Constructing Many Faces in Arrangements of Lines and Segments*

Haitao Wang[†]

Abstract

We present new algorithms for computing many faces in arrangements of lines and segments. Given a set S of n lines (resp., segments) and a set P of m points in the plane, the problem is to compute the faces of the arrangements of S that contain at least one point of P. For the line case, we give a deterministic algorithm of $O(m^{2/3}n^{2/3}\log^{2/3}(n/\sqrt{m}) + (m+n)\log n)$ time. This improves the previously best deterministic algorithm [Agarwal, 1990] by a factor of $\log^{2.22} n$ and improves the previously best randomized algorithm [Agarwal, Matoušek, and Schwarzkopf, 1998] by a factor of $\log^{1/3} n$ in certain cases (e.g., when $m = \Theta(n)$). For the segment case, we present a deterministic algorithm of $O(n^{2/3}m^{2/3}\log n + \tau(n\alpha^2(n) + n\log m + m)\log n)$ time, where $\tau = \min\{\log m, \log(n/\sqrt{m})\}$ and $\alpha(n)$ is the inverse Ackermann function. This improves the previously best deterministic algorithm [Agarwal, 1990] by a factor of $\log^{2.11} n$ and improves the previously best randomized algorithm [Agarwal, Matoušek, and Schwarzkopf, 1998] by a factor of $\log n$ in certain cases (e.g., when $m = \Theta(n)$). We also give a randomized algorithm of $O(m^{2/3}K^{1/3}\log n + \tau(n\alpha(n) + n\log m + m)\log n\log K)$ expected time, where K is the number of intersections of all segments of S. In addition, we consider the query version of the problem, that is, preprocess S to compute the face of the arrangement of S that contains any query point. We present new results that improve the previous work for both the line and the segment cases.

1 Introduction.

We consider the problem of computing many faces in arrangements of lines and segments. Given a set S of n lines (resp., segments) and a set P of m points in the plane, the problem is to compute the faces of the arrangement of S that contain at least one point of P. Note that faces in an arrangement of lines are convex, but they may not even be simply connected in an arrangement of segments. These are classical problems in computational geometry and have been studied in the literature. There has been no progress on these problems for more than two decades. In this paper, we present new algorithms that improve the previous work.

The line case. For the line case where S consists of n lines, it has been proved that the combinatorial complexity of all faces of the arrangement that contain at least one point of P is bounded by $O(m^{2/3}n^{2/3}+n)$ [14] (which matches the $\Omega(m^{2/3}n^{2/3}+n)$ lower bound [22]), as well as bounded by $O(n\sqrt{m})$ and $O(n+m\sqrt{n})$ [22]. To compute these faces, a straightforward approach is to first construct the arrangement of S and then find the faces using point locations [19, 26]. This takes $O(n^2 + m\log n)$ time in total. Edelsbrunner, Guibas, and Sharir [20] gave a randomized algorithm of $O(m^{2/3-\delta}n^{2/3+2\delta}\log n + n\log n\log m)$ expected time for any $\delta > 0$. Later Agarwal [1] presented an improved deterministic algorithm of $O(m^{2/3}n^{2/3}\log^{5/3} n\log^{1.11}(m/\sqrt{n}) + (m+n)\log n)$ time; Agarwal, Matoušek, and Schwarzkopf [2] proposed a randomized algorithm of $O(m^{2/3}n^{2/3}\log(n/\sqrt{m}) + (m+n)\log n)$ expected time. On the other hand, the problem has a lower bound of $\Omega(m^{2/3}n^{2/3} + n\log n + m)$ time due to the above $\Omega(m^{2/3}n^{2/3} + n)$ lower bound [22] on the combinatorial complexity of all these faces and also because computing a single face in line arrangements requires $\Omega(n\log n)$ time.

We propose a new deterministic algorithm of $O(m^{2/3}n^{2/3}\log^{2/3}(n/\sqrt{m}) + (m+n)\log n)$ time. In certain cases (e.g., when $m = \Theta(n)$), our result improves the deterministic algorithm of [1] by a factor of $\log^{2.22} n$ and improves the randomized algorithm of [2] by a factor of $\log^{1/3} n$.

Our algorithm follows the framework of Agarwal [1], which uses a cutting of S to divide the problem into a collection of subproblems. To solve each subproblem, Agarwal [1] derived another algorithm of $O(n\log n + m\sqrt{n}\log^2 n)$ time. Our main contribution is a more efficient algorithm of $O(n\log n + m\sqrt{n\log n})$ time. Using our new algorithm to solve the subproblems induced by the cutting, the asserted result can be

*The full version of the paper can be accessed at https://arxiv.org/abs/2110.08669. This research was supported in part by NSF under Grant CCF-2005323.

[†]Department of Computer Science, Utah State University, Logan, UT 84322, USA. Email: haitao.wang@usu.edu

achieved. The algorithm of [2] also follows a similar framework, but it uses the random sampling technique [15] instead of the cutting to divide the problem, and a randomized algorithm of $O(n \log n + m\sqrt{n} \log n)$ expected time was proposed in [2] to solve each subproblem. In particular, our algorithm runs in $O(n \log n)$ time for $m = O(\sqrt{n \log n})$, which matches the $\Omega(n \log n)$ lower bound for computing a single face (for comparison, the randomized algorithm of [2] runs in $O(n \log n)$ expected time for $m = O(\sqrt{n})$).

The segment case. For the segment case where S consists of n line segments, it is known that the combinatorial complexity of all faces of the arrangement that contain at least one point of P is upper bounded by $O(m^{2/3}n^{2/3} + n\alpha(n) + n \log m)$ [5] and $O(\sqrt{m} n\alpha(n))$ [18], as well as lower bounded by $\Omega(m^{2/3}n^{2/3} + n\alpha(n))$ [20], where $\alpha(n)$ is the inverse Ackermann function. To compute these faces, as in the line case, a straightforward approach is to first construct the arrangement of S and then find the faces using point locations [19, 26]. This takes $O(n^2 + m \log n)$ time in the worst case (more precisely, the arrangement can be constructed in $O(n \log n + K)$ time [11, 6] or by simpler randomized algorithms of the same expected time [15, 12, 29]; throughout the paper, we use K to denote the number of intersections of all segments of S).

Edelsbrunner, Guibas, and Sharir [20] gave a randomized algorithm of $O(m^{2/3-\delta}n^{2/3+2\delta} \log n + n\alpha(n) \log^2 n \log m)$ expected time for any $\delta > 0$. Agarwal [1] presented an improved deterministic algorithm of $O(m^{2/3}n^{2/3} \log n \log^{2.11}(n/\sqrt{m}) + n \log^3 n + m \log n)$ time. Agarwal, Matoušek, and Schwarzkopf [2] derived a randomized algorithm of $O(n^{2/3}m^{2/3} \log^2(K/m) + (n\alpha(n) + n \log m + m) \log n)$ expected time and another algorithm of $O(m^{2/3}K^{1/3} \log^2(K/m) + (n\alpha(n) + n \log m + m) \log n)$ expected time[1]. On the other hand, the lower bound $\Omega(m^{2/3}n^{2/3} + n \log n + m)$ for the line case is also applicable here (and we are not aware of any better lower bound). Note that computing a single face in an arrangement of segments can be done in $O(n\alpha(n) \log n)$ expected time by a randomized algorithm [12] or in $O(n\alpha^2(n) \log n)$ time by a deterministic algorithm [4] (which improve the previous $O(n \log^2 n)$ time algorithm [28] and $O(n\alpha(n) \log^2 n)$ time algorithm [20]; but computing the upper envelope can be done faster in $O(n \log n)$ time [25]).

We propose a new deterministic algorithm of $O(n^{2/3}m^{2/3} \log n + \tau(n\alpha^2(n) + n \log m + m) \log n)$ time, where $\tau = \min\{\log m, \log(n/\sqrt{m})\}$. In certain cases (e.g., when $m = \Theta(n)$ and $K = \Theta(n^2)$), our result improves the deterministic algorithm of [1] by a factor of $\log^{2.11} n$ and improves the randomized algorithm of [2] by a factor of $\log n$. In particular, the algorithm runs in $O(n\alpha^2(n) \log n)$ time for $m = O(1)$, which matches the time for computing a single face [4], and runs in $O(m \log n)$ time for $m = \Theta(n^2)$, which matches the performance of the above straightforward approach. Our algorithm uses a different approach than the previous work [1, 2]. In particular, our above algorithm for the line case is utilized as a subroutine.

