Estimating the longest increasing sequence in polylogarithmic time

Michael Saks
Department of Mathematics
Rutgers University
Piscataway, USA
saks@math.rutgers.edu

C. Seshadhri
IBM Almaden
San Jose, USA
csesha@us.ibm.com

***Abstract*—Finding the length of the longest increasing subsequence (LIS) is a classic algorithmic problem. Let n denote the size of the array. Simple $O(n \log n)$ time algorithms are known that determine the LIS exactly. In this paper, we develop a randomized approximation algorithm, that for any constant $\delta > 0$, runs in time polylogarithmic in n and estimates the length of the LIS of an array up to an additive error of δn. The algorithm presented in this extended abstract runs in time $(\log n)^{O(1/\delta)}$. In the full paper, we will give an improved version of the algorithm with running time $(\log n)^c (1/\delta)^{O(1/\delta)}$ where the exponent c is independent of δ. Previously, the best known polylogarithmic time algorithms could only achieve an additive $n/2$-approximation.**

Our techniques also yield a fast algorithm for estimating the *distance to monotonicity* to within a small multiplicative factor. The distance of f to monotonicity, ε_f, is equal to $1 - |LIS|/n$ (the fractional length of the complement of the LIS). For any $\delta > 0$, we give an algorithm with running time $O((\varepsilon_f^{-1} \log n)^{O(1/\delta)})$ that outputs a $(1+\delta)$-multiplicative approximation to ε_f. This can be improved so that the exponent is a fixed constant. The previously known polylogarithmic algorithms gave only a 2-approximation.

***Keywords*-Sublinear algorithms; Monotonicity; Longest Increasing Subsequence; Dynamic Programming**

I. Introduction

Finding the length of longest increasing subsequence (LIS) of an array is a classic algorithmic problem in computer science. We are given a function $f : [n] \to \mathbb{R}$, which we also call an *array*. An *increasing subsequence* of this array is a sequence of indices $i_1 < i_2 < \cdots < i_k$ such that $f(i_1) \leq f(i_2) \leq \cdots \leq f(i_k)$. An LIS is an increasing subsequence of f having maximum size. The LIS is the longest common subsequence between f and its sorted version.

The LIS problem is a standard example for illustrating the technique of dynamic programming, and the obvious dynamic program runs in time $O(n^2)$. Fredman [Fre75] gave an $O(n \log n)$ algorithm, which can be seen as a more clever way of maintaining the dynamic program. Aldous and Diaconis [AD99] use the elegant algorithm of *patience sorting* to find the LIS.

In some contexts, it is more natural to consider the size of the *complement* of the LIS. This is referred to as the (edit) *distance to monotonicity*, since it is the minimum number of values that need to be changed to make f monotonically nondecreasing. Conventially, this number is divided by the total size n, so it is actually a fraction. For function f, it is denoted by ε_f. Letting λ_f be the size of the LIS as a fraction of n, we have $\varepsilon_f = 1 - \lambda_f$. For exact algorithms, of course, finding λ_f is equivalent to finding ε_f. Approximating these quantities can be very different problems.

In recent years, motivated by the increasing ubiquity of massive sets of data, there has been considerable attention given to the study of approximate solutions of computational problems on huge data sets by sampling the input. In the context of property testing it was shown in [EKK+00], [DGL+99], [Fis01], [ACCL07] that for any $\varepsilon > 0$, $O(\varepsilon^{-1} \log n)$ random samples are necessary and sufficient to distinguish the case that f is increasing ($\varepsilon_f = 0$) from the case that $\varepsilon_f \geq \varepsilon$. Subsequently, results of [PRR06], [ACCL07] gave *multiplicative approximations* to the distance to monotonicity in time (essentially) $O(\varepsilon_f^{-1} \log n)$. For any constant $\tau > 0$, these algorithms output a value that is in the range $[\varepsilon_f, (2+\tau)\varepsilon_f]$, which essentially gives a 2-approximation to ε_f.

While these algorithms give good approximations to the distance to monotonicity, they provide little information about the LIS if λ_f is between 0 and $1/2$. Note that in this case $\varepsilon_f \geq 1/2$ and so a 2-approximation to ε_f may output 1 as the estimate, which corresponds to estimating the LIS size as 0. Indeed, there are simple examples where $\varepsilon_f = 1/2$ and the algorithms of [PRR06], [ACCL07] returns an estimate of 1. These algorithms do well for small values of ε_f because f is nearly increasing, and the algorithm can exploit this. However when λ_f is small (say $1/10$), the global structure of f can vary quite widely, and it is much more difficult to accurately bound the size of the LIS.

In this paper, we focus on getting an additive approximations to λ_f, i.e. for some fixed small $\delta > 0$, outputting an estimate λ' such that with high probability $\lambda' \leq \lambda_f \leq \lambda' + \delta$. Notice that this is equivalent to getting an additive δ-approximation for ε_f. The existing multiplicative 2-approximation algorithm for ε_f gives the rather weak consequence of an additive $1/2$-approximation for λ_f, which was the best known prior to this paper.

We show that polylogarithmic time is sufficient to get

arbitrarily good additive approximations to λ_f. For any constant δ, we get additive δ-approximations in polylogarithmic time. We will use *"with high probability"* to indicate probability at least $1-n^{-\Omega(\log n)}$, where the Ω-notation hides some fixed constant independent of any parameters.

The main result presented here is an algorithm that, for any constant $\delta > 0$, runs in time polylogarithmic in n and with high probability outputs an estimate of the LIS with additive error δn:

Theorem 1.1: There is a randomized algorithm A that takes as input an array f of length n and a parameter $\delta > 0$, and outputs a real number k such that: (1) With high probability, the LIS length lies in the range $[k, k+\delta n]$, and (2) The running time is bounded by $(\log n)^{O(1/\delta)}$.

While the running time of this algorithm is polylogarithmic in n for each fixed δ, the exponent increases as δ decreases. We have a somewhat more complicated version of the algorithm that has better time complexity:

Theorem 1.2: There is a randomized algorithm B that takes as input an array f of length n and a parameters $\delta > 0$, and outputs a real number k such that: (1) With high probability, the LIS length lies in the range $[k, k+\delta n]$, and (2) The running time is bounded by $C(\delta)(\log n)^b$, where b is a fixed constant and $C(\delta)$ is independent of n.

Our techniques provide a somewhat stronger algorithm, that provides a multiplicative $(1+\tau)$-approximation to ε_f for any $\tau > 0$. We note that this is the first improvement over the $(2+\tau)$-approximations of [PRR06], [ACCL07]. As before, c denotes some absolute constant.

Theorem 1.3: Let $\tau > 0$ and $\varepsilon_f > 0$ be the distance to monotonicity for input array f. There exists an algorithm with running time $(1/\varepsilon_f)^{(1/\tau)\log(1/\tau)}(\log n)^c$ that computes a real number ε such that (with high probability) $\varepsilon_f \in [\varepsilon, (1+\tau)\varepsilon]$.

In this extended abstract, we will focus on the algorithm that yields Theorem 1.1. The algorithms that yield Theorems 1.2 and Theorem 1.3 use similar ideas, and will be described in the full paper [SS10].

