Self-improving Algorithms for Coordinate-wise Maxima

[Extended Abstract]

Kenneth L. Clarkson
IBM Almaden Research Center
San Jose, USA
klclarks@us.ibm.com

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

C. Seshadhri
Sandia National Laboratories
Livermore, USA
scomand@sandia.gov

ABSTRACT

Computing the coordinate-wise maxima of a planar point set is a classic and well-studied problem in computational geometry. We give an algorithm for this problem in the *self-improving setting*. We have n (unknown) independent distributions $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ of planar points. An input pointset $(p_1, p_2, \ldots, p_n)$ is generated by taking an independent sample p_i from each $\mathcal{D}_i$, so the input distribution $\mathcal{D}$ is the product $\prod_i \mathcal{D}_i$. A self-improving algorithm repeatedly gets input sets from the distribution $\mathcal{D}$ (which is *a priori* unknown) and tries to optimize its running time for $\mathcal{D}$. Our algorithm uses the first few inputs to learn salient features of the distribution, and then becomes an optimal algorithm for distribution $\mathcal{D}$. Let $\mathrm{OPT}_\mathcal{D}$ denote the expected depth of an *optimal* linear comparison tree computing the maxima for distribution $\mathcal{D}$. Our algorithm eventually has an expected running time of $O(\mathrm{OPT}_\mathcal{D} + n)$, even though it did not know $\mathcal{D}$ to begin with.

Our result requires new tools to understand linear comparison trees for computing maxima. We show how to convert general linear comparison trees to very restricted versions, which can then be related to the running time of our algorithm. An interesting feature of our algorithm is an interleaved search, where the algorithm tries to determine the likeliest point to be maximal with minimal computation. This allows the running time to be truly optimal for the distribution $\mathcal{D}$.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

General Terms

Algorithms, Theory

Keywords

Coordinate-wise maxima; Self-improving algorithms

1. INTRODUCTION

Given a set P of n points in the plane, the *maxima* problem is to find those points $p \in P$ for which no other point in P has a larger x-coordinate and a larger y-coordinate. More formally, for $p \in \mathbb{R}^2$, let $x(p)$ and $y(p)$ denote the x and y coordinates of p. Then p' *dominates* p if and only if $x(p') \geq x(p)$, $y(p') \geq y(p)$, and one of these inequalities is strict. The desired points are those in P that are not dominated by any other points in P. The set of maxima is also known as a *skyline* in the database literature [BKS01] and as a *Pareto frontier*.

This algorithmic problem has been studied since at least 1975 [KLP75], when Kung et al. described an algorithm with an $O(n \log n)$ worst-case time and gave an $\Omega(n \log n)$ lower bound. Results since then include average-case running times of $n + \tilde{O}(n^{6/7})$ point-wise comparisons [Gol94]; output-sensitive algorithms needing $O(n \log h)$ time when there are h maxima [KS86]; and algorithms operating in external-memory models [GTVV93]. A major problem with worst-case analysis is that it may not reflect the behavior of real-world inputs. Worst-case algorithms are tailor-made for extreme inputs, none of which may occur (with reasonable frequency) in practice. Average-case analysis tries to address this problem by assuming some fixed distribution on inputs; for maxima, the property of coordinate-wise independence covers a broad range of inputs, and allows a clean analysis [Buc89], but is unrealistic even so. The right distribution to analyze remains a point of investigation. Nonetheless, the assumption of randomly distributed inputs is very natural and one worthy of further research.

The self-improving model. Ailon et al. introduced the self-improving model to address this issue [ACCL06]. In this model, there is some fixed but unknown input distribution $\mathcal{D}$ that generates independent inputs, that is, whole input sets P. The algorithm initially undergoes a *learning phase*, where it processes inputs with a worst-case guarantee but tries to learn information about $\mathcal{D}$. The aim of the algorithm is to become optimal *for the distribution* $\mathcal{D}$. After seeing some (hopefully small) number of inputs, the algorithm shifts into the *limiting phase*. Now, the algorithm is tuned for $\mathcal{D}$ and the expected running time is (ideally) optimal for $\mathcal{D}$. A self-improving algorithm can be thought of as an algorithm that attains the optimal average-case running time for all, or at least a large class of, distributions $\mathcal{D}$.

Following earlier self-improving algorithms, we assume the input has a product distribution. An input is a set of n points $P = (p_1, p_2, \ldots, p_n)$ 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, and the only assumption made is their independence. There are lower bounds showing that some restriction on $\mathcal{D}$ is necessary for a reasonable self-improving algorithm, as we explain later.

The first self-improving algorithm was for sorting; this was extended to Delaunay triangulations, with these results eventually merged [CS08, ACC$^+$11]. A self-improving algorithm for planar convex hulls was given by Clarkson et al. [CMS10], however their analysis was recently discovered to be flawed.

1.1 Main result

Our main result is a self-improving algorithm for planar coordinate-wise maxima over product distributions. We need some basic definitions before stating our main theorem. We explain what it means for a maxima algorithm to be optimal for a distribution $\mathcal{D}$. This in turn requires a notion of *certificates* for maxima, which allow the correctness of the output to be verified in $O(n)$ time. Any procedure for computing maxima must provide some "reason" to deem an input point p non-maximal. The simplest certificate would be to provide an input point dominating p. Most current algorithms implicitly give exactly such certificates [KLP75, Gol94, KS86].

DEFINITION 1.1. *A* certificate γ *has: (i) the sequence of the indices of the maximal points, sorted from left to right; (ii) for each non-maximal point, a* per-point certificate *of non-maximality, which is simply the index of an input point that dominates it. We say that a certificate γ is* valid *for an input P if γ satisfies these conditions for P.*

The model of computation that we use to define optimality is a linear computation tree that generates query lines using the input points. In particular, our model includes the usual CCW-test that forms the basis for many geometric algorithms.

Let ℓ be a directed line. We use ℓ^+ to denote the open halfplane to the left of ℓ and ℓ^- to denote the open halfplane to the right of ℓ.

DEFINITION 1.2. *A* linear comparison tree $\mathcal{T}$ *is a binary tree such that each node v of $\mathcal{T}$ is labeled with a query of the form "$p \in \ell_v^+$?". Here p denotes an input point and ℓ_v denotes a directed line. The line ℓ_v can be obtained in three ways: (i) it can be a line independent of the input (but dependent on the node v); (ii) it can be a line with a slope independent of the input (but dependent on v) passing through a given input point; (iii) it can be a line through an input point and through a point q independent of the input (but dependent on v); (iv) it can be the line defined by two distinct input points. A linear comparison tree is* restricted *if it only makes queries of type (i).*

A linear comparison tree $\mathcal{T}$ computes the maxima *for P if each leaf corresponds to a certificate. This means that each leaf v of $\mathcal{T}$ is labeled with a certificate γ that is valid for every possible input P that reaches v.*

Let $\mathcal{T}$ be a linear comparison tree and v be a node of $\mathcal{T}$. Note that v corresponds to 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 $P \in \mathcal{R}_v$. If $\mathcal{T}$ is *restricted*, then $\mathcal{R}_v$ is the Cartesian product of a sequence $(R_1, R_2, \ldots, R_n)$ of polygonal regions. The *depth* of v, denoted by d_v, is the length of the path from the root of $\mathcal{T}$ to v. Given $\mathcal{T}$, there exists exactly one leaf $v(P)$ that is reached by the evaluation of $\mathcal{T}$ on input P. The *expected depth* of $\mathcal{T}$ over $\mathcal{D}$, $d_\mathcal{D}(\mathcal{T})$, is defined as $\mathbf{E}_{P\sim\mathcal{D}}[d_{v(P)}]$. Consider some comparison based algorithm A that is modeled by such a tree $\mathcal{T}$. The expected depth of $\mathcal{T}$ is a lower bound on the number of comparisons performed by A.