If $K = o(n^2)$, we further obtain a faster randomized algorithm of $O(m^{2/3}K^{1/3} \log n + \tau(n\alpha(n) + n \log m + m) \log n \log K)$ expected time, where $\tau = \min\{\log m, \log(n/\sqrt{m})\}$. This improves the result of [2] by a factor of $\log n$ for relative large values of K, e.g., when $m = \Theta(n)$ and $K = \Omega(n^{1+\epsilon})$ for any constant $\epsilon \in (0, 1]$. Our above deterministic algorithm (with one component replaced by a faster randomized counterpart) is utilized as a subroutine.

The face query problem. We also consider a face query problem in which we wish to preprocess S so that given a query point p, the face of the arrangement containing p can be computed efficiently.

For the line case, inspired by our techniques for computing many faces and utilizing the randomized partition tree of Chan [9], we construct a data structure of $O(n \log n)$ space in $O(n \log n)$ randomized time, so that the face $F_p(S)$ of the arrangement of S that contains a query point p can be computed and the query time is bounded by $O(\sqrt{n} \log n)$ with high probability. More specifically, the query algorithm returns a binary search tree representing the face $F_p(S)$ so that standard binary-search-based queries on $F_p(S)$ can be handled in $O(\log n)$ time each, and $F_p(S)$ can be output explicitly in $O(|F_p(S)|)$ time. Previously, Edelsbrunner, Guibas, Hershberger, Seidel, Sharir, Snoeyink, and Welzl [17] built a data structure of $O(n \log n)$ space in $O(n^{3/2} \log^2 n)$ randomized time, and the query time is bounded by $O(\sqrt{n} \log^5 n)$ with high probability, which is reduced to $O(\sqrt{n} \log^2 n)$ in [23] using compact interval trees. Thus, our result improves their preprocessing time by a factor of $\sqrt{n} \log n$ and improves their query time by a factor of $\log n$. We further obtain a tradeoff between the storage and the query time. For any value $r < n/\log^{\omega(1)} n$, we construct a data structure of $O(n \log n + nr)$ space in $O(n \log n + nr)$ randomized time, and the query time is bounded by $O(\sqrt{n/r} \log n)$ with high probability.

[1]It appears that their time analysis [2] is based on the assumption that K is known. If K is not known, their algorithm could achieve $O(m^{2/3}K^{1/3} \log^2(K/m) + (m + n \log m + n\alpha(n)) \log n \log K)$ expected time by the standard trick of "guessing", which is also used in this paper.

For the segment case, the authors [17] also gave a data structure for the face query problem with the following performance: the preprocessing takes $\widetilde{O}(n^{5/3})$ randomized time, the space is $\widetilde{O}(n^{4/3})$, and the query time is bounded by $\widetilde{O}(n^{1/3}) + O(\kappa)$ with high probability, where the notation $\widetilde{O}$ hides a polylogarithmic factor and κ is the size of the query face (note that κ can be $\Theta(n\alpha(n))$ in the worst case [20] and the face may not be simply connected). Their preprocessing algorithm uses the query algorithm for the line case as a subroutine. If we follow their algorithmic scheme but instead use our new query algorithm for the line case as the subroutine, then the preprocessing time can be reduced to $\widetilde{O}(n^{4/3})$, while the space is still $\widetilde{O}(n^{4/3})$ and the query time is still bounded by $\widetilde{O}(n^{1/3}) + O(\kappa)$ with high probability.

Outline. The rest of the paper is organized as follows. We define notation and introduce some concepts in Section 2. Our algorithms for computing many faces are described in Sections 3 and 4. The query problem is discussed in Section 5. Many proofs and details are omitted but can be found in the full version.

2 Preliminaries.

We define some notation that is applicable to both the line and segment cases. Let S be a set of n line segments (a line is considered a special line segment) and let P be a set of m points in the plane. For a subset $S' \subseteq S$, we use $\mathcal{A}(S')$ to denote the arrangement of S'. For any point $p \in P$, we use $F_p(S')$ to denote the face of $\mathcal{A}(S')$ that contains p. A face of $\mathcal{A}(S')$ is *nonempty* if it contains a point of P. Hence, the problem of computing many faces is to compute all nonempty cells of $\mathcal{A}(S)$. Note that if a nonempty face contains more than one point of P, then we need to output the face only once.

For any compact region A and a set Q of points in the plane, we often use $Q(A)$ to denote the subset of Q in A, i.e., $Q(A) = Q \cap A$.

Cuttings. Let H be a set of n lines in the plane. For a compact region A in the plane, we use H_A to denote the subset of lines of H that intersect the interior of A (we also say that these lines *cross* A). A *cutting* for H is a collection Ξ of closed cells (each of which is a triangle) with disjoint interiors, which together cover the entire plane [10, 27]. The *size* of Ξ is the number of cells in Ξ. For a parameter r with $1 \le r \le n$, a $(1/r)$-*cutting* for H is a cutting Ξ satisfying $|H_\sigma| \le n/r$ 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* (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 $k = \Theta(\log r)$ so that $\rho^{k-1} < r \le \rho^k$. Hence, the size of 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 structure 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 the sets H_σ for every cell σ of Ξ_i for all $i = 0, 1, \ldots, k$) can be computed in $O(nr)$ time by Chazelle's algorithm [10].

3 Computing many cells in arrangements of lines.

In this section, we consider the line case for computing many cells. Let S be a set of n lines and P be a set of m points in the plane. Our goal is to compute the nonempty cells of the arrangement $\mathcal{A}(S)$. For ease of exposition, we make a general position assumption that no line of S is vertical, no three lines of S are concurrent, and no point of P lies on a line of S. Degenerate cases can be handled by standard techniques [21]. Under the assumption, each point of P is in the interior of a face of $\mathcal{A}(S)$.

First of all, if $m \ge n^2/2$, then the problem can be solved in $O(m \log n)$ time using the straightforward algorithm mentioned in Section 1 (i.e., first compute $\mathcal{A}(S)$ and then find the nonempty cells using point location). In what follows, we assume that $m < n^2/2$. Our algorithm follows the high-level scheme of Agarwal [1] by using a cutting of S to divide the problem into many subproblems. The difference is that we develop an improved algorithm for solving each subproblem. In the following, we first present an algorithm of $O(n \log n + m\sqrt{n \log n})$ time in Section 3.1, and then use it to solve each subproblem and thus obtain our main algorithm with the asserted time in Section 3.2.

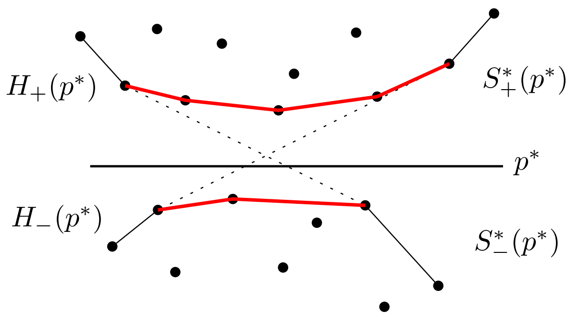

Figure 1: Illustrating the dual plane: The (red) thick edges between the two inner common tangents (the dotted segments) are dual to $F_p(S)$.

3.1 The first algorithm. We say that S and P are in the primal plane and we consider the problem in the dual plane. Let S^* be the set of dual points of S and let P^* be the set of dual lines of P.[2] Consider a point $p \in P$ and the face $F_p(S)$ of $\mathcal{A}(S)$ that contains p. In the dual plane, the dual line p^* of p partitions S^* into two subsets and the portions of the convex hulls of the two subsets between their inner common tangents are dual to the face $F_p(S)$ [1, 17]; see Fig 1.

