to w' and these nodes themselves are not on the path. Since the compact interval trees of these nodes are already computed in the preprocessing, we can produce $T^+(\pi(b_v + 1, k))$ in $O(\log^2 n)$ time by merging these trees.

3.3 Reducing the query time to $O(\log n)$

We now reduce the query time to $O(\log n)$, with additional preprocessing (but still $O(n)$).

To reduce the time for Case 1, we perform the following preprocessing. For each leaf v of $\Psi(\pi)$, we preprocess the path $\pi(v)$ in the same way as above for preprocessing π. This means that we construct an interval tree $T(\pi(v))$ as well as a decomposition tree $\Psi(\pi(v))$ for the subpath $\pi(v)$. To answer a query for Case 1, we instead use $\Psi(\pi(v))$ (and use $T(\pi(v))$ as the reference tree). The query time becomes $O(\log^2 \log n)$ as $|\pi(v)| \leq \log^2 n$. Note that to construct $T(\pi(v))$ and $\Psi(\pi(v))$ in $O(|\pi(v)|)$ time, we need to sort all vertices of $\pi(v)$ by

x-coordinate in $O(|\pi(v)|)$ time. Recall that we already have a sorted list of all vertices of π, from which we can obtain sorted lists for $\pi(v)$ for all leaves v of $\Psi(\pi)$ in $O(n)$ time altogether. Hence, the preprocessing for $\pi(v)$ for all leaves v of $\Psi(\pi)$ takes $O(n)$ time.

We proceed to Case 2. To reduce the query time to $O(\log n)$, we will discuss how to perform additional preprocessing so that $T^+(\pi(i,k))$ can be computed in $O(\log n)$ time. Computing $T^+(\pi(k+1,j))$ can be done in $O(\log n)$ time similarly. Finally we can merge the two trees to obtain $T^+(\pi(i,j))$ in additional $O(\log n)$ time by Lemma 1.

To compute $T^+(\pi(i,k))$ in $O(\log n)$ time, according to our algorithm it suffices to compute both $T^+(i,b_v)$ and $T^+(b_v+1,k)$ in $O(\log n)$ time. We discuss $T^+(i,b_v)$ first.

Dealing with $T^+(\pi(i,b_v))$. To compute $T^+(i,b_v)$ in $O(\log n)$ time, we preform the following additional preprocessing. For each leaf v of $\Psi(\pi)$, recall that $|\pi(v)| \le \log^2 n$; we partition $\pi(v)$ into $t_v \le \log n$ subpaths each of which contains at most $\log n$ vertices. We use $\pi_v(1), \pi_v(2), \ldots, \pi_v(t_v)$ to refer to these subpaths in order along $\pi(v)$. For each subpath $\pi_v(i)$, we compute $T^+(\pi_v(i))$ from scratch in $O(\log n)$ time. The total time for computing all such trees is $O(\log^2 n)$. Next, we compute compact interval trees for t_v *prefix subpaths* of $\pi(v)$. Specifically, for each $t \in [1,t_v]$, we compute $T^+(\pi_v[1,t])$, where $\pi_v[1,t]$ is the concatenation of the paths $\pi_v(1), \pi_v(2), \ldots, \pi_v(t)$. This can be done in $O(\log^2 n)$ time by computing $T^+(\pi_v[1,t])$ incrementally for $t = 1,2,\ldots,t_v$ using the merge algorithm of Lemma 1. Indeed, initially $T^+(\pi_v[1,t]) = T^+(\pi_v(1))$, which is already available. Then, for each $2 \le t \le t_v$, $T^+(\pi_v[1,t])$ can be produced by merging $T^+(\pi_v[1,t-1])$ and $T^+(\pi_v(t))$ in $O(\log n)$ time. Similarly, we compute compact interval trees for t_v *suffix subpaths* of $\pi(v)$: $T^+(\pi_v[t,t_v])$ for all $t = 1,2,\ldots,t_v$, where $\pi_v[t,t_v]$ is the concatenation of the paths $\pi_v(t), \pi_v(t+1), \ldots, \pi_v(t_v)$. This can be done in $O(\log^2 n)$ time by a similar algorithm as above. Thus, the preprocessing on v takes $O(\log^2 n)$ time; the preprocessing on all leaves of $\Psi(\pi)$ takes $O(n)$ time in total.

