Approximation Algorithms for LCS and LIS with Truly Improved Running Times

Aviad Rubinstein*, Saeed Seddighin†, Zhao Song‡, Xiaorui Sun§

*Computer Science Department, Stanford University, Stanford, California, USA, aviad@cs.stanford.edu
†SEAS, Harvard University, Cambridge, Massachusetts, USA, saeedreza.seddighin@gmail.com
‡Simons Institute for the Theory of Computing, Berkeley, California, USA, magic.linuxkde@gmail.com
§Computer Science Department, University of Illinois at Chicago, Chicago, Illinois, USA, xiaorui@uic.edu

Abstract—Longest common subsequence (LCS) is a classic and central problem in combinatorial optimization. While LCS admits a quadratic time solution, recent evidence suggests that solving the problem may be impossible in truly subquadratic time. A special case of LCS wherein each character appears at most once in every string is equivalent to the longest increasing subsequence problem (LIS) which can be solved in quasilinear time. In this work, we present novel algorithms for approximating LCS in truly subquadratic time and LIS in truly sublinear time. Our approximation factors depend on the ratio of the optimal solution size over the input size. We denote this ratio by \( \lambda \) and obtain the following results for LCS and LIS without any prior knowledge of \( \lambda \).

• A truly subquadratic time algorithm for LCS with approximation factor \( O(\lambda^2) \).
• A truly sublinear time algorithm for LIS with approximation factor \( O(\lambda^3) \).

Triangle inequality was recently used by Boroujeni et al. [1] and Chakraborty et al. [2] to present new approximation algorithms for edit distance. Our techniques for LCS extend the notion of triangle inequality to non-metric settings.

Keywords—approximation; LCS; LIS; subquadratic time;

I. INTRODUCTION

In this paper, we consider three of the most classic problems in combinatorial optimization: the longest common subsequence (LCS), the edit distance (ED), and the longest increasing subsequence (LIS). The LCS of two strings \( A \) and \( B \) is simply their longest (not necessarily contiguous) common subsequence. The edit distance is defined as the minimum number of character deletions, insertions, and substitutions required to transform \( A \) into \( B \). For the purpose of our discussion, we consider a more restricted definition of edit distance where substitutions are not allowed\(^1\). Longest increasing subsequence is equivalent to a special case of LCS where the input strings are permutations. All three problems are very fundamental and have been subject to a plethora of studies in the past few decades and specially in recent years [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [1], [2], [15], [16], [17].

\(^1\)Alternatively, the cost of a substitution is doubled as it requires a deletion and an insertion.

If the strings have length \( n \), both LCS and ED can be solved in quadratic time (\( O(n^2) \)) with dynamic programming. These running times are slightly improved to \( O(n^2/\log^2(n)) \) by Masek and Paterson [18], however, efforts to improve the running time to \( O(n^{2-\Omega(1)}) \) for either edit distance or LCS were all futile.

In recent years, our understanding of the source of complexity for these problems has tremendously improved thanks to a sequence of fine-grained complexity developments [10], [11]. We now know that assuming the strong exponential time hypothesis (SETH) [10], or even weaker assumptions such as the orthogonal vectors conjecture (OVC) [10] or branching-program-SETH [11], there are no truly sub-quadratic\(^2\) time algorithms for LCS. Similar results also hold for edit distance [9].

Indeed, the classic approach to break the quadratic barrier for these problems is approximation algorithms. Note that for (multiplicative) approximations, LCS and edit distance are no longer equivalent (much like we have a 2-approximation algorithm for Vertex Cover, but Independent Set is NP-hard to approximate within near-linear factors).

For edit distance, an \( O(n + \Delta^2) \)-time algorithm of [3] (where \( \Delta \) is the true edit distance between the strings) implies a linear-time \( \sqrt{n} \)-approximation algorithm. The approximation factor has been significantly improved in a series of works to \( O(n^{3/7}) \) [4], to \( O(n^{0.34}) \) [5], to \( O(2^{\tilde{O}(\sqrt{\log n})}) \) [6], and finally to polylogarithmic [7]. A recent work of Boroujeni et al. [1] obtains a constant factor approximation quantum algorithm for edit distance that runs in truly subquadratic time. Finally, the breakthrough of Chakraborty et al. [2] gave a classic (randomized) constant factor approximation for edit distance in truly subquadratic time. A key component in both of the latest constant factor approximation algorithms is the application of triangle inequality (for edit distance between certain substrings of the input). A particular challenge in extending these ideas to LCS is that LCS is not a metric and in particular does

\(^2\)By truly sub-quadratic we mean \( O(n^{2-\epsilon}) \), for any constant \( \epsilon > 0 \)

\(^3\)We define \( O(f) \) to be \( f \cdot \log^{O(1)}(f) \).
not satisfy the triangle inequality.

Our understanding of the complexity of approximate solutions for LCS is embarrassingly limited. For general strings, there are several linear-time $1/\sqrt{n}$-approximation algorithms based on sampling techniques. For alphabet size $|\Sigma|$, there is a trivial $1/|\Sigma|$-approximation algorithm that runs in linear time. Whether or not these approximation factors can be improved by keeping the running time subquadratic is one of the central problems in fine-grained complexity. Very recently, both the general $1/\sqrt{n}$-approximation factor, and, for binary strings, the $1/2$-approximation factor, have been slightly improved ([15] and [19], respectively). These works give improved algorithm for the two extreme cases where the size of the alphabet is very small or very large. In comparison, our approximation guarantee depends on the solution size rather than the size of the alphabet. Also, for the special case of balanced strings, we improve upon the result of [19] by obtaining an $o(|\Sigma|)$ approximate solution in subquadratic time. There are a few fine-grained complexity results for approximate LCS, but they only hold against deterministic algorithms, and rely on very strong assumptions [12], [13], [14].

A. Our Results

For simplicity, we use $\text{lcs}(A, B)$ to denote the size (not the whole sequence) of the longest common subsequence for two strings $A$ and $B$. Similarly, we use $\text{ed}(A, B)$ to denote the edit distance and $\text{lis}(A)$ for the size of the longest common subsequence. We sometimes normalize the solution by the length of the strings so that the size of the solution remains in the interval $[0, 1]$. We refer to the normalized solutions by $||\text{lcs}(A, B)|| = \text{lcs}(A, B)/n$ and $||\text{ed}(A, B)|| = \text{ed}(A, B)/2n$ (here both strings have equal length $n$), and $||\text{lis}(A)|| = \text{lis}(A)/n$. In this way, $||\text{ed}(A, B)|| + ||\text{lcs}(A, B)|| = 1$ (assuming both strings have equal length).

As mentioned earlier, recent developments for edit distance are based on a simple but rather useful observation. Edit distance satisfies triangle inequality, or in other words, given three strings $A_1, A_2, A_3$ of length $n$ such that $||\text{ed}(A_1, A_2)|| \leq \delta$ and $||\text{ed}(A_2, A_3)|| \leq \delta$ hold, we can easily imply that $||\text{ed}(A_1, A_3)|| \leq 2\delta$. While lcs does not satisfy the triangle inequality in any meaningful way, it does, on average, satisfy the following birthday-paradox-like property that we call birthday triangle inequality.

Property (birthday triangle inequality). Consider three equal-length strings $A_1$, $A_2$, and $A_3$ such that $||\text{lcs}(A_1, A_2)|| \geq \lambda$ and $||\text{lcs}(A_2, A_3)|| \geq \lambda$. If the common subsequences correspond to random indices of each string, we expect that $||\text{lcs}(A_1, A_3)|| \geq \lambda^2$.

Of course, this is not necessarily the case in general. More precisely, it is easy to construct examples$^4$ in which $||\text{lcs}(A_1, A_2)|| = 1/2$ and $||\text{lcs}(A_2, A_3)|| = 1/2$, but $||\text{lcs}(A_1, A_3)|| = 0$. Our first main result shows that while it only holds on average, we can algorithmically replace the triangle inequality for edit distance with the birthday triangle inequality on worst case inputs.

Theorem (Main Theorem, formally stated as Theorem II.1). Given strings $A, B$ both of length $n$ such that $||\text{lcs}(A, B)|| = \lambda$, we can approximate the length of the LCS between the two strings within an $O(\lambda^3)$ factor in subquadratic time. The approximation factor improves to $(1 - \epsilon)\lambda^2$ when $1/\lambda$ is constant.

We remark that our algorithm is actually able to output the whole sequence of the solution, but we only focus on estimating the size of the solution for simplicity. We begin by comparing our main theorem to previous work on edit distance. In this case, $1/\lambda$ is constant w.l.o.g.$^5$ and therefore the approximation factor of our algorithm is $(1 - \epsilon)\lambda^2$. If $\delta = ||\text{ed}(A, B)||$, then our LCS algorithm outputs a transformation from $A$ to $B$ using at most $2n(1 - (1 - \epsilon)(1 - \delta)^3)$ operations. Observe that when the strings are not overly close and $\delta = \Omega(1)$ by scaling $\epsilon$, we already recover a $(3 + \epsilon')$-approximation for edit distance in truly subquadratic time. For mildly far strings, say $\delta = 0.1$, a more careful look at the expansion of $(1 - \delta)^3$ reveals that we save an additive $\Theta(\delta^2)$ in the approximation factor. For example, with $\delta = 0.1$ our approximation factor for edit distance is 2.71 instead of 3.

An interesting implication of our main result is for LCS over a large alphabet $\Sigma$, where the optimum $||\text{lcs}(A, B)||$ may be much smaller than 1. This is believed to be the hardest regime for approximation algorithms (and indeed the only one for which we have any conditional hardness of approximation results [12], [13], [14]). Here, we consider instances that satisfy a mild balance assumption: we assume that there is a character that appears with frequency at least $1/|\Sigma|$ in both strings$^6$. Then, our main theorem implies an $O(1/|\Sigma|^{3/4})$-approximate solution in truly subquadratic time (the first improvement over the trivial $1/|\Sigma|$ approximation in this regime).

Corollary (LCS, formally stated as Corollary II.3). Given a pair of strings $(A, B)$ of length $n$ over alphabet $\Sigma$ that satisfy the balance condition, we can approximate their LCS within an $O(|\Sigma|^{3/4})$ factor in truly subquadratic time.

Next, we show that a similar result can be obtained for LIS. Perhaps coincidentally, the approximation factor of our algorithm is also $O(\lambda^3)$ which is same to LCS.

$^4$For example, $A_1 = 0^n/20^n/2$, $A_2 = 0^n/21^n/2$, $A_3 = 1^n/21^n/2$.

$^5$When we use our solution to approximate edit distance, we can safely assume that $||\text{lcs}(A, B)|| = \Omega(1)$ since otherwise the edit distance of the two strings is very close to $2n$.

$^6$Note that in every instance in each string there is a character that appears with frequency at least $1/|\Sigma|$, but in general that may not be the same character.
but the technique is completely different. Although LIS can be solved exactly in time $O(n \log n)$, there have been several attempts to approximate the size of LIS and related problems in sublinear time [20], [21], [22], [23], [24], [25]. The best known solution is due to the work of Saks and Seshadhri [8] that obtains a $(1 + \epsilon)$-approximate algorithm for LIS in polylogarithmic time, when the solution size is at least a constant fraction of the input size.

In other words, if $\text{lis}(A) = \lambda$ and $1/\lambda$ is constant, their algorithm approximates $\text{lis}(A)$ in polylogarithmic time. However, this only works if $1/\lambda$ is constant and even if $1/\lambda$ is logarithmically large, their method fails to run in sublinear time.

We complement the work of Saks and Seshadhri [8] by presenting a result for LIS similar to our result for LCS. More precisely, we show that when $\text{lis}(A) = \lambda$, an $O(\lambda^3)$ approximation of LIS can be obtained in truly sublinear time. Although our approximation factor is worse than that of [8], our result works for any (not necessarily constant) $\lambda$.

**Theorem (LIS, formally stated as Theorem IV.8).** Given an array $A$ of $n$ integer numbers such that $\text{lis}(A) = \lambda$. We can approximate the length of the LIS for $A$ in sublinear time within a factor $O(\lambda^3)$.

If one favors the running time over the approximation factor, it is possible to improve the exponent of $n$ in the running time down to any constant $\kappa > 0$ at the expense of incurring a larger multiplicative factor to the approximation.

**B. Preliminaries**

In LCS or edit distance, we are given two strings $A$ and $B$ as input. We assume for simplicity that the two strings have equal length and refer to that by $n$. In LCS, the goal is to find the largest subsequence of the characters which is shared between the two strings. In edit distance, the goal is to remove as few characters as possible from the two strings such that the remainders for the two strings are the same. We use $\text{lcs}(A, B)$ and $\text{ed}(A, B)$ to denote the size of the longest common subsequence and the edit distance of two strings $A$ and $B$.

In LIS, the input contains an array $A$ of $n$ integer numbers and the goal is to find a sequence of elements of $A$ whose values (strictly) increase as their indices increase. For LIS, we denote the solution size for an array $A$ by $\text{lis}(A)$. We also use $\text{lis}^{[\alpha, \beta]}(A)$ to denote the size of the longest increasing subsequence subject to elements whose values lie in range $[\alpha, \beta]$. Longest increasing subsequence is equivalent to LCS when the inputs are two permutations of $n$ distinct characters.

Finally, we define a notation to denote the normalized solution sizes. For LCS, we denote the normalized solution size by $||\text{lcs}(A, B)|| = \text{lcs}(A, B)/\sqrt{\lambda ||A||B}$ for $A$ and $B$ and we use $||\text{ed}(A, B)|| = \text{ed}(A, B) / (2\sqrt{||A||B})$ for edit distance. Note that, when the two strings have equal length we have $||\text{ed}(A, B)|| + ||\text{lcs}(A, B)|| = 1$. Similarly, for longest increasing subsequence, we denote by $||\text{lis}(A)|| = \text{lis}(A)/||A||$ the normalized solution size. We usually refer to the size of the input array by $n$.

Throughout this paper, we call an algorithm $f(\lambda)$-approximation for LCS if it is able to distinguish the following two cases: i) $||\text{lcs}(A, B)|| \geq \lambda$ or ii) $||\text{lcs}(A, B)|| < \lambda f(\lambda)$. A similar definition carries over to LIS. Once an $f(\lambda)$-approximation algorithm is provided for either LCS or LIS, one can turn it into an algorithm that outputs a solution with size $f(\lambda)(1 - \epsilon)n$ provided that the optimal solution has a size $\lambda n$. The algorithm is not aware of the value of $\lambda$ but will start with $\lambda_0 = 1$ and iteratively multiply $\lambda_0$ by $1 - \epsilon$ until a solution is found.

**C. Techniques Overview**

Our algorithm for LCS is closely related to the recent developments for edit distance [1], [2]. We begin by briefly explaining the previous ideas for approximating edit distance and then show how we use these techniques to obtain a solution for LCS. Finally, in Section 1-C2 we outline our algorithm for LIS.

1) **Summary of Previous ED Techniques:** Indeed, edit distance is hard only if the two strings are far ($||\text{ed}(A, B)|| = \delta$ and $\delta = n^{-\alpha(3)}$); otherwise the $O(n + (n\delta)^2)$ algorithm of Landau et al. [3] computes the solution in truly subquadratic time. The algorithm of Chakraborty et al. [2] for edit distance has three main steps that we briefly discuss in the following.

**Step 0 (window-compatible solutions):** In the first step, they construct a set of windows $W_A$ for string $A$ and a set of windows $W_B$ for string $B$. Each window is essentially a substring of the given string. Let $k$ denote the total number of windows of $W_A \cup W_B$. For simplicity, let all the windows have the same size $d$ and $n \simeq O(kd)^9$. The construction features two key properties: 1) provided that the edit distances of the windows between $W_A$ and $W_B$ are available, one can recover a $1 + \epsilon$ approximation of edit distance in time $O(n + k^2)$ via dynamic programming. 2) $k^2 \times d^2 \simeq O(n^2)$. That is, if we naively compute the edit distance of every pair of windows, the overall running time would still asymptotically be the same as that of the classic algorithm.

In order to obtain a solution for edit distance, it suffices to know the distances between the windows. However, Chakraborty et al. [2] show that knowing the distances between most of the window pairs is enough to obtain an approximately optimal solution for edit distance. More precisely, if the distances are not correctly approximated for

---

7Their algorithm obtains an additive error of $\delta n$ in time $2^{\tilde{O}(1/\delta)}$. When the solution size is bounded by $\lambda n$, one needs to set $\delta < \lambda$ in order to guarantee a multiplicative factor approximation.

8There is a term $(1/\lambda)^{1/\lambda}$ in the running time.

9The equality holds if we assume $\delta = \Omega(1)$. 