Let $S_+^*(p^*)$ denote the subset of S^* above p^* and $S_-^*(p^*)$ the subset of S^* below p^* (note that p^* is not vertical). We use $H_+(p^*)$ to denote the half hull of the convex hull of $S_+^*(p^*)$ facing p^* (e.g., if p^* is horizontal, then $H_+(p^*)$ is the lower hull; for this reason, we call $H_+(p^*)$ the *lower hull*; see Fig 1); similarly, we use $H_-(p^*)$ to denote the half hull of the convex hull of $S_-^*(p^*)$ facing p^* and we call it the *upper hull*. According to the above discussion, $F_p(S)$ is dual to the portions of $H_+(p^*)$ and $H_-(p^*)$ between their inner common tangents, and we use $F_p^*(S)$ to denote the dual of $F_p(S)$. Our algorithm to be presented below will implicitly determine $H_+(p^*)$ and $H_-(p^*)$ (more precisely, each of them is maintained in a binary search tree of height $O(\log n)$ that can support standard binary search in $O(\log n)$ time), after which their inner common tangents can be computed in $O(\log n)$ time [23] and then $F_p^*(S)$ can be output in additional $O(|F_p^*(S)|)$ time. Again, if $F_p^*(S)$ is the same for multiple points $p \in P$, then $F_p^*(S)$ will be output only once. In the following, depending on the context, a convex hull (resp., upper hull, lower hull) may refer to a binary search tree that represents it. For example, "computing $H_+(p^*)$" means "computing a binary search tree that represents $H_+(p^*)$".

We compute a hierarchical $(1/r)$-cutting $\Xi_0, \Xi_1, \ldots, \Xi_k$ for the lines of P^* with k and a constant ρ as defined in Section 2, and with r to be determined later, along with the subsets P_σ^* of lines of P^* crossing σ for all cells σ of Ξ_i for all $i = 0, 1, \ldots, k$. This can be done in $O(mr)$ time [10]. Recall that $k = O(\log r)$. For each point $l^* \in S^*$, we find the cell $\sigma \in \Xi_i$ containing l^* for all $i = 0, 1, \ldots, k$ and store l^* in the set $S^*(\sigma)$, i.e., $S^*(\sigma) = S^* \cap \sigma$. Computing the sets $S^*(\sigma)$ for all cells $\sigma \in \Xi_i$, $i = 0, 1, \ldots, k$, takes $O(n \log r)$ time [10]. As each point of S^* is stored in a single cell of Ξ_i, for each $0 \leq i \leq k$, the total size of $S^*(\sigma)$ for all cells σ of the cutting is $O(n \log r)$. If initially we sort all points of S^* by x-coordinate, then we can obtain the sorted lists of all sets $S^*(\sigma)$ of all cells in $O(n \log r)$ time in total. Using the sorted lists, for each cell $\sigma \in \Xi_i$, $i = 0, 1, \ldots, k$, we compute the convex hull of $S^*(\sigma)$ in $O(|S^*(\sigma)|)$ time (and store it in a balanced binary search tree). All above takes $O(mr + n \log r + n \log n)$ time in total.

Next, for each cell σ of the last cutting Ξ_k, if $|S^*(\sigma)| > n/r^2$, then we further triangulate σ (which itself is a triangle) into $\Theta(|S^*(\sigma)| \cdot r^2/n)$ triangles each of which contains at most n/r^2 points of S^*. As points of $S^*(\sigma)$ are already sorted by x-coordinate, the triangulation can be easily done in $O(|S^*(\sigma)|)$ time, as follows. By sweeping the points of $S^*(\sigma)$ from left to right, we can partition σ in to $\lceil |S^*(\sigma)| \cdot r^2/n \rceil$ trapezoids each of which contains no more than n/r^2 points of S^*. Then, we partition each trapezoid into two triangles. In this way, σ is triangulated into at most $2\lceil |S^*(\sigma)| \cdot r^2/n \rceil$ triangles each containing at most n/r^2 points of S^*. Processing all cells of Ξ_k as above takes $O(n)$ time in total. For convenience, we use Ξ_{k+1} to refer to the set of all new triangles obtained above. Since Ξ_k has $O(r^2)$ cells, by our way of computing the triangles of Ξ_{k+1}, the size of Ξ_{k+1} is bounded by $O(r^2)$. For each triangle $\sigma' \in \Xi_{k+1}$, if σ' is in the cell σ of Ξ_k, then we also say that σ is the *parent* of σ' and σ' is a *child* of σ (note that the number of children of σ may not be $O(1)$). We also define $S^*(\sigma') = S^* \cap \sigma'$, and

[2]We use the following duality [7]: A point (a, b) in the primal plane is dual to the line $y = ax - b$ in the dual plane; a line $y = cx + d$ in the primal plane is dual to the point $(c, -d)$ in the dual plane.

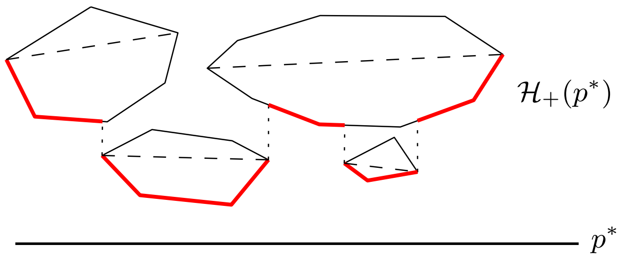

Figure 2: Illustrating the lower envelope $\mathcal{L}(\mathcal{H}_+(p^*))$ (the thick red edges) of the convex hulls of $\mathcal{H}_+(p^*)$. The dashed segment inside each convex hull is the representative segment.

compute and store the convex hull of $S^*(\sigma')$. This takes $O(n)$ time for all triangles σ' of Ξ_{k+1}, thanks to the presorting of S^*.

For reference purpose, we consider the above *the preprocessing step* of our algorithm.

For each line $p^* \in P^*$, we process it as follows. Without loss of generality, we assume that p^* is horizontal. Let $\Psi(p^*)$ denote the set of all cells σ of Ξ_i crossed by p^*, for all $i = 0, 1, \ldots, k$. Let $\Psi_{k+1}(p^*)$ denote the set of all cells σ of Ξ_{k+1} crossed by p^*. For each cell $\sigma \in \Psi_{k+1}(p^*)$, we use $\sigma_+(p^*)$ to denote the portion of σ above p^*, and let $S^*(\sigma_+(p^*)) = S^* \cap \sigma_+(p^*)$. Next we define a set $\Psi_+(p^*)$ of cells of Ξ_i, $i = 0, 1, \ldots, k+1$. For each cell $\sigma' \in \Psi(p^*)$, suppose σ' is in Ξ_i for some $i \in [0, k]$. For each child σ of σ' (thus $\sigma \in \Xi_{i+1}$), if σ is completely above the line p^*, then σ is in $\Psi_+(p^*)$. We have the following lemma, whose proof is omitted.

Lemma 3.1. *$S^*_+(p^*)$ is the union of $\bigcup_{\sigma\in\Psi_+(p^*)} S^*(\sigma)$ and $\bigcup_{\sigma\in\Psi_{k+1}(p^*)} S^*(\sigma_+(p^*))$.*

Lemma 3.1 implies that if we have convex hulls of $S^*(\sigma)$ for all cells $\sigma \in \Psi_+(p^*)$ and convex hulls of $S^*(\sigma_+(p^*))$ for all cells $\sigma \in \Psi_{k+1}(p^*)$, then $H_+(p^*)$ is the lower hull of all these convex hulls. Define $\mathcal{H}_+(p^*)$ as the set of convex hulls mentioned above. Thanks to our preprocessing step, we have the following lemma, whose proof is omitted.

Lemma 3.2. *We can obtain (binary search trees representing) the convex hulls of $\mathcal{H}_+(p^*)$ for all lines $p^* \in P^*$ in $O(mr + mn/r)$ time.*

With the preceding lemma, our next goal is to compute the lower hull of all convex hulls of $\mathcal{H}_+(p^*)$. To this end, the observation in the following lemma is critical, with proof omitted.

