SIAM J. COMPUT.
Vol. 40, No. 2, pp. 350–375

SELF-IMPROVING ALGORITHMS*

NIR AILON†, BERNARD CHAZELLE‡, KENNETH L. CLARKSON§, DING LIU‡, WOLFGANG MULZER‡, AND C. SESHADHRI§

Abstract. We investigate ways in which an algorithm can improve its expected performance by fine-tuning itself automatically with respect to an *unknown* input distribution $\mathcal{D}$. We assume here that $\mathcal{D}$ is of *product type*. More precisely, suppose that we need to process a sequence $I_1, I_2, \ldots$ of inputs $I = (x_1, x_2, \ldots, x_n)$ of some fixed length n, where each x_i is drawn independently from some *arbitrary, unknown* distribution $\mathcal{D}_i$. The goal is to design an algorithm for these inputs so that eventually the expected running time will be optimal for the input distribution $\mathcal{D} = \prod_i \mathcal{D}_i$. We give such *self-improving* algorithms for two problems: (i) sorting a sequence of numbers and (ii) computing the Delaunay triangulation of a planar point set. Both algorithms achieve optimal expected limiting complexity. The algorithms begin with a training phase during which they collect information about the input distribution, followed by a stationary regime in which the algorithms settle to their optimized incarnations.

Key words. average case analysis, Delaunay triangulation, low entropy, sorting

AMS subject classifications. 68Q25, 68W20, 68W40

DOI. 10.1137/090766437

1. Introduction. The classical approach to analyzing algorithms draws a familiar litany of complaints: worst-case bounds are too pessimistic in practice, say the critics, while average-case complexity too often rests on unrealistic assumptions. The charges are not without merit. Hard as it is to argue that the only permutations we ever want to sort are random, it is a different level of implausibility altogether to pretend that the sites of a Voronoi diagram should always follow a Poisson process or that ray tracing in a binary space partitioning (BSP) tree should be spawned by a Gaussian. Efforts have been made to analyze algorithms under more complex models (e.g., Gaussian mixtures and Markov model outputs) but with limited success and lingering doubts about the choice of priors.

Suppose we wish to compute a function f that takes I as input. We get a sequence of inputs $I_1, I_2, \ldots$, and wish to compute $f(I_1)$, $f(I_2), \ldots$. It is quite plausible to assume that all these inputs are somehow related to each other. This relationship, though exploitable, may be very difficult to express concisely. One way of modeling this situation is to postulate a fixed (but complicated) unknown distribution $\mathcal{D}$ of inputs. Each input I_j is chosen independently at random from $\mathcal{D}$. (Refer to Figure 1.1.) Is it possible to quickly learn something about $\mathcal{D}$ so that we can compute $f(I)$ (I chosen from $\mathcal{D}$) faster? (Naturally, this is by no means the only possible input model. For example, we could have a memoryless Markov source, where each I_j depends only

*Received by the editors July 28, 2009; accepted for publication (in revised form) December 28, 2010; published electronically April 5, 2011. Preliminary versions appeared as N. Ailon, B. Chazelle, S. Comandur, and D. Liu, *Self-improving algorithms*, in Proceedings of the 17th SODA, 2006, pp. 261–270; and K. L. Clarkson and C. Seshadhri, *Self-improving algorithms for Delaunay triangulations*, in Proceedings of the 24th SoCG, 2008, pp. 148–155. This work was supported in part by NSF grants CCR-998817 and 0306283 and ARO grant DAAH04-96-1-0181.

http://www.siam.org/journals/sicomp/40-2/76643.html

†Computer Science Faculty, Technion, Haifa, Israel (nailon@google.com).

‡Department of Computer Science, Princeton University, Princeton, NJ (chazelle@cs.princeton.edu, dingliu@cs.princeton.edu, wmilzer@cs.princeton.edu).

§IBM Almaden Research Center, San Jose, CA (klclarks@us.ibm.com, csesha@gmail.com).

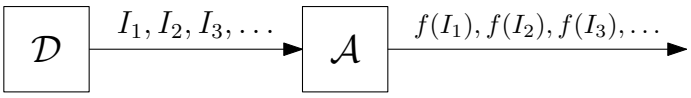

FIG. 1.1. *A self-improving algorithm $\mathcal{A}$ processes a sequence $I_1, I_2, \ldots$ of inputs drawn independently from a random source $\mathcal{D}$.*

on I_{j-1}. However, for simplicity we will here focus on a fixed source that generates the inputs independently.)

That is what a *self-improving algorithm* attempts to do. Initially, since nothing is known about $\mathcal{D}$, our self-improving algorithm can provide only a worst-case guarantee. As the algorithm sees more and more inputs, it can learn something about the structure of $\mathcal{D}$. We call this the *training phase* of the self-improving algorithm. During this phase, the algorithm collects and organizes information about the inputs in the hope that it can be used to improve the running time (with respect to inputs from $\mathcal{D}$). The algorithm then moves to the *limiting phase*. Having decided that enough has been learned about $\mathcal{D}$, the algorithm uses this information to compute $f(I)$ faster. Note that this behavior is tuned to the distribution $\mathcal{D}$.

Obviously, there is no reason why we should get a faster running time for all $\mathcal{D}$. Indeed, if f is the sorting function and $\mathcal{D}$ is the uniform distribution over permutations, then we require an expected $\Omega(n \log n)$ time to sort. On the other hand, if $\mathcal{D}$ was a low-entropy source of inputs, it is quite reasonable to hope for a faster algorithm. So when can we improve our running time? An elegant way of expressing this is to associate (using information theory) an "optimal" running time to each distribution. This is a sort of estimate of the best expected running time we can hope for, given inputs chosen from a fixed distribution $\mathcal{D}$. Naturally, the lower the entropy of $\mathcal{D}$, the lower this running time will be. In the limiting phase, our self-improving algorithm should achieve this optimal running time.

To expect a good self-improving algorithm that can handle *all* distributions $\mathcal{D}$ seems a bit ambitious, and indeed we show that even for the sorting problem there can be no such space-efficient algorithm (even when the entropy is low). Hence, it seems necessary to impose some kind of restriction on $\mathcal{D}$. However, if we required $\mathcal{D}$ to be, say, uniform or a Gaussian, we would again be stuck with the drawbacks of traditional average-case analysis. Hence, for self-improvement to be of any interest, the restricted class of distributions should still be fairly general. One such class is given by product distributions.

1.1. Model and results. We will focus our attention on distributions $\mathcal{D}$ of *product type*. Think of each input as an n-dimensional vector $(x_1, \ldots, x_n)$ over some appropriate domain. This could be a list of numbers (in the case of sorting) or a list of points (for Delaunay triangulations). Each x_i is generated independently at random from an arbitrary distribution $\mathcal{D}_i$, so $\mathcal{D} = \prod_i \mathcal{D}_i$. All the $\mathcal{D}_i$'s are independent of each other. It is fairly natural to think of various portions of the input as being generated by independent sources. For example, in computational geometry, the convex hull of uniformly independently distributed points in the unit square is a well-studied problem.

Note that all our inputs are of the same size n. This might appear to be a rather unnatural requirement for (say) a sorting algorithm. Why must the 10th number in our input come from the same distribution? We argue that this is not a major issue (for concreteness, let us focus on sorting). The right way to think of the input is as a

set of sources $\mathcal{D}_1, \mathcal{D}_2, \ldots$, each independently generating a single number. The actual "order" in which we get these numbers is not important. What *is* important is that for each number, we know its source. For a given input, it is realistic to suppose that some sources may be active, and some may not (so the input may have less than n numbers). Our self-improving sorters essentially perform an *independent* processing on each input number, after which $O(n)$ time is enough to sort.[1] The algorithm is completely unaffected by the inactive sources. To complete the training phase, we need to get only enough information about each source. What if new sources are introduced during the stationary phase? Note that as long as $O(n/\log n)$ new sources (and hence new numbers) are added, we can always include these extra numbers in the sorted list in $O(n)$ time. Once the number of new sources becomes too large, we will have to go back to the training phase. This is, of course, quite acceptable: if the underlying distribution of inputs changes significantly, we have to recalibrate the algorithm. For these reasons, we feel that it is no loss of generality to deal with a fixed input length, especially for product distributions.

Our first result is a self-improving sorter. Given a source $\mathcal{D} = \prod_i \mathcal{D}_i$ of real-number sequences $I = (x_1, \ldots, x_n)$, let $\pi(I)$ denote the permutation induced by the ranks of the x_i's, using the indices i to break ties. Observe that since I is a random variable, so is $\pi(I)$. We can define the entropy $H(\pi(I))$, over the randomness of $\mathcal{D}$, and the limiting complexity of our algorithm will depend on $H(\pi(I))$. Note this quantity may be much smaller than the entropy of the source itself but can never exceed it.

As we mentioned earlier, the self-improving algorithm initially undergoes a training phase. At the end of this phase, some data structures storing information about the distributions are constructed. In the limiting phase, the self-improving algorithm is fixed, and these data structures do not change. In the context of sorting, the self-improving sorter becomes some fixed comparison tree.

THEOREM 1.1. *There exists a self-improving sorter of $O(n + H(\pi(I)))$ limiting complexity for any input distribution $\mathcal{D} = \prod_i \mathcal{D}_i$. Its worst-case running time is $O(n \log n)$. No comparison-based algorithm can sort an input from $\mathcal{D}$ in less than $H(\pi(I))$ time. For any constant $\varepsilon > 0$, the storage can be made $O(n^{1+\varepsilon})$ for an expected running time of $O(\varepsilon^{-1}(n+H(\pi(I))))$. The training phase lasts $O(n^{\varepsilon})$ rounds, and the probability that it fails is at most $1/n$.*

Why do we need a restriction on the input distribution? In section 3.3, we show that a self-improving sorter that can handle *any* distribution requires an exponentially large data structure. Fredman [31] gave an algorithm that could optimally sort permutations from *any* distribution $\mathcal{D}$. His algorithm needs to know $\mathcal{D}$ explicitly, and it constructs lookup tables of exponential size. Our bound shows that Fredman's algorithm cannot be improved. Furthermore, we show that even for product distributions any self-improving sorter needs superlinear space. Hence, our time-space tradeoffs are essentially optimal. We remind the reader that we focus on comparison-based algorithms.

THEOREM 1.2. *Consider a self-improving algorithm that, given any fixed distribution $\mathcal{D}$, can sort a random input from $\mathcal{D}$ in an expected $O(n + H(\pi(I)))$ time. Such an algorithm requires $2^{\Omega(n \log n)}$ bits of storage.*

Let $\varepsilon \in (0, 1)$. Consider a self-improving algorithm that, given any product distribution *$\mathcal{D} = \prod_i \mathcal{D}_i$, can sort a random input from $\mathcal{D}$ in an expected $\varepsilon^{-1}(n + H(\pi(I)))$ time. Such an algorithm requires a data structure of bit size $n^{1+\Omega(\varepsilon)}$.*

[1] The self-improving Delaunay triangulation algorithms behave similarly.

For our second result, we take the notion of self-improving algorithms to the geometric realm and address the classical problem of computing the Delaunay triangulation of a set of points in the Euclidean plane. Given a source $\mathcal{D} = \prod_i \mathcal{D}_i$ of sequences $I = (x_1, \ldots, x_n)$ of points in R^2, let $T(I)$ denote the Delaunay triangulation of I. If we interpret $T(I)$ as a random variable on the set of all undirected graphs with vertex set $\{1, \ldots, n\}$, then $T(I)$ has an entropy $H(T(I))$, and the limiting complexity of our algorithm depends on this entropy.

THEOREM 1.3. *There exists a self-improving algorithm for planar Delaunay triangulations of $O(n + H(T(I)))$ limiting complexity for any input distribution $\mathcal{D} = \prod \mathcal{D}_i$. Its worst-case running time is $O(n \log n)$. For any constant $\varepsilon > 0$, the storage can be made $O(n^{1+\varepsilon})$ for an expected running time of $O(\varepsilon^{-1}(n + H(T(I))))$. The training phase lasts $O(n^{\varepsilon})$ rounds, and the probability that it fails is at most $1/n$.*

From the linear time reduction from sorting to computing Delaunay triangulations [14, Theorems 8.2.2 and 12.1.1], the lower bounds of Theorem 1.2 carry over to Delaunay triangulations.

Both our algorithms follow the same basic strategy. During the training phase, we collect data about the inputs in order to obtain a *typical* input instance V for $\mathcal{D}$ with $|V| = O(n)$, and we compute the desired structure S (a sorted list or a Delaunay triangulation) on V. Then for each distribution $\mathcal{D}_i$, we construct an entropy optimal search structure D_i for S (i.e., an entropy optimal binary search tree or a distribution sensitive planar point location structure). In the limiting phase, we use the D_i's in order to locate the components of a given input I in S. The fact that V is a typical input ensures that I will be broken into individual subproblems of expected *constant* size that can be solved separately, so we can obtain the desired structure for the input $V \cup I$ in expected linear time (plus the time for the D_i-searches). Finally, for both sorting and Delaunay triangulation it suffices to know the solution for $V \cup I$ in order to derive the solution for I in linear expected time [21, 22]. Thus, the running time of our algorithms is dominated by the D_i-searches, and the heart of the analysis lies in relating this search time to the entropies $H(\pi(I))$ and $H(T(I))$, respectively.