1123
at most $O(k^2 - \Omega(1))$ pairs, we can still obtain an approximate solution for edit distance. Step 1 provides estimates for the distances of the windows which is approximately correct except for $O(k^2 - \Omega(1))$ many pairs and Step 2 shows how this can be used to obtain a solution for edit distance. Discretization simplifies the problem substantially. For a fixed $0 \leq \delta \leq 1$, they introduce a graph $G_\delta$ where the nodes correspond to the windows and an edge between window $w_i \in W_A$ and window $w_j \in W_B$ means that $||ed(w_i, w_j)|| \leq \delta$. If we are able to construct $G_\delta$ for logarithmically different choices of $\delta$, we can as well estimate the distances within a $1 + \epsilon$ factor for the windows. Therefore the problem boils down to constructing $G_\delta$ for a fixed given $\delta$ without computing the edit distance between all pairs of windows.

**Step 1 (sparsification via triangle inequality):** This step is the heart of the algorithm. Suppose we choose a high-degree vertex $v$ from $G_\delta$ and discover all its incident edges by computing its edit distance to the rest of the windows. Triangle inequality implies that every pair of windows in $N(v)$ has a distance bounded by $2\delta$. Therefore by losing a factor 2 in the approximation, one can put all these edges in $G_\delta$ and compute the edit distances explicitly. Although this does save some running time, in order to make sure the running time is truly subquadratic, we need to make a similar argument for paths of length 3 and thereby lose a factor 3 in the approximation. This method sparsifies the graph and what remains is to discover the edges of a sparse graph.

**Step 2 (discovering the edges of the sparse graph):** Step 1 uses triangle inequality and discovers many edges between the vertices of $G_\delta$. However, it may not discover all the edges completely. When in the remainder graph, the degrees are small (and hence the graph is sparse) triangle inequality does not offer an improvement and thus a different approach is required. Roughly speaking, Chakraborty et al. [2] subsample the windows of $W_A$ into a smaller set $S$ and discover all pairs of windows $w_i \in S$ and $w_j \in W_B$ such that edge $(i, j)$ is not discovered in Step 1. Next, they compute the edit distance of each pair of windows $(w_i, w_j), w_i \in W_A, w_j \in W_B$ such that there exist two nearby windows $(w_a, w_b)$ satisfying $w_a \in S, w_b \in W_B$ and the edge between $w_a$ and $w_b$ was missed in Step 1. The key observation is that even though this procedure does not discover all the edges, the approximated distances lead to an approximate solution for edit distance.

2) **LCS:** Our algorithm for LCS mimics the same guideline. In addition to this, Steps 0 and 2 of our algorithm are LCS analogues of the ones used by Chakraborty et al. [2]. The main novelty of our algorithm is Step 1 which is a replacement for triangle inequality. Recall that unlike edit distance, triangle inequality does not hold for LCS.

**Challenge 1.1.** How can we introduce a notion similar to triangle inequality to a non-metric setting such as LCS?

We introduce the notion of birthday triangle inequality to overcome the above difficulty. Given windows $w_1, w_2, \text{and} \ w_3$ of size $d$ such that $||\text{lcs}(w_1, w_2)|| \geq \lambda$ and $||\text{lcs}(w_2, w_3)|| \geq \lambda \text{ hold, what can we say about the LCS of } w_1 \text{ and } w_3? \text{ In general, nothing! } ||\text{lcs}(w_1, w_3)|| \text{ could be as small as } 0. \text{ However, let us add some randomness to the setting. Think of the LCS of } w_1 \text{ and } w_2 \text{ as a matching from the characters of } w_1 \text{ to } w_2 \text{ and similarly the LCS for } w_2 \text{ and } w_3 \text{ as another matching between characters of } w_2 \text{ and } w_3. \text{ Assume (for the sake of the thought experiment) that the characters of } w_2 \text{ appear random in each matching. Since } ||\text{lcs}(w_1, w_2)|| \geq \lambda, \text{ each character of } w_2 \text{ appears with probability at least } \lambda \text{ in the matching between } w_1 \text{ and } w_2. \text{ A similar argument implies that each character of } w_2 \text{ appears with probability } \lambda \text{ in the matching of } w_2 \text{ and } w_3. \text{ Thus, (assuming independence), each character of } w_2 \text{ appears in both matchings with probability } \lambda^2. \text{ This means that in expectation, there are } \lambda^2d \text{ paths of length } 2 \text{ between } w_1 \text{ and } w_3 \text{ which suggests } ||\text{lcs}(w_1, w_3)|| \geq \lambda^2 \text{ as shown in Figure 1. This is basically birthday paradox used for the sake of triangle inequality.}

Replacing triangle inequality by birthday triangle inequality is particularly challenging since birthday triangle inequality only holds on average. In contrast, triangle inequality holds for any tuple of arbitrary strings. Most of our technical discussions is dedicated to proving that we can algorithmically use birthday triangle inequality to obtain a solution for the worst-case scenarios. The most inconvenient step of our analysis is to show that our algorithm estimates the LCS of most window pairs in the sparsification phase. While this is straightforward for edit distance, birthday triangle inequality requires a deeper analysis of the underlying graph. In particular, we need to prove that if the undiscovered edges are too many, then birthday triangle inequality can be applied to certain neighborhoods of the graph.

There are two difficulties that we face here. On one hand, in order to apply birthday triangle inequality to a subgraph, we need to have enough structure for that subgraph to show the implication can be made. On the other hand, our assumptions cannot be too strong, otherwise such neighborhoods may not cover the edges of the graph. Therefore, the first challenge that we need to overcome is characterizing subgraphs in which birthday triangle inequality is guaranteed to be applicable. Our suggestion is the bi-cliques structure. Although combinatorial techniques seem unlikely to prove this, we use the Blakley-Roy inequality to show that in a large enough bi-clique, we can use birthday triangle inequality to imply a bound on the LCS of certain pairs. The second challenge is to prove that if the underlying graph is dense enough, the graph contains many bi-cliques that cover almost all the edges that we plan to discover. This is again a challenging graph theoretic problem. We leverage extremal
As we discussed earlier, the hard instances of the problem are the cases where the lcs-graph is dense for which we need a sparsifier. Roughly speaking, in our sparsification technique, our algorithm constructs another graph $G\lambda$ such that $G\lambda$ is valid in the sense that the edges of $G\lambda$ correspond to pairs of windows with large enough LCS. In addition to this, our algorithm guarantees that after the removal of the edges of $G\lambda$ from $G$, the remainder is sparse. In other words, $|E(G\lambda) \setminus E(G_1)| = O(|V(G_\lambda)|^{2-\Omega(1)})$. Of course, if the overall running time of the sparsification phase is truly subquadratic, the error of undiscovered edges can be addressed by the techniques of [2] in Step 2.

Below, we bring a formal definition for sparsification.

**sparsification**

**input:** Windows $w_1, w_2, \ldots, w_k$, parameters $\lambda$, and $\alpha$.

**solution:** A matrix $\tilde{O}_\alpha \in \{0, 1\}^{k \times k}$ such that:
- $\tilde{O}_\alpha[i][j] = 1 \implies ||\text{lcs}(w_i, w_j)|| \geq \alpha \cdot \lambda$
- $\left\{ (i, j) \mid ||\text{lcs}(w_i, w_j)|| \geq \lambda \text{ and } \tilde{O}_\alpha[i][j] = 0 \right\} = k^{2-\Omega(1)}$

We present two sparsification techniques for LCS. The first one (Section III-A), has an approximation factor of $(1 - \epsilon) \cdot \lambda^2$. In Section III-B we present another sparsification technique that has a worse approximation factor $O(\lambda^3)$ but leaves fewer edges behind. Although the second sparsification technique has a worse approximation factor, it has the advantage that the number of edges that remain in the sparse graph is truly subquadratic regardless of the value of $\lambda$ and therefore it extends our solution to the case that $\lambda = o(1)$ (see Section III-B for a detailed discussion).

Let us note one last algorithmic challenge to keep in mind before we begin to describe our sparsification techniques. For edit distance, if window pairs $(w_1, w_2)$ and $(w_2, w_3)$ are close, we are guaranteed that $w_1$ and $w_3$ are also close; for longest common subsequence, we will argue that $(w_1, w_3)$ are likely to be have a long LCS (for a “random” choice of $(w_1, w_3)$). Nonetheless, in order to add $(w_1, w_3)$ as an edge to our graph we have to verify that their LCS is indeed long.
If we were to verify an edge naively, we would need as much time as computing the LCS between \((w_i, w_j)\) from scratch!

**Sparsification 1, \((1 - \epsilon)\lambda^2\)-approximation::** Similar to edit distance, applying birthday triangle inequality to paths of length 2 for LCS does not improve the running time significantly. Therefore, we need to use birthday triangle inequality for paths of length 3. To this end, we define the notion of constructive tuples as follows: a tuple \(<w_i, w_a, w_b, w_j>\) is an \((\epsilon, \lambda)\)-constructive tuple, if we have 
\[
|\text{LCS}(w_i, w_a)| \geq \lambda, |\text{LCS}(w_i, w_j)| \geq \lambda, |\text{LCS}(w_a, w_b)| \geq \lambda 
\]
and by taking the intersection of the three LCS matchings, we are able to imply \(|\text{LCS}(w_a, w_b)| \geq (1 - \epsilon)\lambda^3\) (see Figure 2 for an example). Taking the intersection of the matchings can be done in linear time which is faster than computing the LCS.

Our sparsification technique here is simple but the analysis is very intricate. We subsample a set \(S\) of windows and compute the LCS of every window in \(S\) and all other windows. We set \(|S| = k^\gamma \log k\), where \(\gamma \in (0, 1)\). At this point, for some pairs, we already know their LCS. However, if neither \(w_i\) nor \(w_j\) is in \(S\), we do not know if \(|\text{LCS}(w_i, w_j)| \geq \lambda\) or not. Therefore, for such pairs, we try to find windows \(w_a, w_b \in S\) such that \(<w_i, w_a, w_b, w_j>\) is constructive. If such a constructive tuple is found for a pair of windows, then we conclude that their normalized LCS is at least \((1 - \epsilon)\lambda^3\).

All that remains is to argue that this method discovers almost all the edges of the lcs-graph \(G_\lambda\) and the number of undiscovered edges is \(k^{2 - \Omega(1)}\). This is the most difficult part of the analysis. We note that proving the existence of only one constructive tuple is already non-trivial even when \(G_\lambda\) is complete. However, our goal is to show almost all the edges are discovered via constructive tuples when \(G_\lambda\) is dense (and of course not necessarily complete).

Define a pair of windows \(w_i\) and \(w_j\) to be well-connected, if there are at least \(k^{2 - \gamma}\) different \((w_a, w_b)\) pairs such that \(<w_i, w_a, w_b, w_j>\) is \((\epsilon, \lambda)\)-constructive. Since each window appears in \(S\) with probability \(k^{-\gamma} \log k\), for each well-connected pair we find one constructive tuple via our algorithm with high probability. Therefore, we need to prove that the total number of pairs \((w_i, w_j)\) such that \((w_i, w_j)\) is not well-connected but \(|\text{LCS}(w_i, w_j)| \geq \lambda\) is subquadratic. Let us put these edges in a new graph \(NG_\lambda\) whose vertices are all the windows. We first leverage the Blakley-Roy inequality and a double counting technique to prove that if \(NG_\lambda\) has a large complete bipartite subgraph, then there is one constructive tuple which includes only the vertices of this subgraph (Lemma III.5). Next, we apply the Turan’s theorem to show that if \(NG_\lambda\) is dense, then it has a lot of large complete bipartite subgraphs. Finally, we use a probabilistic method to conclude that \(NG_\lambda\) cannot be too dense otherwise there are a lot of constructive tuples in the graph which implies that at least one edge \((w_i, w_j)\) in \(NG_\lambda\) is well-connected. This is not possible since all the well-connected pairs are detected in our sparsification algorithm with high probability.

The above argument proves that if we sparsify our graph using our sparsification algorithm, the remainder graph would have a subquadratic number of edges. Therefore after plugging Step 2 into the algorithm, the running time remains subquadratic. However, since Turan theorem gives us a weak bound, the running time of the algorithm using this sparsification is \(O(n^{2 - \Omega(\lambda)})\) and is only truly subquadratic when \(1/\lambda\) is constant.

**Sparsification 2, \(O(\lambda^3)\)-approximation::** In Section III-A, we present another sparsification method that although gives us a slightly worse approximation factor \(O(\lambda^3)\) it always leaves a truly subquadratic number of edges behind and therefore the running time of the algorithm would be truly subquadratic regardless of the parameter \(\lambda\). This sparsification is based on a novel data structure.

Let \(opt_{i,a}\) denote the longest common subsequence of \(w_i\) and \(w_a\) (with some fixed tie-breaking rule, e.g. lexicographically first). Define \(lcs_{\lambda}(w_i, w_j)\) to be the size of the longest common subsequence between \(opt_{i,a}\) and \(w_j\). Notice that this definition is no longer symmetric. Let \(|\text{LCS}(w_i, w_j)|\) denote the relative value, i.e., \(|\text{LCS}_{w_a}(w_i, w_j)| = \text{LCS}_{w_a}(w_i, w_j)/\sqrt{|w_i| \cdot |w_j|}\). The first ingredient of the algorithm is a data-structure, namely lcs-cmp. After a preprocess of time \(O(|w_a| \sum_{i \in S} |w_i|)\), lcs-cmp is able to answer queries of the following type in time \(O(|w_i| + |w_j|)\): 

- “For a \(0 \leq \lambda \leq 1\) either certify that \(|\text{LCS}_{w_a}(w_i, w_j)| \geq \Omega(\lambda^2)\) or report that \(|\text{LCS}_{w_a}(w_i, w_j)| < O(\lambda)\)."

In our sparsification, we repeat the following procedure \(k^\gamma\) times, where \(\gamma \in (0, 1)\). We sample a window \(w_a\) uniformly at random and construct lcs-cmp\((w_a, S)\) for \(S = \{w_i | i \neq a \text{ and } |w_i| \geq |w_a|\}\). After the preprocessing step, we make a query for every pair of windows \((w_i, w_j)\) such that \(w_i, w_j \in S\) and determine if \(lcs_{\lambda}(w_i, w_j)\) is at least \(\Omega(\lambda^4)\) or upper bounded by \(O(\lambda^2)\) (here \(\lambda = \lambda^2\)). If their LCS is at least \(\Omega(\lambda^4)\) we report this pair as an edge in our lcs-graph. Finally, we use the Turan theorem to prove that the number of remaining edges in our graph is small.

To be more precise, we first construct a graph \(NG_\lambda\) that reflects the edges that are not detected via our sparsification. We give directions to the edges based on the length of the windows. If \(NG_\lambda\) is dense enough, then there is one vertex \(v\) in \(NG_\lambda\) with a large enough outgoing degree. We use the neighbors of \(v\) to construct another graph \(NF_\lambda\) with vertex set \(N(v)\). An edge exists in \(NF_\lambda\) if 
\[\max\{|\text{LCS}_{w_a}(w_i, w_j)|, |\text{LCS}_{w_a}(w_j, w_i)|\} \geq \Omega(\lambda^2)\]. Edges of \(NF_\lambda\) are directed the same way as \(NG_\lambda\). We prove that \(NF_\lambda\) has no large independent set. In other words, if we select a large enough set of vertices in \(NF_\lambda\), then there is at least one edge between them. Next, we apply the Turan theorem to prove that \(NF_\lambda\) is dense. Finally, we imply that since
\(NF_\lambda\) is dense, there is one vertex \(u\) in the neighbors of \(v\) such that there are a lot of 2-paths between \(v\) and \(u\). This implies that the edge \((u, v)\) should have been detected in our sparsification and therefore must not exist in \(NG_3\). This contradiction implies that \(NF_\lambda\) is sparse in the first place.

3) LIS: In this section, we present our result for longest increasing subsequence. More precisely, we show that when the solution size is lower bounded by \(n\lambda\ (\lambda \in [0, 1])\), one can approximate the solution within a factor \(O(\lambda^3)\) in time \(O(\sqrt{n}/\lambda^7)\). This married with a simple sampling algorithm for the cases that \(\lambda < n^{-O(1)}\), provides an \(O(\lambda^3)\)-approximate algorithm with running time of \(O(n^{0.85})\) (without further dependence on \(\lambda\)). We further extend this result to reduce the running time to \(O(n^{\kappa}\text{poly}(1/\lambda))\) for any \(\kappa > 0\) by imposing a multiplicative factor of \(\text{poly}(1/\lambda)^{10}\) to the approximation.

Our algorithm heavily relies on sampling random elements of the array for which longest increasing subsequence is desired. Denote the input sequence by \(B = \langle a_1, a_2, \ldots, a_n\rangle\). A naive approach to approximate the solution is to randomly subsample the elements of \(B\) to obtain a smaller array \(B'\) and then compute the longest increasing subsequence of \(B'\) to estimate the solution size for \(B\). Let us first show why this approach alone fails to provide a decent approximation factor. First, consider an array \(B = \langle 1, 2, \ldots, n\rangle\) which is strictly increasing. Based on \(B\), we construct two inputs \(B'\) and \(B''\) in the following way:

- \(B'\) is exactly equal to \(B\) except that a \(p\) fraction of the elements in \(B'\) are replaced by \(0\).
- \(B''\) is exactly equal to \(B\) except that every block of length \(\sqrt{n}\) is reversed in \(B''\). In other words, \(B'' = \langle \sqrt{n}, \sqrt{n} - 1, \sqrt{n} - 2, \ldots, 1, 2\sqrt{n}, 2\sqrt{n} - 1, \ldots, \sqrt{n} + 1, \ldots, n, n - 1, n - 2, \ldots, n - \sqrt{n} + 1\rangle\).

