CS369N: Beyond Worst-Case Analysis
Lecture #5: Self-Improving Algorithms*

Tim Roughgarden†

September 18, 2010

1 Preliminaries

Last lecture concluded with a discussion of semi-random graph models, an interpolation between worst-case analysis and average-case analysis designed to identify robust algorithms in the face of strong impossibility results for worst-case guarantees. This lecture and the next two give three more analysis frameworks that blend aspects of worst- and average-case analysis. Today's model, of *self-improving algorithms*, is the closest to traditional average-case analysis. The model and results are by Ailon, Chazelle, Comandar, and Liu [1].

The Setup. For a given computational problem, we posit a distribution over instances. The difference between today's model and traditional average-case analysis is that the distribution is *unknown*. The goal is to design an algorithm that, given an online sequence of instances — each an independent and identically distributed (i.i.d.) sample — quickly converges to an algorithm that is optimal for the underlying distribution. Thus the algorithm is "automatically self-tuning." The challenge is to accomplish this goal with fewer "training samples" and smaller space than a brute-force "learn the data model" approach.

Main Example: Sorting. The obvious first problem to apply the self-improving paradigm to is sorting in the comparison model, and that's what we do here. Each instance is an array of n elements, with the ith element drawn from a real-valued distribution D_i. A key assumption is that the D_i's are independent distributions; Section 5.3 discusses this assumption. The distributions need *not* be identical. Identical distributions are uninteresting in our context, since in this case the relative order of the elements is a uniformly random permutation. Every correct sorting algorithm requires $\Omega(n \log n)$ expected comparisons in this case, and a matching upper is bound is achieved by MergeSort (say).

†Department of Computer Science, Stanford University, 462 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

2 The Entropy Lower Bound

Since a self-improving algorithm is supposed to run eventually as fast as an optimal one for the underlying distribution, we need to understand some things about optimal sorting algorithms. In turn, this requires a lower bound on the running time of every sorting algorithm with respect to a fixed distribution.

The distributions $D_1, \ldots, D_n$ over $x_1, \ldots, x_n$ induce a distribution Π over permutations of $\{1, 2, \ldots, n\}$ via the ranks of the x_i's. (Assume throughout this lecture there are no ties.) As noted earlier, if Π is (close to) the uniform distribution over the set S_n of all permutations, then the worst-case comparison-based sorting bound of $\Omega(n \log n)$ also applies here in the average case. On the other hand, sufficiently trivial distributions Π can obviously be sorted faster. For example, if the support of Π involves only a constant number of permutations, these can be distinguished in $O(1)$ comparisons and then the appropriate permutation can be applied to the input in linear time. More generally, the goal is to beat the $\Omega(n \log n)$ sorting bound when the distribution Π has "low entropy"; and there is, of course, an implicit hope that "real data" can sometimes be well approximated by a low-entropy distribution.[1]

Mathematically, by *entropy* we mean the following.

Definition 2.1 (Entropy of a Distribution) Let $D = \{p_x\}_{x \in X}$ be a distribution over the finite set X. The *entropy* $H(D)$ of D is

$$\sum_{x \in X} p_x \log_2 \frac{1}{p_x}, \tag{1}$$

where we interpret $0 \log_2 \frac{1}{0}$ as 0.

For example, $H(D) = \log_2 |X|$ if D is uniform. When X is the set S_n of all permutations of $\{1, 2, \ldots, n\}$, this is $\Theta(n \log n)$. If D puts positive probability on at most 2^h different elements, then $H(D) \leq h$. [Exercise: prove this assertion. Use convexity to argue that, for a fixed support, the uniform distribution maximizes the entropy.]

Happily, we won't have to work with the formula (1) directly. Instead, we use Shannon's characterization of entropy in terms of average coding length.

Theorem 2.2 (Shannon's Theorem) *For every distribution D over the set X, the entropy $H(D)$ characterizes (up to an additive $+1$ term) the minimum possible expected encoding length of X, where a code is a function from X to $\{0,1\}^*$ and the expectation is with respect to D.*

Proving this theorem would take us too far afield, but it is accessible and you should look it up.