Lemma 3.3. *For each $p^* \in P^*$, the convex hulls of $\mathcal{H}_+(p^*)$ are pairwise disjoint.*

With the convex hulls computed in Lemma 3.2 and the property in Lemma 3.3, the next lemma computes the lower hull $H_+(p^*)$ for all $p^* \in P^*$.

Lemma 3.4. *For each $p^* \in P^*$, suppose the convex hulls of $\mathcal{H}_+(p^*)$ are available; then we can compute (a binary search tree representing) the lower hull $H_+(p^*)$ in $O(|\mathcal{H}_+(p^*)| \log n)$ time.*

Proof. Without loss of generality, we assume that p^* is horizontal. Let $t = |\mathcal{H}_+(p^*)|$. Note that the size of each convex hull of $\mathcal{H}_+(p^*)$ is at most n. Also, since convex hulls of $\mathcal{H}_+(p^*)$ are pairwise disjoint by Lemma 3.3, it holds that $t \le n$. Because we are to compute the lower hull of the convex hulls of $\mathcal{H}_+(p^*)$, it suffices to only consider the lower hull of each convex hull of $\mathcal{H}_+(p^*)$. Note that since binary search trees for all convex hulls of $\mathcal{H}_+(p^*)$ are available, we can obtain binary search trees representing their lower hulls in $O(t \log n)$ time by first finding the leftmost and rightmost vertices of the convex hulls and then performing split/merge operations on the trees.

The first step is to compute the portions of each lower hull H of $\mathcal{H}_+(p^*)$ that is vertically visible to p^* (we say that a point $q \in H$ is *vertically visible* to p^* if the vertical segment connecting q to p^* does not cross any other lower hull of $\mathcal{H}_+(p^*)$). In fact, the visible portions constitute exactly the lower envelope of the lower hulls of $\mathcal{H}_+(p^*)$, denoted by $\mathcal{L}(\mathcal{H}_+(p^*))$ (see Fig. 2). Below we describe an algorithm to compute $\mathcal{L}(\mathcal{H}_+(p^*))$ in $O(t \log n)$ time.

As convex hulls of $\mathcal{H}_+(p^*)$ are pairwise disjoint by Lemma 3.3, the number of (maximal) visible portions of all lower hulls of $\mathcal{H}_+(p^*)$ is at most $2t-1$. For each convex hull H of $\mathcal{H}_+(p^*)$, consider the segment connecting the leftmost and rightmost endpoints of H, and call it the *representative segment* of H (see Fig. 2). Let Q be the set of representative segments of all convex hulls of $\mathcal{H}_+(p^*)$. Because convex hulls of $\mathcal{H}_+(p^*)$ are pairwise disjoint, an easy but crucial observation is that segments of Q are pairwise disjoint and the lower envelope $\mathcal{L}(Q)$ of the segments of Q corresponds to $\mathcal{L}(\mathcal{H}_+(p^*))$ in the following sense: if $\overline{ab}$ is a maximal segment of $\mathcal{L}(Q)$ that lies on a representative segment of a convex hull H of $\mathcal{H}_+(p^*)$, then the vertical projection of $\overline{ab}$ onto the lower hull of H is a maximal portion of the lower hull of H on $\mathcal{L}(\mathcal{H}_+(p^*))$, and that portion can be obtained in $O(\log n)$ time by splitting the binary search tree for the lower hull of H at the x-coordinates of a and b, respectively. As such, once $\mathcal{L}(Q)$ is computed, $\mathcal{L}(\mathcal{H}_+(p^*))$ in which each maximal portion is represented by a binary search tree can be obtained in additional $O(t \log n)$ time. As $|Q| = t$ and segments of Q are pairwise disjoint, $\mathcal{L}(Q)$ can be constructed in $O(t \log t)$ time by an easy plane sweeping algorithm. Hence, $\mathcal{L}(\mathcal{H}_+(p^*))$ can be computed in $O(t \log n)$ time in total.

With $\mathcal{L}(\mathcal{H}_+(p^*))$ in hand, we can now compute the lower hull $H_+(p^*)$ in additional $O(t \log n)$ time, as follows. As discussed above, $\mathcal{L}(\mathcal{H}_+(p^*))$ consists of at most $2t-1$ *pieces* sorted from left to right, each of which is a portion of a lower hull of $\mathcal{H}_+(p^*)$ and is represented by a binary search tree. We merge the first two pieces by computing their common tangent, which can be done in $O(\log n)$ time [30] as the two pieces are separated by a vertical line. After the merge, we obtain a binary search tree that represents the lower hull of the first two pieces of $\mathcal{L}(\mathcal{H}_+(p^*))$. Next, we merge this lower hull with the third piece of $\mathcal{L}(\mathcal{H}_+(p^*))$ in the same way. We repeat this process until all pieces of $\mathcal{L}(\mathcal{H}_+(p^*))$ are merged, after which a binary search tree representing $H_+(p^*)$ is obtained. The runtime is bounded by $O(t \log n)$ as each merge takes $O(\log n)$ time and $\mathcal{L}(\mathcal{H}_+(p^*))$ has at most $2t-1$ pieces.

In summary, once the convex hulls of $\mathcal{H}_+(p^*)$ are available, we can compute the lower hull $H_+(p^*)$ in $O(|\mathcal{H}_+(p^*)| \log n)$ time. $\square$

Applying Lemma 3.4 to all lines of P^* will compute the lower hulls $H_+(p^*)$ for all $p^* \in P^*$. One issue is that after we apply the algorithm for one line $p^* \in P^*$, convex hulls of $\mathcal{H}_+(p^*)$ may have been destroyed due to the split and merge operations during the algorithm. The destroyed convex hulls may be used later when we apply the algorithm for other lines of P^*. The remedy is to use fully persistent binary search trees with path-copying [16, 31] to represent convex hulls so that standard operations on the trees (e.g., merge, split) can be performed in $O(\log n)$ time each and after each operation the original trees are still kept intact (so that future operations can still be performed on the original trees as usual). In this way, whenever we apply the algorithm for a line of P^*, we always have the original trees representing the convex hulls available, and thus the runtime of the algorithm in Lemma 3.4 is not affected (although $O(\log n)$ extra space will be incurred after each operation on the trees).

For the time analysis, by Lemma 3.2, computing convex hulls of $\mathcal{H}_+(p^*)$ for all lines $p^* \in P^*$ takes $O(mr + mn/r)$ time. Then, applying Lemma 3.4 to all lines of P^* takes $O(\sum_{p^* \in P^*} |\mathcal{H}_+(p^*)| \cdot \log n)$ time in total, which is bounded by $O(mr \log n)$ due to the following Lemma 3.5, with proof omitted.

LEMMA 3.5. $\sum_{p^* \in P^*} |\mathcal{H}_+(p^*)| = O(mr)$.

In summary, computing lower hulls $H_+(p^*)$ for all $p^* \in P^*$ can be done in a total of $O(n \log n + n \log r + mr \log n + mn/r)$ time. Analogously, we can also compute the upper hulls $H_-(p^*)$ for all $p^* \in P^*$. Then, for each line $p^* \in P^*$, we compute the two inner common tangents of $H_+(p^*)$ and $H_-(p^*)$, which can be done in $O(\log n)$ time [23]. With the two inner common tangents as well as the two hulls $H_+(p^*)$ and $H_-(p^*)$, the dual face $F_p^*(S)$, or the face $F_p(S)$ in the primal plane, can be implicitly determined. More precisely, given $H_+(p^*)$ and $H_-(p^*)$, we can obtain a binary search tree representing $F_p(S)$ in $O(\log n)$ time. The tree can be used to support standard binary search on $F_p(S)$, which is a convex polygon. Outputting $F_p(S)$ explicitly takes $O(|F_p(S)|)$ additional time.