We subsample the two arrays \(B'\) and \(B''\) with a rate of \(1/\sqrt{n}\) to obtain two smaller arrays \(B'\) and \(B''\) of size roughly \(O(\sqrt{n})\). It is easy to prove that \(\text{lis}(B') = \Omega(\sqrt{n})\) and \(\text{lis}(B'') = \Omega(\sqrt{n})\) both hold even though \(\text{lis}(B') = \Omega(n)\) but \(\text{lis}(B'') = O(\sqrt{n})\). By setting \(p = 1/e^{11}\) we can also make sure that \(\text{lis}(B')\) and \(\text{lis}(B'')\) are within a small multiplicative range even though the gap between \(\text{lis}(B')\) and \(\text{lis}(B'')\) is substantial.

The above observation shows that the problem is very elusive when random sampling is involved. We bring a remedy to this issue in the following. Divide the input array into \(\sqrt{n}\) subarrays of size \(\sqrt{n}\). We denote the subarrays by \(sa_1, sa_2, \ldots, sa_{\sqrt{n}}\) and fix an optimal solution \(A\) for the longest increasing subsequence of \(A\). Define \(\text{sm}(sa_i)\) to be the smallest number in \(sa_i\) that contributes to \(A\) and \(\text{lg}(sa_i)\) to be the largest number in \(sa_i\) that contributes to \(A\). Moreover, define \(\text{lis}^{[\ell, r]}(A)\) to be the longest increasing subsequence of an array subject to the elements whose values lie within the interval \([\ell, r]\). This immediately implies

\[
\text{lis}(A) = \sum_{i=1}^{\sqrt{n}} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i).
\]

Another observation that we make here is that since we assume \(\|\text{lis}(A)\| \geq \lambda\) and the size of each subarray is bounded by \(\sqrt{n}\), then we have

\[
\frac{\text{lis}(A)}{\max_i \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i)} \geq \sqrt{n}\lambda
\]

which essentially means that in order to approximate \(\text{lis}(A)\) it suffices to compute \(\text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i)\) for \(O(1/\lambda)\) many randomly sampled subarrays. This is quite helpful since this shows that we only need to sample \(O(1/\lambda)\) many subarrays and solve the problem for them. However, we do not know the values of \(\text{sm}(sa_i)\) and \(\text{lg}(sa_i)\) in advance. Therefore, the

\(^{10}\)The exponent of \(1/\lambda\) depends exponentially on \(1/\kappa\).

\(^{11}\)\(e \approx 2.7182\).
main challenge is to predict the values of $\text{sm}(s_a_i)$ and $\text{lg}(s_a_i)$ before we sample the subarrays.

Indeed, one needs to read the entire array to correctly compute $\text{sm}(s_a_i)$ and $\text{lg}(s_a_i)$ for each of the subarrays. However, we devise a method to approximately guess these values without losing too much in the size of the solution. Roughly speaking, we show that if we sample $k = O(1/(\lambda \epsilon))$ different elements from a subarray $s_a_i$ for some constant $\epsilon$ and denote them by $a_{j_1}, a_{j_2}, \ldots, a_{j_k}$, then for at least one pair $(\alpha, \beta)$, $[a_{j_\alpha}, a_{j_\beta}]$ is approximately close to $[\text{sm}(s_a_i), \text{lg}(s_a_i)]$ up to a $(1 - \epsilon)$ factor.

The above argument provides $O((1/(\lambda \epsilon))^2)$ candidate domain intervals for each $s_a_i$, However, this does not provide a solution since we do not know which candidate domain interval approximates $[\text{sm}(s_a_i), \text{lg}(s_a_i)]$ for each $s_a_i$. Of course, if we were to randomly choose one candidate interval for every subarray, we would make a correct guess for at least $O(\sqrt{n}/(\lambda \epsilon))^2$ subarrays which provides an approximation guarantee of $O(\lambda^2)$ for our algorithm. However, our assignments have to be monotone too. More precisely, let $[\tilde{\text{sm}}(s_a), \tilde{\text{lg}}(s_a)]$ be the guesses that our algorithm makes, then we should have

$$\tilde{\text{sm}}(s_a) \leq \text{sm}(s_a) \leq \tilde{\text{sm}}(s_a^2) \leq \text{sm}(s_a^2) \leq \ldots \leq \tilde{\text{sm}}(s_{\sqrt{n}}(s_a)) \leq \text{lg}(s_{\sqrt{n}}(s_a)).$$

Random sampling does not guarantee that the sampled intervals are monotone. To address this issue, we introduce the notion of pseudo-solutions. A pseudo-solution is an assignment of monotone intervals to subarrays in order to approximate $\text{sm}(s_a_i)$ and $\text{lg}(s_a_i)$. The quality of a pseudo-solution with intervals $[\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_{\sqrt{n}}, r_{\sqrt{n}}]$ is equal to $\sum_{i} \text{lis}^*(\ell_i, r_i, s_a_i)$. For a fixed pseudo-solution, this can be easily approximated via random sampling. Thus, our goal is to construct a pseudo-solution whose quality is at least an $O(\lambda^3)$ approximation of the size of the optimal solution. To this end, we present a greedy method in Section IV-B to construct the desired pseudo-solution.

Finally, in Section IV-D, we show how the above ideas can be generalized to improve the running time down to $O(n^\kappa \text{poly}(1/\lambda))$ for any arbitrarily small $\kappa > 0$ by imposing a factor $\text{poly}(1/\lambda)$ to the approximation guarantee.

II. Organization of the Paper

Our algorithm for LCS contains 3 steps (Steps 0-2) that are explained in details in the full-version. Since Steps 0 and 2 follow from previous work, we only bring Step 1 in this version and defer the rest to the full-version.

In both our results for LCS and LIS, we assume that the goal is to find approximate solutions, provided that the solution size is at least $\lambda_0 n$. After the algorithms terminate, if the output is smaller than what we expect, we realize that the solution is smaller than $\lambda_0 n$. Therefore, we begin by setting $\lambda_0 = 1$ and iteratively multiply $\lambda_0$ by a $1-\epsilon$ factor until we obtain a solution. This only adds a multiplicative factor of $\log 1/\lambda$ to the running time and a multiplicative factor of $1-\epsilon$ to the approximation. Since we present two different sparsification techniques, we obtain two theorems: one is Theorem II.1 and the other is Theorem II.2.

Theorem II.1. Given strings $A, B$ of length $|A| = |B| = n$ with $||\text{lcs}(A,B)|| = \lambda$, we can approximate the length of the LCS between the two strings within a factor $O(\lambda^2)$ in time $O(n^{39/20}).$

Proof: If $\lambda \leq n^{-20}$ we run the classic $O(n^2 \lambda)$ time algorithm and get an exact solution in the desired time. Otherwise, we begin by setting $\lambda_0 = 1$ and iteratively multiply $\lambda_0$ by a factor $1-\epsilon$ until a solution is found with size at least $\Omega(\lambda^3) n$. This adds an overhead of $O(\log 1/\lambda)$ to the running time. By Choosing $d = \sqrt{n}$ we can bound the total running time of Steps 0, 1, and 2 by $O(n^{13/10} \lambda^{-13}) \leq O(n^{39/20}).$

Theorem II.2. Given strings $A, B$ of length $|A| = |B| = n$ with $||\text{lcs}(A,B)|| = \lambda$, we can approximate the length of the LCS between the two strings within a factor $(1 - \epsilon)\lambda^2$ in $n^{2 - \Omega(\epsilon \lambda)}$ time for any $\epsilon > 0$.

Proof: The proof is identical to Theorem II.1, we omit the details here.

As an immediate corollary of Theorem II.1, we present an algorithm that achieves the $1/|\Sigma|$ approximation factor in truly subquadratic time, when the strings are balanced.

Corollary II.3. Given a pair of strings $(A, B)$ of length $n$ over alphabet $\Sigma$ that satisfy the balance condition, we can approximate their LCS within an $O(|\Sigma|^{3/4})$ factor in time $O(n^{39/20}).$

Proof: Since $A$ and $B$ are balanced, there is a character $\sigma \in \Sigma$ that appears at least $n/|\Sigma|$ times in both strings. Indeed, finding a solution of size $n/|\Sigma|$ by restricting our attention to only character $\sigma$ can be done in time $O(n)$. If $\text{lcs}(A,B) \leq n/|\Sigma|^{1/4}$ this already gives us an $O(|\Sigma|^{3/4})$ approximate solution. Otherwise, $||\text{lcs}(A,B)|| > 1/|\Sigma|^{1/4}$ and the approximation factor of our $O(\lambda^3)$-approximation algorithm would be bounded by $O(|\Sigma|^{3/4}).$

Finally, we bring our results for LIS in Section IV. We show that

Theorem II.4. Given a length-$n$ sequence $A$ with $\text{lis}(A) = n\lambda$. We can approximate the length of the LIS within a factor of $O(\lambda^3)$ in time $O(n^{17/20})$.

Proof: If $\lambda < n^{-1/20}$ we sample the array with a rate of $n^{-3/20}$ and compute the LIS for the sampled array. The running time of the algorithm is $O(n^{17/20})$. The approximation factor is $O(n^{-3/20}) \geq O(\lambda^3)$. Otherwise, by Theorem IV.8, we estimate the size of LIS up to an $O(\lambda^3)$
approximation factor in time $\tilde{O}(\lambda^{-7}\sqrt{n}) \leq \tilde{O}(n^{17/20})$.  

III. LCS STEP 1: SPARSIFICATION VIA BIRTHDAY TRIANGLE INEQUALITY

Recall that we are given two sets of windows $W_A$ and $W_B$ for the strings and our goal is to approximate the LCS of every pair of windows between $W_A$ and $W_B$. For simplicity, we put all the windows in the same basket $W = W_A \cup W_B$ and denote the windows by $w_1, w_2, \ldots, w_k$ where $k$ is the total number of windows. Since the windows have different lengths, we define $w_{\text{max}} = \max_{i \in [k]} |w_i|$ to be the maximum length of the windows. Similarly, we also define $w_{\text{min}} = \min_{i \in [k]} |w_i|$ to be the minimum length of the windows. Let $w_{\text{gap}} = w_{\text{max}}/w_{\text{min}}$. Let $w_{\text{layers}}$ denote the number of different window sizes. Notations $w_{\text{gap}}$ and $w_{\text{layers}}$ will be used in the later analysis.

In order to approximate the LCS’s we fix a $\lambda \in \{\epsilon\lambda_0, (1 + \epsilon)\lambda_0, (1 + \epsilon)^2\lambda_0, \ldots, 1\}$ and sparsify graph $G_{\lambda}$. In Section III-A, we present a sparsification algorithm (Algorithm 1) which provides $\lambda^2$-approximation when $\lambda$ is constant. The formal guarantee of the algorithm is provided in Theorem III.11. In Section III-B, we present a sparsification which provides $O(\lambda^3)$-approximation for any (potentially super-constant) $\lambda$.

A. Sparsification for $O_{\lambda^2}$

In our solution, we fix an arbitrary LCS for every pair of windows and refer to that as $\text{opt}_{i,j}$ for two windows $w_i$ and $w_j$. Note that we do not explicitly compute $\text{opt}_{i,j}$ in our algorithm. Let us for simplicity, think of each $\text{opt}_{i,j}$ as a matching between the characters of the two windows. Also, denote by $(\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j})$ a solution which is constructed for windows $w_i$ and $w_j$ by taking the intersection of $(\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j})$. More precisely, we have $(x, y) \in (\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j}) \iff \exists \alpha, \beta$ such that $(x, \alpha) \in \text{opt}_{i,a}$ and $(\alpha, \beta) \in \text{opt}_{a,b}$ and $(\beta, y) \in \text{opt}_{b,j}$.

Let $\| (\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j}) \| = \frac{|(\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j})|}{\sqrt{|w_i| |w_j|}}$.

**Definition III.1** ($(\epsilon, \lambda)$-constructive). We call a tuple $\langle w_i, w_a, w_b, w_j \rangle$ (where $w_i \neq w_a \neq w_b \neq w_j$) a constructive tuple, if $\| (\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j}) \| \geq (1 - \epsilon)\lambda^3$.

The advantage of a constructive tuple is that if $\text{opt}_{i,a} \cap \text{opt}_{a,b}$ and $\text{opt}_{b,j}$ are provided, one can construct a desirable solution for $\text{opt}_{i,j}$ in linear time by taking the intersection of the given matchings. Our algorithm is actually based on the above observation. We bring our algorithm in the following:

We parameterize our algorithm by a value $0 < \gamma < 1$ to be set later. One may optimize the runtime of the algorithm by setting the value of $\gamma$ in terms of the number of windows and the length of the windows. We first sample a set $S$ of $O(k^\gamma \log k)$ windows. Next, we compute $\text{opt}_{i,j}$ of every window $w_i \in S$ and every other window $w_j$ (not necessarily in $S$). Finally, we find all the constructive tuples $\langle w_i, w_a, w_b, w_j \rangle$ such that $w_a, w_b \in S$ and update $\hat{O}_{\lambda^2}$ accordingly. This is shown in Algorithm 1.

The running time of our algorithm is equal to $O(k |S|w_{\text{max}}^2 + k^2 |S|^2w_{\text{max}}^2)$. The rest of this section is dedicated to proving that what remains in the LCS-graph is sparse.

We first introduce the notion of well-connected pairs.

**Definition III.2** (well-connected pair). We say a pair of windows $(w_i, w_j)$ is well-connected, if there are at least $k^{2-\gamma}$ pairs of windows $(w_a, w_b)$ such that $\langle w_i, w_a, w_b, w_j \rangle$ is $(\epsilon, \lambda)$-constructive.
Algorithm 1: Sparsification for $O_{\lambda^2}$

1: procedure QUADRATICSPARSIFICATION($w_1, w_2, \ldots, w_k, \lambda, \epsilon$)\hfill $\triangleright$ Theorem III.11
2: $\gamma \leftarrow 0.1$
3: $S \leftarrow 40k^2 \log k$ i.i.d. samples of $[k]$
4: $O_{\lambda^2} \leftarrow \{0\}^{k \times k}$
5: for $w_i \in S$ do \hfill $\triangleright$ Takes $|S|kw_{\text{max}}^2$ time
6: for $j \leftarrow 1$ to $k$ do
7: $\text{opt}_{i,j}, \text{opt}_{j,i} \leftarrow \text{lcs}(w_i, w_j)$
8: end for
9: end for
10: for $w_i \in S$ do \hfill $\triangleright$ Takes $|S|kw_{\text{max}}^2$ time
11: for $j \leftarrow 1$ to $k$ do
12: if $|\text{opt}_{i,j}| > \lambda$ then
13: $O_{\lambda^2}[i][j] \leftarrow 1$
14: end if
15: end for
16: end for
17: for $i \leftarrow 1$ to $k$ do \hfill $\triangleright$ Takes $k^2|S|^2w_{\text{max}}$ time
18: for $j \leftarrow 1$ to $k$ do
19: for $w_a \in S$ do
20: for $w_b \in S$ do
21: if $||\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j}|| \geq (1 - \epsilon)^3 \lambda$ then
22: $O_{\lambda^2}[i][j] \leftarrow 1$
23: end if
24: end for
25: end for
26: end for
27: end for
28: return $O_{\lambda^2}$
29: end procedure

It follows from the definition that well-connected pairs are detected in our algorithm with high probability.

Lemma III.3. Let $O_{\lambda^2} \in \{0, 1\}^{k \times k}$ denote the output of Algorithm 1. With probability at least $1 - 1/\text{poly}(k)$, for all $(i, j) \in [k] \times [k]$ such that $(w_i, w_j)$ is a well-connected pair (Definition III.2), we have $O_{\lambda^2}[i][j] = 1$.

Proof: We consider a fixed $(i, j)$ such that pair $(w_i, w_j)$ is well-connected. Let

$$Q_{i,j} = \left\{ (a, b) \mid ||\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j}|| \geq (1 - \epsilon)^3 \lambda \right\}.$$

Conceptually, we divide the process of sampling $S$ into two phases: we sample $20k^2 \log k$ windows in the first phase, and then we sample another $20k^2 \log k$ windows in the second phase.