[1] For sorting, random data is the worst case and hence we propose a parameter to upper bound the amount of randomness in the data. This is an interesting contrast to the next two lectures, where random data is an unrealistically easy case and the analysis framework imposes a *lower bound* on the amount of randomness in the input.

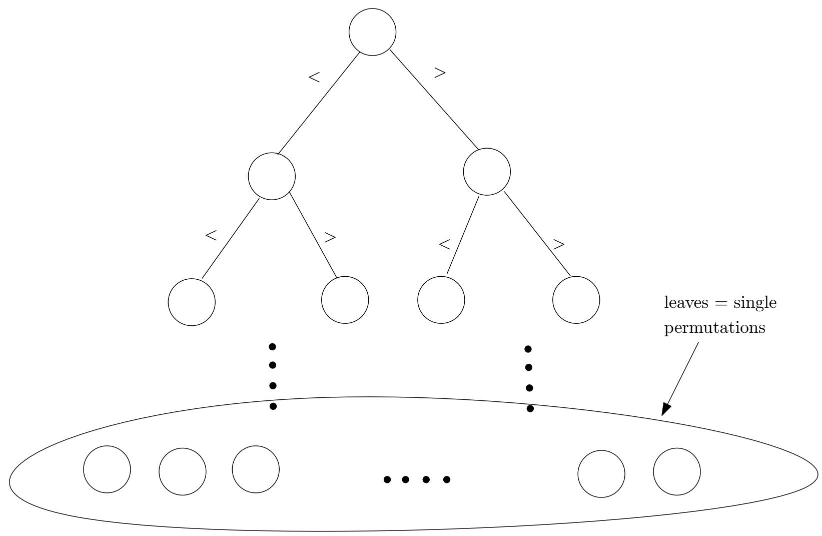

Figure 1: Every correct comparison-based sorting algorithm induces a decision tree, which induces a binary encoding of the permutations S_n on n elements.

A simple but important observation is that a correct sorting algorithm induces a binary encoding of S_n. To see this, recall that such an algorithm can be represented as a tree (Figure 1), with each comparison corresponding to a node with two children — the subsequent execution of the algorithm as a function of the comparison outcome. At a leaf, where the algorithm terminates, correctness implies that the input permutation has been uniquely identified. Label each left branch with a "0"and each right branch with a "1". By correctness, each permutation of S_n occurs in at least one leaf; if it appears at multiple leaves, pick one closest to the root. The sequence of 0's and 1's on a root-leaf path encodes the permutation. If there are distributions $\mathcal{D} = D_1, D_2, \ldots, D_n$ over the x_i's, inducing the distribution $\Pi(\mathcal{D})$ over permutations (leaves), then the expected number of comparisons of the sorting algorithm is at least the expected length of the corresponding encoding, which by Shannon's Theorem is at least $H(\pi(\mathcal{D}))$.

Upshot: Our goal should be to design a correct sorting algorithm that, for every distribution $\mathcal{D}$, quickly converges to the optimal per-instance expected running time of $O(n + H(\Pi(\mathcal{D})))$. [The second term is the necessary expected number of comparisons, and the first term is the time required to read the input and write the output.]

For example, if the induced distribution $\Pi(\mathcal{D})$ has support size $2^{O(n)}$, then the algorithm should converge to a per-instance expected running time of $O(n)$. The rest of this lecture provides such an algorithm.

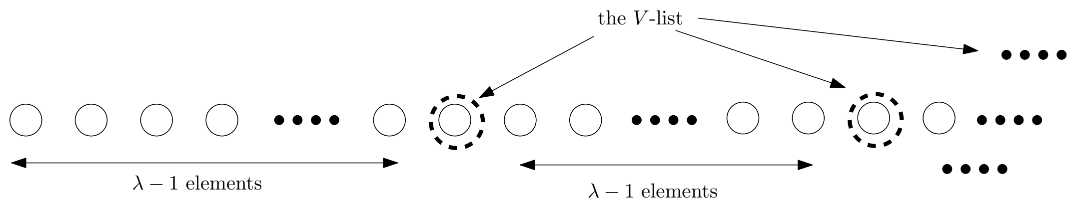

Figure 2: Construction of the V-list. After merging the elements of λ random instances, take every λth element.