To avoid reporting a face more than once, we can remove duplication in the following way. Due to the general position assumption, an easy observation is that $F_{p_1}(S) = F_{p_2}(S)$ for two points p_1 and p_2 of P if and only if the leftmost vertex of $F_{p_1}(S)$ is the same as that of $F_{p_2}(S)$. Also note that the leftmost and rightmost vertices of $F_p(S)$ are dual to the two inner common tangents of $H_+(p^*)$ and $H_-(p^*)$, respectively. Hence, for any two points p_1 and p_2 of P, we can determine whether they are from the same face of $\mathcal{A}(S)$ by comparing the corresponding inner common tangents. In this way, the duplication can be removed in $O(m \log m)$ time, which is $O(m \log n)$ as $m < n^2/2$. After that, we can report all distinct faces. Note that outputting all distinct faces explicitly takes $O(n + m\sqrt{n})$ time as the total combinatorial complexity of m distinct faces in $\mathcal{A}(S)$ is bounded by $O(n + m\sqrt{n})$ [22].

To recapitulate, computing the distinct faces $F_p(S)$ implicitly for all $p \in P$ takes $O(n \log n + n \log r + mr \log n + mn/r)$ time and reporting them explicitly takes additional $O(n + m\sqrt{n})$ time. Setting $r = \min\{m, \sqrt{n/\log n}\}$ leads to the total time bounded by $O(n \log n + m\sqrt{n \log n})$.

Theorem 3.1. *Given a set S of n lines and a set P of m points in the plane, the faces of the arrangement of the lines containing at least one point of P can be computed in $O(n \log n + m\sqrt{n \log n})$ time.*

Remark. The algorithm runs in $O(n \log n)$ for $m = O(\sqrt{n \log n})$, which matches the $\Omega(n \log n)$ lower bound for computing a single face. For comparison, an algorithm of $O(n \log n + m\sqrt{n} \log^2 n)$ time is given in [1] and a randomized algorithm of $O(n \log n + m\sqrt{n} \log n)$ expected time is proposed in [2].

3.2 The second algorithm. We now present our main algorithm, which follows the scheme of Agarwal [1], but replaces a key subroutine by Theorem 3.1.

We first compute a $(1/r)$-cutting Ξ for the lines of S in $O(nr)$ time [10], with the parameter r to be determined later. We then locate the cell of Ξ containing each point of P; this can be done in $O(m \log r)$ time for all points of P [10]. Consider a cell σ of Ξ. Recall that σ is a triangle. Let $P(\sigma) = P \cap \sigma$. Let S_σ be the subset of lines of S crossing σ. Consider a point $p \in P(\sigma)$. Recall the definition in Section 2 that $F_p(S_\sigma)$ denotes the face of the arrangement $\mathcal{A}(S_\sigma)$ that contains p. Observe that the face $F_p(S)$ is $F_p(S_\sigma)$ if and only $F_p(S_\sigma)$ does not intersect the boundary of σ. The *zone* of σ in $\mathcal{A}(S_\sigma)$ is defined as the collection of face portions $F \cap \sigma$ for all faces $F \in \mathcal{A}(S_\sigma)$ that intersect the boundary of σ. If $F_p(S) \neq F_p(S_\sigma)$, then $F_p(S)$ is divided into multiple portions each of which is a face in the zone of some cell of Ξ (and $F_p(S_\sigma)$ is one of these portions). Hence, to find all nonempty faces of $\mathcal{A}(S)$, it suffices to compute, for every cell $\sigma \in \Xi$, the faces of $\mathcal{A}(S_\sigma)$ containing the points of $P(\sigma)$ and the zone of σ. The nonempty faces of $\mathcal{A}(S)$ that are split among the zones can be obtained by merging the zones along the edges of cells of Ξ.

To compute the faces of $\mathcal{A}(S_\sigma)$ containing the points of $P(\sigma)$, we apply Theorem 3.1, which takes $O(n_\sigma \log n_\sigma + m_\sigma\sqrt{n_\sigma \log n_\sigma})$ time, with $n_\sigma = |S_\sigma|$ and $m_\sigma = |P_\sigma|$. Computing the zone for σ can be done in $O(n_\sigma \log n_\sigma)$ time, e.g., by the algorithm of [3] or a recent simple algorithm [32]. After all cells of Ξ are processed as above, we merge the zones of all cells. Since $n_\sigma = O(n/r)$, $\sum_{\sigma \in \Xi} m_\sigma = m$, and Ξ has $O(r^2)$ cells, by setting $r = \max\{m^{2/3}/(n^{1/3} \cdot \log^{1/3}(n/\sqrt{m})), 1\}$, we can obtain the asserted time complexity of the entire algorithm. The detailed time analysis is omitted.

Theorem 3.2. *Given a set S of n lines and a set P of m points in the plane, the faces of the arrangement of S containing at least one point of P can be computed in $O(n^{2/3}m^{2/3} \log^{2/3} \frac{n}{\sqrt{m}} + (n + m) \log n)$ time.*

4 Computing many cells in arrangements of segments.

In this section, we consider the segment case for computing many faces. Let S be a set of n line segments and P be a set of n points in the plane. The problem is to compute all distinct non-empty faces of $\mathcal{A}(S)$. Note that these faces will be output explicitly. For ease of exposition, we make a general position assumption that no segment of S is vertical, no three segments of S are concurrent, no two segments of S share a common endpoint, and no point of P lies on a segment of S. Degenerate cases can be handled by standard techniques [21].

In the following, we first present our deterministic algorithm and then give the randomized result, which uses the deterministic algorithm as a subroutine.

4.1 The deterministic algorithm. If $m \geq n^2/2$, then the problem can be solved in $O(m \log n)$ time using the straightforward algorithm mentioned in Section 2. In what follows, we assume that $m < n^2/2$. Let E denote the set of the endpoints of all segments of S. Let L denote the set of supporting lines of all segments of S.

Initially, we sort the points of E (resp., P) by x-coordinate. We compute a $(1/r)$-cutting Ξ for L in $O(nr)$ time [10], for a sufficiently large constant r. We then locate the cell of Ξ containing each point of P; this can be done in $O(m \log r)$ time for all points of P [10]. Consider a cell σ of Ξ. Recall that σ is a triangle. Let $P(\sigma) = P \cap \sigma$ and $E(\sigma) = E \cap \sigma$. Let S_σ be the subset of segments of S intersecting σ. Note that $|S_\sigma| = O(n/r)$ and Ξ has $O(r^2)$ cells. If $|E(\sigma)| > n/r^2$, then we triangulate σ into at most $2\lceil |E(\sigma)| \cdot r^2/n \rceil$ triangles each of which contains at most n/r^2 points of E. This can be done by first sorting all points of $E(\sigma)$ and then using a sweeping algorithm as described in Section 3.1. Due to the presorting of E, the sorting of $E(\sigma)$ for all cells σ of Ξ can be done in $O(n)$ time and thus the triangulation takes $O(n)$ time in total for all cells of Ξ. By slightly abusing notation, we still use Ξ to denote the set of all new triangles for all original cells (if an original cell was not triangulated, then we also include it in the new Ξ). The new Ξ now has the following properties: each cell of Ξ is intersected by $O(n/r)$ segments of S, Ξ has $O(r^2)$ cells, and each cell of Ξ contains at most n/r^2 points of E.

For each cell $\sigma \in \Xi$, if $|P(\sigma)| > m/r^2$, then we further triangulate σ into at most $2\lceil |P(\sigma)| \cdot r^2/m \rceil$ triangles each of which contains at most m/r^2 points of P. Due to the presorting of P, the triangulation can be done in $O(m)$ time in total for all cells of Ξ, in the same way as above for $E(\sigma)$. By slightly abusing notation, we still use Ξ to denote the set of all new triangles. The new Ξ now has the following properties: each cell of Ξ is intersected by $O(n/r)$ segments of S, Ξ has $O(r^2)$ cells, each cell of Ξ contains at most n/r^2 points of E, and each cell of Ξ contains at most m/r^2 points of P.

For each cell $\sigma \in \Xi$, we define S_σ, $E(\sigma)$, and $P(\sigma)$ in the same way as before. We say that a segment of S_σ is a *short segment* of σ if it has an endpoint in the interior of σ and is a *long segment* otherwise. Let $S_1(\sigma)$ denote the set of long segments of σ and $S_2(\sigma)$ the set of short segments of σ. Since $S_1(\sigma) \subseteq S_\sigma$ and $|S_\sigma| = O(n/r)$, we have $|S_1(\sigma)| = O(n/r)$. Also note that $|S_2(\sigma)| \leq |E(\sigma)|$. As $|E(\sigma)| \leq n/r^2$, it holds that $|S_2(\sigma)| \leq n/r^2$.

