Approximating LCS and Alignment Distance over Multiple Sequences

Debarati Das
Pennsylvania state University, University Park, PA, USA

Barna Saha
University of California, San Diego, CA, USA

Abstract

We study the problem of aligning multiple sequences with the goal of finding an alignment that either maximizes the number of aligned symbols (the longest common subsequence (LCS) problem), or minimizes the number of unaligned symbols (the alignment distance aka the complement of LCS). Multiple sequence alignment is a well-studied problem in bioinformatics and is used routinely to identify regions of similarity among DNA, RNA, or protein sequences to detect functional, structural, or evolutionary relationships among them. It is known that exact computation of LCS or alignment distance of m sequences each of length n requires $\Theta(n^m)$ time unless the Strong Exponential Time Hypothesis is false. However, unlike the case of two strings, fast algorithms to approximate LCS and alignment distance of multiple sequences are lacking in the literature. A major challenge in this area is to break the triangle inequality. Specifically, by splitting m sequences into two (roughly) equal sized groups, then computing the alignment distance in each group and finally combining them by using triangle inequality, it is possible to achieve a 2-approximation in $\tilde{O}_m(n^{\lceil \frac{m}{2} \rceil})$ time. But, an approximation factor below 2 which would need breaking the triangle inequality barrier is not known in $O(n^{\alpha m})$ time for any $\alpha < 1$. We make significant progress in this direction.

First, we consider a semi-random model where, we show if **just one** out of m sequences is (p, B)-pseudorandom then, we can get a below-two approximation in $\tilde{O}_m(nB^{m-1} + n^{\lfloor \frac{m}{2} \rfloor + 3})$ time. Such semi-random models are very well-studied for two strings scenario, however directly extending those works require one but all sequences to be pseudorandom, and would only give an $O(\frac{1}{p})$ approximation. We overcome these with significant new ideas. Specifically an ingredient to this proof is a new algorithm that achives below 2 approximations when alignment distance is large in $\tilde{O}_m(n^{\lfloor \frac{m}{2} \rfloor + 2})$ time. This could be of independent interest.

Next, for LCS of m sequences each of length n, we show if the optimum LCS is λn for some $\lambda \in [0, 1]$, then in $\tilde{O}_m(n^{\lfloor \frac{m}{2} \rfloor + 1})$[1] time, we can return a common subsequence of length at least $\frac{\lambda^2 n}{2+\epsilon}$ for any arbitrary constant $\epsilon > 0$. In contrast, for two strings, the best known subquadratic algorithm may return a common subsequence of length $\Theta(\lambda^4 n)$.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis

Keywords and phrases String Algorithms, Approximation Algorithms

Digital Object Identifier 10.4230/LIPIcs.APPROX/RANDOM.2022.54

Category APPROX

Related Version *Full Version*: `https://arxiv.org/abs/2110.12402`

Funding *Barna Saha*: Partly supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.

[1] In the context of multiple sequence alignment, we use $\tilde{O}_m$ to hide factors like $c^m \log^m n$ where c is a constant.

1 Introduction

Given m sequences each of length n, we are interested to find an alignment that either maximizes the number of aligned characters (the longest common subsequence problem (LCS)), or minimizes the number of unaligned characters (the minimum alignment distance problem, aka the complement of LCS) [2]. Both these problems are extremely well-studied, are known to be notoriously hard, and form the cornerstone of *multiple sequence alignment* [29, 26, 16], which according to the survey in Nature is one of the most widely used modeling methods in biology [31]. Long back in 1978, the multi-sequence LCS problem (and therefore, the minimum alignment distance problem) was shown to be NP Hard [23]. Moreover, for any constant $\delta > 0$, the multi-sequence LCS and alignment distance cannot be approximated within $n^{1-\delta}$ unless $P = NP$ [18]. These hardness results hold even under restricted conditions such as for sequences over relatively small alphabet [9], or with certain structural properties [10]. Various other multi-sequence based problems such as finding the median or center string are shown "hard" by reduction from the minimum alignment distance problem [25]. Interested readers may refer to the chapter entitled "Multi String Comparison-the Holy Grail" of the book [16] for a comprehensive study on this topic.

From a fine-grained complexity viewpoint, an $O(n^{m-\epsilon})$ algorithm to compute alignment distance of m sequences for any constant $\epsilon > 0$ will refute the Strong Exponential Time Hypothesis (SETH) [1]. On the other hand, a basic dynamic programming solves these problems in time $O(mn^m)$. This raises the question whether we can solve these problems faster in $O(n^{\alpha m})$ time for $\alpha < 1$ by allowing approximation. The approximation vs running time trade-off for $m = 2$ (edit distance problem) has received extensive attention over the last two decades with many recent breakthroughs [21, 7, 6, 8, 3, 5, 11, 14, 27, 13, 19, 4]. To bypass the worst-case hardness in the two-strings setting, multiple prior works have studied semi-random models for sequence comparisons [2, 20, 12]. Semi-random models may capture real-life scenarios better where adversarial examples are rare. In addition, it may also carry several inherent difficulties of the worst case model. Thus studying semi-random models can be a stepping stone towards attacking the worst-case model.

More than a decade back, such a study was initiated by Andoni and Krauthgamer [2], where the authors studied smoothed complexity of sequence alignment. They proposed a semi-random model as follows: first, an adversary chooses two binary strings of length n and a longest common subsequence A of them. Then, every character is perturbed independently with probability p, except that A is perturbed in exactly the same way inside the two strings. Kuszmaul further generalized this model and considered one input string to be pseudorandom (any pair of disjoint substrings are at large edit distance) whereas the other input string can be adversarial [20] . Both of these works [2, 20] provide $O(1)$ approximation of the edit distance in almost linear time, and face the triangle inequality barrier. Recently Boroujeni, Seddighin, and Seddighin [12] improved the approximation guarantee to $(1 + \epsilon)$ thus bypassing the triangle inequality hardness while increasing the running time from near-linear to subquadratic. Therefore, if we aim to generalize the pseudorandom model for multiple strings, it is not obvious how to achieve the best of the above two results simultaneously: (i) a running time of $O(n^{m/2})$, and (ii) a below 2 approximation. Another major issue is that any direct generalization of [2, 20, 12] for multiple strings require all but one string to be pseudorandom.

[2] While one may define alignment distance in many different ways among multiple sequences, taking the complement of LCS is possibly the cleanest way of defining such a distance both because indel distance between two strings naturally generalizes to it, and due to its close connection to LCS.

In this work we consider a much stronger model where only one input string is pseudorandom and the rest $m-1$ input strings can be adversarial. We show how it is possible to accomplish the best of both the running time and the approximation guarantee by giving an $\tilde{O}(n^{m/2+3})$ time algorithm that breaks the triangle inequality and computes truly below 2 approximation of alignment distance. Towards this we first design an algorithm that takes as input m adversarially chosen strings and provides a below 2 approximation of their alignment distance in time $\tilde{O}(n^{m/2+2})$ provided the distance is large. Note this model considers all strings to be adversarial and thus can be of independent interest. Moreover we show that these techniques can be extended to design an algorithm computing constant approximation of the LCS of m strings in time $\tilde{O}(n^{m/2+2})$ provided the length of the LCS is large.

It is interesting to note that our results on LCS implies a constant approximation of LCS of three strings is possible in the large distance regime in quadratic time (a reduction from cubic to quadratic time complexity), whereas a worse constant approximation is currently known in the large distance regime for the LCS of two strings to go below the quadratic running time [28].

Contributions. We now describe our results in more details.

Minimizing Alignment Distance of Multiple Sequences with One Pseudorandom String. Let $\mathcal{L}(s_1,\ldots,s_m)$ denote the length of LCS of m strings $s_1, s_2, .., s_m$ (each of length n) and $\mathcal{A}(s_1,\ldots,s_m) = n - \mathcal{L}(s_1,\ldots,s_m)$ denote the optimal alignment distance of $s_1, s_2, ..., s_m$. We consider the case where one input string is pseudorandom out of m strings, and the rest of $m-1$ strings are chosen adversarially. We provide an algorithm that breaks the triangle inequality barrier and provides $(2-\frac{3p}{512}+\epsilon)$ (for any arbitrary small constant $\epsilon > 0$) approximation of $\mathcal{A}(s_1,\ldots,s_m)$ in time $\tilde{O}_m(nB^{m-1} + n^{\lfloor m/2\rfloor+3})$. Formally we show the following.

▶ **Definition 1** ((p,B)-pseudorandom)**.** *Given a string s of length n and parameters $p, B \geq 0$ where p is a constant, we call s a (p,B)-pseudorandom string if for any two disjoint B length substrings x, y of s, $\mathcal{A}(x,y) \geq pB$.*

▶ **Theorem 2.** *Given a (p,B)-pseudorandom string s_1, and $m-1$ adversarial strings $s_2,\ldots,s_m$ of length n, there exists an algorithm that for any arbitrary small constant $\epsilon > 0$ computes $(2-\frac{3p}{512}+\epsilon)$ approximation of $\mathcal{A}(s_1,\ldots,s_m)$ in time $\tilde{O}_m(nB^{m-1} + n^{\lfloor m/2\rfloor+3})$.*

The theorem can be extended to get a $c(1-\frac{3p}{1024}+\epsilon)$ approximation in $\tilde{O}_m(nB^{\lceil 2m/c\rceil-1} + n^{\lceil m/c\rceil+3})$ time. Assuming c to be even, we divide the input strings into $\frac{c}{2}$ groups each containing at most $\lceil\frac{2m}{c}\rceil$ strings. Then for each group we compute a below 2-approximation of the alignment distance in time $\tilde{O}_m(nB^{\lceil 2m/c\rceil-1} + n^{\lceil m/c\rceil+3})$. Finally, we apply triangle inequality $\frac{c}{2}$ times to combine these groups to get a $\frac{c}{2}(2-\frac{3p}{512}+\epsilon) = c(1-\frac{3p}{1024}+\frac{\epsilon}{2})$ approximation.

What do we know in the two strings case? Let us contrast this result to what is known for $m=2$ case [2, 20, 12]. When one of the two strings is (p,B)-pseudorandom, Kuszmaul gave an algorithm that runs in time $\tilde{O}(nB)$ time but only computes an $O(\frac{1}{p})$ (can be large constant) approximation to edit distance [20]. Boroujeni, Seddighin and Seddighin consider a different random model for string generation under which they give a $(1+\epsilon)$ approximation but in subquadratic time [12]. While their model captures the case when one string in generated uniformly at random, it does not extend to pseudorandom strings. In fact, there is no result in the two strings case that breaks the triangle inequality barrier and provides below-2 approximation when one of the strings is pseudorandom. Moreover, in order to apply their technique to multi-string setting, *we would need all but one string to be generated according to their model.* Interestingly, our algorithm obtains all the desired results and provides below-2 approximation of alignment distance with just *one pseudorandom string.* We stress that this is one of the important contributions of our work and is technically involved.

Key Tool: breaking triangle inequality for large alignment distance. To construct the above mentioned algorithm, we first design an algorithm that takes as input m adversarially chosen strings and provides truly below 2 approximation of their alignment distance provided the distance is large. More generally we show if $\mathcal{A}(s_1,\dots,s_m)=\theta n$ then for any arbitrary small constant $\epsilon > 0$, it is possible to obtain a $c(1-\frac{3\theta}{32}+\epsilon)$ approximation[3] in time $\tilde{O}_m(n^{\lceil m/c\rceil+2})$ time.

