Electronic Colloquium on Computational Complexity, Report No. 56 (2016)

On the Fine Grained Complexity of Polynomial Time Problems Given Correlated Instances*

Shafi Goldwasser Dhiraj Holden

April 8, 2016

Abstract

We set out to study the impact of having access to correlated instances on the fine grained complexity of polynomial time problems, which have notoriously resisted improvement.

In particular, we show how to use a logarithmic number of auxiliary correlated instances to obtain $o(n^2)$ time algorithms for the longest common subsequence(LCS) problem and the minimum edit distance (EDIT) problem. For the problem of longest common subsequence of k sequences we show an $O(nk \log n)$ time algorithm with access to a logarithmic number of auxiliary correlated instances. Our results hold for a worst case choice of the primary instance whereas the auxiliary correlated instances are chosen according to a natural correlation model between instances.

Previously, it has been shown that any improvement over $O(n^2)$ for the worst case complexity of the longest common subsequence and minimum edit distance problem would imply radically improved algorithms than currently known for a host of long studied polynomial time problems such as finding a pair of orthogonal vectors as well as imply that the *Strong Exponential Time Hypothesis* is false. The best known algorithm for the multiple sequence longest common subsequence problem is a variant of dynamic programming which requires $O(n^k)$ worst case runtime.

We note that sequence alignment is often used in identifying conserved sequence regions across a group of sequences of DNA, RNA or proteins hypothesized to be evolutionarily related, or as aid in establishing evolutionary relationships by constructing phylogenetic trees, but is notoriously computationally prohibitive for $k > 3$. An intriguing question, which served as an inspiration for our work, is to find correlation models which coincide with evolutionary models and other relationships and for which our approach to multiple sequence alignment gives provable guarantees.

1 Introduction

An intriguing line of research has been launched in the last couple of years classifying the complexity of a host of polynomial-time computations in graphs, string matching, computational geometry and more. This body of work shows hardness of breaking known concrete polynomial runtime bounds based on complexity-theoretic conjectures. Moreover, fine-grained reductions are used to show equivalences between breaking the best known upper bounds for various seemingly-unrelated problems. Some prominent examples of problems that have been studied include edit distance on strings and all-pairs shortest paths on graphs.

*This work was supported by an Akamai Presidential Fellowship, NSF MACS - CNS-1413920 , and SIMONS Investigator award Agreement Dated 6-5-12

ISSN 1433-8092

For example, Backurs and Indyk [5] have shown that if the *edit distance* between two strings (EDIT) – the minimum number of insertions, deletions or substitutions of symbols needed to transform one string into another – can be computed in time $O(n^{2-\delta})$ for some constant $\delta > 0$, then the satisfiability of conjunctive normal form formulas with n variables and m clauses can be solved in time $poly(m)2^{\epsilon n}$ for a constant $\epsilon > 0$. This would violate the Strong Exponential Time Hypothesis (SETH). A $O(n^{2-\delta})$ algorithm for EDIT would also imply a sub-quadratic algorithm for the orthogonal vector problem and a host of others.

Another example in work by Abboud, Backurs and Williams [1] addresses (among other problems) the *longest common subsequence* (LCS) problem – the length of the longest non-consecutive string common to two sequences. A well known dynamic programming algorithm solves this task in $O(n^2)$ time and [1] indicates that this may be the best possible, up to logarithmic factors. In particular, [1] shows that an $O(n^{2-\epsilon})$ algorithm for the LCS of two sequences of length n over a constant size alphabet would refute SETH and show a sub-quadratic algorithm for the orthogonal vectors problem and a host of others. For finding the longest subsequence common to k input sequences, it is similarly argued that achieving $O(n^{k-\epsilon})$ for any $\epsilon > 0$ is unlikely in [1].

In a different vein, a recent work of Dinur, Goldwasser, and Lin [6] ask whether the computational complexity of a problem can radically change if one had access to auxiliary "correlated instances" to the primary instance of the problem to be solved. They argue that access to such instances often arises in natural settings and thus the question is not whether they exist but how and whether it is possible to take advantage of them to gain significant computational game.

A telling example is the integer factoring problem for which the best known algorithm for factoring runs in sub-exponential time, and yet given two (highly) correlated instances can be solved in polynomial time. In particular, for integers $N = pq$ and $M = p'q$ of length n, by computing $gcd(N.M) = q$, the factorization of N can be computed in $O(n^2)$. Surprisingly, variations on such correlations come up when poorly designed pseudo random number generators are used to choose the composite numbers within the RSA encryption algorithm, whose secret factorization underlies the security [10].

Dinur et al. [6] study how access to correlated instances affect constraint satisfaction problems (CSP) which are NP-Hard to solve in the worst case and for which no polynomial time algorithm is known for the average case distributions considered. The question of which types of correlations to consider is obviously the key question. For example, one cannot hope for a significant speedup if the correlated instances can be generated from the primary instance, in time which is less than the fastest known algorithm for the problem at hand. Indeed [6] show that a logarithmic number of correlated instances may significantly speedup solving the CSP from intractable to tractable. The framework which [6] suggest is to explicitly consider distributions over correlated instances where correlations are directly defined between the *solutions* of the primary and auxiliary instance (which in turn imply a correlated between than the instances). They abstract the notion of a generating process G for a search problem P which is initialized with I, a (possibly worst case) instance of P, an underlying solution S for I, and a parameter k , and outputs instances $\{I_j\}_{j=1,...,k}$ chosen by a probabilistic process applied to (I, S). The algorithm designer is then given the tuple of $k + 1$ instances $I, I_1, ..., I_k$ and is tasked with finding a solution to I.

Whereas the work of [6] asked whether access to a polynomial number of auxiliary (but correlated) instances can enable us to find a polynomial time solution to an otherwise intractable problem, In this paper we turn our attention to tractable problems (in P) and ask how their finer grained complexity changes with auxiliary access to correlated instances .

1.1 New Work

In this paper, we set out to study the impact of access to correlated instances on the fine grained complexity of well known polynomial time problems which have consistently resisted improvement. Namely, given problems of known (and conjectured) complexity $O(n^c)$ for some $c > 0$, we seek to improve their complexity to $O(n^{c-\epsilon})$ for some ϵ or even $O(npoly(logn))$ given natural (and hopefully available) auxiliary correlated instances.