For each cell edge e of Ξ, we define its *zone* as the set of faces of $\mathcal{A}(S)$ intersected by e, which can be computed in $O(n\alpha^2(n)\log n)$ time [4] (note that computing the zone in an arrangement of segments can be reduced to computing a single face and the size of the zone is $O(n\alpha(n))$ [18]). Consider a point $p \in P(\sigma)$ for any cell $\sigma \in \Xi$. Recall the definition in Section 2 that $F_p(S_\sigma)$ denotes the face of the arrangement $\mathcal{A}(S_\sigma)$ that contains p. If $F_p(S_\sigma)$ does not intersect any edge of σ, then $F_p(S)$ is $F_p(S_\sigma)$; otherwise, $F_p(S)$ is a face of the zone of an edge of σ (and that face contains p).

In light of the above discussion, our algorithm works as follows. We first compute the zones for all cell edges of Ξ and explicitly store them in a point location data structure [19, 26]. This takes $O(nr^2\alpha^2(n)\log n)$ time in total. Next, for each cell $\sigma \in \Xi$, for each point $p \in P(\sigma)$, using the point location data structure, we determine in $O(\log n)$ time whether p is in a face of the zone of any edge of σ. If yes, we explicitly output the face, which is $F_p(S)$. Otherwise, the face $F_p(S_\sigma)$ is $F_p(S)$. Let $P'(\sigma)$ denote the subset of points p of $P(\sigma)$ in the above second case (i.e., $F_p(S_\sigma)$ is $F_p(S)$). The remaining problem is to compute the faces of $\mathcal{A}(S_\sigma)$ containing at least one point of $P'(\sigma)$. To solve this subproblem, observe that the face $F_p(S_\sigma)$ is in the intersection of $F_p(S_1(\sigma))$ and $F_p(S_2(\sigma))$, which may contain multiple connected components. Hence, more precisely, $F_p(S_\sigma)$ is the connected component of $F_p(S_1(\sigma)) \cap F_p(S_2(\sigma))$ that contains p. Let $L_1(\sigma)$ be the set of the supporting lines of all segments of $S_1(\sigma)$. Because all segments of $S_1(\sigma)$ are long segments, we have the following lemma.

LEMMA 4.1. *For any point $p \in P'(\sigma)$, $F_p(S_\sigma)$ is the connected component of $F_p(L_1(\sigma)) \cap F_p(S_2(\sigma))$ that contains p.*

Proof. Recall that $F_p(S_\sigma)$ is the connected component of $F_p(S_1(\sigma)) \cap F_p(S_2(\sigma))$ that contains p. As $p \in P'(\sigma)$, we know that $F_p(S_\sigma)$ is in the interior of σ. For any segment $s \in S_1(\sigma)$, suppose that we extend s to a full line. As s is a long segment, the extension of s does not intersect the interior of σ and thus does not intersect $F_p(S_\sigma)$. This implies the lemma. $\square$

Due to the above lemma, to compute the faces of $\mathcal{A}(S_\sigma)$ containing the points of $P'(\sigma)$, we do the following: (1) compute the faces of $\mathcal{A}(L_1(\sigma))$ containing the points of $P'(\sigma)$; (2) compute the faces of $\mathcal{A}(S_2(\sigma))$ containing the points of $P'(\sigma)$; (3) compute the faces $F_p(S_\sigma)$ for all points $p \in P'(\sigma)$ by intersecting the faces obtained in the first two steps and computing the connected components containing the points of $P'(\sigma)$.

We implement the above three steps as follows. For the first step, we apply our algorithm for the line case in Theorem 3.2, because $L_1(\sigma)$ is a set of lines. For the second step, we apply our algorithm recursively on $S_2(\sigma)$ and $P'(\sigma)$, so the problem size becomes $(n/r^2, m/r^2)$ as $|S_2(\sigma)| \leq n/r^2$ and $|P'(\sigma)| \leq m/r^2$. The third step can be done by applying the blue-red merge algorithm in [20]. The detailed time analysis is omitted.

THEOREM 4.1. *Given a set S of n line segments and a set P of m points in the plane, the faces of the arrangement of the segments containing at least one point of P can be computed in $O(n^{2/3}m^{2/3}\log n + \tau(n\alpha^2(n) + n\log m + m)\log n)$ time, where $\tau = \min\{\log m, \log(n/\sqrt{m})\}$.*

Remark. The algorithm runs in $O(n\alpha^2(n)\log n)$ time for $m = O(1)$, which matches the time for computing a single face [4], and runs in $O(m \log n)$ time for $m = \Omega(n^2)$, which matches the performance of the straightforward approach.

4.2 The randomized algorithm. In this section, we present a randomized algorithm, whose running time is a function of K, the number of intersections of all segments of S.

We again assume that $m < n^2/2$ since otherwise the problem can be solved in $O(m \log n)$ by the straightforward approach. We resort to a result of de Berg and Schwarzkopf [8]. Given any $r \leq n$ and K, de Berg and Schwarzkopf [8] gave a randomized algorithm that can construct a $(1/r)$-cutting Ξ for S in $O(n \log r + Kr/n)$ expected time and the size of Ξ is $O(r + Kr^2/n^2)$. For each cell $\sigma \in \Xi$ (which is a triangle[3]), σ is intersected by $O(n/r)$ segments of S.

We set $r = n^2/(n+K)$, and thus $1 < r \leq n$ and the size of Ξ is bounded by $O(r)$. By building a point location data structure on Ξ [19, 26], we find, for each point of P, the cell of Ξ containing it. This takes $O(r + m \log r)$ time in total. For each cell $\sigma \in \Xi$, define $P(\sigma) = P \cap \sigma$. If $|P(\sigma)| > m/r$, then in the same way as in Section 3.1, we further triangulate σ into $2 \cdot \lceil |P(\sigma)| \cdot r/m \rceil$ triangles each of which contains at most m/r points of P; we now consider these triangles as cells of Ξ but σ is not a cell of Ξ anymore. As before, if we presort P in $O(n \log n)$ time, then the triangulation for all cells of Ξ can be done in $O(n)$ time in total.

The high-level algorithm scheme is similar to that in Section 3.2 for the line case. For each cell $\sigma \in \Xi$, let S_σ denote the subset of segments of S intersecting σ. Define the *zone* of σ as the collection of face portions of $F \cap \sigma$ for all faces $F \in \mathcal{A}(S_\sigma)$ that intersect the boundary of σ. As in the line case in Section 3.2, to compute all nonempty faces of $\mathcal{A}(S)$, it suffices to compute, for every cell $\sigma \in \Xi$, the faces of $\mathcal{A}(S_\sigma)$ containing the points of $P(\sigma)$ and the zone of σ. The nonempty faces of $\mathcal{A}(S)$ that are split among the zones can be obtained by merging the zones along the edges of the cells of Ξ.

Computing the zone for σ can be done in $O(n_\sigma \alpha(n_\sigma) \log n_\sigma)$ randomized time [12], where $n_\sigma = |S_\sigma|$. For the subproblem of computing the faces of $\mathcal{A}(S_\sigma)$ containing the points of $P(\sigma)$, we apply Theorem 4.1 (but replace the $O(n\alpha^2(n)\log n)$ time algorithm [4] by a slightly faster $O(n\alpha(n)\log n)$ time randomized algorithm [12], as we are satisfied with a randomized procedure). The detailed time analysis is omitted.

THEOREM 4.2. *Given a set S of n line segments and a set P of m points in the plane, the faces of the arrangement of the segments containing at least one point of P can be computed in $O(m^{2/3}K^{1/3}\log n + \tau(n\alpha(n) + n\log m + m)\log n \log K)$ expected time, where $\tau = \min\{\log m, \log(n/\sqrt{m})\}$ and K is the number of intersections of all segments of S.*

5 The face query problem.

In this section, we consider the face query problem. Let S be a set of lines in the plane. The problem is to build a data structure on S so that given a query point p, the face $F_p(S)$ of the arrangement $\mathcal{A}(S)$ that contains p can be computed efficiently. Since $F_p(S)$ is convex, our query algorithm will return the root of a binary search tree storing $F_p(S)$ so that binary-search-based queries on $F_p(S)$ can be performed in $O(\log n)$ time each (e.g., given a query point q, decide whether $q \in F_p(S)$; given a line ℓ, compute its intersection with $F_p(S)$). $F_p(S)$ can be output explicitly in $O(|F_p(S)|)$ additional time using the tree.