▶ **Theorem 3.** *Given m strings $s_1,\dots,s_m$ of length n over some alphabet set Σ such that $\mathcal{A}(s_1,\dots,s_m)=\theta n$, where $\theta\in(0,1)$, there exists an algorithm that for any arbitrary small constant $\epsilon>0$ computes a $(2-\frac{3\theta}{16}+\epsilon)$ approximation of $\mathcal{A}(s_1,\dots,s_m)$ in time $\tilde{O}_m(n^{\lfloor m/2\rfloor+2})$. Moreover, for any integer $c>0$, there exists an algorithm that computes $c(1-\frac{3\theta}{32}+\epsilon)$ approximation of $\mathcal{A}(s_1,\dots,s_m)$ in time $\tilde{O}_m(n^{\lceil m/c\rceil+2})$.*

For constant θ, the above theorem asserts that there exists an algorithm that breaks the triangle inequality barrier and computes a truly below 2-approximation of $\mathcal{A}(s_1,\dots,s_m)$ in time $\tilde{O}_m(n^{\lfloor m/2\rfloor+2})$. Note here all the input strings are adversarially chosen and this result can be of independent interest. Moreover we show these techniques can be extended to compute LCS of multiple strings.

LCS of Multiple Sequences. We show if $\mathcal{L}(s_1,\dots,s_m)=\lambda n$ for some $\lambda\in[0,1]$, then we can return a common subsequence of length $\frac{\lambda^2 n}{2+\epsilon}$ in time $\tilde{O}_m(n^{\lfloor m/2\rfloor+1})$. To contrast, we can get a quadratic algorithm for $m=3$ with $\frac{\lambda}{2+\epsilon}$ approximation (for any arbitrary small constant $\epsilon>0$), whereas the best known bound for $m=2$ case may return a subsequence of length $\Theta(\lambda^4 n)$ in $\tilde{O}(n^{1.95})$ time [28].

▶ **Theorem 4.** *Given m strings $s_1,\dots,s_m$ of length n over some alphabet set Σ such that $\mathcal{L}(s_1,\dots,s_m)=\lambda n$, where $\lambda\in[0,1]$, there exists an algorithm that for any arbitrary small constant $\epsilon>0$ computes an $\frac{\lambda}{2+\epsilon}$ approximation of $\mathcal{L}(s_1,\dots,s_m)$ in time $\tilde{O}_m(n^{\lfloor m/2\rfloor+1})$.*

1.1 Technical Overview

Notation

We use the following notations throughout the paper. Given m strings $s_1,\dots,s_m$, each of length n over some alphabet set Σ, the longest common subsequence (LCS) of $s_1,\dots,s_m$, denoted by $LCS(s_1,\dots,s_m)$ is one of the longest sequences that is present in each s_i. Define $\mathcal{L}(s_1,\dots,s_m)=|LCS(s_1,\dots,s_m)|$. The optimal alignment distance (AD) of $s_1,\dots,s_m$, denoted by $\mathcal{A}(s_1,\dots,s_m)$ is $n-\mathcal{L}(s_1,\dots,s_m)$.

For a given string s, $s[i]$ represents the ith character of s and $s[i,j]$ represents the substring of s starting at index i and ending at index j. Given a LCS σ of $s_1,\dots,s_m$ define $\sigma(s_j)\subseteq[n]$ be the set of indices such that for each $k\in\sigma(s_j)$, $s_j[k]$ is aligned in σ and $\bar{\sigma}(s_j)\subseteq[n]$ be the

[3] We will assume c is even for simplicity. But all the algorithms work equally well if c is odd.

set of indices of the characters in s_j that are not aligned in σ. Define the *alignment cost* of σ to be $|\bar{\sigma}(s_1)|$ and the *cumulative alignment cost* of σ to be $\sum_{j=1}^{m} |\bar{\sigma}(s_j)| = m|\bar{\sigma}(s_1)|$. Given a set $T \subseteq [n]$ and a string s, let s^T denote the subsequence of s containing characters with indices in T.

Given a string s, we define a window w of size d of s to be a substring of s having length d. Given m strings $s_1, \ldots, s_m$, we define a m-window tuple to be a set of m windows denoted by $(w_1, \ldots, w_m)$, where w_j is a window of string s_j.

Given two characters $a, b \in \Sigma$, $a \circ b$ represents the concatenation of b after a. Given two string x, y, $x \circ y$ represents the concatenation of string y after x. For notational simplicity we use $\tilde{O}_m$ to hide factors like $c^m \log^m n$, where c is a constant. Moreover we use $\tilde{O}$ to hide polylog factors.

1.1.1 Breaking the Triangle Inequality Barrier for Large Alignment Distance and Approximating LCS

We first give an overview of our algorithms leading to Theorem 5 (Section 2). Let us consider the problem of minimizing the alignment distance. Given m (say m is even) sequences $s_1, s_2, ..., s_m$ each of length n, partition them into two groups $G_1 = \{s_1, s_2, ..., s_{m/2}\}$ and $G_2 = \{s_{m/2+1}, .., s_m\}$. Suppose the optimum alignment distance of the m sequences is $d = \theta n$. With each alignment, we can associate a set of indices of s_1 that are not aligned in that alignment. Let σ^* be an optimum alignment and $\bar{\sigma^*}(s_1)$ be that set. We have $|\bar{\sigma^*}(s_1)| = d$. Let $\mathcal{X}_1 = \{(\sigma_i, \bar{\sigma_i}(s_1))\}$ denote all possible alignments σ_i of G_1 of cost at most d, $|\bar{\sigma_i}(s_1)| \leq d$. Then $(\sigma^*, \bar{\sigma^*}(s_1)) \in \mathcal{X}_1$. Therefore, if we can (i) find all possible alignments $\mathcal{X}_1$, and (ii) for each $(\sigma_i, \bar{\sigma_i}(s_1)) \in \mathcal{X}_1$ can verify if that is a valid alignment of G_2, we can find an optimal alignment.

Unfortunately, it is possible that $|\mathcal{X}_1| = \sum_{l \leq d} \binom{n}{l}$ which is prohibitively large. Therefore, instead of trying to find all possible alignments, we try to find a *cover* for $\mathcal{X}_1$ using a few alignments $(\tau_j, \bar{\tau_j}(s_1))$, $j = 1, 2, .., k$ such that for any $(\sigma_i, \bar{\sigma_i}(s_1)) \in \mathcal{X}_1$, there exists a $(\tau_j, \bar{\tau_j}(s_1))$ with large $|\bar{\sigma_i}(s_1) \cap \bar{\tau_j}(s_1)|$. In fact, one of the key ingredients of our algorithm is to show such a covering exists and can be obtained in time (roughly) $n^{|G_1|}$. With just $k = \frac{4}{\theta}$ alignments, we show it is possible to cover $\mathcal{X}_1$ such that for any $(\sigma_i, \bar{\sigma_i}(s_1)) \in \mathcal{X}_1$, there exists a $(\tau_j, \bar{\tau_j}(s_1))$ having $|\bar{\sigma_i}(s_1) \cap \bar{\tau_j}(s_1)| \geq \frac{3\theta^2 n}{16}$.

The algorithm to compute the covering starts by finding any optimal alignment $(\sigma_1, \bar{\sigma_1}(s_1))$ of G_1. Next it finds another alignment $(\sigma_2, \bar{\sigma_2}(s_1))$ of cost at most d which is *farthest* from $(\sigma_1, \bar{\sigma_1}(s_1))$, that is $|\bar{\sigma_1}(s_1) \cap \bar{\sigma_2}(s_1)|$ is minimized. We find these alignments using dynamic programming. If $|\bar{\sigma_1}(s_1) \cap \bar{\sigma_2}(s_1)| \sim |\bar{\sigma_2}(s_1)|$, then it stops. Otherwise, it finds another alignment $(\sigma_3, \bar{\sigma_3}(s_1))$ such that $|(\bar{\sigma_1}(s_1) \cup \bar{\sigma_2}(s_1)) \cap \bar{\sigma_3}(s_1)|$ is minimized. We show the process terminates after at most $\frac{4}{\theta}$ rounds.

Suppose without loss of generality, $|\bar{\sigma^*}(s_1) \cap \bar{\tau_1}(s_1)| \geq \frac{3\theta^2 n}{16}$. Given $\bar{\tau_1}(s_1), \bar{\tau_2}(s_1), ..., \bar{\tau_k}(s_1)$, for each $(\tau_i, \bar{\tau_i}(s_1))$, we find an alignment $(\rho_i, \bar{\rho_i}(s_1))$ of $G_2 \cup s_1$ of cost at most d such that $\bar{\rho_i}(s_1)$ is *nearest* to $\bar{\tau_i}(s_1)$, that is $|\bar{\rho_i}(s_1) \cap \bar{\tau_i}(s_1)|$ is maximized. Then, we must have $|\bar{\rho_i}(s_1) \cap \bar{\tau_1}(s_1)| \geq |\bar{\sigma^*}(s_1) \cap \bar{\tau_1}(s_1)| \geq \frac{3\theta^2 n}{16}$. Our alignment cost is $\min_j (|\bar{\tau_j}(s_1) \cup \bar{\rho_j}(s_1)|) \leq |\bar{\tau_1}(s_1) \cup \bar{\rho_1}(s_1)| \leq 2d - \frac{3\theta^2 n}{16} = d(2 - \frac{3\theta}{16})$ giving the desired below-2 approximation when θ is a constant.

Of course, there are two main parts in this algorithm that we have not elaborated; given a set of indices T of s_1, and a group of strings G, we need to find an alignment of cost at most d of $G \cup s_1$ that is farthest from (nearest to) T. In general, any application that needs to compute multiple diverse (or similar) alignments can be benefited by such subroutines. We can use dynamic programming to solve these problems; however, it is important to keep in mind – an alignment that has minimum cost may not necessarily be the farthest (or nearest). Thus, we need to check all possible costs up to the threshold d to find such an alignment.

Our algorithm for obtaining a $\frac{\lambda}{(2+\epsilon)}$ approximation for multi-sequence LCS is nearly identical to the above, and in fact simpler. This helps us to improve the running time slightly (contrast Theorem 4 with Theorem 5). Moreover, the result holds irrespective of the size of LCS. We provide the details in Section 4.

1.1.2 Approximating Alignment Distance with just One Pseudorandom String

Next we consider the case where the input consists of a single (p, B) pseudorandom string and $m-1$ adversarial strings each of length n. We give an overview of our algorithm that returns a below 2 approximation of the optimal alignment distance (even for small regime) proving Theorem 2. The details are provided in Section 3.

In most of the previous literature for computing edit distance of two strings, the widely used framework first partitions both the input strings into windows (substrings) and finds distance between all pairs of windows. Then using dynamic program all these subsolutions are combined to find the edit distance between the input strings. However instead of considering two arbitrary strings as input if one input is (p, B) pseudorandom, then as we know that any pair of disjoint windows from the pseudorandom string have large edit distance, if we consider a window from the adversarial string then by triangle inequality, there exists at most one window in the pseudorandom string with which it can have small edit distance ($\leq \frac{pB}{4}$). We call this low cost match between an adversarial string window and a pseudorandom string window a *unique match*. Notice if we can identify one such unique match that is part of an optimal alignment, we can put restriction on the indices where the rest of the substrings can be matched. This observation still holds for multiple strings but only when we compare a pair of windows, one from the pseudorandom string and the other from an adversarial string. Thus it is not obvious how we can extend this restriction on pairwise matching to a matching of m-window tuples as $(m-1)$-window tuples come from $(m-1)$ different adversarial strings and their *unique matches* with the pseudorandom string can be very different from each other. Another drawback of this approach is that, to optimize the running time, here the algorithm aims to identify only the matchings with low cost i.e. $< \frac{pB}{4}$. Hence the best approximation ratio we can hope for is $O(1/p)$ which can be a large constant.

