Self-improving Algorithms for Coordinate-Wise Maxima and Convex Hulls*

Kenneth L. Clarkson† Wolfgang Mulzer‡ C. Seshadhri§

January 11, 2014

Abstract

Finding the coordinate-wise maxima and the convex hull of a planar point set are probably the most classic problems in computational geometry. We consider these problems in the *self-improving setting*. Here, we have n distributions $\mathcal{D}_1, \ldots, \mathcal{D}_n$ of planar points. An input point set $(p_1, \ldots, p_n)$ is generated by taking an independent sample p_i from each $\mathcal{D}_i$, so the input is distributed according to the product $\mathcal{D} = \prod_i \mathcal{D}_i$. A *self-improving algorithm* repeatedly gets inputs from the distribution $\mathcal{D}$ (which is *a priori* unknown), and it tries to optimize its running time for $\mathcal{D}$. The algorithm uses the first few inputs to learn salient features of the distribution $\mathcal{D}$, before it becomes fine-tuned to $\mathcal{D}$. Let $\text{OPT-MAX}_\mathcal{D}$ (resp. $\text{OPT-CH}_\mathcal{D}$) be the expected depth of an *optimal* linear comparison tree computing the maxima (resp. convex hull) for $\mathcal{D}$. Our maxima algorithm eventually achieves expected running time $O(\text{OPT-MAX}_\mathcal{D} + n)$. Furthermore, we give a self-improving algorithm for convex hulls with expected running time $O(\text{OPT-CH}_\mathcal{D} + n \log\log n)$.

Our results require new tools for understanding linear comparison trees. In particular, we convert a general linear comparison tree to a restricted version that can then be related to the running time of our algorithms. Another interesting feature is an interleaved search procedure to determine the likeliest point to be extremal with minimal computation. This allows our algorithms to be competitive with the optimal algorithm for $\mathcal{D}$.

1 Introduction

The problems of planar maxima and planar convex hull computation are classic computational geometry questions that have been studied since at least 1975 [23]. There are well-known $O(n \log n)$ time comparison-based algorithms (n is the number of input points), with matching lower bounds. Since then, many more advanced settings have been addressed: one can get expected running time $O(n)$ for points uniformly distributed in the unit square; output-sensitive algorithms need $O(n \log h)$ time for output size h [21]; and there are results for external-memory models [19].

A major drawback of worst-case analysis is that it does not always reflect the behavior of real-world inputs. Worst-case algorithms must provide for extreme inputs that may not occur

*Preliminary versions appeared as K. L. Clarkson, W. Mulzer, and C. Seshadhri, *Self-improving Algorithms for Convex Hulls* in Proc. 21st SODA, pp. 1546–1565, 2010; and K. L. Clarkson, W. Mulzer and C. Seshadhri, *Self-improving Algorithms for Coordinate-wise Maxima* in Proc. 28th SoCG, pp. 277–286, 2012.

†IBM Almaden Research Center, San Jose, USA. klclarks@us.ibm.com

‡Institut für Informatik, Freie Universität Berlin, Berlin, Germany. mulzer@inf.fu-berlin.de

§Sandia National Laboratories, Livermore, USA. scomand@sandia.gov

(reasonably often) in practice. Average-case analysis tries to address this problem by assuming some fixed input distribution. For example, in the case of maxima coordinate-wise independence covers a broad range of inputs, and it leads to a clean analysis [8]. Nonetheless, it is still unrealistic, and the right distribution to analyze remains a point of investigation. However, the assumption of randomly distributed inputs is very natural and one worthy of further study.

The self-improving model. Ailon et al. introduced the self-improving model to address the drawbacks of average case analysis [3]. In this model, there is a fixed but unknown distribution $\mathcal{D}$ that generates independent inputs, i.e., whole input sets P. The algorithm initially undergoes a *learning phase* where it processes inputs with a worst-case guarantee while acquiring information about $\mathcal{D}$. After seeing a (hopefully small) number of inputs, the algorithm shifts into the *limiting phase*. Now, it is tuned for $\mathcal{D}$, and the expected running time is (ideally) *optimal for the distribution* $\mathcal{D}$. A self-improving algorithm can be thought of as able to attain the optimal average-case running time for all, or at least a large class of, distributions.

As in earlier work, we assume that the input follows a product distribution. An input $P = (p_1, \ldots, p_n)$ is a set of n points in the plane. Each p_i is generated independently from a distribution $\mathcal{D}_i$, so the probability distribution of P is the product $\prod_i \mathcal{D}_i$. The $\mathcal{D}_i$s themselves are arbitrary, we only assume that they are independent. There are lower bounds [2] showing that some restriction on $\mathcal{D}$ is necessary for a reasonable self-improving algorithm, as we shall explain below.

The first self-improving algorithm was for sorting, and it was later extended to Delaunay triangulations [2, 12]. In both cases, *entropy-optimal* performance is achieved in the limiting phase. Later, Bose et al. [6] described *odds-on trees*, a general method for self-improving solutions to certain query problems, e.g., point location, orthogonal range searching, or point-in-polytope queries.

2 Results

We give self-improving algorithms for planar coordinate-wise maxima and convex hulls over product distributions. Let $P \subseteq \mathbb{R}^2$ be finite. A point $p \in P$ *dominates* $q \in P$, if both the x- and y-coordinate of p are at least as large as the x- and y-coordinate of q. A point in P is *maximal* if no other point in P dominates it, and *non-maximal* otherwise. The *maxima problem* is to find all maximal points in P. The *convex hull* of P is the smallest convex set that contains P. It is a convex polygon whose vertices are points from P. We will focus on the *upper* hull of P, denoted by $\text{conv}(P)$. A point in P is *extremal* if it appears on $\text{conv}(P)$, otherwise it is *non-extremal*. In the *convex hull problem*, we must find the extremal points in P.

2.1 Certificates

We need to make precise the notion of an *optimal algorithm* for a distribution $\mathcal{D}$. The issue with maxima and convex hulls is their output sensitive nature. Even though the actual output size may be small, additional work is necessary to determine which points appear in the output. We also want to consider algorithms that give a correct output on *all* instances, not just those in the support of $\mathcal{D}$. For example, suppose for all inputs in the support of $\mathcal{D}$, there was a set of (say) three points that always formed the maxima. The optimal algorithm just for $\mathcal{D}$ could always output these three points. But such an algorithm is not a legitimate maxima algorithm, since it would be incorrect on other inputs.

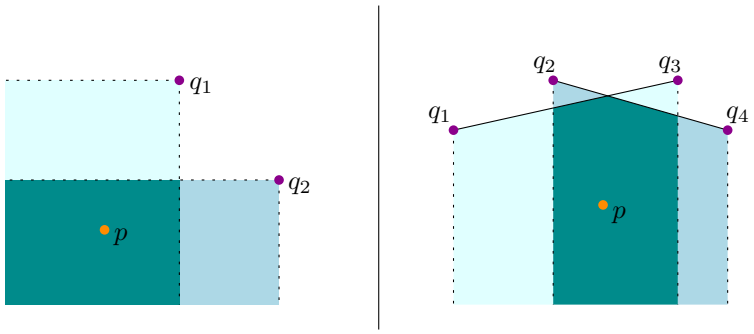

Figure 1: Certificates for maxima and convex hulls: (left) both q_1 and q_2 are certificates of non-maximality for p; (right) both q_1q_3 and q_2q_4 are possible witness pairs for non-extremality of p.

To handle these issues, we demand that any algorithm must provide a simple proof that the output is correct. This is formalized through *certificates* (see Fig. 1).

Definition 2.1. *Let $P \subseteq \mathbb{R}^2$ be finite. A* maxima certificate *γ for P consists of (i) the indices of the maximal points in P, sorted from left to right; and (ii) a* per-point certificate *for each non-maximal point $p \in P$, i.e., the index of an input point that dominates p. A certificate γ is* valid *for P if γ satisfies conditions (i) and (ii) for P.*

Most known algorithms implicitly provide a certificate as in Definition 2.1 [18, 21, 23]. For two points $p, q \in P$, we define the *upper semislab* for p and q, $\mathrm{uss}(p, q)$, as the open planar region bounded by the upward vertical rays through p and q and the line segment $\overline{pq}$. The *lower semislab* for p and q, $\mathrm{lss}(p, q)$, is defined analogously. Two points $q, r \in P$ are a *witness pair* for a non-extremal $p \in P$ if $p \in \mathrm{lss}(q, r)$.

Definition 2.2. *Let $P \subseteq \mathbb{R}^2$ be finite. A* convex hull certificate *γ for P has (i) the extremal points in P, sorted from left to right; and (ii) a witness pair for each non-extremal point in P. The points in γ are represented by their indices in P.*

To our knowledge, most current maxima and convex hull algorithms implicitly output such certificates (for example, when they prune non-extremal points). This is by no means the only possible set of certificates, and one could design different types of certificates. Our notion of optimality crucially depends on the definition of certificates. It is not *a priori* clear how to define optimality with respect to other definitions, though we feel that our certificates are quite natural.

2.2 Linear comparison trees

To define optimality, we need a lower bound model to which our algorithms can be compared. For this, we use linear algebraic computation trees that perform comparisons according to query lines defined by the input points. Let ℓ be a directed line. We write ℓ^+ for the open halfplane to the left of ℓ, and ℓ^- for the open halfplane to the right of ℓ.

Definition 2.3. *A* linear comparison tree *$\mathcal{T}$ is a rooted binary tree. Each node v of $\mathcal{T}$ is labeled with a query of the form "$p \in \ell_v^+$?". Here, p is an input point and ℓ_v a directed line. The line ℓ_v can be obtained in four ways, in increasing complexity:*

1. *a fixed line independent of the input (but dependent on v);*

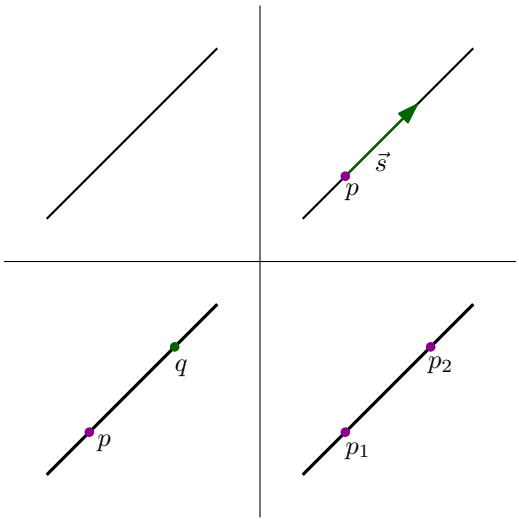

Figure 2: We can compare with (a) a fixed line; (b) a line through input point p, with fixed slope $\vec{s}$; (c) a line through input p and a fixed point q; and (d) a line through inputs p_1 and p_2.

2. *a line with a fixed slope (dependent on v) passing through a given input point;*
3. *a line through an input point and a fixed point q_v, dependent on v; or*
4. *a line through two distinct input points.*

Definition 2.3 is illustrated in Fig. 2. Given an input P, an *evaluation* of a linear comparison tree $\mathcal{T}$ on P is the node sequence that starts at the root and chooses in each step the child according to the outcome of the current comparison on P. For a node v of $\mathcal{T}$ there is a region $\mathcal{R}_v \subseteq \mathbb{R}^{2n}$ such that an evaluation of $\mathcal{T}$ on input P reaches v if and only if $P \in \mathcal{R}_v$.

Why do we choose this model? For starters, it captures the standard "counter-clockwise" (CCW) primitive. This is the is-left-of test that checks whether a point p lies to the left, on, or to the right of the directed line qr, where p, q, and r are input points [5]. The model also contains simple coordinate comparisons, the usual operation for maxima finding. Indeed, most planar maxima and convex hull algorithms only use these operations. Since we are talking about distributions of points, it also makes sense (in our opinion) to consider comparisons with fixed lines. All our definitions of optimality are dependent on this model, so it would be interesting to extend our results to more general models. We may consider comparisons with lines that have more complex dependences on the input points. Or, consider relationships with more than 3 points. Nonetheless, this model is a reasonable starting point for defining optimal maxima and convex hull algorithms.

We can now formalize linear comparison trees for maxima and convex hulls.

Definition 2.4. *A linear comparison tree $\mathcal{T}$* computes the maxima of a planar point set *if every leaf v of $\mathcal{T}$ is labeled with a maxima certificate that is valid for every input $P \in \mathcal{R}_v$. A linear comparison tree for planar convex hulls is defined analogously.*

The *depth* d_v of node v in $\mathcal{T}$ is the length of the path from the root of $\mathcal{T}$ to v. Let $v(P)$ be the leaf reached by the evaluation of $\mathcal{T}$ on input P. The *expected depth* of $\mathcal{T}$ over $\mathcal{D}$ is defined as

$$d_{\mathcal{D}}(\mathcal{T}) = \mathbf{E}_{P\sim\mathcal{D}}[d_{v(P)}].$$

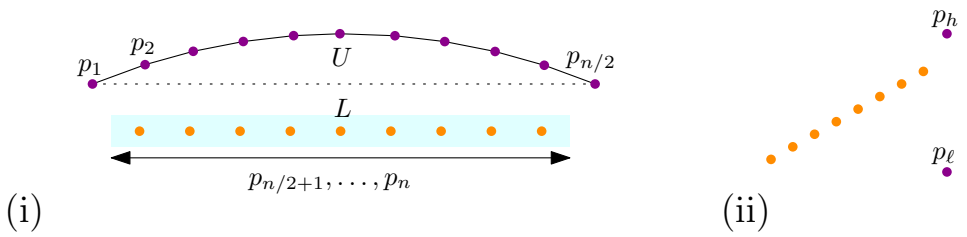

Figure 3: Bad inputs: (i) the upper hull U is fixed, while $p_{n/2+1}, \ldots, p_n$ roughly constitute a random permutation of L; (ii) point p_1 is either at p_h or p_ℓ, so it affects the extremality of the other inputs.

For a comparison based algorithm whose decision structure is modeled by $\mathcal{T}$, the expected depth of $\mathcal{T}$ gives a lower bound on the expected running time.

2.3 Main theorems

Let $\mathbf{T}$ be the set of linear comparison trees that compute the maxima of n points. We define OPT-MAX$_\mathcal{D} = \inf_{\mathcal{T}\in\mathbf{T}} d_\mathcal{D}(\mathcal{T})$. OPT-MAX is a lower bound on the expected running time of *any* linear comparison tree to compute the maxima according to $\mathcal{D}$. We prove the following result:

Theorem 2.5. *Let $\varepsilon > 0$ be a fixed constant and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ continuous planar point distributions. Set $\mathcal{D} = \prod_i \mathcal{D}_i$. There is a self-improving algorithm for coordinate-wise maxima according to $\mathcal{D}$ whose expected time in the limiting phase is $O(\varepsilon^{-1}(n + \text{OPT-MAX}_\mathcal{D}))$. The learning phase takes $O(n^\varepsilon)$ inputs. The space requirement is $O(n^{1+\varepsilon})$.*

We also give a self-improving algorithm for convex hulls. Unfortunately, it is slightly suboptimal. Like before, we set OPT-CH$_\mathcal{D} = \inf_{\mathcal{T}\in\mathbf{T}} d_\mathcal{D}(\mathcal{T})$, where now $\mathbf{T}$ is the set of linear comparison trees for the convex hull of n points. The conference version [11] claimed an optimal result, but the analysis was incorrect. Our new analysis is simpler and closer in style to the maxima result.

Theorem 2.6. *Let $\varepsilon > 0$ be a fixed constant and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ continuous planar point distributions. Set $\mathcal{D} = \prod_i \mathcal{D}_i$. There is a self-improving algorithm for convex hulls according to $\mathcal{D}$ whose expected time in the limiting phase is $O(n \log\log n + \varepsilon^{-1}(n + \text{OPT-CH}_\mathcal{D}))$. The learning phase takes $O(n^\varepsilon)$ inputs. The space requirement is $O(n^{1+\varepsilon})$.*

Any optimal (up to multiplicative factor $1/\varepsilon$ in running time) self-improving sorter requires $n^{1+\Omega(\varepsilon)}$ storage (Theorem 2 of [2]). By the standard reduction of sorting to maxima and convex hulls, this shows that the $O(n^{1+\varepsilon})$ space is necessary. Furthermore, self-improving sorters for arbitrary distributions requires exponential storage (Theorem 2 of [2]). So some restriction on the input distribution is necessary for a non-trivial result.

Prior Algorithms. Before we go into the details of our algorithms, let us explain why several previous approaches fail. We focus on convex hulls, but the arguments are equally valid for maxima. The main problem seems to be that the previous approaches rely on the sorting lower bound for optimality. However, this lower bound does not apply in our model. Refer to Fig. 3(i). The input comes in two groups: the lower group L is not on the upper hull, while all points in the upper group U are vertices of the upper hull. Both L and U have $n/2$ points. The input distribution $\mathcal{D}$ fixes the points $p_1, \ldots, p_{n/2}$ to form U, and for each p_i with $i = n/2 + 1, \ldots, n$, it picks a random

point from L (some points of L may be repeated). The "lower" points form a random permutation of $\Omega(n)$ points from L. The upper hull is always given by U, while all lower points have the same witness pair $p_1, p_{n/2}$. Thus an optimal algorithm requires $O(n)$ time.

In several other models, the example needs $\Omega(n \log n)$ time. The output size is $n/2$, so output-sensitive algorithms require $\Omega(n \log n)$ steps. Also, the *structural entropy* is $\Omega(n \log n)$ [4]. Since the expected size of the upper hull of a random r-subset of $U \cup L$ is $r/2$, randomized incremental construction takes $\Theta(n \log n)$ time [13]. As the entropy of the x-ordering is $\Omega(n \log n)$, self-improving algorithms for sorting or Delaunay triangulations are not helpful [2]. *Instance optimal algorithms* also require $\Omega(n \log n)$ steps for each input from our example [1]: this setting considers the input as a *set*, whereas for us it is essential to know the distribution of each individual input point.

Finally, we mention the paradigm of preprocessing imprecise points [7, 17, 20, 22, 24]. Given a set $\mathcal{R}$ of planar regions, we must preprocess $\mathcal{R}$ to quickly find the (Delaunay) triangulation or convex hull for inputs with exactly one point from each region in $\mathcal{R}$. If we consider inputs with a random point from each region, the self-improving setting applies, and the previous results bound the expected running time in the limiting phase. As a noteworthy side effect, we improve a result by Ezra and Mulzer [17]: they preprocess a set of planar lines so that the convex hull for inputs with one point from each line can be found in near-linear time. Unfortunately, the data structure needs quadratic space. Using self-improvement, this can now be reduced to $O(n^{1+\varepsilon})$.