1.2. Previous work. Concepts related to self-improving algorithms have been studied before. List accessing algorithms and splay trees are textbook examples of how simple updating rules can speed up searching with respect to an adversarial request sequence [5, 15, 35, 45, 46]. It is interesting to note that self-organizing data structures were investigated over stochastic input models first [4, 6, 13, 32, 40, 44]. It was the observation [11] that memoryless sources for list accessing are not terribly realistic that partly motivated work on the adversarial models. It is highly plausible that both approaches are superseded by more sophisticated stochastic models such as hidden Markov models for gene finding or speech recognition, or time-coherent models for self-customized BSP trees [8] or for randomized incremental constructions [23]. Recently, Afshani, Barbay, and Chan [1] introduced the notion of *instance optimality*, which can be seen as a generalization of output-sensitivity. They consider the inputs as sets and try to exploit the structure *within* each input for faster algorithms.

Much research has been done on adaptive sorting [30], especially on algorithms that exploit near-sortedness. Our approach is conceptually different: we seek to exploit properties, not of individual inputs, but of their distribution. Algorithmic self-improvement differs from past work on self-organizing data structures and online computation in two fundamental ways. First, there is no notion of an adversary: the inputs are generated by a fixed, *oblivious*, random source $\mathcal{D}$, and we compare ourselves against an optimal comparison-based algorithm for $\mathcal{D}$. In particular, there is no

concept of competitiveness. Second, self-improving algorithms do not exploit structure within any given input but, rather, within the ensemble of input distributions.

A simple example highlights this difference between previous sorters and the self-improving versions. For $1 \leq i \leq n$, fix two random integers a_i, b_i from $\{1, \ldots, n^2\}$. The distribution $\mathcal{D}_i$ is such that $\Pr[x_i = a_i] = \Pr[x_i = b_i] = 1/2$, and we take $\mathcal{D} = \prod_{i=1}^{n} \mathcal{D}_i$. Observe that *every* permutation generated by $\mathcal{D}$ is a random permutation, since the a_i's and b_i's are chosen randomly. Hence, any solution in the adaptive, self-organizing/adjusting framework requires $\Omega(n \log n)$ time, because no input I_j exhibits any special structure to be exploited. On the other hand, our self-improving sorter will sort a permutation from $\mathcal{D}$ in expected *linear* time during the limiting phase: since $\mathcal{D}$ generates at most 2^n different permutations, we have $H(\pi(I)) = O(n)$.

2. Entropy and comparison-based algorithms. Before we consider sorting and Delaunay triangulations, let us first recall some useful properties of information theoretic entropy [28] and explain how it relates to our notion of comparison-based algorithms.

Let X be a random variable with a finite range $\mathcal{X}$. The *entropy* of X, $H(X)$, is defined as $H(X) := \sum_{x \in \mathcal{X}} \Pr[X = x] \log(1/\Pr[X = x])$. Intuitively, the event that $X = x$ gives us $\log(1/\Pr[X = x])$ bits of information about the underlying elementary event, and $H(X)$ represents the expected amount of information that can be obtained from observing X. We recall the following well-known property of the entropy of the Cartesian product of independent random variables [28, Theorem 2.5.1].

CLAIM 2.1. *Let $H(X_1, \ldots, X_n)$ be the joint entropy of independent random variables $X_1, \ldots, X_n$. Then*

$$H(X_1, \ldots, X_n) = \sum_i H(X_i).$$

We now define our notion of a comparison-based algorithm. Let $\mathcal{U}$ be an arbitrary universe, and let $\mathcal{X}$ be a finite set. A *comparison-based algorithm* to compute a function $X : \mathcal{U} \to \mathcal{X}$ is a rooted binary tree $\mathcal{A}$ such that (i) every internal node of $\mathcal{A}$ represents a comparison of the form $f(I) \leq g(I)$, where $f, g : \mathcal{U} \to \mathsf{R}$ are *arbitrary* functions on the input universe $\mathcal{U}$; and (ii) the leaves of $\mathcal{A}$ are labeled with outputs from $\mathcal{X}$ such that for every input $I \in \mathcal{U}$, following the appropriate path for I leads to the correct output $X(I)$. If $\mathcal{A}$ has maximum depth d, we say that $\mathcal{A}$ needs d comparisons (in the worst case). For a distribution $\mathcal{D}$ on $\mathcal{U}$, the *expected number of comparisons* (with respect to $\mathcal{D}$) is the expected length of a path from the root to a leaf in $\mathcal{A}$, where the leaves are sampled according to the distribution that $\mathcal{D}$ induces on $\mathcal{X}$ via X.

Note that our comparison-based algorithms generalize both the traditional notion of comparison-based algorithms [27, Chapter 8.1], where the functions f and g are required to be projections, as well as the notion of algebraic computation trees [9, Chapter 16.2]. Here the functions f and g must be composed of elementary functions (addition, multiplication, square root) such that the complexity of the composition is proportional to the depth of the node. Naturally, our comparison-based algorithms can be much stronger. For example, deciding whether a sequence $x_1, x_2, \ldots, x_n$ of real numbers consists of n distinct elements needs *one* comparison in our model, whereas every algebraic computation tree for the problem has depth $\Omega(n \log n)$ [9, Chapter 16.2]. However, for our problems of interest, we can still derive meaningful lower bounds.

CLAIM 2.2. *Let $\mathcal{D}$ be a distribution on a universe $\mathcal{U}$, and let $X : \mathcal{U} \to \mathcal{X}$ be a random variable. Then any comparison-based algorithm to compute X needs at least $H(X)$ expected comparisons.*

Proof. This is an immediate consequence of Shannon's noiseless coding theorem [28, Theorem 5.4.1], which states that any binary encoding of an information source such as $X(I)$ must have an expected code length of at least $H(X)$. Any comparison-based algorithm $\mathcal{A}$ represents a coding scheme: the encoder sends the sequence of comparison outcomes, and the decoder descends along the tree $\mathcal{A}$, using the transmitted sequence to determine comparison outcomes. Thus, any comparison-based algorithm must perform at least $H(X)$ comparisons in expectation. □

Note that our comparison-based algorithms include all the traditional sorting algorithms [27] (selection sort, insertion sort, quicksort, etc.) as well as classic algorithms for Delaunay triangulations [12] (randomized incremental construction, divide and conquer, plane sweep). A notable exception are sorting algorithms that rely on table lookup or the special structure of the input values (such as bucket sort or radix sort) as well as *transdichotomous* algorithms for sorting [33, 34] or Delaunay triangulations [16, 17, 18].

The following lemma shows how we can use the running times of comparison-based algorithms to relate the entropy of different random variables. This is a very important tool that will be used to prove the optimality of our algorithms.

LEMMA 2.3. *Let $\mathcal{D}$ be a distribution on a universe $\mathcal{U}$, and let $X : \mathcal{U} \to \mathcal{X}$ and $Y : \mathcal{U} \to \mathcal{Y}$ be two random variables. Suppose that the function f defined by $f : (I, X(I)) \mapsto Y(I)$ can be computed by a comparison-based algorithm with C expected comparisons (where the expectation is over $\mathcal{D}$). Then $H(Y) = C + O(H(X))$, where all the entropies are with respect to $\mathcal{D}$.*

Proof. Let $s : X(\mathcal{U}) \to \{0,1\}^*$ be a unique binary encoding of $X(\mathcal{U})$. By unique encoding, we mean that the encoding is $1-1$. We denote the *expected code length* of s with respect to $\mathcal{D}$, $\mathbf{E}_{\mathcal{D}}[|s(X(I))|]$, by E_s. By another application of Shannon's noiseless coding theorem [28, Theorem 5.4.1]), we have $E_s \geq H(X)$ for any unique encoding s of $X(\mathcal{U})$, and there exists a unique encoding s^* of $X(\mathcal{U})$ with $E_s = O(H(X))$.

Using f, we can convert s^* into a unique encoding t of $Y(\mathcal{U})$. Indeed, for every $I \in \mathcal{U}$, $Y(I)$ can be uniquely identified by a string $t(I)$ that is the concatenation of $s^*(X(I))$ and additional bits that represent the outcomes of the comparisons for the algorithm to compute $f(I, X(I))$. Thus, for every element $y \in Y(\mathcal{U})$, we can define $t(y)$ as the lexicographically smallest string $t(I)$ for which $Y(I) = y$, and we obtain a unique encoding t for $Y(\mathcal{U})$. For the expected code length E_t of t, we get

$$E_t = E_{\mathcal{D}}[|t(Y(I))|] \leq E_{\mathcal{D}}[C + |s^*(X(I))|] = C + E_s = C + O(H(X)).$$

Since Shannon's theorem implies $E_t \geq H(Y)$, the claim follows. □

3. A self-improving sorter. We are now ready to describe our self-improving sorter. The algorithm takes an input $I = (x_1, x_2, \ldots, x_n)$ of real numbers drawn from a distribution $\mathcal{D} = \prod_i \mathcal{D}_i$ (i.e., each x_i is chosen independently from $\mathcal{D}_i$). Let $\pi(I)$ denote the permutation induced by the ranks of the x_i's, using the indices i to break ties. By applying Claim 2.2 with $\mathcal{U} = \mathsf{R}^n$, $\mathcal{X}$ the set of all permutations on $\{1, \ldots, n\}$, and $X(I) = \pi(I)$, we see that any sorter must make at least $H(\pi(I))$ expected comparisons. Since it takes $\Omega(n)$ steps to write the output, any sorter needs $\Omega(H(\pi(I)) + n)$ steps. This is, indeed, the bound that our self-improving sorter achieves.

For simplicity, we begin with the steady-state algorithm and discuss the training phase later. We also assume that the distribution $\mathcal{D}$ is known ahead of time and that we are allowed some amount of preprocessing before having to deal with the first input instance (section 3.1). Both assumptions are unrealistic, so we show how to remove them to produce a bona fide self-improving sorter (section 3.2). The surprise is how strikingly little of the distribution needs to be learned for effective self-improvement.

3.1. Sorting with full knowledge. We consider the problem of sorting $I = (x_1, \ldots, x_n)$, where each x_i is a real number drawn from a distribution $\mathcal{D}_i$. We can assume without loss of generality that all the x_i's are distinct. (If not, simply replace x_i by $x_i + i\delta$ for an infinitesimally small $\delta > 0$, so that ties are broken according to the index i.)

The first step of the self-improving sorter is to sample $\mathcal{D}$ a few times (the training phase) and create a "typical" instance to divide the real line into a set of disjoint, sorted intervals. Next, given some input I, the algorithm sorts I by using the typical instance, placing each input number in its respective interval. All numbers falling into the same intervals are then sorted in a standard fashion. The algorithm needs a few supporting data structures.

- *The V-list.* Fix an integer parameter $\lambda = \lceil \log n \rceil$, and sample λ input instances from $\prod \mathcal{D}_i$. Form their union, and sort the resulting λn-element multiset into a single list $u_1 \le \cdots \le u_{\lambda n}$. Next, extract from it every λth item and form the list $V = (v_0, \ldots, v_{n+1})$, where $v_0 = 0$, $v_{n+1} = \infty$, and $v_i = u_{i\lambda}$ for $1 \le i \le n$. Keep the resulting *V-list* in a sorted table as a snapshot of a "typical" input instance. We will prove the remarkable fact that, with high probability, locating each x_i in the V-list is linearly equivalent to sorting I. We cannot afford to search the V-list directly, however. To do that, we need auxiliary search structures.
- *The D_i-trees.* For any $i \ge 1$, let $\mathcal{B}_i^V$ be the predecessor[2] of a random y from $\mathcal{D}_i$ in the V-list, and let H_i^V be the entropy of $\mathcal{B}_i^V$. The *D_i-tree* is defined to be an optimum binary search tree [41] over the keys of the V-list, where the access probability of v_k is $\Pr_{\mathcal{D}_i}\big[x_i \in [v_k, v_{k+1})\big] = \Pr\big[\mathcal{B}_i^V = k\big]$ for any $0 \le k \le n$. This allows us to compute $\mathcal{B}_i^V$ using $O(H_i^V + 1)$ expected comparisons.

The self-improving sorter. The input I is sorted by a two-phase procedure. First we locate each x_i in the V-list using the D_i-trees. This allows us to partition I into groups $Z_0 < Z_1 < \cdots$ of x_i's sharing the same predecessor in the V-list. The first phase of the algorithm takes an $O(n + \sum_i H_i^V)$ expected time.[3] The next phase involves going through each Z_k and sorting their elements naively, say, using insertion sort, in total time $O(\sum_k |Z_k|^2)$. See Figure 3.1.

The expected running time is $O(n + \mathbf{E}_{\mathcal{D}}[\sum_i H_i^V + \sum_k |Z_k|^2])$, and the total space used is $O(n^2)$. This can be decreased to $O(n^{1+\varepsilon})$ for any constant $\varepsilon > 0$; we describe how at the end of this section. First, we show how to bound the running time of the first phase. This is where we really show the optimality of our sorter.