We can now compute $T^+(i,b_v)$ in $O(\log n)$ time as follows. Recall that $\pi(i,b_v)$ is a subpath of $\pi(v)$ and b_v is the last vertex of $\pi(v)$. We first determine the subpath $\pi_v(t)$ that contains i. Let g be the last vertex of $\pi_v(t)$. We partition $\pi(i,b_v)$ into two subpaths $\pi(i,g)$ and $\pi(g+1,b_v)$, and we will compute their compact interval trees separately and then merge them to obtain $T^+(\pi(i,b_v))$. For $\pi(i,g)$, as $\pi(i,g)$ is a subpath of $\pi_v(t)$ and $|\pi_v(t)| \le \log n$, we can compute $T^+(\pi(i,g))$ from scratch in $O(\log n)$ time. For $\pi(g+1,b_v)$, observe that $\pi(g+1,b_v)$ is exactly the suffix supath $\pi_v[t+1,t_v]$, whose compact interval tree has already been computed in the preprocessing. Hence, $T^+(i,b_v)$ can be produced in $O(\log n)$ time.

Dealing with $T^+(\pi(b_v+1,k))$. To compute $T^+(b_v+1,k)$ in $O(\log n)$ time, we perform the following preprocessing, which was also used by Guibas et al. [18]. Recall that $\pi(b_v+1,k)$ is the concatenation of the canonical paths of $O(\log n)$ nodes that are right children of the nodes on the path in $\Psi(\pi)$ from v's parent to the left child of w (and these nodes themselves are not on the path). Hence, this sequence of nodes can be uniquely determined by the leaf-ancestor pair (v,w); we use $\pi_{v,w}$ to denote the above concatenated subpath of π.

Correspondingly, in the preprocessing, for each leaf v we do the following. For each ancestor w of v, we compute the compact interval tree for the subpath $\pi_{v,w}$. As v has $O(\log n)$ ancestors, computing the trees for all ancestors takes $O(\log^2 n)$ time using the merge algorithm of Lemma 1. Hence, the total preprocessing time on v is $O(\log^2 n)$, and thus the total preprocessing time on all leaves of $\Psi(\pi)$ is $O(n)$, for $\Psi(\pi)$ has $O(n/\log^2 n)$ leaves. Due to the above preprocessing, $T^+(b_v+1,k)$ is available during queries.

Wrapping up. In summary, the query time is $O(\log n)$. Comparing with the method of Guibas et al. [18], our innovation is threefold. First, we process subpaths individually to handle queries of Case 1. Second, we precompute compact interval trees for convex hulls of the prefix and suffix subpaths of $\pi(v)$ for each leaf v of $\Psi(\pi)$. Third, we use a smaller decomposition tree $\Psi(\pi)$ of only $O(n/\log^2 n)$ nodes. Theorem 3 summarizes our result.

▶ **Theorem 3.** *After all vertices of π are sorted by x-coordinate, a data structure of $O(n)$ space can be built in $O(n)$ time so that each subpath hull query can be answered in $O(\log n)$ time. The query algorithm produces a compact interval tree representing the convex hull of the query subpath, which can support all binary-search-based operations on the convex hull in $O(\log n)$ time each. These operations include (but are not limited to) the following (let π' denote the query subpath and let $H(\pi')$ be its convex hull):*

1. *Given a point, decide whether the point is in $H(\pi')$.*
2. *Given a point outside $H(\pi')$, find the two tangents from the point to $H(\pi')$.*
3. *Given a direction, find the most extreme point of π' along the direction.*
4. *Given a line, find its intersection with $H(\pi')$.*
5. *Given a convex polygon (represented in a data structure supporting binary search), decide whether it intersects $H(\pi')$, and if not, find their common tangents (both outer and inner).*

In addition, $H(\pi')$ can be output in time linear in the number of vertices of $H(\pi')$.