3 The Basic Algorithm

3.1 High-Level Approach

Our self-improving algorithm is inspired by BucketSort, a standard method of sorting when you have good statistics about the data. For example, if the input elements are i.i.d. samples from the uniform distribution on $[0, 1]$, then one can have an array of n buckets, with the ith bucket meant for numbers between $(i-1)/n$ and i/n. A linear pass through the input is enough to put elements in their rightful buckets; assuming each bucket contains $O(1)$ elements, one can quickly sort each bucket separately and concatenate the results. The total running time in this case would be $O(n)$.

There are a couple of challenges in making this idea work for a self-improving algorithm. First, since we know nothing about the underlying distribution, what should the buckets be? Second, even if we have the right buckets, how can we quickly place the input elements into the correct buckets? (Recall that unlike traditional BucketSort, here we're working in the comparison model.) Note that if $H(\Pi(\mathcal{D})) = o(n \log n)$, then we can't afford to use standard binary search to distribute the input elements among the buckets.

3.2 Phase I: Constructing the V-List

Our first order of business is identifying good buckets, where "good" means that on future (random) instances the expected size of each bucket is $O(1)$. (Actually, we need something a little stronger than this.)

Set $\lambda = c \log n$ for a sufficiently large constant c. Our self-improving algorithm will, in its ignorance, sort the first λ instances using (say) MergeSort to guarantee a run time of $O(n \log n)$ per instance. At the same time, however, our algorithm will surreptitiously build a sorted "master list" L of the $\lambda n = \Theta(n \log n)$ corresponding elements. [Easy exercise: this can be done without affecting the $O(n \log n)$ per iteration time bound.] Our algorithm defines the set $V \subset L$ — the "V-list" — as every λth element of L (Figure 2), for a total of n overall. The elements in V are our "bucket boundaries", and they split the real line into $n + 1$ buckets in all.

The next lemma (proved in Section 4) justifies our V-list construction by showing that, with high probability over the choice of V, expected bucket sizes (squared, even) of future

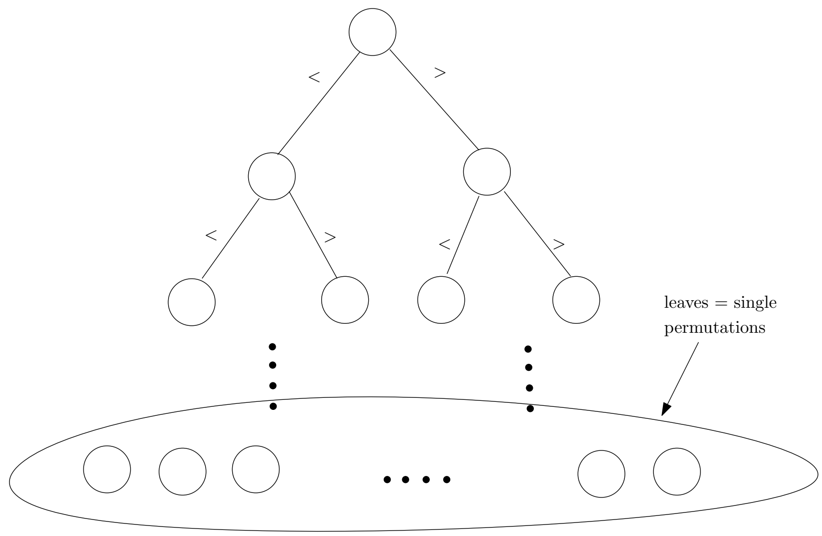

Figure 3: Every correct comparison-based searching algorithm induces a decision tree.

instances have constant size. This ensures that we won't waste any time sorting the elements within a single bucket.

For a fixed choice of V and a bucket i (between the ith and $(i+1)$th element of V), let m_i denote the (random) number of elements of an instance that fall in the ith bucket.

Lemma 3.1 (Elements Are Evenly Distributed) *Fix distributions $\mathcal{D} = D_1, \ldots, D_n$. With probability at least $1 - 1/n^2$ over the choice of V, $\mathbf{E}_{\mathcal{D}}[m_i^2] \leq 20$ for every bucket i.*

3.3 Interlude: Optimal Bucket Classification

The next challenge is, having identified good buckets, how do we quickly classify which element of a random instance belongs to which bucket? Remember that this problem is non-trivial because we need to compete with an entropy bound, which can be small even for quite non-trivial distributions over permutations.