Output sensitivity and dependencies. We introduced certificates in order to deal with output sensitivity. These certificates may or may not be easy to find. In Fig. 3(i), the witness pairs are all "easy". However, if the points of L are placed *just below* the edges of the upper hull, we need to search for the witness pair of each point p_i, for $i = n/2 + 1, \ldots, n$; the certificates are "hard". Furthermore, even though the individual points are independent, the upper hull can exhibit very dependent behavior. In Fig. 3(ii), point p_1 can be either p_h or p_ℓ, while the other points are fixed. The points $p_2, \ldots, p_n$ become extremal *depending* on the position of p_1. This makes life rather hard for entropy-optimality, since only if $p_1 = p_\ell$ the ordering of $p_2, \ldots, p_n$ must be determined.

Our algorithm, and plausibly any algorithm, performs a point location for each input p_i. If p_i is "easily" shown to be non-extremal, the search should stop early. However, it seems impossible to know *a priori* how far to proceed: imagine the points L of Fig. 3(i) doubled up and placed at *both* the "hard" and the "easy" positions, and p_i for $i = n/2 + 1, \ldots, n$ chosen randomly among them. The search depth can only be determined from the actual position. Moreover, the certificates may be easy once the extremal points are known, but finding them is what we wanted in the first place.

3 Preliminaries

Our input point set is called $P = \langle p_1, \ldots, p_n \rangle$, and it comes from a product distribution $\mathcal{D} = \prod_{i=1}^{n} \mathcal{D}_i$. All distributions $\mathcal{D}_i$ are assumed to be continuous. For $p \in \mathbb{R}^2$, we write $x(p)$ and $y(p)$ for the x- and the y-coordinate of p. Recall that ℓ^+ and ℓ^- denote the open halfplanes to the left and to the right of a directed line ℓ. If $R \subseteq \mathbb{R}^2$ is measurable, a *halving line* ℓ for R with respect to distribution $\mathcal{D}_i$ has the property

$$\Pr_{p \sim \mathcal{D}_i}[p \in \ell^+ \cap R] = \Pr_{p \sim \mathcal{D}_i}[p \in \ell^- \cap R].$$

If $\Pr_{p \sim \mathcal{D}_i}[p \in R] = 0$, every line is a halving line for R.

We write c for a sufficiently large constant. We say "with high probability" for any probability larger than $1 - n^{-\Omega(1)}$. The constant in the exponent can be increased by increasing the constant c. We will take union bounds over polynomially many (usually at most n^2) low probability events and still get a low probability bound.

The main self-improving algorithms require a significant amount of preperation. This is detailed in Sections 4, 5, and 6. These sections give some lemmas on the linear comparison trees, search trees, and useful data structures for the learning phase. We would recommend the reader to first skip all the proofs in these sections, as they are somewhat unrelated to the actual self-improving algorithms.

4 Linear comparison trees

We discuss basic properties of linear comparison trees. Crucially, any such tree can be simplified without significant loss in efficiency (Lemma 4.4). Let $\mathcal{T}$ be a linear comparison tree. Recall that for each node v of $\mathcal{T}$, there is an *open* region $\mathcal{R}_v \subseteq \mathbb{R}^{2n}$ such that an evaluation of $\mathcal{T}$ on P reaches v if and only if $P \in \mathcal{R}_v$ (We define the regions as open, because the continuous nature of the input distribution lets us ignore the case that a point lies on a query line.) We call $\mathcal{T}$ *restricted*, if all nodes of depth at most n^2 are of of lowest complexity, i.e., type (1) in Definition 2.3. We show that in a *restricted* linear comparison tree, each $\mathcal{R}_v$ for a node of depth at most n^2 is the Cartesian product of planar polygons. This will enable us to analyze each input point independently.

Proposition 4.1. *Let $\mathcal{T}$ be a restricted linear comparison tree, and v a node of $\mathcal{T}$ with $d_v \leq n^2$. There exists a sequence $R_1, \ldots, R_n$ of (possibly unbounded) convex planar polygons such that $\mathcal{R}_v = \prod_{i=1}^n R_i$. That is, the evaluation of $\mathcal{T}$ on $P = \langle p_1, \ldots, p_n \rangle$ reaches v if and only if $p_i \in R_i$ for all i.*

Proof. We do induction on d_v. For the root, set $R_1 = \cdots = R_n = \mathbb{R}^2$. If $d_v \geq 1$, let v' be the parent of v. By induction, there are planar convex polygons R'_i with $\mathcal{R}_{v'} = \prod_{i=1}^n R'_i$. As $\mathcal{T}$ is restricted, v' is labeled with a test "$p_j \in \ell^+_{v'}$?", the line $\ell_{v'}$ being independent of P. We take $R_i = R'_i$ for $i \neq j$, and $R_j = R'_j \cap \ell^+_{v'}$, if v is the left child of v', and $R_j = R'_j \cap \ell^-_{v'}$, otherwise. □

Next, we restrict linear comparison trees even further, so that the depth of a node v relates to the probability that v is reached by a random input $P \sim \mathcal{D}$. This allows us to compare the expected running time of our algorithms with the depth of a near optimal tree.

Definition 4.2. *A restricted comparison tree is* entropy-sensitive *if the following holds for any node v with $d_v \leq n^2$: let $\mathcal{R}_v = \prod_{i=1}^n R_i$ and v labeled "$p_j \in \ell^+_v$?". Then ℓ_v is a halving line for R_j.*

The depth of a node in an entropy-sensitive linear comparison tree is related to the probability that it is being visited:

Proposition 4.3. *Let v be a node in an entropy-sensitive tree with $d_v \leq n^2$, and $\mathcal{R}_v = \prod_{i=1}^n R_i$. Then,*

$$d_v = -\sum_{i=1}^{n} \log \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R_i].$$

Proof. We do induction on d_v. The root has depth 0 and all probabilities are 1. The claim holds. Now let $d_v \geq 1$ and v' be the parent of v. Write $\mathcal{R}_{v'} = \prod_{i=1}^n R'_i$ and $\mathcal{R}_v = \prod_{i=1}^n R_i$. By induction,

$d_{v'} = -\sum_{i=1}^{n} \log \Pr[p_i \in R'_i]$. Since $\mathcal{T}$ is entropy-sensitive, v' is labeled "$p_j \in \ell^+_{v'}$?", where $\ell_{v'}$ is a halving line for R'_j, i.e.,

$$\Pr[p_j \in R'_j \cap \ell^+_{v'}] = \Pr[p_j \in R'_j \cap \ell^-_{v'}] = \Pr[p_j \in R'_j]/2.$$

Since $R_i = R'_i$, for $i \neq j$, and $R_j = R'_j \cap \ell^+_{v'}$ or $R_j = R'_j \cap \ell^-_{v'}$, it follows that

$$-\sum_{i=1}^{n} \log \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R_i] = 1 - \sum_{i=1}^{n} \log \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R'_i] = 1 + d_{v'} = d_v.$$

□

We prove that it suffices to restrict our attention to entropy-sensitive comparison trees. The following lemma is crucial to the proof, as it gives handles on OPT-MAX and OPT-CH.

Lemma 4.4. *Let $\mathcal{T}$ be a finite linear comparison tree of worst-case depth n^2, and $\mathcal{D}$ a product distribution over points. There is an entropy-sensitive tree $\mathcal{T}'$ with expected depth $d_{\mathcal{D}}(\mathcal{T}') = O(d_{\mathcal{D}}(\mathcal{T}))$, as $d_{\mathcal{D}}(\mathcal{T}) \to \infty$.*

This is proven by converting $\mathcal{T}$ to an entropy-sensitive comparison tree whose expected depth is only a constant factor worse. This is done in two steps. The first, more technical step (Lemma 4.5), goes from linear comparison trees to restricted comparison trees. The second step goes from restricted comparison trees to entropy-sensitive trees (Lemma 4.6). Lemma 4.4 follows immediately from Lemmas 4.5 and 4.6.

Lemma 4.5. *Let $\mathcal{T}$ be a linear comparison tree of worst-case depth n^2 and $\mathcal{D}$ a product distribution. There is a restricted comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{D}}(\mathcal{T}') = O(d_{\mathcal{D}}(\mathcal{T}))$, as $d_{\mathcal{D}}(\mathcal{T}) \to \infty$.*

Lemma 4.6. *Let $\mathcal{T}$ a restricted linear comparison tree. There exists an entropy-sensitive comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{T}'} = O(d_{\mathcal{T}})$.*

For convenience, we move the proofs to a separate subsection.

4.1 Proof of Lemmas 4.5 and 4.6

The heavy lifting is done by representing a single comparison by a restricted linear comparison tree, provided that P is drawn from a product distribution. The final transformation simply replaces each node of $\mathcal{T}$ by the subtree given by the next claim. For brevity, we omit the subscript $\mathcal{D}$ from $d_{\mathcal{D}}$.

Claim 4.7. *Consider a comparison C as in Definition 2.3. Let $\mathcal{D}'$ be a product distribution for P with each p_i drawn from a polygonal region R_i. If C is not of type (1), there is a restricted linear comparison tree $\mathcal{T}'_C$ that resolves C with expected depth $O(1)$ (over $\mathcal{D}'$) and worst-case depth $O(n^2)$.*

Proof. We distinguish several cases according to Definition 2.3; see Fig. 4.
v is of type (2). We must determine whether the input point p_i lies to the left of the directed line with slope a through the input p_j. This is done through binary search. Let R_j be the region in $\mathcal{D}'$ corresponding to p_j, and ℓ_1 a halving line for R_j with slope a. We do two comparisons to determine on which side of ℓ_1 the inputs p_i and p_j lie. If they lie on different sides, we can resolve

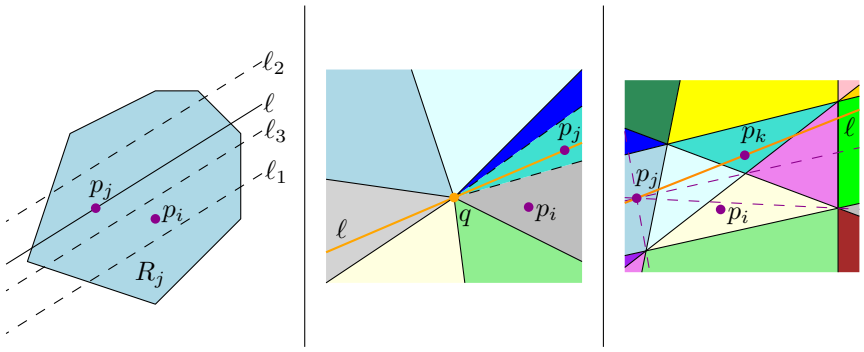

Figure 4: The different cases in the proof of Claim 4.7.

the original comparison. If not, we replace R_j with the new region and repeat. Every time, the success probability is at least $1/4$. As soon as the depth exceeds n^2, we use the original type (2) comparison. The probability of reaching a node of depth k is $2^{-\Omega(k)}$, so the expected depth is $O(1)$.

v is of type (3). We must determine whether the input point p_i lies to the left of the directed line through the input p_j and the fixed point q. We partition the plane by a constant-sized family of cones with apex q, such that for each cone V in the family, the probability that line $\overline{qp_j}$ meets V (other than at q) is at most $1/2$. Such a family can be constructed by a sweeping a line around q, or by taking a sufficiently large, but constant-sized, sample from the distribution of p_j, and bounding the cones by all lines through q and each point of the sample. As such a construction has a positive success probability, the described family of cones exists.

We build a restricted tree that locates a point in the corresponding cone, and for each cone V, we recursively build such a family of cones inside V, together with a restricted tree. Repeating for each cone, this gives an infinite restricted tree $\mathcal{T}'_C$. We search for both p_i and p_j in $\mathcal{T}'_C$. Once we locate them in two different cones of the same family, the comparison is resolved. This happens with probability at least $1/2$, so the probability that the evaluation needs k steps is $2^{-\Omega(k)}$. Again, we revert to the original comparison once the depth exceeds n^2.

v is of type (4). We must determine whether the input point p_i lies to the left of the directed line through inputs p_j and p_k. We partition the plane by a constant-sized family of triangles and cones, such that for each region V in the family, the probability that the line $\overline{p_jp_k}$ meets V is at most $1/2$. Such a family can be constructed by taking a sufficiently large random sample of pairs p_j, p_k and by triangulating the arrangement of the lines through each pair. The construction has positive success probability, so such a family exists. (Other than the source of the random lines, this scheme goes back at least to [10]; a tighter version, called *cutting*, could also be used [9].)

Now suppose p_i is in region V of the family. If the line $\overline{p_jp_k}$ does not meet V, the comparison is resolved. This occurs with probability at least $1/2$. Moreover, finding the region containing p_i takes a constant number of type (1) comparisons. Determining if $\overline{p_jp_k}$ meets V can be done with a constant number of type (3) comparisons: suppose V is a triangle. If $p_j \in V$, then $\overline{p_jp_k}$ meets V. Otherwise, suppose p_k is above all lines through p_j and each vertex of V; then $\overline{p_jp_k}$ does not meet V. Also, if p_k is below all lines through p_j and each vertex, then $\overline{p_jp_k}$ does not meet V. Otherwise, $\overline{p_jp_k}$ meets V. We replace each type (3) query by a type (1) tree, cutting off after n^2 levels.

By recursively building a tree for each region V of the family, comparisons of type (4) can be reduced to a tree of depth n^2+1 whose nodes of depth at most n^2 use comparisons of type (1) only. Since the probability of resolving the comparison $\Omega(1)$ with each family of regions that is visited,

the expected number of nodes visited is constant. □

Given Claim 4.7, we can now prove Lemma 4.5.

Proof of Lemma 4.5. We incrementally transform $\mathcal{T}$ into $\mathcal{T}'$. In each step, we have a partial restricted comparison tree $\mathcal{T}''$ that eventually becomes $\mathcal{T}'$. Furthermore, during the process each node of $\mathcal{T}$ is in one of three different states: *finished*, *fringe*, or *untouched*. We also have a function S that assigns to each finished and fringe node of $\mathcal{T}$ a subset $S(v)$ of nodes in $\mathcal{T}''$. The initial situation is as follows: all nodes of $\mathcal{T}$ are untouched except for the root, which is fringe. The partial tree $\mathcal{T}''$ has a single root node r, and the function S assigns the root of $\mathcal{T}$ to the set $\{r\}$.

The transformation proceeds as follows: we pick a fringe node v in $\mathcal{T}$, and mark it as finished. For each child v' of v, if v' is an internal node of $\mathcal{T}$, we mark it as fringe. Otherwise, we mark v' as finished. For each node $w \in S(v)$, if w has depth more than n^2, we copy the subtree of v in $\mathcal{T}$ to a subtree of w in $\mathcal{T}''$. Otherwise, we replace w by the subtree given by Claim 4.7. This is a valid application of the claim, since w is a node of $\mathcal{T}''$, a restricted tree. Hence $\mathcal{R}_w$ is a product set, and the distribution $\mathcal{D}$ restricted to $\mathcal{R}_w$ is a product distribution. Now $S(v)$ contains the roots of these subtrees. Each leaf of each such subtree corresponds to an outcome of the comparison in v. For each child v' of v, we define $S(v')$ as the set of all such leaves that correspond to the same outcome of the comparison as v'. We continue this process until there are no fringe nodes left. By construction, the resulting tree $\mathcal{T}'$ is restricted.

It remains to argue that $d_{\mathcal{T}'} = O(d_{\mathcal{T}})$. Let v be a node of $\mathcal{T}$. We define two random variables X_v and Y_v: X_v is the indicator random variable for the event that the node v is traversed for a random input $P \sim \mathcal{D}$. The variable Y_v denotes the number of nodes traversed in $\mathcal{T}'$ that correspond to v (i.e., the number of nodes needed to simulate the comparison at v, if it occurs). We have $d_{\mathcal{T}} = \sum_{v \in \mathcal{T}} \mathbf{E}[X_v]$, because if the leaf corresponding input $P \sim \mathcal{D}$ has depth d, exactly d nodes are traversed to reach it. We also have $d_{\mathcal{T}'} = \sum_{v \in \mathcal{T}} \mathbf{E}[Y_v]$, since each node in $\mathcal{T}'$ corresponds to exactly one node v in $\mathcal{T}$. Claim 4.8 below shows that $\mathbf{E}[Y_v] = O(\mathbf{E}[X_v])$, completing the proof. □

Claim 4.8. $\mathbf{E}[Y_v] \leq c\mathbf{E}[X_v]$

Proof. Note that $\mathbf{E}[X_v] = \Pr[X_v = 1] = \Pr[P \in \mathcal{R}_v]$. Since the sets $\mathcal{R}_w$, $w \in S(v)$, partition $\mathcal{R}_v$, we can write $\mathbf{E}[Y_v]$ as

$$\mathbf{E}[Y_v \mid X_v = 0] \Pr[X_v = 0] + \sum_{w \in S(v)} \mathbf{E}[Y_v \mid P \in \mathcal{R}_w] \Pr[P \in \mathcal{R}_w].$$

Since $Y_v = 0$ if $P \notin \mathcal{R}_v$, we have $\mathbf{E}[Y_v \mid X_v = 0] = 0$. Also, $\Pr[P \in \mathcal{R}_v] = \sum_{w \in S(v)} \Pr[P \in \mathcal{R}_w]$. Furthermore, by Claim 4.7, we have $\mathbf{E}[Y_v \mid P \in \mathcal{R}_w] \leq c$. The claim follows. □

Lemma 4.6 is proven using a similar construction.