The immediate question which comes to mind is which *P-time problems* to study and which *correlations* would be interesting and useful to consider. We chose to consider three problems inspired by the work of [5, 1]: the *Longest Common Subsequence* (LCS) problem where an instance is composed of two strings (x, y) over an alphabet Σ and one seeks the maximum length not necessarily consecutive substring common to both; the k *Longest Common Subsequence* (k-LCS) problem where an instance is composed of k strings over alphabet Σ and one seeks the maximum length not necessarily consecutive substring common to all k substrings; and the *Minimum Edit Distance* (EDIT) where an instance is composed of two strings over alphabet Σ one seeks the minimum number of insert, delete and replace operations to obtain one string from the other. We chose these problems due to the mounting evidence produced in the work of [5, 1] and others that the known (stand alone) worst case complexity of these problems represent the complexity of many other problems shown equivalent via fine-grained reductions.

The choices of which auxiliary instances (and which correlations) to consider for the LCS, k-LCS and $EDIT$ problem are problem specific. Indeed, one may argue that this will likely always be the case, as our hope is to consider correlations which would come up via *natural* generating processes of problem instances. For example, In bioinformatics, sequence alignment is a way of arranging the sequences of DNA, RNA, or proteins to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Although we expect that only certain parts of the genome will be in common between say multiple organisms, these parts will have a much higher rate of matching than random chance. Thus, which correlations between sequences one may expect are dictated by evolutionary process which would be highly problem specific.

In particular, for the LCS problem on a worst case primary instance string pair (x, y) with longest common sub-sequence at set of locations A we choose to consider randomly chosen auxiliary instances (x^j, y^j) for which with probability much higher than at random also have a common subsequence at locations A. For the k-LCS problem, on a worst case primary instance k-tuple with longest common sub-sequence at locations A, we will consider randomly chosen auxiliary k-tuple instances which each also have a common subsequence at locations A. For the EDIT problem, on a worst case primary instance string (x, y) we will consider random auxiliary instances (x^j, y^j) where y^j is obtained from x^j by applying the same sequence of edit operations which were applied to x to obtain y. Similar results follow when the x^j's are chosen as a random perturbation of x.

We will show new algorithms which achieve significant runtime complexity improvement for all three problems LCS, k-LCS and $EDIT$ with access to $O(\log n)$ additional sequences correlated as above. We state our results and correlation models in detail below.

A word of caution: one should be careful not to consider correlations which trivialize the problem. An example of such trivializations would be access to "too many" auxiliary instances. Whereas in [6] "too many" would correspond to more than a polynomial number of instances, in the current work it would be more than a linear number of auxiliary instances. Indeed, in all the cases we consider in this paper, a logarithmic number of additional instances suffice to get a marked improvement, as was the case of [6]. Another forbidden trivialization, avoided here, would be access to auxiliary "easy" instances for which both the problem is easy to solve and the correlation enables

trivial extraction of the answer for the primary instance.

1.1.1 The LCS Results

Let Pri denote any distribution over LCS instances. Let $(x, y) \in Pri$ denote a pair of n bit strings over alphabet Σ . Let $\vec{a} = (a_1, ...a_m)$ and $\vec{b} = (b_1, ..., b_m)$ denote two sequences in $[1, n]$ such that $a_1 < a_2... < a_m$ and $b_1 < b_2... < b_m$ corresponding to locations in string x (and y respectively) in which a primary solution to $LCS(x, y)$ resides. Namely, $x_{a_j} = y_{b_j}$ for all j, and m maximizes the length of such sequences. The l auxiliary instances $(x^1, y^1), ...(x^l, y^l)$ are chosen at random in Σ^{2n} such that for each (x^j, y^j) the sequence residing in the locations $\vec{a}$ of x^j matches the sequence in locations $\vec{b}$ of y^j. Namely, $x^j_{a_i} = y^j_{b_i}$ for all j, for $1 \leq i \leq m$.

Note that we selected the primary instance in a worst case manner, whereas the auxiliary instances are selected at random, and then modified so that in each auxiliary pair x^j and y^j share a common subsequence at the same locations that primary x and y contain the longest common subsequence. We note that the actual substring at the $\vec{a}$ locations of x^j are not required (and are unlikely to match) the substrings at the $\vec{a}$ locations of the primary instance y.Thus, a solution for LCS for any of auxiliary pairs, does not yield a solution for the primary pair on its own. Indeed, we need $l = O(logn)$ auxiliary solutions to be able to extract the solution to the primary instance.

A key technical insight is that it is now possible to construct a new instance of the LCS problem over an extended alphabet where each character corresponds to a vector in Σ^{l+1} and which now has enough structure (as opposed to the primary worst case instance) to be solved in sub-quadratic time. Essentially, each character in a position of the new LCS instance is determined by the characters in the original primary and auxiliary instances at the same position. When the number of auxiliary instances is large enough and the distribution is uniform, it will mean that any matching "character" in the new instance pair corresponds to a part of the longest common subsequence of the original primary instance with high probability, making the longest common subsequence problem solvable in sub-quadratic time by an algorithm due to James and Szymanski [11]. The latter show an algorithm for LCS with running time of $O((r + n)logn)$, where r is the total number of ordered pairs of positions at which the two sequences match. Thus in the worst case the algorithm of [11] has a running time of $O(n^2 \log n)$. However, for special cases when most positions of one sequence match relatively few positions in the other sequence, a better running time can be expected. In the our setting, we expect the new instance to exhibit with high probability behavior which will solicit an improved running time.

To (semi) formally state our first result (and for ease of comparison with the one following), we define the inputs to our algorithm via the following generating process $GenLCS1$:

> On input n, l where n is the length of the primary instance, l the number of auxiliary instances, do the following:
>
> 1. Choose a worst case primary instance $(x, y) \in \Sigma^2$ with solution $(a_1, ..., a_m)$ and $(b_1, ..., b_m)$ such that $x_{a_i} = y_{b_i}$ for all i.
> 2. For $j = 1, ..., l$, choose an auxiliary instance (x^j, y^j) at random from Σ^{2n}
> 3. For $j = 1, ..., l$ and $i = 1, ..., m$, set the a_i-th bit of x^j = the b_i-th bit of y^j
>
> Output $(x, y), (x^1, y^1), ..., (x^l, y^l)$

Informal Theorem 1 (Solve LCS with O(log n) instances.) There exists an algorithm which on input pairs $((x, y), (x^1, y^1), ..., (x^l, y^l)$ generated by $GenLCS1(n, l)$ solves the $LCS(x, y)$ problem and runs in expected time $O(n^{2-\varepsilon} \log n)$ for $l = \varepsilon \log n$, $0 < \varepsilon \leq 1$ where the expectation is taken

over the choices of auxiliary inputs in $GenLCS1(n, l)$.