To understand the final algorithm, it is useful to first cheat and assume that the distributions D_i are known. (This is not the case, of course, and our analysis of Phase I does not assume this.) In this case, we might as well proceed in the *optimal* way: that is, for a given element x_i from distribution D_i, we ask comparisons between x_i and the bucket boundaries in the way that minimizes the expected number of comparisons. As with sorting (Figure 1), comparison-based algorithms for searching can be visualized as decision trees (Figure 3), where each leaf corresponds to the (unique) bucket to which the given element can belong given the results of the comparisons. Unlike sorting, however, it can be practical to *explicitly construct* these trees for optimal searching. For starters, they have only $O(n)$ size, as opposed to the $\Omega(n!)$ size trees that are generally required for optimal sorting.

Precisely, let B_i denote the distribution on buckets induced by the distribution D_i. (This is with respect to a choice of V, which is now fixed forevermore.) Computing the optimal search tree T_i for a given distribution B_i is then bread-and-butter dynamic programming. The key recurrence is

$$\mathbf{E}[T_i] = 1 + \min_{j=1,2,\ldots,n} \left\{ \mathbf{E}\left[T_i^{1,2,\ldots,j-1}\right] + \mathbf{E}\left[T_i^{j+1,j+2,\ldots,n}\right] \right\},$$

where $T_i^{a,\ldots,b}$ denotes the optimal search tree for locating x_i with respect to the bucket boundaries $a, a+1, \ldots, b$. In English, this recurrence says that if you knew the right comparison to ask first, then one would solve the subsequent subproblem optimally. The dynamic program effectively tries all n possibilities for the first comparison. There are $O(n^2)$ subproblems (one per contiguous subset of $\{1, 2, \ldots, n\}$) and the overall running time is $O(n^3)$. Details are left to the reader. For a harder exercise, try to improve the running time to $O(n^2)$ by exploiting extra structure in the dynamic program. Given such a solution, the total time needed to construct all n trees — a different one for each D_i, of course — is $O(n^3)$. We won't worry about how to account for this work until the end of the lecture.

Since binary encodings of the buckets $\{0, 1, 2, \ldots, n\}$ and decision trees for searching are essentially in one-to-one correspondence (with bits corresponding to comparison outcomes), Shannon's Theorem implies that the expected number of comparisons used to classify x_i via the optimal search tree T_i is essentially the entropy $H(B_i)$ of the corresponding distribution on buckets. We relate this to the entropy that we actually care about (that of the distribution on permutations) in Section 4.

3.4 Phase II: The Steady State

For the moment we continue to assume that the distributions D_i are known. After the V-list is constructed in Phase I and the optimal search trees $T_1, \ldots, T_n$ over buckets are built in the Interlude phase, the self-improving algorithm runs as shown in Figure 4 for every future random instance.

Input: a random instance $x_1, \ldots, x_n$, with each x_i drawn independently from D_i.

1. For each i, use T_i to put x_i into the correct bucket.
2. Sort each bucket (e.g., using Insertion Sort).
3. Concatenate the sorted buckets and return the result.

Figure 4: The steady state of the basic self-improving sorter.

---

4 Running Time Analysis of the Basic Algorithm

We have already established that Phase I runs in $O(n \log n)$ time per iteration, and that the Interlude requires $O(n^3)$ computation (to be refined in Section 5.2). As for Phase II, it is obvious that the third step can be implemented in $O(n)$ time (with probability 1, for any distribution). Bounding the expected running time of each of the first two steps requires a non-trivial argument.

For the first step, we have already argued that the expected running time is proportional to $\sum_{i=1}^{n} H(B_i)$, where B_i is the distribution on buckets induced by D_i. (Recall that the choice of V in Phase I determines the buckets.) The next lemma shows that this quantity is no more than the target running time bound of $O(n + H(\Pi(\mathcal{D})))$.