Proof. (of Lemma 4.6) The original tree is restricted, so all queries are of the form $p_i \in \ell^+?$, where ℓ^+ only depends on the current node. Our aim is to only have queries with halving lines. Similar to the reduction for type (2) comparisons in Claim 4.7, we use binary search: let ℓ_1 be a halving line for R_i parallel to ℓ. We compare p_i with ℓ_1. If this resolves the original comparison, we are done. If not, we repeat with the halving line for the new region R_i' stopping after n^2 steps. In each step, the success probability is at least $1/2$, so the resulting comparison tree has constant expected depth. We apply the construction of Lemma 4.5 to argue that for a restricted tree $\mathcal{T}$ there is an entropy-sensitive version $\mathcal{T}'$ whose expected depth is higher by at most a constant factor. □

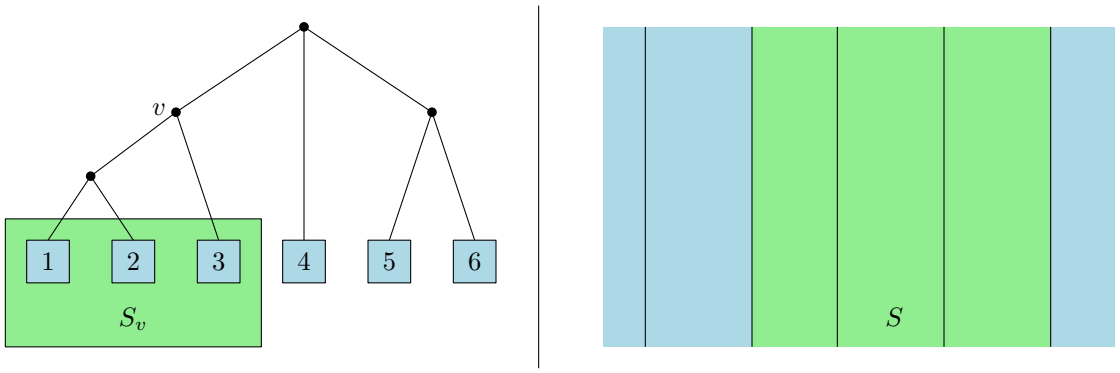

Figure 5: (left) A universe of size 6 and a search tree. The nodes are ternary, with at most two internal children. Node v represents the interval $S_v = \{1, 2, 3\}$. (right) A vertical slab structure with 6 leaf slabs (including the left and right unbounded slab). S is a slab with 3 leaf slabs, $|S| = 3$.

5 Search trees and restricted searches

We introduce the central notion of *restricted searches*. For this we use the following more abstract setting: let $\mathbf{U}$ be an ordered finite set and $\mathcal{F}$ be a distribution over $\mathbf{U}$ that assigns each element $j \in \mathbf{U}$, a probability $q(j)$. Given a sequence $\{a(j)|j \in \mathbf{U}\}$ of numbers and an interval $S \subseteq \mathbf{U}$, we write $a(S)$ for $\sum_{j \in S} a(j)$. Thus, if S is an interval of $\mathbf{U}$, then $q(S)$ is the total probability of S.

Let T be a search tree over $\mathbf{U}$. We think of T as (at most) ternary, each node having at most two internal nodes as children. Each internal node v of T is associated with an interval $S_v \subseteq \mathbf{U}$ so that every element in S_v has v on its search path; see Fig. 5. In our setting, $\mathbf{U}$ is the set of leaf slabs of a slab structure $\mathbf{S}$; see Section 6. We now define restricted searches.

Definition 5.1. *Let $S \subseteq \boldsymbol{U}$ be an interval. An* S-restricted distribution *$\mathcal{F}_S$ assigns to each $j \in \boldsymbol{U}$ the probability $\xi(j)/\sum_{r \in \boldsymbol{U}} \xi(r)$, where $\xi(j)$ fulfills $0 \leq \xi(j) \leq q(j)$, if $j \in S$; and $\xi(j) = 0$, otherwise.*

An S-restricted search *for $j \in S$ is a search for j in T that terminates as soon as it reaches the first node v with $S_v \subseteq S$.*

Definition 5.2. *Let $\mu \in (0, 1)$. A search tree T over $\boldsymbol{U}$ is* μ-reducing *if for any internal node v and for any non-leaf child w of v, we have $q(S_w) \leq \mu \cdot q(S_v)$.*

The tree T is α-optimal for restricted searches over $\mathcal{F}$ *if for every interval $S \subseteq \boldsymbol{U}$ and every S-restricted distribution $\mathcal{F}_S$, the expected time of an S-restricted search over $\mathcal{F}_S$ is at most $\alpha(1 - \log \xi(S))$. (The values $\xi(j)$ are as in Definition 5.1.)*

Our main lemma states that a search tree that is near-optimal for $\mathcal{F}$ also works for restricted distributions.

Lemma 5.3. *Let T be a μ-reducing search tree for $\mathcal{F}$. Then T is $O(1/\log(1/\mu))$-optimal for restricted searches over $\mathcal{F}$.*

5.1 Proof of Lemma 5.3

We bound the expected number of visited nodes in an S-restricted search. Let v be a node of T. In the following, we use q_v and ξ_v as a shorthand for the values $q(S_v)$ and $\xi(S_v)$. Let vis(v) be

the expected number of nodes visited below v, conditioned on v being visited. We prove below, by induction on the height of v, that for all visited nodes v with $q_v \le 1/2$,

$$\mathrm{vis}(v) \le c_1 + c \log(q_v/\xi_v), \tag{1}$$

for some constants $c, c_1 > 0$.

Given (1), the lemma follows easily: since T is μ-reducing, for v at depth k, we have $q_v \le \mu^k$. Hence, we have $q_v \le 1/2$ for all but the root and at most $1/\log(1/\mu)$ nodes below it (at each level of T there can be at most one node with $q_v > 1/2$). Let W be the set of nodes w of T such that $q_w \le 1/2$, but $q_{w'} > 1/2$, for the parent w' of w. Since T has bounded degree, $|W| = O(1/\log(1/\mu))$. The expected number $\mathrm{vis}(T)$ of nodes visited in an S-restricted search is at most

$$\begin{aligned}
\mathrm{vis}(T) &\le 1/\log(1/\mu) + \sum_{w \in W} \Pr_{\mathcal{F}_S}[j \in S_w]\, \mathrm{vis}(w) \\
&\le 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \Pr_{\mathcal{F}_S}[j \in S_w] \log(q_w/\xi_w) \\
&\le 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \Pr_{\mathcal{F}_S}[j \in S_w] \log(1/\xi_w),
\end{aligned}$$

using (1) and $q_w \le 1$. By definition of $\mathcal{F}_S$, we have $\Pr_{\mathcal{F}_S}[j \in S_w] = \xi(S_w)/\xi(S)$ $(= \xi_w/\xi(S))$, so

$$\begin{aligned}
\mathrm{vis}(T) &\le 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \frac{\xi_w}{\xi(S)} \log(1/\xi_w) \\
&= 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \frac{\xi_w}{\xi(S)} (\log(\xi(S)/\xi_w) - \log \xi(S)).
\end{aligned}$$

The sum $\sum_{w \in W} (\xi_w/\xi(S)) \log(\xi(S)/\xi_w)$ represents the entropy of a distribution over W. Hence, it is bounded by $\log |W|$. Furthermore, $\sum_{w \in W} \xi_w \le \xi(S)$, so

$$\mathrm{vis}(T) \le 1/\log(1/\mu) + c_1 + \log |W| - c \log \xi(S) = O(1 - \log \xi(S)).$$

It remains to prove (1). For this, we examine the paths in T that an S-restricted search can lead to. It will be helpful to consider the possible ways how S intersects the intervals corresponding to the nodes visited in a search. The intersection $S \cap S_v$ of S with interval S_v is *trivial* if it is either empty, S, or S_v. It is *anchored* if it shares at least one boundary line with S. If $S \cap S_v = S_v$, the search terminates at v, since we have certified that $j \in S$. If $S \cap S_v = S$, then S is contained in S_v. There can be at most one child of v that contains S. If such a child exists, the search continues to this child. If not, all possible children (to which the search can proceed to) are anchored. The search can continue to any child, at most two of which are internal nodes. If S_v is anchored, at most one child of v can be anchored with S. Any other child that intersects S must be contained in it; see Fig. 6.

Consider all nodes that can be visited by an S-restricted search (remove all nodes that are terminal, i.e., completely contained in S). They form a set of paths, inducing a subtree of S. In this subtree, there is at most one node with two children. This comes from some node r that contains S and has two anchored (non-leaf) children. Every other node of the subtree has a single child; see Fig. 6. We now prove two lemmas.

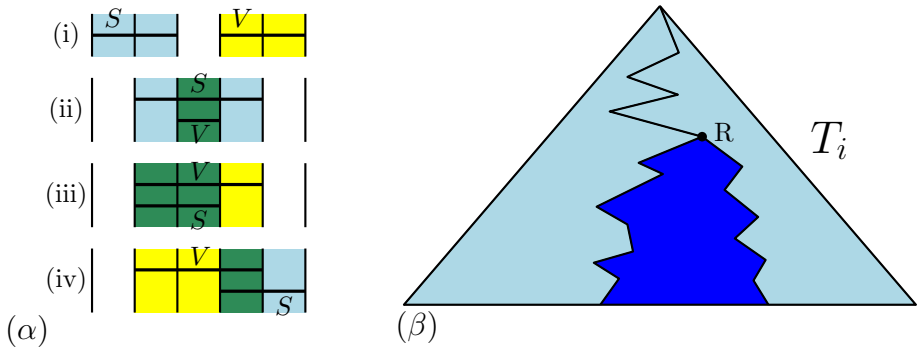

Figure 6: (α) The intersections $S \cap S_v$ in (i)-(iii) are trivial, the intersections in (iii) and (iv) are anchored; (β) every node of T_i has at most one non-trivial child, except for r.

Claim 5.4. *Let $v \neq r$ be a non-terminal node that can be visited by an S-restricted search, and let w be the unique non-terminal child of v. Suppose $q_v \leq 1/2$ and $\mathrm{vis}(w) \leq c_1 + c\log(q_w/\xi_w)$. Then, for $c \geq c_1/\log(1/\mu)$, we have*

$$\mathrm{vis}(v) \leq 1 + c\log(q_v/\xi_v). \tag{2}$$

Proof. From the fact that when a search for j shows that it is contained in a node contained in S, the S-restricted search is complete, it follows that

$$\mathrm{vis}(v) \leq 1 + \frac{\Pr_{\mathcal{F}_S}[j \in S_w]}{\Pr_{\mathcal{F}_S}[j \in S_v]}\,\mathrm{vis}(w) = 1 + \frac{\xi_w}{\xi_v}\,\mathrm{vis}(w). \tag{3}$$

Using the hypothesis, if follows that

$$\mathrm{vis}(v) \leq 1 + \frac{\xi_w}{\xi_v}(c_1 + c\log(q_w/\xi_w)).$$

Since $q_w \leq \mu q_v$, and letting $\beta := \xi_w/\xi_v \leq 1$, this is

$$\begin{aligned} &\leq 1 + \beta(c_1 + c\log(q_v/\xi_w) + c\log\mu) \\ &= 1 + \beta c_1 + \beta c\log q_v + \beta c\log(1/\xi_w) + \beta c\log\mu. \end{aligned}$$

The function $x \mapsto x\log(1/x)$ is increasing for $x \in (0, 1/2)$, so $\xi_w\log(1/\xi_w) \leq \xi_v\log(1/\xi_v)$ for $\xi_v \leq q_v \leq 1/2$. Together with $\beta = \xi_w/\xi_v \leq 1$, this implies

$$\begin{aligned} \mathrm{vis}(v) &\leq 1 + \beta c_1 + c\log q_v + c\log(1/\xi_v) + \beta c\log\mu \\ &= 1 + c\log(q_v/\xi_v) + \beta(c_1 + c\log\mu) \leq 1 + c\log(q_v/\xi_v), \end{aligned}$$

for $c \geq c_1/\log(1/\mu)$. □

Only a slightly weaker statement can be made for the node r having two nontrivial intersections at child nodes r_1 and r_2.

Claim 5.5. *Let r be as above, and let r_1, r_2 be the two non-terminal children of r. Suppose that $\mathrm{vis}(r_i) \leq c_1 + c\log(q_{r_i}/\xi_{r_i})$, for $i = 1, 2$. Then, for $c \geq c_1/\log(1/\mu)$, we have*

$$\mathrm{vis}(r) \leq 1 + c\log(q_r/\xi_r) + c.$$

Proof. Similar to (3), we get

$$\mathrm{vis}(r) \le 1 + \frac{\xi_{r_1}}{\xi_r}\,\mathrm{vis}(r_1) + \frac{\xi_{r_2}}{\xi_r}\,\mathrm{vis}(r_2).$$

Applying the hypothesis, we conclude

$$\mathrm{vis}(r) \le 1 + \sum_{i=1}^{2} \frac{\xi_{r_i}}{\xi_r}[c_1 + c\log(q_{r_i}/\xi_{r_i})].$$

Setting $\beta := (\xi_{r_1} + \xi_{r_2})/\xi_r$ and using $q_{r_i} \le \mu q_r$, we get

$$\mathrm{vis}(r) \le 1 + \beta c_1 + \beta c \log \mu + \beta c \log q_r + c\sum_{i=1}^{2}(\xi_{r_i}/\xi_r)\log(1/\xi_{r_i}).$$

The sum is maximized for $\xi_{r_1} = \xi_{r_2} = \xi_r/2$, so using once again that $\beta \le 1$, it follows that

$$\begin{aligned}\mathrm{vis}(r) &\le 1 + \beta c_1 + \beta c \log \mu + \beta c \log q_r + c\log(2/\xi_r)\\ &\le 1 + \beta(c_1 + c\log\mu) + c\log(q_r/\xi_r) + c\log 2\\ &\le 1 + c\log(q_r/\xi_r) + c,\end{aligned}$$

for $c \ge c_1/\log(1/\mu)$, as in (2), except for the addition of c. □

Now we use Claims 5.4 and 5.5 to prove (1) by induction. The bound clearly holds for leaves. For the visited nodes below r, we may inductively take $c_1 = 1$ and $c = 1/\log(1/\mu)$, by Claim 5.4. We then apply Claim 5.5 for r. For the parent v of r, we use Claim 5.4 with $c_1 = 1 + 1/\log(1/\mu)$ and $c \ge c_1/\log(1/\mu)$, getting $\mathrm{vis}(v) \le 1 + c\log(q_v/\xi_v)$. Repeated application of Claim 5.4 (with the given value of c) gives that this bound also holds for the ancestors of v, at least up until the $1 + 1/\log(1/\mu)$ top nodes. This finishes the proof of (1), and hence of Lemma 5.3.

6 Auxiliary data structures

We start with a simple heap-structure that maintains (key, index) pairs. The indices are distinct elements of $[n]$, and the keys come from the ordered universe $\{1, \ldots, U\}$ ($U \le n$). We store the pairs in a data structure with operations `insert`, `delete` (deleting a pair), `find-max` (finding the maximum key among the stored pairs), and `decrease-key` (decreasing the key of a pair). For `delete` and `decrease-key`, we assume the input is a pointer into the data structure to the appropriate pair.

Claim 6.1. *Suppose there are x* `find-max` *operations and y* `decrease-key` *operations, and that all insertions are performed at the beginning. We can implement the heap structure such that the total time for all operations is $O(n + x + y)$. The storage requirement is $O(n)$.*

Proof. We represent the heap as an array of lists. For every $k \in [U]$, we store the list of indices with key k. We also maintain m, the current maximum key. The total storage is $O(n)$. A `find-max` takes $O(1)$ time, and `insert` is done by adding the element to the appropriate list. To `delete`, we remove the element from the list (assuming appropriate pointers are available), and we update the

maximum. If the list at m is non-empty, no action is required. If it is empty, we check sequentially if the list at $m-1, m-2, \ldots$ is empty. This eventually leads to the maximum. For `decrease-key`, we `delete`, `insert`, and then update the maximum. Since all insertions happen at the start, the maximum can only decrease, and the total overhead for finding new maxima is $O(n)$. □

Our algorithms use several data structures to guide the searches. A *vertical slab structure* $\mathbf{S}$ is a sequence of vertical lines that partition the plane into open *leaf slabs*. (Since we assume continuous distributions, we may ignore the case that an input point lies on a vertical line and consider the leaf slabs to partition the plane.) More generally, a *slab* is the region between any two vertical lines of $\mathbf{S}$. The *size* of a slab S, $|S|$, is the number of leaf slabs in it. The size of $\mathbf{S}$, $|\mathbf{S}|$, is the total number of leaf slabs. For any slab S, the probability that $p_i \sim \mathcal{D}_i$ is in S is denoted by $q(i, S)$. Our algorithms construct slab structures in the learning phase, similar to the algorithm in [2].

Lemma 6.2. *We can build a slab structure $\mathbf{S}$ with $O(n)$ leaf slabs so that the following holds with probability $1 - n^{-3}$ over the construction: for a leaf slab λ of $\mathbf{S}$, let X_λ be the number of points in a random input P that lie in λ. Then $\mathbf{E}[X_\lambda^2] = O(1)$, for every leaf slab λ. The construction takes $O(\log n)$ rounds and $O(n \log^2 n)$ time.*

Proof. The construction is identical to the V-list in Ailon et al. [2, Lemma 3.2]: take $t = \log n$ random inputs $P_1, \ldots, P_t$, and let $-\infty =: x_0, x_1, \ldots, x_{nt}, x_{nt+1} := +\infty$ be the sorted list of the x-coordinates of the points (extended by $-\infty$ and ∞). The n values $x_0, x_t, x_{2t}, \ldots, x_{(n-1)t}$ define the boundaries for the slabs in $\mathbf{S}$. Lemma 3.2 in Ailon et al. [2] shows that for each leaf slab λ of $\mathbf{S}$, the number X_λ of points in a random input P that lie in λ has $\mathbf{E}_{\mathcal{D}}[X_\lambda] = O(1)$ and $\mathbf{E}_{\mathcal{D}}[X_\lambda^2] = O(1)$, with probability at least $1 - n^{-3}$ over the construction of $\mathbf{S}$. The proof is completed by noting that sorting the t inputs $P_1, \ldots, P_t$ takes $O(n \log^2 n)$ time. □

The algorithms construct a specialized search tree on $\mathbf{S}$ for each distribution $\mathcal{D}_i$. It is important to store these trees with little space. The following lemma gives the details the construction.

Lemma 6.3. *Let $\varepsilon > 0$ be fixed and $\boldsymbol{S}$ a slab structure with $O(n)$ leaf slabs. In $O(n^\varepsilon)$ rounds and $O(n^{1+\varepsilon})$ time, we can construct search trees $T_1, \ldots, T_n$ over $\boldsymbol{S}$ such that the following holds: (i) the trees can be need $O(n^{1+\varepsilon})$ total space; (ii) with probability $1 - n^{-3}$ over the construction, each T_i is $O(1/\varepsilon)$-optimal for restricted searches over $\mathcal{D}_i$.*

Once $\mathbf{S}$ is constructed, the search trees T_i can be found using essentially the same techniques in Ailon et al. [2, Section 3.2]: we use $n^\varepsilon \log n$ rounds to build the first $\varepsilon \log n$ levels of each T_i, and we use a balanced search tree for searches that proceed to a deeper level. This only costs a factor of $1/\varepsilon$. The proof of Lemma 6.3 is almost the same as that in Ailon et al. [2, Section 3.2], but since we require the additional property of restricted search optimality, we redo it for our setting.