We next ask what happens if we relax the restriction that the auxiliary instances all contain a common subsequence at the locations of the longest common subsequence of the primary original instance. Instead, say that each pair of the longest common subsequence is present in the auxiliary instances with probability slightly than $\frac{1}{2}$, say $\frac{1}{2} + \epsilon$. Using considerably more complex algorithms and in particular techniques from locality sensitive hashing we show the following.

As before, to (semi) formally state our results, we define the inputs to our algorithm via the following generating process $GenLCS2_\varepsilon$

> On input n, l where n is the length of the primary instance, l the number of auxiliary instances, do the following:
>
> 1. Choose a worst case primary $(x, y) \in \Sigma^2$ with solution $(a_1, ..., a_m)$ and $(b_1, ..., b_m)$ such that $x_{a_i} = y_{b_i}$ for all i and $a_1 < ... < a_m$, $b_1 < ... < b_m$
> 2. For $j = 1, ..., l$, choose an auxiliary instance (x^j, y^j) at random from Σ^{2n}
> 3. For $j = 1, ..., l$ and $i = 1, ..., m$, set a_i-th bit of x^j = the b_ith bit of y^j with probability ε and leave it unchanged with probability $1 - \varepsilon$.
>
> Output $(x, y), (x^1, y^1), ..., (x^l, y^l)$

Informal Theorem 2 (Solve LCS with weaker correlation.) Let $\varepsilon \in [0, \frac{1}{2}]$. There exists an algorithm which on a tuple of input instances $(x, y), (x^1, y^1), ..., (x^l, y^l)$ drawn from $GenLCS2_\varepsilon(n, l)$, outputs $LCS(x, y)$ with probability at least $2/3$ (where the probability is taken over the choices of auxiliary inputs in $GenLCS2_\varepsilon(n, l)$), and runs in time $O(ln^{1+d})$ for $d = \min\left(\frac{\log \frac{2}{1+2\epsilon-2\delta}}{\log \frac{2}{1+2\delta}}, 1 - O(\frac{1}{\frac{l}{\log n}\log^2 \frac{l}{\log n}})\right)$ and $\delta = \sqrt{\frac{9 \log n}{l}}$.

For $l = \Omega(\log n)$, $d < 1$, which means that this algorithm runs in subquadratic time.

1.1.2 The k-LCS Results

The previous section showed how to use correlated instances to get faster algorithms for finding the longest subsequence common to two input strings. We now ask if these results can be extended to computing the longest common subsequence of k input sequences where $k > 2$. It is known that being able to compute the longest common subsequence of k sequences in $n^{k-\varepsilon}$ time would give faster algorithms for NC-SAT and prove circuit lower bounds [2].

We will denote the primary instance as $(x[1], ..., x[k])$, consisting of k strings each in Σ^n, We will denote the primary solution to $kLCS(x[1], ..., x[k])$ by $(a[1], ..., a[k])$ where $a[i] = (a[i]_1, ..., a[i]_m)$ are a sequence of m locations in $x[i]$ such that $x[1]_{a[1]_i} = x[2]_{a[2]_i} = ... = x[k]_{a[k]_i}$ for all $i \in \{1, ..., m\}$ where m is maximal. The auxiliary instances of kLCS will be uniformly chosen subject to the constraint that they each have a random common substring at the location of the longest common substring of the primary instance. Using $l = O(\log n)$ extra instances will enable us to design an algorithm which runs in $O(kn \log n)$ time.

To determine the kLCS of a worst case primary instance, given the auxiliary instances, we will need a different algorithm then the one designed for the case of two sequences. We begin from the basic idea for the LCS algorithm on two input strings. Recall, that the algorithm constructed a new instance over a larger alphabet whose characters are Σ^{l+1} vectors, for which any matching character were part of the longest common subsequence with high probability. Note that the existence of

these pairs does not depend on k. This suggests the following idea: For strings 2 through k, put each character from the larger alphabet, which corresponds to the values of the primary and auxiliary instances at one location, into a bucket along with its index, where the buckets are indexed by vectors of length $l+1$. Then, for each vector of values in the primary and auxiliary instances of string 1, check if there is a vector in string 2 that is equal to the vector in string 1. These locations correspond to the locations of the longest common subsequence in string 1. To find the locations of the longest common subsequence in the other strings, we look for the vectors in string 1 that are part of the longest common subsequence in the other strings using the buckets.

To state the result (semi) formally we consider the following generating process $GenLCS[k]$:

> On input n, l, k where n is the length of the primary instance which is composed of k strings and l the number of auxiliary instances:
>
> 1. Choose a worst case primary instance $(x[1]...x[k])$ be k-tuple of n long strings over alphabet Σ with solution $a[1], ..., a[k]$ where $a[i]$ is an m long sequence of locations and $a[1]_i < ... < a[k]_i$ for all $i = 1, ..., m$
> 2. For $j = 1, ..., l$, choose auxiliary instance $(x[1]^{(1)}, ...x[k]^{(1)}), ..., (x[1]^{(l)}, ..., x[k]^{(l)})$ at random from Σ^{nkl}
> 3. for every $j = 1, ...l$, modify the locations $a[2], .., a[k]$ of $x[2]^{(j)}, ..., x[k]^{(j)}$ (respectively) to contain the subsequence in locations $a[1]$ of $x[1]^{(j)}$. (This corresponds to choosing the one random subsequence to use at locations $a[1], a[2], ..., a[k]$ per each auxiliary $x[1]^{(j)}, ..., x[k]^{(j)}$)
>
> Output $(x[1], x[2], ..., x[k]), (x[1]^1, x[2]^1, ..., x[k]^1), ..., (x[1]^l, x[2]^l, ..., x[k]^l)$

Informal Theorem 3 (Solve k-LCS exactly with $O(\log n)$ instances.) There exists an algorithm A which on inputs generated by $GenLCS[k](n, l)$ solves the $kLCS(x[1], ..., x[k])$ problem and runs in expected time $O(nk \log n)$ where $l = O(\log n)$ where the expectation is taken over the choices of auxiliary inputs in $GenLCS[k](n, l)$.

1.1.3 The EDIT Result

In the *Minimum Edit Distance* (EDIT) an instance is composed of two strings (x, y) over alphabet Σ. A solution provides the minimum number of insert, delete and replace operations to obtain one string from the other.