We work in the dual plane as in Section 3.1 and also follow the notation there. Let S^* denote the set of dual points of S. For a query point p in the primal plane, let p^* denote its dual line. Define $S^*_+(p^*)$, $S^*_-(p^*)$, $H_+(p^*)$, $H_-(p^*)$, and $F^*_p(S)$ in the same way as in Section 3.1.

Inspired by the algorithm of Lemma 3.4, we resort to the randomized optimal partition tree of Chan [9], which is originally for simplex range counting queries in d-dimensional space for any constant $d \geq 2$. We briefly review the partition tree in the planar case. Let P be a set of n points in the plane. Chan's partition tree recursively subdivides the plane into triangles (also referred to as *cells*). Each node v of T corresponds to a triangle $\triangle_v$ and a subset P_v of P such that $P_v = P \cap \triangle_v$. If v is an internal node, then v has $O(1)$ children whose triangles form a disjoint partition of $\triangle_v$. Hence, each point of P appears in P_v for only one node v in each level of T. $\triangle_v$ and the cardinality $|P_v|$ are stored at v. But P_v is not explicitly stored at v unless v is a leaf, in which case

[3]In the algorithm description [8], each cell of the cutting is a constant-sized convex polygon, but we can further triangulate it without increasing the complexity asymptotically.

$|P_v| = O(1)$.[4] The space of T is $O(n)$ and its height is $O(\log n)$. T can be built by a randomized algorithm of $O(n \log n)$ expected time. Given a query half-plane h, the range query algorithm [9] finds a set V_h of nodes of T such that $P \cap h$ is exactly the union of P_v for all nodes $v \in V_h$, the triangles $\triangle_v$ for all $v \in V_h$ are pairwise disjoint (and thus the subsets P_v for all $v \in V_h$ are also pairwise disjoint), and $|V_h| = O(\sqrt{n})$ holds with high probability. The query algorithm runs in $O(\sqrt{n})$ time with high probability.

Preprocessing. To solve our problem, in the preprocessing, we build Chan's partition tree T on the points of S^*, which takes $O(n)$ space and $O(n \log n)$ expected time. Let $S^*_v = S^* \cap \triangle_v$ for each node $v \in T$. We further enhance T as follows. For each node $v \in T$, we compute the convex hull H_v of S^*_v and store H_v at v by a binary search tree. To this end, we can presort all points of S^* by x-coordinate. Then, we sort S^*_v for all nodes $v \in T$, which can be done in $O(n \log n)$ time in total due to the presorting of S^*. Consequently, computing the convex hull H_v can be done in $O(|S^*_v|)$ time. As such, computing convex hulls for all nodes of T takes $O(n \log n)$ time in total. With these convex hulls, the space of T increases to $O(n \log n)$, because the height of T is $O(\log n)$ and the subsets S^*_v for all nodes v in the same level of T form a partition of S^*. This finishes our preprocessing, which takes $O(n \log n)$ space and $O(n \log n)$ expected time.

Queries. Consider a query point p. Without loss of generality, we assume that p^* is horizontal. Using the partition tree T, we compute the lower hull $H_+(p^*)$ as follows.

Let h be the upper half-plane bounded by the line p^*. We apply the range query algorithm [9] on h and find a set V_h of nodes of T, as discussed above. According to the properties of V_h, $S^*_+(p^*)$ is the union of S^*_v for all nodes $v \in V_h$ and the triangles $\triangle_v$ for all $v \in V_h$ are pairwise disjoint. Therefore, $H_+(p^*)$ is the lower hull of the convex hulls H_v of all $v \in V_h$. As the triangles of $\triangle_v$ for all $v \in V_h$ are pairwise disjoint, the convex hulls H_v of all $v \in V_h$ are also pairwise disjoint. Consequently, we can apply the algorithm of Lemma 3.4 to compute $H_+(p^*)$ from convex hulls H_v of all $v \in V_h$, which takes $O(|V_h| \log n)$ time because convex hulls H_v are already available due to the preprocessing. As $|V_h| = O(\sqrt{n})$ holds with high probability, the time for computing $H_+(p^*)$ is bounded by $O(\sqrt{n} \log n)$ with high probability.

Analogously, we can compute the upper hull $H_-(p^*)$. Afterwards, $F_p(S)$ can be obtained as a binary search tree in $O(\log n)$ time by computing the inner common tangents of $H_+(p^*)$ and $H_-(p^*)$, as explained in Section 3.1. The total query time is bounded by $O(\sqrt{n} \log n)$ with high probability. Further, $F_p(S)$ can be output explicitly in additional $|F_p(S)|$ time.

As discussed in Section 3.1, once the binary search tree for $F_p(S)$ is constructed, binary search trees representing convex hulls of some nodes of T may be destroyed unless fully persistent trees are used. To handle future queries, we need to restore those convex hulls. Different from the algorithm in Section 3.1, depending on applications, persistent trees may not be necessary here. For example, if $F_p(S)$ needs to be output explicitly, then after $F_p(S)$ is output, we can restore those destroyed convex hulls by "reversing" the operations that are performed during the algorithm of Lemma 3.4. The time is still bounded by $O(\sqrt{n} \log n)$ with high probability. Hence in this case persistent trees are not necessary. Also, if $F_p(S)$ only needs to be implicitly represented but $F_p(S)$ will not be needed anymore before the next query is performed, then we can also restore the convex hulls as above without using persistent trees. However, if $F_p(S)$ only needs to be implicitly represented and $F_p(S)$ still needs to be kept even after the next query is performed, then we have to use persistent trees.

We summarize our result in the following theorem.

Theorem 5.1. *Given a set S of n lines in the plane, we can preprocess it in $O(n \log n)$ randomized time and $O(n \log n)$ space so that for any query point p, we can produce a binary search tree representing the face of $\mathcal{A}(S)$ that contains p and the query time is bounded by $O(\sqrt{n} \log n)$ with high probability. Using the binary search tree, standard binary-search-based queries on the face can be performed in $O(\log n)$ time each, and outputting the face explicitly can be done in additional time linear in the number of edges of the face.*

As discussed in Section 1, using our result in Theorem 5.1, the algorithm in [17] for the face query problem in the segment case can also be improved accordingly.

5.1 Tradeoff between storage and query time. We further obtain a tradeoff between the preprocessing and the query time. To this end, we make use of Chan's r-partial partition tree [9]. Let P be a set of n point

[4]To simplify the discussion for solving our problem, if v is a leaf and $|P_v| > 1$, then we further triangulate $\triangle_v$ into $O(1)$ triangles each of which contains at most one point of P. This adds one more level to T but has the property that each leaf triangle contains at most one point of P. This change does not affect the performance of the tree asymptotically.

in the plane. For any value $r < n/\log^{\omega(1)} n$, an r-*partial partition tree* $T(r)$ for P is the same as a partition tree discussed before, except that a leaf now may contain up to r points. The number of nodes of T is $O(n/r)$. $T(r)$ can be built in $O(n\log n)$ randomized time. Given a query half-plane h, the range query algorithm [9] finds two sets V_h^1 and V_h^2 of nodes of $T(r)$ with the following property: (1) for each node $v \in V_h^1$, the triangle $\triangle_v$ is inside h; (2) for each node $v \in V_h^2$, v is a leaf and $\triangle_v$ is crossed by the bounding line of h; (3) $P \cap h$ is the union of P_v for all nodes $v \in V_h^1$ as well as the intersection $P_v \cap h$ for all nodes $v \in V_h^2$; (4) the triangles $\triangle_v$ for all $v \in V_1(h) \cup V_2(h)$ are pairwise disjoint; (5) $|V_1(h)| + |V_2(h)| = O(\sqrt{n/r})$ holds with high probability. The query algorithm finds $V_1(h)$ and $V_2(h)$ in $O(\sqrt{n/r})$ time with high probability.