Therefore to shed the approximation factor below 2, we also need to find a good approximation of the cost of pair of windows having distance $\geq \frac{pB}{4}$. We call a matching with cost $\geq \frac{pB}{4}$ a *large cost match*. However as the unique match property fails here, without having any prior knowledge about the optimal alignment we need to compute the cost for all pairs of windows having large cost. However doing it trivially can not provide us the desired running time. Fortunately, as p is a constant $pB/4 = \Omega(B)$ and thus we can use our large alignment distance approximation algorithm to get a improved running time while ensuring below 2 approximation of the cost for these *large cost match* tuples. Though this simple idea seems promising, if we try to compute an approximation over all large distance m-window tuples the running time can become as large as $\tilde{O}_m(n^{\frac{11m}{16}})$. We show this with an example. Given m input strings, start by partitioning each string into windows of length $\beta = n^{\frac{5}{8}}$ (for simplicity assume the windows are disjoint). Hence there are $n^{\frac{3}{8}}$ windows in each string. Now there can be as many as $n^{\frac{3m}{8}}$ many m-window tuples of large cost. If we evaluate each of their cost using our *large alignment distance* algorithm then time taken for each m-tuple is roughly $\tilde{O}_m(n^{\frac{5m}{16}})$. Hence total time required is $\tilde{O}_m(n^{\frac{11m}{16}})$. Note to improve this running

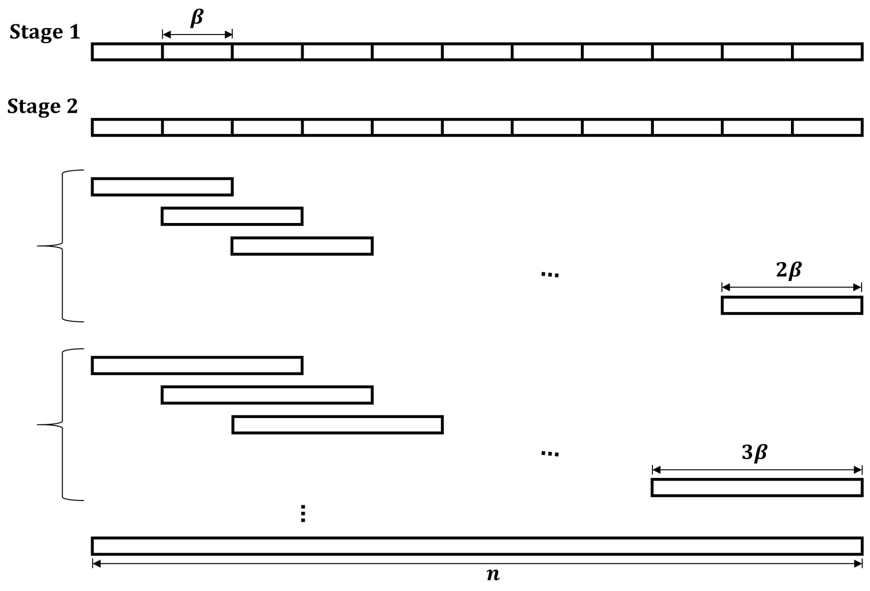


Figure 1 Construction of windows for the (p, B)-pseudorandom string s_1.

time by reducing the number of windows, we can not grow the window size arbitrarily large as in that case identifying the unique (low cost) match m-window tuples will become more time consuming. Later we show for window size β this time can be $\tilde{O}_m(\beta^m)$.

Thus to further reduce the running time, instead of evaluating all window tuples having large cost match, we restrict the computation by estimating the cost of only those tuples that maybe necessary to compute an optimal alignment. This is challenging as we do not have any prior information of the optimal alignment. For this purpose we use an adaptive strategy where depending on the unique matches computed so far, we perform a restricted search to estimate the cost of window tuples having large alignment distance. Moreover the length of these windows are also decided adaptively. We remark that this adaptive strategy differs significantly from the previous windowing strategies where the window lengths are fixed to start with. Next we give a brief overview of the three main steps of our algorithm. In Step 1, we provide the construction of windows of the input strings that will be used as input to Step 2 and 3. In Step 2, we estimate the alignment cost of the m-window tuples such that the matching is unique i.e. the optimal alignment distance is at most $pB/4$. In step 3, we further find an approximation of the cost of m-window tuples that are relevant for an optimal alignment and have large cost i.e. $\geq pB/4$.

1.1.2.1 Step 1

The windows of the input strings are constructed in two stages. In stage one, we follow a rather straightforward strategy similar to the one used in [15] and partition the (p, B) pseudorandom string into $\frac{n}{\beta}$ disjoint windows each of size β (except the right most one). Here $\beta = max(B, \sqrt{n})$. For the rest of the strings we generate a set of overlapping variable sized windows. If the distance threshold parameter is θ and the error tolerance parameter is ϵ, then for each adversarial strings we generate windows of size $\{(\beta - \theta\beta), (1+\epsilon)(\beta - \theta\beta), (1+\epsilon)^2(\beta - \theta\beta), \dots, (\beta + \theta\beta)\}$ and from starting indices in $\{1, \epsilon\theta\beta + 1, 2\epsilon\theta\beta + 1, \dots\}$. These windows are fed as an input to Step 2 of our algorithm.

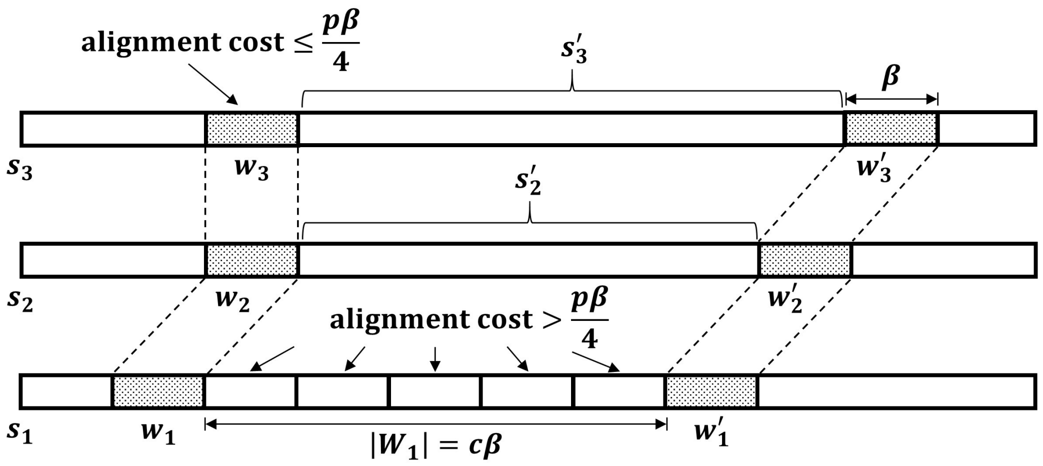
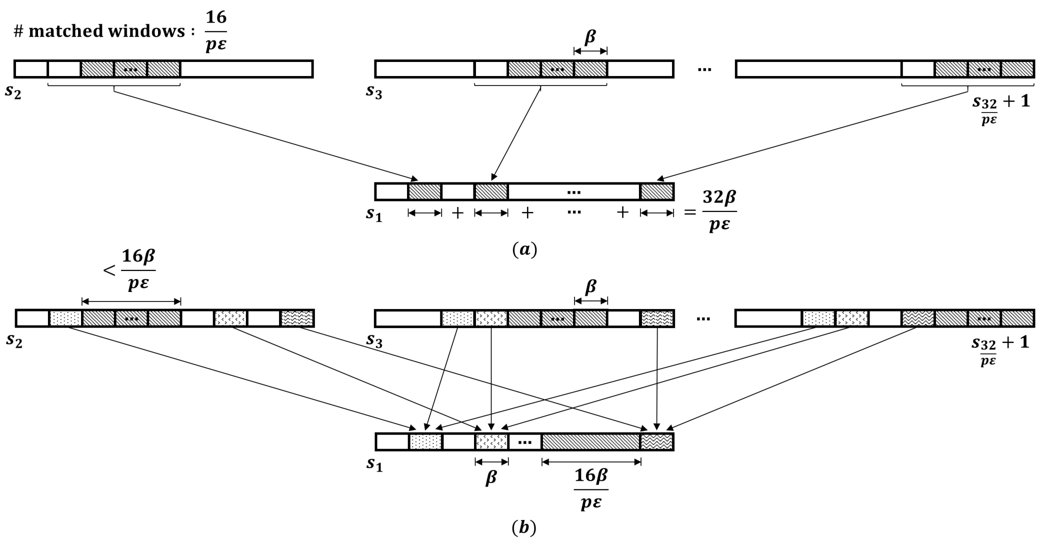


Figure 2 An example of large cost match.

The second stage is much more involved where the strategy significantly differs from the previous literature. The primary difference here is that instead of just considering a fixed length partition of the pseudorandom string (e.g. β in stage one), we take variable sizes that are multiple of β i.e. $\{\beta, 2\beta, \dots, n\}$ and the windows start from indices in $\{1, \beta+1, 2\beta+1, \dots\}$. Next for each adversarial string, we adaptively try to guess a set of useful substrings/windows that can be matched with the large cost windows of the pseudorandom string under the optimal alignment (we fix one for the analysis purpose). These guesses are guided by the unique low cost matches found in Step 2. Next for each such useful substring/window from the adversarial string and each length in $\{\beta, 2\beta, \dots, n\}$, we create a set of overlapping windows as described in stage one. Note the windows created in this stage are used as an input to Step 3 of the algorithm.

We explain the motivation behind the variable sized window partitioning for the pseudorandom string and the restricted window construction for the adversarial strings with an example(see Figure 2).

Let we are given three strings s_1, s_2, s_3 each of length n as input where s_1 is (p, B)-pseudorandom. We divide all three strings in windows (for simplicity assume these windows are disjoint) of size β. For the analysis purpose fix an optimal alignment where window w_1 is aligned with window w_2, w_3 and window w_1' is aligned with window w_2', w_3'. Moreover their costs are $\leq \frac{p\beta}{4}$. Hence we get an estimation of the alignment cost of these window tuples at Step 2. Also assume that all the windows appearing between w_1 and w_1' in s_1 has alignment cost $> \frac{p\beta}{4}$ but $\leq \frac{\beta}{2}$. Hence, for these windows we can not use a trivial maximum cost of β in order to get a below two approximation. Here we estimate the cost of these windows using Algorithm LargeAlign(). Now if we compute this estimation separately for each β size window between w_1 and w_1' in s_1, as there can be as many as $\sqrt{n}$ (assume $\beta = \sqrt{n}$) such windows and each call to algorithm LargeAlign() takes time $O(n^{m/4})$, the total running time will be $O(n^{3m/4})$. Thus to achieve the promised running time, instead of considering all small β size windows separately we consider the whole substring between w_1 and w_1' as one single window W_1 and compute its cost estimation. Observe as we do not have the prior information about the optimal alignment (and therefore w_1 and w_1') we try all possible lengths in $\beta, 2\beta, \dots, n$. The overall idea here is to use a window length, so that we can represent the whole substring with large optimal alignment distance lying between two windows having low cost unique match in the optimal alignment with a single window.