Let $\mathbf{T}$ be the set of trees that compute the maxima of n points. We define $\mathrm{OPT}_\mathcal{D} = \inf_{\mathcal{T}\in\mathbf{T}} d_\mathcal{D}(\mathcal{T})$. This is a lower bound on the expected time taken by *any* linear comparison tree to compute the maxima of inputs distributed according to $\mathcal{D}$. We would like our algorithm to have a running time comparable to $\mathrm{OPT}_\mathcal{D}$.

THEOREM 1.3. *Let $\varepsilon > 0$ be a fixed constant and $\mathcal{D}_1$, $\mathcal{D}_2$, $\ldots, \mathcal{D}_n$ be independent planar point distributions. The input distribution is $\mathcal{D} = \prod_i \mathcal{D}_i$. There is a self-improving algorithm to compute the coordinate-wise maxima whose expected time in the limiting phase is $O(\varepsilon^{-1}(n + OPT_\mathcal{D}))$. The learning phase lasts for $O(n^\varepsilon)$ inputs and the space requirement is $O(n^{1+\varepsilon})$.*

There are lower bounds in [ACC$^+$11] (for sorters) implying that a self-improving maxima algorithm that works for all distributions requires exponential storage, and that the time-space tradeoff (wrt ε) in the above theorem is optimal.

Challenges. One might think that since self-improving sorters are known, an algorithm for maxima should follow directly. But this reduction is only valid for $O(n \log n)$ algorithms. Consider Figure 1(i). The distributions $\mathcal{D}_1$, $\mathcal{D}_2$, $\ldots, \mathcal{D}_{n/2}$ generate the fixed points shown. The remaining distributions generate a random point from a line below L. Observe that an algorithm that wishes to sort the x-coordinates requires $\Omega(n \log n)$ time. On the other hand, there is a simple comparison tree that determines the maxima in $O(n)$ time. For all p_j where $j > n/2$, the tree simply checks if $p_{n/2}$ dominates p_j. After that, it performs a linear scan and outputs a certificate.

We stress that even though the points are independent, the collection of maxima exhibits strong dependencies. In Figure 1(ii), suppose a distribution $\mathcal{D}_i$ generates either p_h or p_ℓ; if p_ℓ is chosen, we must consider the dominance relations among the remaining points, while if p_h is chosen, no such evaluation is required. The optimal search tree for a distribution $\mathcal{D}$ must exploit this complex dependency.

Indeed, arguing about optimality is one of the key contributions of this work. Previous self-improving algorithms employed information-theoretic optimality arguments. These are extremely difficult to analyze for settings like maxima, where some points are more important to process that others, as in Figure 1. (The main error in the self-improving convex hull paper [CMS10] was an incorrect consideration of dependencies.) We focus on a somewhat weaker notion of optimality—linear comparison trees—that nonetheless covers most (if not all) important algorithms for maxima.

In Section 3, we describe how to convert linear comparison trees into restricted forms that use much more structured (and simpler) queries. Restricted trees are much more amenable to analysis. In some sense, a restricted tree decouples the individual input points and makes the maxima com-

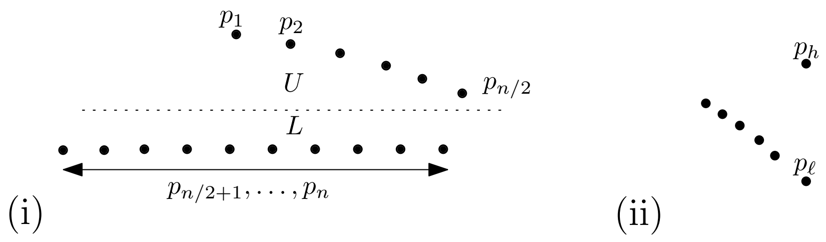

Figure 1: Examples of difficult distributions

putation amenable to separate $\mathcal{D}_i$-optimal searches. A leaf of a restricted tree is associated with a sequence of polygons $(R_1, R_2, \ldots, R_n)$ such that the leaf is visited if and only if every $p_i \in R_i$, and conditioned on that event, the p_i remain independent. This independence is extremely important for the analysis. We design an algorithm whose behavior can be related to the restricted tree. Intuitively, if the algorithm spends many comparisons involving a single point, then we can argue that the optimal restricted tree must also do the same. We give more details about the algorithm in Section 2.

1.2 Previous work

Afshani et al. [ABC09] introduced a model of *instance-optimality* applying to algorithmic problems including planar convex hulls and maxima. (However, their model is different from, and in a sense weaker than, the prior notion of instance-optimality introduced by Fagin et al. [FLN01].) All previous (e.g., output sensitive and instance optimal) algorithms require expected $\Omega(n \log n)$ time for the distribution given in Figure 1, though an optimal self-improving algorithm only requires $O(n)$ expected time. (This was also discussed in [CMS10] with a similar example.),

We also mention the paradigm of preprocessing regions in order to compute certain geometric structures faster (see, e.g., [BLMM11, EM11, HM08, LS10, vKLM10]). Here, we are given a set $\mathcal{R}$ of planar regions, and we would like to preprocess $\mathcal{R}$ in order to quickly find the (Delaunay) triangulation (or convex hull) for any point set which contains exactly one point from each region in $\mathcal{R}$. This setting is adversarial, but if we only consider point sets where a point is randomly drawn from each region, it can be regarded as a special case of our setting. In this view, these results give us bounds on the running time a self-improving algorithm can achieve if $\mathcal{D}$ draws its points from disjoint planar regions.

1.3 Preliminaries and notation

Before we begin, let us define some basic concepts and agree on a few notational conventions. We use c for a sufficiently large constant, and we write $\log x$ to denote the logarithm of x in base 2. All the probability distributions are assumed to be continuous. (It is not necessary to do this, but it makes many calculations a lot simpler.)

Given a polygonal region $R \subseteq \mathbb{R}^2$ and a probability distribution $\mathcal{D}$ on the plane, we call ℓ a *halving line* for R (with respect to $\mathcal{D}$) if

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

Note that if $\Pr_{p\sim\mathcal{D}}[p \in R] = 0$, every line is a halving line for R. If not, a halving line exactly halves the conditional probability for p being in each of the corresponding halfplanes, conditioned on p lying inside R.

