Algorithms for Computing Closest Points for Segments

Haitao Wang
Kahlert School of Computing, University of Utah, Salt Lake City, UT, USA

Abstract

Given a set P of n points and a set S of n segments in the plane, we consider the problem of computing for each segment of S its closest point in P. The previously best algorithm solves the problem in $n^{4/3}2^{O(\log^* n)}$ time [Bespamyatnikh, 2003] and a lower bound (under a somewhat restricted model) $\Omega(n^{4/3})$ has also been proved. In this paper, we present an $O(n^{4/3})$ time algorithm and thus solve the problem optimally (under the restricted model). In addition, we also present data structures for solving the online version of the problem, i.e., given a query segment (or a line as a special case), find its closest point in P. Our new results improve the previous work.

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

Keywords and phrases Closest points, Voronoi diagrams, Segment dragging queries, Hopcroft's problem, Algebraic decision tree model

Digital Object Identifier 10.4230/LIPIcs.STACS.2024.58

Related Version *Full Version*: https://arxiv.org/abs/2401.02636

Funding This research was supported in part by NSF under Grant CCF-2300356.

1 Introduction

Given a set P of n points and a set S of n segments in the plane, we consider the problem of computing for each segment of S its closest point in P. We call it the *segment-closest-point* problem. Previously, Bespamyatnikh [6] gave an $n^{4/3}2^{O(\log^* n)}$ time algorithm for the problem, improving upon an $O(n^{4/3}\log^{O(1)} n)$ time result of Agarwal and Procopiuc. The problem can be viewed as a generalization of Hopcroft's problem [1, 10, 13, 20, 30], which is to determine whether any point of a given set of n points lies on any of the given n lines. Erickson [21] proved an $\Omega(n^{4/3})$ time lower bound for Hopcroft's problem under a somewhat restricted *partition model*. This implies the same lower bound on the segment-closest-point problem. For Hopcroft's problem, Chan and Zheng [10] recently gave an $O(n^{4/3})$ time algorithm, which matches the lower bound and thus is optimal.

In this paper, with some new observations on the problem as well as the techniques from Chan and Zheng [10] (more specifically, the Γ-*algorithm framework* for bounding algebraic decision tree complexities), we present a new algorithm that solves the segment-closest-point problem in $O(n^{4/3})$ time and thus is optimal under Erickson's partition model [21]. It should be noted that our result is not a direct application of Chan and Zheng's techniques [10], but rather many new observations and techniques are needed. For example, one subroutine in our problem is the following *outside-hull segment queries*: Given a segment outside the convex hull of P, find its closest point in P. Bespamyatnikh and Snoeyink [7] built a data structure in $O(n)$ space and $O(n \log n)$ time such that each query can be answered in $O(\log n)$ time. Unfortunately, their query algorithm does not fit the Γ-algorithm framework of Chan and Zheng [10]. To resolve the issue, we develop another algorithm for the problem based on new observations. Our approach is simpler, and more importantly, it fits into the Γ-algorithm framework of Chan and Zheng [10]. The result may be interesting in its own right.

We also consider the online version of the problem, called *the segment query problem*: Preprocess P so that given a query segment, its closest point in P can be found efficiently. For the special case where the query segment is outside the convex hull of P, one can use the data structure of Bespamyatnikh and Snoeyink [7] mentioned above. For simplicity, we use $(T_1(n), T_2(n), T_3(n))$ to denote the complexity of a data structure if its preprocessing time, space, and query time are on the order of $T_1(n)$, $T_2(n)$, and $T_3(n)$, respectively. Using this notation, the complexity of the above data structure of Bespamyatnikh and Snoeyink [7] is $O(n\log n, n, \log n)$. The general problem, however, is much more challenging. Goswami, Das, and Nandy [25]'s method yields a result of complexity $O(n^2, n^2, \log^2 n)$. We present a new data structure of complexity $O(nm(n/m)^\delta, nm\log(n/m), \sqrt{n/m}\log(n/m))$, any m with $1 \leq m \leq n\log^2\log n/\log^4 n$ and any $\delta > 0$. Note that for the large space case (i.e., when $m = n\log^2\log n/\log^4 n$), the complexity of our data structure is $O(n^2/\log^{4-\delta} n, n^2\log^3\log n/\log^4 n, \log^2 n)$, which improves the above result of [25] on the preprocessing time and space by a factor of roughly $\log^4 n$. We also give a faster randomized data structure of complexity $O(nm\log(n/m), nm\log(n/m), \sqrt{n/m})$ for any m with $1 \leq m \leq n/\log^4 n$, where the preprocessing time is expected and the query time holds with high probability. In addition, using Chan's randomized techniques [8] and Chan and Zheng's recent randomized result on triangle range counting [10], one can obtain a randomized data structure of complexity $O(n^{4/3}, n^{4/3}, n^{1/3})$. Note that this data structure immediately leads to a randomized algorithm of $O(n^{4/3})$ expected time for the segment-closest-point problem. As such, for solving the segment-closest-point problem, our main effort is to derive an $O(n^{4/3})$ deterministic time algorithm. Note that this is aligned with the motivation of proposing the Γ-algorithm framework in [10], whose goal was to obtain an $O(n^{4/3})$ deterministic time algorithm for Hopcroft's problem although a much simpler randomized algorithm of $O(n^{4/3})$ expected time was already presented.

If each query segment is a line, we call it the *line query problem*, which has been extensively studied. Previous work includes Cole and Yap [17]'s and Lee and Ching [28]'s data structures of complexity $O(n^2, n^2, \log n)$, Mitra and Chaudhuri [31]'s work of complexity $O(n\log n, n, n^{0.695})$, Mukhopadhyay [32]'s result of complexity $O(n^{1+\delta}, n\log n, n^{1/2+\delta})$ for any $\delta > 0$. As observed by Lee and Ching [28], the problem can be reduced to vertical ray-shooting in the dual plane, i.e., finding the first line hit by a query vertical ray among a given set of n lines. Using the ray-shooting algorithms, the best deterministic result is $O(n^{1.5}, n, \sqrt{n}\log n)$ [35] while the best randomized result is $O(n\log n, n, \sqrt{n})$ [11]; refer to [2, 4, 10, 16] for other (less efficient) work on ray-shootings. We build a new deterministic data structure of complexity $O(nm(n/m)^\delta, nm\log(n/m), \sqrt{n/m})$, for any $1 \leq m \leq n/\log^2 n$. We also have another faster randomized result of complexity $O(nm\log(n/m), nm\log(n/m), \sqrt{n/m})$, for any m with $1 \leq m \leq n/\log^2 n$, where the preprocessing time is expected while the query time holds with high probability. Our results improve all previous work except the randomized result of Chan and Zheng [11]. For example, if $m = 1$, our data structure is the only deterministic one whose query time is $O(\sqrt{n})$ with near linear space; if $m = n/\log^2 n$, our result achieves $O(\log n)$ query time while the preprocessing is subquadratic, better than those by Cole and Yap [17] and Lee and Ching [28].

Other related work. If all segments are pairwise disjoint, then the segment-closest-point problem was solved in $O(n\log^2 n)$ time by Bespamyatnikh [6], improving over the $O(n\log^3 n)$ time algorithm of Bespamyatnikh and Snoeyink [7].

If every segment of S is a single point, then the problem can be easily solved in $O(n\log n)$ time using the Voronoi diagram of P. Also, for any segment $s \in S$, if the point of s closest to P is an endpoint of s, then finding the closest point of s in P can be done using the Voronoi diagram of P. Hence, the remaining issue is to find the first point of P hit by s if we drag s along the directions perpendicularly to s. If all segments of S have the same slope, then the problem can be solved in $O(n \log n)$ time using the segment dragging query data structure of Chazelle [12], which can answer each query in $O(\log n)$ time after $O(n)$ space and $O(n \log n)$ time preprocessing. However, the algorithm [12] does not work if the query segments have arbitrary slopes. As such, the challenge of the problem is to solve the dragging queries for all segments of S when their slopes are not the same.

The *segment-farthest-point* problem has also been studied, where one wants to find for each segment of S its farthest point in P. The problem appears much easier. For the line query problem (i.e., given a query line, find its farthest point in P), Daescu et al. [18] gave a data structure of complexity $O(n \log n, n, \log n)$. Using this result, they also proposed a data structure of complexity $O(n \log n, n \log n, \log^2 n)$ for the segment query problem. Using this segment query data structure, the segment-farthest-point can be solved in $O(n \log^2 n)$ time.