# matched windows : $\frac{16}{p\varepsilon}$
β
s_2
s_3
$s_{\frac{32}{p\varepsilon}+1}$
s_1
$= \frac{32\beta}{p\varepsilon}$
(a)
$< \frac{16\beta}{p\varepsilon}$
β
$\frac{16\beta}{p\varepsilon}$
(b)

Figure 3 (a) Each hatch window of s_1 is matched with $\frac{16}{p\epsilon}$ windows of one adversarial string. Deleting all these windows deletes $\frac{32\beta}{p\epsilon}$ characters from each string. (b) The optimal alignment deletes $< \frac{16\beta}{p\epsilon}$ characters from each string.

1.1.2.2 Step 2

In step 2, we start with a set of windows, each of size β, generated from strings $s_1, \ldots, s_m$ (assume s_1 to be the (p, B)-pseudorandom string). Our objective is to identify all m-window tuples that have optimal alignment distance $\le \frac{p\beta}{4}$. Here we use the fact that for every adversarial window there exists at most one window in the pseudorandom string at distance $\le \frac{p\beta}{4}$. We start the algorithm by computing for each window w from the pseudorandom string, and for each adversarial string s_j the set S_j^w containing all windows from s_j that are at distance $\le \frac{p\beta}{4}$ from w. Notice for two string case, if for a pseudorandom window w, $|S_j^w| \ge \frac{16}{p\epsilon}$ (i.e. many windows from s_j are close to w), then as in the optimal alignment at most one adversarial window from S_j^w can be matched with w with cost $\le \frac{p\beta}{4}$ and the rest of the windows from S_j^w have cost at least $\frac{p\beta}{4}$, the optimal cost for all the windows from S_j^w is $\ge \frac{\beta}{\epsilon}$. Thus even if we take a cost estimation β (maximum cost) for the pseudorandom window w this still gives a $(1+\epsilon)$ approximation of the alignment cost. For multiple strings we can not use a similar idea as for a pseudorandom window w we can not take a trivial cost estimation if for only one adversarial string, there are many windows which are close to w. Thus we need to device a new strategy.

We explain with an example. Consider k windows $w_1, \ldots, w_k$ (where $k \ge \frac{32}{p\epsilon}$) from the pseudorandom string such that for each w_j, there are $\frac{16}{p\epsilon}$ windows from the adversarial string s_{j+1} that are at distance $\le \frac{p\beta}{4}$ with w_j and for every other string there is exactly one window with distance $\le \frac{p\beta}{4}$. Here following the above argument if for each of the k pseudorandom windows, we take a trivial cost estimation of β then the total cost will be $k\beta \ge \frac{32\beta}{p\epsilon}$. Whereas the optimal cost can be $\frac{16\beta}{p\epsilon}$(check Figure 3 (a)).

Therefore for every window w_i of the pseudorandom string, we count the total number of windows in all adversarial strings that are at distance $\le \frac{p\beta}{4}$ and if the count is at least $\frac{16m}{p\epsilon}$, then we take a trivial cost estimation of $|w_i|$ for w_i. (Note here for two different windows from the pseudorandom string, the sets of close windows from the adversarial strings are disjoint.)

Otherwise if the count is small let $\mathcal{W}_j$ be the set of windows from string s_j that are close to w_i. Then we can bound $|\mathcal{W}_2|\times\cdots\times|\mathcal{W}_m| \leq [(|\mathcal{W}_2|+\cdots+|\mathcal{W}_m|)/(m-1)]^{m-1} \leq (\frac{32}{p\epsilon})^m = \tilde{O}_m(1)$ and for each choice of m-window tuples in $w_i \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_m$, we find the exact alignment cost in time $\tilde{O}_m(\beta^m)$. As there are n/β disjoint windows in the pseudorandom string we can bound the total running time by $\tilde{O}_m(n\beta^{m-1})$.

1.1.2.3 Step 3

In this step, our objective is to find a cost estimation for all the windows of the pseudorandom string that have large alignment cost in the optimal alignment. Consider Figure 2 (consider $m = 3$), and let W_1 be such a window form s_1. Also assume that it can not be extended to left or right, i.e. both w_1 and w_1' have small cost match which we have already identified in Step 2. Let the length of W_1 is $c\beta$ where $c \geq 1$ and assume no β length window of W_1 has small cost match. Note, we have the assurance that window W_1 is generated in Step 1. Next to find the match of W_1, instead of checking whole s_2 and s_3, we consider the substring between w_2 and w_2' in s_2 and w_3 and w_3' in s_3. Now if the sum of the length of all the substrings is large (i.e. $|s_2'| + |s_3'| \geq 5m|W_1| = 15|W_1|$), we can claim that the cost of W_1 in the optimal alignment is very large and we can take a trivial cost estimation of $|W_1|$. Otherwise, we can ensure that the total choices for m-window tuples that need to be evaluated for W_1 is at most $\tilde{O}_m(1)$ and for each of them, we calculate a below 2 approximation of the cost using Algorithm LargeAlign(). In the algorithm, as we don't know the optimal alignment, for every window of the pseudorandom string generated in Step 1 stage two, we assume it to be large cost and check whether the β length window appearing just before and after this window has a small cost unique match (notice this observation is very crucial to decide the maximal length of any large cost window). If not we discard it and consider a larger window. Otherwise we use LargeAlign() to find its cost estimation.

Overall in Step 2 and 3, we ensure that for every window of the pseudorandom string, our algorithm provides < 2 cost approximation. Moreover if the window has small cost match then in Step 2 we evaluate the cost of at most $\tilde{O}_m(1)$ m-window tuples where each cost estimation takes time $\tilde{O}_m(\beta^m)$ (as the window size is β) and otherwise if the cost is large we use algorithm LargeAlgin() that computes an approximation in time $\tilde{O}_m(n^{m/2})$ (here the window length can be as large as n). As the number of windows generated for string s_1 is polynomial in n, taking $\beta = max(B, \sqrt{n})$ we get the required running time bound.

Organization

In Section 2, we give the algorithm for minimizing alignment distance when the distance is large. Section 3 provides the details of the below-2 approximation algorithm for alignment distance when one string is (p, B) pseudorandom. In Section 4, we give our $\frac{\lambda}{2+\epsilon}$ approximation algorithm for multi-sequence LCS.

2 Below-2 Approximation for Multi-sequence Alignment Distance

In this section, we provide an algorithm LargeAlign() that given m strings $s_1, \ldots, s_m$ each of length n, such that $\mathcal{A}(s_1, \ldots, s_m) = \theta n$, where $\theta \in (0, 1)$ computes a $(2 - \frac{3\theta}{16} + \epsilon)$ approximation of $\mathcal{A}(s_1, \ldots, s_m)$ in time $\tilde{O}_m(n^{\lfloor m/2 \rfloor + 2})$. Notice when $\theta = \Omega(1)$, this implies a below 2 approximation of $\mathcal{A}(s_1, \ldots, s_m)$.

▶ **Theorem 5.** *Given m strings $s_1, \ldots, s_m$ of length n over some alphabet set Σ such that $\mathcal{A}(s_1, \ldots, s_m) = \theta n$, where $\theta \in (0,1)$, there exists an algorithm that for any arbitrary small constant $\epsilon > 0$ computes a $(2 - \frac{3\theta}{16} + \epsilon)$ approximation of $\mathcal{A}(s_1, \ldots, s_m)$ in time $\tilde{O}_m(n^{\lfloor m/2 \rfloor + 2})$. Moreover, there exists an algorithm that computes a $c(1 - \frac{3\theta}{32} + \epsilon)$ approximation of $\mathcal{A}(s_1, \ldots, s_m)$ in time $\tilde{O}_m(n^{\lceil m/c \rceil + 2})$.*

As we do not have any prior knowledge of θ, instead of proving the theorem directly, we solve the following gap version for a given fixed threshold θ. We define the gap version as follows.

GapMultiAlignDist$(s_1, \ldots, s_m, \theta, c)$. Given m strings $s_1, \ldots, s_m$ of length n over some alphabet set Σ, $\theta \in (0,1)$ and a constant $c > 1$, decide whether $\mathcal{A}(s_1, \ldots, s_m) \leq \theta n$ or $\mathcal{A}(s_1, \ldots, s_m) > c\theta n$. More specifically if $\mathcal{A}(s_1, \ldots, s_m) \leq \theta n$ we output 1, else if $\mathcal{A}(s_1, \ldots, s_m) > c\theta n$ we output 0, otherwise output any arbitrary answer.

▶ **Theorem 6.** *Given m strings $s_1, \ldots, s_m$ of length n over some alphabet set Σ and a parameter $\theta \in (0,1)$, there exists an algorithm that computes GapMultiAlignDist$(s_1, \ldots, s_m, \theta, (2 - \frac{3\theta}{16}))$ in time $\tilde{O}_m(n^{\lfloor m/2 \rfloor + 2})$.*

Proof of Theorem 5 from Theorem 6. Let us consider an arbitrary small constant $\epsilon' > 0$, and fix a sequence of parameters $\theta_0, \theta_1, \ldots$ as follows: for $i = 0, 1, \ldots, \log_{1+\epsilon'} n$, $\theta_i = 1/(1+\epsilon')^i$. Find the largest i such that GapMultiAlignDist$(s_1, \ldots, s_m, \theta_i, (2 - \frac{3\theta}{16})) = 1$. Let $\mathcal{A}(s_1, \ldots, s_m) = \theta n$. Then there exists a θ_i, such that $\theta_i n \geq \theta n > \theta_{i+1} n$ as $\theta_i = (1+\epsilon')\theta_{i+1}$. In this case the algorithm outputs a value at most $(2 - \frac{3\theta}{16})\theta_i n \leq (2 - \frac{3\theta}{16} + \epsilon'(2 - \frac{3\theta}{16}))\theta n$. As $\theta \geq 1/n$, by appropriately scaling ϵ', we get the desired running time of Theorem 5. ◀

The rest of the section is dedicated towards proving Theorem 6. Before providing the the algorithm for computing $GapMultiAlignDist(s_1, \ldots, s_m, \theta_i, (2 - \frac{3\theta}{16}))$, we first outline another two algorithms that will be used as subroutines in our main algorithm.

2.1 Finding Alignment with Maximum Deletion Similarity

Given m strings $s_1, \ldots, s_m$ of length n, a set $S \subseteq [n]$ and a parameter $0 \leq d \leq n$, our objective is to compute an alignment σ_n of $s_1, \ldots, s_m$ with alignment cost at most d such that $|\bar{\sigma}_n(s_1) \cap S|$ is maximized.

▶ **Theorem 7.** *Given m strings $s_1, \ldots, s_m$, each of length n, a set $S \subseteq [n]$ and a parameter $0 \leq d \leq n$, there exists an algorithm that computes a common alignment σ_n such that $\sum_{k \in [m]} |\bar{\sigma}_n(s_k)| \leq dm$ and $|\bar{\sigma}_n(s_1) \cap S|$ is maximized in time $\tilde{O}(2^m mn^{m+1})$.*

We design an algorithm MaxDelSimilarAlignment() that uses dynamic programming to compute σ_n. We defer the details to the full version.

If $d = \theta n$ where $\theta \in (0,1)$, then as we know from every string no more than θn characters will be deleted, using [30] we can claim the following.