A. Related work and relation to other models

The field of *property testing* [RS96], [GGR98] deals with finding sublinear, or even constant time algorithms for distinguishing whether an input has a property, or is far from the property (see surveys [Fis01], [Ron01], [Gol98]). The property of monotonicity has been studied over a variety of domains, of which the boolean hypercube and the set $[n]$ have usually been of special interest [GGL$^+$00], [DGL$^+$99], [EKK$^+$00], [FLN$^+$02], [HK03], [ACCL07], [PRR06], [BGJ$^+$09]. Our result can be seen as a *tolerant tester* [PRR06], that closely approximates the distance to monotonicity.

The LIS has been studied in detail in the streaming model [GJKK07], [SW07], [GG07], [EJ08]. Here, we are allowed a small number (usually, just a single) of passes over the array and we wish to estimate either the LIS or the distance to monotonicity. The distance approximations in the streaming model are based on the sublinear time approximations. The technique of counting inversions used in the property testers and sublinear time distance approximators is a major component of these algorithms. This problem has also been studied in the communication models where various parties may hold different portions of the array, and the aim is to compute the LIS with minimum communication. This is usually studied with the purpose of proving streaming lower bounds [GJKK07], [GG07], [EJ08].

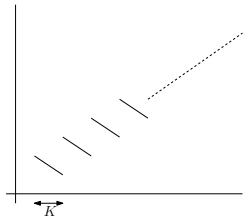

Figure 1. Small random samples will almost always be totally increasing

There has been a body of work on studying the Ulam distance between strings [AK07], [AK08], [AIK09], [AN10]. For permutations, the Ulam distance is exactly the size of the complement of the longest common subsequence. Note that Ulam distance between a permutation and the identity permutation is the distance to monotonicity. Recently, a sublinear time algorithm for approximating the Ulam distance between two permutations was discovered [AN10].

B. Obstacles to additive estimations of the LIS

A first natural approach to estimating λ_f is to choose a small random sample S of indices, compute the LIS within the sample, and output the length of the LIS divided by $|S|$. However, there are functions f having λ_f arbitrarily close to 0 on which this algorithm will answer 1 with high probability. Let K be a large constant and $n = Kt$. For $0 \le i \le t-1$ and $0 \le j < K$, set $f(iK+j+1) = iK - j$. Refer to Figure 1. The LIS of this function is size $t = n/K$, but a small random sample will almost certainly be completely increasing. This is because the scale of violations (pairs $i < j$ with $f(i) > f(j)$) is too small for the sample to detect. Violations can lie at any scale, and the algorithm needs to be able to find them wherever they are.

We refer to elements in the domain $[n]$ as indices. A natural approach to algorithms for estimating the LIS is to give an algorithm which, given an index i, classifies i as *good* or *bad* in such a way that:

- The good indices form an increasing sequence.
- The number of good indices is close to the size of the LIS, so the number of bad indices is small.

This was the approach used in [ACCL07] and [PRR06]. The condition that the good indices form an increasing sequence can be restated as: for every pair of indices that form a violation, at least one of them is classified as bad.

When given an index i to classify, the algorithm must either declare that i is bad, or must be certain that every index that is in violation with i will be declared bad. A key observation from [EKK+00] is that if i, j form a violation with $i < j$, then at least one of i or j is in violation with at least half of the indices in the interval $[i, j]$. This gives the following criterion for bad indices: declare i to be bad if there is an interval with endpoint i such that at least half of the indices in the interval are in violation with i. While checking this condition exactly requires linear time, it can be tested approximately very fast: for suitably small constants $\gamma, \delta > 0$ consider each of the (logarithmically many) intervals having length $\lceil (1+\gamma)^k \rceil$ and also having i as an endpoint, and sample a logarithmic number of points from the interval. Declare i to be good if for each of these intervals, the observed fraction of violations with i is at most $(1-\delta)/2$, and otherwise declare i to be bad. If i is declared good, then (with high probability) in any interval with endpoint i fewer than half the points are in violation with i. Thus the set of good indices forms an increasing sequence. This is essentially the algorithm of [ACCL07] (the algorithm of [PRR06] uses similar ideas). It can be shown that with high probability the number of points declared bad is less than $(2 + O(\delta))\varepsilon_f$.

However, this algorithm can perform very badly on inputs where the LIS has size less than $n/2$. For the function shown in Figure 1, even when K is 2 (so $\lambda_f = 1/2$), the above algorithm will declare all indices to be bad. The problem is that all indices are involved in many violations in an interval of size K.

An even more difficult issue is that when the LIS is not most of the points, the decision about whether to label a point as good *may involve small scale properties of the sequence far from* i. Consider the following example: suppose $n = 6k$ and divide the indices into three contiguous blocks, where the first has size k, the second has size $2k$ and the third has size $3k$. Consider the sequence f whose first block is $100k + 1, \ldots, 101k$, whose second block is $1, 101k+1, 2, 101k+2, \ldots, k, 101k$ and whose third block is some increasing subsequence of $k + 1, \ldots, 99k$. Let f' be a sequence that agrees with f on the first two blocks. The final $3k$ positions is some sequence with values in the range $k+1, \ldots, 99k$ but looks like the function in Figure 1. Figure 2 depicts f and f'.

Notice that the LIS of f has size $4k$ (and excludes the first block of elements) while the LIS for f' has size $2k$ (and includes the first block of elements). Furthermore, in f, an increasing sequence that uses any element from the first block has size at most $2k$. Any good approximation algorithm for the LIS must realize that elements from the first block are part of the LIS for f', but not for f. But the only difference between f and f' is in very local properties in the third block. This is because violations in the third block (for f') are only present at a very small scale. So, small scale violations in the third block are critical to decisions made in the first block.

One can build many variants of this example, where the size and location of the "critical" blocks, and the scale at which critical information is located, vary widely. The algorithm must be able to distinguish these situations and act on them. The algorithm must properly aggregate information about different regions, and decide which regions require further exploraiton. The technique of dynamic programming is a systematic way to aggregate information from all over the array, and at all scales. As the dynamic programming algorithm proceeds from left to right in the array, it must keep track of different alternatives, any one of which may turn out to be critical to obtain (or even estimate) the optimum. A fast approximation algorithm needs to somehow mimic this behavior of dynamic programming, while only looking at a tiny portion of the input. This is the challenge that must be met to design a very fast approximation algorithm.

II. The algorithmic idea and intuition

Taking some inspiration from complexity theory, we consider as a first step the easier problem of developing a fast *interactive proof* for LIS. (Interactive proofs are not mentioned explicitly in the main body of the paper, but underlies our approach.) In this setting, both the prover and verifier have access to the array f. The prover has unlimited computational power, while the verifier is limited. The prover wishes to convince the verifier that $\lambda_f \geq \lambda$ for some λ. The proof is implemented by an interactive protocol between the verifier and prover, and we want this proof to be very fast. As usual for interactive proofs, the protocol should have two properties: *completeness* says that if $\lambda_f \geq \lambda$ then it should be possible for the prover to convince the verifier (with probability near 1). *Soundness* requires that if $\lambda_f \leq (\lambda - \delta)n$ (where δ is a small positive constant), then no matter what the prover does the verifier will reject with high probability.