Define a *vertical slab structure* $\mathbf{S}$ as a sequence of vertical lines partitioning the plane into vertical regions, called *leaf slabs*. (We will consider the latter to be the open regions between the vertical lines. Since we assume that our distributions are continuous, we abuse notation and consider the leaf slabs to partition the plane.) More generally, a *slab* is the region between any two vertical lines of the $\mathbf{S}$. The *size* of the slab structure is the number of leaf slabs it contains. We denote it by $|\mathbf{S}|$. Furthermore, for any slab S, the probability that $p_i \sim \mathcal{D}_i$ is in S is denoted by $q(i, S)$.

A *search tree* T over $\mathbf{S}$ is a comparison tree that locates a point within leaf slabs of $\mathbf{S}$. Each internal node compares the x-coordinate of the point with a vertical line of $\mathbf{S}$, and moves left or right accordingly. We associate each internal node v with a slab S_v (any point in S_v will encounter v along its search).

1.4 Tools from self-improving algorithms

We introduce some tools that were developed in previous self-improving results. The ideas are by and large old, but our presentation in this form is new. We feel that the following statements (especially Lemma 1.6) are of independent interest.

We define the notion of *restricted searches*, introduced in [CMS10]. This notion is central to our final optimality proof. (The lemma and formulation as given here are new.) Let $\mathbf{U}$ be an ordered set and $\mathcal{F}$ be a distribution over $\mathbf{U}$. For any element $j \in \mathbf{U}$, q_j is the probability of j according to $\mathcal{F}$. For any interval S of $\mathbf{U}$, the total probability of S is q_S.

We let T denote a search tree over $\mathbf{U}$. It will be convenient to think of T as (at most) ternary, where each node has at most 2 children that are internal nodes. In our application of the lemma, $\mathbf{U}$ will just be the set of leaf slabs of a slab structure $\mathbf{S}$. We now introduce some definitions regarding restricted searches and search trees.

Definition 1.4. *Consider a distribution $\mathcal{F}$ and an interval S of U. An* S-restricted distribution *is given by the probabilities (for element $r \in U$) $q'_r / \sum_{j\in U} q'_j$, where the sequence $\{q'_j | j \in U\}$ has the following property. For each $j \in S$, $0 \le q'_j \le q_j$. For every other j, $q'_j = 0$.*

Suppose $j \in S$. An S-restricted search *is a search for j in T that terminates once j is located in any interval contained in S.*

For any sequence of numbers $\{q'_j | j \in U\}$ and $S \subseteq U$, we use q'_S to denote $\sum_{j\in S} q'_j$.

Definition 1.5. *Let $\mu \in (0,1)$ be a parameter. A search tree T over $\boldsymbol{U}$ is μ-reducing if: for any internal node S and for any non-leaf child S' of S, $q_{S'} \leq \mu q_S$.*

A search tree T is c-optimal for restricted searches over $\mathcal{F}$ *if: for all S and S-restricted distributions $\mathcal{F}_S$, the expected time of an S-restricted search over $\mathcal{F}_S$ is at most $c(-\log q'_S + 1)$. (The probabilities q' are as given in Definition 1.4.)*

We give the main lemma about restricted searches. A tree that is optimal for searches over $\mathcal{F}$ also works for restricted distributions. The proof is given in the full version of the paper.

Lemma 1.6. *Suppose T is a μ-reducing search tree for $\mathcal{F}$. Then T is $O(1/\log(1/\mu))$-optimal for restricted searches over $\mathcal{F}$.*

We list theorems about data structures that are built in the learning phase. Similar structures were first constructed in [ACC+11], and the following can be proved using their ideas. The data structures involve construction of slab structures and specialized search trees for each distribution $\mathcal{D}_i$. It is also important that these trees can be represented in small space, to satisfy the requirements of Theorem 1.3. The following lemmas give us the details of the data structures required. Because this is not a major contribution of this paper, we relegate the details to §5.

Lemma 1.7. *We can construct a slab structure $\boldsymbol{S}$ with $O(n)$ leaf slabs such that, with probability $1 - n^{-3}$ over the construction of $\boldsymbol{S}$, the following holds. For a leaf slab λ of $\boldsymbol{S}$, let X_λ denote the number of points in a random input P that fall into λ. For every leaf slab λ of $\boldsymbol{S}$, we have $\mathbf{E}[X_\lambda^2] = O(1)$. The construction takes $O(\log n)$ rounds and $O(n \log^2 n)$ time.*

Lemma 1.8. *Let $\varepsilon > 0$ be a fixed parameter. In $O(n^\varepsilon)$ rounds and $O(n^{1+\varepsilon})$ time, we can construct search trees T_1, T_2, ..., T_n over $\boldsymbol{S}$ such that the following holds. (i) the trees can be represented in $O(n^{1+\varepsilon})$ total space; (ii) with probability $1 - n^{-3}$ over the construction of the T_is, every T_i is $O(1/\varepsilon)$-optimal for restricted searches over $\mathcal{D}_i$.*

2. OUTLINE

We start by providing a very informal overview of the algorithm. Then, we shall explain how the optimality is shown.

If the points of P are sorted by x-coordinate, the maxima of P can be found easily by a right-to-left sweep over P: we maintain the largest y-coordinate Y of the points traversed so far; when a point p is visited in the traversal, if $y(p) < Y$, then p is non-maximal, and the point p_j with $Y = y(p_j)$ gives a per-point certificate for p's non-maximality. If $y(p) \geq Y$, then p is maximal, and can be put at the beginning of the certificate list of maxima of P.

This suggests the following approach to a self-improving algorithm for maxima: sort P with a self-improving sorter and then use the traversal. The self-improving sorter of [ACC+11] works by locating each point of P within the slab structure $\mathbf{S}$ of Lemma 1.7 using the trees T_i of Lemma 1.8.

While this approach does use $\mathbf{S}$ and the T_i's, it is not optimal for maxima, because the time spent finding the exact sorted order of non-maximal points may be wasted: in some sense, we are learning much more information about the input P than necessary. To deduce the list of maxima, we do not need the sorted order of *all* points of P: it suffices to know the sorted order of just the maxima! An optimal algorithm would probably locate the maximal points in $\mathbf{S}$ and would not bother locating "extremely non-maximal" points. This is, in some sense, the difficulty that output-sensitive algorithms face.

As a thought experiment, let us suppose that the maximal points of P are known to us, but not in sorted order. We search only for these in $\mathbf{S}$ and determine the sorted list of maximal points. We can argue that the optimal algorithm must also (in essence) perform such a search. We also need to find per-point certificates for the non-maximal points. We use the slab structure $\mathbf{S}$ and the search trees, but now we shall be very conservative in our searches. Consider the search for a point p_i. At any intermediate stage of the search, p_i is placed in a slab S. This rough knowledge of p_i's location may already suffice 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_i. If so, we have a per-point certificate for p_i and we promptly terminate the search for p_i. Otherwise, we continue the search by a single step and repeat. We expect that many searches will not proceed too long, achieving a better position to compete with the optimal algorithm.

