

# Self-Improving Algorithms \*

Nir Ailon<sup>†</sup>      Bernard Chazelle<sup>†</sup>      Kenneth L. Clarkson<sup>‡</sup>      Ding Liu<sup>†</sup>  
Wolfgang Mulzer<sup>†</sup>      C. Seshadhri<sup>†</sup>

## Abstract

We investigate ways in which an algorithm can improve its expected performance by finetuning itself automatically with respect to an *arbitrary, unknown* input distribution. We give such *self-improving* algorithms for sorting and computing Delaunay triangulations. The highlights of this work: (i) an algorithm to sort a list of numbers with optimal expected limiting complexity; and (ii) an algorithm to compute the Delaunay triangulation of a set of points with optimal expected limiting complexity. In both cases, the algorithm begins with a training phase during which it adjusts itself to the input distribution, followed by a stationary regime in which the algorithm settles to its optimized incarnation.

## 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 BSP tree should be spawned by a Gaussian. Efforts have been made to analyze algorithms under more complex models (eg, Gaussian mixtures, Markov model outputs) but with limited success and lingering doubts about the choice of priors.

Ideally, one would like to compute a function  $f$  with the help of a *self-improving* algorithm. Upon receiving its first input instance  $I_0$ , such an algorithm would compute  $f(I_0)$  with, say, good worst-case guarantees and nothing more. Think of newly installed software that knows nothing about the user and runs in its “vanilla” configuration. Subsequently, as it is called upon to compute  $f(I_k)$  for  $k = 1, 2, \dots$ , the algorithm would gradually improve its performance through automatic finetuning. Intuitively, if the  $I_k$ ’s are drawn from a low-entropy distribution, the algorithm should be able to spot that and *learn* to be more efficient.

The obvious analogy is data compression, which seeks to exploit low entropy to minimize encoding size. The analog of Shannon’s noiseless coding theorem would be here: Given an *unknown* distribution  $\mathcal{D}$ , design a self-improving algorithm that converges to one with optimal expected running time. The second goal, which is to optimize the convergence speed, is more strictly speaking

---

\*This work was supported in part by NSF grants CCR-998817, 0306283, ARO Grant DAAH04-96-1-0181.

<sup>†</sup>Department of Computer Science, Princeton University

<sup>‡</sup>IBM Almaden Research Center

of a machine learning nature. One of the surprises of this work is how minimal distribution learning suffices for dramatic self-improvement.

The starting point of this research is the observation that, trimmed of noise, real-world data is often of much lower entropy than size alone suggests. For example, Takens’s embedding theorem asserts that univariate time series obtained from deterministic dynamical systems can be geometrized *canonically* as a (usually) low-dimensional attractor set in finite-dimensional space [31]. Hidden Markov models for speech recognition can be remarkably effective with only a few thousand states. Anecdotal evidence can also be gleaned from the current trend toward personalization in the design of web tools (search engines, recommendation systems, etc). Input data is often lodged in a tiny slice of input space that cannot be captured by closed-form distributions. To make predictions about the slice is the essence of machine learning [18, 23, 27]. To take computational advantage of the slice is what self-improving algorithms are all about.

**Our Results** The performance of a self-improving algorithm is measured with respect to an *unknown* memoryless random source  $\mathcal{D}$  of input instances. The algorithm is given instances  $I_0, I_1, \dots$  drawn independently from  $\mathcal{D}$ , which it must solve one at a time in batch mode with: (1) no prior knowledge of future instances, that is,  $f(I_k)$  must be computed before any of the  $I_j$ ’s ( $j > k$ ) are known; and (2) no prior knowledge of  $\mathcal{D}$ . The algorithm may store auxiliary information to help improve its performance. (Unlike self-organizing data structures, however, none of that information should be *necessary* for the algorithm to complete its task.) We use  $\mathcal{D}$  as shorthand for  $\mathcal{D}^n$ , the  $n$ -th member of an infinite ensemble of distributions—one for each input size. After a training phase, we expect the algorithm to settle into its steady state whose expected running time is called its limiting complexity. Note that from the user’s perspective the only difference noticeable in the training phase is that the system might be a little slower.

Our first result is, in some sense, the first truly optimal sorter. Given a source  $\mathcal{D}$  of real-number sequences  $I = (x_1, \dots, x_n)$ , let  $\pi(I)$  denote the permutation induced by the ranks of the  $x_i$ ’s, using the indices  $i$  to break ties. The complexity of our algorithm depends on the entropy  $H(\pi(I))$ . Note this quantity can be much smaller than the entropy of the source itself but can never exceed it.

- **SORTING:** We give a self-improving algorithm with a limiting complexity of  $O(n + H(\pi(I)))$  and prove that it is optimal. If the input  $I = (x_1, \dots, x_n)$  to be sorted is obtained by drawing each  $x_i$  independently (from a distribution that might depend on  $i$ ), then for any  $\varepsilon > 0$  the storage can be made  $O(n^{1+\varepsilon})$  for an expected running time of  $O(n + \varepsilon^{-1}H(\pi(I)))$ : this tradeoff is optimal for distributions of high enough entropy. The training takes  $O(n^\varepsilon \log n)$  rounds. We also show that independence, or at least some restriction on the distribution of  $I$ , is necessary: there are input distributions for which the storage *must* be exponential in  $n$ .

We take the concept 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.

- **DELAUNAY TRIANGULATIONS:** Assuming a distribution of  $n$  points, each one drawn independently from its own unknown (arbitrary) random source, we give a self-improving algorithm of optimal limiting complexity  $O(n + H(T(I)))$  for computing the Delaunay triangulation  $T(I)$  of input set  $I$ . We get time-space tradeoffs as well as lower bounds similar to those for sorting.

The optimality of these results follows from Shannon’s noiseless coding theorem, which states that any binary encoding of an information source such as  $\pi(I)$  must have an expected code length of least  $H(\pi(I))$  (and similarly for  $T(I)$  and  $H(T(I))$ ). Any comparison-based algorithm implies a coding scheme: the encoder sends the sequence of comparison outcomes, and the decoder simulates the algorithm, using the transmitted sequence to determine comparison outcomes. Thus any comparison-based algorithm must do an expected  $O(H(\pi(I)))$  comparisons, in addition to the  $\Theta(n)$  work needed to report the output.

**Previous Work** Related concepts 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 [2, 9, 22, 29, 30]. It is interesting to note that self-organizing data structures were investigated over stochastic input models first [1, 3, 8, 21, 25, 28]. It was the observation [7] 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: for example, hidden Markov models for gene finding or speech recognition or time-coherent models for self-customized BSP trees [5].

Algorithmic self-improvement differs from past work on self-organizing data structures and online computation in two fundamental ways: (i) self-improving algorithms operate offline and do not lend themselves to competitive analysis; (ii) they do not exploit structure within any given input but, rather, within the ensemble of input distributions. For example, suppose that the distribution  $\mathcal{D}$  consists of two random but *fixed* permutations, each one equally likely. Any solution in the adaptive, self-organizing/adjusting framework requires  $\Omega(n \log n)$  time. It is trivial, however, to design a self-improving algorithm of linear limiting complexity: sort the two permutations and store them; given any input instance, apply both permutations separately and output the permuted instance that is sorted.

Extensions of our memoryless model are easy to imagine. For example, a Bayesian version of self-improvement would postulate a prior and treat the  $I_k$ ’s as data conditioning a posterior distribution. One could also consider time-varying distributions or Markov models. Of course, a purely adversarial model might easily defeat self-improvement: it would observe how the improvement proceeds and render it ineffective by tailoring distributions changing over time. Memoryless sources are obviously the place to start any investigation on self-improvement. We also believe that the assumption is far less restrictive than for online computation. Take speech for example. The weakness of a memoryless model is that the next utterance is highly correlated with the previous ones: hence the use of Markov models. A self-improving algorithm would operate at the level of a sentence or a paragraph—not an utterance—where correlations are more diffuse and a memoryless source might be a good first approximation.