For each $a \in [k]$, let $Q_{i,j,a} = \{b : (a, b) \in Q_{i,j}\}$. Since

$$\sum_{a \in [k]} |Q_{i,j,a}| = |Q_{i,j}| \geq k^{2-\gamma},$$

there are at least

$$\frac{k^{2-\gamma} - k \cdot k^{1-\gamma}/2}{k} = \frac{k^{1-\gamma}}{2}$$

different number $a$’s in $[k]$ such that $|Q_{i,j,a}| \geq k^{1-\gamma}/2$. Hence, in the first phase, there is a sampled number $q$ such that $|Q_{i,j,q}| \geq k^{1-\gamma}/2$ with probability at least

$$1 - \left(1 - \frac{k^{1-\gamma}/2}{k^{1-\gamma} \cdot 20k^2 \log k} \right) < 1/k^5.$$

We fix such a $q$. In the second phase, there is a sampled number $r$ such that $r \in Q_{i,j,q}$ with probability at least

$$1 - \left(1 - \frac{k^{1-\gamma}/2}{k^{1-\gamma} \cdot 20k^2 \log k} \right) < 1/k^5.$$

The lemma is obtained by a union bound on all the well-connected pairs $(w_i, w_j)$.

In what follows, we bound $||\text{lcs}(w_i, w_j)|| \geq \lambda$ and $O_{\lambda^2}[i][j] = 0$). In other words, we prove an upper bound on the number of edges that remain in the graph.

Definition III.4 (NG$_\lambda$). Define a graph $NG_\lambda$ with $k$ vertices and the following edges:

$$E(NG_\lambda) = \left\{ (i, j) \mid ||\text{lcs}(w_i, w_j)|| \geq \lambda \text{ and } O_{\lambda^2}[i][j] = 0 \right\}.$$

We first prove that every $K_{\Omega(\omega_{\text{gap}}/(\epsilon \lambda^3)), \Omega(\omega_{\text{gap}}/(\epsilon \lambda^3))}$ subgraph of NG$_\lambda$ corresponds to at least one constructive tuple.

Lemma III.5. Let $X$ and $Y$ be two sets of windows such that for every $w_i \in X$ and $w_j \in Y$ there is an $(i, j)$ edge in $E(NG_\lambda)$, for every $w_i, w_i' \in X$, $|w_i| = |w_i'|$ and for every $w_j, w_j' \in Y$, $|w_j| = |w_j'|$. If $|X| \geq \Omega(\omega_{\text{gap}}/(\epsilon \lambda^3))$ and $|Y| \geq \Omega(\omega_{\text{gap}}/(\epsilon \lambda^3))$, then there exist $w_i, w_a, w_b, w_j \in X \cup Y$ such that $(w_i, w_a, w_b, w_j)$ is (1, $\lambda$)-constructive, i.e.,

$$||\text{opt}_{i,a} \cap \text{opt}_{a,b} \cap \text{opt}_{b,j}|| \geq (1 - \epsilon)\lambda^3.$$

Proof: By assumption, we know that $|X|, |Y| \geq \Omega(\omega_{\text{gap}}/(\epsilon \lambda^3))$. Let $t_1$ denote the window size for each window in $X$, and $t_2$ denote the window size for each window in $Y$.

We carefully select $X' \subseteq X$ and $Y' \subseteq Y$ such that

1) $|X'| \geq \Omega(1/(\epsilon \lambda^3))$.
2) $|Y'| \geq \Omega(1/(\epsilon \lambda^3))$.
3) $|X'|t_1 \geq (1 + O(\epsilon))|Y'|t_2$.

To do this, if $|X'|t_1 > (1 + O(\epsilon))|Y'|t_2$, then we set $Y' = Y$ and select $X'$ as an arbitrary subset of $X$ with size $\frac{|Y'|t_2}{|X'|t_1}$. If $|X'|t_1 < (1 - O(\epsilon))|Y'|t_2$, then we set $X' = X$ and select $Y'$ an arbitrary subset of $Y$ with size $\frac{|X'|t_1}{t_2}$.

We define a window-based (bipartite) graph $G_W = (V_W, E_W)$ to be the subgraph of NG$_\lambda$ on $V_W = X' \cup Y'$ removing all the edges within $X'$ and all the edges within
Figure 4: The graph on the left is an example of the string-based graph and the graph on the right is an example of the character-based graph.

Let \( l_1 = |X'| \) and \( l_2 = |Y'| \). The total number of nodes in window-based graph is \( l_1 + l_2 \), and the window-based graph is a bi-clique.

We also define a character-based (bipartite) graph \( G_C = (V_C, E_C) \) as follows: Each window of \( X' \) has \( t_1 \) nodes in the character-based graph such that each node represents a character of the window. Similarly, each window of \( Y' \) has \( t_2 \) nodes in the character-based graph. Two nodes \( x, y \) in the character-based graph are adjacent iff \((x, y) \in \text{opt}_{ij}\) where \( w_i \) is the window containing character \( x \) and \( w_j \) is the window containing character \( y \). Hence, the total number of nodes in character-based graph is \( l_1 + l_2 \). The total number of edges in character-based graph is

\[
l_1l_2\lambda\sqrt{t_1t_2} \leq |E(G_C)| \leq l_1l_2\min\{t_1, t_2\}.
\]

The average degree of the character-based graph is at least \( 2l_1l_2\lambda\sqrt{t_1t_2}/(l_1t_1 + l_2t_2) \). By Blakley-Roy inequality (Lemma V.3), the number of walks of length 3 in the character-based graph is at least

\[
|V(G_C)| \cdot \left(\frac{2l_1l_2\lambda\sqrt{t_1t_2}}{l_1t_1 + l_2t_2}\right)^3
\]

\[= (l_1t_1 + l_2t_2) \cdot \left(\frac{2l_1l_2\lambda\sqrt{t_1t_2}}{l_1t_1 + l_2t_2}\right)^3\]

\[= 8\lambda^3 \frac{(l_1l_2\sqrt{t_1t_2})^3}{(l_1t_1 + l_2t_2)^2}.
\]

But many of the 3-walks are degenerate. For the number of 2-walks, we can upper bound it by

\[
\#2\text{-walks} \leq (\text{max degree of } G_W) \cdot 2|E(G_C)|
\]

\[\leq \max\{l_1, l_2\} \cdot 2(l_1l_2\min\{t_1, t_2\}) \]

\[= 2(1 + O(\epsilon))l_1^2l_2t_1.
\]

Note that the \( \#1\)-walks is \( 2|E(G_C)| \). We have

\[
\frac{\#2\text{-walks} + \#1\text{-walks}}{\#3\text{-walks}} \leq 2(1 + O(\epsilon))l_1^2t_1 + 2l_1l_2\min\{t_1, t_2\}
\]

\[\leq 2(1 + O(\epsilon))l_1^2t_1 + 2l_1l_2\min\{t_1, t_2\}\]

\[\leq 2\lambda^3 \frac{V^3}{l_1t_1 + l_2t_2}.
\]

where the second step and third step follow from \( l_1t_1 = (1 \pm O(\epsilon))l_2t_2 \), and the last step follows from \( l_1 = \Omega(1/(\epsilon^3)), l_2 = \Omega(1/(\epsilon^3)) \). Hence, the total number of 3-paths (3-walks that are not degenerate) is at least

\[8(1 - O(\epsilon))\lambda^3 \frac{l_1l_2\sqrt{t_1t_2}}{(l_1t_1 + l_2t_2)^2}.
\]

On the other hand, the number of 4-tuples in window-based graph is \( 8(\epsilon^2)\lambda^2 \leq 2l_1^2t_2 \). Thus, there must exist a 4-tuple containing at least

\[8(1 - O(\epsilon))\lambda^3 \frac{l_1l_2\sqrt{t_1t_2}}{(l_1t_1 + l_2t_2)^2} \cdot \frac{1}{2l_1^2t_2} =
\]

\[8(1 - O(\epsilon))\lambda^3 \frac{l_1l_2\sqrt{t_1t_2}}{(l_1t_1 + l_2t_2)^2} \cdot \frac{1}{2l_1^2t_2} \geq
\]

\[4(1 - O(\epsilon))\lambda^3 \sqrt{t_1t_2} \cdot \frac{1}{4 + O(\epsilon)} \geq
\]

many 3-walks where the third step follows from \( l_1t_1 = (1 \pm O(\epsilon))l_2t_2 \). Rescaling the factor of \( \epsilon \) for \( |X| \) and \(|Y| \) completes the proof.
Definition III.6 (reconstructive tuple). We say a tuple $(w_1, w_a, w_b, w_j)$ is reconstructive if it is $(c, \lambda)$-constructive and also $(i, a), (a, b), (b, j), (i, j) \in E(NG_\lambda)$.

Before proving Lemma III.10, we need to define a new graph $NF_\lambda$.

Definition III.7 ($NF_\lambda$). We construct a graph $NF_\lambda$ in the following way: for each node $v \in NG_\lambda$ we keep $v$ in $NF_\lambda$ with probability $p = k^{-\alpha}/4$ where $\alpha = 1 - \gamma/4$. For each $u, v \in NF_\lambda$, if $(u, v) \in NG_\lambda$ then we also draw an edge $u, v$ in $NF_\lambda$.

It is obvious that $V(NF_\lambda) \subseteq V(NG_\lambda)$ and $E(NF_\lambda) \subseteq E(NG_\lambda)$.

Based on definition of $NF_\lambda$, we are able to show that if $NG_\lambda$ is dense, so is $NF_\lambda$.

Claim III.8 ($NF_\lambda$ is a dense graph). Let $w_{\text{layers}}$ be the number of different window sizes. Let $NG_\lambda$ denote a graph such that $|E(NG_\lambda)| \geq k^{2-\beta} \cdot w_{\text{layers}}$ for some $\beta$ and $NF_\lambda$ be defined as Definition III.7. If $\gamma/4 - \beta = \Omega(1)$, then with probability at least 0.98, we have

$$|E(NF_\lambda)| = \Omega(|V(NF_\lambda)|^{2-4\beta/\gamma} \cdot w_{\text{layers}}^2).$$

Proof: Based on the sampling rate, we know that the following hold in expectation:

$$\mathbb{E}[|V(NF_\lambda)|] = \frac{1}{4k^\alpha}|V(NG_\lambda)|,$$

and

$$\mathbb{E}[|E(NF_\lambda)|] = \frac{1}{4^2k^{2\alpha}}|E(NG_\lambda)|.$$

Using standard Chernoff bound, we have with probability 0.99,

$$|V(NF_\lambda)| \in [1/2, 2] \cdot \frac{1}{4k^\alpha}|V(NG_\lambda)| = [1/2, 2] \cdot \frac{1}{4} k^{1-\alpha} = [1/2, 2] \cdot \frac{1}{4} k^{\gamma/4},$$

To show the concentration of $|E(NF_\lambda)|$, we define a random variable $x_{u,v}$ such that $x_{u,v} = 1$ if $(u, v) \in E(NF_\lambda)$, $0$ otherwise. Then we have

$$\mathbb{E}[|E(NF_\lambda)|] = \mathbb{E}\left[\sum_{(u,v) \in E(NF_\lambda)} x_{u,v}\right] = \mathbb{E}\left[\sum_{(u,v) \in E(NG_\lambda)} x_{u,v}^2\right] = \sum_{(u,v) \in E(NG_\lambda)} \mathbb{E}[x_{u,v}^2] = \sum_{(u,v) \in E(NG_\lambda)} \mathbb{E}[1_{v \neq u'x_{u,v}^2}] + \mathbb{E}[1_{u \neq u'x_{u,v}^2}] \leq p^2|E(NG_\lambda)| + p^3 \cdot \#2\text{-walks} + p^4 \cdot (|E(NG_\lambda)|^2 - \#2\text{-walks} - |E(NG_\lambda)|).$$

In order to use Chebyshev’s inequality, we need to make sure

$$p^2|E(NG_\lambda)| > 1,$$

and

$$p^2|E(NG_\lambda)| \geq 100(2p^3|E(NG_\lambda)||E(NG_\lambda)|)^{1/2}.$$ 

By $p = k^{\gamma/4-1}/4$ and $|E(NG_\lambda)| \geq k^{2-\beta}$, we need $k^{\gamma/2-\beta}/16 > 1$,

and the second condition is equivalent to

$$p|E(NG_\lambda)| \geq 20000|V(NG_\lambda)|.$$ 

We just need $k^{\gamma/4-\beta} \geq 80000$.

Using Chebyshev’s inequality, we have

$$\Pr\left[|E(NF_\lambda)| - \mathbb{E}[|E(NF_\lambda)|] \geq \tau \sqrt{\mathbb{Var}[|E(NF_\lambda)|]}\right] \leq \frac{1}{\tau^2},$$

combining with $\sqrt{\mathbb{Var}[|E(NF_\lambda)|]} \leq \frac{1}{100} \mathbb{E}[|E(NF_\lambda)|]$ together implies

$$\Pr\left[|E(NF_\lambda)| - \mathbb{E}[|E(NF_\lambda)|] \geq \tau \frac{1}{100} \mathbb{E}[|E(NF_\lambda)|]\right] \leq \frac{1}{\tau^2}.$$
Choosing \( \tau = 10 \), we have with probability 0.99 that 
\[ |E(NF_\lambda)| > 0.9|E(NF_\lambda)|, \]
and consequently
\[ |E(NF_\lambda)| > 0.9|E(NF_\lambda)| \]
\[ \geq 0.9 \cdot \frac{1}{4^{k/2}} |E(NG_\lambda)| \]
\[ \geq 0.9 \cdot \frac{1}{4^{k/2-2}} 2^{-\beta} \cdot w_{\text{layers}} \]
\[ = 0.9 \cdot \frac{1}{4^{k/2-2}} 2^{\beta} \cdot w_{\text{layers}} \]
\[ \geq 0.9 \cdot \frac{1}{4^{k/2-2}} 2^{\beta} \cdot w_{\text{layers}} \]
\[ = \Omega(|V(NF_\lambda)|^{2-\beta/\gamma} \cdot w_{\text{layers}}^2) \]
Thus, we complete the proof.

Next, we show

**Claim III.9** (NF_\lambda has no reconstructive tuple). Let NF_\lambda be defined as Definition III.7. With probability at least 0.99, there is no reconstructive tuple in NF_\lambda.

**Proof:** By Lemma III.3 and the definition of NG_\lambda, there is no pair of windows \((w_i, w_j)\) in NG_\lambda such that pair \((i, j)\) is well-connected. Formally speaking, for all pairs of windows \((w_i, w_j)\), there are at most \(k^2 - \gamma\) pairs of windows \((w_a, w_b)\) such that \((w_i, w_j)\) is \((\alpha, \lambda)\)-constructive.

For a fixed \((i, a, b, j)\), the probability that we keep it in NF_\lambda is \((k^2-\alpha)/4\). We can compute the expected number of constructive tuples in NF_\lambda,

\[ E[\#\text{constructive tuple}] \leq k^2 \cdot k^2 - \gamma \cdot (k^2-\alpha)/4 \]
\[ = k^4 - \gamma - 4\alpha/256 = 1/256 \]

where the last step follows from \(\alpha = 1 - \gamma/4\).

By Markov’s inequality, we have
\[ \Pr[\#\text{constructive tuple} \geq 1/2] \leq \frac{E[\#\text{constructive tuple}]}{1/2} \]
\[ \leq 1/100. \]

Thus, with probability 0.99, there is no reconstructive tuple in NF_\lambda.

Finally, we use Lemma III.6 to prove an upper bound on the number of edges in NG_\lambda.

**Lemma III.10.** Let NG_\lambda be as defined in Definition III.4. Then with probability at least 0.9, NG_\lambda has at most \(k^2 - \Omega(\gamma \lambda \omega_{\text{gap}}) \cdot w_{\text{layers}}^2\) edges.

**Proof:** We start with assuming
\[ |E(NG_\lambda)| \geq k^2 - \beta \cdot w_{\text{layers}}^2, \]
and will get the contradiction at the end.

We construct a graph NF_\lambda based on Definition III.7. Using Claim III.9, we know that NF_\lambda has no reconstructive tuple. Next, we are going to show two facts, and they are contradicting to each other by some choice of \(\beta\).

(Fact A). Note that NF_\lambda does not satisfy the assumption in Lemma III.5, thus we cannot apply Lemma III.5 to NF_\lambda directly. Let \(w_{\text{layers}}\) denote the number of different window sizes in total, we can decompose NF_\lambda into \(w_{\text{layers}}\) graphs such that \(NF_\lambda^{i,j}\) only involves size \(i\) and \(j\), \(\forall i, j \in [w_{\text{layers}}] \times [w_{\text{layers}}]\). Now, applying Lemma III.5, for every \(i, j\), we imply that if \(NF_\lambda^{i,j}\) has no reconstructive tuple, then \(NF_\lambda^{i,j}\) has no bipartite graph of certain size, i.e.,

\[ K_{\Omega(w_{\text{gap}}/\epsilon \lambda^3), \Omega(w_{\text{gap}}/\epsilon \lambda^3)} \]

(Fact B). Using the Turán’s Theorem (Lemma V.4), we know for any integer \(s \geq 2\), a graph \(G\) with \(n\) vertices and \(\Omega(n^{2-1/s})\) edges has at least one \(K_{s,s}\) subgraph. We consider graph NF_\lambda. Since

\[ |E(NF_\lambda)| = \Omega(|V(NF_\lambda)|^{2-\gamma/\lambda} \cdot w_{\text{layers}}^2) \]

by the pigeonhole principle, there exist \(i\) and \(j\) such that \(|E(NF_\lambda)| \geq \Omega(|V(NF_\lambda)|^{2-\gamma/\lambda} \cdot w_{\text{layers}}^2)\).