Here's how the proof works. The prover is instructed to construct a new function g that agrees with f on the indices of an LIS, and whose other positions are set so that g is monotone. Note that g must disagree with f outside of the LIS. Note that the prover does not have time to tell the verifier the entire function g. Nonetheless, assuming the prover constructs g as instructed, the probability that $f(i) = g(i)$ for a randomly chosen index i is exactly λ_f. Thus, if the prover behaved properly, the verifier could simply select a random sample of indices and accept if the fraction of indices where $f(i) = g(i)$ is at least $\lambda - \delta/2$. This satisfies completeness, but clearly violates soundness, since a dishonest prover does not need to construct g in advance, but could determine his answers to the queries $g(i)$, based on the indices queried. This could enable the prover to convince the verifier that the LIS is much larger than it really is. We now show how the verifier can ask the prover

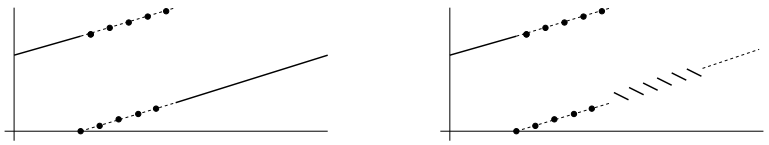

Figure 2. The sequences/functions f and f' where small scale properties in one block affect points of a distant block

a small number of questions and get both completeness and soundness. Either the verifier will catch the prover in a lie (and reject the proof), or will be convinced that the probability that $f(i) = g(i)$ is at most λ_f.

After choosing the random index i, but before revealing i to the prover, the verifier will ask the prover for some additional information. They perform the following subprotocol (which we refer to as the *inner protocol*). The verifier asks the prover to provide the value of g for any index of the prover's choice. Say that the prover provides the value of g at index s. The verifier tells the prover which of the intervals $[1, s]$ and $[s+1, n]$ that the selected index i falls into. They then restrict attention to this subarray and repeat with the prover selecting an index in the current range and providing the value of g at that index. If at any point the prover gives an answer which is inconsistent with the requirement that g be increasing the verifier rejects the proof. Otherwise, this search procedure narrows down to the index i. The verifier reads $f(i)$ and compares it to the value of $g(i)$.

The inner protocol does not change the fact that for a prover who behaves as instructed, the probability that $f(i) = g(i)$ will be exactly the fraction of points in the LIS. So the verifier will accept with high probability. On the other hand, suppose the inner protocol guarantees that the prover provides consistent answers. Then the probability that $f(i) = g(i)$ will never exceed λ_f. Hence, if $\lambda_f < \lambda - \delta$, the verifier will almost certainly reject the proof. Hence the protocol guarantees soundness.

The number of rounds of communication required by the inner protocol depends on the indices that the prover selects to reveal. If, for some constant $\rho \in (0, 1/2)$, the prover always selects the index to be a *ρ-balanced* which means that it is at least an ρ-fraction away from both endpoints of the interval, then the number of rounds is $O(\frac{\log n}{\rho})$.

With this protocol in mind, we turn to describing the main ideas of the algorithm. It is convenient to identify the array/function as the set of points $\{(i, f(i)\}$ in $\mathbb{R}^2$. We take the natural partial order where $(a_1, a_2) \preceq (b_1, b_2)$ iff $\forall i, a_i \preceq b_i$. The LIS is now the size of the longest chain in this partial order. We use the term *interval* for an interval along the x-axis, which is just an interval in the domain of f. We say two points P and Q are *inconsistent*, or that P is a *violation* with Q, if the points are incomparable in this partial order.

The first idea for developing an algorithm is to simulate the prover algorithmically. During the inner protocol, the prover is repeatedly given an interval I of indices, and asked to provide a point $P = (x, y)$ with $x \in I$. We call such a point a *splitter* for the interval. The splitter provided by the prover is supposed to be both consistent with the LIS and ρ-balanced. So to simulate the prover, we want to give an algorithm with the following property. When given an interval I, the algorithm finds a splitter which is ρ-balanced and is *nearly consistent* with an LIS. This implies there is an increasing sequence that is consistent with the splitter that is not much shorter than the LIS. Let $LIS(P)$ be the longest increasing sequence consistent with P. The *cost* of the splitter P is $(|LIS| - |LIS(P)|)/|I|$. If we simulate the interactive proof and always find a splitter that has cost at most μ (where μ is some parameter), and is ρ-balanced, then it can be shown that the probability that the verifier (or rather our algorithm simulating the verifier) sees that $f(i) = g(i)$ is at least $\lambda_f - \mu r$ where $r = O(\log n/\alpha)$. Thus it would be sufficient if we could always find such a splitter whose cost μ is a small fraction of $\rho/\log n$.

Since we do not know what the LIS is, how do we find a splitter of low cost, or even check that a candidate splitter has low cost? One way is to be conservative and require that the splitter be consistent with nearly all of the indices in I. Given a potential splitter P, we consider the fraction (out of I) of violations with P. We remind the reader that this is the fraction of points $(i, f(i))$ with $i \in I$ that are inconsistent with P. This fraction of violations is an upper bound on the cost of the potential splitter P. Note that we can easily estimate the fraction of violations by random sampling within I. We can safely choose P as a splitter if the fraction of violations it has within I is less than μ. Our candidate splitters will themselves be obtained by taking a random sample of I, and for each $i \in I$ checking whether $(i, f(i))$ meets the above requirements for a splitter. (It might seem useful to also consider possible splitters that do not coresond to points $(i, f(i))$, but our algorithm will not need to.) If we identify a splitter we can proceed with the simulation of the proof.

Of course, this search may not succeed in finding a suitable splitter. To deal with this situation, we consider another approach, that of boosting the quality of the approximation. Suppose we have an additive δ-approximation to λ_f. Can we use it to get an additive δ'-approximation for some smaller δ'? If we could, then by using the known additive $1/2$-approximation algorithm as a starting point, we might be able to recursively combine these algorithms into one that achieves any desired error.

Consider an interval I, and suppose we want to estimate the length of the LIS within I. Suppose we have an index i

with the property that for any choice of h the point (i, h) is in violation with at least a μ-fraction of the points $(j, f(j))$ for $j \in I$. So (i, h) violates the criterion for a splitter described previously. For each real number h, let $I_L(h)$ and $I_R(h)$, respectively, denote the set of points, in each half that would be consistent with the splitter (i, h). For convenience, let $LIS(J)$ be the size of the LIS in interval J. By the definition of the LIS, there is some value h^* that is consistent with the LIS, i.e. such that $LIS(I) = LIS(I_L(h^*)) + LIS(I_R(h^*))$. By the assumption about the index i this is less than $(1 - \mu)|I|$.