To state the EDIT results precisely, consider the following generating process. Let $\pi_1, \pi_2, ..., \pi_k$ be the minimum sequence of edits needed to transform x to y. The auxiliary instances are random pairs of n-bit strings whose minimum edit sequence are the same as for the primary instance. Namely, for auxiliary pair $x^{(1)}, y^{(1)}$, $y^{(1)}$ is obtained from $x^{(1)}$ using the same edit sequence which transformed x to y. Somewhat more generally, consider a worst case instance (x, y) and auxiliary instances $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ...$, with $x^{(i)} = x$ with each character changed with probability $\epsilon \in [0, 1]$ and $y^{(i)} = \pi_k(\pi_{k-1}(...(\pi_1(x^{(i)}))...))$. This is the distribution on which our algorithms perform.

Informal Theorem 4(Solve EDIT exactly with $O(\log n)$ instances.) There exists an algorithm A which on inputs generated as above solves the $EDIT(x, y)$ for a worst case primary instance and auxiliary instances as above such that on $O(\log n)$ auxiliary instances , the algorithm computes $EDIT(x, y)$ with high probability and runs in expected time $O(n \log n)$.

1.2 Smoothed Analysis and Correlated Instances for Edit Distance

The celebrated work on smooth analysis by Spielman and Teng[14] introduced a new measure on the complexity of algorithms, first illustrated via the smoothed analysis of the Simplex algorithm in the realm of real valued inputs. The smooth analysis measure for an input x is the expected behavior of the algorithm on correlated inputs which are the result of subjecting x to perturbations (e.g. flip its bits with a certain probability). Spielman and Teng [13] observe that in the discrete input domain, perturbations of the input should probably be restricted to those which preserve the most significant aspect input with respect to a given situation. To address this, they define property-preserving perturbations to inputs and relate this measure to property testing work [9] and the heuristics of Feige and Kilian [7] for finding cliques on semi-random graphs with planted cliques.

Indeed, Andoni and Krauthgamer [4] embarked on the study of the smoothed complexity of the edit distance problem as follows. Given two adversarially chosen binary input strings with a common subsequence A , they develop algorithms to approximate the edit distance of those input strings which result from an independent perturbation (with some fixed probability p) of each character of the original input strings with one restriction: the same perturbation is applied in both strings to the locations of the common subsequence A. They then show constant factor approximation algorithms which runs in linear time (or even sub-linear time assuming the edit distance is not too small) for such perturbed instances.

We remark that the approach of smoothed analysis in general and the work of [4] in particular, differs from our model and results in several aspects. First, we solve the edit distance problem (as well as LCS and k-LCS) on the primary worst case input pair itself with high probably rather than on a perturbed instance. Second, in contrast to smooth analysis our goal is not an "analysis" of when the problem becomes easier (or harder) but rather the development of algorithms which take advantage of (and when) extra correlated information is available in addition to an original input: namely, we assume that the correlated instances are received as an additional input that can help to solve the primary instance rather than aim to solve the correlated instances. Lastly, the correlation we consider are not between instances (per se) but rather between the solutions to the primary and auxiliary instances.

Having said that, there is an interesting interplay between the two explorations of smoothed analysis and improved algorithmics by having access to correlated instances . That is, one may consider any distribution induced by semi-random models over problem instances both as distributions for which 'smoothed complexity" can be studied, or as distributions over the auxiliary correlated instances which are available for improving algorithms.

One of the goals of research in the design and analysis of algorithms is to develop algorithms which work well in practice taking into account all available data. We have demonstrated that access to multiple problem instances with correlated solutions in the domain of sequence alignment and edit distance, can change the complexity of problems significantly. We believe that this research direction should be explored further, for other problems and algorithms.

2 Preliminaries

We denote the ith character of a string x over an alphabet Σ by x_i.

Definition 1. *A longest common subsequence of two strings $x, y \in \Sigma^*$ ($LCS(x, y)$) of length m is a largest set of pairs $(a_1, b_1), ..., (a_m, b_m)$ such that $1 \leq a_1 < a_2 < ... < a_m \leq n$, $1 \leq b_1 < b_2 < ... < b_m \leq n$, and $x_{a_i} = y_{b_i}$ for $i \in \{1, ..., m\}$.*

Definition 2. *A longest common subsequence of k strings (*$LCS(x[1], x[2], ..., x[k])$*) of strings* $x[1], x[2], x[3], ..., x[k]$ *is a largest set of tuples* $(a[1]_1, a[2]_1, ..., a[k]_1), ..., (a[1]_m, a[2]_m, ..., a[k]_m)$ *with* $1 \leq a[j]_1 < a[j]_2 < ... < a[j]_m \leq n$ *for all* $j \in \{1, ..., k\}$ *and* $x[1]_{a[1]_i} = x[2]_{a[2]_i} = ... = x[k]_{a[k]_i}$ *for all* $i \in \{1, ..., m\}$

Definition 3. *The edit distance (EDIT) between two strings* x, y *is the minimum number of insertions, deletions, and character substitutions it takes to get from* x *to* y*.* $EDIT(x, y) = LEV_{x,y}(|x|, |y|)$*, where*

$$LEV_{x,y}(i,j) = \begin{cases} \max i, j & \text{if } \min i, j = 0 \\ \min \begin{cases} LEV_{x,y}(i, j-1) \\ LEV_{x,y}(i-1, j) \\ LEV_{x,y}(i-1, j-1) + \mathbb{1}_{a_i \neq b_j} \end{cases} \end{cases}$$

and $\mathbb{1}_{a_i \neq b_j} = \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{if } a_i = b_j \end{cases}$.

Definition 4. *The generating process* $GenLCS_{Pri,Aux}(n, l)$ *draws* x, y *of length* n *from* Pri *with* $LCS(x, y) = \{(a_1, b_1), (a_2, b_2), ..., (a_m, b_m)\}$ *and outputs* (x, y) *and* l *pairs of strings* $(x^{(i)}, y^{(i)})$ *such that* $x^{(i)}$ *and* $y^{(i)}$ *are drawn from* Aux *conditioned on* $x^{(i)}_{a_j} = y^{(i)}_{b_j}$ *for all* $j \in \{1, ..., m\}$*.*