By Turán’s Theorem, such an \(NF_\lambda^{i,j}\) contains a large complete bipartite subgraph \(K_{\gamma/(\beta \lambda^3), \gamma/(\beta \lambda^3)}\).

In order to get a contradiction, we just need
\[ \gamma/(\beta \lambda^3) \geq c \cdot w_{\text{gap}}/\epsilon \lambda^3. \]

for some sufficiently large constant \(c > 1\) related to Lemma III.5. Thus, we have
\[ \beta \leq \gamma \epsilon \lambda^3/(4 \epsilon \cdot w_{\text{gap}}). \]

**Theorem III.11** (quadratic sparsification for constant \(\lambda\)). Given \(k\) windows \(w_1, \ldots, w_k\). Let \(w_{\text{max}} = \max_{i \in [k]} |w_i|\), \(w_{\text{min}} = \min_{i \in [k]} |w_i|\) and \(w_{\text{gap}} = w_{\text{max}}/w_{\text{min}}\). Let the number of different window sizes be \(w_{\text{layers}}\). For any \(\lambda \in (0, 1)\) and \(\epsilon \in (0, 1/10)\), there is a randomized algorithm (Algorithm I) that runs in time
\[ O(\epsilon^2 \lambda^4 k \log k + k^2 \cdot \log^2 k), \]

outputs a table \(\tilde{O}_{\lambda^2} \in \{0, 1\}^{k \times k}\) such that
\[ ||\text{ls}(w_i, w_j)|| \geq (1 - \epsilon)\lambda^3, \text{ if } \tilde{O}_{\lambda^2}[i][j] = 1 \]
and
\[ \left\{ (i, j) \mid ||\text{ls}(w_i, w_j)|| \geq \lambda, \text{ and } \tilde{O}_{\lambda^2}[i][j] = 0 \right\} = O \left( k^2 - \Omega(\lambda \omega_{\text{gap}}) \cdot w_{\text{layers}}^2 \right). \]

The algorithm has success probability at least \(1 - 1/\text{poly}(k)\).

**Proof:** The overall running time is
\[ O(|S| \cdot w_{\text{max}}^2 + k^2 \cdot |S|^2 w_{\text{max}}) = O(k \cdot k^7 \log k \cdot w_{\text{max}}^2 + k^2 \cdot \log^2 k \cdot w_{\text{max}}) = O(k \cdot k^7 \log k \cdot w_{\text{max}}^2 + k^2 \cdot \log^2 k \cdot w_{\text{max}}) \]
where the first step follows from \(|S| = O(k^7 \log k)\), the second step follows from \(\gamma = 0.1\).
The guarantee of table $\tilde{O}_{\lambda^{2}}$ follows from properties of graph $NG_{\lambda}$ (Algorithm 1 provides the first property of table in Theorem statement, Lemma III.10 provides the second property of table in Theorem statement).

\section*{B. Sparsification for $O_{\lambda^{3}}$}

One shortcoming of Lemma III.10 is that the number of remaining edges is only truly subquadratic if $\lambda$ is constant. As we discuss in Section III-A, the overall running time of the algorithm depends on the number of edges in the remaining graph and in order for the running time to be truly subquadratic, we need to reduce the number of edges to truly subquadratic. In this section, we show how one can obtain this bound even when $\lambda$ is super-constant. However, instead of losing a factor $\lambda^{3}$ in the approximation, our technique loses a factor of $O(\lambda^{3})$.

\textbf{Definition III.12 ($lcs_{w_{a}}(w_{i}, w_{j})$).} For two windows $w_{i}$ and $w_{j}$, and a window $w_{a}$, define $lcs_{w_{a}}(w_{i}, w_{j})$ as the length of LCS of $\text{opt}_{i,a}$ and $w_{j}$, where $\text{opt}_{i,a}$ denotes the LCS of $w_{i}$ and $w_{a}$.

Notice that unlike lcs, this new definition is not symmetric. Similar to lcs, we also normalize the size of $lcs_{a}$ by the geometric mean of the lengths of the two windows. That is, we divide the size of the common string by $\sqrt{|w_{i}||w_{j}|}$. In what follows, we first give an algorithm for detecting close pairs of windows, and then prove that the number of remaining pairs whose lcs is at least $\lambda$ is truly subquadratic.

Our algorithm is based on a data structure which we call $lcs$-cmp$(w_{a}, S, \lambda)$. Roughly speaking, we will choose $\bar{\lambda} = \lambda^{2}$ when we use it. Let us fix a threshold $\bar{\lambda}$ and a window $w_{a}$, lcs-cmp$(w_{a}, S, \bar{\lambda})$ receives a set $S$ of windows as input such that the size of each window of $S$ is at least $|w_{a}|$. Upon receiving the windows, it makes a preprocess in time $O(\lambda^{-2}|w_{a}| \sum_{w_{i} \in S}|w_{i}|)$. Next, lcs-cmp$(w_{a}, S, \lambda)$ would be able to answer each query of the following form in almost linear time:

- For two windows $w_{i}, w_{j} \in S$, either certify that $||lcs_{w_{a}}(w_{i}, w_{j})|| < \bar{\lambda}/2$ or find a solution for $||lcs_{w_{a}}(w_{i}, w_{j})||$ with a size of at least $\bar{\lambda}/2$.

We first show in Section III-B1, how lcs-cmp gives us a sparsification in truly subquadratic time and then discuss the algorithm for lcs-cmp in Section III-B2.

1) $\lambda^{3}$ Sparsification using lcs-cmp: Similar to what we did in Section III-A, we again use a parameter $\gamma$ in our algorithm and in the end, we adjust $\gamma$ to minimize the total running time. In our algorithm, we repeat the following procedure $k^{2} \log k$ times: sample a window $w_{a}$ uniformly at random and let $S$ be the set of all the other windows whose length is not smaller than $|w_{a}|$. Next, we obtain lcs-cmp$(w_{a}, S, \lambda)$ via running the preprocessing step. Finally, for each pair of windows $w_{i}, w_{j} \in S$, we make a query to lcs-cmp to verify one of the following two possibilities:

- $||lcs_{w_{a}}(w_{i}, w_{j})|| < \lambda^{2}/2$;
- $||lcs_{w_{a}}(w_{i}, w_{j})|| \geq \lambda^{4}/8$.

If the latter is verified we set $\tilde{O}_{\lambda^{3}/8}[i][j]$ to 1 otherwise we take no action. In what follows, we prove that after the above sparsification, the number of edges in the remaining graph is truly subquadratic.

\textbf{Algorithm 2 Sparsification for $O_{\lambda^{3}}$}

\begin{enumerate}
\item \textbf{procedure} CUBICSPARSIFICATION($w_{1}, w_{2}, \ldots, w_{k}, \lambda$) \hfill $\triangleright$ Theorem III.19
\item $\gamma \leftarrow 0.1$
\item $\lambda \leftarrow \lambda^{2}$
\item \textbf{for} counter $= 1 \rightarrow k^{2} \log k$ \textbf{do}
\item \textbf{for} $i = 1 \rightarrow k$ \textbf{do}
\item $\tilde{\lambda} \leftarrow \tilde{\lambda}^{2}$
\item \textbf{end for}
\item \textbf{end for}
\item \textbf{end for}
\item \textbf{end procedure}
\end{enumerate}

In the rest of this section, we bound $||(i, j) \in ||lcs_{w_{a}}(w_{i}, w_{j})|| \geq \lambda$ and $\tilde{O}_{\lambda^{3}/8}[i][j] = 0\}$. In other words, we prove an upper bound on the number of edges that remain in the graph. Define a graph $NG_{\lambda}$ such that each vertex of $NG_{\lambda}$ corresponds to a window and an edge $(i, j)$ means that $||lcs_{w_{a}}(w_{i}, w_{j})|| \geq \lambda$ but $\tilde{O}_{\lambda^{3}/8}[i][j] = 0$. The goal is to prove an upper bound on the number of edges of $NG_{\lambda}$. We formally prove this in Lemma III.18.

We define a notation called “close” which is similar to “well-connected” in Section III-A. In order to avoid confusion, we use “close” instead of well-connected.

\textbf{Definition III.13 (close).} Let $\gamma \in (0, 1) \text{ and } \lambda \in (0, 1)$. We say a pair $(w_{i}, w_{j})$ of windows is close, if there are at least $k^{1-\gamma}$ windows $w_{a}$ such that $|w_{a}| \leq \min\{|w_{i}|, |w_{j}|\}$ and $||lcs_{w_{a}}(w_{i}, w_{j})|| \geq \lambda^{2}$.
Our first observation is that Algorithm 2 detects all the close pairs with high probability.

**Lemma III.14.** Let \( \tilde{O}_{3/8} \in \{0, 1\}^{k \times k} \) be the output of Algorithm 2. For each \((i, j) \in [k] \times [k]\), if \((w_i, w_j)\) is close (Definition III.13) and \( ||\text{lcs}(w_i, w_j)|| \geq \lambda\), then \( \tilde{O}_{3/8}[i][j] = 1 \) holds with probability at least \( 1 - 1/poly(k) \).

**Proof:** We consider a fixed \((i, j)\) such that \((w_i, w_j)\) is close. By Definition III.13, there are at least \( k^{1-\gamma} w_a \) such that \( |w_a| \leq \min\{|w_i|, |w_j|\} \) and \( ||\text{lcs}_{w_a}(w_i, w_j)|| \geq \lambda^2 \). Then the probability that none of these windows is sampled is at most
\[
\left(1 - \frac{k^{1-\gamma}}{k}\right)^t = \left(1 - \frac{1}{k^\gamma}\right)^{10k^\gamma \log k} \leq 1/poly(k).
\]

Taking a union over at most \( k^2 \) pairs completes the proof.

Before we proceed to Lemma III.18, we bring Lemma III.15 as an auxiliary observation.

**Lemma III.15** (existence of a correlated pair). Let \( \epsilon \in (0, 1), \lambda \in (0, 1)\). Given a window \( w_a \) and a set of windows \( T \). Let \( w_{\text{gap}} \) denote the maximum size of windows divided by the minimum size of windows. Let \( T \) contain at most \( w_{\text{layers}} \) different sizes and \(|T| = \Omega(w_{\text{layers}} \cdot w_{\text{gap}}/\epsilon\lambda)\). If for each \( w_i \in T \) we have \( ||\text{lcs}(w_i, w_a)|| \geq \lambda\), then there exist two windows \( w_i, w_j \in T \) such that \( ||\text{lcs}_{w_a}(w_i, w_j)|| \geq (1 - \epsilon)\lambda^2 \).

**Proof:** Since \( T \) has size \( \Omega(w_{\text{layers}} \cdot w_{\text{gap}}/\epsilon\lambda) \) and the windows of \( T \) have at most \( w_{\text{layers}} \) different sizes, there must exists a subset \( T' \subseteq T \) such that \( |T'| = \Omega(w_{\text{gap}}/\epsilon\lambda) \) and all the windows in \( T' \) have the same size. In the rest of the proof, we will focus on \( w_a \) and \( T' \).

Let \( t_0 \) be the size of \( w_a \) and \( t \) be the size of all the windows in \( T' \). We consider a window-based bipartite graph, on one side it has one node \( w_a \) and on the other side it has \( |T'| \) nodes. Then we can expand the window-based bipartite graph into a character-based bipartite graph, on one side it has \( t_0 \) nodes, and on the other side it has \( t|T'| \) nodes. Two nodes \( x, y \) in the character-based graph are adjacent iff \((x, y) \in \text{opt}_{w_a,i}\) where \( w_a \) is the window containing character \( x \) and \( w_i \) is the window containing character \( y \). Since \( ||\text{lcs}(w_a, w_i)|| \geq \lambda \) for every \( w_i \in T' \), the number of edges in character-based bipartite graph is at least \( \lambda |T'| \sqrt{t_0 t} \).

For \( i \)-th character in window \( w_a \), we use \( D_i \) to denote the degree of the corresponding node in the character-based bipartite graph. The number of 2-walks is at least
\[
\sum_{i=1}^{t_0} D_i(D_i - 1) = \left( \sum_{i=1}^{t_0} D_i^2 - \sum_{i=1}^{t_0} D_i \right) \\
\geq \left( \frac{1}{t_0} \left( \sum_{i=1}^{t_0} D_i^2 \right)^2 - \sum_{i=1}^{t_0} D_i \right) \\
\geq \left( \frac{1}{t_0} (\lambda |T'| \sqrt{t_0 t})^2 - (\lambda |T'| \sqrt{t_0 t}) \right) \\
= (\lambda^2 t |T'|^2 - \lambda |T'| \sqrt{t_0 t}) \\
\geq (1 - \epsilon) \lambda^2 t |T'|^2.
\]

It remains to show Eq. (1). Since the number of edges in the character-based graph are adjacent (existence of a correlated pair) \( |T'| = \Omega(w_{\text{gap}}/(\epsilon\lambda)) \), we have
\[
\sum_{i=1}^{t_0} D_i \geq \lambda |T'| \sqrt{t_0 t}.
\]

By \( |T'| = \Omega(w_{\text{gap}}/(\epsilon\lambda)) \), we have
\[
\lambda |T'| \sqrt{t_0 t} \geq t_0.
\]

Due to a simple fact : \( x(x - z) > y(y - z) \) as long as \( x > y > z > 0 \), we have
\[
\frac{1}{t_0} \left( \sum_{i=1}^{t_0} D_i^2 \right)^2 - \sum_{i=1}^{t_0} D_i \geq \frac{1}{t_0} (\lambda |T'| \sqrt{t_0 t})^2 - (\lambda |T'| \sqrt{t_0 t}).
\]

Thus, there must exists a pair such that
\[
||\text{lcs}_{w_a}(w_i, w_j)|| \geq (1 - \epsilon)\lambda^2.
\]

**Lemma III.15**

**Definition III.16** (NG\(_\lambda\)). We assume that the edges of NG\(_\lambda\) are directed in the following way: For an edge \((i, j)\) if \(|w_i| \neq |w_j|\) the starting point of the edge would be the vertex corresponding to the longer window and the ending point of the edge would be the one corresponding to the shorter window. If both corresponding windows have the same lengths, then we use an arbitrary direction for \((i, j)\).

**Definition III.17** (NF\(_\lambda\)). We construct graph NF\(_\lambda\) in the following way : let a denote the node in \( V(NG\_\lambda) \) that has the highest outgoing degree, let \( V(NF\_\lambda) = N(a) \). We add an edge between vertices \((i, j)\) in NF\(_\lambda\) if \( ||\text{lcs}_{w_a}(w_i, w_j)|| \geq (1 - \epsilon)\lambda^2 \). Give directions to the edges of NF\(_\lambda\) the same way we did it for NG\(_\lambda\).

Now, we are ready to prove the main observation of this section.

**Lemma III.18 (upper bound on E(NG\(_\lambda\))).** Let NG\(_\lambda\) be defined as Definition III.4. Then
\[
|E(NG\_\lambda)| = O(k^{2-\gamma} \cdot w_{\text{layers}} \cdot w_{\text{gap}}/(\epsilon\lambda)).
\]
holds with probability $1 - 1/\text{poly}(k)$.

Proof: We start with assuming
\[
|E(NG_{\lambda})| \geq \Omega(k^{2-\beta} \cdot w_{\text{layers}} \cdot w_{\text{gap}}/(\epsilon \lambda)).
\]
We construct a graph $NF_{\lambda}$ based on Definition III.17. By Definition III.17, we know that $|V(NF_{\lambda})| \geq \Omega(k^{1-\beta} \cdot w_{\text{layers}} \cdot w_{\text{gap}}/(\epsilon \lambda))$. By choosing some $\beta = \gamma$ we will make a contradiction.

Using Lemma III.15, we have for each set $T \subseteq V(NF_{\lambda})$ with $|T| = \Omega(w_{\text{layers}} \cdot w_{\text{gap}}/(\epsilon \lambda))$, there exist two nodes $u$ and $v$ in $T$ such that edge $(u, v) \in NF_{\lambda}$.