Now our algorithm proceeds as follows. Choose an independent random sample of x-coordinates from I of size $\operatorname{poly}\log n$ and consider the set of corresponding y coordinates. With high probability, among these values is a number y^* that it is inconsistent with only a small fraction of the LIS. Formally, $LIS(I_L(y^*)) + LIS(I_R(y^*)) \geq LIS(I) - |I|/\log(n)$. We will ignore the last "error" term $|I|/\log n$ and just assume equality. For each h in our sample of y values, use the given δ-approximation algorithm to estimate the length of the LIS in each of $I_L(h)$ and $I_R(h)$. The sum of these estimates is our estimate of the length of the LIS consistent with value h. For $h = y^*$, we will have that the LIS length exceeds this sum by at most $\delta(|I_L(y^*)| + |I_R(y^*)|)$. Since (i, y^*) has at least $\mu|I|$ violations within I, $|I_L(y^*)| + |I_R(y^*)| \leq (1 - \mu)|I|$. Thus $\delta(|I_L(y^*)| + |I_R(y^*)|)$ is within an additive $\delta(1 - \mu)|I|$ of LIS length.

However, our algorithm does not know which index is y^*. So it computes the sum $LIS(I_L(h)) + LIS(I_R(h))$ for all h in the sample and takes the maximum of these. It can be shown that this maximum is also within $delta(1 - \mu)|I|$ of the length of the LIS. (Note that the computation of the estimate is a very small dynamic program, since we are maximizing over h the sum of the estimates produced by two subproblems.)

Thus, it seems we have a useful dichotomy: if we find a value for h for which (i, h) has at most $\mu|I|$ violations within I, then we can proceed with the simulation of the interactive proof. If, on the other hand, there is no such h, then we can boost an existing approximation algorithm.

It is not clear how to put these two approaches together to get an algorithm, but there is a more significant difficulty. For the simulation of the interactive protocol to work, we argued that μ should be $O(1/\log n)$. For the recursive boosting to work efficiently, we will need μ to be at least $\Omega(1/\log\log n)$. Why? For each level of recursion, we can improve the additive approximation from δ to $\delta(1 - \mu)$. So we will need $1/\mu$ levels of recursion to improve from δ to $\delta/2$. At each level of the recursion, we make at least 2 recursive calls, for the left and right subproblems generated by any choice of h. So the total number of iterated recursive calls is exponential in $1/\mu$, forcing μ to be $\Omega(1/\log\log n)$ to keep the time polylogarithmic in n. Although the dichotomy seems promising, we have a huge gap from an algorithmic perspective.

To close this gap we will refine both the above procedure for searching for a splitter and the boosting procedure. We begin by modifying the criterion used to identify a splitter. In the old procedure, we considered the total fraction of violations of the splitter (i, h) within I. Now, ee consider every interval J within I that has i as an endpoint and has size at least $\gamma|I|$, where γ is some small appropriately chosen parameter. We now require that the candidate splitter have the property that for every such J the number of violations of the spitter with J should be at most $\mu|J|$. (This is similar to the type of test described earlier as part of the approximation algorithm of [ACCL07].) While it looks like we are imposing a more stringent condition on splitters, the advantage of this condition is that we can set μ to be much larger than $O(1/\log n)$. It can be shown that a splitter which satisfies the above condition with a small constant μ and $\gamma = O(1/\log n)$ is a sufficiently good splitter. The new condition with these parameters is actually less restrictive than the original condition with $\mu = O(1/\log n)$.

So we can ask: what if we have an index i for which there is no h that satisfies the above condition for μ and γ? Is this enough to use the boosting algorithm? The answer, unfortunately, is no. But, by introducing a more sophisticated version of the boosting algorithm, we are able to get the dichotomy we need. For a given interval I, suppose the search algorithm for an ρ-balanced splitter satisfying the above conditions for μ, γ fails. Then the improved boosting algorithm, can convert a δ-approximation algorithm to a $\delta(1 - \mu)$-approximation algorithm. We finish this section by sketching the idea of the better boosting algorithm. Divide the interval I into s equal-sized intervals $I_1, \ldots, I_s$ with s polylogarithmic in n. Suppose we can find values $h_0, h_1, h_2, \ldots, h_s$ that are *consistent* with the LIS. This means that the values of LIS points in I_r are at least h_{r-1} and at most h_r. Refer to the left part of Figure 3. Using these values, we can estimate the LIS length in I. For each subinterval I_r, we look at all input points in I_r with values between h_r and h_{r+1}. For this set of points, we use an additive δ-approximation algorithm to approximate the LIS length. We then add up these estimates over all h, to get our estimate for the LIS length in I. Under the hypothesis that there is no good splitter for some balanced partition I_L, I_R of I, we can show that this sum is a $\delta(1-\mu)$-approximation to the LIS.

Since we do not know the values $h_1, \ldots, h_s$, we need to search for approximate versions of them. This will be done by finding the longest path in a DAG, which can be solved by a (small) dynamic program. We choose a random sample of $\operatorname{poly}\log n$ points from each I_i and let H_i denote the set of y-coordinates of these points. The set H_i is the set of candidate h_i values (for the boundary, we set H_0 (resp. H_s) to have the minimum (resp. maximum)

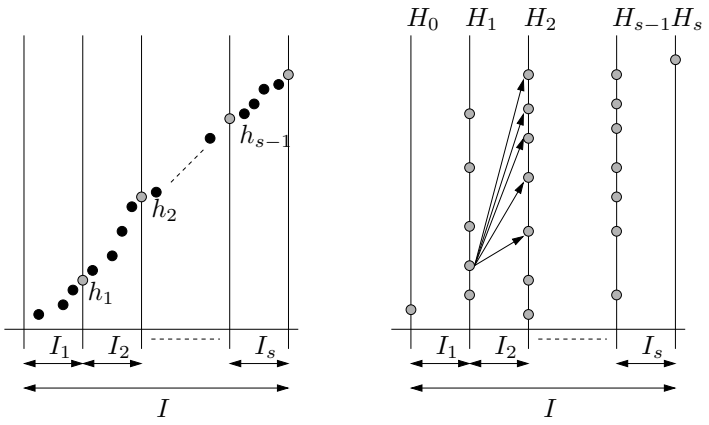

Figure 3. The figure to the left shows the placement of possible values $h_1, h_2, \ldots$. The black points for the LIS and the grey points correspond to the h values. The figure to the right shows the construction of the graph G, and the layers of vertices H_i. The arrows give the directed edges from one vertex in layer H_1.

value in I). We need to find the best h_i values among these candidates. We construct the following directed layered graph with $(t+1)$ layers. Refer to the right part of Figure 3. Layer i has vertices for each value in H_i. We put a directed edge between $h_i \in H_i$ and $h_{i+1} \in H_{i+1}$ if $h_i \leq h_{i+1}$. Consider the set of all points in I_{i+1} with values in the range $[h_i, h_{i+1}]$. We estimate the LIS length of this set using the δ-approximation, and set this to be the weight of the edge (h_i, h_{i+1}). Observe that this graph has polylogarithmic size, and all weights can be computed with polylogarithmically many calls to the additive δ-approximation algorithm. For a path $h_0, h_1, \ldots, h_r$, the length is an estimate of the LIS consistent with these values. So it is natural to use the longest path as our estimate for the total LIS in I. Since this graph is a DAG, the longest path can found in time polynomial in the graph size, which is polylogarithmic in n.