Definition 5. *The generating process* $GenLCS[k]_{Pri,Aux}(n, l)$ *draws* $x[1], x[2], ..., x[k]$ *of length* n *from* Pri *with* $LCS(x[1], x[2], ..., x[k]) = (a[1]_1, a[2]_1, ..., a_[k]_1), (a[1]_2, a[2]_2, ..., a[k]_2), ..., (a[1]_m, a[2]_m, ..., a[k]_m)$ *and outputs strings* $x[1], x[2], ..., x[k]$ *and* l *tuples of strings* $(x[1]^{(i)}, x[2]^{(i)}, ..., x[k]^{(i)})$ *such that the* $x[s]^{(i)}$ *are drawn from* Aux *conditioned on* $x[1]^{(i)}_{a[1]_j} = x[2]^{(i)}_{a[2]_j} = ... = x[k]^{(i)}_{a[k]_j}$ *for all* $j \in \{1, ..., m\}$*.*

Definition 6. *The generating process* $Genedit_{Pri,Aux,\varepsilon}(n, l)$ *draws* x, y *of length* n *and a sequence of character insertions, deletions, and substitutions* $\pi_1, \pi_2, ..., \pi_k$ *that is the minimum sequence of edits needed to transform* x *to* y *from* Pri*. The output is* $x, y, x^{(1)}, y^{(1)}, x^{(2)}, y^{(2)}, ..., x^{(l)}, y^{(l)}$*, with* $x^{(i)}$ *being* x *with each character changed with probability* ϵ *and* $y^{(i)} = \pi_k(\pi_{k-1}(...(\pi_1(x^{(i)}))...))$*.*

Definition 7. *The generating process* $GenLCS_{\varepsilon,Pri,Aux}(n, l)$ *draws* (x, y) *from* Pri *with* $LCS(x, y) = \{(a_1, b_1), (a_2, b_2), ..., (a_m, b_m)\}$ *and draws* $x^{(i)}, y^{(i)}$ *from* Aux *and outputs* (x, y) *and* l *pairs* $(x^{(i)}, y'^{(i)})$ *where* $y'^{(i)}$ *is* $y^{(i)}$ *with* $y'^{(i)}_{b_j}$ *set to* $x^{(i)}_{a_j}$ *with probability* ε *and otherwise left alone.*

3 Results

3.1 Longest Common Subsequence

How can we take advantage of correlations to find the longest common subsequence faster? Suppose that we are given multiple pairs of strings where the longest common subsequence of the primary instance is also a common subsequence of the auxiliary instances. The problem of recovering the longest common subsequence given these instances is equivalent to computing the longest common subsequence of two strings over a larger alphabet, where each character in the string is determined by the characters in the primary and auxiliary instances at the same position. When the number of auxiliary instances is large enough, and the distribution is uniform, this means that each 'character' in the first string over this larger alphabet appears a few times in that string, making the longest common subsequence problem solvable in subquadratic time [11].

Definition 8. *The instance column $(x)_i$ is defined as follows. Suppose we have strings x, y which are primary instances, which we will refer to $x^{(0)}$ and $y^{(0)}$, and auxiliary instances $(x^{(1)}, y^{(1)}), ..., (x^{(l)}, y^{(l)})$. Then, we denote $(x_i^{(0)}, x_i^{(1)}, ..., x^{(n)}) = (x)_i$, and similarly for $(y)_i$. We then denote the string $(x)_1(x)_2...(x)_n$ as $(\mathbf{x})$.*

Claim 1. *Suppose we have x, y with $LCS(x, y) = (a_1, b_1), (a_2, b_2), ..., (a_m, b_m)$ and $x^{(1)}, x^{(2)}, ..., x^{(l)}, y^{(1)}, y^{(2)}, ..., y^{(l)}$ drawn from generating process $GenLCS_{Pri,Aux}(n, l)$ with Pri being a worst-case distribution and Aux being the uniform distribution [1]. If $l > 2\log n + 3$, then the probability that there exists a pair (i, j) not in the longest common subsequence with $(x)_i = (y)_j$ is less than 1/8. Also, the expected number of i, j such that $(x)_i = (y)_j$ is equal to $LCS(x, y) + r/2^l$, where r is the number of i and j such that $x_i = y_j$ and (i, j) is not a pair in the longest common subsequence.*

Proof. If $(x)_i = (y)_j$, then $x_i = y_j$, $x_i^{(1)} = y_i^{(1)}$,...,$x_i^{(l)} = y_i^{(l)}$. If (i, j) is in the longest common subsequence, the probability of this happening is 1, and if (i, j) are not in the longest common subsequence, if $x_i = y_j$ the probability of this happening is $1/2^l$ because the auxiliary instances are drawn independently. Therefore, if $l > 2\log n + 3$, $1/2^l < 1/8n^2$ and by a union bound, the probability that there exists (i, j) not in the longest common subsequence such that $(x)_i = (y)_j$ is less than 1/8. In addition, the expected number of (i, j) such that $(x)_i = (y)_j$ is $LCS(x, y) + r/2^l$. □

Consider $(x)_i$ and $(y)_j$ to be characters of the alphabet Σ^{l+1}. Then $(x)_1(x)_2...(x)_n$ and $(y)_1(y)_2...(y)_n$ are strings over this alphabet. Using the algorithm of [11], we can find the longest common subsequence of $(\mathbf{x})$ and $(\mathbf{y})$.

In FIND-LCS, the location THRESH[k] is the position in $(\mathbf{y})$ of the longest common subsequence of length k so far, MATCHLIST[i] is the list of locations j such that $(x)_i = (y)_j$, LINK[k] has the kth pair in the longest common subsequence of length k so far and a pointer to LINK[$k-1$], and PTR is used to go through the longest common subsequence at the end.

Lemma 1. FIND-LCS *works and runs in expected time $O(n^{2-\varepsilon}\log n)$ for $l = \varepsilon \log n$, $0 < \varepsilon \leq 1$.*

Proof. First we will show that the algorithm works by showing any longest common subsequence of $(\mathbf{x})$ and $(\mathbf{y})$ is also a longest common subsequence of x and y, and vice versa. For the forward direction, by the way the auxiliary instances are generated the longest common subsequence of x and y is a common subsequence of $(\mathbf{x})$ and $(\mathbf{y})$. In the reverse direction, any common subsequence of $(\mathbf{x})$ and $(\mathbf{y})$ must also be a common subsequence of x and y because of the fact that $(x)_i$ has $x_i^{(0)} = x_i$ as its first element. This means that there cannot be a common sequence of $(\mathbf{x})$ and $(\mathbf{y})$ longer than the longest common subsequence of x and y. To prove the expected running time, we use Claim 1. The expected number of pairs i, j such that $(x)_i = (y)_j$ is equal to $LCS(x, y) + r/2^l$, and $2^l = n^\varepsilon$, and r can be as large as n^2, so the expected number of pairs is $O(n^{2-\varepsilon})$. Plugging this in to the runtime of the algorithm of [11], we get an expected running time of $O(n^{2-\varepsilon}\log n)$ □

[1] Recall that $GenLCS_{Pri,Aux}$ returns auxiliary instances from Aux conditioned on the longest common subsequence of the primary instance being a common subsequence of the auxiliary instances.