Outline. The rest of the paper is organized as follows. In Section 2, we introduce some notation and concepts. In Section 3, we present our $O(n^{4/3})$ deterministic time algorithm for the segment-closest-point problem. We actually solve a more general problem where the number of points is not equal to the number of segments, referred to as the *asymmetric case*, and our algorithm runs in $O(n^{2/3}m^{2/3} + n \log n + m \log^2 n)$ time with n as the number of points and m as the number of segments. For the line case of the problem where all segments are lines, a simpler algorithm is presented in the full paper and the algorithm also runs in $O(n^{4/3})$ time (and $O(n^{2/3}m^{2/3} + (n + m) \log n)$ time for the asymmetric case). The online query problem is sketched in Section 4 with details in the full paper. Due to the space limit, many lemma proofs are omitted but can be found in the full paper.

2 Preliminaries

For two closed subsets A and B in the plane, let $d(A, B)$ denote the minimum distance between any point of A and any point of B. The point p of A closest to B, i.e., $d(p, B) = d(A, B)$, is called the *closest point* of B in A. For any two points a and b in the plane, we use $\overline{ab}$ to denote the segment with a and b as its two endpoints.

For any point p in the plane, we use $x(p)$ and $y(p)$ to denote its x- and y-coordinates, respectively. For a point p and a region A in the plane, we say that p is *to the left* of A if $x(p) \leq x(q)$ for all points $q \in A$, and p is *strictly to the left* of A if $x(p) < x(q)$ for all points $q \in A$; the concepts *(strictly) to the right* is defined symmetrically.

For a set Q of points in the plane, we usually use $\mathsf{VD}(Q)$ to denote the Voronoi diagram of Q and use $\mathsf{CH}(Q)$ to denote the convex hull of Q; we also use $Q(A)$ to denote the subset of Q in A, i.e., $Q(A) = Q \cap A$, for any region A in the plane.

Cuttings. Let H be a set of n lines in the plane. Let H_A denote the subset of lines of H that intersect the interior of A (we also say that these lines *cross* A), for a compact region A in the plane. A *cutting* is a collection Ξ of closed cells (each of which is a triangle) with disjoint interiors, which together cover the entire plane [13, 30]. The *size* of Ξ is the number of cells in Ξ. For a parameter r with $1 \leq r \leq n$, a $(1/r)$*-cutting* for H is a cutting Ξ satisfying $|H_\sigma| \leq n/r$ for every cell $\sigma \in \Xi$.

A cutting Ξ' c*-refines* another cutting Ξ if every cell of Ξ' is contained in a single cell of Ξ and every cell of Ξ contains at most c cells of Ξ'. A *hierarchical* $(1/r)$*-cutting* for H (with two constants c and ρ) is a sequence of cuttings $\Xi_0, \Xi_1, \ldots, \Xi_k$ with the following properties.

Ξ_0 is the entire plane. For each $1 \le i \le k$, Ξ_i is a $(1/\rho^i)$-cutting of size $O(\rho^{2i})$ which c-refines Ξ_{i-1}. In order to make Ξ_k a $(1/r)$-cutting, we set $k = \lceil \log_\rho r \rceil$. Hence, the size of the last cutting Ξ_k is $O(r^2)$. If a cell $\sigma \in \Xi_{i-1}$ contains a cell $\sigma' \in \Xi_i$, we say that σ is the *parent* of σ' and σ' is a *child* of σ. As such, one could view Ξ as a tree in which each node corresponds to a cell $\sigma \in \Xi_i$, $0 \le i \le k$.

For any $1 \le r \le n$, a hierarchical $(1/r)$-cutting of size $O(r^2)$ for H (together with H_σ for every cell σ of Ξ_i for all $i = 0, 1, \ldots, k$) can be computed in $O(nr)$ time [13]. Also, it is easy to check that $\sum_{i=0}^{k} \sum_{\sigma \in \Xi_i} |H_\sigma| = O(nr)$.

3 The segment-closest-point problem

In this section, we consider the segment-closest-point problem. Let P be a set of n points and S a set of n segments in the plane. The problem is to compute for each segment of S its closest point in P. We make a general position assumption that no segment of S is vertical (for a vertical segment, its closest point can be easily found, e.g., by building a segment dragging query data structure [12] along with the Voronoi diagram of P).

We start with a review of an algorithm of Bespamyatnikh [6], which will be needed in our new approach.

3.1 A review of Bespamyatnikh's algorithm [6]

As we will deal with subproblems in which the number of lines is not equal to the number of segments, we let m denote the number of segments in S and n the number of points in P. As such, the size of our original problem (S, P) is (m, n).

Let H be the set of the supporting lines of the segments of S. For a parameter r with $1 \le r \le \min\{m, \sqrt{n}\}$, compute a hierarchical $(1/r)$-cutting $\Xi_0, \Xi_1, \ldots, \Xi_k$ for H. For each cell $\sigma \in \Xi_i$, $0 \le i \le k$, let $P(\sigma) = P \cap \sigma$, i.e., the subset of the points of P in σ; let $S(\sigma)$ denote the subset of the segments of S intersecting σ. We further partition each cell of Ξ_k into triangles so that each triangle contains at most n/r^2 points of P and the number of new triangles in Ξ_k is still bounded by $O(r^2)$. For convenience, we consider the new triangles as new cells of Ξ_k (we still define $P(\sigma)$ and $S(\sigma)$ for each new cell σ in the same way as above; so now $|P(\sigma)| \le n/r^2$ and $|S(\sigma)| \le m/r$ hold for each cell $\sigma \in \Xi_k$).

For each cell $\sigma \in \Xi_k$, form a subproblem $(S(\sigma), P(\sigma))$ of size $(m/r, n/r^2)$, i.e., find for each segment s of $S(\sigma)$ its closest point in $P(\sigma)$. After the subproblem is solved, to find the closest point of s in P, it suffices to find its closest point in $P \setminus P(\sigma)$. To this end, observe that $P \setminus P(\sigma)$ is exactly the union of $P(\sigma'')$ for all cells σ'' such that σ'' is a child of an ancestor σ' of σ and $s \notin S(\sigma'')$. As such, for each of such cells σ'', find the closest point of s in $P(\sigma'')$. For this, since $s \notin S(\sigma'')$, s is outside σ'' and thus is outside the convex hull of $P(\sigma'')$. Hence, finding the closest point of s in $P(\sigma'')$ is an outside-hull segment query and thus the data structure of Bespamyatnikh and Snoeyink [7] (referred to as *the BS data structure* in the rest of the paper) is used, which takes $O(|P(\sigma'')|)$ space and $O(|P(\sigma'')| \log |P(\sigma'')|)$ time preprocessing and can answer each query in $O(\log |P(\sigma'')|)$ time. More precisely, the processing can be done in $O(|P(\sigma'')|)$ time if the Voronoi diagram of $P(\sigma'')$ is known.

For the time analysis, let $T(m, n)$ denote the time of the algorithm for solving a problem of size (m, n). Then, solving all subproblems takes $O(r^2) \cdot T(m/r, n/r^2)$ time as there are $O(r^2)$ subproblems of size $(m/r, n/r^2)$. Constructing the hierarchical cutting as well as computing $S(\sigma)$ for all cells σ in all cuttings Ξ_i, $0 \le i \le k$, takes $O(mr)$ time [13]. Computing $P(\sigma)$ for all cells σ can be done in $O(n \log r)$ time. Preprocessing for constructing the BS data structure for $P(\sigma)$ for all cells σ can be done in $O(n \log n \log r)$ time as $\sum_{\sigma \in \Xi_i} |P(\sigma)| = n$