LEMMA 3.1.

$$\sum_i H_i^V = O(n + H(\pi(I))).$$

[2]Throughout this paper, the predecessor of y in a list refers to the index of the largest list element $\le y$; it does not refer to the element itself.

[3]The H_i^V's themselves are random variables depending on the choice of the V-list. Therefore, this is a conditional expectation.

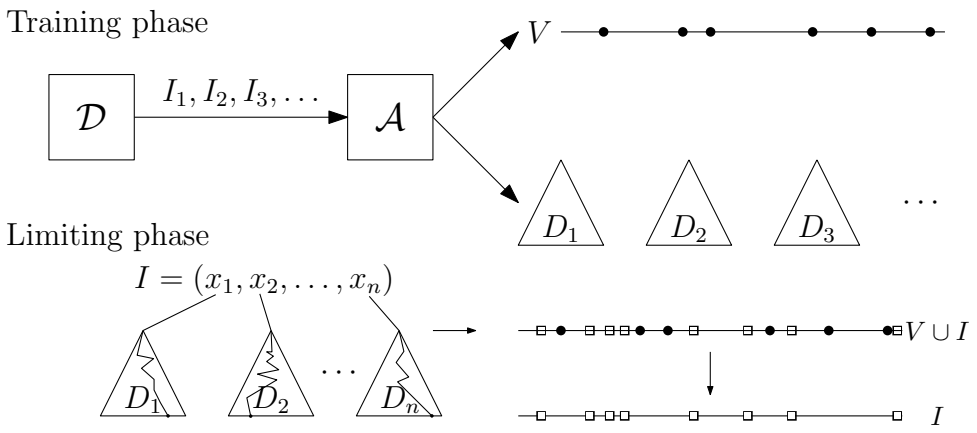

FIG. 3.1. *The self-improving sorter: during the training phase, the algorithm constructs a typical sorted list, the V-list, and a sequence D_1, D_2, ... of optimal search trees for V with respect to $\mathcal{D}_1$, $\mathcal{D}_2$, ... In the limiting phase, the algorithm uses the D_i's to locate the x_i's in the V-list, sorts the individual buckets, and removes the elements from V.*

Proof. Our proof actually applies to *any* linear-sized sorted list V. Let $\mathcal{B}^V := (\mathcal{B}_1^V, \ldots, \mathcal{B}_n^V)$ be the sequence of predecessors for all elements in I. By Claim 2.1, we have $H(\mathcal{B}^V) = \sum_i H_i^V$, so it suffices to bound the entropy of $H(\mathcal{B}^V)$. By Lemma 2.3 applied with $\mathcal{U} = \mathsf{R}^n$, $X(I) = \pi(I)$, and $Y(I) = \mathcal{B}^V$, it suffices to give a comparison-based algorithm that can determine $\mathcal{B}^V(I)$ from $(I, \pi(I))$ with $O(n)$ comparisons. But this is easy: just use $\pi(I)$ to sort I (which needs no further comparisons), and then merge the sorted list I with V. Now the lemma follows from Claim 2.1. ∎

Next we deal with the running time of the second phase. As long as the groups Z_k are small, the time to sort each group will be small. The properties of the V-list ensure that this is the case.

LEMMA 3.2. *For $0 \le k \le n$, let $Z_k = \{x_i \mid v_k \le x_i < v_{k+1}\}$ be the elements with predecessor k. With probability at least $1 - n^{-2}$ over the construction of the V-list, we have $\mathbf{E}_{\mathcal{D}}\big[|Z_k|\big] = O(1)$ and $\mathbf{E}_{\mathcal{D}}\big[|Z_k|^2\big] = O(1)$ for all $0 \le k \le n$.*

Proof. Remember that the V-list was formed by taking certain elements from a sequence $\hat{I} = s_1, s_2, \ldots, s_{\lambda n}$ that was obtained by concatenating $\lambda = \lceil \log n \rceil$ inputs $I_1, I_2, \ldots$. Let $s_i \le s_j$ be any two elements from $\hat{I}$, and let $t = [s_i, s_j)$. Note that all the other $\lambda n - 2$ numbers are independent of s_i and s_j. Suppose we fix the values of s_i and s_j (in other words, we condition on the values of s_i and s_j). For every $\ell \in \{1, \ldots, \lambda n\} \setminus \{i, j\}$, let $Y_\ell^{(t)}$ be the indicator random variable for the event that $s_\ell \in t$, and let $Y^{(t)} := \sum_\ell Y_\ell^{(t)}$. Since all the $Y_\ell^{(t)}$'s are independent, by Chernoff's bound [42, Theorem 4.2], for any $\beta \in [0, 1]$,

$$\Pr[Y^{(t)} \le (1-\beta)\mathbf{E}[Y^{(t)}]] \le \exp\big(-\beta^2 \mathbf{E}[Y^{(t)}]/2\big). \tag{3.1}$$

Setting $\beta = 10/11$, we see that if $\mathbf{E}[Y^{(t)}] > 11\lceil \log n \rceil$, then $Y^{(t)} > \lceil \log n \rceil$ with probability at least $1 - 1/(\lambda^2 n^4)$. Note that this is true for *every* fixing of s_i and s_j. Therefore, we get the above statement even with the unconditioned random variable $Y^{(t)}$. Now, by applying the same argument to any pair s_i, s_j with $i \ne j$ and taking a union bound over all $\binom{\lambda n}{2}$ such pairs, we get that with probability at least $1 - n^{-2}$ over the construction of $\hat{I}$ the following holds for all half-open intervals t defined by

pairs s_i, s_j with $i \neq j$: if $\mathbf{E}[Y^{(t)}] > 11\lceil \log n \rceil$, then $Y^{(t)} > \lceil \log n \rceil$. From now on we assume that this implication holds.

The V-list is constructed such that for $t_k = [v_k, v_{k+1})$, $Y^{(t_k)} \leq \lceil \log n \rceil$, and hence $\mathbf{E}[Y^{(t_k)}] = O(\log n)$. Let $X_i^{(t_k)}$ be the indicator random variable for the event that $x_i \in_R \mathcal{D}_i$ lies in t_k, and let $X^{(t_k)} := \sum_i X_i^{(t_k)} = |Z_k|$. Note that (where a and b denote the indices of v_k and v_{k+1} in $\hat{I}$)

$$\mathbf{E}[Y^{(t_k)}] = \sum_{\ell \neq a,b} \mathbf{E}[Y_\ell^{(t_k)}] \geq \sum_i \lambda \mathbf{E}[X_i^{(t_k)}] - 2 = \lceil \log n \rceil \mathbf{E}[X^{(t_k)}] - 2,$$

and therefore $\mathbf{E}[X^{(t_k)}] = O(1)$. Now, since the expectation of $X^{(t_k)}$ is constant, and since $X^{(t_k)}$ is a sum of independent indicator random variables, we can apply the following standard claim in order to show that the second moment of $X^{(t_k)}$ is also constant.

CLAIM 3.3. *Let $X = \sum_i X_i$ be a sum of independent positive random variables with $X_i = O(1)$ for all i and $\mathbf{E}[X] = O(1)$. Then $\mathbf{E}[X^2] = O(1)$.*

Proof. By linearity of expectation,

$$\mathbf{E}\left[X^2\right] = \mathbf{E}\left[\left(\sum_i X_i\right)^2\right] = \sum_i \mathbf{E}\left[X_i^2\right] + 2\sum_{i<j} \mathbf{E}[X_i]\mathbf{E}[X_j]$$
$$\leq \sum_i O\left(\mathbf{E}[X_i]\right) + \left(\sum_i \mathbf{E}[X_i]\right)^2 = O(1). \qquad \square$$

This concludes the proof of Lemma 3.2. $\square$

Combining Lemmas 3.1 and 3.2, we get the running time of our self-improving sorter to be $O(n + H(\pi(I)))$. This proves the optimality of time taken by the sorter.

We now show that the storage can be reduced to $O(n^{1+\varepsilon})$ for any constant $\varepsilon > 0$. The main idea is to prune each D_i-tree to depth $\varepsilon \log n$. This ensures that each tree has size $O(n^\varepsilon)$, so the total storage used is $O(n^{1+\varepsilon})$. We also construct a completely balanced binary tree T for searching in the V-list. Now, when we wish to search for x_i in the V-list, we first search using the pruned D_i-tree. At the end, if we reach a leaf of the *unpruned* D_i-tree, we stop since we have found the right interval of the V-list which contains x_i. On the other hand, if the search in the D_i-tree was unsuccessful, then we use T for searching.

In the first case, the time taken for searching is simply the same as with unpruned D_i-trees. In the second case, the time taken is $O((1+\varepsilon)\log n)$. But note that the time taken with unpruned D_i-trees is at least $\varepsilon \log n$ (since the search on the pruned D_i-tree failed, we must have reached some internal node of the unpruned tree). Therefore, the extra time taken is only a $O(\varepsilon^{-1})$ factor of the original time. As a result, the space can be reduced to $O(n^{1+\varepsilon})$ with only a constant factor increase in running time (for any fixed $\varepsilon > 0$).

3.2. Learning the distribution. In the last section we showed how to obtain a self-improving sorter if $\mathcal{D}$ is known. We now explain how to remove this assumption. The V-list is built in the first $\lceil \log n \rceil$ rounds, as before. The D_i-trees will be built after $O(n^\varepsilon)$ additional rounds, which will complete the training phase. During that phase, sorting is handled via, say, mergesort, to guarantee $O(n \log n)$ complexity. The training part consists of learning basic information about $\mathcal{B}_i^V$ for each i. For notational

simplicity, fix i, and let $p_k = \Pr[\mathcal{B}_i^V = k] = \Pr_{\mathcal{D}_i}[\, v_k \le x_i < v_{k+1}\,]$. Let $M = cn^\varepsilon$ for a large enough constant c. For any k, let χ_k be the number of times, over the first M rounds, that v_k is found to be the V-list predecessor of x_i. (We use a standard binary search to compute predecessors in the training phase.) Finally, define the D_i-tree to be a weighted binary search tree defined over all the v_k's such that $\chi_k > 0$. Recall that the crucial property of such a tree is that the node associated with a key of weight χ_k is at depth $O(\log(M/\chi_k))$. We apply this procedure for each $i = 1, \ldots, n$.

This D_i-tree is essentially the pruned version of the tree we used in section 4.1. Like before, its size is $O(M) = O(n^\varepsilon)$, and it is used in a similar way as in section 4.1, with a few minor differences. For completeness, we go over it again: given x_i, we perform a search down the D_i-tree. If we encounter a node whose associated key v_k is such that $x_i \in [v_k, v_{k+1})$, we have determined $\mathcal{B}_i^V$, and we stop the search. If we reach a leaf of the D_i-tree without success, we simply perform a standard binary search in the V-list.

LEMMA 3.4. *Fix i. With probability at least $1 - 1/n^3$, for any k, if $p_k > n^{-\varepsilon/3}$, then $Mp_k/2 < \chi_k < 3Mp_k/2$.*

Proof. The expected value of χ_k is Mp_k. If $p_k > n^{-\varepsilon/3}$, then, by Chernoff's bound [7, Corollary A.17], the count χ_k deviates from its expectation by more than $a = Mp_k/2$ with probability less than (recall that $M = cn^\varepsilon$)

$$2\exp(-2a^2/M) = 2\exp(-Mp_k^2/2) < 2\exp(-(c/2)n^{2\varepsilon/3}) \le n^{-4}$$

for c large enough. A union bound over all k completes the proof. ∎

Suppose now the implication of Lemma 3.4 holds for all k (and fixed i). We show now that the expected search time for x_i is $O(\varepsilon^{-1} H_i^V + 1)$. Consider each element in the sum $H_i^V = \sum_k p_k \log(1/p_k)$. We distinguish two cases.

- *Case* 1: $p_k > n^{-\varepsilon/3}$. In this case, v_k must be in D_i, as otherwise we would have $\chi_k = 0$ by the definition of D_i, which is a contradiction (for n large enough) to Lemma 3.4, which states that $\chi_k > Mn^{-\varepsilon/3}/2$. Hence, the cost of the search is $O(\log(M/\chi_k))$, and its contribution to the expected search time is $O(p_k \log(M/\chi_k))$. By Lemma 3.4, this is also $O(p_k(1 + \log p_k^{-1}))$, as desired.
- *Case* 2: $p_k \le n^{-\varepsilon}$. The search time is always $O(\log n)$; hence the contribution to the expected search time is $O(\varepsilon^{-1} p_k \log p_k^{-1})$.

By summing up over all k, we find that the expected search time is $O(\varepsilon^{-1} H_i^V + 1)$. This assumes the implication of Lemma 3.4 for all i. By a union bound, this holds with probability at least $1 - 1/n^2$. The training phase fails when either this does not hold, or if the V-list does not have the desired properties (Lemma 3.2). The total probability of this is at most $1/n$.

3.3. Lower bounds. Can we hope for a result similar to Theorem 1.1 if we drop the independence assumption? The short answer is no. As we mentioned earlier, Fredman [31] gave a comparison-based algorithm that can optimally sort any distribution of permutations. This uses an *exponentially* large data structure to decide which comparisons to perform. Our lower bound shows that the storage used by Fredman's algorithm is essentially optimal.