6.1 Proof of Lemma 6.3

Let $\delta > 0$ be a sufficiently small constant and $c > 0$ be sufficiently large. We take $t = c\delta^{-2} n^\varepsilon \log n$ random inputs, and for each p_i, we record the leaf slab of $\mathbf{S}$ that contains it. We break the proof into smaller claims.

Claim 6.4. *Using t inputs, we can obtain estimates $\hat{q}(i, S)$ for each input point p_i and each slab S such that following holds (for all i and S) with probability at least $1 - n^{-3}$ over the construction: if at least $(c/10e\delta^2) \log n$ instances of p_i fell in S, then $\hat{q}(i, S) \in [(1-\delta)q(i, S), (1+\delta)q(i, S)]$.*[1]

[1] We remind the reader that this the probability that $p_i \in S$.

Proof. Fix p_i and S, and let $N(i,S)$ be the number of times p_i was in S. Let $\hat{q}(i,S) = N(i,S)/t$ be the empirical probability for this event. $N(i,S)$ is a sum of independent random variables, and $\mathbf{E}[N(i,S)] = tq(i,S)$. If $\mathbf{E}[N(i,S)] < (c/10e\delta^2)\log n$, then $2e\mathbf{E}[N(i,S)] < (c/5\delta^2)\log n$, so by a Chernoff bound [16, Theorem 1.1, Eq. (1.8)],

$$\Pr[N(i,S) > (c/5\delta^2)\log n] \leq 2^{-(c/5\delta^2)\log n} \leq n^{-6}.$$

Hence, with probability at least $1-n^{-6}$, if $N(i,S) > (c/5\delta^2)\log n$, then $\mathbf{E}[N(i,S)] \geq (c/10e\delta^2)\log n$. If $\mathbf{E}[N(i,S)] \geq (c/10e\delta^2)\log n$, multiplicative Chernoff bounds [16, Theorem 1.1, Eq. (1.7)] give

$$\Pr[N(i,S) \notin [(1-\delta)\mathbf{E}[N(i,S)], (1+\delta)\mathbf{E}[N(i,S)]]] < 2\exp(-\delta^2\mathbf{E}[N(i,S)]/3) < n^{-6}.$$

The proof is completed by taking a union bound over all i and S. □

Assume that the event of Claim 6.4 holds. If at least $(c/10e\delta^2)\log n$ inputs fell in S, then $\hat{q}(i,S) = \Omega(n^{-\varepsilon})$ and $q(i,S) = \Omega(n^{-\varepsilon})$. The tree T_i is constructed recursively. We first create a partial search tree, where some leaves may not correspond to leaf slabs. The root of T_i corresponds to $\mathbb{R}^2$. Given a slab S, we proceed as follows: if $N(S) < (c/10e\delta^2)\log n$, we make S a leaf. If not, we pick a leaf slab λ such that the subslab $S_l \subseteq S$ with all leaf slabs strictly to the left of λ and the subslab $S_r \subseteq S$ with all leaf slabs strictly to the right of λ have $\hat{q}(i,S_l) \leq (2/3)\hat{q}(i,S)$ and $\hat{q}(i,S_r) \leq (2/3)\hat{q}(i,S)$. We make λ a leaf child of S, and we recursively create trees for S_l and S_r and attach them to S. For any internal node S, we have $q(i,S) = \Omega(n^{\varepsilon})$, so the depth is $O(\varepsilon\log n)$. Furthermore, the partial tree T_i is β-reducing (for some constant β). We get a complete tree by constructing a balanced tree for each T_i-leaf that is not a leaf slab. This yields a tree of depth at most $(1+O(\varepsilon))\log n$. We only need to store the partial tree, so the total space is $O(n^{1+\varepsilon})$.

Claim 6.5. *The tree T_i is $O(1/\varepsilon)$-optimal for restricted searches.*

Proof. Fix an S-restricted distribution $\mathcal{F}_S$. For each leaf slab λ, let $q'(i,\lambda)$ be the probability according to $\mathcal{F}_S$. Note that $q'(i,S) \leq q(i,S)$. If $q'(i,S) \leq n^{-\varepsilon/2}$, then $-\log q'(i,S) \geq \varepsilon(\log n)/2$. Any search in T_i takes at most $(1+O(\varepsilon))\log n$ steps, so the search time is $O(\varepsilon^{-1}(-\log q'(i,S)+1))$.

Now suppose $q'(i,S) > n^{-\varepsilon/2}$. Consider a search for p_i. We classify the search according to the leaf that it reaches in the partial tree. By construction, any leaf S' of T_i is either a leaf slab or has $q(i,S') = O(n^{-\varepsilon})$. The search is of *Type 1* if the leaf of the partial tree represents a leaf slab (and hence the search terminates). The search is of *Type 2* (resp. *Type 3*) if the leaf of the partial tree is an internal node of T_i and the depth is at least (resp. less than) $\varepsilon(\log n)/3$.

As a thought experiment, we construct a related tree T_i': start with the partial T_i, and for every leaf that is not a leaf slab, extend it using the true probabilities $q(i,S)$. That is, construct the subtree rooted at a new node S in the following manner: pick a leaf slab λ with $q(i,S_l) \leq (2/3)q(i,S)$ and $q(i,S_r) \leq (2/3)q(i,S)$ (with S_l and S_r as above). This ensures that T_i' is β-reducing. By Lemma 5.3, T_i' is $O(1)$-optimal for restricted searches over $\mathcal{F}_i$ (we absorb β into the $O(1)$).

If the search is of Type 1, it is identical in both T_i and T_i'. If it is of Type 2, it takes at least $\varepsilon(\log n)/3$ steps in T_i' and at most $(1+O(\varepsilon))(\log n)$ steps in T_i. Consider Type 3 searches. The total number of leaves (that are not leaf slabs) of the partial tree at depth less than $\varepsilon(\log n)/3$ is at most $n^{\varepsilon/3}$. The total probability mass of $\mathcal{F}_i$ on such leaves is $O(n^{\varepsilon/3}\cdot n^{-\varepsilon}) < O(n^{-2\varepsilon/3})$. Since $q'(i,S) > n^{-\varepsilon/2}$, the probability of a Type 3 search is at most $O(n^{-\varepsilon/6})$.

Choose a random $p_i \sim \mathcal{F}_S$. Let $\mathcal{E}$ be the event that a Type 3 search occurs. Furthermore, let Z be the depth of the search in T_i and Z' be the depth in T_i'. If $\mathcal{E}$ does not occur, we have argued that $Z = O(Z'/\varepsilon)$. Also, $\Pr(\mathcal{E}) = O(n^{-\varepsilon/6})$. The expected search time is $\mathbf{E}[Z]$. Hence,

$$\begin{aligned}\mathbf{E}[Z] = \Pr[\overline{\mathcal{E}}]\mathbf{E}_{\overline{\mathcal{E}}}[Z] + \Pr[\mathcal{E}]\mathbf{E}_{\mathcal{E}}[Z] &\leq O(\varepsilon^{-1}\mathbf{E}_{\overline{\mathcal{E}}}[Z']) + n^{-\varepsilon/6}(1 + O(\varepsilon))\log n \\ &= O(\varepsilon^{-1}\mathbf{E}_{\overline{\mathcal{E}}}[Z'] + 1).\end{aligned}$$

Since $\Pr[\overline{\mathcal{E}}] > 1/2$, $\mathbf{E}_{\overline{\mathcal{E}}}[Z'] \leq 2\Pr[\overline{\mathcal{E}}]\mathbf{E}_{\overline{\mathcal{E}}}[Z'] \leq 2\mathbf{E}[Z']$. Combining everything, the expected search time is $O(\varepsilon^{-1}\mathbf{E}[Z'] + 1)$. Since T_i' is $O(1)$-optimal for restricted searches, T_i is $O(\varepsilon^{-1})$-optimal. □

7 A self-improving algorithm for coordinate-wise maxima

We begin with an informal overview. If P is sorted by x-coordinate, we can do a right-to-left sweep: we maintain the maximum y-coordinate Y seen so far. When a point p is visited, if $y(p) < Y$, then p is non-maximal, and the point q with $Y = y(q)$ gives a per-point certificate for p. If $y(p) \geq Y$, then p is maximal. We update Y and put p at the beginning of the maxima list of P. This suggests the following approach to a self-improving algorithm: sort P with a self-improving sorter and then do the sweep. The sorting algorithm of [2] works by locating each point of P within the slab structure $\mathbf{S}$ of Lemma 6.2 using the trees T_i of Lemma 6.3.

As discussed in Section 2, this does not work. We need another approach: as a thought experiment, suppose that the maximal points of P are available, though not in sorted order. We locate the maxima in $\mathbf{S}$ and determine their sorted order. We can argue that the optimal algorithm must also (in essence) perform such a search. To find per-point certificates for the non-maximal points, we use the slab structure $\mathbf{S}$ and the search trees, proceeding very conservatively. Consider the search for a point p. At any intermediate stage, p is placed in a slab S. This rough knowledge of p's location may be enough to certify its non-maximality: let m denote the leftmost maximal point to the right of S (since the sorted list of maxima is known, this information can be easily deduced). We check if m dominates p. If so, we have a per-point certificate, and we terminate the search. Otherwise, we continue the search by a single step in the search tree for p, and we repeat.

Non-maximal points that are dominated by many maximal points should have a short search, while points that are "nearly" maximal should need more time. Thus, this approach should derive just the "right" amount of information to determine the maxima. Unfortunately, our thought experiment requires that the maxima be known. This, of course, is too much to ask, and due to the strong dependencies, it is not clear how to determine the maxima before performing the searches.

The final algorithm overcomes this difficulty by interleaving the searches for sorting the points with confirmation of the maximality of some points, in a rough right-to-left order that is a more elaborate version of the traversal scheme given above. The searches for all points p_i (in their respective trees T_i) are performed "together", and their order is carefully chosen. At any intermediate stage, each point p_i is located in some slab S_i, represented by a node of its search tree. We choose a specific point and advance its search by one step. This choice is very important, and is the basis of optimality. The algorithm is described in detail and analyzed in Section 7.2.

7.1 Restricted Maxima Certificates

We modify the maxima certificate from Definition 2.1 in order to get easier proofs of optimality. For this, we need the following observation, see Fig. 7.

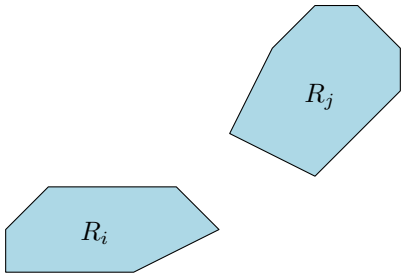

Figure 7: Every point in R_j dominates every point in R_i.

Proposition 7.1. *Let $\mathcal{T}$ be a linear comparison tree for computing the maxima. Let v be a leaf of $\mathcal{T}$ and R_i be the region associated with non-maximal point $p_i \in P$ in $\mathcal{R}_v$. There is a region R_j associated with a maximal point p_j such that every point in R_j dominates every point in R_i.*

Proof. The leaf v is associated with a certificate γ that is valid for every input that reaches v. The certificate γ associates the non-maximal point p_i with p_j such that p_j dominates p_i. For any input P in $\mathcal{R}_v$, p_j dominates p_i. First, we argue that p_j can be assumed to be maximal. We construct a directed graph G with vertex set $[n]$ such that G has an edge (u, v) if and only if (according to γ) p_u is dominated by p_v. All vertices have outdegree at most 1, and there are no cycles in G (since dominance is transitive). Hence, G consists of trees with edges directed towards the root. The roots are maximal vertices, and any point in a tree is dominated by the point corresponding to the root. We can thus rewrite γ so that all dominating points are extremal.

Since $\mathcal{T}$ is restricted, the region $\mathcal{R}_v \subseteq \mathbb{R}^{2n}$ for v is a Cartesian product of polygonal regions $R_1, \ldots, R_n$. Suppose there are two points $p_i \subseteq R_i$ and $p_j \subseteq R_j$ such that p_j does not dominate p_i. Take an input P where the remaining points are arbitrarily chosen from their respective regions. The certificate γ is not valid for P, contradicting the nature of $\mathcal{T}$. Hence, every point in R_j dominates every point in R_i. □

We need points in the maxima certificate to be "well-separated" according to the slab structure $\mathbf{S}$. By Proposition 7.1, every non-maximal point is associated with a dominating region.

Definition 7.2. *Let $\boldsymbol{S}$ be a slab structure. A maxima certificate for an input P is* $\mathbf{S}$-labeled *if (i) every maximal point is labeled with the leaf slab of $\boldsymbol{S}$ containing it; and (ii) every non-maximal point is either placed in the containing leaf slab, or is separated from its dominating region by a slab boundary.*

A tree $\mathcal{T}$ *computes the $\boldsymbol{S}$-labeled maxima* if the leaves are associated with $\mathbf{S}$-labeled certificates.

Lemma 7.3. *There is an entropy-sensitive comparison tree $\mathcal{T}$ for computing the $\boldsymbol{S}$-labeled maxima whose expected depth over $\mathcal{D}$ is $O(n + \text{OPT-MAX}_{\mathcal{D}})$.*

Proof. We start with a linear comparison tree of depth $O(\text{OPT-MAX}_D)$ that computes the maxima, with certificates as in Proposition 7.1. Each leaf has a list M with the maximal points in sorted order. We merge M with the slab boundaries of $\mathbf{S}$ to label each maximal point with the leaf slab of $\mathbf{S}$ containing it. This needs $O(n)$ additional comparisons. Now let R_i be the region associated with a non-maximal point p_i, and R_j the maximal dominating region. Let λ be the leaf slab containing R_j. The x-projection of R_i cannot extend to the right of λ. If there is a slab boundary separating R_i

from R_j, nothing needs to be done. Otherwise, R_i intersects λ. With one more comparison, we can place p_i inside λ or strictly to the left of it. In total, it takes $O(n)$ additional comparisons in each leaf to that get a tree for the **S**-labeled maxima. Hence, the expected depth is $O(n + \text{OPT-MAX}_{\mathcal{D}})$. We apply Lemma 4.4 to get an entropy-sensitive tree with the desired properties. □

7.2 The algorithm

In the learning phase, the algorithm constructs a slab structure **S** and search trees T_i as in Lemmas 6.2 and 6.3. Henceforth, we assume that we have these structures, and we describe the algorithm in the limiting phase. The algorithm searches each point p_i progressively in its tree T_i, while interleaving the searches carefully.

At any stage of the algorithm, each point p_i is placed in some slab S_i. The algorithm maintains a set A of *active points*. All other points are either proven non-maximal, or placed in a leaf slab. The heap structure $L(A)$ from Claim 6.1 is used to store pairs of indices of active points and associated keys. Recall that $L(A)$ supports the operations `insert`, `delete`, `decrease-key`, and `find-max`. The key for an active point p_i is the right boundary of the slab S_i (represented as an element of $[|\mathbf{S}|]$). We list the variables of the algorithm. Initially, $A = P$, and each S_i is the largest slab in **S**. Hence, all points have key $|\mathbf{S}|$, and we `insert` all these pairs into $L(A)$.

1. $A, L(A)$: the list A of active points is stored in heap structure $L(A)$, with their associated right slab boundary as key.
2. $\widehat{\lambda}, B$: Let m be the largest key in $L(A)$. Then $\widehat{\lambda}$ is the leaf slab with right boundary is m and B is a set of points located in $\widehat{\lambda}$ so far. Initially B is empty and m is $|S|$, corresponding to the $+\infty$ boundary of the rightmost, infinite, slab.
3. $M, \hat{p}$: M is a sorted (partial) list of the maximal points so far, and $\hat{p}$ is the leftmost among those. Initially M is empty and $\hat{p}$ is a "null" point that dominates no input point.

The algorithm involves a main procedure **Search**, and an auxiliary procedure **Update**. The procedure **Search** chooses a point and advances its search by a single node in the corresponding search tree. Occasionally, **Search** invokes **Update** to change the global variables. The algorithm repeatedly calls **Search** until $L(A)$ is empty. After that, we make a final call to **Update** in order to process any remaining points.

Search: Perform a `find-max` in $L(A)$ and let p_i be the resulting point. If the maximum key m in $L(A)$ is less than the right boundary of $\widehat{\lambda}$, invoke **Update**. If p_i is dominated by $\hat{p}$, delete p_i from $L(A)$. If not, advance the search of p_i in T_i by a single node, if possible. This updates the slab S_i. If the right boundary of S_i has decreased, perform a `decrease-key` operation on $L(A)$. (Otherwise, do nothing.) Suppose the point p_i reaches a leaf slab λ. If $\lambda = \widehat{\lambda}$, remove p_i from $L(A)$ and insert it in B (in time $O(|B|)$). Otherwise, leave p_i in $L(A)$.

Update: Sort the points in B and update the list of maxima. As Claim 7.4 will show, we know the sorted list of maxima to the right of $\widehat{\lambda}$. Hence, we can append to this list in $O(|B|)$ time. We reset $B = \emptyset$, set $\widehat{\lambda}$ to the leaf slab to the left of m, and return.

The following claim states the main important invariant of the algorithm.

Claim 7.4. *At any time in the algorithm, all maxima to the right of $\widehat{\lambda}$ have been found, in order from right to left.*

Proof. The proof is by backward induction on m, the right boundary of $\widehat{\lambda}$. For $m = |S|$, the claim is trivially true. Assume it holds for a given value of m, and trace the algorithm's behavior until the maximum key becomes smaller than m (which happens in **Search**). When **Search** processes

a point p with key m then either (i) the key value decreases; (ii) p is dominated by $\hat{p}$; or (iii) p is placed in $\widehat{\lambda}$ (whose right boundary is m). In all cases, when the maximum key decreases below m, all points in $\widehat{\lambda}$ are either proven to be non-maximal or are in B. By the induction hypothesis, we already have a sorted list of maxima to the right of m. The procedure **Update** sorts the points in B and all maximal points to the right of $m-1$ are determined. □

7.2.1 Running time analysis

We prove the following lemma.

Lemma 7.5. *The maxima algorithm runs in $O(n + \text{OPT-MAX}_{\mathcal{D}})$ time.*

We can easily bound the running time of all calls to **Update**.

Claim 7.6. *The total expected time for calls to **Update** is $O(n)$.*

Proof. The total time for the calls to **Update** is at most the time needed for sorting points within each leaf slabs. By Lemma 6.2, this takes expected time

$$\mathbf{E}\Big[\sum_{\lambda\in\mathbf{S}} X_\lambda^2\Big] = \sum_{\lambda\in\mathbf{S}} \mathbf{E}\big[X_\lambda^2\big] = \sum_{\lambda\in\mathbf{S}} O(1) = O(n).$$

□

The following claim is key to relating the time spent by **Search** to entropy-sensitive comparison trees.