▶ **Corollary 8.** *Given m strings $s_1, \ldots, s_m$, each of length n, a set $S \subseteq [n]$ and a parameter $d = \theta n$, where $\theta \in (0,1)$ there exists an algorithm that computes a common alignment σ_n such that $\sum_{k \in [m]} |\bar{\sigma}_n(s_k)| \leq dm$ and $|\bar{\sigma}_n(s_1) \cap S|$ is maximized in time $\tilde{O}(2^m \theta^m mn^{m+1})$.*

2.2 Finding Alignment with Minimum Deletion Similarity

Given m strings $s_1, \dots, s_m$ of length n, q sets $S_1, \dots, S_q \subseteq [n]$ and a parameter $0 \le d \le n$, our objective is to compute an alignment σ_n of $s_1, \dots, s_m$ with alignment cost at most d such that $| \cup_{j\in[q]} (\bar{\sigma}_n(s_1) \cap S_j)|$ is minimized.

▶ **Theorem 9.** *Given m strings $s_1, \dots, s_m$, each of length n, q sets $S_1, \dots, S_q \subseteq [n]$ and a parameter $0 \le d \le n$, there exists an algorithm that computes a common alignment σ_n such that $\sum_{k\in[m]} |\bar{\sigma}_n(s_k)| \le dm$ and $| \cup_{j\in[q]} (\bar{\sigma}_n(s_1) \cap S_j)|$ is minimized in time $\tilde{O}(2^m mn^{m+1})$.*

We design an algorithm MinDelSimilarAlignment() that uses dynamic programming just like the one used in finding alignment with maximum deletion similarity to compute σ_n. We defer the details to the full version. If $d = \theta n$ where $\theta \in (0, 1)$, again using [30] we can claim the following. Note in this case we only need to compute $\theta^m n^m$ entries in the dynamic program table.

▶ **Corollary 10.** *Given m strings $s_1, \dots, s_m$, each of length n, q sets $S_1, \dots, S_q \subseteq [n]$ and a parameter $0 \le d \le \theta n$, there exists an algorithm that computes a common alignment σ_n such that $\sum_{k\in[m]} |\bar{\sigma}_n(s_k)| \le dm$ and $| \cup_{j\in[q]} (\bar{\sigma}_n(s_1) \cap S_j)|$ is minimized in time $\tilde{O}(2^m \theta^m mn^{m+1})$.*

2.3 Algorithm for $(2 - \frac{3\theta}{16} + \epsilon)$-approximation of $\mathcal{A}(s_1, \dots, s_m)$

To compute the value of GapMultiAlignDist$(s_1, \dots, s_m, \theta, (2 - \frac{3\theta}{16}))$ the procedure GapMultiAlignDist$(s_1, \dots, s_m, \theta)$ calls procedure MultiAlign$(s_1, \dots, s_m, \theta)$ that returns a string σ. If σ is a null string it outputs 0 and otherwise it outputs 1.

We show if $\mathcal{A}(s_1, \dots, s_m) \le \theta n$, MultiAlign$(s_1, \dots, s_m, \theta)$ computes a common subsequence σ such that $|\bar{\sigma}(s_1)| \le (2 - \frac{3\theta}{16})\theta n$ and otherwise if $\mathcal{A}(s_1, \dots, s_m) > (2 - \frac{3\theta}{16})\theta n$ it computes a null string. Next we describe Algorithm MultiAlign$(s_1, \dots, s_m, \theta)$. It starts by partitioning the input strings into two groups G_1 and G_2 where G_1 contains the strings $s_1, \dots, s_{\lceil m/2 \rceil}$ and G_2 contains the strings $s_1, s_{\lceil m/2 \rceil+1}, \dots, s_m$.

Assume $\mathcal{A}(S_1, \dots, s_m) \le \theta n$. Next we state an observation that is used as one of the key elements to conceptualize our algorithm. Any common subsequence of $s_1, \dots, s_m$ is indeed a common subsequence of G_1. Therefore, as $\mathcal{A}(s_1, \dots, s_m) \le \theta n$, if we can enumerate all common subsequences of G_1 of length at least $n - \theta n$, we generate the optimal alignment as well. Notice after generating each common subsequence of G_1, it can be checked whether it is a common subsequence of G_2 or not. The main hurdle here is that enumerating all common subsequences of G_1 of length at least $n - \theta n$ is time consuming.

We overcome this barrier by designing Algorithm EnumerateAlignments$(s_1, \dots, s_m, \theta)$ that generates k (where $k = O(1/\theta)$) different sets $L_1, \dots, L_k$ where $L_j \subseteq [n]$, $|L_j| \le \theta n$ and each L_j corresponds to a common subsequence σ_j of G_1 such that $L_j = \bar{\sigma}_j(s_1)$. Moreover, we can ensure that either $\exists j \in [k]$ where $|L_j| \le \frac{3\theta n}{4}$ or for any common subsequence σ of G_1 with $\bar{\sigma}(s_1) \le \theta n$ there exists a L_i where $|\bar{\sigma}(s_1) \cap L_i| \ge \frac{3\theta^2 n}{16}$.

Algorithm EnumerateAlignments() starts by computing a LCS σ_1 of G_1 such that $|\bar{\sigma_1}(s_1)| \le \theta n$. Let $L_1 = \bar{\sigma_1}(s_1)$. If $|L_1| \le \frac{3\theta n}{4}$ return L_1. Otherwise it calls the algorithm MinDelSimilarAlignment() to compute a common subsequence σ_2 of G_1 of cost at most θn such that $L_2 \cap L_1$, where $L_2 = \bar{\sigma_2}(s_1)$ is minimized. At the ith step given $i - 1$ sets $L_1, \dots, L_{i-1}$, the algorithm computes an alignment σ_i of G_1 with L_i being the set of indices of unaligned characters of s_1 such that the intersection of L_i with $\cup_{j\in[i-1]} L_j$ is minimized. The algorithm continues with this process until it reaches a round k such that $| \cup_{i\in[k-1]} (L_k(s_1) \cap L_i(s_1))| \ge \frac{\theta^2 n(k-1)}{4}$. Let $L_1, \dots, L_k$ be the sets generated. Output all these sets.

Next for each L_i returned by Algorithm EnumerateAlignments(), call the algorithm MaxDelSimilarAlignment() to find an alignment σ' of G_2 of cost at most θn such that the intersection of L_i and $L'_i = \bar{\sigma}'(s_1)$ is maximized. If $|L_i \cup L'_i| \leq (2 - \frac{3\theta}{16})\theta n$, define $\sigma = s_1[i_1] \circ \dots \circ s_1[i_p]$ where, $[n] \setminus (L_i \cup L'_i) = \{i_1, \dots, i_p\}$. Output σ.

We now prove two crucial lemmas to establish the correctness.

▶ **Lemma 11.** *Given strings $s_1, \dots, s_{|G_1|}$ of length n such that $\mathcal{A}(s_1, \dots, s_{|G_1|}) \leq \theta n$, where $\theta \in (0,1)$, there exists an algorithm that computes $k \leq 4/\theta$, different sets $L_1, \dots, L_k \subseteq [n]$ each of size at most θn such that $\forall j \in [k]$, there exists a common subsequence σ_j of G_1 where, $L_j = \bar{\sigma}_j(s_1)$ and one of the following is true.*

1. *$\exists j \in [k]$ such that $|L_j| \leq \frac{3\theta n}{4}$.*
2. *For any common subsequence σ of G_1 with $\bar{\sigma}(s_1) \leq \theta n$ there exists a L_i, where $|\bar{\sigma}(s_1) \cap L_i| \geq \frac{3\theta^2 n}{16}$. The running time of the algorithm is $\tilde{O}(2^m mn^{|G_1|+1})$.*

Proof. Let $\sigma' = LCS(s_1, \dots, s_{|G_1|})$. If $|\sigma'| \geq n - \frac{3\theta n}{4}$, then $|\bar{\sigma'}(s_1)| \leq \frac{3\theta n}{4}$ and we satisfy condition 1. Otherwise assume $|\bar{\sigma'}(s_1)| > \frac{3\theta n}{4}$. Let $L_1, \dots, L_k$ be the sets returned by Algorithm EnumerateAlignments(). By construction, for each L_i there exists a common subsequence σ_i of G_1 where $L_i = \bar{\sigma}_i(s_1)$. Note every L_i has size at least $\frac{3\theta n}{4}$. Moreover if the algorithm does not terminate at round i, then $|L_i(s_1) \setminus \{L_1(s_1) \cup \dots \cup L_{i-1}(s_1)\}| \geq \frac{3\theta n}{4} - \frac{\theta^2 n(i-1)}{4}$. Hence after k steps we have

$$
\begin{aligned}
|\cup_{i\in[k]} L_i(s_1)| &\geq \frac{3\theta n}{4} + (\frac{3\theta n}{4} - \frac{\theta^2 n}{4}) + \dots + (\frac{3\theta n}{4} - \frac{(k-1)\theta^2 n}{4}) \\
&= \frac{3k\theta n}{4} - \frac{\theta^2 n}{4}(1 + 2 + \dots + (k-1)) \\
&= \frac{3k\theta n}{4} - \frac{k(k-1)\theta^2 n}{8} \\
&> \frac{3k\theta n}{4} - \frac{k^2\theta^2 n}{8}
\end{aligned}
$$

Substituting $k = 4/\theta$, we get $|\cup_{i\in[k]} L_i(s_1)| > n$. Now if the algorithm stops at round $i < 4/\theta$, then we know for each common subsequence σ of G_1 of length at least $n - \theta n$ if $\sigma \notin \{L_1, \dots, L_{i-1}\}$, $|\cup_{j\in[i-1]} (\sigma(s_1) \cap L_j(s_1))| \geq \frac{(i-1)\theta^2 n}{4}$. Hence there exists at least one $j \in [i-1]$ such that $|L_j(s_1) \cap \sigma(s_1)| \geq \frac{\theta^2 n}{4}$. Otherwise if the algorithm runs for $4/\theta$ rounds then $\cup_{i\in[4/\theta]} L_i(s_1) = [n]$. Hence for each common subsequence σ of cost in $[\frac{3\theta n}{4}, \theta n]$, $\bar{\sigma}(s_1)$ will have intersection at least $\frac{3\theta^2 n}{16}$ with at least one L_i.

As $|k| \leq 4/\theta$ the algorithm runs for at most $4/\theta$ rounds where at the ith round it calls MinDelSumilarAlignment() with strings in G_1, and sets $L_1, \dots, L_{i-1}$ (where $i \leq 4/\theta$) and parameter θn. By Corollary 10 each call to MinDelSumilarAlignment() takes time $\tilde{O}(2^{|G_1|}\theta^{|G_1|}|G_1|n^{|G_1|+1})$. Hence the total running time taken is $\tilde{O}(2^{|G_1|}|G_1|n^{|G_1|+1})$. ◀

We set $|G_1| = \lceil \frac{m}{2} \rceil \leq \lfloor m/2 \rfloor + 1$ to obtain a running time of $\tilde{O}(2^{\lfloor m/2 \rfloor+1}mn^{\lfloor m/2 \rfloor+2})$.