FIND-LCS($x, y, x^{(1)}, y^{(1)}, ..., x^{(l)}, y^{(l)}$) [11]

Construct the strings ($\mathbf{x}$) and ($\mathbf{y}$).
Initialize arrays THRESH, MATCHLIST, LINK, and pointer PTR.
For $i = 1, 2, ..., n$
    Set MATCHLIST[i] = $(j_1, j_2, ...j_p)$ such that $j_1 > j_2 > ... > j_p$ and $(x)_i = (y)_{j_q}$ for $1 \leq q \leq p$
Set THRESH[0] to 0
For $i = 1, ..., n$
    THRESH[i] = $n + 1$
For $i = 1, ..., n$
    For j on MATCHLIST[i]
        Find k such that THRESH[$k-1$] $< j \leq$ THRESH[k]
        If $j <$ THRESH[k]
            THRESH[k] = j
            LINK[k] = newnode (i, j,LINK[$k-1$])
k = largest k such that THRESH[k] $\neq n + 1$
PTR = LINK[k]
While PTR $\neq$ null
    Output (i, j) pointed to by PTR
    Advance PTR

Remark. *For $l \geq \log n$, the algorithm runs in expected time $O(n \log n)$.*

Remark. *We note that if the correlation model is changed such that the auxiliary sequences are the original sequences with each character changed with probability p, the algorithm still works with $O(\log n)$ sequences when p is constant.*

3.2 Longest Common Subsequence of k sequences

Section 3.1 gives us a way to use correlated instances to get faster algorithms for finding the longest common subsequence. This raises the question of whether we can extend these results to computing the longest common subsequence of k sequences. It is known that being able to compute the longest common subsequence of k sequences in $n^{k-\varepsilon}$ time would give faster algorithms for NC-SAT and prove circuit lower bounds [2].

To determine the longest common subsequence of k sequences, we will need a different algorithm. Note that in Claim 1, if the number of correlated instances is large enough (i.e. $O(\log n)$), every matching pair (i.e. (i, j) such that $(x)_i = (y)_j$) corresponds to part of the longest common subsequence with high probability. This means that if there was an efficient algorithm for finding these pairs, we could obtain the longest common subsequence efficiently. In addition, the existence of these pairs does not depend on k. Given strings $x[1], x[2], ..., x[k]$ and auxiliary instances from $GenLCS[k]_{Pri,Aux}$ with Pri worst-case and Aux uniform (i.e. the auxiliary instances will be random conditioned on the longest common subsequence of the primary instance being a common subsequence of the auxiliary instances), with high probability the pairs (i, j) such that $(x[1])_i = (x[s])_j$ are also $(a[1]_t, a[s]_t)$ for some t where $LCS(x[1], x[2], ..., x[k]) = \{(a[1]_1, a[2]_1, ..., a[k]_1), (a[1]_2, a[2]_2, ..., a[k]_2), ..., (a[1]_m, a[2]_m, ..., a[k]_m)\}$.

This suggests the following algorithm which runs in $O(kn \log n)$ time, or $O(n \log n)$ if k is constant. If we are given auxiliary instances from $GenLCS[k]_{Pri,Aux}$ with Pri a worst-case distribution and Aux the uniform distribution, we can construct buckets indexed by elements of Σ_{l+1} and the

entries of each bucket are the indices of the columns that have the same value as the bucket's index. Then, it looks up the columns of string 1 to see where the matches are. FindLCS-k proceeds in three steps; first, for the last $k-1$ strings, it puts each column of the string in the corresponding bucket in the set of buckets for that string. Next, it checks the columns of string 1 against the columns of string 2 to see which positions in string 1 are in the longest common subsequence; with high probability, only the positions in the LCS will match with columns in string 2. The final step is to find the positions in strings 2 through k corresponding to the positions in string 1 and output this matching.

In FindLCS-k, the H_j are arrays of buckets where the $(x[j])_i$ are stored for $j = 2, ..., n$, and A is the array of k-tuples in the longest common subsequence. $H_j[s]$ denotes the contents of the bucket for the jth string, and is filled iff there exists i such that $(x[j])_i = s$. $A_{i,j}$ is the index in the jth string of the ith entry of the longest common subsequence.

FindLCS-$k(x[1], x[2], ..., x[k], x^{(1)}[1], ..., x^{(1)}[k], ..., x^{(l)}[1], ..., x^{(l)}[k])$

Put columns into buckets

For $j = 2, ..., k$
    Construct the array H_j with elements linked lists.
    For $i = 1, ..., n$
        Add $((x[j])_i, i)$ to $H_j[(x[j])_i]$.
Make the $n \times k$ array A.
$m = 1$

Check which positions in string 1 are part of the LCS

For $i = 1, ..., n$
    For (v, j) in $H_2[(x[1])_i]$
        If $(x[1])_i = v$
            $A_{m,1} = i$
            $A_{m,2} = j$
            $m = m + 1$

Match positions in strings 2 through k to positions in string 1

For $s = 3, ..., k$
    For $l = 1, ..., m$
        $i = A_{l,1}$
        If $H_s[(x[1])_i]$ is empty
            Remove A_l and skip to the next l
        For (v, j) in $H_s[(x[1])_i]$
            If $(x[1])_i = v$
                $A_{m,s} = j$
Output A_i from $i = 1, ..., m$.

Claim 2. FindLCS-k *works with high probability in time* $O(nk \log n)$ *given instances from* $GenLCS[k]_{Pri,Aux}(n, l)$ *with Pri worst-case and Aux uniform when* $l = O(\log n)$. [2]

Proof. If $x[1]_i = x[s]_j$ and there is no t such that $a[1]_t = i$ and $a[s]_t = j$, which means that the ith character of $x[1]$ does not correspond to the jth character of $x[s]$ in the longest common subsequence, the probability that $(x[1])_i = (x[s])_j$ is equal to $1/2^l$ for any s because $GenLCS[k]_{Pri,Aux}$ samples independently and Aux is the uniform distribution. If $l = 2 \log n + \log k + 3$, the probability of

[2] Recall that $GenLCS[k]_{Pri,Aux}$ draws auxiliary instances from Aux conditioned on the longest common subsequence of the primary instance being a common subsequence of the auxiliary instances

this happening is $1/8kn^2$, and then taking the union bound over all $(k-1)n^2$ pairs gets us that the probability that there exists an i, j, s such that $(x[1])_i = (x[s])_j$ and there is no t such that $a[1]_t = i$ and $a[s]_t = j$ is $< 1/8$. Thus, with probability at least 7/8 FindLCS-k recovers the longest common subsequence.