for each $0 \le i \le k$, and $k = O(\log r)$. We can further reduce the time to $O(n(\log r + \log n))$ as follows. We build the BS data structure for cells of the cuttings in a bottom-up manner, i.e., processing cells of Ξ_k first and then Ξ_{k-1} and so on. After the preprocessing for $P(\sigma)$ for a cell $\sigma \in \Xi_k$, which takes $O(|P(\sigma)| \log(n/r^2))$ time since $|P(\sigma)| \le n/r^2$, the Voronoi diagram of $P(\sigma)$ is available. After the preprocessing for all cells σ of Ξ_k is done, for each cell σ' of Ξ_{k-1}, to construct the Voronoi diagram of $P(\sigma')$, merge the Voronoi diagrams of $P(\sigma)$ for all children σ of σ'. To this end, as σ' has $O(1)$ children, the merge can be done in $O(|P(\sigma')|)$ time by using the algorithm of Kirkpatrick [27], and thus the preprocessing for $P(\sigma')$ takes only linear time. In this way, the total preprocessing time for all cells in all cuttings Ξ_i, $0 \le i \le k$, is bounded by $O(n(\log r + \log(n/r^2)))$ time, i.e., the time spent on cells of Ξ_k is $O(n \log(n/r^2))$ and the time on other cuttings is $O(n \log r)$ in total. Note that $\log r + \log(n/r^2) = \log(n/r)$. As for the outside-hull segment queries, according to the properties of the hierarchical cutting, $\sum_{i=0}^{k} \sum_{\sigma \in \Xi_i} |S(\sigma)| = O(mr)$. Hence, the total number of outside-hull segment queries on the BS data structure is $O(mr)$ and thus the total query time is $O(mr \log n)$. In summary, the following recurrence is obtained for any $1 \le r \le \min\{m, \sqrt{n}\}$:

$$T(m,n) = O(n \log(n/r) + mr \log n) + O(r^2) \cdot T(m/r, n/r^2). \qquad (1)$$

Using the duality, Bespamyatnikh [6] gave a second algorithm (we will not review this algorithm here because it is not relevant to our new approach) and obtained the following recurrence for any $1 \le r \le \min\{n, \sqrt{m}\}$:

$$T(m,n) = O(nr \log n + m \log r \log n) + O(r^2) \cdot T(m/r^2, n/r). \qquad (2)$$

Setting $m = n$ and applying (2) and (1) in succession (using the same r) obtain $T(n,n) = O(nr \log n) + O(r^4) \cdot T(n/r^3, n/r^3)$. Setting $r = n^{1/3}/\log n$ leads to

$$T(n,n) = O(n^{4/3}) + O((n/\log^3 n)^{4/3}) \cdot T(\log^3 n, \log^3 n). \qquad (3)$$

The recurrence solves to $T(n,n) = n^{4/3} 2^{O(\log^* n)}$, which is the time bound obtained in [6].

3.2 Our new algorithm

In this section, we improve the algorithm to $O(n^{4/3})$ time.

By applying recurrence (3) three times we obtain the following:

$$T(n,n) = O(n^{4/3}) + O((n/b)^{4/3}) \cdot T(b,b), \qquad (4)$$

where $b = (\log\log\log n)^3$.

Using the property that b is tiny, we show in the following that after $O(n)$ time preprocessing, we can solve each subproblem $T(b,b)$ in $O(b^{4/3})$ time (for convenience, by slightly abusing the notation, we also use $T(m,n)$ to denote a subproblem of size (m,n)). Plugging the result into (4), we obtain $T(n,n) = O(n^{4/3})$.

More precisely, we show that after $O(2^{\text{poly}(b)})$ time preprocessing, where $\text{poly}(\cdot)$ is a polynomial function, we can solve each $T(b,b)$ using $O(b^{4/3})$ comparisons, or alternatively, $T(b,b)$ can be solved by an algebraic decision tree of height $O(b^{4/3})$. As $b = (\log\log\log n)^3$, $2^{\text{poly}(b)}$ is bounded by $O(n)$. To turn this into an algorithm under the standard real-RAM model, we explicitly construct the algebraic decision tree for the above algorithm (we may also consider this step as part of preprocessing for solving $T(b,b)$), which can again be done in $O(2^{\text{poly}(b)})$ time. As such, that after $O(n)$ time preprocessing, we can solve each $T(b,b)$ in $O(b^{4/3})$ time. In the following, for notational convenience, we will use n to denote b, and our goal is to prove the following lemma.

▶ **Lemma 1.** *After $O(2^{poly(n)})$ time preprocessing, $T(n,n)$ can be solved using $O(n^{4/3})$ comparisons.*

We apply recurrence (1) by setting $m = n$ and $r = n^{1/3}$, and obtain the following

$$T(n,n) = O(n \log n + n^{4/3} \log n) + O(n^{2/3}) \cdot T(n^{2/3}, n^{1/3}). \tag{5}$$

Recall that the term $n^{4/3} \log n$ is due to that there are $O(n^{4/3})$ outside-hull segment queries. To show that $T(n,n)$ can be solved by $O(n^{4/3})$ comparisons, there are two challenges: (1) solve all outside-hull segment queries using $O(n^{4/3})$ comparisons; (2) solve each subproblem $T(n^{2/3}, n^{1/3})$ using $O(n^{2/3})$ comparisons.

Γ-algorithm framework. To tackle these challenges, we use a Γ-*algorithm framework* for bounding decision tree complexities proposed by Chan and Zheng [10]. We briefly review it here (see Section 4.1 [10] for the details). Roughly speaking, this framework is an algorithm that only counts the number of comparisons (called Γ-comparisons in [10]) for determining whether a point belongs to a semialgebraic set of $O(1)$ degree in a constant-dimensional space. Solving our segment-closest-point problem is equivalent to locating the cell C^* containing a point p^* parameterized by the input of our problem (i.e., the segments of S and the points of P) in an arrangement $\mathcal{A}$ of the boundaries of poly(n) semialgebraic sets in $O(n)$-dimensional space. This arrangement can be built in $O(2^{\text{poly}(n)})$ time without examining the values of the input and thus does not require any comparisons. In particular, the number of cells of $\mathcal{A}$ is bounded by $n^{O(n)}$. As a Γ-algorithm progresses, it maintains a set Π of cells of $\mathcal{A}$. Initially, Π consisting of all cells of $\mathcal{A}$. During the course of the algorithm, Π can only shrink but always contains the cell C^*. At the end of the algorithm, C^* will be found. Define the potential $\Phi = \log |\Pi|$. As $\mathcal{A}$ has $n^{O(n)}$ cells, initially $\Phi = O(n \log n)$. For any operation or subroutine of the algorithm, we use $\Delta\Phi$ to denote the change of Φ. As Φ only decreases during the algorithm, $\Delta\Phi \leq 0$ always holds and the sum of $-\Delta\Phi$ during the entire algorithm is $O(n \log n)$. This implies that we may afford an expensive operation/subroutine during the algorithm as long as it decreases Φ a lot.

Two algorithmic tools are developed in [10] under the framework: *basic search lemma* (Lemma 4.1 [10]) and *search lemma* (Lemma A.1 [10]). Roughly speaking, given r predicates (each predicate is a test of whether $\gamma(x)$ is true for the input vector x), suppose it is promised that at least one of them is true for all inputs in the active cells; then the basic search lemma can find a predicate that is true by making $O(1 - r \cdot \Delta\Phi)$ comparisons. Given a binary tree (or a more general DAG of $O(1)$ degree) such that each node v is associated with a predicate γ_v, suppose for each internal node v, γ_v implies γ_u for a child u of v for all inputs in the active cells. Then, the search lemma can find a leaf v such that γ_v is true by making $O(1 - \Delta\Phi)$ comparisons.

An application of both lemmas particularly discussed in [10] is to find a predecessor of a query number among a sorted list of input numbers. In our algorithm, as will be seen later, the subproblem that needs the Γ-algorithm framework is also finding predecessors among sorted lists and thus both the basic search lemma and the search lemma are applicable.

In the following two subsections, we will tackle the above two challenges, respectively. By slightly abusing the notation, let P be a set of n points and S a set of n segments for the problem in recurrence (5).

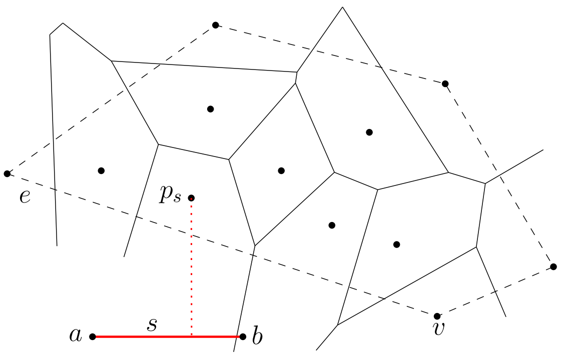

Figure 1 Illustrating an outside-hull segment query.