To understand the lower bound, let us examine the behavior of a self-improving sorter. Given inputs from a distribution $\mathcal{D}$, at each round, the self-improving sorter is just a comparison tree for sorting. After any round, the self-improving sorter may wish to update the comparison tree. At some round (eventually), the self-improving sorter must be able to sort with expected $O(n + H(\pi(I)))$ comparisons: the algorithm has

"converged" to the optimal comparison tree. The algorithm uses some data structure to represent (implicitly) this comparison tree.

We can think of a more general situation. The algorithm is explicitly given an input distribution $\mathcal{D}$. It is allowed some space where it stores information about $\mathcal{D}$ (we do not care about the time spent to do this). Then, (using this stored information) it must be able to sort a permutation from $\mathcal{D}$ in expected $O(n + H(\pi(I)))$ comparisons. So the information encodes some fixed comparison-based procedure. As shorthand for the above, we will say that the *algorithm, on input distribution $\mathcal{D}$, optimally sorts $\mathcal{D}$.* How much space is required to deal with all possible $\mathcal{D}$'s? Or just to deal with product distributions? These are the questions that we shall answer.

LEMMA 3.5. *Let $h = (n \log n)/\alpha$ for some sufficiently large constant $\alpha < 0$, and let $\mathcal{A}$ be an algorithm that can optimally sort any input distribution $\mathcal{D}$ with $H(\pi(I)) \le h$. Then $\mathcal{A}$ requires $2^{\Omega(n \log n)}$ bits of storage.*

Proof. Consider the set of all $n!$ permutations of $\{1, \dots, n\}$. Every subset Π of 2^h permutations induces a distribution $\mathcal{D}^\Pi$ defined by picking every permutation in Π with equal probability and none other. Note that the total number of such distributions $\mathcal{D}^\Pi$ is $\binom{n!}{2^h} > (n!/2^h)^{2^h}$ and $H(\mathcal{D}^\Pi_<) = h$, where $\mathcal{D}^\Pi_<$ is the distribution on the output $\pi(I)$ induced by $\mathcal{D}^\Pi$. Suppose there exists a comparison-based procedure $\mathcal{A}_\Pi$ that sorts a random input from $\mathcal{D}^\Pi$ in expected time at most $c(n + h)$ for some constant $c > 0$. By Markov's inequality this implies that at least half of the permutations in Π are sorted by $\mathcal{A}_\Pi$ in at most $2c(n + h)$ comparisons. But, within $2c(n + h)$ comparisons, the procedure $\mathcal{A}_\Pi$ can only sort a set P of at most $2^{2c(n+h)}$ permutations. Therefore, any other Π' such that $\mathcal{A}_\Pi = \mathcal{A}_\Pi$ will have to draw at least half of its elements from P. This limits the number of such Π' to

$$\binom{n!}{2^h/2}\binom{2^{2c(n+h)}}{2^h/2} < (n!)^{2^{h-1}} 2^{c(n+h)2^h}.$$

This means that the number of distinct procedures needed exceeds

$$(n!/2^h)^{2^h}/((n!)^{2^{h-1}} 2^{c(n+h)2^h}) > (n!)^{2^{h-1}} 2^{-(c+1)(n+h)2^h} = 2^{\Omega(2^h n \log n)},$$

assuming that $h/(n \log n)$ is small enough. A procedure is entirely specified by a string of bits; therefore, at least one such procedure must require storage logarithmic in the previous bound. ∎

We now show that a self-improving sorter dealing with product distributions requires superlinear size. In fact, the achieved tradeoff between the $O(n^{1+\varepsilon})$ storage bound and an expected running time off the optimal by a factor of $O(1/\varepsilon)$ is optimal.

LEMMA 3.6. *Let $c > 0$ be a large enough parameter, and let $\mathcal{A}$ be an algorithm that, given a product distribution $\mathcal{D}$, can sort a random permutation from $\mathcal{D}$ in expected time $c(n + H(\pi(I)))$. Then $\mathcal{A}$ requires a data structure of bit size $n^{1+\Omega(1/c)}$.*

Proof. The proof is a specialization of the argument used for proving Lemma 3.5. Let $h = (n \log n)/(3c)$ and $\kappa = 2^{\lfloor h/n \rfloor}$. We define $\mathcal{D}_i$ by choosing κ distinct integers in $\{1, \dots, n\}$ and making them equally likely to be picked as x_i. (For convenience, we use the tie-breaking rule that maps $x_i \mapsto nx_i + i - 1$. This ensures that $\pi(I)$ is unique.) We then set $\mathcal{D} := \prod_i \mathcal{D}_i$. By Claim 2.1, $\mathcal{D}$ has entropy $n \cdot \lfloor h/n \rfloor = \Theta(h)$. This leads to $\binom{n}{\kappa}^n > (n/\kappa)^{\kappa n}$ choices of distinct distributions $\mathcal{D}$. Suppose that $\mathcal{A}$ uses s bits of storage and can sort each such distribution in $c(n + h)$ expected comparisons. Some fixing $\mathcal{S}$ of the bits must be able to accommodate this running time for a set $\mathcal{G}$ of at least $(n/\kappa)^{\kappa n} 2^{-s}$ distributions $\mathcal{D}$. In other words, some comparison-based procedure

can deal with $(n/\kappa)^{\kappa n}2^{-s}$ distributions $\mathcal{D}$. Any input instance that is sorted in at most $2c(h+n)$ time by $\mathcal{S}$ is called *easy*: the set of easy instances is denoted by $\mathcal{E}$.

Because $\mathcal{S}$ has to deal with many distributions, there must be many instances that are easy for $\mathcal{S}$. This gives a lower bound for $|\mathcal{E}|$. On the other hand, since easy instances are those that are sorted extremely quickly by $\mathcal{S}$, there cannot be too many of them. This gives an upper bound for $|\mathcal{E}|$. Combining these two bounds, we get a lower bound for s. We will begin with the easier part: the upper bound for $|\mathcal{E}|$.

CLAIM 3.7. $|\mathcal{E}| \leq 2^{2c(h+n)+2}$.

Proof. In the comparison-based algorithm represented by $\mathcal{S}$, each instance $I \in \mathcal{E}$ is associated with a leaf of a binary decision tree of depth at most $2c(h+n)$, i.e., with one of at most $2^{2c(h+n)}$ leaves. This would give us an upper bound on s if each $I \in \mathcal{E}$ were assigned a distinct leaf. However, it may well be that two distinct inputs $I, I' \in \mathcal{E}$ have $\pi(I) = \pi(I')$ and lead to the same leaf. Nonetheless, we have a *collision* bound, saying that for any permutation π, there are at most 4^n instances $I \in \mathcal{E}$ with $\pi(I) = \pi$. This implies that $|\mathcal{E}| \leq 4^n 2^{2c(h+n)}$.

To prove the collision bound, first fix a permutation π. How many instances can map to this permutation? We argue that knowing that $\pi(I) = \pi$ for an instance $I \in \mathcal{E}$, we need only $2n-1$ additional bits to encode I. This immediately shows that there must be fewer than 4^n such instances I. Write $I = (x_1, \ldots, x_n)$, and let I be sorted to give the vector $\overline{I} = (y_1, \ldots, y_n)$. Represent the ground set of I as an n-bit vector α ($\alpha_i = 1$ if some $x_j = i$, or else $\alpha_i = 0$). For $i = 2, \ldots, n$, let $\beta_i = 1$ if $y_i = y_{i-1}$; else $\beta_i = 0$. Now, given α and β, we can immediately deduce the vector $\overline{I}$, and by applying π^{-1} to $\overline{I}$, we get I. This proves the collision bound. ∎

CLAIM 3.8. $|\mathcal{E}| \geq n^n \kappa^{-2n} 2^{-2s/\kappa}$.

Proof. Each $\mathcal{D}_i$ is characterized by a vector $v_i = (a_{i,1}, \ldots, a_{i,\kappa})$, so that $\mathcal{D}$ itself is specified by $v = (v_1, \ldots, v_n) \in \mathsf{R}^{n\kappa}$. (From now on, we view v both as a vector and as a distribution of input instances.) Define the jth projection of v as $v^j = (a_{1,j}, \ldots, a_{n,j})$. Even if $v \in \mathcal{G}$, it could well be that none of the projections of v are easy. However, if we consider the projections obtained by permuting the coordinates of each vector $v_i = (a_{i,1}, \ldots, a_{i,\kappa})$ in all possible ways, we enumerate each input instance from v the same number of times. Note that applying these permutations gives us different vectors which also represent $\mathcal{D}$. Since the expected time to sort an input chosen from $\mathcal{D} \in \mathcal{G}$ is at most $c(h+n)$, by Markov's inequality, there exists a choice of permutations (one for each $1 \leq i \leq n$) for which at least half of the projections of the vector obtained by applying these permutations are easy.

Let us count how many distributions have a vector representation with a choice of permutations placing half its projections in $\mathcal{E}$. There are fewer than $|\mathcal{E}|^{\kappa/2}$ choices of such instances, and, for any such choice, each $v_i' = (a_{i,1}, \ldots, a_{i,\kappa})$ has half its entries already specified, so the remaining choices are fewer than $n^{\kappa n/2}$. This gives an upper bound of $n^{\kappa n/2}|\mathcal{E}|^{\kappa/2}$ on the number of such distributions. This number cannot be smaller than $|\mathcal{G}| \geq (n/\kappa)^{\kappa n}2^{-s}$; therefore $|\mathcal{E}| \geq n^n\kappa^{-2n}2^{-2s/\kappa}$, as desired. ∎

It now just remains to put the bounds together:

$$\begin{aligned} & n^n\kappa^{-2n}2^{-2s/\kappa} && \leq && 2^{2c(h+n)+2} \\ \Longrightarrow\ & n\log n - 2n\log\kappa - 2s/\kappa && \leq && 2ch + 2cn + 2 \\ \Longrightarrow\ & \kappa n(\log n - 2\log\kappa) - 2c\kappa h - 2c\kappa n - 2\kappa && \leq && 2s. \end{aligned}$$

We have $\kappa = n^{\Theta(1/c)}$ and $h = (n\log n)/(3c)$. Since c is sufficiently large, we get $s = n^{1+\Omega(1/c)}$. ∎

4. Delaunay triangulations. We now consider self-improving algorithms for Delaunay triangulations. The aim of this section is to prove Theorem 1.3. Let $I = (x_1, \dots, x_n)$ denote an input instance, where each x_i is a point in the plane, generated by a point distribution $\mathcal{D}_i$. The distributions $\mathcal{D}_i$ are arbitrary and may be continuous, although we never explicitly use such a condition. Each x_i is independent of the others, so in each round the input I is drawn from the product distribution $\mathcal{D} = \prod_i \mathcal{D}_i$, and we wish to compute the Delaunay triangulation of I, $T(I)$. To keep our arguments simple, we will assume that the points of I are in *general position* (i.e., no four points in I lie on a common circle). This is no loss of generality and does not restrict the distribution $\mathcal{D}$, because the general position assumption can always be enforced by standard symbolic perturbation techniques [29]. Also we will assume that there is a bounding triangle that always contains all the points in I. Again, this does not restrict the distribution $\mathcal{D}$ in any way, because we can always simulate the bounding triangle symbolically by adding virtual points at infinity.

The distribution $\mathcal{D}$ induces a (discrete) distribution on the set of Delaunay triangulations, viewed as undirected graphs with vertex set $\{1, \dots, n\}$. Consider the entropy of this distribution: for each graph G on $\{1, \dots, n\}$, let p_G be the probability that it represents the Delaunay triangulation of $I \in_R \mathcal{D}$. We have the output entropy $H(T(I)) := -\sum_G p_G \log p_G$. By Claim 2.2, any comparison-based algorithm to compute the Delaunay triangulation of $I \in_R \mathcal{D}$ needs at least $H(T(I))$ expected comparisons. Hence, an *optimal* algorithm will be one that has an expected running time of $O(n + H(T(I)))$ (since it takes $O(n)$ steps to write the output).

We begin by describing the basic self-improving algorithm. (As before, we shall first assume that some aspects of the distribution $\mathcal{D}$ are known.) Then, we shall analyze the running time using our information theory tools to argue that the expected running time is optimal. Finally, we remove the assumption that $\mathcal{D}$ is known and give the time-space tradeoff in Theorem 1.3.

4.1. The algorithm. We describe the algorithm in two parts. The first part explains the learning phase and the data structures that are constructed (section 4.1.1). Then, we explain how these data structures are used to speed up the computation in the limiting phase (section 4.1.2). As before, the expected running time will be expressed in terms of certain parameters of the data structures obtained in the learning phase. In the next section (section 4.2), we will prove that these parameters are comparable to the output entropy $H(T(I))$. First, we will assume that the distributions $\mathcal{D}_i$ are known to us and the data structures described will use $O(n^2)$ space. Section 4.3 repeats the arguments of section 3.2 to remove this assumption and to give the space-time tradeoff bounds of Theorem 1.3.

As outlined in Figure 4.1, our algorithm for Delaunay triangulation is roughly a generalization of our algorithm for sorting. This is not surprising, but note that while the steps of the two algorithms, and their analyses, are analogous, in several cases a step for sorting is trivial, but the corresponding step for Delaunay triangulation uses some relatively recent and sophisticated prior work.