Non-maximal points that are dominated by many maximal points will usually have a very short search. Points that are "nearly" maximal will require a much longer search. So this approach should derive just the "right" amount of information to determine the maxima output. But wait! Didn't we assume that the maximal points were known? Wasn't this crucial in cutting down the search time? This is too much of an assumption, and because the maxima are highly dependent on each other, it is not clear how to determine which points are maximal before performing 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 for sorted points. 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 some node of its search tree. We choose a specific point and advance its search by one step. This order is very important, and is the basis of our optimality. The algorithm is described in detail and analyzed in §4.

Arguing about optimality. A major challenge of self-improving algorithms is the strong requirement of optimality for the distribution $\mathcal{D}$. We focus on the model of linear comparison trees, and let $\mathcal{T}$ be an optimal tree for distribution $\mathcal{D}$. (There may be distributions where such an exact $\mathcal{T}$ does not exist, but we can always find one that is near optimal.) One of our key insights is that when $\mathcal{D}$ is a product distribution, then we can convert $\mathcal{T}$ to $\mathcal{T}'$, a restricted comparison tree whose expected depth is only a constant factor worse. In other words, there exists a near optimal restricted comparison tree that computes the maxima.

In such a tree, a leaf is labeled with a sequence of regions $\mathcal{R} = (R_1, R_2, \ldots, R_n)$. Any input $P = (p_1, p_2, \ldots, p_n)$ such that $p_i \in R_i$ for all i, will lead to this leaf. Since the distributions are independent, we can argue that the probability that an input leads to this leaf is $\prod_i \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R_i]$. Furthermore, the depth of this leaf can be shown to be $-\sum_i \log \Pr[p_i \in R_i]$. This gives us a concrete bound that we can exploit.

It now remains to show that if we start with a random input from $\mathcal{R}$, the expected running time is bounded by the sum given above. We will argue that for such an input, as soon as the search for p_i locates it inside R_i, the search will terminate. This leads to the optimal running time.

3. THE COMPUTATIONAL MODEL AND LOWER BOUNDS

3.1 Reducing to restricted comparison trees

We prove that when P is generated probabilistically, it suffices to focus on restricted comparison trees. To show this, we provide a sequence of transformations, starting from the more general comparison tree, that results in a restricted linear comparison tree of comparable expected depth. The main lemma of this section is the following.

LEMMA 3.1. *Let $\mathcal{T}$ a finite linear comparison tree and $\mathcal{D}$ be a product distribution over points. Then there exists a restricted comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{D}}(\mathcal{T}') = O(d_{\mathcal{D}}(\mathcal{T}))$, as $d_{\mathcal{D}}(\mathcal{T}) \to \infty$.*

We will describe a transformation from $\mathcal{T}$ into a restricted comparison tree with similar depth. The first step is to show how to represent a single comparison by a restricted linear comparison tree, provided that P is drawn from a product distribution. The final transformation basically replaces each node of $\mathcal{T}$ by the subtree given by the next claim. For convenience, we will drop the subscript of $\mathcal{D}$ from $d_{\mathcal{D}}$, since we only focus on a fixed distribution.

CLAIM 3.2. *Consider a comparison C as described in Definition 1.2, where the comparisons are listed in increasing order of simplicity. Let $\mathcal{D}'$ be a product distribution for P such that each p_i is drawn from a polygonal region R_i. Then either C is the simplest, type (i) comparison, or there exists a restricted linear comparison tree $\mathcal{T}'_C$ that resolves the comparison C such that the expected depth of $\mathcal{T}'_C$ (over the distribution $\mathcal{D}'$) is $O(1)$, and all comparisons used in $\mathcal{T}'_C$ are simpler than C.*

PROOF. **v is of type (ii).** This means that v needs to determine whether an input point p_i lies to the left of the directed line ℓ through another input point p_j with a fixed slope a. We replace this comparison with a binary search. Let R_j be the region in $\mathcal{D}'$ corresponding to p_j. Take a halving line ℓ_1 for R_j with slope a. Then perform two comparisons to determine on which side of ℓ_1 the inputs p_i and p_j lie. If p_i and p_j lie on different sides of ℓ_1, we declare success and resolve the original comparison accordingly. Otherwise, we replace R_j with the appropriate new region and repeat the process until we can declare success. Note that in each attempt the success probability is at least $1/4$. The resulting restricted tree $\mathcal{T}'_C$ can be infinite. Nonetheless, the probability that an evaluation of $\mathcal{T}'_C$ leads to a node of depth k is at most $2^{-\Omega(k)}$, so the expected depth is $O(1)$.

v is of type (iii). Here the node v needs to determine whether an input point p_i lies to the left of the directed line ℓ through another input point p_j and a fixed point q.

We partition the plane by a constant-sized family of cones, each 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 could 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. Such a construction has a non-zero probability of success, and therefore the described family of cones exists.

We build a restricted tree that locates a point in the corresponding cone. For each cone V, we can recursively build such a family of cones (inside V), and build a tree for this structure as well. Repeating for each cone, this leads to an infinite restricted tree $\mathcal{T}'_C$. We search for both p_i and p_j in $\mathcal{T}'_C$. When we locate p_i and p_j in two different cones of the same family, then comparison between p_i and $\overline{qp_j}$ is resolved and the search terminates. The probability that they lie in the same cones of a given family is at most $1/2$, so the probability that the evaluation leads to k steps is at most $2^{-\Omega(k)}$.

v is of type (iv). Here the node v needs to determine whether an input point p_i lies to the left of the directed line ℓ through input points 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 through p_j and p_k meets V is at most $1/2$. Such a family could be constructed by taking a sufficiently large random sample of pairs p_j and p_k and triangulating the arrangement of the lines through each pair. Such a construction has a non-zero probability of success, and therefore such a family exists. (Other than the source of the random lines used in the construction, this scheme goes back at least to [Cla87]; a tighter version, called a *cutting*, could also be used [Cha93].)

When computing C, suppose p_i is in region V of the family. If the line $\overline{p_jp_k}$ does not meet V, then the comparison outcome is known immediately. This occurs with probability at least $1/2$. Moreover, determining the region containing p_i can be done with a constant number of comparisons of type (i), and determining if $\overline{p_jp_k}$ meets V can be done with a constant number of comparisons of type (iii); for the latter, suppose V is a triangle. If $p_j \in V$, then $\overline{p_jp_k}$ meets V. Otherwise, suppose p_k is above all the 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 the lines through p_j and each vertex, then $\overline{p_jp_k}$ does not meet V. Otherwise, $\overline{p_jp_k}$ meets V. So a constant number of type (i) and type (iii) queries suffice.