If we look at the complement of graph $NF_{\lambda}$, we know there is no clique $K_{r+1}$ where $r = O(w_{\text{layers}} \cdot w_{\text{gap}}/(\epsilon \lambda))$. Using Turan’s theorem (Lemma V.5) we know that complement of graph $NF_{\lambda}$ has at most $(1 - \frac{1}{r}) \frac{k^2}{2}$ edges, where $q = |V(NF_{\lambda})|$. Then we have
\[
|E(NF_{\lambda})| \geq \frac{q(q-1)}{2} - (1 - \frac{1}{r}) \frac{k^2}{2} = \frac{q^2 - q}{2} - \frac{q^2}{2r}.
\]
This implies that there is one vertex $j$ whose outgoing degree in $NF_{\lambda}$ is at least
\[
|V(NF_{\lambda})| - \frac{1}{2} \geq |V(NF_{\lambda})| \cdot \Omega \left( \frac{\epsilon \lambda}{w_{\text{layers}} \cdot w_{\text{gap}}} \right)
\geq k^{1-\beta}
\geq k^{1-\gamma}.
\]
Thus, there is a node in $NG_{\lambda}$ that has a lot of outgoing edges, which means $NG_{\lambda}$ has a close pair.

On the other hand, Using definition of $NG_{\lambda}$ and Lemma III.14, we know that $NG_{\lambda}$ should not contain any close pair. Thus, we get a contradiction.

Theorem III.19 (cubic sparsification for arbitrary $\lambda$). Given $k$ windows $w_1, \ldots, w_k$. Let $w_{\text{layers}}$ denote the number of different sizes for windows. Let $w_{\text{max}} = \max_{i \in [k]} |w_i|$, $w_{\text{min}} = \min_{i \in [k]} |w_i|$ and $w_{\text{gap}} = w_{\text{max}}/w_{\text{min}}$. For any $\lambda \in (0, 1)$, $\gamma \in (0, 1)$, there is a randomized algorithm (Algorithm 2) that runs in time
\[
O(\lambda^{-4}k^{1+\gamma}w_{\text{max}}^2 \log k + k^{2+\gamma}w_{\text{max}} \log k)
\]
and outputs a table $\tilde{\Omega}_{\lambda^\gamma} \in \{0, 1\}^{k \times k}$ such that
\[
||\text{lcs}(w_i, w_j)|| \geq \lambda^{3/4}, \text{ if } \tilde{\Omega}_{\lambda^\gamma}[i][j] = 1
\]
and
\[
\left| \{(i, j) \mid ||\text{lcs}(w_i, w_j)|| \geq \lambda, \text{ and } \tilde{\Omega}_{\lambda^\gamma}[i][j] = 0 \} \right| = O(k^{2-\gamma}w_{\text{layers}}w_{\text{gap}}/\lambda).
\]
The algorithm has success probability $1 - 1/\text{poly}(k)$.

Proof: The running time of each round of Algorithm 2 is
\[
= O(\text{ constructing } S \text{ time } + \text{lcs-cmp.INITIAL time }) + |S|^2 \cdot (\text{lcs-cmp.QUERY time })
= O(k + \tilde{\lambda}^{-2}|S|w_{\text{max}}^2 + |S|^2w_{\text{max}})
= O(\lambda^{-4}k^2w_{\text{max}}^2 + k^2w_{\text{max}})
\]
where the last step follows by $\tilde{\lambda} = \lambda^2$.
Since it repeats $O(k\gamma \log k)$ rounds, thus the overall running time is
\[
O(k^\gamma \log k) \cdot O(\lambda^{-4}k^2w_{\text{max}}^2 + k^2w_{\text{max}})
= O(\lambda^{-4}k^{1+\gamma}w_{\text{max}}^2 \log k + k^{2+\gamma}w_{\text{max}} \log k).
\]

2) Implementation of lcs-cmp. Given $\tilde{\lambda}$, $w_a$ and $w_1, w_2, \ldots, w_s$, we present an $O(\lambda^{-1}|w_a|\sum_{i=1}^s |w_i|)$ time preprocessing algorithm and $O(|w_a|)$ time query algorithm for lcs-cmp such that for any $i, j \in [s]$
1) if $||\text{lcs}_{w_a}(w_i, w_j)|| > \lambda^2/8$, then the query algorithm outputs accept;
2) if $||\text{lcs}_{w_a}(w_i, w_j)|| \leq \lambda^2/8$, then the query algorithm outputs reject.

Algorithm 3 is our lcs-cmp preprocessing and query algorithm. The high level idea is to compute $\text{opt}_{a,i}$ for every $i \in [s]$ and a set of at most $2/\lambda$ common subsequences between $w_a$ and $w_i$ for each $i$ such that every character of $w_a$ appears at most once among the sequences for a fixed $i$. $\text{lcs}_{w_a}(w_i, w_j)$ is approximated by taking the longest intersection of $\text{opt}_{a,i}$ and some sequence in the set of window $w_j$. Also see Figure 5 for the intuition.

Now we prove that Algorithm 3 implements lcs-cmp.

Lemma III.20 (INITIAL). Given parameter $\tilde{\lambda}$, a window $w_a$ and a set of windows $w_1, \ldots, w_s$. The INITIAL of data structure lcs-cmp (Algorithm 3) takes $O(\lambda^{-1}|w_a|\sum_{i=1}^s |w_i|)$ time, and outputs $\{X_{a,i} \}_{i \in [s]}$ and $\{Y_{a,i} \}_{i \in [s]}$ such that
1) $X_{a,i}$ corresponds to indices of a longest common subsequence between $w_a$ and $w_i$ for every $i \in [s]$;
2) $Y_{a,i}$ corresponds to indices of at most $2/\tilde{\lambda}$ common subsequence between $w_a$ and $w_i$ for every $i \in [s]$;
3) If none of the element indices of a common subsequence between $w_a$ and $w_i$ is in $Y_{a,i}$, then the length of this common subsequence is less than $\tilde{\lambda}|w_a|/2$.

Proof: The first property follows from the definition of the algorithm.
For the second and third property, if an $\text{opt}'_{a,i}$ obtained in Line 9 has length less than $\tilde{\lambda}|w_a|/2$, then the element indices of $\text{opt}'_{a,i}$ are not in $Y_{a,i}$. Hence, the third property holds. Also, it means that every time that Line 13 is executed, the size of $Y_{a,i}$ increases by at least $\tilde{\lambda}|w_a|/2$. For a fixed
Lemma III.21 (QUERY). For any \((i, j) \in \{s\} \times \{s\}\), the query of data structure \(lcs\)-cmp (Algorithm 3) runs in time \(O(|w_a|)\) with the following properties:

1. If \(|lcs_{wa}(w_i, w_j)| > \lambda\), then it outputs accept.
2. If \(|lcs_{wa}(w_i, w_j)| \leq \lambda^2/8\), then it outputs reject.

Proof: Let \(opt_{a,i,j}\) be the LCS between \(opt_{a,i}\) and \(w_j\).
By the definition, we have

\[|lcs_{wa}(w_i, w_j)| = \frac{|opt_{a,i,j}|}{\sqrt{|w_i||w_j|}}\]

Consider the case of \(|lcs_{wa}(w_i, w_j)| \geq \lambda\). By the third property of Lemma III.20, there are less than \(\lambda|w_a|/2\) elements of \(opt_{a,i,j}\) with indices not in \(Y_{a,j}\). So we have

\[|X_{a,i} \cap Y_{a,j}| > \lambda \sqrt{|w_i||w_j|} - \frac{1}{2} \lambda|w_a| \geq \frac{1}{2} \lambda \sqrt{|w_i||w_j|}\]

where the last inequality follows from the fact that \(|w_a|\) is smaller than or equal to \(|w_i|\) for every \(i \in \{s\}\). Hence the algorithm accepts.

Consider the case of \(|lcs_{wa}(w_i, w_j)| \leq \lambda^2/8\). By the second property of Lemma III.20, \(Y_{a,j}\) corresponds to at most \(2\lambda/\lambda\) mutually non-overlapping common subsequence between \(w_a\) and \(w_j\). Since \(|lcs_{wa}(w_i, w_j)| \leq \lambda^2/8\), the set of indices of every such a common subsequence has an intersection with \(X_{a,i}\) of size at most \(\lambda^2 \sqrt{|w_i||w_j|}/8\). Thus, we have

\[|X_{a,i} \cap Y_{a,j}| \leq \frac{1}{8} \lambda^2 \sqrt{|w_i||w_j|} \cdot \frac{2}{\lambda} \leq \frac{1}{4} \lambda \sqrt{|w_i||w_j|} \cdot \frac{2}{\lambda}.\]

Hence the algorithm rejects for this case. 

IV. LONGEST INCREASING SUBSEQUENCE
We outlined the algorithm in Section I-C3. Here, we bring the details for each step of the algorithm. In the interest of space, we omit some of the proofs and bring them in the full-version. In Section IV-A we discuss the solution domains and show how we construct them. Next, in Section IV-B we discuss the details of constructing pseudo-solutions and finally in Section IV-C we show how we can obtain an approximate solution from the pseudo-solutions. Also, in Section IV-D we bring an improvement to the running time at the expense of having a larger approximation factor for the algorithm.

A. Solution Domains
We assume from now on that \(lisc(a) > n\lambda\) holds. As mentioned earlier, we divide the input array into \(\sqrt{n}\) subarrays of size \(\sqrt{n}\) and denote them by \(s_{a1}, s_{a2}, \ldots, s_{a\sqrt{n}}\). For a fixed optimal solution \(opt\), our goal is to approximate the smallest and the largest number of each subarray that contributes to \(opt\). Let us refer to these numbers as the domain of each subarray. Let \(\epsilon = 1/1000\) be accuracy. For a subarray \(sa_i\), we sample \(k\) (will be decided later) different elements and refer to them by \(a_{j1}, a_{j2}, \ldots, a_{jk}\).

We first prove that,

Lemma IV.1 (constructing candidate domains). Let \(\lambda \in (0, 1), \epsilon \in (0, 1/2)\) and \(\delta \in (0, 1/10)\). Let \(sa_i\) be a length-\(\sqrt{n}\) subarray whose contribution to the optimal solution is at least \(\epsilon \sqrt{n}\lambda\), i.e., \(lisc(s_{am(sa_i)})(sa_i) \geq \epsilon \sqrt{n}\lambda\).
If we uniformly sample \(k = 20\log(1/\delta)/(\lambda^2)\) elements \(a_{j1}, a_{j2}, \ldots, a_{jk}\) from \(sa_i\), then with probability at least

...
1. **data structure** lcs-cmp
2. **members**
3. \(X_{a,1}, \ldots, X_{a,s}\)
4. \(Y_{a,1}, \ldots, Y_{a,s}\)
5. \(\lambda \in (0,1)\)
6. end members
7. procedure INITIAL \((w_a, \{w_i\}_{i \in [s]}, \tilde{\lambda})\) \(\triangleright\) Lemma III.20
8. for \(i \in [s]\) do
9. compute \(\text{opt}_{a,i}\), a LCS between \(w_a\) and \(w_i\)
10. let \(X_{a,i}\) be the set of indices of all the elements of \(\text{opt}_{a,i}\) with respect to \(w_a\)
11. end for
12. for \(i \in [s]\) do
13. let \(Y_{a,i}\) be an empty set
14. while true do
15. compute \(\text{opt}'_{a,i}\), a LCS between \(w_a\) removing the elements with index in \(Y_{a,i}\) and \(w_i\)
16. if the length of \(\text{opt}'_{a,i}\) is less than \(\tilde{\lambda} \lfloor |w_a| / 2 \rfloor\) then
17. break
18. else
19. put the indices of all the elements of \(\text{opt}'_{a,i}\) with respect to \(w_a\) into \(Y_{a,i}\)
20. end if
21. end while
22. end for
23. return \(\{X_{a,i}\}_{i \in [s]}, \{Y_{a,i}\}_{i \in [s]}\)
24. end procedure
25. procedure QUERY \((w_1,w_j)\) \(\triangleright\) Lemma III.21
26. if \(|X_{a,i} \cap Y_{a,j}| > \lambda \sqrt{|w_i| \cdot |w_j|} \cdot 4\) then
27. return accept
28. else
29. return reject
30. end if
31. end procedure
32. end data structure

**Algorithm 4 Constructing the candidate domains**

1. **procedure** \(\text{CONSTRUCTCANDIDATEDOMAINS}(sa_i)\) \(\triangleright\) Lemma IV.1
2. \(\triangleright\) Given random access to a subarray \(sa_i\)
3. \(k \leftarrow 20 \log(1/\delta) / (\lambda \epsilon^2)\)
4. Sample \(k\) elements from \(sa_i\), and denote the sampled elements by \(a_{j_1}, a_{j_2}, \ldots, a_{j_k}\)
5. for \(\alpha\) in \([k]\) do
6. for \(\beta\) in \([k]\) do
7. If \(a_{j_\alpha} \leq a_{j_\beta}\), then construct a candidate domain \([a_{j_\alpha}, a_{j_\beta}]\)
8. end for
9. end for
10. return all the constructed candidate domains
11. end procedure

**Algorithm 3 lcs-cmp data structure**

1138
1 - \exp(-\Omega(\epsilon^2\sqrt{n}\lambda)), there exist a set \( T \subseteq [\sqrt{n}] \) and elements \( \alpha_i \) and \( \beta_i \) sampled from \( sa_i \) for each \( i \in T \) such that the following conditions hold:

1) For any \( i \in T \), \( \alpha_i \leq \beta_i \).
2) For any \( i, j \in T \) satisfying \( i < j \), \( \beta_i < \alpha_j \).
3) \( \sum_{i \in T} \text{lis}^{[\alpha_i, \beta_i]}(sa_i) \geq (1 - 4\epsilon)\text{lis}(A).

**Proof:** Lemma IV.1 holds for all subarrays whose contribution to opt is at least \( \epsilon \sqrt{n}\lambda \). Let \( S \subseteq [\sqrt{n}] \) denote the set of coordinates such that for each \( i \in S \)

\[
\text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \geq n\lambda \quad \text{and} \quad \sum_{i \in S} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \leq \sqrt{n}.
\]

By Hoeffding inequality we have

\[
\sum_{i \in S} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \geq \sqrt{n} \cdot \epsilon \sqrt{n}\lambda \geq \text{lis}(A) - \epsilon n\lambda.
\]

Let \( T \subseteq S \) denote the set of coordinates such that for each \( i \in T \),

\[
\text{lis}^{[\alpha_i, \beta_i]}(sa_i) \geq (1 - \epsilon)\text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i),
\]

and

\[
\text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \geq \epsilon \sqrt{n}\lambda.
\]

Now we show that with probability at least \( 1 - \exp(-\Omega(\epsilon^2\sqrt{n}\lambda)) \),

\[
\sum_{i \in T} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \geq (1 - 2\epsilon) \sum_{i \in S} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i).
\]

For each \( i \in S \), let \( X_i \) denote a random variable such that

\[
X_i = \begin{cases} 
\text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) & \text{with probability } i \in T; \\
0 & \text{with probability } i \notin T,
\end{cases}
\]

and \( X = \sum_{i \in S} X_i \). By Lemma IV.1, we have \( \mathbb{E}[X] \geq (1 - \epsilon) \sum_{i \in S} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \). By Hoeffding bound (Theorem V.2, Equation (3)), we have

\[
\Pr[X - \mathbb{E}[X] \geq \epsilon\mathbb{E}[X]] \leq 2 \exp(-2\epsilon^2(\mathbb{E}[X])^2) \leq 2 \exp(\frac{-2\epsilon^2(1 - \epsilon)^2(\sum_{i \in S} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i))^2}{n^{3/2}}) \leq 2 \exp(-2\epsilon^2(1 - \epsilon)^4\sqrt{n}\lambda) \leq \exp(-\Omega(\epsilon^2\sqrt{n}\lambda)).
\]

Hence, Equation (3) holds with probability at least \( 1 - \exp(-\Omega(\epsilon^2\sqrt{n}\lambda)) \).

Conditioned on Equation (3), we have

\[
\sum_{i \in T} \text{lis}^{[\alpha_i, \beta_i]}(sa_i) \geq (1 - \epsilon)(1 - 2\epsilon) \sum_{i \in S} \text{lis}^{[\text{sm}(sa_i), \text{lg}(sa_i)]}(sa_i) \geq (1 - \epsilon)(1 - 2\epsilon)\text{lis}(A) - \epsilon n\lambda.
\]

where the first inequality follows from the definition of \( T \), the second inequality follows from Equation (3), the third inequality follows from Equation (3), and the last inequality follows from \( \text{lis}(A) \geq n\lambda \).

Finally, by Equation (3) and (4), we have

\[
\Pr \left[ \sum_{i \in T} \text{lis}^{[\alpha_i, \beta_i]}(sa_i) \geq (1 - 4\epsilon)\text{lis}(A) \right] \geq 1 - \exp(-\Omega(\epsilon^2\sqrt{n}\lambda)).
\]

**B. Constructing Approximately Optimal Pseudo-solutions**

We call a sequence of \( \sqrt{n} \) intervals \( [\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_{\sqrt{n}}, r_{\sqrt{n}}] \) a pseudo-solution if all of the intervals are monotone. That is \( \ell_1 \leq r_1 < \ell_2 \leq r_2 < \ell_3 \leq r_3 < \ldots < \ell_{\sqrt{n}} \leq r_{\sqrt{n}} \). These intervals denote solution-domains for the subarrays. We also may decide not assign any solution domain to a subarray in which case we show the corresponding interval by \( \emptyset \). Indeed, \( \emptyset \) does not break the monotonicity of a pseudo-solution. The quality of a pseudo-solution is defined as \( \sum_{i = 1}^{\sqrt{n}} \text{lis}^{[\ell_i, r_i]}(sa_i) \). We denote the quality of a pseudo-solution \( ps \) by \( q(ps) \).