Lemma 4.1 *With probability 1 over the choice of V in Phase I,*

$$\sum_{i=1}^{n} H(B_i) = O(n + H(\Pi(\mathcal{D}))).$$

Proof: Fix an arbitrary choice of bucket boundaries V. By Shannon's Theorem, we only need to exhibit a binary encoding of the buckets to which $x_1, \ldots, x_n$ belong that has expected length $O(n + H(\Pi(\mathcal{D})))$ over $\mathcal{D}$. As usual, our encoding is the comparison results of an algorithm, which we define for the purposes of analysis only. Given $x_1, \ldots, x_n$, the first step is to sort the x_i's using an optimal (for $\mathcal{D}$) sorting algorithm. Such an algorithm exists (in principle), and Shannon's Theorem implies that it uses $\approx H(\Pi(\mathcal{D}))$ comparisons on average. The second step is to merge the sorted list of x_i's together with the bucket boundaries V (which are also sorted). Merging these two lists requires $O(n)$ comparisons, the results of which uniquely identify the correct buckets for all of the x_i's — the last bucket boundary to which x_i is compared is the right endpoint of its bucket. The comparison results of this two-step algorithm can be interpreted as a binary encoding of the buckets to which the x_i's belong that has expected length $O(n + H(\Pi(\mathcal{D})))$.[2] ■

Finally, we prove that the expected running time of the second step of Phase II is $O(n)$. This follows easily from Lemma 3.1. With probability at least $1 - \frac{1}{n^2}$ over the choice of V in Phase I, the expectation of the squared size of every bucket is at most 20, so Insertion Sort runs in $O(1)$ time on each of the $O(n)$ buckets. For the remaining probability of at most $\frac{1}{n^2}$, we can use the $O(n^2)$ worst-case running time bound for Insertion Sort; even then, this exceptional case contributes only $O(1)$ to the expected running time of the self-improving sorter.

We conclude the section by proving Lemma 3.1. The proof is a conceptually straightforward but nicely executed application of the Chernoff and Union bounds.

[2]The astute reader will have noticed that this proof actually upper bounds the entropy of the joint distribution $(B_1, \ldots, B_n)$. Independence of the x_i's implies that the entropy $H(B_1, \ldots, B_n)$ equals the sum of the entropies of the components $\sum_i H(B_i)$, which gives the desired bound.

Proof of Lemma 3.1: The basic reason the lemma is true, and the reason for our choice of λ, is because the Chernoff bound has the form

$$\Pr[X < (1-\delta)\mu] \approx e^{-\mu\cdot\delta^2}, \tag{2}$$

where we are ignoring constants in the exponent on the right-hand side, and where μ is defined as $\mathbf{E}[X]$. Thus: *when the expected value of the sum of Bernoulli random variables is at least logarithmic, even constant-factor deviations are highly unlikely.*

To make use of this fact, consider the λn elements S that belong to the first λ inputs, from which we draw V. Consider fixed indices $k, \ell \in \{1, 2, \ldots, \lambda n\}$, and define $\mathcal{E}_{k,\ell}$ as the (bad) event (over the choice of S) that

(i) the kth and ℓth elements take on a pair of values $a, b \in \mathcal{R}$ for which $\mathbf{E}_{\mathcal{D}}[m_{ab}] \geq 4\lambda$, where m_{ab} denotes the number of elements of λ random inputs (i.i.d. samples from $\mathcal{D}$) that lie between a and b; and

(ii) at most λ other elements of S lie between a and b.

That is, $\mathcal{E}_{k,\ell}$ is the event that the kth and ℓth samples are pretty far apart and yet there is an unusually sparse population of other samples between them.

We claim that for all k, ℓ, $\Pr[\mathcal{E}_{k,\ell}]$ is very small, at most $1/n^5$. To see this, fix k, ℓ and condition on the event that (i) occurs, and the corresponding values of a and b. Even then, the Chernoff bound (2) implies that the (conditional) probability that (ii) occurs is at most $1/n^5$, provided the constant c in the definition of λ is sufficiently large. (We can take $\mu = 4c\log n$ and $\delta = \frac{1}{2}$, for example.) Taking a Union Bound over the at most $(\lambda n)^2$ choices for k, ℓ, we find that $\Pr[\vee_{k,\ell}\mathcal{E}_{k,\ell}] \leq 1/n^2$. This implies that, with probability at least $1 - \frac{1}{n^2}$, for every pair $a, b \in S$ of samples with less than λ other samples between them, the expected number of elements of λ future random instances that lie between a and b is at most 4λ. This guarantee applies in particular to consecutive bucket boundaries, since only $\lambda - 1$ samples separate them.