By recursively building a tree for each region V of the family, comparisons of type (iv) can be done via a tree whose nodes use comparisons of type (i) and (iii) only. Since the probability of resolving the comparison is at least $1/2$ with each family of regions that is visited, the expected number of nodes visited is constant. □

PROOF OF LEMMA 3.1. We transform $\mathcal{T}$ into a tree $\mathcal{T}'$ that has no comparisons of type (iv), by using the construction of Claim 3.2 where nodes of type (iv) are replaced by a tree. We then transform $\mathcal{T}'$ into a tree $\mathcal{T}''$ that has no comparisons of type (iii) or (iv), and finally transform $\mathcal{T}'''$ into a restricted tree. Each such transformation is done in the same general way, using one case of Claim 3.2, so we focus on the first one.

We incrementally transform $\mathcal{T}$ into the tree $\mathcal{T}'$. In each such step, we have a partial restricted comparison tree $\mathcal{T}''$ that will eventually become $\mathcal{T}'$. Furthermore, during the process each node of $\mathcal{T}$ is in one of three different states. It is either *finished*, *fringe*, or *untouched*. Finally, we have a function S that assigns to each finished and to each 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. Furthermore, the partial tree $\mathcal{T}''$ consists of a single root node r and the function S assigns the root of $\mathcal{T}$ to the set $\{r\}$.

Now our transformation proceeds as follows. We pick a fringe node v in $\mathcal{T}$, and mark v 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. Next, we apply Claim 3.2 to each node $w \in S(v)$. Note that 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. Hence, replace each node $w \in S(v)$ in $\mathcal{T}''$ by the subtree given by Claim 3.2. Now $S(v)$ contains the roots of these subtrees. Each leaf of each such subtree corresponds to an outcome of the comparison in v. (Potentially, the subtrees are countably infinite, but the expected number of steps to reach a leaf is constant.) 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. The variable 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 to an 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 3.3 below shows that $\mathbf{E}[Y_v] = O(\mathbf{E}[X_v])$, which completes the proof. $\square$

Claim 3.3. $\mathbf{E}[Y_v] \le c\mathbf{E}[X_v]$

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 $\mathbf{E}[Y_v \mid X_v = 0] = 0$ and also $\Pr[P \in \mathcal{R}_v] = \sum_{w\in S(v)} \Pr[P \in \mathcal{R}_w]$. Furthermore, by Claim 3.2, we have $\mathbf{E}[Y_v \mid P \in \mathcal{R}_w] \le c$. The claim follows. $\square$

3.2 Entropy-sensitive comparison trees

Since every linear comparison tree can be made restricted, we can incorporate the entropy of $\mathcal{D}$ into the lower bound. For this we define entropy-sensitive trees, which are useful because the depth of a node v is related to the probability of the corresponding region $\mathcal{R}_v$.

Definition 3.4. *We call a restricted linear comparison tree* entropy-sensitive *if each comparison "$p_i \in \ell^+$?" is such that ℓ is a halving line for the current region R_i.*

Lemma 3.5. *Let v be a node in an entropy-sensitive comparison tree, and let $\mathcal{R}_v = R_1 \times R_2 \times \cdots \times R_n$. Then $d_v = -\sum_{i=1}^n \log \Pr[R_i]$.*

Proof. We use induction on the depth of v. For the root r we have $d_r = 0$. Now, let v' be the parent of v. Since $\mathcal{T}$ is entropy-sensitive, we reach v after performing a comparison with a halving line in v'. This halves the measure of exactly one region in $\mathcal{R}_v$, so the sum increases by one. $\square$

As in Lemma 3.1, we can make every restricted linear comparison tree entropy-sensitive without affecting its expected depth too much.

Lemma 3.6. *Let $\mathcal{T}$ a restricted linear comparison tree. Then there exists an entropy-sensitive comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{T}'} = O(d_{\mathcal{T}})$.*

Proof. The proof extends the proof of Lemma 3.1, via an extension to Claim 3.2. We can regard a comparison against a fixed halving line as simpler than an comparison against an arbitrary fixed line. Our extension of Claim 3.2 is the claim that any type (i) node can be replaced by a tree with constant expected depth, as follows. A comparison $p_i \in \ell^+$ can be replaced by a sequence of comparisons to halving lines. Similar to the reduction for type (ii) comparisons in Claim 3.2, this is done by binary search. That is, let ℓ_1 be a halving line for R_i parallel to ℓ. We compare p_i with ℓ. If this resolves the original comparison, we declare success. Otherwise, we repeat the process with the halving line for the new region R_i'. In each step, the probability of success is at least $1/2$. The resulting comparison tree has constant expected depth; we now apply the construction of Lemma 3.1 to argue that for a restricted tree $\mathcal{T}$ there is an entropy-sensitive version $\mathcal{T}'$ whose expected depth is larger by at most a constant factor. $\square$

Recall that $\mathrm{OPT}_{\mathcal{D}}$ is the expected depth of an optimal linear comparison tree that computes the maxima for $P \sim \mathcal{D}$. We now describe how to characterize $\mathrm{OPT}_{\mathcal{D}}$ in terms of entropy-sensitive comparison trees. We first state a simple property that follows directly from the definition of certificates and the properties of restricted comparison trees.

Proposition 3.7. *Consider a leaf v of a restricted linear comparison tree $\mathcal{T}$ computing the maxima. Let R_i be the region associated with non-maximal point $p_i \in P$ in $\mathcal{R}_v$. There exists some region R_j associated with an extremal point p_j such that every point in R_j dominates every point in R_i.*

We now enhance the notion of a certificate (Definition 1.1) to make it more useful for our algorithm's analysis. For technical reasons, we want points to be "well-separated" according to the slab structure $\mathbf{S}$. By Prop. 3.7, every non-maximal point is associated with a dominating region.

Definition 3.8. *Let $\mathbf{S}$ be a slab structure. A certificate for an input P is called* $\mathbf{S}$-labeled *if the following holds. Every maximal point is labeled with the leaf slab of $\mathbf{S}$ containing it. Every non-maximal point is either placed in the containing leaf slab, or is separated from a dominating region by a slab boundary.*

We naturally extend this to trees that compute the **S**-labeled maxima.

DEFINITION 3.9. *A linear comparison tree $\mathcal{T}$ computes the $\boldsymbol{S}$-labeled maxima of P if each leaf v of $\mathcal{T}$ is labeled with a $\boldsymbol{S}$-labeled certificate that is valid for every possible input $P \in \mathcal{R}_v$.*

LEMMA 3.10. *There exists an entropy-sensitive comparison tree $\mathcal{T}$ computing the $\boldsymbol{S}$-labeled maxima whose expected depth over $\mathcal{D}$ is $O(n + \mathrm{OPT}_{\mathcal{D}})$.*

PROOF. Start with an optimal linear comparison tree $\mathcal{T}'$ that computes the maxima. At every leaf, we have a list M with the maximal points in sorted order. We merge M with the list of slab boundaries of **S** to label each maximal point with the leaf slab of **S** containing it. We now deal with the non-maximal points. Let R_i be the region associated with a non-maximal point p_i, and R_j be the dominating region. Let λ be the leaf slab containing R_j. Note that the x-projection of R_i cannot extended to the right of λ. If there is no slab boundary separating R_i from R_j, then R_i must intersect λ. With one more comparison, we can place p_i inside λ or strictly to the left of it. All in all, with $O(n)$ more comparisons than $\mathcal{T}'$, we have a tree $\mathcal{T}''$ that computes the **S**-labeled maxima. Hence, the expected depth is $\mathrm{OPT}_{\mathcal{D}} + O(n)$. Now we apply Lemmas 3.1 and 3.6 to $\mathcal{T}''$ to get an entropy-sensitive comparison tree $\mathcal{T}$ computing the **S**-labeled maxima with expected depth $O(n + \mathrm{OPT}_{\mathcal{D}})$. ☐