## 2 A Self-Improving Sorter

The self-improving sorter takes an input  $I = (x_1, x_2, \dots, x_n)$  of numbers drawn from a distribution  $\mathcal{D} = \prod_i \mathcal{D}_i$  (ie, 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 an information theoretic argument, it is easy to see that any sorter must take expected  $\Omega(H(\pi(I)) + n)$  comparisons. 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 (§2.1). Both assumptions are unrealistic, so we show how to remove them to produce a bona fide self-improving sorter (§2.2). The surprise is how strikingly little of the distribution needs to be learned for effective self-improvement.

**Theorem 2.1** *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  $O(n \log n)$ . If the input  $I = (x_1, \dots, x_n)$  to be sorted is obtained by drawing each  $x_i$  independently (from a distribution that might depend on  $i$ ), then for any  $\varepsilon > 0$  the storage can be made  $O(n^{1+\varepsilon})$  for an expected running time of  $O(n + \varepsilon^{-1}H(\pi(I)))$ : this tradeoff is optimal for distributions of high enough entropy. The algorithm reaches its steady state within  $O(n^\varepsilon \log n)$  rounds.*

REMARK: Much research has been done on adaptive sorting [19], especially on algorithms that exploit near-sortedness. Our approach is conceptually different. As we mentioned in the previous section, we seek to exploit properties, not of individual inputs, but of their distribution. In particular, our sorter runs in linear time for permutations drawn from a linear-entropy source, even though any individual input might be a *perfectly random* permutation. We are not aware of any previous algorithm that can achieve that.

Can we hope for a result similar to Theorem 2.1 if we drop the independence assumption? The short answer is no.

**Lemma 2.2** *There exists an input distribution  $\mathcal{D}$  such that any comparison-based algorithm that can sort a random input from  $\mathcal{D}$  in expected  $O(n + H(\pi(I)))$  time requires at least  $\Omega(2^{H(\pi(I))} n \log n)$  storage. This holds for any value of the entropy  $H(\pi(I))$  that is smaller than  $n \log n$  by a large enough constant factor.*

**Proof:** Consider the set of all  $n!$  permutations. Every subset  $\Pi$  of  $2^h$  permutations induces a distribution  $\mathcal{D}^\Pi$  defined by picking every permutation in  $\Pi$  with equal probability and none other. Note that the total number of distributions 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  $\Pi$ . Suppose there exists a comparison-based algorithm  $\mathcal{A}_\Pi$  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 of  $\Pi$  are sorted by  $\mathcal{A}_\Pi$  in at most  $2c(n+h)$  comparisons. But, within  $2c(n+h)$  comparisons, the algorithm  $\mathcal{A}_\Pi$  can sort a set  $P$  of at most  $2^{2c(n+h)}$  permutations. Therefore, any other  $\Pi'$  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  $\Pi'$  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 algorithms 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. An algorithm is entirely specified by a string of bits; therefore at least one such algorithm must require storage logarithmic in the previous bound.  $\square$

Fredman [20] gives a comparison-based algorithm that can optimally sort any distribution of permutations, but uses an exponentially large data structure to decide which comparisons to perform. This result shows that the storage used by Fredman's algorithm is essentially optimal.

## 2.1 Sorting with Full Knowledge

We consider the problem of sorting  $I = (x_1, \dots, x_n)$ , where each  $x_i$  is drawn from a distribution  $\mathcal{D}_i$ , which is specified by a vector  $(p_{i,1}, \dots, p_{i,N})$ , where  $p_{i,j} = \Pr[x_i = j]$ <sup>1</sup>. We can assume without loss of generality that all the  $x_i$ 's are distinct. (If not, simply replace  $x_i$  by  $nx_i + i - 1$  for tie-breaking purposes and enlarge  $N$  to  $n(N + 1)$ . All probabilities and entropies remain the same.)

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  $\lambda = c \log n$ , for large enough  $c$ , and sample  $\lambda$  input instances from  $\prod \mathcal{D}_i$ . Form their union and sort the resulting  $\lambda n$ -element multiset into a single list  $u_1 \leq \dots \leq u_{\lambda n}$ . Next, extract from it every  $\lambda$ -th item and form the list  $V = (v_0, \dots, v_{n+1})$ , where  $v_0 = 0$ ,  $v_{n+1} = \infty$ , and  $v_i = u_{i\lambda}$  for  $0 < i \leq n$ . Keep the  $V$ -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  $i > 0$ , let  $\mathcal{B}_i^V$  be the predecessor<sup>2</sup> 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 an optimum binary search tree [26] over the keys of the  $V$ -list, where the access probability of  $v_k$  is  $\sum_j \{p_{i,j} \mid v_k \leq j < v_{k+1}\}$ <sup>3</sup>, for any  $0 \leq k \leq n$ : the same distribution used to define  $H_i^V$ . This allows us to compute  $\mathcal{B}_i^V$  using  $O(H_i^V + 1)$  expected comparisons.

The total space used is  $O(n^2)$ . This can be decreased to  $O(n^{1+\epsilon})$  for any  $\epsilon > 0$ ; we describe how later. As we explained earlier, 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  $G_1 < G_2 < \dots$  of  $x_i$ 's sharing the same predecessor in the  $V$ -list. The next phase involves going through each  $G_j$  and sorting their elements naively, say using insertion sort. The first phase of the algorithm takes  $O(n + \sum_i H_i^V)$  expected time.<sup>4</sup> What about the second? Its complexity is  $O(n)$ , as follows from:

**Lemma 2.3** *With probability  $> 1 - n^{-2}$  over the construction of the  $V$ -list*

$$\mathbf{E}_{\mathcal{D}} [|\{i \mid v_k \leq x_i < v_{k+1}\}|^2] = O(1), \text{ for all } 0 \leq k \leq n.$$

**Proof:** Remember that the  $V$ -list was formed by taking certain elements from a list  $u_1 \leq \dots \leq u_{\lambda n}$ , where  $\lambda = c \log n$ . Consider two points  $u_i$  and  $u_j$ . Note that all the other  $\lambda n - 2$  points are independent of these two points. For every  $\ell \notin \{i, j\}$ , let  $Y_\ell^{(t)}$  be the indicator random variable

<sup>1</sup>All the arguments we give shall hold directly (and obviously) even if the  $\mathcal{D}_i$ 's are continuous. We have made this assumption for ease of presentation.

<sup>2</sup>Throughout this paper, the predecessor of  $y$  in a list refers to the index of the largest list element  $\leq y$ ; it does not refer to the element itself.

<sup>3</sup>If the  $\mathcal{D}_i$ 's were continuous, then this would be defined as the probability of  $x_j$  falling in  $[v_k, v_{k+1})$ .

<sup>4</sup>The  $H_i^V$ 's themselves are random variables depending on the choice of the  $V$ -list. Therefore, this is a conditional expectation.

for the event that  $u_\ell \in [u_i, u_j) = t$ . Let  $Y^{(t)} = \sum_\ell Y_\ell^{(t)}$ . Since all the  $Y_\ell^{(t)}$ 's are independent, by Chernoff's bound [4], for any  $\beta \in (0, 1]$ ,

$$\Pr[Y^{(t)} \leq (1 - \beta)\mathbf{E}[Y^{(t)}]] \leq e^{-\beta^2\mathbf{E}[Y^{(t)}]/2}.$$

With probability at least  $1 - n^{-4}$ , if  $\mathbf{E}[Y^{(t)}] > 4c \log n$ , then  $Y^{(t)} > 2c \log n$ . We can apply the same argument for any pair  $u_i, u_j$ . Taking a union bound over all pairs, we get that with probability  $> 1 - n^{-2}$ , if for the pair  $t$ ,  $\mathbf{E}[Y^{(t)}] > 4c \log n$ , then  $Y^{(t)} > 2c \log n$ .