3.3 Solving outside-hull segment queries

Recall that we have used the BS data structure to answer the outside-hull segment queries. Unfortunately the algorithm does not fit the Γ-algorithm framework. Indeed, the BS data structure is a binary tree. However, each node of the tree represents a convex hull of a subset of points and it is not associated with a predicate that we can use to apply the Γ-algorithm framework (e.g., the search lemma as discussed above).

In the following, we first present a new algorithm for solving the outside-hull segment queries. Our algorithm, whose performance matches that of the BS data structure, is simpler, and thus may be of independent interest; more importantly, it leads to an algorithm that fits the Γ-algorithm framework to provide an $O(n^{4/3})$ upper bound.

Let Q be a set of n' points. The problem is to preprocess Q so that given any query segment s outside the convex hull $\mathsf{CH}(Q)$ of Q, the closest point of s in Q can be computed efficiently. Recall that in our original problem (i.e., the recurrence (5)) Q is a subset of P and the sum of n' for all subsets of P that we need to build the outside-hull query data structures is $O(n \log n)$. We make this an observation below, which will be referred to later.

▶ **Observation 2.** *The size of the subsets of P that we need to build the outside-hull query data structures is $O(n \log n)$, i.e., $\sum n' = O(n \log n)$.*

In the preprocessing, we compute the Voronoi diagram $\mathsf{VD}(Q)$ of Q, from which we can obtain the convex hull $\mathsf{CH}(Q)$ in linear time. For each edge e of $\mathsf{CH}(Q)$, we determine the subset Q_e of points of Q whose Voronoi cells intersect e in order along e. This order is exactly the order of the perpendicular projections of the points of Q_e onto e [7].

Consider a query segment s that is outside $\mathsf{CH}(Q)$. Let p_s be the first point of Q hit by s if we drag s along the direction perpendicularly to s and towards $\mathsf{CH}(Q)$; see Fig. 1. For ease of exposition, we assume that p_s is unique. Our goal is to compute p_s in the case where the point of s closest to Q is not an endpoint of s since the other case can be easily solved by using $\mathsf{VD}(Q)$. Henceforth, we assume that the point of s closest to Q is not an endpoint of s, implying that p_s is the point of Q closest to s. Without loss of generality, we assume that s is horizontal and s is below $\mathsf{CH}(Q)$. Let a and b be the left and right endpoints of s, respectively (see Fig. 1).

We first find the lowest vertex v of $\mathsf{CH}(Q)$, which can be done in $O(\log n')$ time by doing binary search on $\mathsf{CH}(Q)$. If $x(a) \leq x(v) \leq x(b)$, then v is p_s and we are done with the query. Otherwise, without loss of generality, we assume that $x(b) < x(v)$. By binary search on $\mathsf{CH}(Q)$, we find the edge e in the lower hull of $\mathsf{CH}(Q)$ that intersects the vertical line through b. Since $x(a) \leq x(b) < x(v)$, e must have a negative slope (see Fig. 1). Then, as discussed in [7], p_s must be in Q_e. To find p_s efficiently, we first make some observations (which were not discovered in the previous work).

Suppose $p_1, p_2, \ldots, p_m$ are the points of Q_e, sorted following the order of their Voronoi cells in $\mathsf{VD}(Q)$ intersecting e from left to right. We define two special indices i^* and j^* of Q_e with respect to a and b, respectively.

▶ **Definition 3.** *Define j^* as the largest index of the point of Q_e that is to the left of b. Define i^* as the smallest index of the point of Q_e such that p_j is to the right of a for all $j \geq i^*$.*

Note that j^* must exist as p_s is in Q_e and is to the left of b. We have the following lemma.

▶ **Lemma 4.** *If i^* does not exist or $i^* > j^*$, then p_s cannot be the closest point of s in Q.*

By Lemma 4, if i^* does not exist or if $i^* > j^*$, then we can simply stop the query algorithm. In the following, we assume that i^* exists and $i^* \leq j^*$. Let $Q_e[i^*, j^*]$ denote the subset of points of Q_e whose indices are between i^* and j^* inclusively. The following lemma implies that we can use the supporting line of s to search p_s.

▶ **Lemma 5.** *Suppose p_s is the closest point of s in Q. Then, p_s is the point of $Q_e[i^*, j^*]$ closest to the supporting line of s (i.e., the line containing s).*

Based on Lemma 5, we have the following three steps to compute p_s: (1) compute j^*; (2) compute i^*; (3) find the point of $Q_e[i^*, j^*]$ closest to the supporting line ℓ_s of s.

The following Lemma 6, which is for outside-hull segment queries, is a by-product of our above observations. Its complexity is the same as that in [7]. However, we feel that our new query algorithm is simpler and thus this result may be interesting in its own right.

▶ **Lemma 6.** *Given a set Q of n' points in the plane, we can build a data structure of $O(n')$ space in $O(n' \log n')$ time such that each outside-hull query can be answered in $O(\log n')$ time. The preprocessing time is $O(n')$ if the Voronoi diagram of Q is known.*

The query algorithm of Lemma 6 actually does not fit the Γ-algorithm framework. Instead, following the above observations we will give another query algorithm that fits the Γ-algorithm framework. We now give a new algorithm that fits the Γ-algorithm framework. The new algorithm requires slightly more preprocessing than Lemma 6. But for our purpose, we are satisfied with $O(n^{4/3})$ preprocessing time. We have different preprocessing for each of the three steps of the query algorithm, as follows.

The first step: computing j^*. For computing j^*, we will use the *basic search lemma* (i.e., Lemma 4.1) in [10]. In order to apply the lemma, we perform the following preprocessing.

Recall that $Q_e = \{p_1, p_2, \ldots, p_m\}$ is ordered by their Voronoi cells intersecting e. We partition the sequence into r contiguous subsequences of size roughly m/r each. Let Q_e^i denote the i-th subsequence, with $1 \leq i \leq r$. For each $i \in [1, r]$, we compute and explicitly maintain the convex hull $\mathsf{CH}(i)$ of all points in the union of the subsequences Q_e^j, $j = i, i+1, \ldots, r$. Next, for each subsequence Q_e^i, we further partition it into r contiguous sequences of size roughly $|Q_e^i|/r$ and process it in the same way as above. We do this recursively until the subsequence has no more than r points. In this way, we obtain a tree T with m leaves such that each node has r children. For each node v, we use $\mathsf{CH}(v)$ to denote the convex hull that is computed above corresponding to v (e.g., if v is the child of the root corresponding to Q_e^i, then $\mathsf{CH}(v)$ is $\mathsf{CH}(i)$ defined above). The total time for constructing T can be easily bounded by $O(mr \log m \log_r m)$ as the height of T is $O(\log_r m)$.

Now to compute j^*, we search the tree T: starting from the root, for each node v, we apply the basic search lemma on all r children of v. Indeed, this is possible due to the following. Consider the root v. For each i with $1 \leq i \leq r$, let x_i denote the x-coordinate of the leftmost point of the union of the subsequences Q_e^j, $j = i, i+1, \ldots, r$; note that x_i is also the leftmost vertex of $\mathsf{CH}(i)$. It is not difficult to see that $x_1 \leq x_2 \leq \ldots \leq x_r$. Observe that p_{j^*} is in Q_e^i if and only if $x_i \leq x(b) < x_{i+1}$. Therefore, we find the index i such that $x_i \leq x(b) < x_{i+1}$ and then proceed to the child of v corresponding to Q_e^i. This property satisfies the condition of the basic search lemma (essentially, we are looking for the predecessor of b in the sequence $x_1, x_2, \ldots, x_r$ and this is somewhat similar to the insertion sort algorithm of Theorem 4.1 [10], which uses the basic search lemma). By the basic search lemma, finding the index i can be done using $O(1 - r\Delta\Phi)$ comparisons provided that the x-coordinates $x_1, x_2, \ldots, x_r$ are available to us (we will discuss how to compute them later). We then follow the same idea recursively until we reach a leaf. In this way, the total number of comparisons for computing j^* is $O(\log_r m - r\Delta\Phi)$.