4. THE ALGORITHM

In the learning phase, the algorithm constructs a slab structure **S** and search trees T_i, as given in Lemmas 1.7 and 1.8. Henceforth, we assume that we have these data structures, and will describe the algorithm in the limiting (or stationary) phase. Our algorithm proceeds by searching progressively each point p_i in its tree T_i. However, we need to choose the order of 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*. An inactive point is either proven to be non-maximal, or it has been placed in a leaf slab. The active points are stored in a data structure $L(A)$. This structure is similar to a heap and supports the operations *delete*, *decrease-key*, and *find-max*. The key associated with 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 that the algorithm maintains. The algorithm is initialized with $A = P$, and each S_i is the largest slab in **S**. Hence, all points have key $|\mathbf{S}|$, and we insert all these keys into $L(A)$.

- $A, L(A)$: the list A of active points stored in data structure $L(A)$.
- $\widehat{\lambda}, B$: Let m be the largest key among the active points. Then $\widehat{\lambda}$ is the leaf slab whose right boundary is m and B is a set of points located in $\widehat{\lambda}$. Initially B is empty and m is $|S|$, corresponding to the $+\infty$ boundary of the rightmost, infinite, slab.
- $M, \hat{p}$: M is a sorted (partial) list of currently discovered maximal points 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 proceeds its search by a single step in the appropriate tree. Occasionally, it will invoke **Update** to change the global variables. The algorithm repeatedly calls **Search** until $L(A)$ is empty. After that, we perform a final call to **Update** in order to process any points that might still remain in B.

Search. Let p_i be obtained by performing a *find-max* in $L(A)$. If the maximum key m in $L(A)$ is less than the right boundary of $\widehat{\lambda}$, we invoke **Update**. If p_i is dominated by $\hat{p}$, we delete p_i from $L(A)$. If not, we advance the search of p_i in T_i by a single step, if possible. This updates the slab S_i. If the right boundary of S_i has decreased, we perform the appropriate *decrease-key* operation on $L(A)$. (Otherwise, we do nothing.)

Suppose the point p_i reaches a leaf slab λ. If $\lambda = \widehat{\lambda}$, we remove p_i from $L(A)$ and insert it in B (in time $O(|B|)$). Otherwise, we leave p_i in $L(A)$.

Update. We sort all the points in B and update the list of current maxima. As Claim 4.1 will show, we have 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.

We prove some preliminary claims. We state an important invariant maintained by the algorithm, and then give a construction for the data structure $L(A)$.

CLAIM 4.1. *At any time in the algorithm, the maxima of all points to the right of $\widehat{\lambda}$ have been determined in sorted order.*

PROOF. The proof is by backward induction on m, the right boundary of $\widehat{\lambda}$. When $m = |S|$, then this is trivially true. Let us assume it is true for a given value of m, and trace the algorithm's behavior until the maximum key becomes smaller than m (which is done in **Update**). When **Search** processes a point p with a key of m then either (i) the key value decreases; (ii) p is dominated by $\hat{p}$; or (iii) p is eventually 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** will sort the points in B and all maximal points to the right of $m-1$ will be determined. ☐

CLAIM 4.2. *Suppose there are x* find-max *operations and y* decrease-key *operations. We can implement the data structure $L(A)$ such that the total time for the operations is $O(n+x+y)$. The storage requirement is $O(n)$.*

PROOF. We represent $L(A)$ as an array of lists. For every $k \in [|\mathbf{S}|]$, we keep a list of points whose key values are k. We maintain m, the current maximum key. The total storage is $O(n)$. A *find-max* can trivially be done in $O(1)$ time, and an *insert* is done by adding the element to the appropriate list. A *delete* is done by deleting the element from the list (supposing appropriate pointers are available). We now have to update the maximum. If the list at m is non-empty, no action is required. If it is empty, we check sequentially whether the list at $m-1, m-2, \ldots$ is empty. This will eventually lead to the maximum. To do a *decrease-key*, we *delete*, *insert*, and then update the maximum.

Note that since all key updates are *decrease-keys*, the maximum can only decrease. Hence, the total overhead for scanning for a new maximum is $O(n)$. $\square$

4.1 Running time analysis

The aim of this section is to prove the following lemma.

LEMMA 4.3. *The algorithm runs in $O(n + \mathrm{OPT}_{\mathcal{D}})$ time.*

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

CLAIM 4.4. *The expected time for all calls to* **Update** *is $O(n)$.*

PROOF. The total time taken for all calls to **Update** is at most the time taken to sort points within leaf slabs. By Lemma 1.7, this takes expected time

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

$\square$

The important claim is the following, since it allows us to relate the time spent by **Search** to the entropy-sensitive comparison trees. Lemma 4.3 follows directly from this.

CLAIM 4.5. *Let $\mathcal{T}$ be an entropy-sensitive comparison tree computing* **S***-labeled maxima. Consider a leaf v labeled with the regions $\mathcal{R}_v = (R_1, R_2, \ldots, R_n)$, and let d_v denote the depth of v. 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 **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 (this is 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, we argue that an S_i-restricted search suffices to process p_i. This follows from Claim 4.1: by the time an S_i-restricted search finishes, all maxima to the right of S_i have been determined. In particular, we have found p_j, and thus $\hat{p}$ dominates p_i. Hence, the search for p_i will proceed 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 1.6, 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 Lemma 3.5, $d_v = \sum_i -\log \Pr[p_i \in R_i]$, completing the proof. $\square$

We can now prove the main lemma.

PROOF OF LEMMA 4.3. By Lemma 3.10, there exists an entropy-sensitive comparison tree $\mathcal{T}$ that computes the **S**-labeled maxima with expected depth $O(\mathrm{OPT} + n)$. According to Claim 4.5, the expected running time of **Search** is $O(\mathrm{OPT} + n)$. Claim 4.4 tells us the expected time for **Update** is $O(n)$, and we add these bounds to complete the proof. $\square$

5. DATA STRUCTURES OBTAINED DURING THE LEARNING PHASE

Learning the vertical slab structure **S** is very similar to to learning the V-list in Ailon et al. [ACC+11, Lemma 3.2]. We repeat the construction and proof for convenience: take the union of the first $k = \log n$ inputs $P_1, P_2, \ldots, P_k$, and sort those points by x-coordinates. This gives a list $x_0, x_1, \ldots, x_{nk-1}$. Take the n values $x_0, x_k, x_{2k}, \ldots, x_{(n-1)k}$. They define the boundaries for **S**. We recall a useful and well-known fact [ACC+11, Claim 3.3].