4 Ray-shooting

The ray-shooting problem among lines is discussed in Section 4.1. Section 4.2 is concerned with the intersection detection problem and the ray-shooting problem among segments.

4.1 Ray-shooting among lines

Given a set of n lines in the plane, we wish to build a data structure so that the first line hit by a query ray can be found efficiently. The problem is usually tackled in the dual plane, e.g., [11]. Let P be the set of dual points of the lines. In the dual plane, the problem is equivalent to the following: Given a query line l_q, a pivot point $q \in l_q$, and a rotation direction (clockwise or counterclockwise), find the first point of P hit by rotating l_q around q.

A spanning path $\pi(P)$ of P is a polygonal path connecting all points of P such that P is the vertex set of the path. Hence, $\pi(P)$ corresponds to a permutation of P. For any line l in the plane, let $\sigma(l)$ denote the number of edges of $\pi(P)$ crossed by l. The *stabbing number* of $\pi(P)$ is the largest $\sigma(l)$ of all lines l in the plane. It is known that a spanning path of P with stabbing number $O(\sqrt{n})$ always exists [8], which can be computed in $O(n^{1+\delta})$ time using Matoušek's partition tree [21] (e.g., by a method in [8]). Let $\pi'(P)$ denote such a path. Note that $\pi'(P)$ may have self-intersections. Using $\pi'(P)$, Edelsbrunner et al. [15] gave an algorithm that can produce another spanning path $\pi(P)$ of P such that the stabbing number of $\pi(P)$ is also $O(\sqrt{n})$ and $\pi(P)$ has no self-intersections (i.e., $\pi(P)$ is a simple path); the runtime of the algorithm is $O(n^{1.5})$. Below we will use $\pi(P)$ to solve our problem.

▶ **Lemma 4** (Chazelle and Guibas [7])**.** *We can build a data structure of $O(n)$ size in $O(n \log n)$ time for any simple path of n vertices, so that given any query line l_q, if l_q intersects the path in k edges, then these edges can be found in $O(k \log \frac{n}{k})$ time.*

We first build the data structure in Lemma 4 for $\pi(P)$. Then, we construct the subpath hull query data structure of Theorem 3 for $\pi(P)$. This finishes our preprocessing.

Given a query line l_q, along with the pivot q and the rotation direction, we first use Lemma 4 to find the edges of $\pi(P)$ intersecting l_q. As the stabbing number of $\pi(P)$ is $O(\sqrt{n})$, this steps finds $O(\sqrt{n})$ edges intersecting l_q in $O(\sqrt{n}\log n)$ time. Then, using these edges we can partition $\pi(P)$ into $O(\sqrt{n})$ subpaths each of which does not intersect l_q. For each subpath, we use our subpath hull query data structure to compute its convex hull in $O(\log n)$ time. Next, we compute the tangents from the pivot q to each of these $O(\sqrt{n})$ convex hulls, in $O(\log n)$ time each by Theorem 3. Using these $O(\sqrt{n})$ tangents, based on the rotation direction of l_q, we can determine the first point of P hit by l_q in additional $O(\sqrt{n})$ time. Hence, the total time of the query algorithm is $O(\sqrt{n}\log n)$.