The  $V$ -list is constructed such that for  $t_k = [v_k, v_{k+1})$ ,  $Y^{(t_k)} \leq c \log n$ . 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  $X^{(t_k)} = \sum_i X_i^{(t_k)} = |\{i \mid v_k \leq x_i < v_{k+1}\}|$ . Note that  $\mathbf{E}[Y^{(t_k)}] \geq (\log n - 2)\mathbf{E}[X^{(t_k)}]$  and therefore,  $\mathbf{E}[X^{(t_k)}] = O(1)$ . Now we apply the following standard claim to  $X^{(t_k)}$ :

**Claim 2.4** *Let  $Z = \sum_i Z_i$  be a sum of independent positive random variables with  $Z_i = O(1)$  for all  $i$  and  $\mathbf{E}[Z] = O(1)$ . Then  $\mathbf{E}[Z^2] = O(1)$ .*

**Proof:** By linearity of expectation,

$$\mathbf{E}[Z^2] = \mathbf{E}\left[\left(\sum_i Z_i\right)^2\right] = \sum_i \mathbf{E}[Z_i^2] + 2 \sum_{i < j} \mathbf{E}[Z_i]\mathbf{E}[Z_j] \leq \sum_i O(\mathbf{E}[Z_i]) + \left(\sum_i \mathbf{E}[Z_i]\right)^2 = O(1).$$

□  
□

We have shown that the algorithm takes  $O(n + \sum_i H_i^V)$  time (given a fixed  $V$ -list) plus an  $O(n)$  additive expected term (over  $V$  and  $\mathcal{D}$ ). We now show that this running time is indeed optimal.

**Lemma 2.5**

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

We will actually show this to be the case for *any* linear sized sorted list  $V$ . We will need a basic claim, which shall be proven for completeness, about the joint entropy of independent random variables.

**Claim 2.6** *Let  $H(Z_1, \dots, Z_n)$  be the joint entropy of independent random variables  $Z_1, \dots, Z_n$ . Then*

$$H(Z_1, \dots, Z_n) = \sum_i H(Z_i).$$

**Proof:** This is a consequence of the independence of the  $Z_i$ 's. We will prove this by induction over the number of variables the joint entropy includes. For the base case,  $H(Z_1) = H(Z_1)$  by definition. Assume inductively that  $H(Z_1, \dots, Z_k) = \sum_{i=1}^k H(Z_i)$ . By the chain rule for conditional entropy<sup>5</sup> and independence,

$$H(Z_1, \dots, Z_{k+1}) = H(Z_1, \dots, Z_k | Z_{k+1}) + H(Z_{k+1}) = \sum_{i=1}^{k+1} H(Z_i).$$

---

<sup>5</sup>Given two random variables  $X$  and  $Y$  over supports  $\mathcal{X}$  and  $\mathcal{Y}$ , the conditional entropy  $H(Y|X) = \sum_{x \in \mathcal{X}} \Pr(X = x)H(Y|X = x)$ . The chain rule tells us that  $H(Y, X) = H(Y|X) + H(X)$

□

Thus, it suffices to relate the joint entropy of  $\mathcal{B}^V := (\mathcal{B}_1^V, \dots, \mathcal{B}_n^V)$  with the entropy of  $\pi(I)$ . This is done via a neat trick:

**Claim 2.7** *Let  $\mathcal{D}$  be a distribution on a universe  $\mathcal{U}$ , and let  $X : \mathcal{U} \rightarrow \mathcal{X}$  and  $Y : \mathcal{U} \rightarrow \mathcal{Y}$  be two random variables. Suppose that the function  $f : \mathcal{U} \times X(\mathcal{U}) \rightarrow \mathcal{Y}$  defined by  $f : (I, X(I)) \mapsto Y(I)$  can be computed with  $O(n)$  expected comparisons (where the expectation is over  $\mathcal{D}$ ). Then  $H(Y) = O(n + H(X))$ , where all the entropies are with respect to  $\mathcal{D}$ .*

**Proof:** By a classical result from information theory (eg, [16, Theorem 5.4.1]), any unique encoding  $s : X(\mathcal{U}) \rightarrow \{0, 1\}^*$  of  $X(\mathcal{U})$  has expected code length  $E_{\mathcal{D}}[|s(X(I))|] \geq H(X)$ , and there exists an encoding  $s^*$  that has expected code length  $O(H(X))$ . Using  $f$ , this can be converted into an encoding  $t$  of  $Y(\mathcal{U})$ . Indeed, for every  $I$ ,  $Y(I)$  can be uniquely identified using  $s^*(X(I))$  and additional bits that represent the outcomes of the comparisons for the computation of  $f(I, X(I))$ . By taking a shortest such string for each element of  $Y(\mathcal{U})$ , we obtain a unique encoding  $t$  for  $Y(\mathcal{U})$  with expected code length  $E_{\mathcal{D}}[|t(Y(I))|] = O(n + E_{\mathcal{D}}[|s(X(I))|]) = O(n + H(X))$ . Since any encoding of  $Y(\mathcal{U})$  has expected code length at least  $H(Y)$ , the claim follows. □

**Lemma 2.8**

$$H(\mathcal{B}^V) = O(n + H(\pi(I))).$$

**Proof:** Apply Claim 2.7 with  $\mathcal{U} = \{1, \dots, n\}$ ,  $X(I) = \pi(I)$ , the permutation induced by input  $I$ , and  $Y(I) = \mathcal{B}^V$ . The function  $f$  can be computed in linear time by using  $\pi(I)$  to sort  $I$  and then merging this sorted list with  $V$ . □

Lemma 2.5 now follows from Lemmas 2.6 and 2.8. This completes the proof for the optimality of the time taken by the sorter. We now show that the storage can be reduced to  $O(n^{1+\varepsilon})$ , for any  $\varepsilon > 0$ . The main idea is to prune each of the  $D_i$  trees to depth  $\varepsilon \log n$ . This ensures that each of these trees has size  $O(n^\varepsilon)$  and 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 that it would have taken 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  $> \varepsilon \log n$  (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  $\varepsilon > 0$ ).

We can show that the storage cannot be reduced to linear. In fact, the 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 2.9** *For any  $c$  large enough and any  $h \leq \frac{3}{c} n \log n$ , there is a distribution  $\mathcal{D} = \prod_i \mathcal{D}_i$  of entropy  $h$  such that any comparison-based algorithm that can sort a random permutation from  $\mathcal{D}$  in expected time  $c(h + n)$  requires a data structure of bit size  $\Omega(2^{h/n} n \log n)$ .*

**Proof:** The proof is a specialization of the argument used for proving Lemma 2.2. Let  $\kappa = 2^{\lfloor h/n \rfloor}$ . We define  $\mathcal{D}_i$  by choosing  $\kappa$  distinct integers in  $[1, n]$  and making them equally likely to be picked as  $x_i$ . This leads to  $\binom{n}{\kappa}^n > (n/\kappa)^{\kappa n}$  choices of distinct distributions  $\mathcal{D}$ . Suppose that there is a data structure of size  $s$  that can accommodate any such distribution with an expected running time of at most  $c(h+n)$ . Then one such data structure  $\mathcal{W}$  must be able to accommodate this running time for a set  $\mathcal{G}$  of at least  $(n/\kappa)^{\kappa n} 2^{-s}$  distributions  $\mathcal{D}$ . Any input instance that is sorted in at most  $2c(h+n)$  time by this data structure is called *easy*: the set of easy instances is denoted by  $\mathcal{E}$ .