Claim 7.7. *Let $\mathcal{T}$ be an entropy-sensitive comparison tree computing **S**-labeled maxima. Consider a leaf v with depth $d_v \le n^2$ labeled with the regions $\mathcal{R}_v = R_1 \times \cdots \times R_n$. Conditioned on $P \in \mathcal{R}_v$, the expected running time of* **Search** *is $O(n + d_v)$.*

Proof. For each R_i, let S_i be the smallest slab of $\mathbf{S}$ that completely contains R_i. We will show that the algorithm performs at most an S_i-restricted search for input $P \in \mathcal{R}_v$. If p_i is maximal, then R_i is contained in a leaf slab (because the output is **S**-labeled). Hence S_i is a leaf slab and an S_i-restricted search for a maximal p_i is just a complete search.

Now consider a non-maximal p_i. By the properties of **S**-labeled maxima, the associated region R_i is either inside a leaf slab or is separated by a slab boundary from the dominating region R_j. In the former case, an S_i-restricted search is a complete search. In the latter case, an S_i-restricted search suffices to process p_i: by Claim 7.4, when an S_i-restricted search finishes, all maxima to the right of S_i have been determined. In particular, we have found p_j, so $\hat{p}$ dominates p_i. Hence, the search for p_i proceeds no further.

The expected search time taken conditioned on $P \in \mathcal{R}_v$ is the sum (over i) of the conditional expected S_i-restricted search times. Let $\mathcal{E}_i$ denote the event that $p_i \in R_i$, and $\mathcal{E}$ be the event that $P \in \mathcal{R}_v$. We have $\mathcal{E} = \bigwedge_i \mathcal{E}_i$. By the independence of the distributions and linearity of expectation

$$\begin{aligned}\mathbf{E}_{\mathcal{E}}[\text{search time}] &= \sum_{i=1}^{n} \mathbf{E}_{\mathcal{E}}[S_i\text{-restricted search time for } p_i] \\ &= \sum_{i=1}^{n} \mathbf{E}_{\mathcal{E}_i}[S_i\text{-restricted search time for } p_i].\end{aligned}$$

By Lemma 5.3, the time for an S_i-restricted search conditioned on $p_i \in R_i$ is $O(-\log \Pr[p_i \in R_i]+1)$. By Proposition 4.3, $d_v = \sum_i -\log \Pr[p_i \in R_i]$, completing the proof. □

We can now prove the main lemma.

Proof of Lemma 7.5. By Lemma 7.3, there is an entropy-sensitive tree that computes the **S**-labeled maxima with expected depth $O(\text{OPT-MAX} + n)$. Since the algorithm never exceeds $O(n^2)$ steps and by Claim 7.7, the expected running time of **Search** is $O(\text{OPT-MAX} + n)$, and by Claim 7.6 the total expected time for **Update** is $O(n)$. Adding these bounds completes the proof. □

8 A self-improving algorithm for convex hulls

We outline the main ideas. The basic approach is the same as for maxima. We set up a slab structure **S**, and each distribution has a dedicated tree for searching points. At any stage, each point is at some intermediate node of the search tree, and we wish to advance searches for points that have the greatest potential for being extremal. Furthermore, we would like to quickly ascertain that a point is not extremal, so that we can terminate its search.

For maxima, this strategy is easy enough to implement. The "rightmost" active point is a good candidate for being maximal, so we always proceed its search. The leftmost known maximal point can be used to obtain certificates of non-maximality. For convex hulls, this is much more problematic. At any stage, there are many points likely to be extremal, and it is not clear how to choose. We also need a procedure that can quickly identify non-extremal points.

We give a high-level description of the main algorithm. We construct a *canonical hull* $\mathcal{C}$ in the learning phase. The canonical hull is a crude representative for the actual upper hull. The canonical hull has two key properties. First, any point that is below $\mathcal{C}$ is likely to be non-extremal. Second, there are not too many points above $\mathcal{C}$.

The curve $\mathcal{C}$ is constructed as follows. For every (upward) direction v, take the normal line ℓ_v such that the expected total number of points above ℓ_v is $\log n$. We can take the intersection of ℓ_v^- over all v, to get an upper convex curve $\mathcal{C}$. Any point below this curve is highly likely to be non-extremal. Of course, we need a finite description, so we choose some finite set $\mathbf{V}$ of directions, and only consider ℓ_v^- for these directions to construct $\mathcal{C}$. We choose $\mathbf{V}$ to ensure that the expected number of extremal points in the slab corresponding to a segment of $\mathcal{C}$ is $O(\log n)$. We build the slab structure **S** based on these segments of $\mathcal{C}$, and search for points in **S**. Each search for point p will result in one of the three conclusions: p is located above $\mathcal{C}$, p is located below $\mathcal{C}$, or p is located in a leaf slab. This procedure is referred to as the *location algorithm*.

Now, we have some partial information about the various points that is used by a *construction algorithm* to find $\text{conv}(P)$. We can ignore all points below $\mathcal{C}$, and prove that the $\text{conv}(P)$ can be found on $O(n \log\log n)$ time.

8.1 The canonical directions

We describes the structures obtained in the learning phase. In order to characterize the typical behavior of a random input $P \sim \mathcal{D}$, we use a set $\mathbf{V}$ of *canonical directions*. A *direction* is a two-dimensional unit vector. Directions are ordered clockwise, and we only consider directions that point upwards. Given a direction v, we say that $p \in P$ is *extremal* for v if the scalar product $\langle p, v\rangle$ is maximum in P. We denote the lexicographically smallest input point that is extremal for v by

e_v. The canonical directions are described in the following lemma, whose proof we postpone to Section 9.1. They are computed in the learning phase. (Refer to Definition 2.2 and just above it for some of the basic notation below.)

Lemma 8.1. *Let $k := n/\log^2 n$. There is an $O(n \operatorname{poly}\log n)$ time procedure that takes $\operatorname{poly}(\log n)$ random inputs and outputs an ordered sequence $\mathbf{V} = v_1, \dots, v_k$ of directions with the following properties (with probability at least $1 - n^{-4}$ over construction). Let $P \sim \mathcal{D}$. For $i = 1, \dots, k$, let $e_i = e_{v_i} \in P$, let X_i be the number of points from P inside $\operatorname{uss}(e_i, e_{i+1})$, and Y_i the number of* extremal *points inside $\operatorname{uss}(e_i, e_{i+1})$. Then*

$$\mathbf{E}_{P\sim\mathcal{D}}\Big[\sum_{i=1}^{k} X_i \log(Y_i + 1)\Big] = O(n \log\log n).$$

We construct some special lines that are normal to the canonical directions. The details are in Section 9.2.

Lemma 8.2. *We can construct (in $O(n \operatorname{poly}\log n)$ time with one random input) lines $\ell_1, \dots, \ell_k$ with ℓ_i normal to v_i, and with the following property (with probability at least $1 - n^{-4}$ over the construction). For $i = 1, \dots, k$ (and c large enough), we have*

$$\Pr_{P\sim\mathcal{D}}[|\ell_i^+ \cap P| \in [1, c\log n]] \geq 1 - n^{-3}.$$

We henceforth assume that the learning phase succeeds, so the directions and lines have properties from Lemma 8.1 and 8.2. We call $p \in P$ is $\mathbf{V}$-*extremal* if $p = e_v$ for some $v \in \mathbf{V}$. Using the canonical directions from Lemma 8.1 and the lines from Lemma 8.2, we construct a *canonical hull* $\mathcal{C}$ that is "typical" for random P. It is the intersection of the halfplanes below the ℓ_i, i.e., $\mathcal{C} = \bigcap_{i=1}^k \ell_i^-$. Thus, $\mathcal{C}$ is a convex polygonal region bounded by the ℓ_i. The following corollary follows from a union bound of Lemma 8.2 over all i. It implies that the total number of points outside $\mathcal{C}$ is $O(n/\log n)$.

Corollary 8.3. *Assume the learning phase succeeds. With probability at least $1 - n^{-2}$, the following holds: for all i, the extremal point for v_i lies outside $\mathcal{C}$. The number of pairs (p, s), where $p \in P \setminus \mathcal{C}$, s is an edge of $\mathcal{C}$, and s is visible from p, is $O(n/\log n)$.*

To give some intuition about $\mathbf{V}$, consider the simple example where each distribution outputs a fixed point. We set v_1 to be the direction pointing leftwards, so the extremal point e_1 is the leftmost point. Starting from e_1, continue to the first extremal point e_2 such that there are $O(\log n)$ extremal points between e_1 and e_2. Take any direction v_2 such that e_2 is extremal for it. Continue in this manner to get $\mathbf{V}$. For each v_i, the line ℓ_i is normal to v_i and has $\Theta(\log n)$ points above it. So $\mathcal{C} = \bigcap_{i=1}^k \ell_i^-$ is "well under" $\operatorname{conv}(P)$, but not too far below.

We list some preliminary concepts related to $\mathcal{C}$, see Fig. 8. By drawing a vertical line through each vertex of $\mathcal{C}$, we obtain a subdivision of the plane into vertical open slabs, the $\mathcal{C}$-*leaf-slabs*. A contiguous interval of $\mathcal{C}$-leaf slabs is again a vertical slab, called $\mathcal{C}$-*slab*. The $\mathcal{C}$-leaf-slabs define the slab structure for the upper hull algorithm, and we use Lemma 6.3 to construct appropriate search trees $T_1, \dots, T_n$ for the $\mathcal{C}$-leaf slabs and for each distribution $\mathcal{D}_i$.

For a $\mathcal{C}$-slab C, we let $\operatorname{seg}(\mathcal{C}, C)$ be the line segment between the two vertices of $\mathcal{C}$ that lie on the vertical boundaries of C. Let p be a point outside of $\mathcal{C}$, and let a_1 and a_2 be the vertices of $\mathcal{C}$

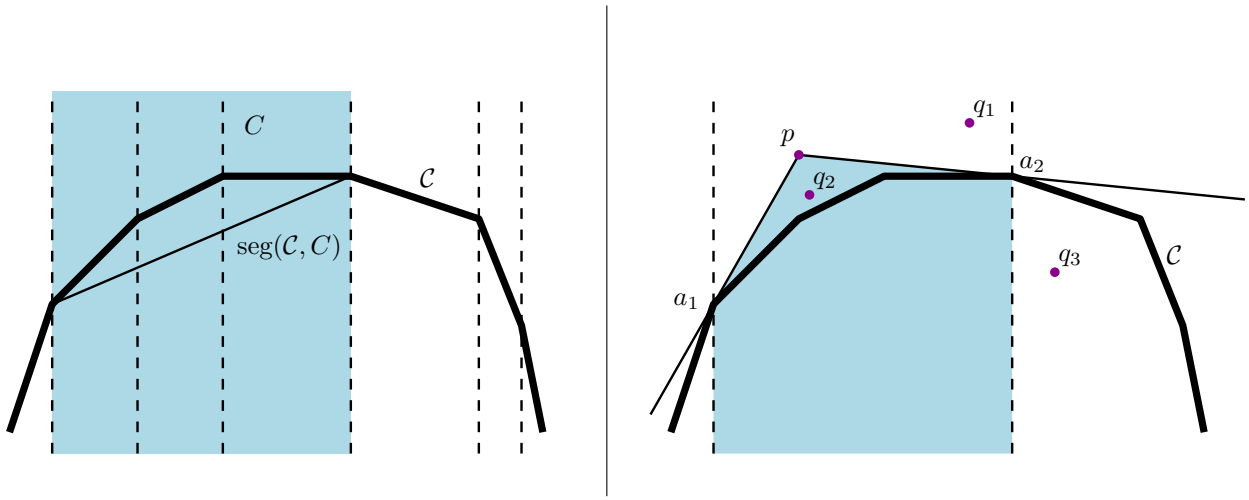

Figure 8: (left) The $\mathcal{C}$-leaf slabs are shown dashed. The shaded portion represents a $\mathcal{C}$-slab C. (right) pen(p) is shown shaded: q_1 lies above the pencil: q_2 inside it; q_3 is not comparable to it.

where the two tangents for $\mathcal{C}$ through p touch $\mathcal{C}$. The *pencil slab* for p is the $\mathcal{C}$-slab bounded by the vertical lines through a_1 and a_2. The *pencil of* p, pen(p) is the region inside the pencil slab for p that lies below the line segments $\overline{a_1 p}$ and $\overline{p a_2}$. A point q is *comparable* to pen(p) if it lies inside the pencil slab for p. It lies *above* pen(p) if it is comparable to pen(p) but not inside it.

8.2 Restricted Convex Hull Certificates

We need to refine the certificates from Definition 2.2. Recall that a upper hull certificate has a sorted list of extremal points in P, and a witness pair for each non-extremal point in P. The points (q, r) form a witness pair for p if $p \in \text{lss}(q, r)$. A witness pair (q, r) is *extremal* if both q and r are extremal; it is $\mathbf{V}$-*extremal* if both q and r are $\mathbf{V}$-extremal. Two distinct extremal points q and r are called *adjacent* if there is no extremal point with x-coordinate strictly between the x-coordinates of q and r. Adjacent $\mathbf{V}$-extremal points are defined analogously.

We now define a $\mathcal{C}$-*certificate* for P. It consists of (i) a list of the $\mathbf{V}$-extremal points of P, sorted from left to right; and (ii) a list that has a $\mathcal{C}$-slab S_p for every other point $p \in P$. The $\mathcal{C}$-slab S_p contains p and can be of three different kinds; see Fig. 9. Either

1. S_p is a $\mathcal{C}$-leaf slab; or
2. p lies below seg($\mathcal{C}, S_p$); or
3. S_p is the pencil slab for a $\mathbf{V}$-extremal vertex e_v such that p lies in the pencil of e_v.

The following key lemma is crucial to the analysis. We defer the proof to the next section. The reader may wish to skip that section and proceed to learn about the algorithm.

Lemma 8.4. *Assume $\mathcal{C}$ is obtained from a successful learning phase. Let $\mathcal{T}$ be a linear comparison tree that computes the upper hull of P. Then there is an entropy-sensitive linear comparison tree with expected depth $O(n + d_{\mathcal{T}})$ that computes $\mathcal{C}$-certificates for P.*

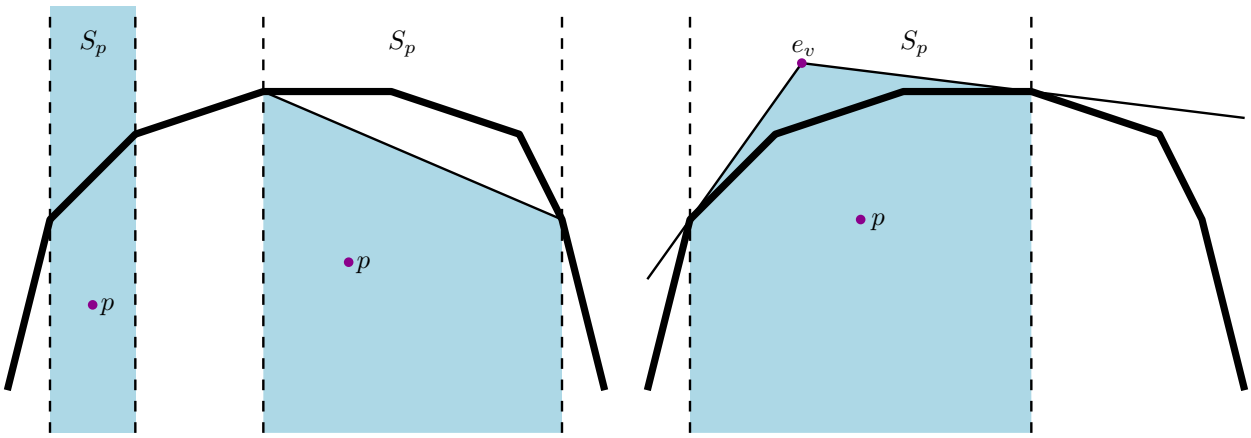

Figure 9: The $\mathcal{C}$-slab S_p associated with p can either be (i) a leaf slab; (ii) such that p lies below $\text{seg}(\mathcal{C}, S_p)$; or (iii) such that p lies in $\text{pen}(e_v)$ for a $\mathbf{V}$-extremal vertex e_v.

8.3 Proof of Lemma 8.4

The proof goes through several intermediate steps that successively transform a upper hull certificate into a $\mathcal{C}$-certificate. Each step incurs expected linear overhead. Then, it suffices to apply Lemma 4.4 to obtain an entropy-sensitive comparison tree with the claimed depth. A certificate γ is *extremal* if all witness pairs in γ are extremal. We provide the required chain of lemmas and give each proof in a different subsection. The following lemma is proved in Section 8.3.1.

Lemma 8.5. *Let $\mathcal{T}$ be a linear comparison tree for* $\text{conv}(P)$*. There exists a linear comparison tree with expected depth $d_{\mathcal{T}} + O(n)$ that computes an extremal certificate for P.*

A certificate is $\mathbf{V}$*-extremal* if it contains (i) a list of the $\mathbf{V}$-extremal points of P, sorted from left to right; and (ii) a list that stores for every other point $p \in P$ *either* a $\mathbf{V}$-extremal witness pair for p *or* two adjacent $\mathbf{V}$-extremal points e_1 and e_2 such that $x(e_1) \leq x(p) \leq x(e_2)$. The next lemma is proved in Section 8.3.2.

Lemma 8.6. *Let $\mathcal{T}$ be a linear comparison tree that computes extremal certificates. There is a linear comparison tree with expected depth $d_{\mathcal{T}} + O(n)$ that computes $\mathbf{V}$-extremal certificates.*

Finally, we go from $\mathbf{V}$-extremal certificates to $\mathcal{C}$-certificates. The proof is in Section 8.3.3.

Lemma 8.7. *Let $\mathcal{T}$ be a linear comparison tree that computes $\mathbf{V}$-extremal certificates. There is a linear comparison tree with expected depth $d_{\mathcal{T}} + O(n)$ that computes $\mathcal{C}$-certificates.*

Lemma 8.4 follows by combining Lemmas 8.5, 8.6 and 8.7 with Lemma 4.4.

8.3.1 Proof of Lemma 8.5

We transform $\mathcal{T}$ into a tree for extremal certificates. Since each leaf v of $\mathcal{T}$ corresponds to a certificate that is valid for all $P \in \mathcal{R}_v$, it suffices to show how to convert a given certificate γ for P to an extremal certificate by performing $O(n)$ additional comparisons on P. We describe an algorithm for this task.