Another way to interpret Corollary IV.2 is that one can construct a pseudo-solution using the sampled elements whose quality is at least a \( 1 - \epsilon \) fraction of \( \text{lis}(A) \). In this section, we present an algorithm to construct a small set of pseudo-solutions with the promise that at least one of them has a quality of at least \( \text{lis}(A)/t \), where \( t \) is the number of pseudo-solutions. Finally, in Section IV-C, we present a method to approximate the size of the optimal solution using pseudo-solutions.

We construct the pseudo-solutions via Algorithm 5. The input of Algorithm 5 is the set of candidate domain intervals obtained by Algorithm 4 on every subarray. We first find an assignment of candidate solution domains to the subarrays which is monotone and has the largest number of candidate domain intervals. (This step can be implemented by dynamic programming.) We make a pseudo-solution out of this assignment and update the set of candidate intervals by removing the ones which are used in our pseudo-solution. We then repeat the same procedure to construct the second

1139
Algorithm 5: Constructing the pseudo-solutions

1: \textbf{procedure} \hspace{1em} \text{CONSTRUCTPSEUDOSOLUTIONS}(cdi_1, \ldots, cdi_{\sqrt{n}}) \hspace{1em} \triangleright \hspace{1em} \text{Lemma IV.3, IV.4, IV.5}
2: \begin{align*}
& \triangleright \{cdi_i\}_{i \in [\sqrt{n}]} \text{ is } \sqrt{n} \text{ sets of candidate domain intervals} \\
& \text{pseudo-solutions} \leftarrow \emptyset \\
& \text{while true do} \\
& \quad \text{assg} \leftarrow \text{largest assignment of candidate domain intervals to subarrays which is monotone} \\
& \quad \text{if assg contains less than } \epsilon \sqrt{n} \lambda \text{ non-empty candidate domain intervals then} \\
& \quad \quad \text{break} \\
& \quad \text{else} \\
& \quad \quad \text{Add assg to pseudo-solutions} \\
& \quad \quad \text{for } i \leftarrow 1 \text{ to } \sqrt{n} \text{ do} \\
& \quad \quad \quad \text{if assg contains a candidate domain interval for subarray } sa_i \\
& \quad \quad \quad \quad \text{then} \\
& \quad \quad \quad \quad \text{remove the corresponding candidate domain interval from } cdi_i \\
& \quad \quad \quad \text{end if} \\
& \quad \quad \text{end for} \\
& \quad \text{end if} \\
& \text{end while} \\
& \text{return pseudo-solutions } p_{s_1}, p_{s_2}, \ldots, p_{s_t}
\end{align*}
\textbf{end procedure}

 pseudo-solution and update the candidate solution domains accordingly. We continue on, until the number of solution domains used in our pseudo-solution drops below $\epsilon \sqrt{n}$ in which case we stop.

We first prove in Lemma IV.3 that the number of pseudo-solutions constructed in Algorithm 5 is bounded by $O(k^2/(\lambda \epsilon))$. Next, we show in Lemma IV.4 that at least one of the pseudo-solutions constructed by Algorithm 5 has a quality of at least $\Omega(\text{lis}(A)/t)$ where $t$ is the number of pseudo-solutions. Finally we prove in Lemma IV.5 that the running time of Algorithm 5 is $O(tk^2/\sqrt{n} \log n)$.

Lemma IV.3 (number of pseudo-solutions). For each $i \in [\sqrt{n}]$, let $cdi_i$ be a set of at most $k^2$ candidate domain intervals. Let $t$ denote the number of pseudo-solutions constructed in Algorithm 5. Then, we have $t \leq k^2/(\lambda \epsilon)$.

Proof: Note that for each subarray we have at most $k^2$ candidate domain intervals. Since there are $\sqrt{n}$ subarrays, then in total we have $\sqrt{n}k^2$ candidate domain intervals. Each time we construct a pseudo-solution, the total number of the candidate domain intervals is decreased by at least $\epsilon \sqrt{n} \lambda$. Thus, the total number of pseudo-solutions $t$ can be upper bounded,

$$t \leq \frac{\sqrt{n}k^2}{\epsilon \sqrt{n} \lambda} \leq \frac{k^2}{\lambda \epsilon}.$$ 

Lemma IV.4 (quality of pseudo-solutions). Let $p_{s_1}, p_{s_2}, \ldots, p_{s_t}$ be the pseudo-solutions constructed by Algorithm 5. If $\text{lis}(A) \geq n \lambda$ holds, then with probability at least $1 - \exp(-\Omega(\sqrt{n} \lambda))$, there exists an $i \in [t]$ such that

$$q(p_{s_i}) \geq \frac{\text{lis}(A)}{2t}.$$ 

Proof: Let us focus again on the actual solution domains $[\text{sm}(sa_1), \text{lg}(sa_1)], [\text{sm}(sa_2), \text{lg}(sa_2)], \ldots,$ $[\text{sm}(sa_{\sqrt{n}}), \text{lg}(sa_{\sqrt{n}})]$.

We define set $S \subseteq [\sqrt{n}]$ such that

$$\text{lis}[\text{sm}(sa_i), \text{lg}(sa_i)] \geq \epsilon \sqrt{n} \lambda, \forall i \in S.$$ 

Using Corollary IV.2 with $\epsilon = 1/10$, we know that there is a monotone pseudo-solution $[\alpha_1, \beta_1] \in T (T \subseteq S)$ such that $[\alpha_1, \beta_1]$ are candidate domain intervals and

$$\sum_{i \in T} \text{lis}[\alpha_i, \beta_i](sa_i) \geq (1 - 4\epsilon)\text{lis}(A).$$

Denote this pseudo-solution as $s$.

At the time we terminate Algorithm 5, there are at most $\epsilon \sqrt{n} \lambda$ candidate domain intervals of $s$ does not belongs to any pseudo-solution of the pseudo-solution set. Also, since each candidate domain interval contributes at most $\sqrt{n}$ to the quality of the pseudo-solution containing the interval, we have

$$\sum_{i=1}^{t} q(p_{s_i}) \geq (1 - 4\epsilon)\text{lis}(A) - \epsilon \sqrt{n} \lambda \cdot \sqrt{n} \geq (1 - 4\epsilon)\text{lis}(A) - \epsilon \text{lis}(A) \implies (1 - 5\epsilon)\text{lis}(A).$$

Thus, there exists an $i \in [t]$ such that

$$q(p_{s_i}) \geq (1 - 5\epsilon)\text{lis}(A)/t = \text{lis}(A)/(2t).$$

Lemma IV.5 (running time). For each $i \in [\sqrt{n}]$, let $cdi_i$ be a set of at most $k^2$ candidate domain intervals. Let $t$ denote the number of pseudo-solutions. The running time of Algorithm 5 is bounded by $O(tk^2/\sqrt{n} \log n)$.

Proof: Lemma IV.3 states that Algorithm 5 terminates after constructing $t$ pseudo-solutions.

Now we show that constructing each pseudo-solution takes time $O(k^2/\sqrt{n} \log n)$. Our solution is based on a dynamic programming technique. Let $D : [\sqrt{n}] \times [k^2] \rightarrow \mathbb{N}$ be an array such that $D[i][j]$ stores the size of the largest monotone pseudo-solution for the first $i$ subarrays which ends with the $j$’th candidate domain interval of $sa_i$. Using classic segment-tree data structure (this data structure can be found in many textbooks, e.g. [26]), one can compute
Algorithm 6 Evaluate the pseudo solutions

1: procedure EVALUATEPSEUDOSOLUTIONS(ps₁, . . . , psᵢ) ▷ Lemma IV.6
2: \( p \leftarrow \frac{1000}{\log^{2} n} \) is a set of pseudo solutions
3: for each \( ps_j \) do
4: \( \tilde{q}(ps_j) \leftarrow 0 \)
5: for each interval \( [\ell_i, r_i] \) in \( ps_j \) do
6: if \( i \in W \) then
7: \( \tilde{q}(ps_j) \leftarrow \tilde{q}(ps_j) + \text{lis}[\ell_i, r_i](sa_i)/p \)
8: end if
9: end for
10: end for
11: return largest \( \tilde{q}(ps_j) \) for all \( j \in [I] \)
end procedure

the value of \( D[i][j] \) in time \( O(\log n) \) from the previously computed elements of the array.

Thus, the total running is bounded by \( O(tk^2\sqrt{n}\log n) \).

C. Evaluating the Pseudo-solutions

We finally use a concentration bound to show that the quality of a pseudo-solution can be approximated well by sampling a small number of subarrays. Since a pseudo-solution specifies the range of the numbers used in every subarray, the quality of a pseudo-solution, or in other words, the size of the corresponding increasing subsequence of a pseudo-solution can be formulated as

\[ q(p) = \sum_{i=1}^{n} \text{lis}[\ell_i, r_i](sa_i) \]

where \( [\ell_i, r_i] \) denotes the corresponding solution domain of \( ps \) for \( sa_i \).

In Lemma IV.6, we prove that by sampling \( O(\log^{O(1)} n/\lambda^4) \) many subarrays and computing \( \text{lis}[\ell_i, r_i](sa_i) \) for them, one can approximate the quality of a pseudo-solution pretty accurately.

Lemma IV.6 (the quality of pseudo-solution). Let \( \lambda \in (0, 1) \) and \( \epsilon \) be a constant in \( (0, 1/100) \). Let \( ps_1, ps_2, \ldots, ps_I \) be a set of \( t \) pseudo-solutions. With probability at least \( 1 - \exp(-\Omega(\log^2 n)) \), Algorithm 6 runs in time \( O(t^2\sqrt{n}\log^{O(1)} n/\lambda) \) such that

1. If there exists an \( i \in [I] \), \( q(ps_i) \geq \frac{\lambda n}{2t} \), then the algorithm outputs an estimation at least \( \frac{\lambda n}{2t} \).
2. If \( q(ps_i) < \frac{\lambda n}{2t} \) for all \( i \in [I] \), then the algorithm outputs an estimation smaller than \( \frac{\lambda n}{2t} \).

Putting all the previous Lemmas together gives the following result:

Corollary IV.7 (algorithm for LIS decision problem). Given a length-\( n \) sequence \( A \), let \( \lambda \in [1/n, 1] \). There is a randomized algorithm that runs in time \( O(\lambda^{-7}\sqrt{n}\log^{O(1)} n) \) such that with probability \( 1 - 1/poly(n) \)

- The algorithm accepts if \( \text{lis}(A) > n\lambda \).
- The algorithm rejects if \( \text{lis}(A) = O(n\lambda^4) \).

Proof: The correctness follows from Lemma IV.6, Lemma IV.4, and Lemma IV.5.

Running time: The running time is

\[
\text{time} = O(tk^2\sqrt{n}\log^{O(1)} n) + O(t^2\lambda^{-1}\sqrt{n}\log^{O(1)} n)
\]

Thus, we complete the proof.

Finally, by starting with \( \lambda = 1 \) and iteratively multiplying \( \lambda \) by a \( 1/(1 + \epsilon) \) factor until a solution is found, we can approximate \( \text{lis}(A) \) within an approximation factor of \( O(\lambda^3) \).

Theorem IV.8 (polynomial approximation for LIS). Given a length-\( n \) sequence \( A \) such that \( \text{lis}(A) = n\lambda \) where \( \lambda \in [1/n, 1] \) is unknown to the algorithm. There is an algorithm that runs in time \( O(\lambda^{-7}\sqrt{n}) \) and outputs a number \( \text{est} \) such that

\[ \Omega(\text{lis}(A)\lambda^3) \leq \text{est} \leq O(\text{lis}(A)) \]

with probability at least \( 1 - 1/poly(n) \).

We remark that one can turn Theorem IV.8 into an algorithm with running time \( O(n^{17/20}) \) by considering two cases separately. If \( \lambda < n^{-1/20} \) we sample the array with a rate of \( n^{-3/20} \) and compute the LIS for the sampled array. Otherwise, the running time of the algorithm is already bounded by \( O(n^{17/20}) \).

D. An \( O(n^\epsilon) \) Time Algorithm via Bootstrapping

Let us move a step backward and analyze the previous algorithm for obtaining an \( O(\lambda^3) \) approximate solution. We first divide the input array into \( \sqrt{n} \) subarrays of size \( \sqrt{n} \) and after constructing the pseudo-solutions, we sample \( O(\lambda^4) \) subarrays to estimate the size of the solution for pseudo-solutions. The reason we set the size of the subarrays to \( \sqrt{n} \) is that there is a trade-off between the first and the last steps of the algorithm. More precisely, if we have more than \( \sqrt{n} \) subarrays then the number of samples we draw in the beginning would exceed \( O(\lambda\sqrt{n}) \). On the other hand, having fewer than \( \sqrt{n} \) subarrays results in larger subarrays which would be costly in the last step.

If one favors the running time over the approximation factor, the following improvement can be applied to the algorithm: In the last step of the algorithm, instead of sampling the entire subarrays and computing \( \text{lis} \) for every
Algorithm 7 Recursive Estimate LIS with Oracle

1: procedure RECURSIVEESTIMATIONWITHORACLE(ORACLE, A, \(\lambda, \ell, r\))

2: \(\triangleright\) input: sequence A, parameter \(\lambda\), domain interval \([\ell, r]\)
3: \(\triangleright\) assume \(sa_1, sa_2, \ldots, sa_n\) are subarrays of A
4: \(\triangleright\) subroutine ORACLE approximate LIS for subarrays with approximation factor \(f(\lambda)\)
5: for \(i \in [\zeta]\) do
6: \(\triangleright\) \(cd_i \leftarrow\) CONSTRUCTCANDIDATEDOMAINS\((sa_i)\)
7: discard all the intervals which are not in \([\ell, r]\)
from \(cd_i\)
8: end for
9: \(\{ps_1, \ldots, ps_\zeta\} \leftarrow\) CONSTRUCTPSEUDOSOLUTIONS\((cd_1, \ldots, cd_\zeta)\)
10: \(\lambda_0 \leftarrow \left(\frac{\lambda}{2}\right)^4\)
11: \(p \leftarrow\) \(20\log^2 n\) \(\frac{1}{\lambda_0}\)
12: randomly sample each \(i \in [\zeta]\) with probability \(p\),
and put all the samples in a set \(Q\)
13: for \(j \in [t]\) do
14: \(c \leftarrow 0\)
15: for \(i \in W\) do
16: if \(\exists (\ell, r) \in ps_j\) and ORACLE\((sa_i, \lambda_0, \ell, r)\)
accepts then
17: \(c \leftarrow c + 1\)
18: end if
19: end for
20: if \(c \geq 3\lambda_0 p \zeta / 4\) then
21: return accept
22: end if
23: end for
24: return reject
25: end procedure

Lemma IV.9

Assume we partition the sequence into \(\zeta\) subarrays, where \(\zeta\) is polynomially related to the length of the input sequence. For parameter \(\lambda \in (0, 1)\), let ORACLE be a \(f(\lambda)\)-approximate LIS algorithm (with respect to a domain interval) with running time \(g(n, \lambda)\) and success probability \(1 - \exp(-\Omega(\log^2 n))\) where \(n\) is the length of the input sequence. Then Algorithm 7 using ORACLE as a subroutine is a \(f\left(\frac{\lambda^4}{256}\right)\)-approximate LIS algorithm with

\[
O\left(\lambda^{-4} g\left(\frac{n}{\zeta} \frac{\lambda^4}{256}\right) \log_O(1) n + \lambda^{-7} \zeta \log_O(1) n\right)
\]

running time and success probability \(1 - \exp(-\Omega(\log^2 n))\), where \(\zeta\) is the number of subarrays.