4.1.1. Learning phase. For each round in the learning phase, we use a standard algorithm to compute the output Delaunay triangulation. We also perform some extra computation to build some data structures that will allow speedup in the limiting phase.

The learning phase is as follows. Take the first $\lambda := \lceil \log n \rceil$ input lists $I_1, I_2, \dots, I_\lambda$. Merge them into one list $\hat{I}$ of $\lambda n = n\lceil \log n \rceil$ points. Setting $\varepsilon := 1/n$, find an ε-net $V \subseteq \hat{I}$ for the set of all open disks. In other words, find a set V such that for

Sorting	Delaunay triangulation
Intervals (x_i, x_i) containing no values of I	Delaunay disks
Typical set V	Range space ε-net V [26, 39], ranges are disks, $\varepsilon = 1/n$
$\log n$ training instance points with the same $\mathcal{B}_V$ value	$\log n$ training instance points in each Delaunay disk
Expect $O(1)$ values of I within each bucket (of the same $\mathcal{B}^V$ index)	Expect $O(1)$ points of I in each Delaunay disk of V
Optimal weighted binary trees D_i	Entropy-optimal planar point location data structures D_i [10]
Sorting within buckets	Triangulation within $\mathcal{V}(Z_s) \cap s$ (Claim 4.5)
Sorted list of $V \cup I$	$T(V \cup I)$
Build sorted V from sorted $V \cup I$ (trivial)	Build $T(I)$ from $T(V \cup I)$ [21, 22]
(analysis) merge sorted V and I	(analysis) merge $T(V)$ and $T(I)$ [19]
(analysis) recover the indices $\mathcal{B}_i^V$ from the sorted I (trivial)	(analysis) recover the triangles $\mathcal{B}_i^V$ in $T(V)$ from $T(I)$ (Lemma 4.8)

FIG. 4.1. *Delaunay triangulation algorithm as a generalization of the sorting algorithm.*

any open disk C that contains more than $\varepsilon\lambda n = \lceil \log n \rceil$ points of $\hat{I}$, C contains at least one point of V. It is well known that that there exist ε-nets of size $O(1/\varepsilon)$ for disks [26, 38, 39, 43], which here is $O(n)$. Furthermore, it is folklore that our desired ε-net V can be constructed in time $n(\log n)^{O(1)}$, but there seems to be no explicit description of such an algorithm for our precise setting. Thus, we present an algorithm based on a construction by Pyrga and Ray [43] in the appendix.

Having obtained V, we construct the Delaunay triangulation of V, which we denote by $T(V)$. This is the analogue of the V-list for the self-improving sorter. We also build an optimal planar point location structure (called D) for $T(V)$: given a point, we can find in $O(\log n)$ time the triangle of $T(V)$ that it lies in [12, Chapter 6]. Define the random variable $\mathcal{B}_i^V$ to be the triangle of $T(V)$ that x_i falls into.[4] Now let the entropy of $\mathcal{B}_i^V$ be H_i^V. If the probability that x_i falls into triangle t of $T(V)$ is p_i^t, then $H_i^V = -\sum_t p_i^t \log p_i^t$. For each i, we construct a search structure D_i of size $O(n)$ that finds $\mathcal{B}_i^V$ in an expected $O(H_i^V)$ time. These D_i's can be constructed using the results of Arya et al. [10], for which the expected number of primitive comparisons is $H_i^V + o(H_i^V)$. These correspond to the D_i-trees used for sorting.

We will now prove an analogue to Lemma 3.2 which shows that the triangles of $T(V)$ do not contain many points of a new input $I \in_R \mathcal{D}$ on the average. Consider a triangle t of $T(V)$, and let C_t be its circumscribed disk; C_t is a Delaunay disk of V. If a point $x_i \in I$ lies in C_t, we say that x_i is *in conflict* with t and call t a *conflict triangle* for x_i. Refer to Figure 4.2. (The "conflict" terminology arises from the fact that if x_i were added to V, triangles with which it conflicts would no longer be in the Delaunay triangulation.) Let $Z_t := I \cap C_t$, the random variable that represents the points of $I \in_R \mathcal{D}$ that fall inside C_t, the *conflict set* of t. Furthermore, let $X_t := |Z_t|$.

[4] Assume that we add the vertices of the bounding triangle to V. This will ensure that x_i will always fall within some triangle $\mathcal{B}_i^V$.

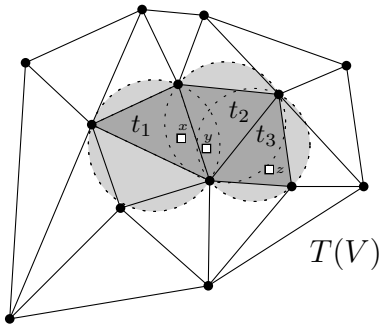

FIG. 4.2. *Conflicts between $T(V)$ and the inputs: the input point x conflicts with triangles t_1 and t_2, y conflicts with t_1, t_2, and t_3, and z conflicts only with t_3.*

Note that the randomness comes from the random distribution of $\hat{I}$ (on which V and $T(V)$ depend) as well as the randomness of I. We are interested in the expectation $\mathbf{E}[X_t]$ over I of X_t. All expectations are taken over a random input I chosen from $\mathcal{D}$.

LEMMA 4.1. *For any triangle t of $T(V)$, let $Z_t = \{x_i \mid x_i \in C_t\}$ be the conflict set of t, and define $X_t := |Z_t|$. With probability at least $1 - n^{-2}$ over the construction of $T(V)$, we have $\mathbf{E}[X_t] = O(1)$ and $\mathbf{E}[X_t^2] = O(1)$ for all triangles t of $T(V)$.*

Proof. This is similar to the argument given in Lemma 3.2 with a geometric twist. Let the list of points $\hat{I}$ be $s_1, \ldots, s_{\lambda n}$, the concatenation of I_1 through I_λ. Fix three distinct indices i, j, k and the triangle t with vertices s_i, s_j, s_k (so we are effectively conditioning on s_i, s_j, s_k). Note that all the remaining $\lambda n - 3$ points are chosen independently of s_i, s_j, s_k from some distribution $\mathcal{D}_\ell$. For each $\ell \in \{1, \ldots, \lambda n\} \setminus \{i, j, k\}$, let $Y_\ell^{(t)}$ be the indicator variable for the event that s_ℓ is inside C_t. Let $Y^{(t)} = \sum_\ell Y_\ell^{(t)}$. Setting $\beta = 11/12$ in (3.1), we get that if $\mathbf{E}[Y^{(t)}] > 12\lceil \log n \rceil$, then $Y^{(t)} > \lceil \log n \rceil$ with probability at least $1 - 1/(\lambda^3 n^5)$. This is true for every fixing of s_i, s_j, s_k, so it is also true unconditionally. By applying the same argument to any triple i, j, k of distinct indices, and taking a union bound over all $\binom{\lambda n}{3}$ triples, we obtain that with probability at least $1 - n^{-2}$, for any triangle t generated by the points of $\hat{I}$, if $\mathbf{E}[Y^{(t)}] > 12\lceil \log n \rceil$, then $Y^{(t)} > \lceil \log n \rceil$. We henceforth assume that this event happens.

Consider a triangle t of $T(V)$ and its circumcircle C_t. Since $T(V)$ is Delaunay, C_t contains no point of V in its interior. Since V is a $(1/n)$-net for all disks with respect to $\hat{I}$, C_t contains at most $\lceil \log n \rceil$ points of $\hat{I}$, that is, $Y^{(t)} \leq \lceil \log n \rceil$. This implies that $\mathbf{E}[Y^{(t)}] = O(\log n)$, as in the previous paragraph. Since $\mathbf{E}[Y^{(t)}] > \log n \mathbf{E}[X_t] - 3$, we obtain $\mathbf{E}[X_t] = O(1)$, as claimed. Furthermore, since X_t can be written as a sum of independent indicator random variables, Claim 3.3 shows that $\mathbf{E}[X_t^2] = O(1)$. ∎

4.1.2. Limiting phase. We assume that we are done with the learning phase and have $T(V)$ with the property given in Lemma 4.1: for every triangle $t \in T(V)$, $\mathbf{E}[X_t] = O(1)$ and $\mathbf{E}[X_t^2] = O(1)$. We have reached the limiting phase, where the algorithm is expected to compute the Delaunay triangulation with the optimal running time. We will prove the following lemma in this section.

LEMMA 4.2. *Using the data structures from the learning phase, and the properties of them that hold with probability at least $1 - 1/n^2$, in the limiting phase the Delaunay triangulation of input I can be generated in an expected $O(n + \sum_{i=1}^n H_i^V)$ time.*

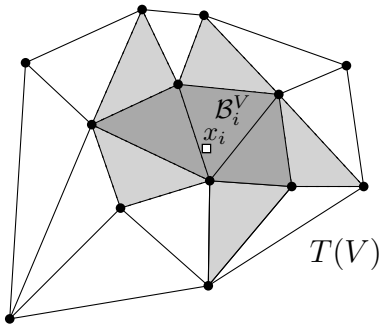

FIG. 4.3. *Determining the conflict set for x_i: the triangle $\mathcal{B}_i^V$ containing x_i is found via D_i. Then we perform a breadth-first search from $\mathcal{B}_i^V$ until we encounter triangles that no longer conflict with x_i. The dark gray triangles form the conflict set of x_i, and the light gray triangles mark the end of the breadth-first search. Since the conflict set S_i is connected, and since the dual graph has bounded degree, this takes $O(|S_i|)$ steps.*

The algorithm, and the proof of this lemma, has two steps. In the first step, $T(V)$ is used to quickly compute $T(V \cup I)$, with the time bounds of the lemma. In the second step, $T(I)$ is computed from $T(V \cup I)$, using a randomized splitting algorithm proposed by Chazelle et al. [21], who provide the following theorem.

THEOREM 4.3 (see [21, Theorem 3]). *Given a set of n points P and its Delaunay triangulation, for any partition of P into two disjoint subsets P_1 and P_2, the Delaunay triangulations $T(P_1)$ and $T(P_2)$ can be computed in $O(n)$ expected time, using a randomized algorithm.*

The remainder of this section is devoted to showing that $T(V \cup I)$ can be computed in an expected time $O(n + \sum_{i=1}^{n} H_i^V)$. The algorithm is as follows. For each $x_i \in I$, we use D_i to find the triangle $\mathcal{B}_i^V$ of $T(V)$ that contains it. By the properties of the D_i's as described in section 4.1.1, this takes $O(\sum_{i=1}^{n} H_i^V)$ expected time. We now need to argue that given the $\mathcal{B}_i^V$'s, the Delaunay triangulation $T(V \cup I)$ can be computed in expected linear time. For each x_i, we walk through $T(V)$ and find all the Delaunay disks of $T(V)$ that contain x_i, as in incremental constructions of Delaunay triangulations [12, Chapter 9]. This is done by breadth-first search of the dual graph of $T(V)$, starting from $\mathcal{B}_i^V$. Refer to Figure 4.3. Let S_i denote the set of triangles whose circumcircles contain x_i. We remind the reader that Z_t is the conflict set of triangle t.

CLAIM 4.4. *Given all $\mathcal{B}_i^V$'s, all S_i and Z_t sets can be found in expected linear time.*

Proof. To find all Delaunay disks containing x_i, do a breadth-first search from $\mathcal{B}_i^V$. For any triangle t encountered, check if C_t contains x_i. If it does not, then we do not look at the neighbors of t. Otherwise, add t to S_i and x_i to Z_t and continue. Since S_i is connected in the dual graph of $T(V)$,[5] we will visit all C_t's that contain x_i. The time taken to find S_i is $O(|S_i|)$. The total time taken to find all S_i's (once all the $\mathcal{B}_i^V$'s are found) is $O(\sum_{i=1}^{n} |S_i|)$. Define the indicator function $\chi(t, i)$ that takes value 1 if $x_i \in C_t$ and zero otherwise. We have

$$\sum_{i=1}^{n} |S_i| = \sum_{i=1}^{n} \sum_{t \in T(V)} \chi(t, i) = \sum_{t \in T(V)} \sum_{i=1}^{n} \chi(t, i) = \sum_{t} X_t.$$

[5] Since the triangles in S_i cover exactly the planar region of triangles incident to x_i in $T(V \cup \{x_i\})$.