Preprocessing. To solve our problem, in the preprocessing we build an r-partial partition tree $T(r)$ on the points of S^*. For each node $v \in T(r)$, we still compute and store the convex hull H_v of P_v. This still takes $O(n\log n)$ space and $O(n\log n)$ expected time as before. Next, we perform additional preprocessing for each leaf v of $T(r)$. Note that $|S_v^*| \le r$. Let S_v denote the subset of the lines of S in the primal plane dual to the points of S_v^*. We compute explicitly the arrangement $\mathcal{A}(S_v)$. For each face $F \in \mathcal{A}(S_v)$, its leftmost and rightmost vertices divide the boundary of F into an upper portion and a lower portion; for each portion, we use a binary search tree to store it. We also build a point location data structure on $\mathcal{A}(S_v)$ [19, 26]. This finishes the preprocessing for v, which takes $O(r^2)$ time and space. As $T(r)$ has $O(n/r)$ leaves, the preprocessing for all leaves takes $O(nr)$ time and space. Overall, the preprocessing takes $O(n\log n + nr)$ expected time and $O(n\log n + nr)$ space.

Queries. Consider a query point p. Again, we assume that its dual line p^* is horizontal. We compute the lower hull $H_+(p^*)$ as follows.

Let h be the upper half-plane bounded by the line p^*. We apply the range query algorithm [9] on h and find two sets V_h^1 and V_h^2 of nodes of $T(r)$, as discussed above. Due to the property (3) of V_h^1 and V_h^2 discussed above, $H_+(p^*)$ is the lower hull of the convex hulls H_v of all $v \in V_h^1$ and the convex hulls H'_v of the subset of points of S_v^* above the line p^* for all $v \in V_h^2$. For each $v \in V_h^1$, the convex hull H_v is available due to the preprocessing. For each $v \in V_h^2$, H'_v can be obtained in $O(\log n)$ time as follows. Using the point location data structure on $\mathcal{A}(S_v)$, we find the face $F_p(S_v)$ of $\mathcal{A}(S_v)$ containing p, and then H'_v is dual to the lower portion of the boundary of $F_p(S_v)$[5], whose binary search tree is computed in the preprocessing. Due to the property (4) of V_h^1 and V_h^2, all convex hulls H_v, $v \in V_h^1$, and H'_v, $v \in V_h^2$, are pairwise disjoint. Thus, we can again apply the algorithm of Lemma 3.4 to compute $H_+(p^*)$ from these convex hulls in $O((|V_h^1| + |V_h^2|)\log n)$ time. As $|V_h^1| + |V_h^2| = O(\sqrt{n/r})$ holds with high probability, the time for computing $H_+(p^*)$ is bounded by $O(\sqrt{n/r}\log n)$ with high probability.

Analogously, we can compute upper hull $H_-(p^*)$. Afterwards, $F_p(S)$ can be obtained as a binary search tree in $O(\log n)$ time by computing the inner common tangents of $H_+(p^*)$ and $H_-(p^*)$, as explained in Section 3.1. The total query time is bounded by $O(\sqrt{n/r}\log n)$ with high probability. Further, $F_p(S)$ can be output explicitly in additional $|F_p(S)|$ time.

As before, depending on applications, one can decide whether persistent trees are needed for representing convex hulls of the nodes of $T(r)$ as well as the boundary portions of the faces of the arrangements $\mathcal{A}(S_v)$ of the leaves v of $T(r)$.

We summarize our result in the following theorem.

Theorem 5.2. *Given a set S of n lines in the plane, for any value $r < n/\log^{\omega(1)} n$, we can preprocess it in $O(n\log n + nr)$ randomized time and $O(n\log n + nr)$ space so that for any query point p, we can produce a binary search tree representing the face of $\mathcal{A}(S)$ that contains p and the query time is bounded by $O(\sqrt{n/r}\log n)$ with high probability. Using the binary search tree, standard binary-search-based queries can be performed on the face in $O(\log n)$ time each, and outputting the face explicitly can be done in time linear in the number of edges of the face.*

Remark. Using the random sampling techniques [13, 24], a tradeoff between the preprocessing and the query time was also provided in [17] roughly with the following performance: the preprocessing takes $O(n^{3/2}r^{1/2}\log^{3/2} r\log^2 n)$ randomized time, the space is $O(nr\log r\log n)$, and the query time is bounded by $O(\sqrt{n/r}\log^2 n)$ with high probability (combining with the compact interval trees [23]). Hence, our result improves on all three aspects, albeit on a smaller range of r.

[5]In fact, the lower portion of the boundary of $F_p(S_v)$ may be only part of the dual of H'_v. However, since $F_p(S) \subseteq F_p(S_v)$, using the lower portion of $F_p(S_v)$ as the dual of H'_v to compute $H_+(p^*)$ and then compute $F_p(S)$ will give the correct answer.

References

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

[2] P.K. Agarwal, J. Matoušek, and O. Schwarzkopf. Computing many faces in arrangements of lines and segments. *SIAM Journal on Computing*, 27:491–505, 1998.

[3] P. Alevizos, J.-D. Boissonnat, and F.P. Preparata. An optimal algorithm for the boundary of a cell in a union of rays. *Algorithmica*, 5:573–590, 1990.

[4] N.M. Amato, M.T. Goodrich, and E.A. Ramos. Computing faces in segment and simplex arrangements. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 672–682, 1995.

[5] B. Aronov, H. Edelsbrunner, L.J. Guibas, and M. Sharir. The number of edges of many faces in a line segment arrangement. *Combinatorica*, 12:261–274, 1992.

[6] I. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the 11th Annual ACM Symposium on Computational Geometry (SoCG)*, pages 211–219, 1995.

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry — Algorithms and Applications*. Springer-Verlag, Berlin, 3rd edition, 2008.

[8] M. de Berg and O. Schwarzkopf. Cuttings and applications. *International Journal of Computational Geometry and Applications*, 5:343–355, 1995.

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

[10] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9(2):145–158, 1993.

[11] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[12] B. Chazelle, H. Edelsbrunner, L.J. Guibas, M. Sharir, and J. Snoeyink. Computing a face in an arrangement of line segments and related problems. *SIAM Journal on Computing*, 22:1286–1302, 1993.

[13] K.L. Clarkson. New applications of random sampling in computational geometry. *Discrete and Computational Geometry*, 2:195–222, 1987.

[14] K.L. Clarkson, H. Edelsbrunner, L.J. Guibas, M. Sharir, and E. Welzl. Combinatorial complexity bounds for arrangement of curves and spheres. *Discrete and Computational Geometry*, 5:99–160, 1990.

[15] K.L. Clarkson and P.W. Shor. Application of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4:387–421, 1989.

[16] 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.

[17] 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.

[18] H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir. Arrangements of curves in the plane–topology, combinatorics, and algorithms. *Theoretical Computer Science*, 92(2):319–336, 1992.

[19] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[20] H. Edelsbrunner, L.J. Guibas, and M. Sharir. The complexity and construction of many faces in arrangement of lines and of segments. *Discrete and Computational Geometry*, 5:161–196, 1990.

[21] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990.

[22] H. Edelsbrunner and E. Welzl. On the maximal number of edges of many faces in an arrangement. *Journal of Combinatorial Theory, Series A*, 41:159–166, 1986.

[23] 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.

[24] D. Haussler and E. Welzl. ϵ-nets and simplex range queries. *Discrete and Computational Geometry*, 2:127–151, 1987.

[25] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, 33:169–174, 1989.

[26] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

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

[28] J.S.B. Mitchell. On computing a single face in an arrangement of line segments. Manuscript, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, 1990.

[29] K. Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10:253–280, 1990.

[30] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[31] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[32] H. Wang. A simple algorithm for computing the zone of a line in an arrangement of lines. In *Proceedings of the 5th SIAM Symposium on Simplicity in Algorithms (SOSA)*, page to appear, 2022.