By setting $r = m^\epsilon$ for a small constant ϵ, the preprocessing time is $O(m^{1+\epsilon}\log m)$ and computing j^* can be done using $O(1 - m^\epsilon\Delta\Phi)$ comparisons. Recall that there are $O(n^{4/3})$ queries in our original problem (i.e., recurrence (5)) and the total time for $-\Delta\Phi$ during the entire algorithm is $O(n\log n)$. Also, since m is the number of points of Q whose Voronoi cells intersecting the edge e of $\mathsf{CH}(Q)$, the sum of m for all outside-hull segment query data structures for all edges of $\mathsf{CH}(Q)$ is $|Q|$, which is n'. By Observation 2, the sum of n' for all data structures in our original problem is $O(n\log n)$. Hence, the total preprocessing time for our original problem is $O(n^{1+\epsilon}\log^{2+\epsilon} n)$, which is bounded by $O(n^{4/3})$ if we set ϵ to a small constant (e.g., $\epsilon = 1/4$). As such, with a preprocessing step of $O(n^{4/3})$ time, we can compute j^* for all queries using a total of $O(n^{4/3})$ comparisons.

The above complexity analysis for computing j^* is based on the assumption that the leftmost point of $\mathsf{CH}(v)$ for each node v of T is known. To find these points during the queries, we take advantage of the property that all queries are offline, i.e., we know all query segments before we start the queries. Notice that although there are $O(n^{4/3})$ queries, the number of distinct query segments is n, i.e., those in S (a segment may be queried on different subsets of P). Let s be the current query segment and p be the leftmost point of a convex hull $CH(v)$ with respect to s (i.e., by assuming s is horizontal). Let ρ_1 be the ray from p going vertically upwards. Let ρ_2 be another ray from p going through the clockwise neighbor of p on CH_v, i.e., ρ_2 contains the clockwise edge of CH_v incident to v. Observe that for another query segment s', p is still the leftmost point of CH_v with respect to s' as long as the direction perpendicular to s' is within the angle from ρ_1 clockwise to ρ_2. Based on this observation, before we start any query, we sort the perpendicular directions of all segments of S along with the directions of all edges of all convex hulls of all nodes of the trees T for all outside-hull segment query data structures in our original problem (i.e., the recurrence (5)). As analyzed above, the total size of convex hulls of all trees T is $O(n^{1+\epsilon}\log^{2+\epsilon} n)$. Hence, the sorting can be done in $O(n^{1+\epsilon}\log^{3+\epsilon} n)$ time. Let L be the sorted list. We solve the queries for segments following their order in L. Let s and s' be two consecutive segments of S in L. After we solve all queries for s, the directions between s and s' in L correspond to those nodes of the trees T whose leftmost points need to get updated, and we then update the leftmost points of those nodes before we solve queries for s'. The total time we update the tree nodes for all queries is proportional to the total size of all trees, which is $O(n^{1+\epsilon}\log^{2+\epsilon} n)$.

In summary, after $O(n^{4/3})$ time preprocessing, computing j^* for all $O(n^{4/3})$ outside-hull segment queries can be done using $O(n^{4/3})$ comparisons.

The second step: computing i^*. For computing i^*, the idea is similar and we only sketch it. In the preprocessing, we build the same tree T as above for the first step. One change is that we add the first point p of the subsequence Q_e^i to the end of Q_e^{i-1}, i.e., p appears in both Q_e^i and Q_e^{i-1}. This does not change the complexities asymptotically.

For each query, to compute i^*, consider the root v. Observe that i^* is in Q_e^i if and only if $x_i < x(a) \leq x_{i+1}$ (x_i and x_{i+1} are defined in the same way as before). As such, we can apply the basic search lemma to find i^* in $O(1 - m^{\epsilon}\Delta\Phi)$ comparisons. We can use the same approach as above to update the leftmost points of convex hulls of nodes of the trees T (i.e., computing a sorted list L and process the queries of the segments following their order in L).

In summary, after $O(n^{4/3})$ time preprocessing, computing i^* for all $O(n^{4/3})$ outside-hull segment queries can be done using $O(n^{4/3})$ comparisons.

The third step. The third step is to find the point p_s of $Q_e[i^*, j^*]$ closest to $\ell(s)$, where $\ell(s)$ is the supporting line of s. We first discuss the preprocessing step on Q_e.

We build a balanced binary search tree T_e whose leaves corresponding to the points of $Q_e = \{p_1, p_2, \ldots, p_m\}$ in their index order as discussed before. For each node v of T_e, we use $Q_e(v)$ to denote the set of points in the leaves of the subtree rooted at v. For each node v of T_e, we explicitly store the convex hull of $Q_e(v)$ at v. Further, for each leaf v, which stores a point p_i of Q_e, for each ancestor u of v, we compute the convex hull $\mathsf{CH}_r(v, u)$ of all points $p_i, p_{i+1}, \ldots, p_j$, where p_j is the point in the rightmost leaf of the subtree at u. We do this in a bottom-up manner starting from v following the path from v to u. More specifically, suppose we are currently at a node w, which is v initially. Suppose we have the convex hull $\mathsf{CH}_r(v, w)$. We proceed on the parent w' of w as follows. If w is the right child of w', then $\mathsf{CH}_r(v, w')$ is $\mathsf{CH}_r(v, w)$ and thus we do nothing. Otherwise, we merge $\mathsf{CH}_r(v, w)$ with the convex hull of $Q_e(w'')$ at w'', where w'' is the right child of w'. Since points of $\mathsf{CH}_r(v, w)$ are separated from points of $Q_e(w'')$ by a line perpendicular to e [7], we can merge the two hulls by computing their common tangents in $O(\log m)$ time [33]. We use a persistent tree to maintain the convex hulls (e.g., by a path-copying method) [19, 34] so that after the merge we still keep $\mathsf{CH}_r(v, w)$. In this way, we have computed $\mathsf{CH}_r(v, w')$ and we then proceed on the parent of w'. We do this until we reach the root. As such, the total time and extra space for computing the convex hulls for a leaf v is $O(\log^2 m)$, and the total time and space for doing this for all leaves is $O(m \log^2 m)$. Symmetrically, for each leaf v, which stores a point p_i of Q_e, for each ancestor u of v, we compute the convex hull $\mathsf{CH}_l(v, u)$ of all points $p_h, p_{h+1}, \ldots, p_i$, whether p_h is the point in the leftmost leaf of the subtree at u. Computing the convex hulls $\mathsf{CH}_l(v, u)$ for all ancestors u for all leaves v can be done in $O(m \log^2 m)$ in a similar way as above. In addition, we construct a lowest common ancestor (LCA) data structure on the tree T_e in $O(m)$ time so that the LCA of any two query nodes of T_e can be found in $O(1)$ time [5, 26]. The total preprocessing time for constructing the tree T_e as above is $O(m \log^2 m)$. Recall that the sum of m for all outside-hull segment query data structures is $O(n \log n)$. Hence, the total preprocessing time of all data structures is $O(n \log^3 n)$.

Now consider the third step of the query algorithm. Suppose i^* and j^* are known. The problem is to compute the point p_s of $Q_e[i^*, j^*]$ closest to the supporting line $\ell(s)$ of s. Let u and v be the two leaves of T_e storing the two points p_{i^*} and p_{j^*}, respectively. Let w be the lowest common ancestor of u and v. Let u' and v' be the left and right children of w, respectively. It is not difficult to see that the convex hull of $\mathsf{CH}_r(u, u')$ and $\mathsf{CH}_l(v, v')$ is the convex hull of $Q_e[i^*, j^*]$. As such, to find p_s, it suffices to compute the vertex of $\mathsf{CH}_r(u, u')$ closest to $\ell(s)$ and the vertex of $\mathsf{CH}_l(v, v')$ closest to $\ell(s)$, and among the two points, return the one closer to $\ell(s)$ as p_s. To implement the algorithm, finding w can be done in $O(1)$ time using the LCA data structure [5, 26]. To find the closest vertex of $\mathsf{CH}_r(u, u')$ to $\ell(s)$, recall that the preprocessing computes a balanced binary search tree (maintained by a persistent tree), denoted by $T_r(u, u')$, for maintaining $\mathsf{CH}_r(u, u')$. We apply a *search lemma* of Chan and Zheng (Lemma A.1 [10]) on the tree $T_r(u, u')$. Indeed, the problem is equivalent to finding the predecessor of the slope of $\ell(s)$ among the slopes of the edges of $\mathsf{CH}_r(u,u')$. Using the search lemma, we can find the vertex of $\mathsf{CH}_r(u,u')$ closest to $\ell(s)$ using $O(1-\Delta\Phi)$ comparisons. Similarly, the vertex of $\mathsf{CH}_l(v,v')$ closest to $\ell(s)$ can be found using $O(1-\Delta\Phi)$ comparisons. In this way, p_s can be computed using $O(1-\Delta\Phi)$ comparisons.