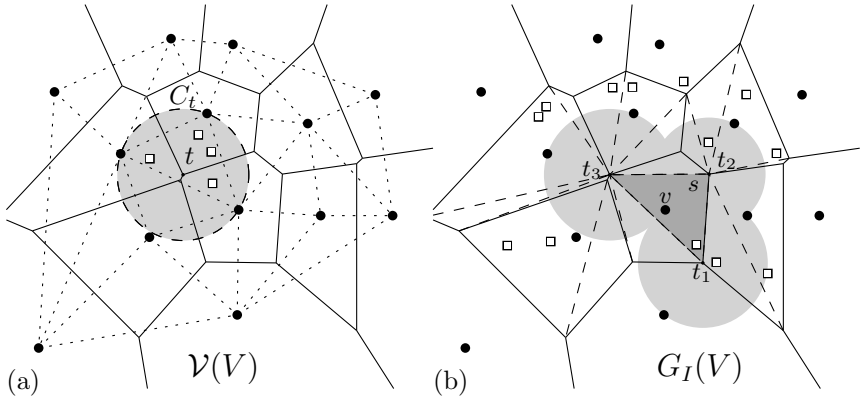

FIG. 4.4. (a) $\mathcal{V}(V)$ *is dual to* $T(V)$. *Each vertex* t *of* $\mathcal{V}(V)$ *corresponds to the center of the circumcircle of a triangle* t *of* $T(V)$, *and it has the same conflict set* Z_t *of size* X_t. (b) *The geode triangulation* $G_I(V)$ *is obtained by connecting the vertices of each region of* $\mathcal{V}(V)$ *to the lexicographically smallest incident vertex with the smallest* X_t. *The conflict set of a triangle* s *is the union of the conflict sets of its vertices and point* v *defining the region.*

Therefore, by Lemma 4.1,

$$\mathbf{E}\left[\sum_{i=1}^{n} |S_i|\right] = \mathbf{E}\left[\sum_t X_t\right] = \sum_t \mathbf{E}[X_t] = O(n).$$

This implies that all S_i's and Z_t's can be found in expected linear time. ∎

Our aim is to build the Delaunay triangulation $T(V \cup I)$ in linear time using the conflict sets Z_t. To that end, we will use divide-and-conquer to compute the *Voronoi diagram* $\mathcal{V}(V \cup I)$, using a scheme that has been used for nearest neighbor searching [24] and for randomized convex hull constructions [20, 25]. It is well known that the Voronoi diagram of a point set is dual to the Delaunay triangulation, and that we can go from one to the other in linear time [12, Chapter 9]. Refer to Figure 4.4(a). Consider the Voronoi diagram of V, $\mathcal{V}(V)$. By duality, the vertices of $\mathcal{V}(V)$ correspond to the triangles in $T(V)$, and we identify the two. In particular, each vertex t of $\mathcal{V}(V)$ has a conflict set Z_t, the conflict set for the corresponding triangle in $T(V)$, and $|Z_t| = X_t$, by our definition of X_t (see Figure 4.4(a)). We triangulate the Voronoi diagram as follows: for each region r of $\mathcal{V}(V)$, determine the lexicographically smallest Voronoi vertex t_r in r with minimum X_t. Add edges from all the Voronoi vertices in r to t_r. Since each region of $\mathcal{V}(V)$ is convex, this yields a triangulation[6] of $\mathcal{V}(V)$. We call it the *geode triangulation* of $\mathcal{V}(V)$ with respect to I, $G_I(V)$ [20, 24]. Refer to Figure 4.4(b). Clearly, $G_I(V)$ can be computed in linear time. We extend the notion of conflict set to the triangles in $G_I(V)$: Let s be a triangle in $G_I(V)$, and let t_1, t_2, t_3 be its incident Voronoi vertices. Then the conflict set of s, Z_s, is defined as $Z_s := Z_{t_1} \cup Z_{t_2} \cup Z_{t_3} \cup \{v\}$, where $v \in V$ is the point whose Voronoi region contains the triangle s. In the following, for any two points x and y, $|x - y|$ denotes the Euclidean distance between them.

[6] We need to be a bit careful when handling unbounded Voronoi regions: we pretend that there is a Voronoi vertex p_∞ at infinity which is the endpoint of all unbounded Voronoi edges, and when we triangulate the unbounded region, we also add edges to p_∞. By our bounding triangle assumption, there is no point in I outside the convex hull of V, and hence the conflict set of p_∞ is empty.

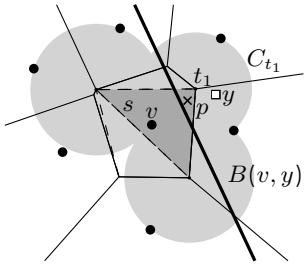

FIG. 4.5. *The nearest neighbor of a point $y \in s$ is either v or needs to be in the conflict set of one of its vertices.*

CLAIM 4.5. *Let s be a triangle of $G_I(V)$, and let Z_s be its conflict set. Then the Voronoi diagram of $V \cup I$ restricted to s, $\mathcal{V}(V \cup I) \cap s$, is the same as the Voronoi diagram of Z_s restricted to s, $\mathcal{V}(Z_s) \cap s$.*

Proof. Consider a point p in the triangle s, and let y be the nearest neighbor of p in $V \cup I$. If $y \in V$, then y has to be v, since s lies in the Voronoi region of v with respect to V. Now suppose that $y \in I$. Let $B(v, y)$ be the *perpendicular bisector* of the line segment (v, y) (i.e., the line containing all points in the plane that have equal distance from v and y). Refer to Figure 4.5. Let B^+ be the halfplane defined by $B(v, y)$ that contains y. Since B^+ intersects s, by convexity it also contains a vertex of s, say, t_1. Because t_1 and y are on the same side (B^+), $|y - t_1| < |v - t_1|$. Note that C_{t_1} has center t_1 and radius $|v - t_1|$, because t_1 is a vertex of the Voronoi region corresponding to v (in $\mathcal{V}(V)$). Hence, $y \in Z_{t_1}$. It follows that $y \in Z_s$, so $\mathcal{V}(V \cup I) \cap s = \mathcal{V}(Z_s) \cap s$, as claimed. ∎

Claim 4.5 implies that $\mathcal{V}(V \cup I)$ can be found as follows: for each triangle s of $G_I(V)$, compute $\mathcal{V}(Z_s) \cap s$, the Voronoi diagram of Z_s restricted to s. Then, traverse the edges of $G_I(V)$ and fuse the bisectors of the adjacent diagrams, yielding $\mathcal{V}(V \cup I)$.

LEMMA 4.6. *Given $\mathcal{V}(V)$, the Voronoi diagram $\mathcal{V}(V \cup I)$ can be computed in expected $O(n)$ time.*

Proof. The time to find $\mathcal{V}(Z_s) \cap s$ for a triangle s in $G_I(V)$ is $O(|Z_s| \log |Z_s|) = O(|Z_s|^2)$ [12, Chapter 7]. For a region r of $\mathcal{V}(V)$, let $S(r)$ denote the set of triangles of $G_I(V)$ contained in r, and let $E(r)$ denote the set of edges in $\mathcal{V}(V)$ incident to r. Recall that t_r denotes the common vertex of all triangles in $S(r)$. The total running time is $O(\mathbf{E}[\sum_{s \in G_I(V)} |Z_s|^2])$, which is proportional to

$$\mathbf{E}\left[\sum_{r \in \mathcal{V}(V)} \sum_{s \in S(r)} |Z_s|^2\right] \leq \mathbf{E}\left[\sum_{r \in \mathcal{V}(V)} \sum_{(t_1,t_2) \in E(r)} (1 + X_{t_r} + X_{t_1} + X_{t_2})^2\right]$$
$$\leq \mathbf{E}\left[\sum_{r \in \mathcal{V}(V)} \sum_{(t_1,t_2) \in E(r)} (1 + 2X_{t_1} + X_{t_2})^2\right],$$

since $X_{t_r} \leq \min(X_{t_1}, X_{t_2})$. For $e = (t_1, t_2)$, let $Y_e = 1 + 2X_{t_1} + X_{t_2}$. Note that $\mathbf{E}[Y_e] = O(1)$, by Lemma 4.1. We can write $Y_e = \sum_i (1/n + 2\chi(t_1, i) + \chi(t_2, i))$, where $\chi(t, i)$ was the indicator random variable for the event that $x_i \in C_t$. Hence, since

$1/n + 2\chi(t_1, i) + \chi(t_2, i) < 4$, Claim 3.3 implies that $\mathbf{E}[Y_e^2] = O(1)$. Thus,

$$\mathbf{E}\left[\sum_{s \in G_I(V)} |Z_s|^2\right] \leq \sum_{r \in \mathcal{V}(V)} \sum_{\substack{e \in E(r) \\ e=(t_1,t_2)}} \mathbf{E}[(Y_e)^2] = \sum_{r \in \mathcal{V}(V)} \sum_{\substack{e \in E(r) \\ e=(t_1,t_2)}} O(1).$$

The number of edges in $\mathcal{V}(V)$ is linear, and each edge e is incident to exactly two Voronoi regions r. Therefore, $\mathbf{E}[\sum_{s \in G_I(V)} |Z_s|^2] = O(n)$. Furthermore, assembling the restricted diagrams takes time $O\big(\mathbf{E}\big[\sum_{s \in G_I(V)} |Z_s|\big]\big)$, and as $|Z_s| \leq |Z_s|^2$, this is also linear. ∎

4.2. Running time analysis. In this section, we prove that the running time bound in Lemma 4.2 is indeed optimal. As discussed at the beginning of section 4, Claim 2.2 implies that any comparison-based algorithm for computing the Delaunay triangulation of input $I \in_R \mathcal{D}$ needs at least $H(T(I))$ expected comparisons. Recall that by Lemma 4.2, the expected running time of our algorithm is $O(n + \sum_i H_i^V)$. The following is the main theorem of this section.

THEOREM 4.7. *For H_i^V, the entropy of the triangle $\mathcal{B}_i^V$ of $T(V)$ containing x_i, and $H(T(I))$, the entropy of the Delaunay triangulation of I, considered as a labeled graph,*

$$\sum_i H_i^V = O(n + H(T(I))).$$

Proof. Let $\mathcal{B}^V := (\mathcal{B}_1^V, \ldots, \mathcal{B}_n^V)$ be the vector of all the triangles that contain the x_i's. By Claim 2.1, we have $H(\mathcal{B}^V) = \sum_i H_i^Y$. Now we apply Lemma 2.3 with $\mathcal{U} = \big(\mathsf{R}^2\big)^n$, $X = T(I)$, and Y. In Lemma 4.8 we will show that the function $f : (I, T(I)) \mapsto (\mathcal{B}_1^V, \ldots, \mathcal{B}_n^V)$ can be computed in linear time, so $H(\mathcal{B}_i^V) = O(n + H(T(I))$, by Lemma 2.3. This proves the theorem. ∎

We first define some notation. For a point set $P \subseteq V \cup I$ and $p \in P$, let $\Gamma_P(p)$ denote the neighbors of p in $T(P)$. It remains to prove the following lemma.[7]

LEMMA 4.8. *Given I and $T(I)$, for every x_i in I we can compute the triangle $\mathcal{B}_i^V$ in $T(V)$ that contains x_i in total expected time $O(n)$.*

Proof. First, we compute $T(V \cup I)$ from $T(V)$ and $T(I)$ in linear time [19, 36]. Thus, we now know $T(V \cup I)$ and $T(V)$, and we want to find for every point $x_i \in I$ the triangle $\mathcal{B}_i^V$ of $T(V)$ that contains it. For the moment, let us be a little less ambitious and try to determine, for each $x_i \in I$, a *conflict triangle* $\mathcal{C}_i^V$ in $T(V)$; i.e., $\mathcal{C}_i^V$ is a triangle t with $x_i \in Z_t$. If $x \in I$ and $v \in V$ such that $\overline{xv}$ is an edge of $T(V \cup I)$, we can find a conflict triangle for x in $T(V)$ in time $O(n)$ by inspecting all the incident triangles of v in $T(V)$. Actually, we can find conflict triangles for *all* neighbors of v in $T(V \cup I)$ that lie in I by merging the two neighbor lists (see below). Noting that on average the size of these lists will be constant, we could almost determine all the $\mathcal{C}_i^V$, except for one problem: there might be inputs $x \in I$ that are not adjacent to any $v \in V$ in $T(V \cup I)$. Thus, we need to dynamically modify $T(V)$ to ensure that there is always a neighbor present. Details follow.

CLAIM 4.9. *Let $p \in V \cup I$ and write $V_p := V \cup \{p\}$. Suppose that $T(V \cup I)$ and $T(V_p)$ are known. Then, in total time $O(|\Gamma_{V \cup I}(p)| + |\Gamma_{V_p}(p)|)$, for every $x_i \in \Gamma_{V \cup I}(p) \setminus V_p$, we can compute a conflict triangle $\mathcal{C}_i^{V_p}$ of x_i in $T(V_p)$.*

[7] A similar lemma is used in [22] in the context of hereditary algorithms for three-dimensional polytopes.

Proof. Let $x_i \in \Gamma_{V \cup I}(p) \setminus V_p$, and let $\mathcal{C}_i^{V_p}$ be the triangle of $T(V_p)$ incident to p that is intersected by line segment $\overline{px_i}$. We claim that $\mathcal{C}_i^{V_p}$ is a conflict triangle for x_i. Indeed, since $\overline{px_i}$ is an edge of $T(V \cup I)$, by the characterization of Delaunay edges (e.g., [12, Theorem 9.6(ii)]), there exists a circle C through p and x_i which does not contain any other points from $V \cup I$. In particular, C does not contain any other points from $V_p \cup \{x_i\}$. Hence $\overline{px_i}$ is also an edge of $T(V_p \cup \{x_i\})$, again by the characterization of Delaunay edges applied in the other direction. Therefore, triangle $\mathcal{C}_i^{V_p}$ is destroyed when x_i is inserted into $T(V \cup J)$ and is a conflict triangle for x_i in $T(V_p)$. It follows that the conflict triangles for $\Gamma_{V \cup I}(p) \setminus V_p$ can be computed by merging the cyclically ordered lists $\Gamma_{V \cup I}(p)$ and $\Gamma_{V_p}(p)$. This requires a number of steps that is linear of the size of the two lists, as claimed. ∎

For certain pairs of points p, x_i, the previous claim provides a conflict triangle $\mathcal{C}_i^{V_p}$. The next claim allows us to get $\mathcal{C}_i^V$ from this, which is what we wanted in the first place.