These two ingredients - the simulation of the interactive protocol and the improved boosting algorithm - are combined together to give our algorithm. There are various parameters such as ρ, μ, γ involved in the algorithm, and we must choose their values carefully. We also need to determine how many levels of boosting are required to get a desired approximation. A direct choice of parameters leads to a $(\log n)^{1/\delta}$ approximation algorithm. The better algorithm claimed in Theorem 1.2 is obtained by a more involved version of the algorithm, which involves modifying the various parameters as the algorithm proceeds. This enables us to reduce the number of recursive calls needed for the basic boosting algorithm from polylogarithmic to a constant depending on δ.

In this extended abstract, we have space only to formally state the weaker algorithm that gives Theorem 1.1, and to state some of the key lemmas. The authors invite the interested reader to see the full version [SS10] for all the details.

III. Preliminaries

We introduce our notation and define many important concepts. As mentioned earlier, it will be convenient to think of the input as a set $\mathcal{F}$ of (geometric) points $\{(i, f(i))\}$ in $\mathbb{R}^2$, with x and y denoting the axes. We will assume, wlog, that all coordinates are positive and distinct[1]. We take the natural partial order where $(a_1, a_2) \preceq (b_1, b_2)$ iff $\forall i, a_i \preceq b_i$. An increasing sequence is just a chain in this partial order. A point P is to the *left* (resp. right) of Q, if the x-coordinate of P is less (resp. more) than that of Q. We use capital letters to denote points, calligraphic letters for sets of points, and bold letter for collections of sets of points. Two points P, Q are *consistent* if $P \prec Q$ or $Q \prec P$. Otherwise, they form a *violation*.

We now list out some basic definitions that will be used throughout the paper.

Box: A *box* $\mathcal{B}$ is the set of input points *internal* to an axis-parallel rectangle in the place. (We do not include points on the boundary in $\mathcal{B}$.) Although intuitively a box is a geometric object, formally, it is just a set of points in $\mathcal{F}$. Abusing notation, we will refer to corners of $\mathcal{B}$ as if it is a axis-parallel rectangle in the plane. A *$\mathcal{B}$-point* refers to a point inside $\mathcal{B}$. Our algorithm will represent and store a box $\mathcal{B}$ by just its corners (a constant size representation), and will not explicitly store the set $\mathcal{B}$. So when a box $\mathcal{B}$ is passed as an argument, we just pass this representation.

The *size* of box $\mathcal{B}$ is the number of points, also denoted by $|\mathcal{B}|$. We let $\varepsilon_{\mathcal{B}}$ denote the distance to monotonicity for the input points in $\mathcal{B}$. Given a pair of points $(P, Q), P \preceq Q$, the box $Box(P, Q)$ is formed by taking the axis-parallel rectangle formed by points P and Q. We will say $Box(P, Q) \preceq Box(P', Q')$ if $Q \preceq P'$ (so all points in the first box are

[1]We use a simple tie-breaking rule for two y-coordinates that are the same. So for $i < j$ if $f(i) = f(j)$, the algorithm will assume that $f(i) < f(j)$.

dominated by all points in the latter). We use $\mathcal{F}$ to also denote the box containing all points.

Chain: A *chain of boxes* is a sequence of boxes $\mathcal{B}_1 \preceq \mathcal{B}_2 \preceq \mathcal{B}_3 \cdots$. A *maximal chain of boxes* is a chain where the upper right corner of $\mathcal{B}_i$ has the same y-coordinate as the lower right corner of $\mathcal{B}_{i+1}$. The size of a chain is the total number of points in the chain.

Strip: For a box $\mathcal{U}$, the $\mathcal{U}$-strip is a box that has the same vertical extent as $\mathcal{U}$. We use $x(\mathcal{B})$ to denote the x-interval corresponding by the $\mathcal{B}$. Let $\mathcal{B},\mathcal{U}$ be boxes such that $\mathcal{B} \subseteq \mathcal{U}$. The *$\mathcal{U}$-strip* of a box $\mathcal{B}$, denoted by $st(\mathcal{B},\mathcal{U})$, is the box of all points in $\mathcal{U}$ with x-coordinates in $x(\mathcal{B})$. In other words, $st(\mathcal{B},\mathcal{U})$ is a box formed by the x-extent of $\mathcal{B}$ and the y-extent of $\mathcal{U}$. The points (if any) contained in the left and right edges of the rectangle corresponding to $st(\mathcal{B},\mathcal{U})$ are the *endpoints* of the strip. (By the uniqueness of coordinates, the left and right endpoints, if they exist, are unique.) The *width* of box $\mathcal{B}$ is the size of the strip $st(\mathcal{B},\mathcal{F})$ and is denoted by $w(\mathcal{B})$.

Grid: Consider a box $\mathcal{B}$. The *grid*, $\text{Grid}_{\mathcal{B}}(r_x, r_y)$, is formed by a set of vertical and horizontal lines inside this box. A *grid point* is simply a point that is an intersection point for some vertical and horizontal line of that grid. A *grid box* for $\text{Grid}_{\mathcal{B}}(r_x, r_y)$ is a box formed by taking two consistent grid points on *adjacent* vertical lines. Note that grid boxes can overlap, but their x-intervals never have a non-trivial intersection. $\text{Grid}_{\mathcal{B}}(r_x, r_y)$ has a crucial property: the number of $\mathcal{B}$-points between two adjacent vertical lines is at most $r_x|\mathcal{B}|$. The total number of vertical lines is at most $2/r_x$. The number of $\mathcal{B}$-points between adjacent horizontal lines is at most $r_y|\mathcal{B}|$. There are at most $2/r_y$ horizontal lines.

The only points that we will ever use are either input points (in $\mathcal{F}$) or grid points. Henceforth, we will just use *point* for an input point. For clarity, we will always use *grid point* for the corresponding concept.

A crucial definition is that of *safeness* of points. Since it involves many parameters and the reader may have to come back to it often, we put it in a separate environment. This definition is inspired from [ACCL07] (which is in turn based on ideas from [EKK+00]).

Definition 3.1: Let $\mathcal{B} \subseteq \mathcal{U}$. A point $P \in \mathcal{B}$ is $(\mathcal{B},\mathcal{U},\gamma,\mu)$-*safe* if the following is true: take any $\mathcal{U}$-strip $\mathcal{S}$ with P as an endpoint such that $\mathcal{S} \subseteq st(\mathcal{B},\mathcal{U})$ and $|\mathcal{S}| \geq \gamma|st(\mathcal{B},\mathcal{U})|$. The number of violations with P in this strip is at most $\mu|\mathcal{S}| + (\alpha^2/\log n)w(\mathcal{B})$.

A point P is $(\mathcal{B},\mathcal{U},\gamma,\mu)$-*unsafe* if the above does not hold: some $\mathcal{U}$-strip $\mathcal{S}$ with endpoint P and size $\geq \gamma|st(\mathcal{B},\mathcal{U})|$ has at least $\mu|\mathcal{S}| + (\alpha^2/\log n)w(\mathcal{B})$ violations with P.

Finally, we list out some auxiliary procedures used by our LIS estimation algorithm. These are obtained via routine random sampling calculations. Hence, we will not describe these completely here, but merely state the relevant claims.