Each  $\mathcal{D}_i$  is characterized by a vector  $v_i = (a_{i,1}, \dots, a_{i,\kappa})$ , so that  $\mathcal{D}$  itself is specified by  $v = (v_1, \dots, v_n) \in \mathbb{R}^{n\kappa}$ . (From now on, we view  $v$  both as a vector and a distribution of input instances.) Define the  $j$ -th projection of  $v$  as  $v^j = (a_{1,j}, \dots, a_{n,j})$ . 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  $v_i = (a_{i,1}, \dots, a_{i,\kappa})$  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  $v'_i = (a_{i,1}, \dots, a_{i,\kappa})$  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

$$|\mathcal{E}| \geq n^n \kappa^{-2n} 2^{-2s/\kappa}. \quad (1)$$

In a comparison-based decision tree model, each input instance is associated with the leaf of a binary decision tree of depth at most  $2c(h+n)$ , ie, with one with at most  $2^{2c(h+n)}$  leaves. This would give us a lower bound on  $s$  if each instance was assigned a distinct leaf. But this may not be the case. However, we have a *collision* bound, saying that at most  $4^n$  instances can be mapped to the same leaf. This implies that  $|\mathcal{E}| 4^{-n} \leq 2^{2c(h+n)}$ ; and by (1),  $s = \Omega(\kappa n \log n)$ ; hence the lemma.

To prove the collision bound, we use the tie-breaking rule mentioned earlier:  $x_i \mapsto nx_i + i - 1$ . It is clear that two instances mapping to two distinct permutations must lead to two different leaves of the decision tree. So the only question left is to bound the number of instances mapping to a given permutation. Let  $x = (x_1, \dots, x_n)$  be an input instance (no tie-breaking). Represent the ground set of this instance as an  $n$ -bit vector  $\alpha$  ( $\alpha_i = 1$  if some  $x_j = i$ , else  $\alpha_i = 0$ ). Let  $x$  be sorted to give the vector  $y = (y_1, \dots, y_n)$ . For  $i = 2, \dots, n$ , let  $\beta_i = 1$  if  $y_i = y_{i-1}$ , else  $\beta_i = 0$ . Given the vectors  $\alpha, \beta$  and the induced permutation, the input instance  $x$  can be recovered. This proves the collision bound.  $\square$

## 2.2 Learn & Sort

The  $V$ -list is built in the first  $O(\log n)$  rounds. The  $D_i$ -trees will be built after  $O(n^\epsilon \log n)$  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 per se consists of learning basic information about  $H_i^V$  for each  $i$ . For notational simplicity, fix  $i$  and let  $p_k = \Pr_{\mathcal{D}_i} [v_k \leq y < v_{k+1}]$ . Let  $M = cn^\epsilon \log n$ , for a large enough constant  $c$ . For any  $k$ , let  $\chi_k$  be the number of times, over

the first  $M$  rounds, that  $v_k$  is found to be the  $V$ -list predecessor of some  $x_i$ . (We use 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  $\chi_k > Mn^{-\varepsilon}$ . Recall that the defining property of such a tree is that the node associated with a key of weight  $\chi_k$  is at depth  $O(\log \chi/\chi_k)$ , where  $\chi = \sum \chi_k$ . We apply this procedure for each  $i = 1, \dots, n$ .

This  $D_i$ -tree is essentially the pruned version of the one mentioned earlier. Like before, its size is  $O(M/(Mn^{-\varepsilon})) = O(n^\varepsilon)$ . The way we use it is similar to what we described, with a few minor differences. For completeness, we go over it again: given  $x_i$ , we perform a binary search down the  $D_i$ -tree, stopping as soon as we encounter a node whose associated key  $v_k$  is such that  $x_i \in [v_k, v_{k+1})$ , in which case we have the predecessor of  $x_i$  in the  $V$ -list and we are done. If we reach the bottom of the  $D_i$ -tree without success, we simply perform a standard binary search in the  $V$ -list.

**Lemma 2.10** *Fix  $i$ . With probability at least  $1 - 1/n^2$ , for any  $k$ ,  $p_k > n^{-\varepsilon}$  implies that  $Mp_k/2 < \chi_k < 3Mp_k/2$ .*

**Proof:** The expected value of  $\chi_k$  is  $Mp_k$ . If  $p_k = \Omega(n^{-\varepsilon})$  then, by Chernoff's bound [4] (pages 267–268), the count  $\chi_k$  deviates from its expectation by more than  $a = Mp_k/2$  with probability less than

$$e^{-a^2/(2p_k M) + a^3/(2p_k^2 M^2)} + e^{-a^2/(2p_k M)} < n^{-b},$$

for some constant  $b$  growing linearly with  $c$ . A union bound (over all  $k$ ) completes the proof.  $\square$

Suppose the condition of the lemma holds for each  $k$  (and fixed  $i$ ). We show now that the expected search time is  $O(\varepsilon^{-1} H_i^V + 1)$ . Consider each element in the sum  $H_i^V = \sum_k p_k \log p_k^{-1}$ .

- $p_k > n^{-\varepsilon}$ : if  $v_k$  is in the  $D_i$ -tree, then the cost of the search is  $O(\log \chi/\chi_k)$ , so its contribution to the expected running time is  $O(p_k \log \chi/\chi_k)$ . By the lemma, this is also  $O(p_k(1 + \log p_k^{-1}))$ , as desired. If  $v_k$  is not in the  $D_i$ -tree, then the search is unsuccessful and costs  $O(\log n)$  time: its contribution to the expected running time is  $O(p_k \log n)$ . Not being in the tree, however, means that  $\chi_k \leq Mn^{-\varepsilon}$ ; hence  $p_k < 2n^{-\varepsilon}$  and the contribution is  $O(\varepsilon^{-1} p_k \log p_k^{-1})$ .
- $p_k \leq n^{-\varepsilon}$ : the search time is always  $O(\log n)$  time; hence the contribution to the expected running 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 conditions of the lemma. But these are satisfied for all  $i$  with probability at least  $1 - 1/n$ . This leaves a probability  $1/n$  that the training fails and we are stuck with  $\Theta(n \log n)$  sorting—note that we do not try to detect failure. But this adds only an additive sublinear term to the expected complexity and is therefore negligible.

### 3 Delaunay Triangulations

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, and so the input  $I$  is drawn from the product distribution  $\mathcal{D} = \prod_i \mathcal{D}_i$ . In each round, a new input  $I$  is drawn from  $\mathcal{D}$ ,

and we wish to compute the Delaunay triangulation of  $I$ . We are in the comparison model, so any operation consists of evaluating a polynomial at some point (more details about this are given in §3.2). Although it is not critical, for the sake of simplicity, we will assume that the points of  $I$  are in general position, which is true with probability one when all the  $\mathcal{D}_i$ 's are continuous. Also we will assume that there is a bounding triangle such that all points always lie inside this triangle.

The distribution  $\mathcal{D}$  also induces a (discrete) distribution on the set of Delaunay triangulations, viewed as labelled graphs on the vertex set  $[1, n]$ . Consider the entropy of this distribution: for each graph  $G$  on  $[1, n]$ , let  $p_G$  be the probability that it represents the Delaunay triangulation of  $I \in_R \mathcal{D}$ . Abusing notation, let the output entropy  $H(T(I)) := -\sum_G p_G \log p_G$ . By information-theoretic arguments, this quantity is a lower bound on the expected time required by any comparison-based algorithm to compute the Delaunay triangulation of  $I \in_R \mathcal{D}$ . An *optimal* algorithm will be one that has an expected running time of  $O(n + H(T(I)))$ . Our main result is the following:

**Theorem 3.1** *For inputs  $(x_1, x_2, \dots, x_n)$  drawn from the product distribution  $\mathcal{D} = \prod_i \mathcal{D}_i$ , and for any constant  $\varepsilon > 0$ , there is a self-improving algorithm for finding the Delaunay triangulations of the  $x_i$  that has a learning phase of  $O(n^\varepsilon)$  rounds and uses  $O(n^{1+\varepsilon})$  space<sup>6</sup>. The limiting running time is  $O(\varepsilon^{-1}(n + H(T(I))))$ , and therefore optimal.*

From the linear time reduction of sorting to computing Delaunay triangulations, the lower bounds for sorting carry over to Delaunay triangulations. As an immediate corollary of Lemma 2.2, we get

**Corollary 3.2** *There exists an input distribution  $\mathcal{D}$  such that any self-improving algorithm computing the Delaunay triangulation of inputs from  $\mathcal{D}$  in  $O(n + H(T(I)))$  limiting running time requires  $\Omega(2^n)$  space.*

Furthermore, by Lemma 2.9, the time-space tradeoff we provide is essentially optimal.