CLAIM 4.10. *Let $x_i \in I$, and let $p \in V \cup I$. Let $\mathcal{C}_i^{V_p}$ be the conflict triangle for x_i in $T(V_p)$ incident to p, as determined in Step 2(c). Then we can find a conflict triangle $\mathcal{C}_i^V$ for x_i in $T(V)$ in constant time.*

Proof. If $p \in V$, there is nothing to prove, so assume that $p \in I$. If $\mathcal{C}_i^{V_p}$ has all vertices in V, then it is also a triangle in $T(V)$, and we are trivially done. So assume that one vertex of $\mathcal{C}_i^{V_p}$ is p. Let e be the edge of $\mathcal{C}_i^{V_p}$ not incident to p, and let v, w be the endpoints of e. We will show that x_i is in conflict with at least one of the two triangles in $T(V)$ that are incident to e. Given e, such a triangle can clearly be found in constant time. Refer to Figure 4.6 for a depiction of the following arguments.

Since $v, w \in V$, by the characterization of Delaunay edges, it follows that e is also an edge of $T(V)$. If x_i does not lie in $\mathcal{C}_i^{V_p}$, then x_i must also be in conflict with the other triangle t that is incident to e (since t is intersected by the Delaunay edge $\overline{px_i}$). Note that t cannot have p as a vertex and is a triangle of $T(V)$.

Suppose x_i lies in $\mathcal{C}_i^{V_p}$. Since $\mathcal{C}_i^{V_p}$ is a triangle in $T(V_p)$, the interior has no points other than x_i. Thus, the segments $\overline{vx_i}$ and $\overline{wx_i}$ are edges of $T(V_p \cup \{x_i\})$. These must also be edges of $T(V \cup \{x_i\})$. But this means that x_i must conflict with the triangle in $T(V)$ incident to e at the same side as $\mathcal{C}_i^{V_p}$. ∎

Algorithm 1 Determining the conflict triangles.

1. Let Q be a queue containing the elements in V.
2. While $Q \neq \emptyset$:
   (a) Let p be the next point in Q.
   (b) If $p = x_i \in I$, then insert p into $T(V)$ using the conflict triangle $\mathcal{C}_i^V$ for x_i, to obtain $T(V_p)$. If $p \in V$, then $T(V_p) = T(V)$.
   (c) Using Claim 4.9, for each unvisited neighbor $x_j \in \Gamma_{V \cup I}(p) \cap I$, compute a conflict triangle $\mathcal{C}_j^{V_p}$ in $T(V_p)$.
   (d) For each unvisited neighbor $x_j \in \Gamma_{V \cup I}(p) \cap I$, using $\mathcal{C}_j^{V_p}$, compute a conflict triangle $\mathcal{C}_j^V$ of x_j in $T(V)$. Then insert x_j into Q, and mark it as visited.

The conflict triangles for all points in I can now be computed using breadth-first search (see Algorithm 1). The loop in Step 2 maintains the invariant that for each point $x_i \in Q \cap I$, a conflict triangle $\mathcal{C}_i^V$ in $T(V)$ is known. Step 2(b) is performed as in the traditional randomized incremental construction of Delaunay triangulations [12,

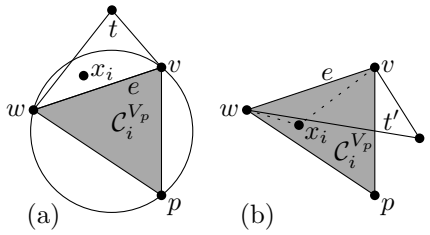

FIG. 4.6. (a) *If x_i is outside $\mathcal{C}_i^{V_p}$, it conflicts with the triangle t of $T(V)$ on the other side of e.* (b) *If x_i lies inside $\mathcal{C}_i^{V_p}$, it conflicts with the triangle t' of $T(V)$ at the same side of e, since $\overline{vx_i}$ and $\overline{wx_i}$ are both edges of $T(V)$.*

Chapter 9]: walk from $\mathcal{C}_i^V$ through the dual graph of $T(V)$ to determine the conflict set S_i of x_i (as in the proof of Claim 4.4), insert new edges from all points incident to the triangles in S_i to x_i, and remove all the old edges that are intersected by these new edges. The properties of the conflict set ensure that this yields a valid Delaunay triangulation. By Claim 4.10, Step 2(d) can be performed in constant time.

The loop in Step 2 is executed at most once for each $p \in V \cup I$. It is also executed at least once for each point, since $T(V \cup I)$ is connected and in Step 2(d) we perform a breadth-first search. The insertion in Step 2(b) takes $O(|\Gamma_{V_{x_i}}(x_i)|)$ time. Furthermore, by Claim 4.9, the conflict triangles of p's neighbors in $T(V \cup I)$ can be computed in $O(|\Gamma_{V_p}(p)| + |\Gamma_{V\cup I}(p)|)$ time. Finally, as we argued above, Step 2(d) can be carried out in total $O(|\Gamma_{V\cup I}(p)|)$ time. Now note that for $x_i \in I$, $|\Gamma_{V_{x_i}}(x_i)|$ is proportional to $|S_i|$, the number of triangles in $T(V)$ in conflict with x_i. Hence, the total expected running time is proportional to

$$\mathbf{E}\left[\sum_{p\in V\cup I}\left(|\Gamma_{V_p}(p)| + |\Gamma_{V\cup I}(p)|\right)\right] = \mathbf{E}\left[\sum_{v\in V}|\Gamma_V(v)| + \sum_{i=1}^{n}|S_i| + \sum_{p\in V\cup I}|\Gamma_{V\cup I}(p)|\right] = O(n).$$

Finally, using breadth-first search as in the proof of Claim 4.4, given the conflict triangles $\mathcal{C}_i^V$, the triangles $\mathcal{B}_i^V$ that contain the x_i's can be found in an $O(n)$ expected time, and the result follows. ∎

4.3. The time-space tradeoff. We show how to remove the assumption that we have prior knowledge of the $\mathcal{D}_i$'s (to build the search structures D_i) and prove the time-space tradeoff given in Theorem 1.3. These techniques are identical to those used in section 3.2. For the sake of clarity, we give a detailed explanation for this setting. Let $\varepsilon \in (0, 1)$ be any constant. The first $\lceil \log n \rceil$ rounds of the learning phase are used as in section 4.1.1 to construct the Delaunay triangulation $T(V)$. We first build a standard search structure D over the triangles of $T(V)$ [12, Chapter 6]. Given a point x, we can find the triangle of $T(V)$ that contains x in $O(\log n)$ time.

The learning phase takes $M = cn^{\varepsilon}$ rounds for some large enough constant c. The main trick is to observe that (up to constant factors), the only probabilities that are relevant are those that are at least $n^{-\varepsilon/3}$. In each round, for each x_i, we record the triangle of $T(V)$ that x_i falls into. Fix i, and for any triangle t of $T(V)$, let χ_t be the number of times over the first M rounds that $\mathcal{B}_i^V = t$. At the end of M rounds,

we take the set R_i of triangles t with $\chi_t > 0$. We remind the reader that $p(t,i)$ is the probability that x_i lies in triangle t. The proof of the following lemma is identical to the proof of Lemma 3.4.

LEMMA 4.11. *Fix i. With probability at least $1 - 1/n^3$, for every triangle t of $T(V)$, if $p(t,i) > n^{-\varepsilon/3}$, then $Mp(t,i)/2 < \chi_t < 3Mp(t,i)/2$.*

For every triangle t in R_i, we estimate $p(t,i)$ as $\hat{p}(t,i) = \chi_t/M$, and we use $\hat{p}(t,i)$ to build the approximate search structure D_i. For this, we take the planar subdivision G_i induced by the triangles in R_i, compute the convex hull of G_i, and triangulate the remaining polygonal facets. Then we use the construction of Arya et al. [10] to build an optimal planar point location structure D_i for G_i according to the distribution $\hat{p}_i$ (the triangles of G_i not in R_i are assigned probability 0). This structure G_i has the property that a point in a triangle t with probability $\hat{p}(t,i)$ can be located in $O(\log(1/\hat{p}(t,i)))$ steps [10, Theorems 1.1 and 1.2].

The limiting phase uses these structures to find $\mathcal{B}_i^V$ for every x_i: given x_i, we use D_i to search for it. If the search does not terminate in $\log n$ steps or D_i fails to find $\mathcal{B}_i^V$ (i.e., $\mathcal{B}_i^V \notin R_i$), then we use the standard search structure, D, to find $\mathcal{B}_i^V$. Therefore, we are guaranteed to find $\mathcal{B}_i^V$ in $O(\log n)$ time. Clearly, each D_i stores $O(M) = O(n^\varepsilon)$ triangles, so by the bounds given in [10], each D_i can be constructed with size $O(n^\varepsilon)$ in $O(n^\varepsilon \log n)$ time. Hence, the total space is bounded by $n^{1+\varepsilon}$ and the time required to build all the D_i's is $O(n^{1+\varepsilon} \log n)$.

Now we just repeat the argument given in section 3.2. Instead of doing it through words, we write down the expressions (for some variety). Let $s(t,i)$ denote the time to search for x_i given that $p(i,t) > n^{-\varepsilon/3}$. By Lemma 4.11, we have $\chi_t > Mn^{-\varepsilon/3}/2$, so $t \in R_i$, for c large enough, and thus $s(t,i) = O(\log(1/\hat{p}(t,i))) = O(1 - \log p(t,i))$. Thus,

$$\sum_{t:p(t,i)>n^{-\varepsilon/3}} p(t,i)s(t,i) = O\left(\sum_{t:p(t,i)>n^{-\varepsilon/3}} p(t,i)(1-\log p(t,i))\right)$$
$$= O\left(1 - \sum_{t:p(t,i)>n^{-\varepsilon/3}} p(t,i)\log p(t,i)\right).$$

We now bound the expected search time for x_i:

$$\sum_t p(t,i)s(t,i) = \sum_{t:p(t,i)\le n^{-\varepsilon/3}} p(t,i)s(t,i) + \sum_{t:p(t,i)>n^{-\varepsilon/3}} p(t,i)s(t,i)$$
$$= O\left(1 + \sum_{t:p(t,i)\le n^{-\varepsilon/3}} p(t,i)\log n - \sum_{t:p(t,i)>n^{-\varepsilon/3}} p(t,i)\log p(t,i)\right).$$

Noting that for $p(t,i) \le n^{-\varepsilon/3}$, we have $O(\log n) = O(\varepsilon^{-1}\log(1/p(t,i)))$, we get

$$\sum_t p(t,i)s(t,i)$$
$$= O\left(1 - \varepsilon^{-1}\sum_{t:p(t,i)\le n^{-\varepsilon/3}} p(t,i)\log p(t,i) - \sum_{t:p(t,i)>n^{-\varepsilon/3}} p(t,i)\log p(t,i)\right)$$
$$= O\left(1 - \varepsilon^{-1}\sum_t p(t,i)\log p(t,i)\right) = O(1 + \varepsilon^{-1}H_i^V).$$

If follows that the total expected search time is $O(n+\varepsilon^{-1}\sum_i H_i^V)$. By the analysis of section 4.1 and Theorem 4.7, we have that the expected running time in the limiting phase is $O(\varepsilon^{-1}(n+H(T(I))))$. If the conditions in Lemmas 4.1 and 4.11 do not hold, then the training phase fails. But this happens with probability at most $1/n$. This completes the proof of Theorem 1.3. ∎

5. Conclusions and future work. Our overall approach has been to deduce a "typical" instance for the distribution and then use the solution for the typical instance to solve the current problem. This is a very appealing paradigm; even though the actual distribution $\mathcal{D}$ could be extremely complicated, it suffices to learn just *one* instance. It is very surprising that such a single instance exists for product distributions. One possible way of dealing with more general distributions is to have a small set of typical instances. It seems plausible that even with two typical instances, we might be able to deal with some dependencies in the input.

We could imagine distributions that are very far from being generated by independent sources. Maybe we have a graph labeled with numbers, and the input is generated by a random walk. Here, there is a large dependency between various components of the input. This might require a completely different approach than the current one.

Currently, the problems we have focused upon already have $O(n \log n)$ time algorithms. So the best improvement in the running time we can hope for is a factor of $O(\log n)$. The entropy optimality of our algorithms is extremely pleasing, but our running times are always between $O(n)$ and $O(n \log n)$. It would be very interesting to get self-improving algorithms for problems where there is a much larger scope for improvement. Ideally, we want a problem where the optimal (or even best known) algorithms are far from linear. Geometric range searching seems to a good source of such problems. We are given some set of points, and we want to build data structures that answer various geometric queries about these points [2]. Suppose the points came from some distribution. Can we speed up the construction of these structures?