▶ **Theorem 5.** *There exists a data structure of complexity $O(n^{1.5}, n, \sqrt{n}\log n)$ for the ray-shooting problem among lines. The preprocessing time can be reduced to $O(n\log n)$ time by a randomized algorithm while the query time is bounded by $O(\sqrt{n}\log n)$ with high probability.*

Proof. The deterministic result has been discussed above. For the randomized result, Chan [6] gave an $O(n\log n)$ time randomized algorithm to compute a spanning path $\pi''(P)$ for P such that $\pi''(P)$ is a simple path and the stabbing number of $\pi''(P)$ is at most $O(\sqrt{n})$ with high probability. After having $\pi''(P)$, we build the data structure for Lemma 4 and the subpath hull query data structure. Hence, the preprocessing takes $O(n\log n)$ time and $O(n)$ space, and the query time is bounded by $O(\sqrt{n}\log n)$ with high probability. ◀

We can extend the algorithm to obtain the result for the first-k-hit queries. The details are omitted but can be found in the full paper.

4.2 Intersection detection and ray-shooting among segments

Given a set S of n segments in the plane, an intersection detection query asks whether a query line intersects at least one segment of S. One motivation to study the problem is that it is a subproblem in our algorithm for the ray-shooting problem among segments.

To find a data structure to store the segments of S, we adapt the techniques of Overmars et al. [23] to the partition trees of Matoušek [20, 21] (to obtain the deterministic result) as well as that of Chan [6] (to obtain the randomized result). To store segments, Overmars et al. [23] used a so-called *interval partition tree*, whose underling structure is a conjugation tree of Edelsbrunner and Welzl [16]. The idea is quite natural due to the nice properties of conjugation trees: Each parent region is partitioned into exactly two disjoint children regions by a line. The drawback of conjugation trees is the slow $\widetilde{O}(n^{0.695})$ query time. When adapting the techniques to more query-efficient partition trees such as those in [6, 20, 21], two issues arise. First, each parent region may have more than two children. Second, children regions may overlap. Chan's partition tree [6] does not have the second issue while both issues appear in Matoušek's partition trees [20, 21]. As a matter of fact, the second issue incurs a much bigger challenge. In the following, we first present our randomized result by using Chan's partition tree [6], which is relatively easy, and then discuss the more complicated deterministic result using Matoušek's partition trees [20, 21].

We begin with the following lemma, which solves a special case of the problem. The lemma will be needed in both our randomized and deterministic results.

▶ **Lemma 6.** *Suppose all segments of S intersect a given line segment.*

1. *We can build a data structure of $O(n)$ space in $O(n\log n)$ time so that whether a query line intersects any segment of S can be determined in $O(\log n)$ time.*
2. *If the segments of S are nonintersecting, we can build a data structure of $O(n)$ space in $O(n\log n)$ time so that the first segment hit by a query ray can be found in $O(\log n)$ time.*

4.2.1 The randomized result

We briefly review Chan's partition tree [6] (for simplicity we only discuss it in 2D, which suffices for our problem). Chan's tree for a set P of n points, denoted by T, is a hierarchical structure by recursively subdividing the plane into triangles. Each node v of T corresponds to a triangle, denoted by $\triangle(v)$. If v is the root, then $\triangle(v)$ is the entire plane. If v is not a leaf, then v has $O(1)$ children whose triangles form a disjoint partition of $\triangle(v)$. Define $P(v) = P \cap \triangle(v)$. The set $P(v)$ is not explicitly stored at v unless v is a leaf, in which case $|P(v)| = O(1)$. The height of T is $O(\log n)$. Let $\kappa(T)$ denote the maximum number of triangles of T that are crossed by any line in the plane. Chan [6] gave an $O(n \log n)$ time randomized algorithm to compute T such that $\kappa(T)$ is at most $O(\sqrt{n})$ with high probability.