CLAIM 5.1. *Let $Z = \sum_i Z_i$ be a sum of nonnegative random variables such that $Z_i = O(1)$ for all i, $\mathbf{E}[Z] = O(1)$, and for all i, j, $\mathbf{E}[Z_i Z_j] = \mathbf{E}[Z_i]\mathbf{E}[Z_j]$. Then $\mathbf{E}[Z^2] = O(1)$.*

Now let λ be a leaf slab in **S**. Recall that we denote by X_λ the number of points of a random input P that end up in λ. Using Claim 5.1, we quickly obtain the following lemma.

LEMMA 5.2. *With probability $1 - n^{-3}$ over the construction of $\boldsymbol{S}$, we have $\mathbf{E}[X_\lambda^2] = O(1)$ for all leaf slabs $\lambda \in \boldsymbol{S}$.*

PROOF. Consider two values x_i, x_j from the original list. Note that all the other $kn - 2$ values are independent of these two points. For every $r \notin \{i, j\}$, let $Y_t^{(r)}$ be the indicator random variable for $x_r \in t := [x_i, x_j)$. Let $Y_t = \sum_r Y_t^{(r)}$. Since the $Y_t^{(r)}$'s are independent, by Chernoff's bound [AS00], for any $\beta \in (0, 1]$,

$$\Pr[Y_t \leq (1 - \beta)\mathbf{E}[Y_t]] \leq \exp(-\beta^2 \mathbf{E}[Y_t]/2).$$

With probability at least $1 - n^{-5}$, if $\mathbf{E}[Y_t] > 12 \log n$, then $Y_t > \log n$. By applying the same argument for any pair x_i, x_j and taking a union bound over all pairs, with probability at least $1 - n^{-3}$ the following holds: for any pair t, if $Y_t \leq \log n$, then $\mathbf{E}[Y_t] \leq 12 \log n$.

For any leaf slab $\lambda = [x_{ak}, x_{(a+1)k}]$, we have $Y_\lambda \leq \log n$. Let $X_\lambda^{(i)}$ be the indicator random variable for the event that $x_i \sim \mathcal{D}_i$ lies in λ, so that $X_\lambda = \sum_i X_\lambda^{(i)}$. Since $\mathbf{E}[Y_\lambda] \geq (\log n - 2)\mathbf{E}[X_\lambda]$, we get $\mathbf{E}[X_\lambda] = O(1)$. By independence of the $\mathcal{D}_i$'s, for all i, j, $\mathbf{E}\big[X_\lambda^{(i)} X_\lambda^{(j)}\big] = \mathbf{E}\big[X_\lambda^{(i)}\big]\mathbf{E}\big[X_\lambda^{(j)}\big]$, so $\mathbf{E}[X_\lambda^2] = O(1)$, by Claim 5.1. $\square$

Lemma 1.7 follows immediately from Lemma 5.2 and the fact that sorting the k inputs $P_1, P_2, \ldots, P_k$ takes $O(n \log^2 n)$ time. After the leaf slabs have been determined, the search trees T_i can be found using essentially the same techniques as before [ACC+11, Section 3.2]. The main idea is to use $n^\varepsilon \log n$ rounds to find the first $\varepsilon \log n$ levels of T_i, and to use a balanced search tree for searches that need to proceed to a deeper level. This only costs a factor of ε^{-1}. We restate Lemma 1.8 for convenience.

LEMMA 5.3. *Let $\varepsilon > 0$ be a fixed parameter. In $O(n^\varepsilon)$ rounds and $O(n^{1+\varepsilon})$ time, we can construct search trees T_1, T_2, ..., T_n over* $\boldsymbol{S}$ *such that the following holds. (i) the trees can be totally represented in $O(n^{1+\varepsilon})$ space; (ii) probability $1-n^{-3}$ over the construction of the T_is: every T_i is $O(1/\varepsilon)$-optimal for restricted searches over $\mathcal{D}_i$.*

PROOF. Let $\delta > 0$ be some sufficiently small constant and c be sufficiently large . For $k = c\delta^{-2}n^\varepsilon \log n$ rounds and each p_i, we record the leaf slab of $\mathbf{S}$ that contains it. We break the proof into smaller claims.

CLAIM 5.4. *Using k inputs, we can compute estimates $\hat{q}(i,S)$ for each index i and slab S. The following guarantee holds (for all i and S) with probability $> 1-1/n^3$ over the choice of the k inputs. If at least $5\log n$ instances of p_i fell in S, then $\hat{q}(i,S) \in [(1-\delta)q(i,S),(1+\delta)q(i,S)]$*[1].

PROOF. For a slab S, let $N(S)$ be the number of times p_i was in S, and let $\hat{q}(i,S) = N(S)/k$ be the empirical probability for this event ($\hat{q}(i,S)$ is an estimate of $q(i,S)$). Fix a slab S. If $q(i,S) \leq 1/2n^\varepsilon$, then by a Chernoff bound we get $\Pr[N(S) \geq 5\log n \geq 10kq(i,S)] \leq 2^{-5\log n} = n^{-5}$. Furthermore, if $q(i,S) \geq 1/2n^\varepsilon$, then $q(i,S)k \geq (c/2\delta^2)\log n$ and $\Pr[N(S) \leq (1-\delta)q(i,S)k] \leq \exp(-q(i,S)\delta^2 k/4) \leq n^{-5}$ as well as $\Pr[N(S) \geq (1+\delta)q(i,S)k] \leq \exp(-\delta^2 q(i,S)k/4) \leq n^{-5}$. Thus, by taking a union bound, we get that with probability at least $1-n^{-3}$ for any slab S, if $N(S) \geq 5\log n$, then $q(i,S) \geq n^{-\varepsilon}/2$ and hence $\hat{q}(i,S) \in [(1-\delta)q(i,S),(1+\delta)q(i,S)]$. □

We will henceforth assume that this claims holds for all i and S. Based on the values $\hat{q}(i,S)$, we construct the search trees. The tree T_i is constructed recursively. We will first create a partial search tree, where some searches may end in non-leaf slabs (or, in other words, leaves of the tree may not be leaf slabs). The root is the just the largest slab. Given a slab S, we describe how the create the sub-tree of T_i rooted at S. If $N(S) < 5\log n$, then we make S a leaf. Otherwise, we pick a leaf slab λ such that for the slab S_l consisting of all leaf slabs (strictly) to the left of λ and the slab S_r consisting of all leaf slabs (strictly) to the right of λ we 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. Then we recursively create trees for S_l and S_r and attach them as children to S. For any internal node of the tree S, we have $q(i,S) \geq n^\varepsilon/2$, and hence the depth is at most $O(\varepsilon \log n)$. Furthermore, this partial tree is β-reducing (for some constant β). The partial tree T_i is extended to a complete tree in a simple way. From each T_i-leaf that is not a leaf slab, we perform a basic binary search for the leaf slab. This yields a tree T_i of depth at most $(1+O(\varepsilon))\log n$. Note that we only need to store the partial T_i tree, and hence the total space is $O(n^{1+\varepsilon})$.