In summary, with $O(n\log^3 n)$ time preprocessing, the third step of the query algorithm for all $O(n^{4/3})$ queries can be done using a total of $O(n^{4/3})$ comparisons (recall that the sum of $-\Delta\Phi$ in the entire algorithm is $O(n\log n)$).

Summary. Combining the three steps discussed above, all $O(n^{4/3})$ outside-hull segment queries can be solved using $O(n^{4/3})$ comparisons. Recall that the above only discussed the query on the data structure for a single edge e of the convex hull of Q. As the first procedure of the query, we need to find the vertex of $\mathsf{CH}(Q)$ closest to the supporting line of s. For this, we can maintain the convex hull $\mathsf{CH}(Q)$ by a balanced binary search tree and apply the search lemma of Chan and Zheng (Lemma A.1 [10]) in the same way as discussed above. As such, this procedure for all queries uses $O(n^{4/3})$ comparisons. The second procedure of the query is to find the edge of $\mathsf{CH}(Q)$ intersecting the line through one of the endpoints of s and perpendicular to s. This operation is essentially to find a predecessor of the above endpoint of s on the vertices of the lower hull of $\mathsf{CH}(Q)$. Therefore, we can also apply the search lemma of Chan and Zheng, and thus this procedure for all queries also uses $O(n^{4/3})$ comparisons. As such, we can solve all $O(n^{4/3})$ outside-hull segment queries using $O(n^{4/3})$ comparisons, or alternatively, we have an algebraic decision tree of height $O(n^{4/3})$ that can solve all $O(n^{4/3})$ queries.

3.4 Solving the subproblems $T(n^{2/3}, n^{1/3})$

We now tackle the second challenge, i.e., solve each subproblem $T(n^{2/3}, n^{1/3})$ in recurrence (5) using $O(n^{2/3})$ comparisons, or solve all $O(n^{2/3})$ subproblems $T(n^{2/3}, n^{1/3})$ in (5) using $O(n^{4/3})$ comparisons.

Recall that P is the set of n points and S is the set of n segments for the original problem in recurrence (5). If the closest point of a segment $s \in S$ to P is an endpoint of s, then finding the closest point of s in P can be done using the Voronoi diagram of P. Hence, it suffices to find the first point of P hit by s if we drag s along the directions perpendicularly to s. There are two such directions, but in the following discussion we will only consider dragging s along the upward direction perpendicularly to s (recall that s is not vertical due to our general position assumption) and let p_s be the first point of P hit by s, since the algorithm for the downward direction is similar. As such, the goal is to compute p_s for each segment $s \in S$.

For notational convenience, let $m = n^{1/3}$ and thus we want to solve $T(m^2, m)$ using $O(m^2)$ comparisons. More specifically, we are given m points and m^2 segments; the problem is to compute for each segment s the point p_s (with respect to the m points, i.e., the first point hit by s if we drag s along the upward direction perpendicular to s). Our goal is to solve all $O(m^2)$ segment dragging queries using $O(m^2)$ comparisons after certain preprocessing. In what follows, we begin with the preprocessing algorithm.

Preprocessing. For two sets $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_m\}$ of m points each, we say that they have the same *order type* if for each i, the index order of the points of A sorted around a_i is the same as that of the points of B sorted around b_i (equivalently, in the dual plane, the index order of the dual lines intersecting the dual line of a_i is the same as that of the dual lines intersecting the dual line of b_i); the concept has been used elsewhere, e.g., [3, 10, 23]. Because constructing the arrangement of a set of m lines can be computed in $O(m^2)$ time [15], we can decide whether two sets A and B have the same order type in $O(m^2)$ time, e.g., simply follow the incremental line arrangement construction algorithm [15]. We actually build an algebraic decision tree T_D so that each node of T_D corresponds to a comparison of the algorithm. As such, the height of T_D is $O(m^2)$ and T_D has $2^{O(m^2)}$ leaves, each of which corresponds to an order type (note that the number of distinct order types is at most m^{6m} [24], but here using $2^{O(m^2)}$ as an upper bound suffices for our purpose).

Let Q be a set of m points whose order type corresponds to a leaf v of T_D. Let K_Q denote the set of the slopes of all lines through pairs of points of Q. Note that $|K_Q| = O(m^2)$. We sort the slopes of K_Q. Consider two consecutive slopes k_1 and k_2 of the sorted K_Q. In the dual plane, for any vertical line ℓ whose x-coordinate is between k_1 and k_2, ℓ intersects the dual lines of all points of Q in the same order (because k_1 and k_2 respectively are x-coordinates of two consecutive vertices of the arrangement of dual lines). This implies the following in the primal plane. Consider any two lines ℓ_1 and ℓ_2 whose slopes are between k_1 and k_2 such that all points of Q are above ℓ_i for each $i = 1, 2$. Then, the order of the lines of Q by their distances to ℓ_1 is the same as their order by the distances to ℓ_2. However, if we project all points of Q onto ℓ_1 and ℓ_2, the orders of their projections along the two lines may not be the same. To solve our problem, we need a stronger property that the above projection orders are also the same. To this end, we further refine the order type as follows.

For each pair of points q_i and q_j of Q, we add the slope of the line perpendicular to the line through p and q to K_Q. As such, the size of K_Q is still $O(m^2)$. Although K_Q has $O(m^2)$ values, all these values are defined by the m points of Q. Using this property, K_Q can be sorted using $O(m^2)$ comparisons [10, 22].

For two sets $Q = \{q_1, q_2, \ldots, q_m\}$ and $Q' = \{q'_1, q'_2, \ldots, q'_m\}$ of m points each with the same order type, we say that they have the same *refined order type* if the order of K_Q is the same as that of $K_{Q'}$, i.e., the slope of the line through q_i and q_j (resp., the slope of the line perpendicular to the line through q_i and q_j) is in the k-th position of the sorted list of K_Q if and only if the slope of the line through q'_i and q'_j (resp., the slope of the line perpendicular to the line through q'_i and q'_j) is in the k-th position of the sorted list of $K_{Q'}$. We further enhance the decision tree T_D by attaching a new decision tree at each leaf v of T_D for sorting K_Q (recall that K_Q can be sorted using $O(m^2)$ comparisons, i.e., there is an algebraic decision tree of height $O(m^2)$ that can sort K_Q), where Q is a set of m points whose order type corresponds to v. We still use T_D to refer to the new tree. The height of T_D is still $O(m^2)$.

We perform the following preprocessing work for each leaf v of T_D. Let Q be a set of m points that has the refined order type of v. We associate Q with v, compute and sort K_Q, and store the sorted list using a balanced binary search tree. Let k_1 and k_2 be two consecutive slopes in the sorted list of K_Q. Consider a line ℓ whose slope is in (k_1, k_2) such that ℓ is below all points of Q. We project all points perpendicularly onto ℓ. According to the definition of K_Q, the order of the projections is fixed for all such lines ℓ whose slopes are in (k_1, k_2). Without loss of generality, we assume that ℓ is horizontal. Let $q_1, q_2, \ldots, q_m$ denote the points of Q ordered by their projections on ℓ from left to right and we maintain the sorted list in a balanced binary search tree. For each pair (i, j) with $1 \le i \le j \le m$, let $Q[i, j] = \{q_i, q_{i+1}, \ldots, q_j\}$; we sort all points of $Q[i, j]$ by their distances to ℓ and store the sorted list in a balanced binary search tree. As such, the time we spent on the preprocessing at v is $O(m^5 \log m)$.

Since T_D is a decision tree of height $O(m^2)$, the number of leaves of T_D is $2^{O(m^2)}$. Therefore, the total preprocessing time for all leaves of T_D is $m^5 \log m \cdot 2^{O(m^2)}$. As T_D can be built in $O(2^{\mathrm{poly}(m)})$ time, the total preprocessing time is bounded by $O(2^{\mathrm{poly}(m)})$.