To recap, with probability at least $1 - \frac{1}{n^2}$ (over S), for every bucket B defined by two consecutive boundaries a and b, the expected number of elements of λ independent random inputs that lie in B is at most 4λ. By linearity, the expected number of elements of a *single* random input that land in B is at most 4.

To finish the proof, consider an arbitrary bucket B_i and let X_j be the random variable indicating whether or not the jth element of a random instance lies in B_i. We have the following upper bound on the expected squared size m_i^2 of bucket i:

$$\begin{aligned}\mathbf{E}\left[(X_1 + X_2 + \cdots + X_n)^2\right] &= \sum_j \mathbf{E}\left[X_j^2\right] + 2\sum_{j<h}\mathbf{E}[X_j]\cdot\mathbf{E}[X_h] \\ &\leq \left(\sum_j \mathbf{E}[X_j]\right) + \left(\sum_j \mathbf{E}[X_j]\right)^2,\end{aligned}$$

where the equality uses linearity of expectation and the independence of the X_j's, and the inequality uses the fact that the X_j's are 0-1 random variables. With probability at least

$1 - \frac{1}{n^2}$, $\sum_j \mathbf{E}[X_j] \leq 4$ and hence this upper bound is at most 20, simultaneously for every bucket B_i. ■

5 Extensions

5.1 Optimizing the Space

We begin with a simple and clever optimization that also segues into the next extension, which is about generalizing the basic algorithm to unknown distributions.

The space required by our self-improving sorter is $\Theta(n^2)$, for the n optimal search trees (the T_i's). Assuming still that the D_i's are known, suppose that we truncate each T_i after level $\epsilon \log n$, for some $\epsilon > 0$. These truncated trees require only $O(n^\epsilon)$ space each, for a total of $O(n^{1+\epsilon})$. What happens now when we search for x_i's bucket in T_i and fall off the bottom? We just locate the correct bucket by standard binary search! This takes $O(\log n)$ time, and we would have had to spend at least $\epsilon \log n$ time searching for it in the original tree T_i anyways. Thus the price we pay for maintaining only the truncated versions of the optimal search trees is a $\frac{1}{\epsilon}$ blow-up in the expected bucket location time, which is a constant-factor loss for any fixed $\epsilon > 0$. Conceptually, the big gains from using an optimal search tree instead of standard binary search occur at leaves that are at very shallow levels (and presumably are visited quite frequently).

5.2 Unknown Distributions

The elephant in the room is that the Interlude and Phase II of our self-improving sorter currently assume that the distributions $\mathcal{D}$ are known a priori, while the whole point of the self-improving paradigm is to design algorithms that work well for *unknown* distributions. The fix is the obvious one: we build (near-optimal) search trees using empirical distributions, based on how frequently the ith element lands in the various buckets.

More precisely, the general self-improving sorter is defined as follows. Phase I is defined as before and uses $O(\log n)$ training phases to identify the bucket boundaries V. The new Interlude uses further training phases to estimate the B_i's (the distributions of the x_i's over the buckets).

The analysis in Section 5.1 suggests that, for each i, only the $\approx n^\epsilon$ most frequent bucket locations for x_i are actually needed to match the entropy lower bound. For V and i fixed, call a bucket *frequent* if the probability that x_i lands in it is at least $1/n^\epsilon$, and *infrequent* otherwise. An extra $\Theta(n^\epsilon \log n)$ training phases suffice to get accurate estimates of the probabilities of frequent buckets (for all i) — after this point one expects $\Omega(\log n)$ appearances of x_i in every frequent bucket for i, and using Chernoff bounds as in Section 4 implies that all of the empirical frequency counts are close to their expectations (with high probability). The algorithm then builds a search tree $\hat{T}_i$ for i using only the buckets (if any) in which $\Omega(\log n)$ samples of x_i landed. This involves $O(n^\epsilon)$ buckets and can be done in $O(n^{2\epsilon})$ time and $O(n^\epsilon)$ space. For $\epsilon \leq \frac{1}{2}$, this does not affect the $O(n \log n)$ worst-case running time bound for every