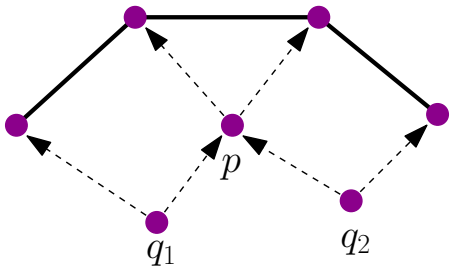

Figure 10: The **shortcut** operation: observe that computing the upper hull of the out-neighbors of p, q_1, and q_2 suffices for removing p from all witness pairs.

The algorithm uses two data structures: (i) a directed graph G whose vertices are a subset of P; and (ii) a stack S. Initially, S is empty and G has a vertex for every $p \in P$. For each non-extremal $p \in P$, we add two directed edges pq and pr to G, where (q, r) is the witness pair for p according to γ. In each step, the algorithm performs one of the following operations, until G has no more edges left (we will use the terms *point* and *vertex* interchangeably, since we always mean some $p \in P$).

- **Prune.** If G has a non-extremal vertex p with indegree zero, we delete p from G (together with its outgoing edges) and push it onto S.
- **Shortcut.** If G has a non-extremal vertex p with indegree 1 or 2, we find for each in-neighbor q of p a witness pair that does not include p, and we replace the out-edges from q by edges to this new pair. (We explain shortly how to do this.) The indegree of p is now zero.

An easy induction shows that the algorithm maintains the following invariants: (i) all non-extremal vertices in G have out-degree 2; (ii) all extremal vertices of G have out-degree 0; (iii) for each non-extremal vertex p of G, the two out-neighbors of p constitute a witness pair for p; (iv) every $p \in P$ is either in G or in S, but never both; (iv) when a point p is added to S, then we have a witness pair (q, r) for p such that $q, r \notin S$.

We analyze the number of comparisons on P. **Prune** needs no comparisons. **Shortcut** is done as follows: we consider for each in-neighbor q of p the upper convex hull U for p's two out-neighbors and q's other out-neighbor, and we find the edge of U that lies above q. Since the U constant size and since p has in-degree at most 2, this takes $O(1)$ comparisons, see Fig. 10. There are at most n calls to **Shortcut**, so the total number of comparisons is $O(n)$. Deciding which operation to perform depends solely on G and requires no comparisons on P.

We now argue that the algorithm cannot get stuck. That means that if G has at least one edge, **Prune** or **Shortcut** can be applied. Suppose that we cannot perform **Prune**. Then each non-extremal vertex has in-degree at least 1. Consider the subgraph G' of G induced by the non-extremal vertices. Since all extremal vertices have out-degree 0, all vertices in G' have in-degree at least 1. The average out-degree in G' is at most 2, so there must be a vertex with in-degree (in G') 1 or 2. This in-degree is the same in G, so **Shortcut** can be applied.

Thus, we can perform **Prune** or **Shortcut** until G has no more edges and all non-extremal points are on the stack S. Now we pop the points from S and find extremal witness pairs for them. Let p be the next point on S. By invariant (iv), there is a witness pair (q, r) for p whose vertices are not on S. Thus, each q and r is either extremal or we have an extremal witness pair for it. Therefore, we can find an extremal witness pair for p with $O(1)$ comparisons, as in **Shortcut**. We

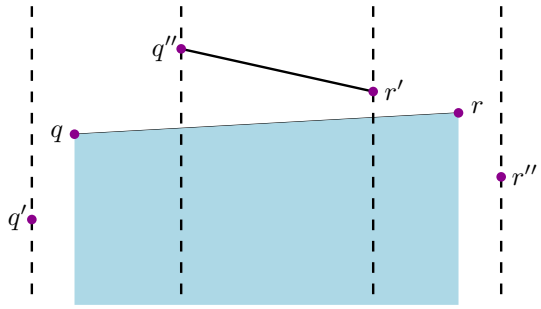

Figure 11: If p is in the blue region then $x(p) \in [x(q'), x(q'')]$, $p \in \text{lss}(q'', r')$, or $x(p) \in [x(r'), x(r'')]$.

repeat this process until S is empty. This takes $O(n)$ comparisons overall, so we obtain an extremal certificate γ' from γ with $O(n)$ comparisons on P.

8.3.2 Proof of Lemma 8.6

As in Section 8.3.1, it suffices to show how to convert a given extremal certificate into a $\mathbf{V}$-extremal one with $O(n)$ comparisons on P. This is done as follows. First, we determine the $\mathbf{V}$-extremal points on $\text{conv}(P)$. This takes $O(n)$ comparisons by a simultaneous traversal of $\text{conv}(P)$ and $\mathbf{V}$. Without further comparisons, we can now find for each extremal point p in P the two adjacent $\mathbf{V}$-extremal points that have p between them. This information is stored in the $\mathbf{V}$-extremal certificate.

Now let $p \in P$ be non-extremal, and let (q, r) be the corresponding extremal witness pair. We show how to find either a $\mathbf{V}$-extremal witness pair or the right pair of adjacent $\mathbf{V}$-extremal points. We have determined adjacent $\mathbf{V}$-extremal points q', q'' such that $x(q) \in [x(q'), x(q'')]$. (If q is itself $\mathbf{V}$-extremal, set $q' = q'' = q$.) Similarly, define adjacent $\mathbf{V}$-extremal points r', r''. We know that p lies in $\text{lss}(q, r)$ and hence $x(p) \in [x(q), x(r)]$. Furthermore, the points q', q, q'', r', r, r'' are in convex position. Since p is in $\text{lss}(q, r)$, one of the following must happen: $x(p) \in [x(q'), x(q'')]$, p lies in $\text{lss}(q'', r')$, or $x(p) \in [x(r'), x(r'')]$; see Fig. 11. We can determine which in $O(1)$ comparisons.

8.3.3 Proof of Lemma 8.7

As in Sections 8.3.1 and 8.3.2, we convert a $\mathbf{V}$-extremal certificate γ into a $\mathcal{C}$-certificate with $O(n)$ expected comparisons. For each $\mathbf{V}$-extremal point in γ, we perform a binary search to find the $\mathcal{C}$-leaf slab that contains it. This requires $o(n)$ comparisons, since there are at most $n/\log^2 n$ $\mathbf{V}$-extremal points and since each binary search needs $O(\log n)$ comparisons. Next, we check for each $i \leq k$ if the extremal point for v_i lies in ℓ_i^+. This takes one comparison per point. If any check fails, we declare failure and use binary search to find for every $p \in P$ a $\mathcal{C}$-leaf slab that contains it.

We now assume that there exists a $\mathbf{V}$-extremal point in every ℓ_i^+. (This implies that all $\mathbf{V}$-extremal points lie outside $\mathcal{C}$.) We use binary search to determine the pencil of each $\mathbf{V}$-extremal point. Again, this takes $o(n)$ comparisons. Now let $p \in P$ be not $\mathbf{V}$-extremal. We use $O(1)$ comparisons and either find the slab S_p or determine that p lies above $\mathcal{C}$. The certificate γ assigns to p two $\mathbf{V}$-extremal points e_1 and e_2 such that either (i) (e_1, e_2) is a $\mathbf{V}$-extremal witness pair for p; or (ii) e_1 and e_2 are adjacent and $x(e_1) \leq x(p) \leq x(e_2)$. We define f_1 as the rightmost visible point of $\mathcal{C}$ from e_1 and f_2 as the leftmost visible point from e_2.

Let us consider the first case; see Fig. 12(left). The point p is below $\overline{e_1 e_2}$. Since e_1, f_1, f_2, e_2 are in convex position, $\overline{e_1 e_2}$ is below their upper hull. This means that one of the following holds:

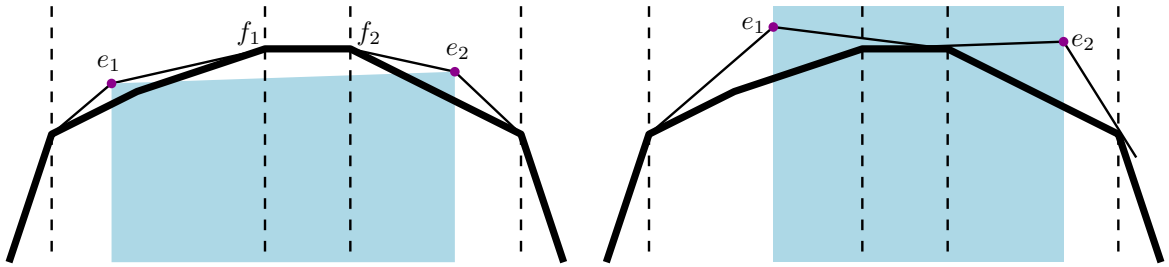

Figure 12: $\mathcal{C}$-certificates: in each part, p is contained in the shaded region.

$x(p) \in [x(e_1), x(f_1)]$, $x(p) \in [x(f_2), x(e_2)]$, or p is below $\overline{f_1 f_2}$. This can be determined in $O(1)$ comparisons. In the first two cases, p lies in a pencil (and hence we find an appropriate S_p), and in the last case, we find a witness $\mathcal{C}$-slab. Now for the second case. We need the following claim.

Claim 8.8. *If, for all i there is a $\mathbf{V}$-extremal point in ℓ_i^+, then the pencils of any two adjacent $\mathbf{V}$-extremal points either overlap or share a slab boundary.*

Proof. Refer again to Fig. 12(left). Let e_1 and e_2 be two adjacent $\mathbf{V}$-extremal vertices such that their pencil slabs neither overlap nor share a boundary. Then f_1 is not visible from e_2. Consider the edge a of $\mathcal{C}$ where f_1 is the left endpoint. The edge a is not visible from either e_1 or e_2 and is between them. By assumption, there is an extremal point x of P that sees a. But the point x cannot lie to the left of e_1 or to the right of e_2 (that would violate the extremal nature of e_1 or e_2). Hence, x must be between e_1 and e_2, contradicting the fact that they are adjacent. □

Claim 8.8 implies that p is comparable to one of $\text{pen}(e_1)$, $\text{pen}(e_2)$. By $O(1)$ comparisons, we can check if p is contained in either pencil or is above $\mathcal{C}$. Finally, for all points determined to be above $\mathcal{C}$, we use binary search to place them in a $\mathcal{C}$-leaf slab. This gives an appropriate S_p for each $p \in P$, and the canonical certificate is complete. We analyze the total number of comparisons. Let X be the indicator random variable for the event that there exist some ℓ_i^+ without a $\mathbf{V}$-extremal point. Let Y denote the number of points above $\mathcal{C}$. By Corollary 8.2, $\mathbf{E}[X] \leq n^{-3}$ and $\mathbf{E}[Y] = O(n/\log n)$. The number of comparisons is at most $O(Xn \log n + n + Y \log n)$, the expectation of which is $O(n)$.

8.4 The algorithm

Finally, we are ready to describe the details of our convex hull algorithm. It has two parts: the *location algorithm* and the *construction algorithm*. The former algorithm determines the location of the input points with respect to the canonical hull $\mathcal{C}$. It must be careful to learn just the right amount of information about each point. The latter algorithm uses this information to compute the convex hull of P quickly.

8.4.1 The location algorithm

Using Lemma 6.3, we obtain near-optimal search trees T_i for the $\mathcal{C}$-leaf slabs. The algorithm searches progressively for each $p_i \in P$ in its tree T_i. Again, we interleave the coordinate searches, and we abort the search for a point as soon as we have gained enough information about it. The location algorithm maintains the following information.

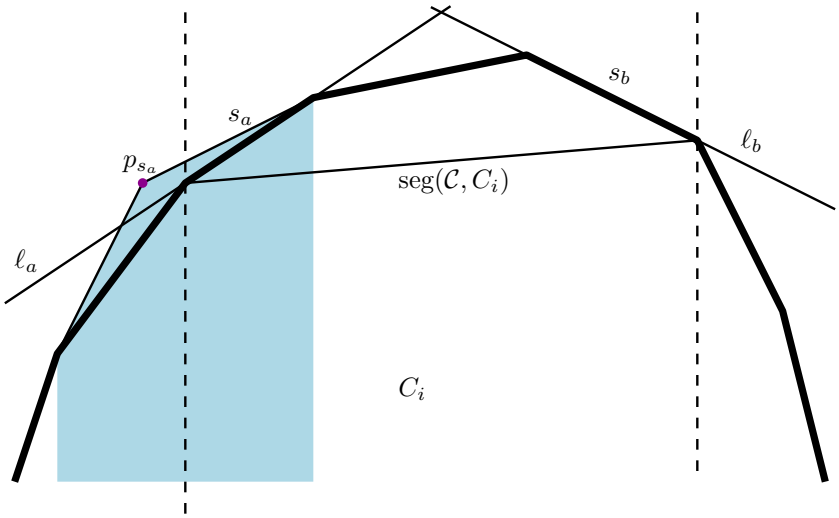

Figure 13: The algorithm: the boundary of C_i is shown dashed, the pencil $\mathrm{pen}(p_{s_a})$ is shaded.

- **Current slabs C_i.** For each point $p_i \in P$, we store a current $\mathcal{C}$-slab C_i containing p_i that corresponds to a node of T_i.
- **Active points A.** The active points are stored in a priority-queue $L(A)$ as in Claim 6.1. The key associated with an active point $p_i \in A$ is the size of the associated current slab C_i (represented as an integer between 1 and k).
- **Extremal candidates $\tilde{e}_v$.** For each canonical direction $v \in \mathbf{V}$, we store a point $\tilde{e}_v \in P$ that lies outside of $\mathcal{C}$. We call $\tilde{e}_v$ an *extremal candidate* for v.
- **Pencils for the points outside of $\mathcal{C}$.** For each point p that has been located outside of $\mathcal{C}$, we store its pencil $\mathrm{pen}(p)$.
- **Points with the left- and rightmost pencils.** For each edge s of $\mathcal{C}$, we store two points p_{s1} and p_{s2} such that (i) p_{s1} and p_{s2} lie outside of $\mathcal{C}$; (ii) s lies in $\mathrm{pen}(p_{s1})$ and $\mathrm{pen}(p_{s2})$; (iii) among all pencils seen so far that contain s, the left boundary of $\mathrm{pen}(p_{s1})$ lies furthest to the left and the right boundary of $\mathrm{pen}(p_{s2})$ lies furthest to the right.

Initially, we set $A = P$ and each C_i to the root of the corresponding search tree T_i. The extremal candidates $\tilde{e}_v$ as well as the points p_{s1}, p_{s2} with the left- and rightmost pencils are set to the null pointer. The location algorithm proceeds in *rounds*. In each round, we perform a `find-max` on $L(A)$. Suppose that `find-max` returns p_i. We compare p_i with the vertical line that corresponds to its current node in T_i and advance C_i to the appropriate child. This reduces the size of C_i, so we also perform a `decrease-key` on $L(A)$. Next, we distinguish three cases:
Case 1: p_i lies below $\mathrm{seg}(\mathcal{C}, C_i)$. We declare p_i inactive and `delete` it from $L(A)$.

For the next two cases, we know that p_i lies above $\mathrm{seg}(\mathcal{C}, C_i)$. Let ℓ_a, ℓ_b be the canonical lines that support the edges s_a and s_b of $\mathcal{C}$ that are incident to the boundary vertices of C_i and lie inside of C_i; see Fig. 13. We check where p_i lies with respect to ℓ_a and ℓ_b.
Case 2: p_i is above ℓ_a or above ℓ_b. This means that p_i is outside of $\mathcal{C}$. We declare p_i inactive and `delete` it from $L(A)$. Next, we perform a binary search to find $\mathrm{pen}(p_i)$ and all the edges of $\mathcal{C}$ that

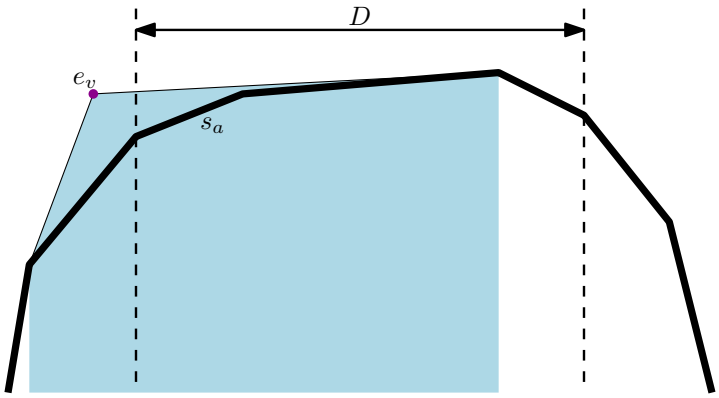

Figure 14: The left boundary of slab D is contained in the pencil slab of e_v.

are visible from p_i. For each such edge s, we compare p_i with the extremal candidate for s, and if p_i is more extreme in the corresponding direction, we update the extremal candidate accordingly. We also update the points p_{s1} and p_{s2} to p_i, if necessary.

Case 3: p_i lies below ℓ_a and ℓ_b. Recall that ℓ_a corresponds to the edge s_a of $\mathcal{C}$ and ℓ_b corresponds to the edge s_b of $\mathcal{C}$. We take the rightmost pencil for s_a and the leftmost pencil for s_b (if they exist); see Fig. 13. We compare p_i with these pencils. If p_i lies inside a pencil, we are done. If p_i is above a pencil, we learn that p_i lies outside of $\mathcal{C}$, and we process as in Case 2. In both situations, we declare p_i inactive and `delete` it from $L(A)$. If neither of these happen, p_i remains active.

The location algorithm continues until A is empty (note that every point becomes inactive eventually, because as soon as C_i is a leaf slab, either Case 1 or Case 2 applies).

8.4.2 Running time of the location algorithm

We now analyze the running time of the location algorithm, starting with some preliminary claims. The algorithm is deterministic, so we can talk of deterministic properties of the behavior on any input.

Claim 8.9. *Fix an input P. Let $e_v \in P$ be $\boldsymbol{V}$-extremal, and let S be the pencil slab for e_v. Once the search for e_v reaches a slab D with $|D| \leq |S|$, e_v will be identified as an extremal point for direction v.*

Proof. At least one vertical boundary line of D lies inside (the closure of) S and $D \cap S$ contains at least one leaf slab. Since S is a pencil slab, e_v sees all edges of $\mathcal{C}$ in $D \cap S$, so one of the boundary edges s_a or s_b corresponding to D, as used in Cases 2 and 3 of the algorithm (see Fig. 13), must be visible to e_v. Hence, e_v lies in $\ell_a^+ \cup \ell_b^+$, and this is detected in Case 2 of the location algorithm. □

Claim 8.10. *Let $e_v \in P$ be $\boldsymbol{V}$-extremal, and S the pencil slab for e_v. Suppose $p \in P$ lies in* $\mathrm{pen}(e_v)$. *Once the search for p reaches a slab D with $|D| \leq |S|$, the point p becomes inactive in the next round that it is processed.*

Proof. Consider the situation after the round in which p reaches D with $|D| \leq |S|$. The location algorithm schedules points according to the size of their current slab. Thus, when p is processed

next, all other active points are placed in slabs of size at most $|S|$. By Claim 8.9, if e_v is ever placed in slab of size at most $|S|$, the algorithm detects that it is $\mathbf{V}$-extremal and makes it inactive.