Let P be the set of the endpoints of all segments of S (so $|P| = 2n$). We first build the tree T as above. We then store the segments of S in T, as follows. For each segment s, we do the following. Starting from the root of T, for each node v, we assume that s is contained in $\triangle(v)$, which is true when v is the root. If v is a leaf, then we store s at v; let $S(v)$ denote all segments stored at v. If v is not a leaf, then we check whether s is in $\triangle(u)$ for a child u of v. If yes, we proceed on u. Otherwise, for each child u, for each edge e of $\triangle(u)$, if s intersects e, then we store s at the edge e (in this case we do not proceed to the children of u); denote by $S(e)$ the set of edges stored at e. This finishes the algorithm for storing s. As each node of T has $O(1)$ children, s is stored $O(1)$ times and the algorithm runs in $O(\log n)$ time. In this way, it takes $O(n \log n)$ time to store all segments of S, and the total sum of $|S(e)|$ and $|S(v)|$ for all triangle edges e and all leaves v is $O(n)$. In addition, $|S(v)| = O(1)$ for any leaf v, since $|P(v)| = O(1)$ and both endpoints of each segment $s \in S(v)$ are in $P(v)$.

Next, for each triangle edge e, since all edges of $S(e)$ intersect e, we preprocess $S(e)$ using Lemma 6(1). Doing this for all triangle edges e takes $O(n \log n)$ time and $O(n)$ space.

Consider a query line l. Our goal is to decide whether l intersects a segment of S. Starting from the root, we determine the set of nodes v whose triangles $\triangle(v)$ are crossed by l. For each such node v, if v is a leaf, then we check whether s intersects l for each segment $s \in S(v)$; otherwise, for each edge e of $\triangle(v)$, we use the query algorithm of Lemma 6(1) to determine whether l intersects any segment of $S(e)$. As the number of nodes v whose triangles $\triangle(v)$ crossed by l is at most $\kappa(T)$ and $S(v) = O(1)$ for each leaf v, the total time of the query algorithm is $O(\kappa(T) \cdot \log n)$. The algorithm correctness is discussed in the proof of Theorem 7.

▶ **Theorem 7.** *There exists a data structure of complexity $O(n \log n, n, \sqrt{n} \log n)$ for the intersection detection problem, where the query time holds with high probability.*

Proof. We have discussed the preprocessing time and space. Since the query time is $O(\kappa(T) \cdot \log n)$ and $\kappa(T)$ is at most $O(\sqrt{n})$ with high probability, the query time is bounded by $O(\sqrt{n} \log n)$ with high probability. For the correctness of the query algorithm, suppose l intersects a segment s, say, at a point p. If s is stored at $S(v)$ for a leaf v, then l must cross $\triangle(v)$ and thus our algorithm will detect the intersection. Otherwise, s must be stored in $S(e)$ for an edge e of a triangle $\triangle(u)$ that contains p. Since $p \in l$, l must cross $\triangle(u)$. According to our query algorithm, the query algorithm of Lemma 6(1) will be invoked on $S(e)$, and thus the algorithm will report the existence of intersection. ◀

If the segments of S are nonintersecting, by replacing Lemma 6(1) with Lemma 6(2) in both the above preprocessing and query algorithms, we can obtain the following result.

▶ **Theorem 8.** *There exists a data structure of complexity $O(n \log n, n, \sqrt{n} \log n)$ for the ray-shooting among nonintersecting segments, where the query time holds with high probability.*

To solve the ray-shooting problem among (possibly intersecting) segments, as discussed in Section 1.1, using our results in Theorems 5 and 7 and following the algorithmic scheme of Cheng and Janardan [11], we can obtain Theorem 9 (see the full paper for details).

▶ **Theorem 9.** *There exists a data structure of complexity* $O(n\log^2 n, n\log n, \sqrt{n}\log n)$ *for the ray-shooting among intersecting segments, where the query time holds with high probability.*

4.2.2 The deterministic result

To obtain the deterministic result, we resort to Matoušek's partition trees [20, 21].