Let us construct, as a thought experiment, a related tree T'_i. Start with the partial T_i. For every leaf that is not a leaf slab, extend it downward using the true probabilities $q(i,S)$. In other words, let us construct the subtree rooted at a new node S in the following manner. We pick a leaf slab λ such that $q(i,S_l) \leq (2/3)q(i,S)$ and $q(i,S_r) \leq (2/3)q(i,S)$ (where S_l and S_r are as defined above). This ensures that T'_i is β-reducing. By Lemma 1.6, T'_i is $O(1)$-optimal for restricted searches over $\mathcal{D}_i$ (we absorb the β into $O(1)$ for convenience).

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

CLAIM 5.5. *The tree T_i is $O(1/\varepsilon)$-optimal for restricted searches.*

PROOF. Fix a slab S and an S-restricted distribution $\mathcal{D}_S$. Let $q'(i,\lambda)$ (for each leaf slab λ) be the series of values defining $\mathcal{D}_S$. Note that $q'(i,S) \leq q(i,S)$. Suppose $q'(i,S) \leq n^{-\varepsilon/2}$. Then $-\log q'(i,S) \geq \varepsilon(\log n)/2$. Since any search in T_i takes at most $(1+O(\varepsilon))\log n$ steps, the search time is at most $O(\varepsilon^{-1}(-\log q'(i,S)+1))$.

Suppose $q'(i,S) > n^{-\varepsilon/2}$. Consider a single search for some p_i. We will classify this search based on the leaf of the partial tree that is encountered. By the construction of T_i, any leaf S' is either a leaf slab or has the property that $q(i,S') \leq n^{-\varepsilon}/2$. The search is of *Type 1* if the leaf of the partial tree actually 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 a slab S is an internal node of T_i and the depth is at least (resp. less than) $\varepsilon(\log n)/3$.

When the search is of Type 1, it is identical in both T_i and T'_i. When the search is of Type 2, it takes at $\varepsilon(\log n)/3$ in T'_i and at most (trivially) $(1+O(\varepsilon))(\log n)$ in T_i. 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 $\mathcal{D}_i$ inside such leaves is at most $n^{\varepsilon/3} \times n^{-\varepsilon}/2 < n^{-2\varepsilon/3}$. Since $q'(i,S) > n^{-\varepsilon/2}$, in the restricted distribution $\mathcal{D}_S$, the probability of a Type 3 search is at most $n^{-\varepsilon/6}$.

Choose a random $p \sim \mathcal{D}_S$. Let $\mathcal{E}$ denote the event that a Type 3 search occurs. Furthermore, let X_p denote the depth of the search in T_i and X'_p denote the depth in T'_i. When $\mathcal{E}$ does not occur, we have argued that $X_p \leq O(X'_p/\varepsilon)$. Also, $\Pr(\mathcal{E}) \leq n^{-\varepsilon/6}$. The expected search time is just $\mathbf{E}[X_p]$. By Bayes' rule,

$$\begin{aligned}
\mathbf{E}[X_p] &= \Pr(\overline{\mathcal{E}})\mathbf{E}_{\overline{\mathcal{E}}}[X_p] + \Pr(\mathcal{E})\mathbf{E}_{\mathcal{E}}[X_p] \\
&\leq O(\varepsilon^{-1}\mathbf{E}_{\overline{\mathcal{E}}}[X'_p]) + n^{-\varepsilon/6}(1+O(\varepsilon))\log n \\
\mathbf{E}[X'_p] &= \Pr(\overline{\mathcal{E}})\mathbf{E}_{\overline{\mathcal{E}}}[X'_p] + \Pr(\mathcal{E})\mathbf{E}_{\mathcal{E}}[X_p] \\
\implies \quad & \mathbf{E}_{\overline{\mathcal{E}}}[X'_p] \leq \mathbf{E}[X'_p]/\Pr(\overline{\mathcal{E}}) \leq 2\mathbf{E}[X'_p]
\end{aligned}$$

Combining, the expected search time is $O(\varepsilon^{-1}(\mathbf{E}[X'_p]+1))$. Since T'_i is $O(1)$-optimal for restricted searches, T_i is $O(\varepsilon^{-1})$-optimal. □

6. ACKNOWLEDGMENTS

C. Seshadhri was funded 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.

We would like to thank Eden Chlamtáč for suggesting a simple proof for Claim 3.3.

7. REFERENCES

[ABC09] Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-optimal geometric algorithms. In *Proc. 50th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 129–138, 2009.

[ACC$^+$11] Nir Ailon, Bernard Chazelle, Kenneth L. Clarkson, Ding Liu, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms. *SIAM Journal on Computing*, 40(2):350–375, 2011.

[ACCL06] Nir Ailon, Bernard Chazelle, Seshadhri Comandur, and Ding Liu. Self-improving algorithms. In *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 261–270, 2006.

[AS00] Noga Alon and Joel H. Spencer. *The probabilistic method*. Wiley-Interscience, New York, second edition, 2000.

[BKS01] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[BLMM11] Kevin Buchin, Maarten Löffler, Pat Morin, and Wolfgang Mulzer. Preprocessing imprecise points for Delaunay triangulations: Simplified and extended. *Algorithmica*, 61(3):674–693, 2011.

[Buc89] C. Buchta. On the average number of maxima in a set of vectors. *Inform. Process. Lett.*, 33(2):63–66, 1989.

[Cha93] Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9:145–158, 1993. 10.1007/BF02189314.

[Cla87] Kenneth L. Clarkson. New applications of random sampling to computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[CMS10] K. Clarkson, W. Mulzer, and C. Seshadhri. Self-improving algorithms for convex hulls. In *Proc. 21st Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, 2010.

[CS08] Kenneth L. Clarkson and C. Seshadhri. Self-improving algorithms for Delaunay triangulations. In *Proc. 24th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 226–232, 2008.

[EM11] Esther Ezra and Wolfgang Mulzer. Convex hull of imprecise points in $o(nlogn)$ time after preprocessing. In *Proc. 27th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 11–20, 2011.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM.

[Gol94] M. J. Golin. A provably fast linear-expected-time maxima-finding algorithm. *Algorithmica*, 11:501–524, 1994. 10.1007/BF01189991.

[GTVV93] M. T. Goodrich, Jyh-Jong Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science*, pages 714–723, Washington, DC, USA, 1993. IEEE Computer Society.

[HM08] Martin Held and Joseph S. B. Mitchell. Triangulating input-constrained planar point sets. *Inform. Process. Lett.*, 109(1):54–56, 2008.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22:469–476, October 1975.

[vKLM10] Marc J. van Kreveld, Maarten Löffler, and Joseph S. B. Mitchell. Preprocessing imprecise points and splitting triangulations. *SIAM Journal on Computing*, 39(7):2990–3000, 2010.

[KS86] David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.

[LS10] Maarten Löffler and Jack Snoeyink. Delaunay triangulation of imprecise points in linear time after preprocessing. *Comput. Geom. Theory Appl.*, 43(3):234–242, 2010.