Hence, when p is processed next, e_v has been identified as the extremal point in direction v. Note that $D \cap S \neq \emptyset$, since $p \in D \cap S$. Some boundary (suppose it is the left one) of D lies inside S. Let s_a be the corresponding edge of $\mathcal{C}$, as used by the location algorithm; see Fig. 14. Since s_a is visible from e_v, and since e_v has been processed, it follows that the pencil slab of the rightmost pencil for s_a spans all of $D \cap S$. In Case 3 of the location algorithm (in this round), p will either be found inside this pencil, or outside of $\mathcal{C}$. Either way, p becomes inactive. □

We arrive at the main lemma of this section.

Lemma 8.11. *The total number of rounds in the location algorithm is $O(n + \text{OPT-CH})$.*

Proof. Let $\mathcal{T}$ be an entropy-sensitive comparison tree that computes a $\mathcal{C}$-certificate for P in expected depth $O(n + \text{OPT-CH})$. Such a tree exists by Lemma 8.4. Let v be a leaf of $\mathcal{T}$ with $d_v \leq n^2$. By Proposition 4.1, $\mathcal{R}_v$ is a Cartesian product $\mathcal{R}_v = \prod_{i=1}^{n} R_i$. The depth of v is $d_v = -\sum_{i=1}^{n} \log \Pr[p_i \in R_i]$, by Proposition 4.3. Now consider a random input P, conditioned on $P \in \mathcal{R}_v$. We show that expected number of rounds for P is $O(n + d_v)$. This also holds for $d_v > n^2$, since there are never more than n^2 rounds. The lemma follows, as the expected number of rounds is

$$\sum_{v \text{ leaf of } \mathcal{T}} \Pr[P \in \mathcal{R}_v] O(n + d_v) = O(n + d_{\mathcal{T}}).$$

Let v be a leaf with $d_v \leq n^2$ and γ the $\mathcal{C}$-certificate for v. The main technical argument is summarized in the following claim.

Claim 8.12. *Let $P \in \mathcal{R}_v$ and $p_i \in P$. The number of rounds involving p_i is at most one more than the number of steps required for an S_{p_i}-restricted search for p_i in T_i.*

Proof. By definition of $\mathcal{C}$-certificates, S_{p_i} is one of three types. Either S_{p_i} is a $\mathcal{C}$-leaf slab, p_i is below $\text{seg}(S_{p_i}, \mathcal{C})$, or S_{p_i} is a pencil slab of a $\mathbf{V}$-extremal vertex. In all cases, S_{p_i} contains R_i. When S_{p_i} is a leaf slab, an S_{p_i}-restricted search for p is a complete search. Hence, this is always at least the number of rounds involving p_i. Suppose p_i is below $\text{seg}(S_{p_i}, \mathcal{C})$. For any slab $S \subseteq S_{p_i}$, $\text{seg}(S, \mathcal{C})$ is above $\text{seg}(S_{p_i}, \mathcal{C})$. If p_i is located in any slab $S \subseteq S_{p_i}$, it is made inactive (Case 1 of the algorithm).

Now for the last case. The slab S_{p_i} is the pencil slab for a $\mathbf{V}$-extremal vertex e_v, such that the $\text{pen}(e_v)$, contains p_i. Suppose the search for p_i leads to slab $D \subseteq S_{p_i}$ and p_i is still active. By Claim 8.10, since $|D| \leq |S_{p_i}|$, p_i becomes inactive in the next round. □

Suppose P is chosen randomly from $\mathcal{R}_v$. The distribution restricted to p_i is simply random from R_i. By Lemma 5.3, the expected S_{p_i}-restricted search time is $O(1 - \log \Pr[p \in R_i])$. Combining with Claim 8.12, the expected number of rounds is

$$O(n - \sum_{i=1}^{n} \log \Pr[p_i \in R_i]) = O(n + d_v).$$

□

Lemma 8.13. *The expected running time of the location algorithm is $O(n + \text{OPT-CH})$.*

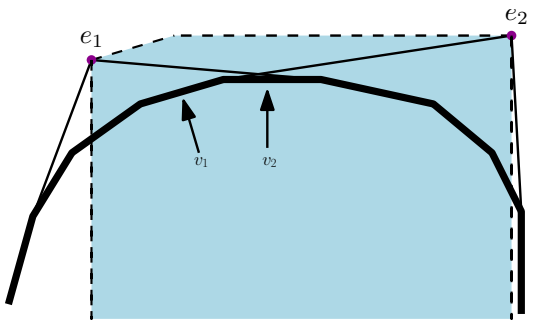

Figure 15: The lines perpendicular to directions v_1 and v_2 define the upper boundary of the shaded region where p lies. All edges seen by a point in the shaded region can be seen by either e_1 or e_2.

Proof. By Claim 6.1, the total overhead for the heap structure is linear in the number of rounds. The time to implement Cases 1 and 3 is $O(1)$, as we only need to compare p_i with a constant number of lines. Hence, the total time for this is at most proportional to the number of rounds.

In Case 2, we do a binary search for p_i and possibly update an extremal point (and pencil) for each edge visible from p_i. The case only occurs if p_i lies outside $\mathcal{C}$. By Corollary 8.3, the expected number such updates is $O(n/\log n)$. Overall, the total cost for Case 2 operations is $O(n)$. Combining with Lemma 8.11, the expected running time is $O(n + \text{OPT-CH})$. □

8.4.3 The construction algorithm

We now describe the upper hull construction that uses the information from the location algorithm to compute $\text{conv}(P)$ quickly. First, we dive into the geometry of pencils.

Claim 8.14. *Suppose that all* $\mathbf{V}$*-extremal points of* P *lie outside of* $\mathcal{C}$*, and let* e_v *be a* $\mathbf{V}$*-extremal point. Then* e_v *does not lie in the pencil of any other point outside* $\mathcal{C}$*.*

Proof. Suppose that $e_v \in \text{pen}(p)$ for another point $p \in P$ outside of $\mathcal{C}$. Then a vertex of $\text{pen}(p)$ would be more extremal in direction v than e_v. It cannot be p, since then e_v would not be extremal in direction v. It also cannot be a vertex of $\mathcal{C}$, because e_v lies in ℓ_v^+, while all vertices of $\mathcal{C}$ lie on ℓ_v or in ℓ_v^-. Thus, p cannot exist. □

Claim 8.15. *Suppose* $\mathbf{V}$*-extremal points of* P *lie outside of* $\mathcal{C}$*. Let* e_1 *and* e_2 *be two adjacent* $\mathbf{V}$*-extremal points and let* $p \in P$ *be above* $\mathcal{C}$ *such that the* x*-coordinate of* p *lies between the* x*-coordinates of* e_1 *and* e_2*. Then, the portion of* $\text{pen}(p)$ *below* $\mathcal{C}$ *is contained in* $\text{pen}(e_1) \cup \text{pen}(e_2)$*.*

Proof. By Claim 8.8, the (closures of the) pencil slabs of e_1 and e_2 overlap. Let v_1 be the last canonical direction for which e_1 is extremal and v_2 the first canonical direction for which e_2 is extremal. As e_1 and e_2 are adjacent, v_1 and v_2 are consecutive in $\mathbf{V}$; see Fig. 15. Consider the convex region bounded by the vertical downward ray from e_1, the vertical downward ray from e_2, the line parallel to ℓ_{v_1} through e_1, and the line parallel to ℓ_{v_2} through e_2. By construction, p lies inside this convex region (the shaded area in Fig. 15). By convexity, for every $v \in \mathbf{V}$, at least one of e_1 or e_2 is more extremal with respect to v than p. Hence, any edge of $\mathcal{C}$ visible from p is visible by either e_1 or e_2. The portion of $\text{pen}(p)$ below $\mathcal{C}$ is the union of regions below edges of $\mathcal{C}$ visible from p. Therefore, it lies in $\text{pen}(e_1) \cup \text{pen}(e_2)$. □

As described in Section 8.4.1, the location algorithm determines for for each $p \in P$ that either (a) p lies outside of $\mathcal{C}$; (b) p lies inside of $\mathcal{C}$, as witnessed by a segment $\text{seg}(\mathcal{C}, C_p)$; or (c) p lies in the pencil of a point located outside of $\mathcal{C}$. We also have the $\mathbf{V}$-extremal vertices e_v for all $v \in \mathbf{V}$. We now use this information in order to find $\text{conv}(P)$. By Corollary 8.3, with probability at least $1 - n^{-2}$, for each canonical direction in $\mathbf{V}$ there is a extremal point outside of $\mathcal{C}$ and the total number of points outside $\mathcal{C}$ is $O(n/\log n)$. We assume that these conditions hold. (Otherwise, we can compute $\text{conv}(P)$ in $O(n \log n)$ time, affecting the expected work only by a lower order term.)

For any point a, the *$\mathbf{V}$-pair for a* is the pair of adjacent $\mathbf{V}$-extremal points such that a lies between them. The construction algorithm goes through a series of steps. The exact details of some of these steps will be given in subsequent claims.

1. Compute the upper hull of the $\mathbf{V}$-extremal points.
2. For each vertex a of $\mathcal{C}$, compute the $\mathbf{V}$-pair for a.
3. For each input point p outside $\mathcal{C}$, compute its $\mathbf{V}$-pair by binary search.
4. For each input point p below a segment $\text{seg}(\mathcal{C}, C_p)$, in $O(1)$ time, either find its $\mathbf{V}$-pair or find a segment between $\mathbf{V}$-extremal points above it. (Details in Claim 8.18.)
5. For each input point p located in the pencil of a non-$\mathbf{V}$-extremal point, in $O(1)$ time, either locate p in the pencil of a $\mathbf{V}$-extremal point or determine that it is outside $\mathcal{C}$. In the latter case, use binary search to find its $\mathbf{V}$-pair.
6. For each input point p located in the pencil of an $\mathbf{V}$-extremal point, in $O(1)$ time, find a segment between $\mathbf{V}$-extremal points above it or find its $\mathbf{V}$-pair. (Details for both steps in Claim 8.19.)
7. By now, for every non-$\mathbf{V}$-extremal $p \in P$, we found a $\mathbf{V}$-pair or proved p non-extremal through a $\mathbf{V}$-extremal segment above it. For every pair (e_1, e_2) of adjacent $\mathbf{V}$-extremal points, find the set Q of points that lie above $\overline{e_1 e_2}$. Use an output-sensitive upper hull algorithm [21] to find the convex hull of Q. Finally, concatenate the resulting convex hulls to obtain $\text{conv}(P)$.

Claim 8.16. *After the location algorithm, for each canonical direction $v \in V$, the extremal candidate $\tilde{e}_v$ is the actual extremal point e_v in direction v.*

Proof. By Claim 8.14, e_v does not lie in the pencil of any other point in P. Hence, the location algorithm classifies e_v as the extremal candidate for v, and this choice does not change later on. □

Claim 8.17. *The total running time for Steps 1,2,3 and all binary searches in Step 5 is $O(n)$.*

Proof. There are $k = n/\log^2 n$ $\mathbf{V}$-extremal points, so finding their upper hull takes $O(n)$ time. We simultaneously traverse this upper hull and $\mathcal{C}$ to obtain the $\mathbf{V}$-pairs for all vertices of $\mathcal{C}$. As there are $O(n/\log n)$ points outside of $\mathcal{C}$, the total time for the binary searches is $O(n)$. □

Claim 8.18. *Suppose $p \in P$ lies below a segment $\text{seg}(\mathcal{C}, C_p)$. Using the information gathered before Step 4, we can either find its $\mathbf{V}$-pair or a segment between $\mathbf{V}$-extremal points above it in $O(1)$ time.*

Proof. Let a and b be the endpoints of $\text{seg}(\mathcal{C}, C_p)$. Consider the upper hull Z of the at most four $\mathbf{V}$-extremal points that define the $\mathbf{V}$-pairs of a and b; see Fig. 16(left). The hull Z has at most three edges, and only the middle one (if it exists) might not be between two adjacent $\mathbf{V}$-extremal points. If the middle edge of Z exists, it lies strictly above $\text{seg}(\mathcal{C}, C_p)$. This is because the endpoints of the middle edge have x-coordinates between $x(a)$ and $x(b)$ and lie outside of $\mathcal{C}$ (since they are $\mathbf{V}$-extremal), while $\text{seg}(\mathcal{C}, C_p)$ is inside $\mathcal{C}$. Now we compare p with the upper hull Z. This either finds a $\mathbf{V}$-pair for p (if p lies in the interval corresponding to the leftmost or rightmost edge of

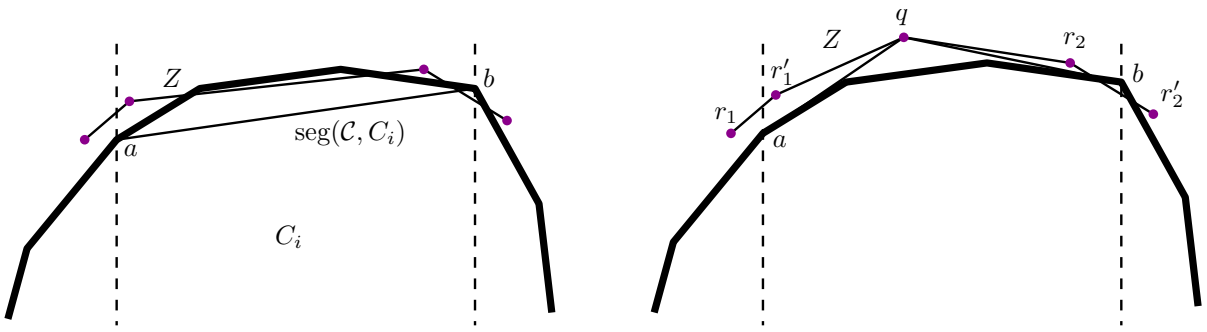

Figure 16: (left) In Step 4, the region below $\text{seg}(\mathcal{C}, C_i)$ is either below the middle segment of Z or between one of the two $\mathbf{V}$-pairs. (right) In Step 6, $\text{pen}(q)$ can be partitioned into regions below $\overline{qr'_1}$, below $\overline{qr_2}$, between r_1, r'_1, or between r_2, r'_2.

Z) or shows that p lies below a segment between two $\mathbf{V}$-extremal points (if it lies in the interval corresponding to the middle edge of Z). □

Claim 8.19. *Suppose $p \in P$ lies in $\text{pen}(q)$, where q is above $\mathcal{C}$, and the construction algorithm has completed Step 4. If q is non-$\mathbf{V}$-extremal, in $O(1)$ time, we can either find a $\mathbf{V}$-extremal point q' such that $p \in \text{pen}(q')$, or determine that p is above $\mathcal{C}$. If q is $\mathbf{V}$-extremal, then in $O(1)$ time we can find a $\mathbf{V}$-segment above p or find the $\mathbf{V}$-pair for p.*

Proof. Let q be non-$\mathbf{V}$-extremal. As q is outside $\mathcal{C}$, we know the $\mathbf{V}$-pair $\{e_1, e_2\}$ for q. By Claim 8.15, if p lies below $\mathcal{C}$, it is in $\text{pen}(e_1)$ or in $\text{pen}(e_2)$. We can determine which (if at all) in $O(1)$ time.

Let q be $\mathbf{V}$-extremal, and a, b the vertices of $\mathcal{C}$ on the boundary of $\text{pen}(q)$, where a is to the left; see Fig. 16(right). Let (r_1, r'_1) be a's $\mathbf{V}$-pair, where r_1 is to the left. Similarly, (r_2, r'_2) is b's $\mathbf{V}$-pair. The segments $\overline{qr'_1}$ and $\overline{qr_2}$ are above $\text{pen}(q)$. Furthermore, the pencil slab of q is between r_1 and r'_2. One of the following must be true for any point in $\text{pen}(q)$: it is below $\overline{qr'_1}$, below $\overline{qr_2}$, between (r_1, r'_1), or between (r_2, r'_2). This can be determined in $O(1)$ time. □

We are now armed with all the facts to bound the running time.

Lemma 8.20. *With the information from the location algorithm, $\text{conv}(P)$ can be computed in expected time $O(n \log\log n)$.*

Proof. By Claims 8.17, 8.18, and 8.19, the first six steps take $O(n)$ time. Let the $\mathbf{V}$-extremal points be ordered $e_1, \ldots, e_k$. Let X_i be the number of points in $\text{uss}(e_i, e_{i+1})$ and Y_i the number of extremal points in this set. We use an output-sensitive upper hull algorithm, so the running time of Step 7 is $O(\sum_{i \le k} X_i \log(Y_i + 1))$. By Lemma 8.1, this is $O(n \log\log n)$, as desired. □

9 Proofs of Lemma 8.1 and Lemma 8.2

We begin with some preliminaries about projective duality and a probabilistic claim about geometric constructions over product distributions. Consider an input P. As is well known, there is a *dual* set P^* of lines that helps us understand the properties of P. More precisely, we use the standard duality along the unit paraboloid that maps a point $p = (x(p), y(p))$ to the line $p^* : y = 2x(p)x - y(p)$ and vice versa. The *lower envelope* of P^* is the pointwise minimum of the n lines $p^*_1, \ldots, p^*_n$,

considered as univariate functions. We denote it by $\mathrm{lev}_0(P^*)$. There is a one-to-one correspondence between the vertices and edges of $\mathrm{lev}_0(P)$ and the edges and vertices of $\mathrm{conv}(P)$. More generally, for $z = 0, \ldots n$, the *z-level* of P^* is the closure of the set of all points that lie on lines of P^* and that have exactly z lines of P^* strictly below them. The z-level is an x-monotone polygonal curve, and we denote it by $\mathrm{lev}_z(P^*)$; see Fig. 17. Finally, the *($\leq z$)-level* of P^*, $\mathrm{lev}_{\leq z}(P^*)$, is the set of all points on lines in P^* that are on or below $\mathrm{lev}_z(P^*)$.