▶ **Lemma 12.** *Given $\mathcal{A}(s_1, \dots, s_m) \leq \theta n$, Algorithm MultiAlign($s_1, \dots, s_m, \theta$) generates a string σ, such that σ is a common sequence of $s_1, \dots, s_m$ and the alignment cost of σ is at most $(2 - \frac{3\theta}{16})\theta n$. Moreover the running time of the algorithm is $\tilde{O}_m(n^{\lfloor \frac{m}{2} \rfloor+2})$.*

Proof. Let η be some LCS of $s_1, \dots, s_m$ such that $|\bar{\eta}(s_1)| \leq \theta n$. First assume $\mathcal{A}(G_1) \leq \frac{3\theta n}{4}$. Then Algorithm EnumerateAlignments() computes a LCS σ_1 of G_1 and returns the set $L_1 = |\bar{\sigma_1}(s_1)|$ where $|L_1| \leq \frac{3\theta n}{4}$ to Algorithm MultiAlign(). Next Algorithm MultiAlign() calls MaxDelSimilarAlignmemnt($G_2, L_1, \theta n$) which returns a set $L'_1 = \bar{\sigma}'(s_1)$, where σ'

is a common subsequence of G_2 and $|L'_1| \leq \theta n$. Notice $\sigma \leftarrow s_1[i_1] \circ \cdots \circ s_1[i_p]$ (where, $[n] \setminus (L_i \cup L'_i) = \{i_1, \dots, i_p\}$) is a common subsequence of $s_1, \dots, s_m$ and $|\bar{\sigma}(s_1)| \leq (|L_1| + |L'_1|) \leq (\frac{3\theta n}{4} + \theta n) = (2 - \frac{1}{4})\theta n \leq (2 - \frac{\theta}{4})\theta n$ as $\theta \in (0, 1)$. Hence Algorithm MultiAlign() computes a common subsequence σ of $s_1, \dots, s_m$ such that the alignment cost of σ is at most $(2 - \frac{\theta}{4})\theta n$.

Next assume $\mathcal{A}(G_1) > \frac{3\theta n}{4}$. Then by Lemma 11, Algorithm EnumerateAlignments() computes a set $L_i = \bar{\sigma}_i(s_1)$ where σ_i is a common subsequence of G_1, $|L_i| \leq \theta n$ and $L_i \cap \bar{\eta}(s_1) \geq \frac{3\theta^2 n}{16}$. Notice as η is a common subsequence of G_2, when Algorithm MultiAlign() calls MaxDelSimilarAlignmemnt$(G_2, L_i, \theta n)$, it returns a set $L'_1 = \bar{\sigma'}(s_1)$, where σ' is a common subsequence of G_2, $|L'_1| \leq \theta n$ and $|L_i \cap L'_1| \geq \frac{3\theta^2 n}{16}$. Therefore $|L_i \cup L'_1| \leq \theta n + \theta n - \frac{3\theta^2 n}{16} = (2 - \frac{3\theta n}{16})\theta n$, and Algorithm MultiAlign() computes a common subsequence σ of $s_1, \dots, s_m$ such that the alignment cost of σ is at most $(2 - \frac{3\theta}{16})\theta n$.

Next we analyze the running time of Algorithm MultiAlign(). First we compute $LCS(G_1)$ which takes time $\tilde{O}(2^{\lfloor m/2 \rfloor + 1} n^{\lfloor m/2 \rfloor + 1})$ using the classic dynamic program algorithm (note $|G_1| = \lceil \frac{m}{2} \rceil \leq \lfloor \frac{m}{2} \rfloor + 1$). Next we call $EnumerateAlignments(s_1, \dots, s_{m/2}, \theta)$. By Lemma 11 this takes time $\tilde{O}(2^{\lfloor m/2 \rfloor + 1} m n^{\lfloor m/2 \rfloor + 2})$. Moreover it returns at most $O(1/\theta)$ sets and for each of them Algorithm MultiAlign() calls $MaxDelSimilarAlignment()$ on G_2. As $|G_2| \leq \lfloor \frac{m}{2} \rfloor + 1$ and each set has size at most θn, each call takes time $\tilde{O}(2^{\lfloor m/2 \rfloor + 1} \theta^{\lfloor m/2 \rfloor + 1} m n^{\lfloor m/2 \rfloor + 2})$. Hence total time taken is $\tilde{O}(2^{\lfloor m/2 \rfloor + 1} m n^{\lfloor m/2 \rfloor + 2})$. Next each union of L_i and L'_i and corresponding σ can be computed in time $O(n)$. Hence the running time of Algorithm MultiAlign() is $\tilde{O}(2^{\lfloor m/2 \rfloor + 1} m n^{\lfloor m/2 \rfloor + 2}) = \tilde{O}_m(n^{\lfloor \frac{m}{2} \rfloor + 2})$. ◀

▶ **Lemma 13.** *GapMultiAlignDist$(s_1, \dots, s_m, \theta)$ computes GapMultiAlignDist$(s_1, \dots, s_m, \theta, (2 - \frac{3\theta}{16}))$ in time $\tilde{O}_m(n^{\lfloor \frac{m}{2} \rfloor + 2})$.*

Proof. First assume the case where $\mathcal{A}(s_1, \dots, s_m) \leq \theta n$. from Lemma 12 we have Algorithm MultiAlign() returns a common subsequence σ of $s_1, \dots, s_m$ such that the alignment cost of σ is at most $(2 - \frac{3\theta}{16})\theta n$. Hence, Algorithm GapMultiAlignDist() outputs 1. Next assume $\mathcal{A}(s_1, \dots, s_m) > (2 - \frac{3\theta}{16})\theta n$. In this case in Algorithm MultiAlign(), for each set $L_i \in \mathcal{E}$, $|L_i \cup L'_i| > (2 - \frac{3\theta}{16})\theta n$. Hence Algorithm MultiAlign() returns a null string and Algorithm GapMultiAlignDist() outputs 0. The bound on the running time is directly implied by the running time bound of Algorithm MultiAlign(). ◀

3 Below-2 Approximation for Multi-sequence Alignment Distance with One Pseudorandom String

In the last section we present an algorithm that given m strings, computes a truly below 2 approximation of the optimal alignment distance of the input strings provided the distance is large i.e. $\Omega(n)$. In this section we use this algorithm as a black box and show given a (p, B)-pseudorandom string s_1, and $m - 1$ adversarial strings $s_2, \dots, s_m$, there exists an algorithm that for any arbitrary small constant $\epsilon > 0$ computes $(2 - \frac{3p}{512} + \epsilon)$ approximation of $\mathcal{A}(s_1, \dots, s_m)$ in time $\tilde{O}_m(n\beta^{m-1} + n^{\lfloor m/2 \rfloor + 3})$. Here $\beta = max(B, \sqrt{n})$. Notice as the approximation factor is independent of $\theta = \frac{\mathcal{A}(s_1, \dots, s_m)}{n}$, we can assure truly below-2 approximation of the alignment cost for any distance regime. Formally we show the following.

▶ **Theorem 14** (2). *Given a (p, B)-pseudorandom string s_1, and $m - 1$ adversarial strings $s_2, \dots, s_m$ each of length n, there exists an algorithm that for any arbitrary small constant $\epsilon > 0$ computes $(2 - \frac{3p}{512} + 89\epsilon)$ approximation of $\mathcal{A}(s_1, \dots, s_m)$ in time $\tilde{O}_m(n\beta^{m-1} + n^{\lfloor m/2 \rfloor + 3})$. Here $\beta = max(B, \sqrt{n})$.*

The details of the algorithm and the analysis are provided in the full version.

4 $\frac{\lambda}{2+\epsilon}$-approximation for Multi-sequence LCS

In this section we provide an algorithm that given m strings $s_1, \ldots, s_m$ each of length n, such that $\mathcal{L}(s_1, \ldots, s_m) = \lambda n$, where $\lambda \in (0, 1)$ computes an $\frac{\lambda}{2+\epsilon}$ approximation of $\mathcal{L}(s_1, \ldots, s_m)$. The algorithm is nearly identical to the algorithm described in Section 2, and in fact slightly simpler which helps us to improve the running time further. In particular, we get the following theorem.

▶ **Theorem 15** (4). *For any constant $\epsilon > 0$, given m strings $s_1, \ldots, s_m$ of length n over some alphabet set Σ such that $\mathcal{L}(s_1, \ldots, s_m) = \lambda n$, where $\lambda \in (0, 1)$, there exists an algorithm that computes an $\frac{\lambda}{2+\epsilon}$ approximation of $\mathcal{L}(s_1, \ldots, s_m)$ in time $\tilde{O}_m(n^{\lfloor m/2 \rfloor + 1} + mn^2)$.*

Since we do not have any prior knowledge of λ, we solve a gap version: given m strings $s_1, \ldots, s_m$ of length n over some alphabet set Σ, $\lambda \in (0, 1)$ and a constant $c > 1$, the objective is to decide whether $\mathcal{L}(s_1, \ldots, s_m) \geq \lambda n$ or $\mathcal{L}(s_1, \ldots, s_m) < \frac{\lambda^2 n}{c}$. More specifically if $\mathcal{L}(s_1, \ldots, s_m) \geq \lambda n$ we output 1, else if $\mathcal{L}(s_1, \ldots, s_m) < \frac{\lambda^2 n}{c}$ we output 0 otherwise output any arbitrary answer. We design an algorithm that decides this gap version for $c = 2$. This immediately implies an $(\frac{\lambda}{2+\epsilon})$ approximation of $LCS(s_1, s_2, .., s_m)$ following a similar logic of going from Theorem 6 to Theorem 5. We now prove Theorem 4.

4.1 Algorithm for $\frac{\lambda}{2+\epsilon}$-approximation of $LCS(s_1, \ldots, s_m)$

We partition the input strings into two groups G_1 and G_2 where G_1 contains the strings $s_1, \ldots, s_{\lceil m/2 \rceil}$ and G_2 contains the strings $s_{\lceil m/2 \rceil + 1}, \ldots, s_m$. Next compute a longest common subsequence L_1 of G_1 with $|L_1| \geq \lambda n$. Remove all aligned characters of s_1 in L_1. We represent the modified s_1 by $s_1^{[n] \setminus L_1(s_1)}$: string s_1 restricted to the characters with indices in $[n] \setminus L_1(s_1)$. Compute an LCS L_2 of $s_1^{[n] \setminus L_1(s_1)}, s_2, \ldots, s_{\lceil m/2 \rceil}$. At ith step given $i-1$ common subsequences $L_1, \ldots, L_{i-1}$, we compute an LCS L_i of $s_1^{[n] \setminus \cup_{j \in [i-1]} L_j(s_1)}, s_2, \ldots, s_{\lceil m/2 \rceil}$. We continue this process until it reaches a round k such that $|L_k| < \lambda n - \frac{\lambda^2 n(k-1)}{2}$. Let $L_1, \ldots, L_k$ be the sequences generated.

Next for each L_i returned, we compute a longest common subsequence L'_i of $L_i, s_{\lceil m/2 \rceil + 1}, \ldots, s_m$. If there exists an $i \in [k]$ such that $|L'_i| \geq \frac{\lambda^2 n}{2}$, output L'_i.