Proof: We first prove the correctness of the algorithm. Let \(A\) be a sequence of length \(n\), and \(sa_1, \ldots, sa_\zeta\) be the subarrays. Consider the case of \(\text{lis}^{[\ell,r]}(A) \geq \lambda_0 n\). By Corollary IV.2 and Lemma IV.4 with \(\epsilon = \delta = 1/10\), with probability \(1 - \exp(-\Omega(\zeta \lambda))\), there exists a pseudo-solution \(ps_j\) within interval \([\ell, r]\) satisfying

\[
q(ps_j) \geq \frac{\text{lis}^{[\ell,r]}(A)}{2t} \geq \frac{\text{lis}^{[\ell,r]}(A) \lambda e}{2k^2} \geq \frac{\text{lis}^{[\ell,r]}(A) \lambda e}{2 \cdot 20^2 \log^2 (1/\delta) / (x^2 e^4)} \geq \frac{\text{lis}^{[\ell,r]}(A) \lambda e}{8 \log^2 (1/\delta) / (x^2 e^4)} \geq \frac{\lambda^4}{800 \log^2 (1/\delta)} n \geq \frac{\lambda^4}{320^2} n.
\]

Let \(\alpha\) denote the number of subarrays \(sa_i\) such that \(\text{lis}^{[\ell,r]}(sa_i) \geq \lambda_0 n / \zeta\) where \([\ell, r]\) is the interval for subarray \(sa_i\) in \(ps_j\). We have

\[
\alpha \geq \frac{q(ps_j) - \lambda^4 n / 252}{n / \zeta} \geq \frac{\lambda^4 \zeta}{252}.
\]

By Chernoff bound, Step 14 to Step 22 of Algorithm 7 accepts on \(ps_j\) with probability at least \(1 - \exp(-\Omega(\log^2 n))\). Consider the case of

\[
\text{lis}^{[\ell,r]}(A) \leq f\left(\frac{\lambda^4}{252}\right) \frac{\lambda^4}{253} n.
\]

Then for any pseudo-solution \(ps_j\), we have

\[
q(ps_j) \leq f\left(\frac{\lambda^4}{252}\right) \frac{\lambda^4}{253} n.
\]

Let \(\beta\) denote the number of subarrays \(sa_i\) such that \(\text{lis}^{[\ell,r]}(sa_i) \geq f(\lambda^4 / 252) n / \zeta\) where \([\ell, r]\) is the interval

factor \(f(\lambda)\), then we can get a \(f\left(\frac{\lambda^4}{252}\right) \cdot \frac{\lambda^4}{253}\)-approximate LIS algorithm with better running time using the \(f(\lambda)\)-approximate algorithm as a subroutine.
for subarray $s_{a_i}$ in $p_{s_j}$. We have

$$\beta \leq \frac{q(p_{s_j})}{f(\lambda^4/2^{22}n)\zeta} \leq \frac{\lambda^4}{2^{23}3^\zeta}.$$  

By Chernoff bound, Step 14 to Step 22 of Algorithm 7 do not accept on $p_{s_j}$ with probability at least $1 - \exp(-\Omega(\log^2 n))$. By union bound, Algorithm 7 rejects with probability at least $1 - \exp(-\Omega(\log^2 n))$.

Hence, Algorithm 7 is a $(f\left(\frac{\lambda^4}{2^{32}}, \frac{\lambda^4}{2^{32}}\right))$-approximate algorithm for LIS of length $n$ with success probability at least $1 - \exp(-\Omega(\log^2 n))$.

Finally, we discuss the running time of Algorithm 7. By the definition of procedures CONSTRUCTCANDIDATEDOMAINS and CONSTRUCTPSEUDOSOLUTIONS, the running time of Step 5 to Step 9 of Algorithm 7 is $O(\lambda^{-9}\log(\zeta)n)$. By Lemma IV.3, $t = O(\lambda^{-3})$, and by Chernoff bound with probability $1 - \exp(-\Omega(\log^2 n))$ the size of $Q$ is at most $O(\lambda^{-4}\log^4 n)$. Hence, the running time of Step 12 to Step 24 of Algorithm 7 is $O(\lambda^{-7}\log(\zeta)\lambda^4/2^{22}\log^4 n)$.

By definition, we have the following basic fact about approximate ratio.

**Fact IV.10.** Let $f$ and $f'$ be two functions mapping $(0, 1)$ to $(0, 1)$ such that $f(\lambda) \geq f'(\lambda)$ for any $\lambda \in (0, 1)$. If there is a $f(\lambda)$-approximate LIS algorithm, then the algorithm is also $f'(\lambda)$-approximate.

Now we present algorithm to approximate LIS using $\tilde{O}(n^\kappa\text{poly}(\lambda^{-1}))$ space by applying the pseudo-solution construction-evaluation framework recursively. In particular, we use the same algorithm on subarrays as an oracle and apply Lemma IV.9 recursively to approximate the entire sequence with slightly worse approximation ratio (compared with approximation ratio of the oracle).

**Lemma IV.11.** Let $\kappa$ be a constant of $(0, 1)$ and $\lambda \in (0, 1)$.

![Figure 6: The flowchart of the $O_\lambda(n^\kappa)$ time algorithm is shown.](image)

**Algorithm 8 Recursive Estimate LIS**

1: procedure RECURRENTELIS($A, \lambda, \ell, r$) \textit{\triangleright} Lemma IV.11
2: \> input: sequence $A$, parameter $\lambda$, domain interval $[\ell, r]$
3: \> if the length of $A$ is greater than $n^{2\kappa}$ then
4: \> \> return \textsc{RecursiveLISWithOracle}(\textsc{RecursiveLIS}, $A, \lambda, \ell, r$) with $\zeta = n^\kappa$
5: \> else
6: \> \> for $i \in [n^\kappa]$ do
7: \> \> \> $c_{di_i} \leftarrow$ \textsc{ConstructCandidateDomains}($s_{a_i}$)
8: \> \> \> discard all the intervals which are not in $[\ell, r]$
9: \> \> \> from $c_{di_i}$
10: \> \> end for
11: \> \> \{ps_{1}, \ldots, ps_{\kappa}\} \leftarrow \textsc{ConstructPseudoSolutions}(c_{di_1}, \ldots, c_{di_n})$
12: \> \> if $\text{EvaluatePseudoSolutions}(ps_{1}, \ldots, ps_{\kappa}) \geq \lambda |A|$ then
13: \> \> \> return accept
14: \> \> else
15: \> \> \> return reject
16: \> \> end if
17: \> end if
18: end procedure

Algorithm 8 approximates LIS with approximation ratio

$$\frac{\lambda^{2.4^\kappa}}{256^{3.4^\kappa}}$$

and running time $O(n^{\kappa}\lambda^{-4^{O(1/\kappa)}}\log(\zeta)n)$ and success probability $1 - \exp(-\Omega(\log^3 n))$.

**Proof:** We first prove the correctness of the algorithm by induction. Without loss of generality, we assume $1/\kappa$ is an integer.
For $i \in \{2, 3, \ldots, 1/k\}$, denote

$$h_i(\lambda) = \frac{\lambda^2 \cdot d^{i+1} \cdot 4}{256 \cdot 4^{i+1} \cdot 3 \cdot 4^{i+1} \cdot 2 - 7}.$$ 

We show that if the length of the input sequence is $n^{k \cdot n}$ then Algorithm 8 is $h_i(\lambda)$-approximate.

If the length of input sequence is $n^{2k}$, then $h_2(\lambda) = \frac{\lambda^4}{2^3}$. By Corollary IV.2, Lemma IV.4, and Lemma IV.6, Algorithm 8 is $h_2(\lambda)$-approximate.

In the induction step, for an integer $2 \leq i < 1/k$, we assume Algorithm 8 is $h_i(\lambda)$-approximate for input instance of length $n^{i \cdot k}$. By Lemma IV.9, Algorithm 8 is $(\frac{h_i(\lambda)}{2^3}) \cdot \frac{\lambda^4}{2^3}$-approximate for input instance of length $n^{(i+1) \cdot k}$. Since

$$h_i(\lambda) \geq \frac{(2 \cdot d^{i+1} \cdot 4)}{256 \cdot 4^{i+1} \cdot 3 \cdot 4^{i+1} \cdot 2 - 7} \cdot \lambda^4,$$

by Fact IV.10, Algorithm 8 is $h_{i+1}(\lambda)$-approximate for input instance of length $n^{(i+1) \cdot k}$. Hence

$$h_{1/k}(\lambda) \geq \frac{\lambda^2 \cdot d^{(1/k)-1}}{256 \cdot 4^{(1/k)-1} \cdot 3 \cdot 4^{(1/k)-1} \cdot 2 - 7},$$

by Fact IV.10, Algorithm 8 is $(\frac{\lambda^2 \cdot d^{(1/k)-1}}{256 \cdot 4^{(1/k)-1} \cdot 3 \cdot 4^{(1/k)-1} \cdot 2 - 7})$-approximate for input instance of length $n^k$.

By Corollary IV.2, Lemma IV.4, Lemma IV.6 and Lemma IV.9, we have the desired running time. The success probability is obtained by same corollaries/lemmas and union bound.

Finally, by starting with $\lambda = 1$ and iteratively multiplying $\lambda$ by a $1/(1 + \epsilon)$ factor until a solution is found, we can approximate $\text{lis}(A)$ within an approximation factor of $\lambda^{O(1/\epsilon)}$.

Theorem IV.12. Let $\kappa$ be a constant of $(0, 1)$ and $\lambda \in (0, 1)$. There exists a $\tilde{O}(n^k \cdot \lambda^{O(1/\epsilon)})$ time algorithm for lis with approximation factor $\lambda^{O(1/\epsilon)}$ and success probability $1 - \exp(-\Omega(\log^2 n))$.

V. Probability and Graph Tools

In this section, we restate probability and graph tools that we use throughout this paper. All these theorems are proven in previous work.

Theorem V.1 (Chernoff Bounds). Let $X = \sum_{i=1}^n X_i$, where $X_i = 1$ with probability $p_i$, and $X_i = 0$ with probability $1 - p_i$, and all $X_i$ are independent. Let $\mu = \mathbb{E}[X] = \sum_{i=1}^n p_i$. Then

1. $\Pr[X \geq (1 + \delta)\mu] \leq \exp(-\delta^2 \mu / 3), \forall \delta > 0$;
2. $\Pr[X \leq (1 - \delta)\mu] \leq \exp(-\delta^2 \mu / 2), \forall \delta < 1$.

Theorem V.2 (Hoeffding bound). Let $X_1, \ldots, X_n$ denote $n$ independent bounded variables in $[a_i, b_i]$. Let $X = \sum_{i=1}^n X_i$, then we have

$$\Pr[|X - \mathbb{E}[X]| \geq \varepsilon] \leq 2\exp\left(-\frac{\varepsilon^2}{\sum_{i=1}^n (b_i - a_i)^2}\right).$$

Theorem V.3 (Blakley-Roy inequality, [27], see also Proposition 3.1 in [28]). Let $G$ denote a graph that has $n$ vertices and average degree $d$. The number of walks of length $k$ in graph $G$ is at least $n d^k$.

Theorem V.4 (Turán theorem for bipartite graphs, [29], see also [30]). For a graph $G$ the Turán number $\text{ex}(G, n)$ is the maximum number of edges that a graph on $n$ vertices can have without containing a copy of $G$. For any $s \leq t$, $\text{ex}(K_{s,t}, n) \leq \frac{1}{2}((t - 1)^{1/s}n^{2-1/s} + o(n^{2-1/s})$.

Theorem V.5 (Turán theorem for cliques [31]). Let $G$ be any graph with $n$ vertices, such that $G$ is $K_{r+1}$-free. Then the number of edges in $G$ is at most

$$\frac{n}{2} - (1 - \frac{1}{r}) \cdot \frac{n^2}{2}.$$ 

Corollary V.6 (of Theorem V.5). Let $G$ be any graph with $n$ vertices, such that $G$ has no independent set of size $r + 1$. Then the number of edges in $G$ is at least

$$\frac{n}{2} - (1 - \frac{1}{r}) \cdot \frac{n^2}{2} = \frac{nr + n^2}{2r}.$$ 

REFERENCES

[1] M. Boroujeni, S. Ehsani, M. Ghodsi, M. HajiAghayi, and S. Seddighin, “Approximating edit distance in truly sub-quadratic time: Quantum and mapreduce,” in Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 2018.

[2] D. Charkaborty, D. Das, E. Goldenberg, M. Koucky, and M. Saks, “Approximating edit distance within constant factor in truly sub-quadratic time,” in Proceedings of the Fifty-Ninth Annual IEEE Symposium on Foundations of Computer Science, 2018.

[3] G. M. Landau, E. W. Myers, and J. P. Schmidt, “Incremental string comparison,” SIAM Journal on Computing, vol. 27, no. 2, pp. 557–582, 1998.

[4] Z. Bar-Yossef, T. Jayram, K. Krauthgamer, and R. Kumar, “Approximating edit distance efficiently,” in Proceedings of the Forty-Fifth Annual IEEE Symposium on Foundations of Computer Science, 2004.

[5] T. Batu, F. Ergun, and C. Sahinalp, “Oblivious string embeddings and edit distance approximations,” in Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2006.
[6] A. Andoni and K. Onak, “Approximating edit distance in near-linear time,” in Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, 2009.

[7] A. Andoni, R. Krauthgamer, and K. Onak, “Polylogarithmic approximation for edit distance and the asymmetric query complexity,” in Proceedings of the Fifty-First IEEE Annual Symposium on Foundations of Computer Science, 2010.

[8] M. Saks and C. Seshadhri, “Estimating the longest increasing sequence in polylogarithmic time,” in Proceedings of the Fifty-First Annual IEEE Symposium on Foundations of Computer Science, 2010.

[9] A. Backurs and P. Indyk, “Edit distance cannot be computed in strongly subquadratic time (unless SETH is false),” in Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing, 2015.

[10] A. Abboud, A. Backurs, and V. V. Williams, “Tight hardness results for LCS and other sequence similarity measures,” in Proceedings of the Fifty-Sixth IEEE Annual Symposium on Foundations of Computer Science, 2015.

[11] A. Abboud, T. D. Hansen, V. V. Williams, and R. Williams, “Simulating branching programs with edit distance and friends,” in Proceedings of the Forth-Eighth Annual ACM SIGACT Symposium on Theory of Computing, 2016.

[12] A. Abboud and A. Backurs, “Towards hardness of approximation for polynomial time problems,” in Proceedings of the Eighth Innovations in Theoretical Computer Science Conference, 2017.

[13] A. Abboud and A. Rubinstein, “Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds,” in Proceedings of the Ninth Innovations in Theoretical Computer Science Conference, 2018.

[14] L. Chen, S. Goldwasser, K. Lyu, G. N. Rothblum, and A. Rubinstein, “Fine-grained complexity meets IP=PSpace,” Proceedings of the Thirtieth Annual ACM-SIAM symposium on Discrete Algorithm, 2019.

[15] M. Hajiaghayi, M. Seddighin, S. Seddighin, and X. Sun, “Approximating LCS in linear time: Beating the barrier,” in Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, 2019.

[16] M. Boroujeni and S. Seddighin, “Improved mpc algorithms for edit distance and ulam distance,” in SPAA, 2019.

[17] M. Hajiaghayi, S. Seddighin, and X. Sun, “Massively parallel approximation algorithms for edit distance and longest common subsequence,” in SODA. Society for Industrial and Applied Mathematics, 2019.

[18] W. J. Masek and M. S. Paterson, “A faster algorithm computing string edit distances,” Journal of Computer and System Sciences, vol. 20, no. 1, pp. 18–31, 1980.

[19] A. Rubinstein and Z. Song, “Reducing approximate longest common subsequence to approximate edit distance,” in SODA. https://arxiv.org/pdf/1904.05451, 2020.

[20] C. Schensted, “Longest increasing and decreasing subsequences,” Canadian Journal of Mathematics, vol. 13, pp. 179–191, 1961.

[21] M. L. Fredman, “On computing the length of longest increasing subsequences,” Discrete Mathematics, vol. 11, no. 1, pp. 29–35, 1975.

[22] Y. Dodis, O. Goldreich, E. Lehman, S. Raskhodnikova, D. Ron, and A. Samorodnitsky, “Improved testing algorithms for monotonicity,” in Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques. Springer, 1999, pp. 97–108.

[23] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan, “Spot-checkers,” Journal of Computer and System Sciences, vol. 60, no. 3, pp. 717–751, 2000.

[24] E. Fischer, “The art of uninformed decisions: A primer to property testing,” Current Trends in Theoretical Computer Science: The Challenge of the New Century, vol. 1, pp. 229–264, 2004.

[25] N. Ailon, B. Chazelle, S. Comandur, and D. Liu, “Estimating the distance to a monotone function,” Random Structures & Algorithms, vol. 31, no. 3, pp. 371–383, 2007.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. MIT press, 2009.

[27] G. R. Blakley and P. Roy, “A hölder type inequality for symmetric matrices with nonnegative entries,” Proceedings of the American Mathematical Society, vol. 16, no. 6, pp. 1244–1245, 1965.

[28] P. Keevash, B. Sudakov, and J. Verstraëte, “On a conjecture of erdős and simonovits: Even cycles,” Combinatorica, vol. 33, no. 6, pp. 699–732, 2013.

[29] T. Kovári, V. Sós, and P. Turán, “On a problem of k. zarankiewicz,” in Colloquium Mathematicum, vol. 1, no. 3, 1954, pp. 50–57.

[30] P. V. Blagojević, B. Bukh, and R. Karasev, “Turán numbers for $k_s,t$-free graphs: Topological obstructions and algebraic constructions,” Israel Journal of Mathematics, vol. 197, no. 1, pp. 199–214, 2013.

[31] P. Turán, “On an external problem in graph theory,” Mat. Lapok, vol. 48, pp. 436–452, 1941.