We need to deal with a slight technical issue. Our input is given as an array, and we can easily sample indices at random from any desired interval. We often need to get a random sample from a box $\mathcal{B}$. Unfortunately, we don't have direct access to the points of $\mathcal{B}$. The box $\mathcal{B}$ spans an interval I, but some (or even most) of the points $(i, f(i))$ for $i \in I$ may be outside of the box. So to sample from $\mathcal{B}$, we sample randomly from I and take only those points $(i, f(i))$ that lie in $\mathcal{B}$. This works provided that $|\mathcal{B}|$ is a large enough fraction of $w(\mathcal{B})$, If not, then we must analyze these boxes in a different way. It turns out that we can essentially ignore such boxes. This leads to the following definition.

Definition 3.2: A box $\mathcal{B}$ is *disposable* if $|\mathcal{B}| \leq \alpha^2 w(\mathcal{B})/\log n$.

We now state the claims about various auxiliary procedures used. Our procedures sometime output *labels*, usually indicating a terminal or exceptional case. These will be represented in typewriter typeface. Greek letters will always represent parameters taking values in $(0, 1)$. The parameter α is a sort of error parameter, and is chosen based on the desired additive error for the LIS. We express the running times using the big-Oh notation, and stress that this only hides absolute constants. We do *not* consider any of our parameters to be constants, even though they may finally be set to constant values.

Claim 3.3: **(Size)** There is a procedure $Size(\mathcal{B})$ that outputs either outputs an estimate of $|\mathcal{B}|$ or labels $\mathcal{B}$ as `disposable`. The running time is $(\log n/\alpha)^{O(1)}$ and the following hold with high probability. If an estimate is output, then it is correct up to an additive $\alpha^3 w(\mathcal{B})/\log n$. If the label is output, then $\mathcal{B}$ is disposable.

Claim 3.4: **(Sample)** There is a procedure $Sample(\mathcal{B}, k)$ that outputs either a set of k points in $\mathcal{B}$ or labels $\mathcal{B}$ as `disposable`. The running time is $(k \log n/\alpha)^{O(1)}$ and the following hold with high probability. If a set is output, then each point in the set is an independent random sample from $\mathcal{B}$. If the label is output, then $\mathcal{B}$ is disposable.

Claim 3.5: **(Grid)** There is a randomized procedure $Grid(\mathcal{B}, r_x, r_y)$ that with high probability, either outputs $\text{Grid}_{\mathcal{B}}(r_x, r_y)$ or labels $\mathcal{B}$ correctly as `disposable`. The running time is $(\log n/(\alpha r_y))^{O(1)}$ time.

Claim 3.6: **(Find)** There is a procedure $Find(\mathcal{B},\mathcal{U},\gamma,\mu,\rho)$ that outputs either a point s, the label `sparse`, or the label `disposable`. The running time is at most $(\log n/(\alpha\gamma\mu\rho))^{O(1)}$ and the following hold with high probability. If the output is s, then s is $(\mathcal{B},\mathcal{U},2\gamma,\mu+\alpha)$-safe and there are at least $\rho|\mathcal{B}|/5$ points in $\mathcal{B}$ both to the left and right of s. If the output is `sparse`, the number of $(\mathcal{B},\mathcal{U},\gamma,\mu-\alpha)$-safe points in $\mathcal{B}$ is at most $\rho|\mathcal{B}|/2$. If the output is `disposable`, then $\mathcal{B}$ is disposable.

IV. The basic algorithm

We now describe the algorithm that gives Theorem 1.1. The output guarantee in its most general form is both a

multiplicative and additive approximation to the distance to monotonicity. We first state the main theorem of this section, which gives this guarantee. We can derive the desired purely additive approximation for the LIS and purely multiplicative approximation for the distance by a simple choice of parameters.

Theorem 4.1: Let $f : [n] \to \mathbb{R}$ be an array and ε_f be the distance to monotonicity. Let positive parameters δ, τ be at most some small constant, and $\delta \leq \tau$.

There exists a procedure that, given oracle access to f, outputs a real number ε such that with high probability $\varepsilon_f \in [\varepsilon, (1+\tau)\varepsilon + \delta]$. The running time of this procedure is $(\log n/\delta)^{O(1/\tau)}$.

We describe the choice of parameters in Theorem 4.1 that leads to Theorem 1.1. For convenience, let the parameters in Theorem 4.1 be τ' and δ', so we get an output ε such that $\varepsilon_f \in [\varepsilon, (1+\tau')\varepsilon + \delta']$. If we set $\tau' = \delta' = \delta/2$, then we get an estimate ε such that $\varepsilon_f \leq (1+\tau')\varepsilon + \delta' \leq \varepsilon + \delta$. So $n(1-\varepsilon)$ is an additive δn-approximation for the LIS. A different choice of parameter values leads to a multiplcative $(1+\tau)$-approximation of the distance (see full version for details).

A. The procedures $ApproxLIS$ and $Classify$

We are now ready to describe the procedure $ApproxLIS$. Since we have an iterative boosting technique, we actually have a suite of procedures $ApproxLIS_t$, for integer $t > 0$. These form a sort of hierarchy of procedures. The aim of the procedure $ApproxLIS_t(\mathcal{U})$ is to output an estimate for the LIS length in the box $\mathcal{U}$. For convenience, we assume that such an estimate is computed wherever needed. $ApproxLIS_t$ indirectly uses $ApproxLIS_{t-1}$ as a subroutine. As the parameter t increases, the quality of the approximation increases.

The heart of the algorithm is actually the procedure $Classify_t(P,\mathcal{U})$, which implicitly constructs an increasing sequence for $\mathcal{U}$ by deciding whether point P should be included in the LIS. Concretely, it labels the points in $\mathcal{U}$ as $good_t$ or bad_t. While the procedure will only be called on a small subset of points, it has the property that if it were run on all points, the good$_t$ points form an increasing sequence and the fraction of bad$_t$ points is an approximation of the distance.

The procedure $Classify_t(P,\mathcal{U})$ begins by calling a procedure $Unsplittable$. This procedure basically tries to run the interactive protocol, and can be thought of as our prover. It tries to find a splitter in $\mathcal{U}$. Suppose it is successful and P is consistent with this splitter. Then it goes to a smaller box containing P and tries to find a splitter again. This process is repeated until either P turns out to be inconsistent with a splitter, or P is located in an *unsplittable* box. This is a box where $Find$ fails to locate a splitter. So $Unsplittable$ either returns a label `reject`, indicating that P is inconsistent with some splitter, or a box $\mathcal{B} \ni P$ that contains very few (too few to sample) potential splitters.

This box $\mathcal{B}$ is returned to $Classify_t$. A grid is built on $\mathcal{B}$ and then $ApproxLIS_{t-1}$ is invoked on all the (logarithmically many) grid boxes. The outputs of these calls are used to determine the structure of an approximate LIS. Finally, if P is potentially on the approximate LIS, an call to $Classify_{t-1}(P,\mathcal{C})$ (for an appropriate grid box $\mathcal{C}$) is made to determine whether to label P as good$_t$. There are some corner cases that occur when a box $\mathcal{B}$ is disposable, and $w(\mathcal{B})$ is extremely small.