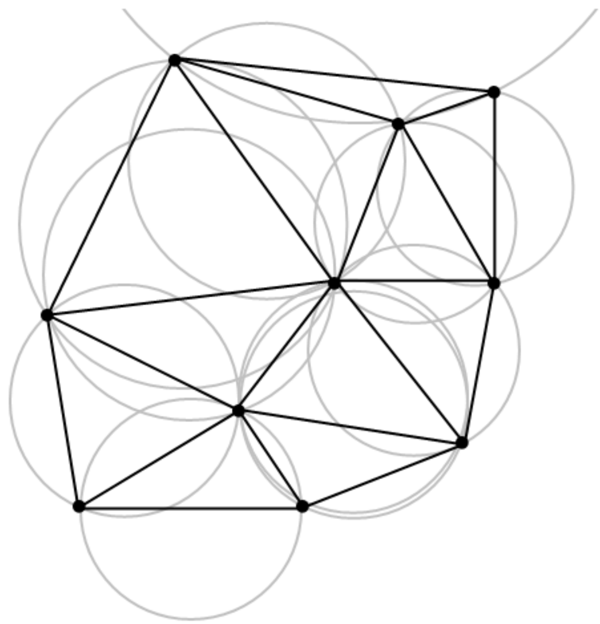

Figure 5: A Delaunay triangulation in the plane with circumcircles shown.

training phase. One can show that the $\hat{T}_i$'s are essentially as good as the truncated optimal search trees of Section 5.1, with buckets outside the trees being located via standard binary search, so that the first step of Phase II of the self-improving sorter continues to have an expected running time of $O(\sum_i H(B_i)) = O(n + H(\Pi(\mathcal{D})))$. The arguments have a similar flavor to the proof of Lemma 3.1, although the details are a bit different and are left to the reader (or see [1]). Finally, the second and third steps of Phase II of the self-improving sorter obviously continue to run in time $O(n)$ [expected] and $O(n)$ [worst-case], respectively.

5.3 Beyond Independent Distributions

The assumption that the D_i's are independent distributions is strong. Some assumption is needed, however, as a self-improving sorter for arbitrary distributions Π over permutations provably requires exponential space — intuitively, there are too many fundamentally distinct distributions that need to be distinguished (see [1] for the counting argument that proves this). An interesting open question is to find an assumption weaker than (or incomparable to) independence that is strong enough to allow interesting positive results. The reader is encouraged to go back over the analysis above and identify all of the (many) places where we used the independence of the D_i's.

5.4 Delaunay Triangulations

Clarkson and Seshadhri [2] give a non-trivial extension of the algorithm and analysis in [1], to the problem of computing the Delaunay triangulation of a point set. The input is n points in the plane, where each point x_i is an independent draw from a distribution D_i. One definition of a Delaunay triangulation is that, for every face of the triangulation, the circle that goes through the three corners of the face encloses no other points of the input (see Figure 5 for an example and the textbook [3, Chapter 9] for much more on the problem). The main result in [2] is again an optimal self-improving algorithm, with steady-state expected running time $O(n + H(\Delta(\mathcal{D})))$, where $H(\Delta(\mathcal{D}))$ is the suitable definition of entropy for the induced distribution $\Delta(\mathcal{D})$ over triangulations. The algorithm is again an analog of BucketSort, but a number of the details are challenging. For example, while the third step of Phase II of the self-improving sorter — concatenating the sorted results from different buckets — is trivially linear-time, it is much less obvious how to combine Delaunay triangulations of constant-size "buckets" into one for the entire point set. It can be done, however; see [2].

References

[1] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Self-improving algorithms. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 261–270, 2006.

[2] K. L. Clarkson and C. Seshadhri. Self-improving algorithms for Delaunay triangulations. In *Proceedings of the 24th Annual ACM Symposium on Computational Geometry (SCG)*, pages 148–155, 2008.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000. Second Edition.