### 3.1 The algorithm

We describe the algorithm in two parts. The first part explains the learning phase and the data structures that are constructed (§3.1.1). Then, we explain the how these data structures are used to speed up the computation in the limiting phase (§3.1.2). 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 (§3.2), 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 3.3 repeats the arguments of Section 2.2 to give the space-time tradeoff bounds of Theorem 3.1.

As outlined in Figure 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.

---

<sup>6</sup>The total time required for the learning phase is also  $O(n^{1+\varepsilon})$ .

Sorting	Delaunay Triangulation
Intervals $(x_i, x_{i'})$ containing no values of $I$	Delaunay disks
Typical set $V$	Range space $\varepsilon$ -net $V$ [15, 24], 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 $T_i$	Entropy-optimal planar point location data structures $T_i$ [6]
Sorting within buckets	Triangulation within $\mathcal{V}(Z_s) \cap s$ (Claim 3.8)
Sorted list of $V \cup I$	$T(V \cup I)$
Build sorted $V$ from sorted $V \cup I$ (trivial)	Build $T(I)$ from $T(V \cup I)$ [12]
(analysis) merge sorted $V$ and $I$	(analysis) merge $T(V)$ and $T(I)$ [10]
(analysis) recover indices $\mathcal{B}_i^V$ from sorted $I$ (trivial)	(analysis) recover triangles $\mathcal{B}_i^V$ in $T(V)$ from $T(I)$ (Lemma 3.11)

Figure 1: Delaunay triangulation algorithm as a generalization of the sorting algorithm

### 3.1.1 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  $\lambda := c \log n$  input lists  $I_1, I_2, \dots, I_\lambda$ , where  $c$  is a sufficiently large constant. Merge them into one list  $\hat{I}$  of  $\lambda n = cn \log n$  points. Setting  $\varepsilon := 1/n$ , find an  $\varepsilon$ -net  $V \subseteq \hat{I}$  for the set of all open disks. In other words, find a set  $V$  such that for any open disk  $C$  that contains more than  $\varepsilon \lambda n = c \log n$  points of  $\hat{I}$ ,  $C$  contains at least one point of  $V$ . Matousek *et al.* [24] show that there exist  $\varepsilon$ -nets of size  $O(1/\varepsilon)$  for disks, which here is  $O(n)$ . Furthermore, a construction and analysis similar to that of Clarkson and Varadarajan [15] yields a randomized construction (with polynomially small error probability) that takes  $n(\log n)^{O(1)}$  time.

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 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. Define the random variable  $\mathcal{B}_i$  to be the triangle of  $T(V)$  that  $x_i$  falls into<sup>7</sup>. Now let the entropy of  $\mathcal{B}_i$  be  $H_i^V$ . If the probability that  $x_i$  falls in 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  $\mathcal{B}_i$  in expected  $O(H_i^V)$  time. These  $D_i$ 's can be constructed using the results of Arya *et al.* [6], for which the number of primitive comparisons is  $H_i^V + o(H_i^V)$ . These correspond to the  $D_i$ -trees used for sorting.

We will show 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; this 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  $x_i$ . (The ‘‘conflict’’ terminology arises from the fact that if  $x_i$  were added to  $V$ , triangles with

<sup>7</sup>Assume that we add the vertices of the bounding triangle to  $V$ . This will ensure that  $x_i$  will always fall in some triangle  $\mathcal{B}_i$ .

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  $X_t := |Z_t|$ . Note that the randomness comes from the random distribution of  $\hat{I}$ , and so  $V$  and  $T(V)$ , 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  $\mathcal{D}$ .

**Claim 3.3** *With probability at least  $1 - n^{-3}$  over the construction of  $T(V)$ , for every triangle  $t$  of  $T(V)$ ,  $\mathbf{E}[X_t] = O(1)$  and  $\mathbf{E}[X_t^2] = O(1)$ .*

**Proof:** This is similar to the argument given in Lemma 2.3 with a geometric twist. Let the list of points  $\hat{I}$  be  $s_1, \dots, s_{\lambda n}$ , the concatenation of  $I_1$  through  $I_\lambda$ . Consider the triangle  $t$  with vertices  $s_1, s_2, s_3$ . Note that all the remaining  $\lambda n - 3$  points are chosen independently of these three, from some distribution  $\mathcal{D}_j$ . For each  $j \in [4, \lambda n]$ , let  $Y_j^{(t)}$  be the indicator variable for the event that  $s_j$  is inside  $C_t$ . Let  $Y^{(t)} = \sum_j Y_j^{(t)}$ . By the Chernoff bound, for any  $\beta \in (0, 1]$ ,

$$\Pr [Y^{(t)} \leq (1 - \beta)\mathbf{E}[Y^{(t)}]] \leq e^{-\beta^2 \mathbf{E}[Y^{(t)}]/2}.$$

Setting  $\beta = 1/2$ , if  $\mathbf{E}[Y^{(t)}] > 48 \log n$ , then  $Y^{(t)} > 24 \log n$  with probability at least  $1 - n^{-6}$ . We can now consider any triangle generated by some triple of points  $s_i, s_j, s_k$ , for  $i, j, k \in [4, \lambda n]$ , and apply the same argument as above. Taking a union bound over all triples of the points in  $\hat{I}$ , we obtain that with probability at least  $1 - n^{-3}$ , for any triangle  $t$  generated by the points of  $\hat{I}$ , if  $\mathbf{E}[Y^{(t)}] > 48 \log n$ , then  $Y^{(t)} > 24 \log n$ . 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  $c \log n$  points of  $\hat{I}$ , that is,  $Y^{(t)} \leq c \log n$ . This implies that  $\mathbf{E}[Y^{(t)}] = O(\log n)$ , as in the previous paragraph. Since  $\mathbf{E}[Y^{(t)}] > (\log n - 3)\mathbf{E}[X_t]$ , 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,  $\mathbf{E}[X_t^2] = O(1)$ , by Claim 2.4.  $\square$

### 3.1.2 Limiting Phase

We assume that we are done with learning phase, and have  $T(V)$  with the property given in Claim 3.3: 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 3.4** *Using the data structures from the learning phase, and the properties of them that hold with probability  $1 - O(1/n)$ , in the limiting phase the Delaunay triangulation of input  $I$  can be generated in expected  $O(n + \sum_{i=1}^n H_i^V)$  time.*

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.* [12], whose Theorem 3 is as follows.

**Theorem 3.5** *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.*

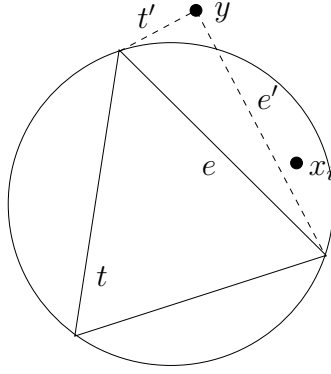


Figure 2: Proof of Claim 3.6

The remainder of the proof of the lemma, and of this subsection, is devoted to showing that  $T(V \cup I)$  can be computed in the time bound of the lemma. The algorithm is as follows. For each  $x_i$ , we use  $D_i$  to find the triangle  $t_i$  of  $T(V)$  that contains it. By the arguments given in the previous section, this takes time  $O(\sum_{i=1}^n H_i^V)$ . We now need to argue that given the  $t_i$ 's, the Delaunay triangulation  $T(I)$  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. This is done by breadth-first search of the dual graph of  $T(V)$ , starting from  $t_i$ . Let  $S_i$  denote the set of circumcircles containing  $x_i$ . The following standard claim implies that this procedure will work.