A different approach to self-improving algorithms would be to change the input model. We currently have a memoryless model, where each input is independently drawn from a fixed distribution. We could have a Markov model, where the input I_k depends (probabilistically) only on I_{k-1}, or maybe on a small number of previous inputs.

Appendix. Constructing the ε-net V. Recall that $\lambda = \lceil \log n \rceil$. Given a set $\hat{I}$ of $m := n\lambda$ points in the plane, we would like to construct a set $V \subseteq \hat{I}$ of size $O(n)$ such that any open disk C with $|C \cap \hat{I}| > \lambda$ intersects V. (This is a $(1/n)$-net for disks.) We describe how to construct V in deterministic time $n(\log n)^{O(1)}$, using a technique by Pyrga and Ray [43]. This is by no means the only way to obtain V. Indeed, it is possible to use the older techniques of Clarkson and Varadarajan [26] to get a another, randomized, construction with a better running time.

We set some notation. For a set of points S, a *k-set* of S is a subset of S of size k obtained by intersecting S with an open disk. A *$(> k)$-set* is is such a subset with size more than k. We give a small sketch of the construction. We take the collection $\hat{I}_{=\lambda}$ of all λ-sets of $\hat{I}$. We need to obtain a small hitting set for $\hat{I}_{=\lambda}$. To do this, we trim $\hat{I}_{=\lambda}$ to a collection of λ-sets that have small pairwise intersection. Within each such set, we will choose an ε-net (for some ε). The union of these ε-nets will be our final $(1/n)$-net. We now give the algorithmic construction of this set and argue that it is a $(1/n)$-net. Then, we will show that it has size $O(n)$.

It is well known that the collection $\hat{I}_{=\lambda}$ has $O(m\lambda)$ sets [25, 37] and that an explicit description of $\hat{I}_{=\lambda}$ can be found in time $O(m\lambda^2)$ [3, 37], since $\hat{I}_{=\lambda}$ corresponds to the λth-order Voronoi diagram of $\hat{I}$, each of whose cells represents some λ-set of $\hat{I}$ [37]. Let $\mathcal{I} \subseteq \hat{I}_{=\lambda}$ be a maximal subset of $\hat{I}_{=\lambda}$ such that for any $J_1, J_2 \in \mathcal{I}$, $|J_1 \cap J_2| \leq \lambda/100$. We will show in Claim A.1 how to construct $\mathcal{I}$ in $O(m\lambda^5)$ time. To construct V, take a (1/200)-net V_J for each $J \in \mathcal{I}$, and set $V := \bigcup_{J\in\mathcal{I}} V_J$.[8] It is well known that each V_J has constant size and can be found in time $O(|J|) = O(\lambda)$ [20, p. 180, Proof I]. The set V is an $(1/n)$-net for $\hat{I}$: if an open disk C intersects $\hat{I}$ in more than λ-points, by the maximality of $\mathcal{I}$, it must intersect a set $J \in \mathcal{I}$ in more than $\lambda/100$ points. Now V contains a (1/200)-net for J (recall that $|J| = \lambda$), so V must meet the disk C. We will argue in Claim A.2 that $|V| = O(n)$. This completes the proof. ∎

Claim A.1. *The set $\mathcal{I}$ can be constructed in time $O(m\lambda^5)$.*

Proof. We use a simple greedy algorithm. For each $J \in \hat{I}_{=\lambda}$, construct the collection $J_{>\lambda/100}$ of all $(> \lambda/100)$-sets of J. The set J has size λ, and the total number of disks defined by the points in J is at most λ^3. Thus, there are at most λ^3 sets in $J_{>\lambda/100}$, and they can all be found in $O(\lambda^4)$ time. Since there are at most $O(m\lambda)$ sets J (as we argued earlier), the total number of $(> \lambda/100)$-sets is $O(m\lambda^4)$, and they can be obtained in $O(m\lambda^5)$ time. Next, perform a radix sort on the multiset $\mathcal{J} := \bigcup_{J\in\hat{I}_{=\lambda}} J_{>\lambda/100}$. This again takes time $O(m\lambda^5)$. Note that for any $J_1, J_2 \in \hat{I}_{=\lambda}$, $|J_1 \cap J_2| > \lambda/100$ precisely if J_1 and J_2 share some $(> \lambda/100)$-set. Now $\mathcal{I}$ is obtained as follows: pick a set $J \in \hat{I}_{=\lambda}$, put J into $\mathcal{I}$, and use the sorted multiset $\mathcal{J}$ to find all $J' \in \hat{I}_{=\lambda}$ that share a $(> \lambda/100)$-set with J. Discard those J' from $\hat{I}_{=\lambda}$. Iterate until $\hat{I}_{=\lambda}$ is empty. The resulting set $\mathcal{I}$ has the desired properties. ∎

Claim A.2. $|V| = O(n)$.

Proof. The set V is the union of (1/200)-nets for each set $J \in \mathcal{I}$. Since each net has constant size, it suffices to prove that $\mathcal{I}$ has $O(n)$ sets. This follows from a charging argument due to Pyrga and Ray [43, Theorem 12]. They show [43, Lemma 7] how to construct a graph $G_{\mathcal{I}} = (\mathcal{I}, E_{\mathcal{I}})$ on vertex set $\mathcal{I}$ with at most $|E_{\mathcal{I}}| \leq 24|\mathcal{I}|$ edges with the following property: for $p \in \hat{I}$, let $\mathcal{I}_p$ be the set of all $J \in \mathcal{I}$ that contain p, and let $G_p = (\mathcal{I}_p, E_p)$ be the induced subgraph on vertex set $\mathcal{I}_p$. Then, for all p, $|E_p| \geq |\mathcal{I}_p|/4 - 1$. Thus,

$$\sum_{p\in\hat{I}} (|\mathcal{I}_p|/4 - |E_p|) \leq |\hat{I}| = m.$$

Consider the sum $\sum_{p\in\hat{I}} |\mathcal{I}_p|$. All sets in $\mathcal{I}$ contain exactly λ points, so each set contributes λ to the sum. By double counting, $\sum_{p\in\hat{I}} |\mathcal{I}_p|/4 = \lambda|\mathcal{I}|/4$. Furthermore, an edge $(J_1, J_2) \in E_{\mathcal{I}}$ can appear in E_p only if $p \in J_1 \cap J_2$, so again by double-counting,

$$\sum_{p\in\hat{I}} |E_p| \leq \lambda|E_{\mathcal{I}}|/100 \leq 24\lambda|\mathcal{I}|/100.$$

Hence, $m \geq \sum_{p\in\hat{I}} (|\mathcal{I}_p|/4 - |E_p|) \geq \lambda|\mathcal{I}|/100$, and $|\mathcal{I}| = O(m/\lambda) = O(n)$. ∎

[8] That is, V_J is a subset of J such that any open disk that contains more than $|J|/200$ points from J intersects V_J.

REFERENCES

[1] P. Afshani, J. Barbay, and T. M. Chan, *Instance-optimal geometric algorithms*, in Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2009, pp. 129–138.

[2] P. Agarwal and J. Erickson, *Geometric range searching and its relatives*, in Advances in Discrete and Computational Geometry, Contemp. Math. 223, AMS, Providence, RI, 1998, pp. 1–56.

[3] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor, *A linear-time algorithm for computing the Voronoi diagram of a convex polygon*, Discrete Comput. Geom., 4 (1989), pp. 591–604.

[4] S. Albers and M. Mitzenmacher, *Average case analyses of list update algorithms, with applications to data compression*, Algorithmica, 21 (1998), pp. 312–329.

[5] S. Albers and J. Westbrook, *Self-organizing data structures*, in Online Algorithms (Schloss Dagstuhl, 1996), Lecture Notes in Comput. Sci. 1442, Springer-Verlag, Berlin, 1998, pp. 13–51.

[6] B. Allen and I. Munro, *Self-organizing binary search trees*, J. ACM, 25 (1978), pp. 526–535.

[7] N. Alon and J. H. Spencer, *The Probabilistic Method*, 2nd ed., Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley-Interscience, New York, 2000.

[8] S. Ar, B. Chazelle, and A. Tal, *Self-customized BSP trees for collision detection*, Comput. Geom. Theory Appl., 15 (2000), pp. 91–102.

[9] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, Cambridge, UK, 2009.

[10] S. Arya, T. Malamatos, D. M. Mount, and K. C. Wong, *Optimal expected-case planar point location*, SIAM J. Comput., 37 (2007), pp. 584–610.

[11] J. L. Bentley and C. C. McGeoch, *Amortized analyses of self-organizing sequential search heuristics*, Comm. ACM, 28 (1985), pp. 404–411.

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

[13] J. R. Bitner, *Heuristics that dynamically organize data structures*, SIAM J. Comput., 8 (1979), pp. 82–110.

[14] J.-D. Boissonnat and M. Yvinec, *Algorithmic Geometry*, Cambridge University Press, Cambridge, UK, 1998.

[15] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, UK, 1998.

[16] K. Buchin and W. Mulzer, *Delaunay triangulations in $O(sort(n))$ time and more*, in Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2009, pp. 139–148.

[17] T. M. Chan and M. Pătraşcu, *Voronoi diagrams in $n2^{O(\sqrt{\lg \lg n})}$ time*, in Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), 2007, pp. 31–39.

[18] T. M. Chan and M. Pătraşcu, *Transdichotomous results in computational geometry,* I: *Point location in sublogarithmic time*, SIAM J. Comput., 39 (2009), pp. 703–729.

[19] B. Chazelle, *An optimal algorithm for intersecting three-dimensional convex polyhedra*, SIAM J. Comput., 21 (1992), pp. 671–696.

[20] B. Chazelle, *The Discrepancy Method*, Cambridge University Press, Cambridge, UK, 2000.

[21] B. Chazelle, O. Devillers, F. Hurtado, M. Mora, V. Sacristán, and M. Teillaud, *Splitting a Delaunay triangulation in linear time*, Algorithmica, 34 (2002), pp. 39–46.

[22] B. Chazelle and W. Mulzer, *Computing hereditary convex structures*, in Proceedings of the 25th Annual ACM Symposium on Computational Geometry (SoCG), 2009, pp. 61–70.

[23] B. Chazelle and W. Mulzer, *Markov incremental constructions*, Discrete Comput. Geom., 42 (2009), pp. 399–420.

[24] K. L. Clarkson, *A randomized algorithm for closest-point queries*, SIAM J. Comput., 17 (1988), pp. 830–847.

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

[26] K. L. Clarkson and K. Varadarajan, *Improved approximation algorithms for geometric set cover*, Discrete Comput. Geom., 37 (2007), pp. 43–58.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Cambridge, MA, 2009.

[28] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed., Wiley-Interscience, New York, 2006.

[29] H. Edelsbrunner and E. P. Mücke, *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms*, ACM Trans. Graph., 9 (1990), pp. 66–104.

[30] V. Estivill-Castro and D. Wood, *A survey of adaptive sorting algorithms*, ACM Comput. Surv., 24 (1992), pp. 441–476.
[31] M. L. Fredman, *How good is the information theory bound in sorting?*, Theoret. Comput. Sci., 1 (1975/76), pp. 355–361.
[32] G. H. Gonnet, J. I. Munro, and H. Suwanda, *Exegesis of self-organizing linear search*, SIAM J. Comput., 10 (1981), pp. 613–637.
[33] Y. Han, *Deterministic sorting in $O(n \log\log n)$ time and linear space*, J. Algorithms, 50 (2004), pp. 96–105.
[34] Y. Han and M. Thorup, *Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space*, in Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2002, pp. 135–144.
[35] J. H. Hester and D. S. Hirschberg, *Self-organizing linear search*, ACM Comput. Surv., 17 (1985), pp. 295–311.
[36] D. G. Kirkpatrick, *Efficient computation of continuous skeletons*, in Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 1979, pp. 18–27.
[37] D. T. Lee, *On k-nearest neighbor Voronoi diagrams in the plane*, IEEE Trans. Comput., 31 (1982), pp. 478–487.
[38] J. Matoušek, *Reporting points in halfspaces*, Comput. Geom. Theory Appl., 2 (1992), pp. 169–186.
[39] J. Matoušek, R. Seidel, and E. Welzl, *How to net a lot with little: Small ϵ-nets for disks and halfspaces*, in Proceedings of the 6th Annual ACM Symposium on Computational Geometry (SoCG), 1990, pp. 16–22.
[40] J. McCabe, *On serial files with relocatable records*, Operations Res., 13 (1965), pp. 609–618.
[41] K. Mehlhorn, *Data Structures and Algorithms* 1: *Sorting and Searching*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1984.
[42] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, UK, 1995.
[43] E. Pyrga and S. Ray, *New existence proofs for ϵ-nets*, in Proceedings of the 24th Annual ACM Symposium on Computational Geometry (SoCG), 2008, pp. 199–207.
[44] R. Rivest, *On self-organizing sequential search heuristics*, Comm. ACM, 19 (1976), pp. 63–67.
[45] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.
[46] D. D. Sleator and R. E. Tarjan, *Self-adjusting binary search trees*, J. ACM, 32 (1985), pp. 652–686.