To prove the time bound, the first loop takes time $O(nk \log n)$, the second loop takes time $O(nk \log n)$, and the third loop takes time $O(nk \log n)$, which makes the total runtime $O(nk \log n)$. □

Remark. *As is the case in the previous section, if the correlation model is changed such that the auxiliary sequences are the original sequences with each character changed with probability p, the algorithm still works with $O(\log n)$ sequences when p is constant.*

3.3 Edit Distance

Suppose that we have two strings x, y for which we want to compute the minimum edit distance. Algorithm FindLCS-k for $k = 2$, given strings x, y and correlated instances $x^{(1)}, y^{(1)}, ..., x^{(l)}, y^{(l)}$ can be used to find i, j such that $(x)_i = (y)_j$. What does this mean in the edit distance setting? If our auxiliary instances have the same sequence of edit operations, then if $(x)_i = (y)_j$, with high probability x_i moved to y_j during the sequence of inserts, deletes, and changes. This means that we can find the parts of the original string that are preserved under the edit and their new positions. Our algorithm works as follows. First, we get $O(\log n)$ auxiliary instances from $Genedit_{Pri,Aux,\varepsilon}$, where Pri is worst-case, and then put the $(y)_j$ in buckets and check which $(x)_i$ have $(x)_i = (y)_j$. Then, with the pairs $(a_1, b_1), ..., (a_k, b_k)$, the edit distance is $\sum_{i=0}^{k} \max a_{i+1} - a_i - 1, b_{i+1} - b_i - 1$ with high probability. This algorithm obviously runs in $O(n \log n)$ time.

In Find-EDIT, the array H of buckets is used to store $(y)_i$, and $COMMONPAIRS$ holds the i, j such that $(x)_i = (y)_j$.

Find-EDIT$(x, y, x^{(1)}, y^{(1)}, ..., x^{(l)}, y^{(l)})$

1 Initialize an $n \times 2$ array $COMMONPAIRS$.
2 Construct an array H of size n of linked lists.
3 For $j = 1, ..., n$
4 Add $((y)_j, j)$ to $H[(y)_j]$.
5 For $i = 1, ..., n$
6 For (v, j) in $H[(x)_i]$
7 If $(x)_i = v$ add (i, j) to $COMMONPAIRS$.
8 Output $\sum_{i=0}^{k} \max (a_{i+1} - a_i - 1, b_{i+1} - b_i - 1)$,
where $COMMON - PAIRS = ((a_1, b_1), (a_2, b_2), ..., (a_k, b_k))$.

Claim 3. *The algorithm* Find-EDIT *on $O(\log n)$ instances generated by $Genedit_{Pri,Aux,\varepsilon}(n, l)$ with Pri worst-case and Aux uniform works and runs in time $O(n \log n)$ with high probability.* [3]

Proof. By Claim 2, the hashing finds the pairs $(a_1, b_1), ..., (a_k, b_k)$ with probability at least 7/8 in expected time $O(n \log n)$. We claim that the edit distance is equal to $\sum_{i=0}^{k} \max (a_{i+1} - a_i - 1, b_{i+1} - b_i - 1)$ if this is true. To see this, the original edit takes x_{a_i} to y_{b_i}, and thus everything between x_{a_i} and $x_{a_{i+1}}$ must be matched to things between y_{b_i} and $y_{b_{i+1}}$. This edit distance is equal to $\max (a_{i+1} - a_i - 1, b_{i+1} - b_i - 1)$. Suppose that the edit distance was less. Then there must be

[3] Recall that $Genedit_{Pri,Aux,\varepsilon}$ outputs auxiliary instances where the first string is a perturbation of the original string and the second string is obtained by applying the same edits to the first string.

a coordinate x_c or y_d that was in the original string. But then, we could extend this to a smaller edit for the entire string than the original edit, which is a contradiction. Thus, the edit distance is at least $\max(a_{i+1} - a_i - 1, b_{i+1} - b_i - 1)$, and this is achieved by the original edit, because none of them are preserved. The running time is $O(n \log n)$ because each loop takes time $O(n \log n)$. □

3.4 Less correlated generating processes

In section 3.2, we showed that if we are given auxiliary instances which contain the longest common subsequence of the primary instances as a common subsequence, we can find the longest common subsequence of k sequences with high probability in almost-linear time, which contrasts with the n^k lower bound of Abboud, Hansen, Williams, and Williams assuming there is no way to solve satisfiability of $o(n)$-depth bounded-fanin circuits in time $O((2-\delta)^n)$ [2]. What happens if we relax the restriction that the auxiliary instances all contain the longest common subsequence of the original instance, and instead say that each pair of the longest common subsequence is present in the auxiliary instances with probability $1/2 + \varepsilon$ instead of $1/2$? Clearly our original algorithm will not work, because $(x)_i$ and $(y)_j$ for i, j in the longest common subsequence are not guaranteed to map to the same hash value. However, there will still be a difference in how correlated the columns are, and we can use that to distinguish between pairs of columns that are in the longest common subsequence and pairs of columns that are not in the longest common subsequence. Finding these pairs of columns is equivalent to finding the pairs of columns with small Hamming distance, because the more correlated columns will have a smaller Hamming distance. There are two algorithms for solving Hamming nearest neighbors in subquadratic time; an algorithm based on the locality sensitive hashing techniques of Gionis, Indyk, and Motwani [8], and an algorithm of Alman and Williams based on probabilistic polynomials [3]. A technical lemma follows, and then a description of the algorithm based on the paper of Gionis, Indyk, and Motwani.