An overview. To solve the simplex range searching problem (e.g., the counting problem), Matoušek built a partition tree in [20] with complexity $O(n\log n, n, \sqrt{n}(\log n)^{O(1)})$; subsequently, he presented a more query-efficient result in [21] with complexity $O(n^{1+\delta}, n, \sqrt{n})$. Ideally, we want to use his second approach. In order to achieve the $O(n^{1+\delta})$ preprocessing time, Matoušek used multilevel data structures (called partial simplex decomposition scheme in [21]). In our problem, however, the multilevel data structures do not work any more because they do not provide a "nice" way to store the segments of S. Without using multilevel data structures, the preprocessing time would be too high (indeed Matoušek [21] gave a *basic algorithm* without using multilevel data structures but he only showed that its runtime is polynomial). By a careful implementation, we can bound the preprocessing time by $O(n^2)$. To improve it, we resort to the *simplicial partition* in [20]. Roughly speaking, let P be the set of endpoints of the segments of S; we partition P into $r = \Theta(\sqrt{n})$ subsets of size $\sqrt{n}$ each, using r triangles such that any line in the plane only crosses $O(\sqrt{r})$ triangles. Then, for each subset, we apply the algorithm of [21]. This guarantees the $O(n^{1.5})$ upper bound on the preprocessing time for all subsets. To compute the simplicial partition, Matoušek [20] first provided a *basic algorithm* of polynomial time and then used other techniques to reduce the time to $O(n\log n)$. For our purpose, these techniques are not suitable (for a similar reason to multilevel data structures). Hence, we can only use the basic algorithm, whose time complexity is only shown to be polynomial in [20]. Further, we cannot directly use the algorithm because the produced triangles may overlap (the algorithm in [21] has the same issue). Nevertheless, we manage to modify the algorithm and bound its time complexity by $O(n^{1.5})$. Also, even with the above modification that avoids certain triangle overlap, using the approach in [21] directly still cannot lead to an $O(\sqrt{n}\log n)$ time query algorithm. Instead we have to further modify the algorithm (e.g., choose a different weight function).

In the following, we first describe our algorithm for computing the simplicial partition.

4.2.3 Computing a simplicial partition

Recall that P is the set of the endpoints of S and $|S| = n$. To simplify the notation, we let $|P| = n$ in the following (and thus $|S| = n/2$).

A *simplicial partion* of size m for P is a collection $\Pi = \{(P_1, \triangle_1), \ldots, (P_m, \triangle_m)\}$ with the following properties: (1) The subsets P_i's form a disjoint partition of P; (2) each $\triangle_i$ is an open triangle containing P_i; (3) $\max_{1\le i\le m}|P_i| \le 2\cdot\min_{1\le i\le m}|P_i|$; (4) the triangles may overlap and a triangle $\triangle_i$ may contain points in $P\setminus P_i$. We define the *crossing number* of Π as the largest number of triangles that are intersected by any line in the plane.

▶ **Lemma 10** ([20]). *For any integer* z *with* $2 \le z < |P|$*, there exists a simplicial partition* Π *of size* $\Theta(r)$ *for* P*, whose subsets* P_i*'s satisfy* $z \le |P_i| < 2z$*, and whose crossing number is* $O(\sqrt{r})$*, where* $r = |P|/z$*.*

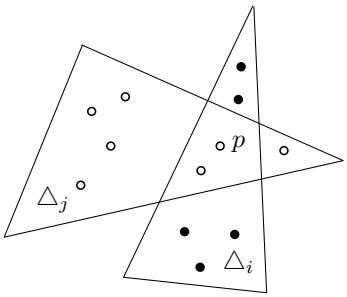

Figure 2 Illustrating the weakly-overlapped property: P_j consists of all circle points and P_i consists of all disk points. A point $p \in P_j$ is also contained in $\triangle_i$, but all points of P_i are outside $\triangle_j$.