▶ **Lemma 16.** *Given strings $s_1, \ldots, s_{|G_1|}$ of length n and a parameter λn as input, where $\lambda \in (0, 1]$, there exists a set of $k \leq 2/\lambda$ different common subsequences $L_1, \ldots, L_k$ of $s_1, \ldots, s_{|G_1|}$ each of length at least $\lambda n - \lambda^2 n(k-1)/2$ such that for any common subsequence σ of $s_1, \ldots, s_m$ of length at least λn there exists a L_i, where $|\sigma(s_1) \cap L_i(s_1)| \geq \frac{\lambda^2 n}{2}$. These k subsequences can be computed in time $\tilde{O}_m(n^{|G_1|} + mn^2)$.*

Proof. Given k subsequences $L_1, L_2, \ldots, L_k$ of s_1 such that for each $i \in [k]$, $|L_i| \geq \lambda n - \frac{n\lambda^2(k-1)}{4}$, and $L_1(s_1), L_2(s_1), \ldots$ are disjoint, we have

$$\begin{aligned} |\cup_{i \in [k]} L_i(s_1)| &\geq \lambda n + (\lambda n - \frac{\lambda^2 n}{2}) + \cdots + (\lambda n - \frac{(k-1)\lambda^2 n}{2}) \\ &= k\lambda n - \frac{\lambda^2 n}{2}(1 + 2 + \cdots + (k-1)) \\ &> k\lambda n - \frac{k^2 \lambda^2 n}{4} \end{aligned}$$

Substituting $k = 2/\lambda$ we get $|\cup_{i \in [k]} L_i(s_1)| > n$. Now if we compute $L_1, \ldots, L_j$ and $j < 2/\lambda$, then we know for each common subsequence σ of $s_1, \ldots, s_m$ with length at least λn if $\sigma \notin \{L_1, \ldots, L_{j-1}\}$, then $|\cup_{i \in [j-1]} (\sigma(s_1) \cap L_j(s_1))| \geq \frac{(j-1)\lambda^2 n}{2}$. Hence there exists at least one $i \in [j-1]$ such that $|L_i(s_1) \cap \sigma(s_1)| \geq \frac{\lambda^2 n}{2}$. Otherwise if the algorithm runs for $2/\lambda$ rounds then $\cup_{j \in [2/\lambda]} L_j(s_1) = [n]$. Then for any common subsequence σ of $s_1, \ldots, s_m$ with length at least λn, σ will have an intersection at least $\frac{\lambda^2 n}{2}$ with at least one L_i.

As $|k| \leq 2/\lambda$ the algorithm runs for at most $2/\lambda$ rounds where at each round it computes the LCS of G_1 strings such that $\sum_j |L_j| \leq n$ where L_js are pairwise disjoint. This can be performed using Theorem 17 in $\tilde{O}_m(n^{|G_1|} + mn\lambda) = \tilde{O}_m(n^{|G_1|} + mn^2)$ time. ◀

By setting $|G_1| = \lceil m/2 \rceil \leq \lfloor \frac{m}{2} \rfloor + 1$, we get a running time of $\tilde{O}_m(n^{\lfloor m/2 \rfloor + 1} + mn^2)$. Now by Lemma 16 if σ is an LCS of $s_1, s_2, .., s_m$, then there exists a L_i such that $|L_i(s_1) \cap \sigma(s_1)| \geq \frac{\lambda^2 n}{2}$. Thus, when we compute the LCS of $L_i(s_1), s_{\lceil m/2 \rceil + 1}, .., s_m$, we are guaranteed to return a common subsequence of $s_1, s_2, ..., s_m$ of length at least $\frac{\lambda^2 n}{2}$. Hence, taking the right choice of λ following the gap version we get the claimed approximation bound. Using Theorem 17, the running time to compute a common subsequence of $L_j(s_1), s_{\lceil m/2 \rceil + 1}, .., s_m$ for all j is $\tilde{O}_m(n^{\lfloor m/2 \rfloor + 1} + mn^2)$. This completes the proof of Theorem 4. ◀

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 59–78, 2015.

2 Alexandr Andoni and Robert Krauthgamer. The smoothed complexity of edit distance. *ACM Transactions on Algorithms (TALG)*, 8(4):1–25, 2012.

3 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 377–386, 2010.

4 Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, 2020.

5 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM J. Comput.*, 41(6):1635–1648, 2012.

6 Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *45th Symposium on Foundations of Computer Science, FOCS 2004*, pages 550–559, 2004.

7 Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. A sublinear algorithm for weakly approximating edit distance. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 316–324, 2003.

8 Tugkan Batu, Funda Ergün, and Süleyman Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, pages 792–801, 2006.

9 Amey Bhangale, Diptarka Chakraborty, and Rajendra Kumar. Hardness of approximation of (multi-)lcs over small alphabet. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020*, volume 176, pages 38:1–38:16, 2020.

10 Guillaume Blin, Laurent Bulteau, Minghui Jiang, Pedro J Tejada, and Stéphane Vialette. Hardness of longest common subsequence for sequences with bounded run-lengths. In *Annual Symposium on Combinatorial Pattern Matching*, pages 138–148, 2012.

11 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1170–1189, 2018.

12 Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and LCS: beyond worst case. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 1601–1620, 2020.

13 Joshua Brakensiek and Aviad Rubinstein. Constant-factor approximation of near-linear edit distance in near-linear time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 685–698, 2020.

14 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 979–990, 2018.

15 Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. Does preprocessing help in fast sequence comparisons? In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 657–670, 2020.

16 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

17 Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

18 Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

19 Michal Koucký and Michael E. Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 699–712, 2020.

20 William Kuszmaul. Efficiently approximating edit distance between pseudorandom strings. In *Proceedings of the thirtieth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1165–1180. SIAM, 2019.

21 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, 1998.

22 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

23 David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, 1978.

24 Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

25 François Nicolas and Eric Rivals. Hardness results for the center and median string problems under the weighted and unweighted edit distances. *J. Discrete Algorithms*, 3(2-4):390–415, 2005.

26 Pavel A Pevzner. Multiple alignment, communication cost, and graph matching. *SIAM Journal on Applied Mathematics*, 52(6):1763–1779, 1992.

27 Aviad Rubinstein. Approximating edit distance, 2018.

28 Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for LCS and LIS with truly improved running times. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 1121–1145, 2019.

29 Julie D Thompson, Desmond G Higgins, and Toby J Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22:4673–4680, 1994.

30 Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.

31 Nuzzo R. Van Noorden R, Maher B. The top 100 papers. *Nature*, 2014.

A An $\tilde{O}(m^2k^m)$ algorithm for Alignment Distance of Multiple Sequences with $\tilde{O}(mn)$ Preprocessing

We first recall an algorithm developed in [30, 21, 22, 24] that computes edit distance between two strings in $O(n+k^2)$ time.

Warm-up: An $O(n+k^2)$ algorithm for Edit Distance

The well-known dynamic programming algorithm computes an $(n+1)\times(n+1)$ edit-distance matrix $D[0...n][0...n]$ where entry $D[i,j]$ is the edit distance, $ED(A^i,B^j)$ between the prefixes $A[1,i]$ and $B[1,j]$ of A and B, where $A[1,i]=a_1a_2...a_i$ and $B[1,j]=b_1b_2...b_j$. The following is well-known and easy to verify coupled with the boundary condition $D[i,0]=D[0,i]=i$ for all $i\in[0,n]$.

For all $i,j\in[0,n]$

$$D[i,j]=\min\begin{cases} D[i-1,j]+1 & \text{if } i>0;\\ D[i,j-1]+1 & \text{if } j>0;\\ D[i-1,j-1]+1(a_i\neq b_j) & \text{if } i,j>0.\end{cases}$$

The computation cost for this dynamic programming is $O(n^2)$. To obtain a significant cost saving when $ED(A,B)\le k << n$, the $O(n+k^2)$ algorithm works as follows. It computes the entries of D in a greedy order, computing first the entries with value 0, $1,2,...k$ respectively. Let diagonal d of matrix D, denotes all $D[i,j]$ such that $j=i+d$. Therefore, the entries with values in $[0,k]$ are located within diagonals $[-k,k]$. Now since the entries in each diagonal of D are non-decreasing, it is enough to identify for every $d\in[-k,k]$, and for all $h\in[0,k]$, the last entry of diagonal d with value h. The rest of the entries can be inferred automatically. Hence, we are overall interested in identifying at most $(2k+1)k$ such points. The $O(n+k^2)$ algorithm shows how building a suffix tree over a combined string $A\$B$ (where \$ is a special symbol not in Σ) helps identify each of these points in $O(1)$ time, thus achieving the desired time complexity.

Let $L^h(d)=\max\{i : D[i,i+d]=h\}$. The h-wave is defined by $L^h=\langle L^h(-k),...,L^h(k)\rangle$. Therefore, the algorithm computes L^h for $h=0,..k$ in the increasing order of h until a wave e is computed such that $L^e(0)=n$ (in that case $ED(A,B)=e$), or the wave L^k is computed in the case the algorithm is thresholded by k. Given L^{h-1}, we can compute L^h as follows.

Define

$$Equal(i,d)=\max_{q\ge i}\,(q\mid A[i,q]=B[i+d,q])$$

Then, $L^0(0)=Equal(0,0)$ and

$$L^h(d)=\max\begin{cases} Equal(L^{h-1}(d)+1,d) & \text{if } h-1\ge 0;\\ Equal(L^{h-1}(d-1),d) & \text{if } d-1\ge -k, h-1\ge 0;\\ Equal(L^{h-1}(d+1)+1,d) & \text{if } d+1, h+1\le k.\end{cases}$$

Using a suffix tree of the combined string $A\$B$, any $Equal(i,d)$ query can be answered in $O(1)$ time, and we get a running time of $O(n+k^2)$.

An $\tilde{O}(m^2k^m)$ algorithm for Alignment Distance of Multiple Sequences with $\tilde{O}(nm)$ Preprocessing

We now extend the above $\tilde{O}(n+k^2)$ algorithm to computing alignment distance of m strings. Recall that we are given m strings $s_1,s_2,...,s_m$ each of length n. The following is an $O(n^m)$ time-complexity dynamic programming to obtain the edit distance of m strings. We fill up an

m-dimensional dynamic programming matrix $D[[0,n]....[0,n]]$ where the entry $D[i_1,i_2,..,i_m]$ computes the edit distance among the prefixes $s_1[1,i_1], s_2[1,i_2],...,s_m[1,i_m]$. As a starting condition, we have $D[0,...,0]=0$. Let $\vec{e_i}=[0,0,..,\underbrace{1}_{i\text{th index}},0,0..]$, and $\mathbb{V}_j$ represents an m-dimensional vector $\langle j,j,j,...,j\rangle$. The dynamic programming is given by the following recursion. For all $i_1,i_2,...,i_m \in [0,n]$

$$D[i_1,i_2,...,i_m] = \min \begin{cases} D[\langle i_1,i_2,..,i_m\rangle - \vec{e_j}]+1 & \text{for all } j=1,2,..,m \\ & \text{if } \langle i_1,i_2,..,i_m\rangle - \vec{e_j} \geq \langle 0,0,...,0\rangle; \\ D[i_1-1,i_2-1,..,i_m-1] & \text{if } s_1[i_1]=s_2[i_2]=...=s_m[i_m] \text{ and} \\ & i_1,i_2,...,i_m > 0. \end{cases}$$

In order to compute $D[i_1,i_2,...,i_m]$, we can either delete an element $s_j[i_j]$, or align $s_1[i_1], s_2[i_2],..,s_m[i_m]$ if they all match. The time to compute each entry $D[i_1,i_2,...,i_m]$ is m. The overall running time is $O(mn^m)$.

Observation. In order to design an $\tilde{O}(m^2k^m)$ algorithm with preprocessing $\tilde{O}(mn)$, first observe that if $\mathcal{A}(s_1,s_2,...,s_m)\leq k$ then it is not possible that in the final alignment, we have m indices $i_1,i_2,...,i_m$ aligned to each other such that $\max|i_j-i_k|, j,k\in[1,2,..,m] > k$. Since all strings have equal length, this would imply a total number of deletions $> mk$, or $\mathcal{A}(s_1,s_2,...,s_m) > k$.

Algorithm. Let diagonal $\vec{d}$ of matrix D denotes an $(m-1)$ dimensional vector, and contains all $D[i_1,i_2,..i_m]$ such that $\langle i_2,i_3,..,i_m\rangle = \mathbb{V}_{i_1}+\vec{d}$. Let $D_k=\{\vec{d}\mid \max_j|d[j]|\leq k\}$. Then $|D_k|=(2k+1)^{m-1}$ since each entry $i_j\in i_i+\{-k,-k+1,..,0,1,...,k\}$, for $j=2,3,..,k$. We want to identify the entries with values in $[0,km]$ located within diagonals D_k.