Solving a subproblem $T(m^2, m)$. Consider a subproblem $T(m^2, m)$ with a set P' of m points and a set S' of m^2 segments. We arbitrarily assign indices to points of P' as $\{p_1, p_2, \ldots, p_m\}$. By using the decision tree T_D, we first find the leaf v of T_D that corresponds to the refined order type of P', which can be done using $O(m^2)$ comparisons as the height of T_D is $O(m^2)$. Let $Q = \{q_1, q_2, \ldots, q_m\}$ be the set of m points associated with v. Below we find for each segment $s \in S'$ its point p_s in P'. Let ℓ denote the supporting line of s.

We first find two consecutive slopes k_1 and k_2 in $K_{P'}$ such that the slope of ℓ is in $[k_1, k_2)$. Note that we do not explicitly have the sorted list of $K_{P'}$, but recall that we have the sorted list of K_Q stored at v. Since P' and Q have the same refined order type, a slope defined by two points p_i and p_j is in the k-th position of $K_{P'}$ if and only if the slope defined by two points q_i and q_j is in the k-th position of K_Q. Hence, we can search K_Q instead; however, whenever we need to use a slope whose definition involves a point $q_i \in Q$, we use p_i instead. In this way, we could find k_1 and k_2 using $O(\log m)$ comparisons. Further, since we have the balanced binary search tree storing K_Q, we can apply the search lemma of Chan and Zheng [10] as discussed above to find k_1 and k_2 using only $O(1 - \Delta\Phi)$ comparisons.

Without loss of generality, we assume that s is horizontal. Let a and b denote the left and right endpoints of s, respectively. Suppose we project all points of P' perpendicularly onto ℓ. Let $p_{\pi(1)}, p_{\pi(2)}, \ldots, p_{\pi(m)}$ be the sorted list following their projections along ℓ from left to right, where $\pi(i)$ is the index of the i-th point in this order. We wish to find the index i such that a is between $p_{\pi(i-1)}$ and $p_{\pi(i)}$ as well as the index j such that b is between $p_{\pi(j)}$ and $p_{\pi(j+1)}$. To this end, we do the following. Since P' and Q have the same refined order type, if we project all points of Q perpendicularly onto ℓ, then $q_{\pi(1)}, q_{\pi(2)}, \ldots, q_{\pi(m)}$ is the sorted list following their projections along ℓ with the same permutation $\pi(\cdot)$. Hence, to find the index i, we can query a in the sorted list $q_{\pi(1)}, q_{\pi(2)}, \ldots, q_{\pi(m)}$, which is maintained at v due to our preprocessing, but again, whenever we need to use a point $q_{\pi(k)}$, we use $p_{\pi(k)}$ instead. Using the search lemma of Chan and Zheng as discussed before, we can find i using $O(1 - \Delta\Phi)$ comparisons. Similarly, the index j can be found using $O(1 - \Delta\Phi)$ comparisons.

Let $P'_\ell[i, j] = \{p_{\pi(i)}, p_{\pi(i+1)}, \ldots, p_{\pi(j)}\}$. By the definitions of i and j, the point p_s we are looking for is the point of $P'_\ell[i, j]$ closest to the line ℓ. To find p_s, we do the following. Let ℓ' be a line parallel to ℓ but is below all points of P' and Q. Let $P'_{\ell'}[i, j]$ denote the sorted list of $P'_\ell[i, j]$ ordered by their distances from ℓ'. Then, p_s can be found by binary search on $P'_{\ell'}[i, j]$. Since P' and Q have the same refined order type, we can instead do binary search on $Q_{\ell'}[i, j]$, whose order is consistent with that of $Q[i, j]$, which is maintained at v due to the preprocessing. As such we can search $Q[i, j]$, but again whenever the algorithm wants to use a point $q_k \in Q[i, j]$, we will use p_k instead to perform a comparison. Using the search lemma of Chan and Zheng, we can find p_s using $O(1 - \Delta\Phi)$ comparisons.

The above shows that p_s can be found using $O(1 - \Delta\Phi)$ comparisons. Therefore, doing this for all $O(m^2)$ segments can be done using $O(m^2 - \Delta\Phi)$ comparisons.

In summary, with $O(2^{\mathrm{poly}(n)})$ time preprocessing, we can solve each subproblem $T(n^{2/3}, n^{1/3})$ using $O(n^{2/3})$ comparisons without considering the term $-\Delta\Phi$, whose total sum in the entire algorithm of recurrence (5) is $O(n \log n)$.

3.5 Wrapping things up

The above proves Lemma 1, and thus $T(n, n)$ in (5) can be bounded by $O(n^{4/3})$ after $O(2^{\mathrm{poly}(n)})$ time preprocessing as discussed before. Equivalently, $T(b, b)$ in (4) can be bounded by $O(b^{4/3})$ after $O(2^{\mathrm{poly}(b)})$ time preprocessing. Notice that the preprocessing work is done only once and for all subproblems $T(b, b)$ in (4). Since $b = (\log\log\log n)^3$, we have $2^{\mathrm{poly}(b)} = O(n)$. As such, $T(n, n)$ in (4) solves to $O(n^{4/3})$ and we have the following.

▶ **Theorem 7.** *Given a set of n points and a set of n segments in the plane, we can find for each segment its closest point in $O(n^{4/3})$ time.*

The following solves the asymmetric case of the problem (see the full paper for details).

▶ **Corollary 8.** *Given a set of n points and a set of m segments in the plane, we can find for each segment its closest point in $O(n^{2/3}m^{2/3} + n \log n + m \log^2 n)$ time.*

4 The online query problem

Let P be a set of n points in the plane. We wish to build a data structure so that the point of P closest to a query segment can be computed efficiently.

4.1 The line query problem

We first consider the special case where the query segment is a line ℓ. The main idea is to adapt the simplex range searching data structures [9, 29, 30] (which works in any fixed dimensional space; but for our purpose it suffices to only consider half-plane range counting queries in the plane). Each of these half-plane range counting query data structures [9, 29, 30] defines canonical subsets of P and usually only maintains the cardinalities of them. To solve our problem, roughly speaking, the change is that we compute and maintain the convex hulls of these canonical subsets, which increases the space by a factor proportional to the height of the underlying trees (which is $O(\log n)$ for the data structures in [9, 30] and is $O(\log\log n)$ for the one in [29]). To answer a query, we follow the similar algorithms as half-plane range counting queries on these data structures. The difference is that for certain canonical subsets, we do binary search on their convex hulls to find their closest vertices to the query line, which does not intersect these convex hulls (in the half-plane range counting query algorithms only the cardinalities of these canonical subsets are added to a total count). This increases the query time by a logarithmic factor comparing to the original half-plane range counting query algorithms. We manage to reduce the additional logarithmic factor using fractional cascading [14] on the data structures of [9, 30] because each node in the underlying trees of these data structures has $O(1)$ children. Some extra efforts are also needed to achieve the claimed performance. Finally, the trade-off is obtained by combining these results with cuttings in the dual space.

In the rest of this subsection, we present a randomized result based on Chan's partition tree [9] while the deterministic results are given in the full paper.

A randomized result based on Chan's partition tree [9]. We first review Chan's partition tree [9]. Chan's partition tree T for the point set P is a tree structure by recursively subdividing the plane into triangles. Each node v of T is associated with a triangle $\triangle(v)$, which is the entire plane if v is the root. If v is an internal node, it has $O(1)$ children, whose associated triangles form a disjoint partition of $\triangle(v)$. Let $P(v) = P \cap \triangle(v)$, i.e., the subset of points of P in $\triangle(v)$. For each internal node v, the cardinality $|P(v)|$ is stored at v. If v is a leaf, then $|P(v)| = O(1)$ and $P(v)$ is explicitly stored at v. The height of T is $O(\log n)$ and the space of T is $O(n)$. Let $\alpha(T)$ denote the maximum number of triangles $\triangle(v)$ among all nodes v of T crossed by any line in the plane. Given P, Chan's randomized algorithm can compute T in $O(n \log n)$ expected time such that $\alpha(T) = O(\sqrt{n})$ holds with high probability.

To solve our problem, we modify the tree T as follows. For each node v, we compute the convex hull $\mathsf{CH}(v)$ of $P(v)$ and store $\mathsf{CH}(v)$ at v. This increases the space to $O(n \log n)$, but the preprocessing time is still bounded by $O(n \log n)$.