To compute such a simplicial partition as in Lemma 10, Matoušek [20] first presented a basic algorithm whose runtime is polynomial and then improved the time to $O(n \log n)$ by other techniques. As discussed before, the techniques are not suitable for our purpose and we can only use the basic algorithm. In addition, the above property (4) prevents us from using the partition directly. Instead we use an *enhanced simplicial partition* with the following modified/changed properties. In property (2), each $\triangle_i$ is either a triangle or a convex quadrilateral; we now call $\triangle_i$ a *cell*. In property (4), the cells may still overlap, and a cell $\triangle_i$ may still contain points in $P \setminus P_i$; however, if $\triangle_i$ contains a point $p \in P_j$ with $j \neq i$, then all points of P_i are outside $\triangle_j$ (e.g., see Fig. 2). This modified property (4), which we call *the weakly-overlapped property*, is the key to guarantee the success of our approach. We use convex quadrilaterals instead of only triangles to make sure the weakly-overlapped property can be achieved. The crossing number of the enhanced partition is defined as the largest number of cells that are intersected by any line in the plane. By modifying Matoušek's basic algorithm [20], we can compute in $O(n^{1.5})$ time an enhanced simplicial partition $\Pi = \{(P_1, \triangle_1), \ldots, (P_m, \triangle_m)\}$ with $m = \Theta(r)$, which satisfies the property of Lemma 10 with $z = \sqrt{n}$ (and thus $r = \sqrt{n}$); in particular, the crossing number of Π is $O(\sqrt{r})$. The algorithm is omitted but can be found in the full paper.

Storing the segments in Π. For each segment s of S, if both endpoints of s are in the same subset P_i of Π, then s is in the cell $\triangle_i$ and we store s in $\triangle_i$; let S_i denote the set of segments stored in $\triangle_i$. Otherwise, let P_i and P_j be the two subsets that contain the endpoints of s, respectively. The weakly-overlapped property of Π leads to the following observation.

▶ **Observation 11.** *The segment s intersects the boundary of at least one cell of $\triangle_i$ and $\triangle_j$.*

By Observation 11, we find a cell $\triangle$ of $\triangle_i$ and $\triangle_j$ whose boundary intersects s. Let e be an edge of $\triangle$ that intersects s. We store s at e; let $S(e)$ denote the set of segments of S that are stored at e. In this way, each segment of S is stored exactly once. Next, for each cell $\triangle \in \Pi$ and for each edge e of $\triangle$, we preprocess $S(e)$ using Lemma 6. With Π, the above preprocessing on S takes $O(n \log n)$ time and $O(n)$ space.

We further have the following Lemma 12, whose proof can be found in the full paper.

▶ **Lemma 12.**

1. *For each subset P_i of Π, with $O(|P_i|^2)$ time and $O(|P_i|)$ space preprocessing, we can decide whether a query line intersects any segment of S_i in $O(\sqrt{|P_i|} \log |P_i|)$ time.*
2. *If segments of S_i are nonintersecting, with $O(|P_i|^2)$ time and $O(|P_i|)$ space preprocessing, we can find the first segment of S_i hit by a query ray in $O(\sqrt{|P_i|} \log |P_i|)$ time.*

We preprocess each P_i using Lemma 12. As Π has $O(\sqrt{n})$ subsets P_i and the size of each P_i is $O(\sqrt{n})$, the total preprocessing time is $O(n^{1.5})$ and the total space is $O(n)$.

Answering queries. Consider a query line ℓ. First, for each cell $\triangle_i$ of Π, for each edge e of $\triangle_i$, we determine whether ℓ intersects a segment of $S(e)$, in $O(\log n)$ time by Lemma 6(1). As Π has $\Theta(\sqrt{n})$ cells and each cell has at most four edges, the total time of this step is $O(\sqrt{n}\log n)$. Second, by checking every cell of Π, we find those cells that are crossed by ℓ. For each such cell $\triangle_i$, by Lemma 12(1), we determine whether ℓ intersects any segment of S_i in $O(n^{1/4}\log n)$ time, for $|P_i| = \Theta(\sqrt{n})$. As ℓ can cross at most $O(n^{1/4})$ cells of Π, this step takes $O(\sqrt{n}\log n)$ time. Hence, the query time is $O(\sqrt{n}\log n)$.