**Claim 3.6** *The set of  $t \in T(V)$  with  $C_t \in S_i$  is a connected set in the dual graph of  $T(V)$ .*

**Proof:** Consider some triangle  $t$  with  $C_t \in S_i$ . We will show that  $t$  is connected to  $t_i$  by a path in the dual graph of  $T(V)$ . Consider the edge  $e$  such that  $x_i$  is in the sector bounded by  $C_t$  and  $e$ . Let  $t'$  be the neighbor of  $t$  adjacent to  $e$ . Note that since  $C_t$  is a Delaunay triangle,  $t' \in S_i$ . If  $t'$  is  $t_i$ , we are done. If not, then consider the edge  $e'$  such that  $x_i$  is in the sector bounded by  $C_{t'}$  and  $e'$ . Refer to Figure 2. The edge  $e'$  is closer to  $x_i$  than  $e$ . We now consider the neighbor of  $t'$  adjacent to  $e'$  and continue in this manner. Eventually, we must reach  $t_i$  by a connected path in the dual graph of  $T(V)$ .  $\square$

**Claim 3.7** *Given all  $t_i$ 's, all  $S_i$  and  $Z_t$  sets can be found in expected linear time.*

**Proof:** To find all circles containing  $x_i$ , do a breadth-first search from  $t_i$ . For any triangle  $t$  encountered, check if  $C_t$  contains  $x_i$ . If it does not, then we do not look at the neighbours of  $t$ . Otherwise, add  $C_t$  to  $S_i$  and  $x_i$  to  $Z_t$  and continue. By Claim 3.6, 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  $t_i$ '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

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

Therefore, by Claim 3.3,

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

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

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 [13] and for randomized convex hull constructions [11, 14]. 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. Consider the Voronoi diagram of  $V$ ,  $\mathcal{V}(V)$ . By duality, the nodes of  $\mathcal{V}(V)$  correspond to the triangles in  $T(V)$ , and we identify the two. In particular, each node  $t$  of  $\mathcal{V}(V)$  has a conflict set  $Z_t$ , the conflict set for the corresponding triangle in  $T(V)$ , and  $|Z_t| = X_t$ . We triangulate the Voronoi diagram: for each region  $r$  of  $\mathcal{V}(V)$ , determine the lexicographically smallest Voronoi node  $t_r$  in  $r$  with minimum  $X_t$ . Add edges from all the Voronoi nodes in  $r$  to  $t_r$ . Since each region of  $\mathcal{V}(V)$  is convex, this yields a triangulation of  $\mathcal{V}(V)$ . We call it the *geode triangulation* of  $\mathcal{V}(V)$  with respect to  $I$ ,  $G_I(V)$ <sup>8</sup>. 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 nodes. 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  $s$  (See, for example Figure 2 of [13]).

**Claim 3.8** *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:** Recall that the Voronoi diagram of a set of points  $P \subseteq \mathbb{R}^2$  can be considered as the  $xy$ -projection of the upper envelope of a set  $H_P$  of hyperplanes in  $\mathbb{R}^3$ , where  $H_P$  contains a hyperplane  $h_p$  for each point  $p \in P$ . The hyperplane  $h_p$  is obtained by lifting the point  $p = (x, y)$  to the point  $\tilde{p} := (x, y, x^2 + y^2)$  on the unit paraboloid and taking the hyperplane  $h_p$  tangent to the unit paraboloid in  $\tilde{p}$ . The hyperplane  $h_p$  is called the *lifted hyperplane* for  $p$ . Each Voronoi node  $x$  of  $\mathcal{V}(P)$  corresponds to the intersection  $\tilde{x}$  of three hyperplanes in  $H_P$ . Furthermore, a point  $q \in \mathbb{R}^2$  is in conflict with Voronoi node  $x$  if and only if the lifted hyperplane  $h_q$  for  $q$  is intersected by the infinite upward ray extending from  $\tilde{x}$ .

Now consider  $\mathcal{V}(V)$  as an upper envelope of the set of lifted hyperplanes  $H_V$ . Let  $v \in V$  be the point whose Voronoi region contains  $s$ . Then  $s$  is the  $xy$ -projection of a triangle  $\tilde{s}$  contained in the lifted hyperplane  $h_v$  for  $v$ . Let  $Q$  be the unbounded polytope obtained by extending  $\tilde{s}$  in positive  $z$ -direction. Then  $\mathcal{V}(V \cup I) \cap s$  corresponds to the intersection of the upper envelope of  $H_I$  with  $Q$ . But each hyperplane in  $H_I$  that intersects  $Q$  also intersects at least one infinite upward ray extending from a vertex of  $\tilde{s}$ <sup>9</sup>, and hence all the relevant hyperplanes correspond to points in  $Z_s$ .  $\square$

<sup>8</sup>We need to be a bit careful when handling unbounded Voronoi regions: we pretend that there is a Voronoi node  $p_\infty$  at infinity which is the endpoint of all unbounded Voronoi edges, and when we triangulate the unbounded region, we also add edges to  $p_\infty$ . By our bounding triangle assumption, there is no point in  $I$  outside the convex hull of  $V$  and hence the conflict set of  $p_\infty$  is empty.

<sup>9</sup>This also holds for unbounded regions, since the only unbounded regions on the upper envelope of  $H_{V \cup I}$  correspond to the vertices of the bounding triangle.

Claim 3.8 implies that  $\mathcal{V}(V \cup I)$  can be computed 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, by traversing the edges of  $G_I(V)$  and fusing the bisectors of the restricted diagrams, put all the  $\mathcal{V}(Z_s) \cap s$  together to obtain  $\mathcal{V}(V \cup I)$ .

**Lemma 3.9** *The Voronoi diagram  $\mathcal{V}(V \cup I)$  can be computed in  $O(n)$  time.*

**Proof:** The time to compute  $\mathcal{V}(Z_s) \cap s$  for a triangle  $s \in G_I(V)$  is  $O(|Z_s| \log |Z_s|) = O(|Z_s|^2)$ . For a region  $r$  of  $\mathcal{V}(I)$ , 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  $r$  and let  $N(r)$  denote the set of nodes in  $\mathcal{V}(V)$  incident to  $r$ . Recall that  $t_r$  denotes the common vertex of all triangles in  $S(r)$ . Then the running time is proportional to

$$\begin{aligned} \mathbf{E} \left[ \sum_{s \in G_I(V)} |Z_s|^2 \right] &= \mathbf{E} \left[ \sum_{r \in \mathcal{V}(V)} \sum_{s \in S(r)} |Z_s|^2 \right] \leq \mathbf{E} \left[ \sum_{r \in \mathcal{V}(V)} \sum_{\substack{e \in E(r) \\ e=(t_1, t_2)}} (1 + X_{t_r} + X_{t_1} + X_{t_2})^2 \right] \\ &\leq \mathbf{E} \left[ \sum_{r \in \mathcal{V}(V)} \sum_{\substack{e \in E(r) \\ e=(t_1, t_2)}} (1 + 2X_{t_1} + X_{t_2})^2 \right], \end{aligned}$$

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  $\mathbf{E}[Y_e] = O(1)$ . We can express  $Y_e = \sum_i (1/n + 2\chi(t_1, i) + \chi(t_2, i))$ , and  $(1/n + 2\chi(t_1, i) + \chi(t_2, i)) < 4$ . By Claim 2.4,  $\mathbf{E}[Y_e^2] = O(1)$ . The number of edges in a Voronoi diagram is linear and each edge appears in two Voronoi regions. Therefore, the sum is in  $O(n)$ . Furthermore, assembling the restricted diagrams takes time  $O\left(\mathbf{E} \left[ \sum_{s \in G_I(V)} |Z_s| \right]\right)$ , and as  $|Z_s| \leq |Z_s|^2$ , this is also linear.  $\square$

## 3.2 Running time analysis

In this section, we prove the running time bound in Lemma 3.4 is indeed optimal. Before we go on, it is important to clarify the model of computation. We are using comparison based algorithms, where a single step (or ‘‘comparison’’) involves evaluating a point  $(z_1, z_2, \dots, z_d) \in \mathbb{R}^d$  (for constant  $d$ ) at some polynomial  $f(z_1, z_2, \dots, z_d) : \mathbb{R}^d \rightarrow \mathbb{R}$  and checking if the result is positive or negative. Based on this result, the algorithm chooses the next comparison to make. An algorithm can be completely represented by a decision tree, with each node representing some comparison. In this model, we get an information-theoretic lower bound of  $H(T(I))$  for computing the Delaunay triangulation of input  $I \in_R \mathcal{D}$ .