Given a query line ℓ, our goal is to compute the point of P closest to ℓ. We only discuss how to find the closest point of ℓ among all points of P below ℓ since the other case is similar. Starting from the root of T, consider a node v. We assume that ℓ crosses $\triangle(v)$, which is true initially when v is the root. For each child u of v, we do the following. If ℓ crosses $\triangle(u)$, then we proceed on u recursively. Otherwise, if $\triangle(u)$ is below ℓ, we do binary search on the convex hull $\mathsf{CH}(u)$ to find in $O(\log n)$ time the closest point to ℓ among the vertices of $\mathsf{CH}(u)$ and keep the point as a candidate. Since each internal node of T has $O(1)$ children, the algorithm eventually finds $O(\alpha(T))$ candidate points and among them we finally return the one closest to ℓ as our solution. The total time of the algorithm is $O(\alpha(T) \cdot \log n)$.

To further reduce the query time, we observe that all nodes v whose triangles $\triangle(v)$ are crossed by ℓ form a subtree T_ℓ of T containing the root. This is because if the triangle $\triangle(v)$ of a node v is crossed by ℓ, then the triangle $\triangle(u)$ is also crossed by ℓ for any ancestor u of v. In light of the observation, we can further reduce the query algorithm time to $O(\alpha(T)+\log n)$ by constructing a fractional cascading structure [14] on the convex hulls of all nodes of T so that if a tangent to the convex hull at a node v is known, then the tangents of the same slope to the convex hulls of the children of v can be found in constant time. The total time for constructing the fractional cascading structure is linear in the total size of all convex hulls, which is $O(n \log n)$. With the fractional cascading structure, we only need to perform binary search on the convex hull at the root and then spend only $O(1)$ time on each node of T_ℓ and each of their children. As such, the query time becomes $O(\alpha(T) + \log n)$, which is bounded by $O(\sqrt{n})$ with high probability.

▶ **Lemma 9.** *Given a set P of n points in the plane, we can build a data structure of $O(n \log n)$ space in $O(n \log n)$ expected time such that for any query line its closest point in P can be computed in $O(\sqrt{n})$ time with high probability.*

4.2 The segment query problem

To answer the general segment queries, the main idea is essentially the same as the line case with one change: whenever we compute the convex hull for a canonical subset of P (e.g., the subset $P(v)$ for a node v in a partition tree) for outside-hull line queries, we instead build the BS data structure [7] for outside-hull segment queries. Because the fractional cascading does not help anymore, the query time in general has an additional logarithmic factor, with the exception that when using Chan's partition tree [9] we still manage to bound the query time by $O(\sqrt{n})$ due to some nice properties of the partition tree.

In the rest of this subsection, we present a randomized result based on Chan's partition tree [9] while the deterministic results are given in the full paper.

The randomized result. For our randomized result using Chan's partition tree [9] (by modifying the one in Section 4.1), for each node v of the partition tree T, we construct the BS data structure for $P(v)$. The total space is still $O(n \log n)$. For the preprocessing time, constructing the BS data structure can be done in linear time if we know the Voronoi diagram of $P(v)$. For this, as discussed in Section 3.1, we can process all nodes of T in a bottom-up manner and using the linear-time Voronoi diagram merge algorithm of Kirkpatrick [27]. As such, constructing the BS data structures for all nodes of T can be done in $O(n \log n)$ time in total. Therefore, the total preprocessing time is still $O(n \log n)$ expected time.

The query algorithm follows the same scheme as before but instead use the BS algorithm to answer outside-hull segment queries. The total query time becomes $O(\sqrt{n} \log n)$ with high probability. In fact, due to certain properties of Chan's partition tree, the time is bounded by $O(\sqrt{n})$, as shown in the following lemma (similar idea was used elsewhere, e.g., [11]).

▶ **Lemma 10.** *The query time is bounded by $O(\sqrt{n})$ with high probability.*

As such, we obtain the following result.

▶ **Lemma 11.** *Given a set P of n segments in the plane, we can build a data structure of $O(n \log n)$ space in $O(n \log n)$ expected time such that for any query segment its closest point in P can be computed in $O(\sqrt{n})$ time with high probability.*

As discussed in Section 1, another randomized solution of complexity $O(n^{4/3}, n^{4/3}, n^{1/3})$ can be obtained using Chan's randomized techniques [8] and Chan and Zheng's recent randomized result on triangle range counting [10]. Refer to the full paper for details. We thank an anonymous reviewer for suggesting the idea.

References

1 Pankaj K. Agarwal. Partitioning arrangements of lines II: Applications. *Discrete and Computational Geometry*, 5:533–573, 1990.

2 Pankaj K. Agarwal and Micha Sharir. Applications of a new space-partitioning technique. *Discrete and Computational Geometry*, 9:11–38, 1993.

3 Boris Aronov, Mark de Berg, Jean Cardinal, Esther Ezra, John Iacono, and Micha Sharir. Subquadratic algorithms for some 3Sum-Hard geometric problems in the algebraic decision tree model. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*, pages 3:1–3:15, 2021.

4 Reuven Bar-Yehuda and Sergio Fogel. Variations on ray shootings. *Algorithmica*, 11:133–145, 1994.

5 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

6 Sergei Bespamyatnikh. Computing closest points for segments. *International Journal of Computational Geometry and Application*, 13:419–438, 2003.

7 Sergei Bespamyatnikh and Jack Snoeyink. Queries with segments in Voronoi diagrams. *Computational Geometry: Theory and Applications*, 16:23–33, 2000.

8 Timothy M. Chan. Geometric applications of a randomized optimization technique. *Discrete and Computational Geometry*, 22:547–567, 1999.

9 Timothy M. Chan. Optimal partition trees. *Discrete and Computational Geometry*, 47:661–690, 2012.

10 Timothy M. Chan and Da Wei Zheng. Hopcroft's problem, log-star shaving, 2D fractional cascading, and decision trees. In *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 190–210, 2022.

11 Timothy M. Chan and Da Wei Zheng. Simplex range searching revisited: How to shave logs in multi-level data structures. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1493–1511, 2023.

12 Bernard Chazelle. An algorithm for segment-dragging and its implementation. *Algorithmica*, 3:205–221, 1988.

13 Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9:145–158, 1993.

14 Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

15 Bernard Chazelle, Leonidas J. Guibas, and D.T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

16 Siu Wing Cheng and Ravi Janardan. Algorithms for ray-shooting and intersection searching. *Journal of Algorithms*, 13:670–692, 1992.

17 Richard Cole and Chee-Keng Yap. Geometric retrieval problems. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 112–121, 1983.

18 Ovidiu Daescu, Ningfang Mi, Chan-Su Shin, and Alexander Wolff. Farthest-point queries with geometric and combinatorial constraints. *Computational Geometry: Theory and Applications*, 33:174–185, 2006.

19 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

20 Herbert Edelsbrunner, Leonidas J. Guibas, and Micha Sharir. The complexity and construction of many faces in arrangement of lines and of segments. *Discrete and Computational Geometry*, 5:161–196, 1990.

21 Jeff Erickson. New lower bounds for Hopcroft's problem. *Discrete and Computational Geometry*, 16:389–418, 1996.

22 Michael L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

23 Jacob E. Goodman and Richard Pollack. Multidimensional sorting. *SIAM Journal on Computing*, 12:484–507, 1983.

24 Jacob E. Goodman and Richard Pollack. Upper bounds for configurations and polytopes in R^d. *Discrete and Computational Geometry*, pages 219–227, 1986.

25 Partha P. Goswami, Sandip Das, and Subhas C. Nandy. Triangular range counting query in 2D and its application in finding k nearest neighbors of a line segment. *Computational Geometry: Theory and Applications*, 29:163–175, 2004.

26 Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.

27 David G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 18–27, 1979.

28 D. T. Lee and Y. T. Ching. The power of geometric duality revisited. *Information Processing Letters*, 21:117–122, 1985.

29 Jiří Matoušek. Efficient partition trees. *Discrete and Computational Geometry*, 8:315–334, 1992.

30 Jiří Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry*, 10:157–182, 1993.

31 Pinaki Mitra and Bidyut B. Chaudhuri. Efficiently computing the closest point to a query line. *Pattern Recognition Letters*, 19:1027–1035, 1998.

32 Asish Mukhopadhyay. Using simplicial paritions to determine a closest point to a query line. *Pattern Recognition Letters*, 24:1915–1920, 2003.

33 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

34 Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

35 Haitao Wang. Algorithms for subpath convex hull queries and ray-shooting among segments. In *Proceedings of the 36th International Symposium on Computational Geometry (SoCG)*, pages 69:1–69:14, 2020.