Consider the following abstract procedure. Let b be a constant, and for any set Q of b lines, let $\mathrm{reg}(Q)$ be some geometric region defined by the lines in Q. That is, $\mathrm{reg}(\cdot)$ is a function from sets of lines of size b to geometric regions (i.e., subsets of of $\mathbb{R}^2$). For example, $\mathrm{reg}(\cdot)$ may be a triangle or trapezoid formed by some lines in Q. For some such region R and a line ℓ, let $\chi(\ell, R)$ be a boolean function, taking as input a line and a geometric region.

Suppose we take a random instance $Q^* \sim \mathcal{D}$, and we apply some procedure to determine various subsets $Q_1, Q_2, \ldots$ of b lines from Q^*, chosen based on the sums $\sum_{i=1}^{n} \chi(q_i^*, \mathrm{reg}(Q_j))$. Now generate another random instance $P^* \sim \mathcal{D}$. What can be say about the values of $\sum_i \chi(p_i^*, \mathrm{reg}(Q_j))$? We expect them to resemble $\sum_i \chi(q_i^*, \mathrm{reg}(Q_j))$, but we have to deal with subtle issues of dependencies. In the former case, Q_j actually depends on Q^*, while in the latter case it does not. Nonetheless, we can apply concentration inequalities to make statements about $\sum_i \chi(p_i^*, \mathrm{reg}(Q_j))$. Let $J \subseteq [n]$ be a set of b indices in $[n]$, and set $Q_J^* := \{q_j^* \mid j \in J\}$. The following lemma can be seen as generalization of Lemma 3.2 in Ailon et al. [2], with a very similar proof.

Lemma 9.1. *Let $b > 0$ an integer, and $f_l(n)$, $f_u(n)$ increasing functions such that $f_u(n) \geq f_l(n) \geq c' b \log n$, for a constant $c' > 0$ and large enough n. The following holds with probability at least $1 - n^{-4}$ over a random $Q^* \sim \mathcal{D}$. For all index sets J of size b, if $\sum_{i \leq n} \chi(q_i^*, \mathrm{reg}(Q_J^*)) \in [f_l(n), f_u(n)]$, then for some absolute constant $\alpha \in (0, 1)$,*

$$\Pr_{P \sim \mathcal{D}}\Big[\sum_{i \leq n} \chi(p_i^*, \mathrm{reg}(Q_J^*)) \in [\alpha f_l(n), f_u(n)/\alpha]\Big] \geq 1 - n^{-3}.$$

Proof. Fix an index set $J \subseteq [n]$ of size b, and a set $Q_J = \{q_j^* \mid j \in J\}$ of lines. By independence, the distributions $\mathcal{D}_i$, $i \notin J$, remain unchanged, and we generate a random Q^* conditioned on Q_J being fixed. (This means that we sample random lines $q_i^* \sim \mathcal{D}_i$, for $i \notin J$.) We have $|\sum_{i \notin J} \chi(q_i^*, \mathrm{reg}(Q_J)) - \sum_i \chi(q_i^*, \mathrm{reg}(Q_J))| \leq b$, so if $\sum_i \chi(q_i^*, \mathrm{reg}(Q_J)) \in [f_l(n), f_u(n)]$, then $\sum_{i \notin J} \chi(q_i^*, \mathrm{reg}(Q_J)) \in [g_l(n), g_u(n)]$, for $g_l(n) = f_l(n) - b$ and $g_u(n) = f_u(n) + b$. What can we say about $\sum_{i \notin J} \chi(p_i^*, \mathrm{reg}(Q_J))$, for an independent $P^* \sim \mathcal{D}$? Since $\mathrm{reg}(Q_J)$ is fixed, $\chi(p_i^*, \mathrm{reg}(Q_J))$ and $\chi(q_i^*, \mathrm{reg}(Q_J))$ are identically distributed.

Define (independent) indicator variables $Z_i = \chi(q_i^*, \mathrm{reg}(Q_J))$, and let $\hat{Z} = \sum_{i \notin J} Z_i$ and $Z = \sum_i Z_i$. Given that one draw of $\hat{Z}$ is in $[g_l(n), g_u(n)]$, we want to give bounds on another draw. This is basically a Bayesian problem, in that we effectively construct a prior over $\mathbf{E}[\hat{Z}]$. Two Chernoff bounds suffice for the argument.

Claim 9.2. *Consider a single draw of $\hat{Z}$ and suppose that $\hat{Z} \in [g_l(n), g_u(n)]$, With probability at least $1 - n^{-c'b/5}$, $\mathbf{E}[\hat{Z}] \in [g_l(n)/6, 2g_u(n)]$.*

Proof. Apply Chernoff bounds [16, Theorem 1.1]: if $\mu := \mathbf{E}[\hat{Z}] < g_l(n)/6$, then $2e\mu < g_l(n)$, so

$$\Pr[\hat{Z} \geq g_l(n)] < 2^{-g_l(n)} < n^{-c'b/2},$$

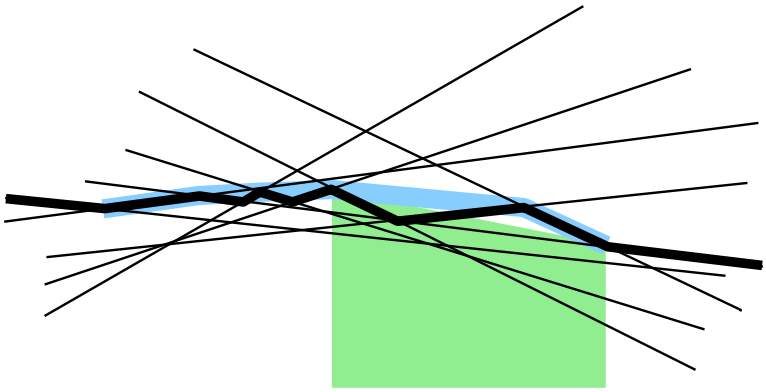

Figure 17: The arrangement of Q^*: the dark black line is $\mathrm{lev}_4(Q^*)$. The thick lighter line is H', the upper hull of the vertices in the level. The shaded region is a possible trapezoid τ_j.

noting that $g_l(n) = f_l(n) - b > (c'b/2)\log_2 n$. With probability at least $1 - n^{c'b/2}$, if $\hat{Z} \geq g_l(n)$, then $\mathbf{E}[\hat{Z}] \geq g_1(n)/6$. We repeat the argument with a lower tail Chernoff bound. If $\mu > 2g_u(n)$,

$$\Pr[\hat{Z} \leq g_u(n)] \leq \Pr[\hat{Z} \leq (1-1/2)\mu] < e^{-g_u(n)/4} < n^{-c'b/4}.$$

With probability at least $1 - n^{c'b/4}$, if $\hat{Z} \leq g_u(n)$, then $\mathbf{E}[\hat{Z}] \leq 2g_u(n)$. Now take a union bound. □

In Claim 9.2, we conditioned on a fixed Q_J, but the bound holds irrespective of Q_J, and hence is holds unconditionally. Therefore, for a fixed J, with probability at least $1 - n^{-c'b/5}$ over $Q^* \sim \mathcal{D}$, if $\hat{Z} \in [g_l(n), g_u(n)]$, then $\mathbf{E}[\hat{Z}] \in [g_l(n)/6, 2g_u(n)]$. Given that $|\hat{Z} - Z| \leq b$, this implies: if $Z \in [f_l(n), f_u(n)]$, then $\mathbf{E}[Z] \in [f_l(n)/7, 3f_u(n)]$.

There are $O(n^b)$ choices for J, so by a union bound the above holds for all J simultaneously with probability at least $1 - n^{-c'b/6}$. Suppose we choose a Q with this property, and consider drawing $P \sim \mathcal{D}$. This is effectively an independent draw of Z, so applying Chernoff bounds again, for sufficiently small constants α, β,

$$\Pr\Big[\sum_{i \leq n} \chi(p_i^*, \mathrm{reg}(Q_J^*)) \in [\alpha f_l(n), f_u(n)/\alpha]\Big] > 1 - \exp(-\beta f_l(n)) > 1 - n^{-3}.$$

□

9.1 Proof of Lemma 8.1

We sample a random input $Q^* \sim \mathcal{D}$ and take the $(\log^4 n)$-level of Q^*. Let H' the upper hull of its vertices; see Fig. 17.

Claim 9.3. *The hull H' has the following properties:*

1. *the curve H' lies below $\mathrm{lev}_{2\log^4 n}(Q^*)$;*
2. *each line of Q^* either supports an edge of H' or intersects it at most twice; and*
3. *H' has $O(n)$ vertices.*

Proof. Let p be any point on H', and let a, b be the closest vertices of H' with $x(a) \leq x(p) \leq x(b)$. Any line in Q^* below p must also be below a or b. There are exactly $\log^4 n$ lines under a and under

b, as they lie on the $(\log^4 n)$-level. Hence, there are at most $2\log^4 n$ lines below p. The second property is a direct consequence of convexity. The third property follows from the second: every vertex of H' lies on some line of Q^*, and hence there can be at most $2n$ vertices. □

Let $r_0, \ldots, r_k$ be the points given by every $\log^2 n$-th point in which a line of Q^* meets H' (either as an intersection point or as the endpoint of a segment), ordered from right to left. By Claim 9.3(2), there are $k = O(n/\log^2 n)$ points r_i. Let H be their upper upper hull. Clearly, H lies below H'. Draw a vertical downward ray through each vertex r_i. This subdivides the region below H into semi-unbounded trapezoids $\tau_0, \tau_1, \ldots$ with the following properties: (i) each vertical boundary ray of a trapezoid τ_j is intersected by at least $\log^4 n$ and at most $2\log^4 n$ lines of Q^* (Claim 9.3(1)); and (ii) the upper boundary segment of each τ_j is intersected by at most $\log^2 n$ lines in Q^* (by construction); see Fig. 17. The next claim follows from an application of Lemma 9.1.

Claim 9.4. *With probability at least $1 - n^{-4}$ (over Q), the following holds for all trapezoids τ_j: generate an independent $P^* \sim \mathcal{D}$.*

1. With probability (over P^*) at least $1 - n^{-3}$, there exists a line in P^* that intersects both boundary rays of τ_j; and

2. with probability (over P^*) at least $1 - n^{-3}$, at most $\log^5 n$ lines of P^* intersect τ_j.

Proof. We apply Lemma 9.1 for both parts. For a set $L = \{\ell_1, \ell_2, \ell_3, \ell_4\}$ of four lines, define $\mathrm{reg}(L)$ as the downward unbounded vertical trapezoid formed by the segment between the intersection points of ℓ_1, ℓ_2 and ℓ_3, ℓ_4. All trapezoids τ_j are of this form, with L a set of four lines from Q^*. Set $\chi(\ell, \tau)$ (for line ℓ and trapezoid τ) to 1 if ℓ intersects both parallel sides of τ, and 0 otherwise.

Since τ_j is an unbounded trapezoid, a line that intersects it either intersects the upper segment or intersects both boundary rays. In our sample Q, the number of lines with the former property is at most $\log^2 n$ and the number of lines with the latter property is in $[\log^4 n, 4\log^4 n]$. Hence, the sum $\sum_{i=1}^{n} \chi(q_i^*, \tau_j)$ is at least $\log^4 n - \log^2 n \geq (1/2)\log^4 n$ and at most $5\log^4 n$. By Lemma 9.1, the number of lines in P^* intersecting both vertical lines is $\Omega(\log^4 n)$ with probability at least $1 - n^{-3}$.

For the second part, we set $\chi(\ell, \tau) = 1$ if ℓ intersects τ, and 0 otherwise. Any line that intersects τ_j must intersect one of the vertical boundaries, so $\sum_{i=1}^{n} \chi(q_i^*, \tau_j) \in [\log^4 n, 4\log^4 n]$. By Lemma 9.1, the number of lines in P^* intersecting τ_j is $O(\log^4 n)$ with probability at least $1 - n^{-3}$. □

Each point r_i is dual to a line r_i^*. We define the directions in $\mathbf{V}$ by taking upward unit normals to the lines r_j^*. (Since the r_i's are ordered from right to left, this gives $\mathbf{V}$ in clockwise order.) These directions can be found in $O(n \operatorname{poly}\log(n))$ time: we can compute $\mathrm{lev}_{\log^4 n}(P^*)$ and its upper hull H' in $O(n \operatorname{poly}\log n)$ time [14, 15]. To determine the points r_j, we perform $O(n)$ binary searches over H', and then sort the intersection points. When r_j is known, v_j can be found in constant time.

Now consider a random P. The $\mathbf{V}$-extremal vertices e_i and e_{i+1} correspond to the lowest line in P^* that intersects the left and the right boundary ray of τ_i. The number of extremal points between e_i and e_{i+1} is the number of edges on the lower envelope of P^* between $x(r_i)$ and $x(r_{i+1})$. By Claim 9.4(1), this lower envelope lies entirely inside τ_i with probability at least $1 - n^{-3}$. By Claim 9.4(2) (and a union bound), the number Y_i of extremal points between e_i and e_{i+1} is at most

$\log^5 n$ with probability at least $1 - 2n^{-3}$. Thus,

$$\begin{aligned}
&\mathbf{E}[X_i \log(Y_i + 1)] \\
&\leq \mathbf{E}[X_i \log(\log^5 n + 1) \mid Y_i \leq \log^5 n] \Pr[Y_i \leq \log^5 n] + \mathbf{E}[X_i \log(Y_i + 1) \mid Y_i > \log^5 n] \Pr[Y_i > \log^5 n] \\
&\leq \mathbf{E}[X_i \mid Y_i \leq \log^5 n] \Pr[Y_i \leq \log^5 n] O(\log \log n) + O(n^2)(1/2n^3) \\
&\leq \mathbf{E}[X_i] O(\log \log n) + O(1).
\end{aligned}$$

Adding over i,

$$\begin{aligned}
\sum_{i=1}^{k} \mathbf{E}[X_i \log(Y_i + 1)] &\leq \sum_{i=1}^{k} \mathbf{E}[X_i] O(\log \log n) + O(1) \\
&= \mathbf{E}\Big[\sum_{i=1}^{k} X_i\Big] O(\log \log n) + O(n) = O(n \log \log n).
\end{aligned}$$

9.2 Proof of Lemma 8.2

To compute the canonical lines ℓ_j for the directions $v_j \in \mathbf{V}$, we consider again the dual sample Q^*. Let s_j be the point on $\mathrm{lev}_{\gamma c \log n}(Q)$ with the same x-coordinate as r_j, where $\gamma > 0$ is a sufficiently small constant. Set $\ell_j = s_j^*$. Then ℓ_j is normal to v_j, and the construction takes $O(n \operatorname{poly} \log n)$ time. We restate the main technical part of Lemma 8.2.

Lemma 9.5. *With probability at least $1 - n^{-4}$ over the construction, for every ℓ_j,*

$$\Pr_{P \sim \mathcal{D}}[|\ell_j^+ \cap P| \in [1, c \log n]] \geq 1 - n^{-3}.$$

Proof. A point p lies in ℓ_j^+ if and only if p^* intersects the downward vertical ray R_j from s_j. We set up an application of Lemma 9.1. For a pair of lines ℓ_1, ℓ_2 (all in dual space), define $\mathrm{reg}(\ell_1, \ell_2)$ as the downward vertical ray from $\ell_1 \cap \ell_2$. Every s_j is formed by the intersection of two lines from Q^*. For such a region R_j and line ℓ', set $\chi(\ell', R)$ to be 1 if ℓ' intersects R_j and 0 otherwise. By construction, $\sum_i \chi(q_i^*, R_j) = \gamma c \log n$. We apply Lemma 9.1. With probability at least $1 - n^{-4}$ over Q^* (for sufficiently large c and small enough γ), $\Pr_{P \sim \mathcal{D}}[\sum_i \chi(p_i^*, R_j) \in [1, c \log n]] \geq 1 - n^{-3}$. □

Acknowledgements

C. Seshadhri was supported by the Early Career LDRD program at Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. W. Mulzer was supported in part by DFG grant MU/3501/1.

References

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *Proc. 50th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 129–138, 2009.

[2] N. Ailon, B. Chazelle, K. L. Clarkson, D. Liu, W. Mulzer, and C. Seshadhri. Self-improving algorithms. *SIAM J. Comput.*, 40(2):350–375, 2011.

[3] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Self-improving algorithms. In *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 261–270, 2006.

[4] J. Barbay. Adaptive (analysis of) algorithms for convex hulls and related problems. `http://www.cs.uwaterloo.ca/~jbarbay/Recherche/Publishing/Publications/#asimuht`, 2008.

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer Verlag, Berlin, third edition, 2008.

[6] P. Bose, L. Devroye, K. Douïeb, V. Dujmović, J. King, and P. Morin. Odds-on trees. `arXiv:1002.1092`, 2010.

[7] K. Buchin, M. Löffler, P. Morin, and W. Mulzer. Preprocessing imprecise points for Delaunay triangulation: Simplified and extended. *Algorithmica*, 61(3):674–693, 2011.

[8] C. Buchta. On the average number of maxima in a set of vectors. *Inform. Process. Lett.*, 33(2):63–66, 1989.

[9] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(1):145–158, 1993.

[10] K. L. Clarkson. New applications of random sampling to computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[11] K. L. Clarkson, W. Mulzer, and C. Seshadhri. Self-improving algorithms for convex hulls. In *Proc. 21st Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1546–1565, 2010.

[12] K. L. Clarkson and C. Seshadhri. Self-improving algorithms for Delaunay triangulations. In *Proc. 24th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 226–232, 2008.

[13] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4(1):387–421, 1989.

[14] R. Cole, M. Sharir, and C.-K. Yap. On k-hulls and related problems. *SIAM J. Comput.*, 16(1):61–77, 1987.

[15] T. K. Dey. Improved bounds for planar k-sets and related problems. *Discrete Comput. Geom.*, 19(3):373–382, 1998.

[16] D. P. Dubhashi and A. Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, Cambridge, 2009.

[17] E. Ezra and W. Mulzer. Convex hull of points lying on lines in time after preprocessing. *Comput. Geom. Theory Appl.*, 46(4):417–434, 2013.

[18] M. J. Golin. A provably fast linear-expected-time maxima-finding algorithm. *Algorithmica*, 11(6):501–524, 1994.

[19] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 714–723, 1993.

[20] M. Held and J. S. B. Mitchell. Triangulating input-constrained planar point sets. *Inform. Process. Lett.*, 109(1):54–56, 2008.

[21] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, 1986.

[22] M. J. van Kreveld, M. Löffler, and J. S. B. Mitchell. Preprocessing imprecise points and splitting triangulations. *SIAM J. Comput.*, 39(7):2990–3000, 2010.

[23] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[24] M. Löffler and J. Snoeyink. Delaunay triangulation of imprecise points in linear time after preprocessing. *Comput. Geom. Theory Appl.*, 43(3):234–242, 2010.