Recall that by Lemma 3.4, the running time of the our algorithm is expected  $O(n + \sum_i H_i^V)$ . The aim of this section is to prove the optimality of the algorithm by the following theorem.

**Theorem 3.10** *For  $H_i^V$ , the entropy of the triangle  $t_i$  of  $T(V)$  containing  $x_i$ , and  $H(T(I))$ , the entropy of the Delaunay triangulation of  $I$ , considered as a labelled graph,*

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

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  $u_i$  for  $x_i$ .
  - (c) Using Claim 3.12, for each unvisited neighbor  $x_j \in \Gamma_{V \cup I}(p) \cap I$ , compute a conflict triangle  $\tilde{u}_j$  in  $T(V \cup \{p\})$ .
  - (d) For each unvisited neighbor  $x_j \in \Gamma_{V \cup I}(p) \cap I$ , using  $\tilde{u}_j$ , compute a conflict triangle  $u_j$  in  $T(V)$ . Then insert  $x_j$  into  $Q$ , and mark it as visited.

Figure 3: Determining the conflict triangles.

**Proof:** The theorem is proved by an application of Claim 2.7 with  $\mathcal{U} = (\mathbb{R}^2)^n$ ,  $X = T(I)$  and  $Y = (t_1, \dots, t_n)$ , the set of conflict triangles for  $I$ . In Lemma 3.11 we will show that the function  $f : (I, T(I)) \mapsto (t_1, \dots, t_n)$  can be computed in linear time. The theorem now follows by Claims 2.6 and 2.7.  $\square$

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:

**Lemma 3.11** *Given  $I$  and  $T(I)$ , for each  $x_i$  in  $I$  we can compute the triangle  $t_i$  in  $T(V)$  that contains  $x_i$  in  $O(n)$  total expected time.*

**Proof:** First, we compute  $T(V \cup I)$  from  $T(V)$  and  $T(I)$  in linear time, using an algorithm by Chazelle [10]. We now show how to compute the conflict triangles in a special case.

**Claim 3.12** *Let  $J \subseteq I$  and assume that  $T(V \cup I)$  and  $T(V \cup J)$  are known. Furthermore, let  $p \in V \cup J$ . Then, in total time  $O(|\Gamma_{V \cup I}(p)| + |\Gamma_{V \cup J}(p)|)$ , for every  $x_i \in \Gamma_{V \cup I}(p) \setminus (V \cup J)$ , we can compute a conflict triangle  $\tilde{u}_i$  in  $T(V \cup J)$ .*

**Proof:** Let  $x_i \in \Gamma_{V \cup I}(p) \setminus (V \cup J)$ , and let  $\tilde{u}_i$  be the triangle of  $T(V \cup J)$  incident to  $p$  that is intersected by line segment  $\overline{px_i}$ . We claim that  $\tilde{u}_i$  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 (eg, [17, 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  $V \cup J \cup \{x_i\}$ , and hence  $\overline{px_i}$  is also an edge of  $T(V \cup J \cup \{x_i\})$ , again by the characterization of Delaunay edges applied in the other direction. Hence, triangle  $\tilde{u}_i$  is destroyed when  $x_i$  is inserted into  $T(V \cup J)$ , and thus  $x_i$  is in conflict with  $\tilde{u}_i$ .

It follows that the conflict triangles for  $\Gamma_{V \cup I}(p) \setminus (V \cup J)$  can be computed by merging the cyclically ordered lists  $\Gamma_{V \cup I}(p)$  and  $\Gamma_{V \cup J}(p)$ . This can be done in the claimed time.  $\square$

The conflict triangles for  $I$  can now be computed using breadth-first search (see Figure 3). The loop in Step 2b maintains the invariant that for each point  $x_i \in Q \cap I$ , a conflict triangle  $u_i$  in  $T(V)$  is known. Next, we claim that Step 2d can be done in constant time:

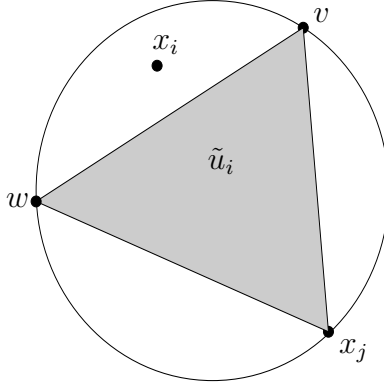


Figure 4: Proof of Claim 3.13

**Claim 3.13** *Let  $x_i, x_j \in I$ , and let  $\tilde{u}_i$  be a conflict triangle for  $x_i$  in  $T(V \cup \{x_j\})$  incident to  $x_j$ . Then we can find a conflict triangle  $u_i$  for  $x_i$  in  $T(V)$  in constant time.*

**Proof:** Let  $e$  be the edge of  $\tilde{u}_i$  not incident to  $x_j$ , and let  $v, w$  be the endpoints of  $e$  (refer to Figure 4). Since  $v, w \in V$ , by the characterization of Delaunay edges, it follows that  $e$  is also an edge of  $T(V)$ . Furthermore, since  $\tilde{u}_i$  is in conflict with  $x_i$ , we know that  $\overline{vx_i}$  and  $\overline{wx_i}$  are edges of  $T(V \cup \{x_i, x_j\})$ , and hence also edges of  $T(V \cup \{x_i\})$ . But this means 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.  $\square$

The while-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 2d we perform a BFS. The insertion in Step 2b takes  $O(|\Gamma_{V \cup \{x_i\}}(x_i)|)$  time. Furthermore, by Claim 3.12, the conflict triangles of  $p$ 's neighbors in  $T(V \cup I)$  can be computed in  $O(|\Gamma_{V \cup \{p\}}(p)| + |\Gamma_{V \cup I}(p)|)$  time. Finally, as we argued above, Step 2d can be carried out in  $O(|\Gamma_{V \cup I}(p)|)$  time. Now note that for  $x_i \in I$ ,  $|\Gamma_{V \cup \{x_i\}}(x_i)|$  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

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

Finally, using BFS as in the proof of Lemma 3.7, given the conflict triangles  $u_i$ , the triangles  $t_i$  can be found in  $O(n)$  time, and the result follows.  $\square$

### 3.3 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 trees  $\mathcal{D}_i$ ) and prove the time-space tradeoff given in Theorem 3.1. These techniques are identical to those used in Section 2.2. For the sake of clarity, we give a detailed explanation for this setting. Let  $\varepsilon > 0$  be any constant. The first  $O(\log n)$  rounds of the learning phase are used as before to construct the Delaunay triangulation  $T(V)$ . We first build a standard search structure  $D$  over the triangles of  $T(V)$ . Given a point  $x$ , we can find the triangle of  $T(V)$  that contains  $x$  in  $O(\log n)$  time.