If the segments of S_i are nonintersecting, the ray-shooting query algorithm is similar. We can thus obtain our results for the segment intersection and the ray-shooting problems.

References

1. P.K. Agarwal. Ray shooting and other applications of spanning trees with low stabbing number. *SIAM Journal on Computing*, 21:540–570, 1992.
2. P.K. Agarwal and M. Sharir. Applications of a new space-partitioning technique. *Discrete and Computational Geometry*, 9(1):11–38, 1993.
3. R. Bar-Yehuda and S. Fogel. Variations on ray shootings. *Algorithmica*, 11:133–145, 1994.
4. B. Becker, P.G. Franciosa, S. Gschwind, S. Leonardi, T. Ohler, and P. Widmayer. Enclosing a set of objects by two minimum area rectangles. *Journal of Algorithms*, 21:520–541, 1996.
5. B. Becker, P.G. Franciosa, S. Gschwind, T. Ohler, T. Ohler, G. Thiemt, and P. Widmayer. An optimal algorithm for approximating a set of rectangles by two minimum area rectangles. In *Workshop on Computational Geometry*, pages 13–25, 1991.
6. T.M. Chan. Optimal partition trees. *Discrete and Computational Geometry*, 47:661–690, 2012.
7. B. Chazelle and L. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(1):163–191, 1986.
8. B. Chazelle and E. Welzl. Quasi-optimal range searching in spaces of finite VC-dimension. *Discrete and Computational Geometry*, 4(5):467–489, 1989.
9. D.Z. Chen, J. Li, and H. Wang. Efficient algorithms for the one-dimensional k-center problem. *Theoretical Computer Science*, 592:135–142, 2015.
10. D.Z. Chen and H. Wang. Approximating points by a piecewise linear function. *Algorithmica*, 88:682–713, 2013.
11. S.W. Cheng and R. Janardan. Algorithms for ray-shooting and intersection searching. *Journal of Algorithms*, 13:670–692, 1992.
12. T. Christ, M. Hoffmann, Y. Okamoto, and T. Uno. Improved bounds for wireless localization. *Algorithmica*, 57:499–516, 2010.
13. B.-S. Dai, M.-J. Kao, and D.T. Lee. Optimal time-convex hull under the L_p metrics. In *Proceedings of the 13rd Algorithms and Data Structures Symposium (WADS)*, pages 268–279, 2013.
14. J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
15. H. Edelsbrunner, L. Guibas, J. Hershberger, R. Seidel, M. Sharir, J. Snoeyink, and E. Welzl. Implicitly representing arrangements of lines or segments. *Discrete and Computational Geometry*, 4:433–466, 1989.
16. H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $O(n^{0.695})$ query time. *Information Processing Letters*, 23:289–293, 1986.
17. R.L. Graham and F.F. Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4:324–331, 1983.

18 L. Guibas, J. Hershberger, and J. Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry and Applications*, 1(1):1–22, 1991. First appeared in SODA 1990.

19 L. Guibas, M. Overmars, and M. Sharir. Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 64–73, 1988.

20 J. Matoušek. Efficient partition trees. *Discrete and Computational Geometry*, 8(3):315–334, 1992.

21 J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry*, 10(1):157–182, 1993.

22 A. Melkman. On-line construction of the convex hull of a simple polygon. *Information Processing Letters*, 25:11–12, 1987.

23 M.H. Overmars, H. Schipper, and M. Sharir. Storing line segments in partition trees. *BIT Numerical Mathematics*, 30:385–403, 1990.

24 H. Wagener. Optimal parallel hull construction for simple polygons in $O(\log \log n)$ time. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–599, 1992.

25 H. Wang and W. Zhang. On top-k weighted sum aggregate nearest and farthest neighbors in the L_1 plane. *International Journal of Computational Geometry and Applications*, 29:189–218, 2019.