$ApproxLIS_t(\mathcal{U})$ $(t > 0)$

1) Run $Sample(\mathcal{U}, c\log^2 n/\alpha^2)$ and call $Size(\mathcal{U})$ to get estimate u. If $Sample$ or $Size$ outputs `disposable`, output 1 as estimated distance and 0 for LIS estimate. Terminate procedure. Otherwise, we have a random sample $\mathcal{R}$.
2) Call $Classify_t(P,\mathcal{U})$ for each $P \in \mathcal{R}$ and let the fraction of bad$_t$ points be ε.
3) Output ε as the estimate of $\varepsilon_{\mathcal{U}}$ and $(1-\varepsilon)u$ times as the estimate of the LIS in $\mathcal{U}$.

$Classify_t(P,\mathcal{U})$ $(t > 0)$

1) Run $Unsplittable_t(P,\mathcal{U},\mathcal{U})$. If this outputs `failure`, output *bad*$_t$ and terminate. Otherwise, we get box $\mathcal{B}$.
2) If $w(\mathcal{B}) \leq \log n/\alpha$, find the LIS of $\mathcal{B}$. If p is on the LIS output *good*$_t$, otherwise output *bad*$_t$. Terminate procedure.
3) Call $Grid(\mathcal{B}, \alpha_t^3/\log n, \alpha_t^4/\log n)$ to generate a grid for $\mathcal{B}$. For each grid box $\mathcal{C}$, call $ApproxLIS_{t-1}(\mathcal{C})$ to get an estimate on the LIS length in $\mathcal{C}$. Let this estimate be the *length* of the $\mathcal{C}$.
4) Determine the longest length grid chain (by dynamic programming) $\mathcal{C}'_1, \mathcal{C}'_2, \cdots$. If p does not belong to the longest grid chain, output *bad*. Otherwise, let $\mathcal{C}'_j$ be the grid box containing p. Run $Classify_{t-1}(P,\mathcal{B})$. If p is good$_{t-1}$, output *good*$_t$. Otherwise, output *bad*$_t$.

$Unsplittable_t(P,\mathcal{B},\mathcal{U})$

1) Run $Find(\mathcal{B},\mathcal{U},\gamma_t,\mu_t,\alpha)$. If the output is `disposable`, output `reject` and terminate procedure.
2) If $Find(\mathcal{B},\mathcal{U},\gamma_t,\mu_t,\alpha)$ outputs point S: Call this the *splitter* for $\mathcal{B}$.
   a) If P is a violation with S, output `reject` and terminate procedure.
   b) Let $\mathcal{B}$ be $Box(B_l, B_r)$. Set $\mathcal{B}_l = Box(B_l, S)$ and $\mathcal{B}_r = Box(S, B_r)$. Let $\mathcal{B}'$ be the box among these that contains P. (If $P = S$, then set $\mathcal{B}'$ arbitrarily.)
   c) Recursively call $Unsplittable_t(P,\mathcal{B}',\mathcal{U})$.
3) If $Find(\mathcal{B},\mathcal{U},\gamma_t,\mu_t,\alpha)$ outputs `sparse`, output $\mathcal{B}$.

For the base case of $Classify$ and $ApproxLIS$, we use trivial algorithms. The procedure $ApproxLIS_0$ always outputs 0 as the estimated length of the LIS. The procedure $Classify_0$ always labels a point as bad_0.

We observe that $ApproxLIS_r$ (for various values r) is invoked on many different boxes. So we estimate the LIS length on various boxes and piece the information together to estimate the overall LIS. This leads to a fairly complex recursive scheme. Hence, there will be an array of different parameters that are used by these procedures. These are intimately related to the approximation guarantees that will be finally provided. The parameter α is just some sort of error parameter and is chosen depending on the final desired approximation. The reader should just think of this as something extremely small. If we invoke some $ApproxLIS_t$, we will always choose α so that it is at most $\zeta_t^2/30$. The following values are the only choice of arguments that will be used by the various procedures.

- $\zeta_2 = 2.1$, $\zeta_t = \zeta_{t-1} - \zeta_{t-1}^2/40$
- $\xi_2 = \alpha$, $\xi_t = (1+\zeta_t)\xi_{t-1} + \alpha$
- $\mu_1 = 0.5$, $\mu_t = \zeta_t/4$, $\gamma_t = \alpha^2/30$

We observe that $Classify_t(P, \mathcal{B})$ labels a point P *with respect to a box* $\mathcal{B}$. We use a shorthand for saying that the output of $Classify_t(P, \mathcal{B})$ is good_t. We just say that P is labeled $\text{good}_t(\mathcal{B})$ (or $\text{bad}_t(\mathcal{B})$). We refer to t as the *level number*, so it denotes the number of levels of recursion used.

Before we state our main lemma, it is important to look at the use of randomness by these procedures. This is somewhat subtle and we discuss it in the next subsection. Then, we shall be ready to state our main approximation boosting lemma.

1) The use of randomness: Randomess is used through the invocations of $Sample$, $Find$, and $Grid$. It will be important for us that these procedures give the same output when given the same arguments. This will ensure consistency over various calls to the procedures. Since these are randomized algorithms, independent calls to them will give different outputs. For this purpose, we associate a fixed random seed for a given procedure with a given set of arguments.

All calls to a procedure made during a fixed level t with the same arguments will use the same random seed. For example, there is a fixed random seed associated with $Find(\mathcal{B}, \mathcal{U}, \gamma_t, \mu_t, \alpha)$, for a fixed choice of these arguments. Every call to this procedure will use the same random seed, thereby ensuring the output is always the same.

Claim 4.2: With high probability over the choice of random seeds, no invocation to $Find$, $Grid$, $Size$, or $Sample$ made by any call to $ApproxLIS_t, Classify_t$, or $Unsplittable_t$ will fail.

Suppose we focus on a fixed t. For every possible of call to $Find$, $Grid$, $Size$, or $Sample$, every level number $\leq t$, and every possible choice of arguments, we can write down a random seed. (Calls at *different* levels use independent random seeds.) The total list of these random seeds is the seed for $ApproxLIS_t$. Note that *any* call to $ApproxLIS_t(\mathcal{U})$ (for any set $\mathcal{U}$) behaves deterministically, now that the random seed is fixed. Note that we do not really have to know the whole seed in advance, but can generate it as $ApproxLIS_t$ proceeds. The key is that the random seed is reused, whenever an auxiliary procedure is repeatedly called with the same argument. Note a major benefit of this approach. For a level r and box $\mathcal{B}$, the labels $good_r(\mathcal{B})$ and $bad_r(\mathcal{B})$ are now fixed. Since this labeling is now static, it is convenient to make arguments about these points.

Definition 4.3: A random seed of $ApproxLIS_t$ is *sound* if the following hold. No call to any auxiliary procedure with any choice of arguments fails. For box $\mathcal{U}$ and level r, let $g_r(\mathcal{U})$ (resp. $b_r(\mathcal{U})$) be the number of $\text{good}_r(\mathcal{U})$ (resp. $\text{bad}_r(\mathcal{U})$) points. Let the LIS estimate of $ApproxLIS_r(\mathcal{U})$ be $\ell(\mathcal{U})$ and the distance estimate be $\varepsilon(\mathcal{U})$. For all boxes $\mathcal{U}$ and $r \leq t$, we have $|\varepsilon(\mathcal{U})|\mathcal{U}| - b_r(\mathcal{U})| \leq \alpha|\mathcal{U}| + \alpha^2 w(\mathcal{U})/\log n$ and $|\ell(\mathcal{U}) - g_r(\mathcal{U})| \leq \alpha|\mathcal{U}| + 2\alpha^2 w(\mathcal{U})/\log n$.