The learning phase goes on for  $O(n^\varepsilon \log n)$  rounds. The main trick is to observe that (up to constant factors), the only probabilities that are relevant are those that are  $> n^{-\varepsilon}$ . In each round, for each  $x_i$ , we record the triangle of  $T(V)$  that  $x_i$  falls into. At the end of  $O(n^\varepsilon \log n)$  rounds, we take the set  $R_i$  of triangles such that for  $t \in R_i$ ,  $x_i$  was in  $t$  for at least  $\Omega(\log n)$  rounds. We remind the reader that  $p(t, i)$  is the probability that  $x_i$  lies in triangle  $t$ . For every triangle in  $R_i$ , we have an estimate of the probability  $\hat{p}(t, i)$  (obtained by simply taking the total number of times that  $x_i$  lay in  $t$ , divided by the total number of rounds). By a standard Chernoff bound argument, for all  $t \in R_i$ ,  $\hat{p}(t, i) = \Theta(p(t, i))$ . Furthermore, for any triangle  $t$ , if  $p(t, i) = \Omega(n^{-\varepsilon})$ , then  $t \in R_i$ .

For each  $x_i$ , we build the approximate search structure  $D_i$ . Consider the following probability distribution  $\bar{p}_i$  over the triangles of  $T(V)$ : if  $t \in R_i$ , set  $\bar{p}(t, i) := \hat{p}(t, i)/N_i$ , where  $N_i := \sum_{t \in R_i} \hat{p}(t, i)$ , and otherwise  $\bar{p}(t, i) := 0$ . Using the construction of [6], we can build the optimal planar point location structure  $D_i$  according to the distribution  $\bar{p}_i$ . The limiting phase uses these structures to find  $t_i$  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  $t_i$  (since  $t_i \notin R_i$ ), then we use the standard search structure,  $D$ , to find  $t_i$ . Therefore, we are guaranteed to find  $t_i$  in  $O(\log n)$  time. Without loss of generality, we can assume that each  $D_i$  deals with only  $n^\varepsilon$  triangles (and therefore, a planar subdivision of size  $n^\varepsilon$ ). By the bounds given in [6], each  $D_i$  can be constructed with size  $n^\varepsilon$  in  $n^\varepsilon \log n$  time. The total space is bounded by  $n^{1+\varepsilon}$  and the time required to build them is at most  $n^{1+\varepsilon} \log n$ .

Now we just repeat the argument given in Section 2.2. 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  $x_i \in t$ . By the properties of  $D_i$ , and noting that  $N_i \leq 1$ ,

$$\begin{aligned}
\sum_{t \in R_i} \bar{p}(t, i) s(t, i) &= \sum_{t \in R_i} \bar{p}(t, i) \log(1/\bar{p}(t, i)) \\
&= N_i^{-1} \sum_{t \in R_i} \hat{p}(t, i) \log(N_i/\hat{p}(t, i)) \\
&= N_i^{-1} \left[ \sum_{t \in R_i} \hat{p}(t, i) \log N_i - \sum_{t \in R_i} \hat{p}(t, i) \log \hat{p}(t, i) \right] \\
&\leq -N_i^{-1} \sum_{t \in R_i} \hat{p}(t, i) \log \hat{p}(t, i) \\
&= O \left( N_i^{-1} \left( 1 - \sum_{t \in R_i} p(t, i) \log p(t, i) \right) \right).
\end{aligned}$$

We now bound the expected search time for  $x_i$ .

$$\begin{aligned}
\sum_t p(t, i) s(t, i) &= \sum_{t \in R_i} p(t, i) s(t, i) + \sum_{t \notin R_i} p(t, i) s(t, i) \\
&= O \left( \sum_{t \in R_i} \hat{p}(t, i) s(t, i) + \sum_{t \notin R_i} p(t, i) \log n \right) \\
&= O \left( N_i \sum_{t \in R_i} \bar{p}(t, i) s(t, i) + \sum_{t \notin R_i} p(t, i) \log n \right).
\end{aligned}$$

Noting that for  $t \notin R_i$ ,  $p(t, i) = O(n^{-\varepsilon})$  and therefore  $\log p(t, i) \leq -\varepsilon \log n + O(1)$ , and so

$$\begin{aligned} \sum_t p(t, i) s(t, i) &= O \left( \left( 1 - \sum_{t \in R_i} p(t, i) \log p(t, i) \right) + \sum_{t \notin R_i} p(t, i) \varepsilon^{-1} (1 - \log p(t, i)) \right) \\ &= O \left( \varepsilon^{-1} \left( 1 - \sum_t p(t, i) \log p(t, i) \right) \right) \\ &= O(\varepsilon^{-1} (1 + H_i^V)) \end{aligned}$$

The total expected search time is  $O(\varepsilon^{-1}(n + \sum_i H_i^V))$ . By the analysis of Section 3.1 and Theorem 3.10, we have that the expected running time in the limiting phase is  $O(\varepsilon^{-1}(n + H(T(I))))$ . This completes the proof of Theorem 3.1.

## References

- [1] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21(3):312–329, 1998.
- [2] S. Albers and J. Westbrook. Self-organizing data structures. In *Online algorithms (Schloss Dagstuhl, 1996)*, volume 1442 of *Lecture Notes in Comput. Sci.*, pages 13–51. Springer Verlag, Berlin, 1998.
- [3] B. Allen and I. Munro. Self-organizing binary search trees. *J. ACM*, 25(4):526–535, 1978.
- [4] N. Alon and J. H. Spencer. *The probabilistic method*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience [John Wiley & Sons], New York, second edition, 2000.
- [5] S. Ar, B. Chazelle, and A. Tal. Self-customized BSP trees for collision detection. *Comput. Geom.*, 15(1–3):91–102, 2000.
- [6] S. Arya, T. Malamatos, D. M. Mount, and K. C. Wong. Optimal expected-case planar point location. *SIAM J. Comput.*, 37(2):584–610, 2007.
- [7] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Commun. ACM*, 28(4):404–411, 1985.
- [8] J. R. Bitner. Heuristics that dynamically organize data structures. *SIAM J. Comput.*, 8(1):82–110, 1979.
- [9] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, 1998.
- [10] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM J. Comput.*, 21(4):671–696, 1992.
- [11] B. Chazelle. *The discrepancy method*. Cambridge University Press, Cambridge, 2000.

- [12] B. Chazelle, O. Devillers, F. Hurtado, M. Mora, V. Sacristán, and M. Teillaud. Splitting a Delaunay triangulation in linear time. *Algorithmica*, 34(1):39–46, 2002.
- [13] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, pages 830–847, 1988.
- [14] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(1):387–421, 1989.
- [15] K. L. Clarkson and K. Varadarajan. Improved approximation algorithms for geometric set cover. *Discrete Comput. Geom.*, 37(1):43–58, 2007.
- [16] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, second edition, 2006.
- [17] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, Berlin, 2000.
- [18] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. Wiley-Interscience, New York, second edition, 2001.
- [19] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992.
- [20] M. L. Fredman. How good is the information theory bound in sorting? *Theoret. Comput. Sci.*, 1(4):355–361, 1975/76.
- [21] G. H. Gonnet, J. I. Munro, and H. Suwanda. Exegesis of self-organizing linear search. *SIAM J. Comput.*, 10(3):613–637, 1981.
- [22] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Comput. Surv.*, 17(3):295–311, 1985.
- [23] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, 1994.
- [24] J. Matoušek, R. Seidel, and E. Welzl. How to net a lot with little: small  $\varepsilon$ -nets for disks and halfspaces. In *SCG '90: Proceedings of the sixth annual symposium on Computational geometry*, pages 16–22, New York, NY, USA, 1990. ACM.
- [25] J. McCabe. On serial files with relocatable records. *Operations Res.*, 13:609–618, 1965.
- [26] K. Mehlhorn. *Data structures and algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, 1984.
- [27] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
- [28] R. Rivest. On self-organizing sequential search heuristics. *Commun. ACM*, 19(2):63–67, 1976.
- [29] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

- [30] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
- [31] F. Takens. Detecting strange attractors in turbulence. In *Dynamical systems and turbulence, Warwick 1980 (Coventry, 1979/1980)*, volume 898 of *Lecture Notes in Math.*, pages 366–381. Springer Verlag, Berlin, 1981.