Definition 9. $\mathbb{1}_{x_i^{(m)}=y_j^{(m)}}$ *is 1 if* $x_i^{(m)} = y_j^{(m)}$ *and 0 otherwise.*

Lemma 2. *Suppose we have* x *and* y *such that* $LCS(x,y) = ((a_1,b_1),(a_2,b_2),...,(a_k,b_k))$ *and* $x^{(1)}, x^{(2)}, ..., x^{(l)}, y^{(1)}, y^{(2)}, ..., y^{(l)}$ *drawn from* $GenLCS_{\varepsilon,Pri,Aux}(n,l)$ *with Pri worst-case and Aux uniform. Then, for any* δ *such that* $0 < \delta < 1/2 + \varepsilon$*, if* $i = a_k$ *and* $j = b_k$ *for some* k*,*

$$\Pr[\mathbb{E}_m[\mathbb{1}_{x_i^{(m)}=y_j^{(m)}}] \leq 1/2 + \varepsilon - \delta] \leq e^{-\frac{(\delta/(1/2+\varepsilon))^2 l}{2}}$$

and otherwise

$$\Pr[\mathbb{E}_m[\mathbb{1}_{x_i^{(m)}=y_j^{(m)}}] \geq 1/2 + \delta] \leq e^{-4\delta^2 l/3}$$

Proof. We use the form of the Chernoff bound given in [12]: for X a sum of independent random variables $X_1, ..., X_n$ taking values $\{0,1\}$ with expectation μ, we have that $\Pr[X \leq (1-\delta)\mu] \leq e^{-\delta^2\mu/2}$ and $\Pr[X \geq (1+\delta)\mu] \leq e^{-\delta^2\mu/3}$ for $0 < \delta < 1$. The first inequality is proven by plugging in $\mu = (1/2 + \varepsilon) * l$ into the first Chernoff bound and noting that the expectation is the sum of the indicator random variables divided by l, and the second inequality follows similarly. □

In FindLCS-WeakCorrelation, f_i is the hash function used to store the column $(y)_j$ in H_i. M is a candidate list of matches for every $(x)_i$.

FindLCS-WeakCorrelation$(x, y, x^{(1)}, y^{(1)}, ..., x^{(l)}, y^{(l)}, \varepsilon, \delta, k, m)$

Initialize the hash tables for the locality sensitive hashing algorithm

For $i = 1, ..., m$.
  Let $f_i(x) = (x_{c_1}, x_{c_2}, ..., x_{c_k})$ with the c_t drawn uniformly from $\{1, ..., l\}$.
  Let H_i be an array of size 2^k of buckets.
  For $j = 1, ..., n$
    Add $(j, (y)_j)$ to $H_i[f_i((y)_j)]$.

Find the columns of x that have a match in one of the tables

For $i = 1, ..., n$
  Create a list M
  For $j = 1, ..., m$
    Add everything in $H_j[f_j((x)_i)]$ to M.

Check if one of the matches is correct

  For (j, v) in M
    If $|\{t | x_i^{(t)} = y_j^{(t)}\}| \geq (1/2 + \delta) * l$
      Output (i, j).

Claim 4. FindLCS-WeakCorrelation *with* ε, $\delta = \sqrt{\frac{9 \log n}{l}}$, $k = \frac{\log n}{\log \frac{1}{1/2+\delta}}$, *and* $m = 8n^{\frac{\log \frac{2}{1+2\epsilon-2\delta}}{\log \frac{2}{1+2\delta}}}$ *and instances drawn from* $GenLCS_{\varepsilon,Pri,Aux}(n, l)$ *with* Pri *worst-case and* Aux *uniform works with probability at least 2/3 and runs in time* $O(n^{1+\frac{\log \frac{2}{1+2\epsilon-2\delta}}{\log \frac{2}{1+2\delta}}} l)$ [4].

Proof. Suppose that i, j are such that there is no k such that $i = a_k$ and $j = b_k$. By the lemma, $\Pr[\Pr[(x)_i = (y)_j] > 1/2 + \sqrt{9 \log n/l}] \leq e^{-4*(9 \log n/l)*l/3} \leq 1/n^3$. If there exists k such that $i = a_k$ and $j = b_k$, then using the lemma, $\Pr[\Pr[(x)_i = (y)_j] < 1/2 + \epsilon - \sqrt{9 \log n/l}] \leq e^{-(9 \log n/l)/(1/2+\varepsilon)^2 * l/2} \leq 1/n^3$. By a union bound, the probability that either of these things happens for any pair of $(x)_i$ and $(y)_j$ is $\leq 1/n$ which is smaller than any constant for sufficiently large n. Otherwise, we have that if $(x)_i$ and $(y)_j$ do not match they agree in a less than $1/2 + \delta$ fraction of elements, and if $(x)_i$ and $(y)_j$ do match then they agree in a more than $1/2 + \varepsilon - \delta$ fraction of elements. By Theorem 1 of [8] and the fact that $(1/2 + 1/e)^8 < 1/3$, with probability at least $2/3$ every x_i will match to y_j when i, j is in the longest common subsequence, and the number of i, j such that there exists a k such that $f_k((x)_i) = f_k((y)_j)$ and i, j is not in the longest common subsequence is at most $O(n^{\frac{\log \frac{2}{1+2\epsilon-2\delta}}{\log \frac{2}{1+2\delta}}})$. Then, the comparison step takes time at most $O(n^{1+\frac{\log \frac{2}{1+2\epsilon-2\delta}}{\log \frac{2}{1+2\delta}}} l)$. □

This algorithm also works for finding the longest common subsequence in k sequences, by applying this process to every string paired with the first string.

Acknowledgements

We are grateful to Guy Rothblum for early important discussions on this work and the choices of correlation models. Thank you Guy! We also thank Aviv Regev for helpful discussion on correlations in sequence alignment in bioinformatics.

[4] Recall that $GenLCS_{\varepsilon,Pri,Aux}$ adjusts each pair in the longest common subsequence of the primary instance in the auxiliary instances with probability ε.

References

[1] Amir Abboud, Arturs Backurs, and V. Vassilevska Williams. Tight hardness results for lcs and other sequence similarity measures. In *FOCS 2015*.

[2] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends or: A polylog shaved is a lower bound made, 2015.

[3] Josh Alman and Ryan Williams. Probabilistic polynomials and hamming nearest neighbors. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 136–150. IEEE, 2015.

[4] Alexandr Andoni and Robert Krauthgamer. The smoothed complexity of edit distance. *ACM Transactions on Algorithms (TALG)*, 8(4):44, 2012.

[5] Piotr Indyk Arturs Backurs. Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *STOC15*.

[6] Irit Dinur, Shafi Goldwasser, and Huijia Lin. The computational benefit of correlated instances. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 219–228, 2015.

[7] Uriel Feige and Joe Kilian. Heuristics for semirandom graph problems. *Journal of Computer and System Sciences*, 63(4):639–671, 2001.

[8] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[9] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM (JACM)*, 45(4):653–750, 1998.

[10] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, 2012.

[11] James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[12] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[13] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis. In *Algorithms and data structures*, pages 256–270. Springer, 2003.

[14] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de