We similarly define $L^h(d)=\max\{i : D[i,\mathbb{V}_i+\vec{d}]=h\}$. The h-wave is defined by $L^h=\{L^h(\vec{d})\mid \vec{d}\in D_k\}$. Therefore, the algorithm computes L^h for $h=0,..,km$ in the increasing order of h until a wave e is computed such that $L^e(\vec{0})=n$ (in that case $\mathcal{A}(A,B)=e$), or the wave L^{km} is computed in the case the algorithm is thresholded by k. Given L^{h-1}, we can compute L^h as follows.

Define

$$Equal(i,\vec{d}) = \max_{q\geq i}(q \mid s_1[i,q]=s_2[i+d[1],q]) = s_3[i+d[2],q] = ... = s_m[i+d[m-1],q]).$$

That is $Equal(i,\vec{d})$ computes the longest prefix of the first string starting at index i that can be matched to all the other strings following diagonal $\vec{d}$.

Next, we define the neighboring diagonals $N_1(\vec{d})$ and $N_2(\vec{d})$ of $\vec{d}$.

$$N_1(\vec{d})=\{\vec{d'}\mid ||d-d'||_1=1 \;\&\; \vec{d'}<\vec{d}\}.$$

$$N_2(\vec{d})=\{\vec{d'}=\vec{d}+\mathbb{V}_{+1}\}.$$

Then, $L^0(0)=Equal(0,\vec{0})$ and

$$L^h(\vec{d}) = \max\begin{cases} Equal(L^{h-1}(\vec{d'}\in N_1(\vec{d})),\vec{d}) & \text{if } \vec{d'}\in D_k \text{ , } h-1\geq 0; \\ Equal(L^{h-1}(\vec{d'}\in N_2(\vec{d}))+1,\vec{d}) & \text{if } \vec{d'}\in D_k,\ h-1\geq 0. \end{cases}$$

Next, we show that it is possible to preprocess s_i, $i=1,2,..m$ separately so that even then each $Equal(i,\vec{d})$ query can be implemented in $O(m\log n)$ time.

Preprocessing Algorithm

The preprocessing algorithm constructs $\log(n)+1$ hash tables for each string s. The ℓ-th hash table corresponds to window size 2^ℓ; we use a rolling hash function (e.g. Rabin fingerprint) to construct a hash table of all contiguous substrings of s of length 2^ℓ in time $O(n)$. Since there are $\log n + 1$ levels, the overall preprocessing time for s is $O(n \log n)$. Let $H_{s_i}[\ell]$ store all the hashes for windows of length 2^ℓ of s_i for $i = 1, 2, .., m$. Hence the total preprocessing time is $\tilde{O}(nm)$.

Answering $Equal(i, \vec{d})$ in $O(m \log n)$ time

$Equal(i, \vec{d})$ queries can be implemented by doing a simple binary search over the presorted hashes in $O(m \log n)$ time. Suppose $Equal(i, \vec{d}[j]) = q_j$. We identify the smallest $\ell \geq 0$ such that $q_j < 2^\ell$, and then do another binary search for q_j between $i + 2^{\ell-1}$ to $i + 2^\ell$. Finally, we set $Equal(i, \vec{d}) = \min(q_2, ..., q_m)$.

B An $\tilde{O}_m(\lambda n^m)$ Algorithm for Multi-sequence LCS

▶ **Theorem 17.** *Given m strings $s_1, \ldots, s_m$ each of length n such that $\mathcal{L}(s_1, \ldots, s_m) = \lambda n$ where $\lambda \in (0, 1)$, there exists an algorithm that computes $\mathcal{L}(s_1, \ldots, s_m)$ in time $\tilde{O}_m(\lambda n^m + nm)$.*

The algorithm is build over the algorithm of [17], that given two strings x, y of length n such that $\mathcal{L}(x, y) = \lambda n$, computes $\mathcal{L}(x, y)$ in time $\tilde{O}(\lambda n^2)$. Though the algorithm of [17] shares a similar flavor with the classical quadratic time dynamic program algorithm, the main contribution of this work is that it introduces the concept of minimal ℓ-candidates that ensure that to compute the LCS, instead of enumerating the whole DP, it is enough to compute some selective entries that are important. Moreover they show if the LCS is small then the total number of minimal ℓ-candidates can be bounded. Also they can be constructed efficiently.

An $\tilde{O}(\lambda n^2)$ Algorithm for LCS

We first provide a sketch of the algorithm of [17]. We start with a few notations. Given two indices $i, j \in [n]$, let $\mathcal{L}(i, j)$ denotes the length of the LCS of $x[1, i]$ and $y[1, j]$ and x_i denotes the ith character of string x.

Given indices i, j we call $< i, j >$ an ℓ-candidate if $x_i = y_j$ and $\exists i', j' \in [n]$ such that $i' < i$, $j' < j$ and $< i', j' >$ is an $(\ell - 1)$-candidate. We say that $< i, j >$ is generated over $< i', j' >$. Also define $< 0, 0 >$ to be the 0-candidate. (for this purpose add a new symbol α at the beginning of both x, y. Hence $x[0] = y[0] = \alpha$) With this definition using induction we can claim that $< i, j >$ is an ℓ-candidate *iff* $\mathcal{L}(i, j) \geq \ell$ and $x_i = y_j$. Moreover as $\mathcal{L}(x, y) = \lambda n$, the maximum value of ℓ for which there exists an ℓ-candidate is λn. Hence to compute the LCS what we need to do is to construct a sequence of 0-candidate, 1-candidate, $\ldots$, $(\lambda n - 1)$-candidate and a λn-candidate such that the ith candidate can be generated from the $(i - 1)$th candidate. Note as for each i, there can be many ℓ-candidates enumerating all of them will be time consuming.

Therefor the authors bring the notion of minimal ℓ-candidate that are generated as follows. Consider two ℓ-candidates $< i_1, j_1 >$ and $< i_2, j_2 >$. If $i_1 \geq i_2$ and $j_1 \geq j_2$, then it is enough to keep only $< i_2, j_2 >$ as any $(\ell + 1)$-candidate that is generated from $< i_1, j_1 >$, can be generated from $< i_2, j_2 >$ as well. Call $< i_1, j_1 >$ a spurious candidate.

▶ **Lemma 18.** *Let the set $\{< i_\ell, j_\ell >, \ell \in \{1, 2, \dots\}\}$ denotes the set of ℓ-candidates. After discarding all spurious ℓ-candidates it can be claimed that $i_1 < i_2 < \dots$ and $j_1 > j_2 > \dots$.*

Proof. For any two ℓ-candidates $< i_1, j_1 >$ and $< i_2, j_2 >$, either 1) $i_1 < i_2$ and $j_1 \leq j_2$ or 2) $i_1 < i_2$ and $j_1 > j_2$ or 3) $i_1 = i_2$ and $j_1 \leq j_2$ or 4) $i_1 = i_2$ and $j_1 > j_2$. In the first and third case $< i_2, i_2 >$ is spurious and in the forth case $< i_1, j_1 >$ is spurious. Hence after removing all spurious candidates it can be ensured that $i_1 < i_2 < \dots$ and $j_1 > j_2 > \dots$. ◀

The ℓ-candidates which are left after the removal of all spurious candidates are called minimal ℓ-candidates. Notice as for each i there is at most one minimal ℓ candidate, total number of minimal ℓ-candidates for all choices of i and ℓ is at most λn^2. Using this bound and Lemma 3 in [17], an algorithm can be designed to compute all the minimal ℓ-candidates and thus $\mathcal{L}(x, y)$ in time $\tilde{O}(\lambda n^2)$.

Generalization for m strings

Now we provide an upper bound on the number of minimal ℓ-candidates for m strings given $\mathcal{L}(s_1, \dots, s_m) = \lambda n$. Given indices $i_1, \dots, i_m$ we call $< i_1, \dots, i_m >$ an ℓ-candidate if $s_1[i_1] = \dots = s_m[i_m]$ and $\exists i'_1, \dots, i'_m \in [n]$ such that $i'_j < i_j$, and $< i'_1, \dots, i'_m >$ is an $(\ell - 1)$-candidate. We say that $< i_1, \dots, i_m >$ is generated over $< i'_1, \dots, i'_m >$. Similar to the two string case using induction we can prove $< i_1, \dots, i_m >$ is an an ℓ-candidate *iff* $\mathcal{L}(s_1, \dots, s_m) \geq \ell$ and $s_1[i_1] = \dots = s_m[i_m]$. Therefore to compute $\mathcal{L}(s_1, \dots, s_m)$ it will be enough to generate a sequence of 0-candidate, 1-candidate, ..., $(\lambda n - 1)$-candidate and a λn-candidate such that the ith candidate can be generated from the $(i - 1)$th candidate. Next we describe the notion of spurious candidates and bound the total number of minimal ℓ-candidates.

For two ℓ-candidates $< i_1, \dots, i_{m-2}, i_{m-1}, i_m >$ and $< i_1, \dots, i_{m-2}, i'_{m-1}, i'_m >$, if $i_{m-1} \geq i'_{m-1}$ and $i_m \geq i'_m$ then we call the tuple $< i_1, \dots, i_{m-2}, i_{m-1}, i_m >$ spurious and discard it as any $(\ell + 1)$ tuple that is generated from $< i_1, \dots, i_{m-2}, i_{m-1}, i_m >$ can be generated from $< i_1, \dots, i_{m-2}, i'_{m-1}, i'_m >$ as well. The tuples that survives are called minimal ℓ-candidates. Hence following a similar argument as given for Lemma 18, we can claim the following.

▶ **Lemma 19.** *Let the set $\{< i_1, \dots, i_{m-2}, i^\ell_{m-1}, i^\ell_m >, \ell \in \{1, 2, \dots\}\}$ denotes the set of ℓ-candidates for fixed values of $i_1, \dots, i_{m-2}$. After discarding all spurious ℓ-candidates it can be claimed that $i^1_{m-1} < i^2_{m-1} < \dots$ and $i^1_m > i^2_m > \dots$.*

Note this implies that for a fixed choice of $i_1, \dots, i_{m-1}$, there exists at most one minimal ℓ-candidate. As $\ell = \lambda n$, total number of minimal ℓ-candidates over all choices of $i_1, \dots, i_{m-1}$ and ℓ is at most λn^m.

Next we state a lemma that is a generalisation of Lemma 3 of [17] for m strings.

▶ **Lemma 20.** *For $\ell \geq 1$ $< i_1, \dots, i_{m-2}, i_{m-1}, i_m >$ is a minimal ℓ-candidate* iff *$< i_1, \dots, i_{m-2}, i_{m-1}, i_m >$ is a ℓ-candidate with the minimum mth coordinate value such that $low < i_m < high$ where high is the minimum mth coordinate value of all ℓ-candidates having first $m - 2$ coordinate values $i_1, \dots, i_{m-2}$ and the $(m - 1)$th coordinate value less than i_{m-1} and low is the minimum mth coordinate value of all $(\ell - 1)$-candidates having first $m - 2$ coordinate values $i_1, \dots, i_{m-2}$ and the $(m - 1)$th coordinate value less than i_{m-1}.*

Together with the above lemma and the bound on the number of minimal ℓ-candidates following the algorithm of [17], we can design an algorithm that computes $\mathcal{L}(s_1, \dots, s_m)$ in time $\tilde{O}_m(\lambda n^m + nm)$.