Claim 4.4: Let $t \leq \log n$. With probability at most $n^{-\Omega(\log n)}$, a seed for $ApproxLIS_t$ chosen uniformly at random is unsound.

Claim 4.5: For a fixed random seed, all $\text{good}_t(\mathcal{U})$ points form an increasing sequence.

Henceforth, we will assume that a *sound random seed* is one that is sound for a sufficiently large value of t (say $\log n$). This means that, using this random seed, all calls to any of our procedures will succeed.

B. Iterative boosting

The proof of Theorem 4.1 is an induction argument. Assuming that $Classify_{t-1}$ has some good guarantee, we want to show that $Classify_t$ has a better guarantee. By quantifying this improvement, we can determine what the value of t should be for some desired approximation. We will state the boosting lemma. This will first have a "base case", expressing the guarantee of $Classify_1$. Then, we have a claim that is the "induction step", showing how approximations are improved. It will actually be convenient for the analysis to express the approximation guarantee of $Classify_{t-1}$ using ζ_t and ξ_t (instead of the $(t-1)$-subscript versions).

Lemma 4.6: Fix a sound seed.

- The number of $\text{bad}_1(\mathcal{U})$ points is at most $[(1+\zeta_2)\varepsilon_{\mathcal{U}} + \xi_2]|\mathcal{U}| + c\alpha^2 w(\mathcal{U})/\log n$.
- Suppose, for any box $\mathcal{B}$, the number of $\text{bad}_{t-1}(\mathcal{B})$ points is at most $[(1+\zeta_t)\varepsilon_{\mathcal{B}} + \xi_t]|\mathcal{B}| + ct\alpha^2 w(\mathcal{B})/\log n$ in number. Then, for any box $\mathcal{C}$, the number of $\text{bad}_t(\mathcal{B})$ points is at most $[(1+\zeta_{t+1})\varepsilon_{\mathcal{C}} + \xi_{t+1}]|\mathcal{C}| + c(t+1)\alpha^2 w(\mathcal{C})/\log n$ in number.

The proof of this lemma is long and appears in the full paper. Theorem 4.1 follows fairly directly from this lemma. and the following technical claim.

Claim 4.7: There exists $t \leq c/\tau$, such that $\zeta_t \leq \tau$ and $\xi_t \leq \alpha/\tau^{c_1}$.

Essentially, choosing $\alpha = \delta\tau^{O(1)}$ and setting $t = \Omega(1/\tau)$, we show that $ApproxLIS_t$ has the guarantees stated in Theorem 4.1.

V. Acknowledgements

The first author was supported in part by NSF under CCF 0832787. The second author would like to thank Robi Krauthgamer and David Woodruff for useful discussions.

References

[ACCL07] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Estimating the distance to a monotone function. *Random Structures and Algorithms*, 31(3):371–383, 2007.

[AD99] D. Aldous and P. Diaconis. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johannson theorem. *Bulletin of the American Mathematical Society*, 36:413–432, 1999.

[AIK09] A. Andoni, P. Indyk, and R. Krauthgamer. Overcoming the ℓ_1 non-embeddability barrier: algorithms for product matrices. In *Proceedings of the 20th Symposium on Discrete Algorithms (SODA)*, pages 865–874, 2009.

[AK07] A. Andoni and R. Krauthgamer. The computational hardness of estimating edit distance. In *Proceedings of the 48th Symposium on Foundations of Computer Science (FOCS)*, pages 724–734, 2007.

[AK08] A. Andoni and R. Krauthgamer. The smoothed complexity of edit distance. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 357–369, 2008.

[AN10] A. Andoni and H. L. Nguyen. Near-optimal sublinear time algorithms for ulam distance. In *Proceedings of the 21st Symposium on Discrete Algorithms (SODA)*, 2010.

[BGJ+09] A. Bhattacharyya, E. Grigorescu, K. Jung, S. Raskhodnikova, and D. Woodruff. Transitive-closure spanners. In *Proceedings of the 18th Annual Symposium on Discrete Algorithms (SODA)*, pages 531–540, 2009.

[DGL+99] Y. Dodis, O. Goldreich, E. Lehman, S. Raskhodnikova, D. Ron, and A. Samorodnitsky. Improved testing algorithms for monotonicity. *Proceedings of the 3rd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 97–108, 1999.

[EJ08] F. Ergun and H. Jowhari. On distance to monotonicity and longest increasing subsequence of a data stream. In *Proceedings of the 19th Symposium on Discrete Algorithms (SODA)*, pages 730–736, 2008.

[EKK+00] F. Ergun, S. Kannan, R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. *Journal of Computer Systems and Sciences (JCSS)*, 60(3):717–751, 2000.

[Fis01] E. Fischer. The art of uninformed decisions: A primer to property testing. *Bulletin of EATCS*, 75:97–126, 2001.

[FLN+02] E. Fischer, E. Lehman, I. Newman, S. Raskhodnikova, R. Rubinfeld, and A. Samorodnitsky. Monotonicity testing over general poset domains. In *Proceedings of the 34th Annual Symposium on Theory of Computing (STOC)*, pages 474–483, 2002.

[Fre75] M. Fredman. On computing the length of the longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[GG07] A. Gal and P. Gopalan. Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence. In *Proceedings of the 48th Symposium on Foundations of Computer Science (FOCS)*, pages 294–304, 2007.

[GGL+00] O. Goldreich, S. Goldwasser, E. Lehman, D. Ron, and A. Samordinsky. Testing monotonicity. *Combinatorica*, 20:301–337, 2000.

[GGR98] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, 1998.

[GJKK07] P. Gopalan, T. S. Jayram, R. Krauthgamer, and R. Kumar. Estimating the sortedness of a data stream. In *Proceedings of the 18th Symposium on Discrete Algorithms (SODA)*, pages 318–327, 2007.

[Gol98] O. Goldreich. Combinatorial property testing - a survey. *Randomization Methods in Algorithm Design*, pages 45–60, 1998.

[HK03] S. Halevy and E. Kushilevitz. Distribution-free property testing. *Proceedings of the 7th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 302–317, 2003.

[PRR06] M. Parnas, D. Ron, and R. Rubinfeld. Tolerant property testing and distance approximation. *Journal of Computer and System Sciences*, 6(72):1012–1042, 2006.

[Ron01] D. Ron. Property testing. *Handbook on Randomization*, II:597–649, 2001.

[RS96] R. Rubinfeld and M. Sudan. Robust characterization of polynomials with applications to program testing. *SIAM Journal of Computing*, 25:647–668, 1996.

[SS10] M. Saks and C. Seshadhri. Estimating the longest increasing sequence in polylogarithmic time. 2010. Available at http://www.cs.princeton.edu/∼csesha/lis-full.pdf.

[SW07] X. Sun and D. Woodruff. The communication and streaming complexity of computing the longest common and increasing subsequences. In *Proceedings of the 18th Symposium on Discrete Algorithms (SODA)*, pages 336–345, 2007.