Streaming and Small Space Approximation Algorithms for Edit Distance and Longest Common Subsequence*

Kuan Cheng
Peking University, Beijing, China

Alireza Farhadi
University of Maryland, College Park, MD, USA

MohammadTaghi Hajiaghayi
University of Maryland, College Park, MD, USA

Zhengzhong Jin
Johns Hopkins University, Baltimore, MD, USA

Xin Li
Johns Hopkins University, Baltimore, MD, USA

Aviad Rubinstein
Stanford University, CA, USA

Saeed Seddighin
Toyota Technological Institute, Chicago, IL, USA

Yu Zheng
Johns Hopkins University, Baltimore, MD, USA

Abstract

The *edit distance* (ED) and *longest common subsequence* (LCS) are two fundamental problems which quantify how similar two strings are to one another. In this paper, we first consider these problems in the asymmetric streaming model introduced by Andoni, Krauthgamer and Onak [11] (FOCS'10) and Saks and Seshadhri [64] (SODA'13). In this model we have random access to one string and streaming access the other one. Our main contribution is a constant factor approximation algorithm for ED with memory $\tilde{O}(n^{\delta})$ for any constant $\delta > 0$. In addition to this, we present an upper bound of $\tilde{O}_{\epsilon}(\sqrt{n})$ on the memory needed to approximate ED or LCS within a factor $1 \pm \epsilon$. All our algorithms are deterministic and run in polynomial time in a single pass.

We further study small-space approximation algorithms for ED, LCS, and *longest increasing sequence* (LIS) in the non-streaming setting. Here, we design algorithms that achieve $1 \pm \epsilon$ approximation for all three problems, where $\epsilon > 0$ can be any constant and even slightly sub-constant. Our algorithms only use poly-logarithmic space while maintaining a polynomial running time. This significantly improves previous results in terms of space complexity, where all known results need to use space at least $\Omega(\sqrt{n})$. Our algorithms make novel use of triangle inequality and carefully designed recursions to save space, which can be of independent interest.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Edit Distance, Longest Common Subsequence, Longest Increasing Subsequence, Space Efficient Algorithm, Approximation Algorithm

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.54

Category Track A: Algorithms, Complexity and Games

Related Version The full proofs of our results can be found in [36] and [31].
Full Version: `https://arxiv.org/abs/2002.11342`
Full Version: `https://arxiv.org/abs/2002.08498`

Funding *Kuan Cheng*: supported by a start-up funding of Peking University.
Alireza Farhadi: supported in part by Facebook Fellowship.
MohammadTaghi Hajiaghayi: supported in part by NSF BIGDATA grant IIS-1546108, and NSF SPX grant CCF-1822738.

* This paper is obtained by a recent merge of [36] and [31]. As a result of this joint effort, we not only improve the running times of all our algorithms to polynomial time. (Our streaming algorithms had exponential running times before.) But also extensively study algorithms for edit distance and LCS with small space in depth by achieving several novel results.

Zhengzhong Jin: supported by NSF Award CCF-1617713 and NSF CAREER Award CCF-1845349.
Xin Li: supported by NSF Award CCF-1617713 and NSF CAREER Award CCF-1845349.
Aviad Rubinstein: supported in part by a David and Lucile Packard Fellowship.
Saeed Seddighin: supported by an Adobe Research Award and a Google Research Gift.
Yu Zheng: supported by NSF Award CCF-1617713 and NSF CAREER Award CCF-1845349.

1 Introduction

We consider *edit distance* (ED) and *longest common subsequence* (LCS) which are classic problems measuring the similarity between two strings. Edit distance is defined on two strings s and $\bar{s}$ and seeks the smallest number of character insertions, character deletions, and character substitutions to transform s into $\bar{s}$. While in edit distance the goal is to make a transformation, longest common subsequence asks for the largest string that appears as a subsequence in both s and $\bar{s}$.

Edit distance and longest common subsequence have applications in various contexts, such as computational biology, text processing, compiler optimization, data analysis, image analysis, among others. As a result, both problems have been subject to a plethora of studies since 1950 (e.g. see [19, 20, 2, 14, 13, 11, 47, 17, 32, 34, 46, 57, 33, 42, 54, 23, 6, 26, 63, 24, 62, 48, 40, 38, 59, 3, 25, 45, 8, 10, 9, 7, 15, 39, 29, 53, 64, 43, 4, 21, 44, 26]).

Both problems are often used to measure the similarity of long strings. For example, a human genome consists of almost three billion base pairs that are modeled as a string for similarity testing. Classic algorithms for the problems require quadratic runtime as well as linear memory to find a solution. These bounds might be impractical for some real-world applications. Therefore, recent work on ED and LCS focus on obtaining fast algorithms [61, 44, 20, 13, 11, 8, 9, 60, 22, 52, 26] as well as solutions with small memory [45, 20, 28, 41].

As can be seen from the aforementioned references, while there have been a lot of previous works on obtaining fast algorithms, the equally important question of achieving algorithms using small space has not been studied in depth. In this paper our focus is to design algorithms with significantly improved space complexity for these string measures, in several different models.

Our first model is related to the *streaming* setting studied in most previous works on space complexity for ED and LCS. This is an increasingly popular framework to model memory constraints. In this setting, the input arrives as a data stream while only sublinear memory is available to the algorithm. The goal is to design an algorithm that solves/approximates the solution by taking a few passes over the data. While several works have studied ED and LCS in the streaming model (see Section 1.1 for a detailed discussion), positive results are known only for the low-distance regime [56, 66, 18, 28, 27]. In addition to this, strong lower bounds are given for the streaming variant of LCS [56, 66].

Inspired by the work of [11] (FOCS'10), Saks and Seshadhri [64] (SODA'13) studied the problem of approximating $n -$ LCS (which is half of the the edit distance between two strings of length n when only insertions and deletions are allowed) in the asymmetric model. In this model we have random access to one of the strings (say $\bar{s}$) and streaming access to the other string (say s). They showed that $(1+\epsilon)$ approximation of $n -$ LCS can be found with a memory of $\tilde{O}_\epsilon(\sqrt{n})$. Another work by Saha [63] gives an algorithm for ED in this setting, using a memory of $O(\frac{\sqrt{n}}{\epsilon})$, and achieving an ϵn *additive* approximation.

In this work, we study ED and LCS in the asymmetric streaming model. We present a single-pass deterministic constant factor approximation algorithm for ED that uses only $\tilde{O}(n^\delta)$ memory for any constant $\delta > 0$. In addition to this, we show that with the memory of $\tilde{O}_\epsilon(\sqrt{n})$ one can approximate both ED and LCS within a factor of $1 \pm \epsilon$. All our algorithms

are deterministic and run in a single-pass and in polynomial time. Moreover, the memory bound of our algorithm for LCS is tight due to a lower bound given in [37]. It is also worth mentioning that the lower bound of $\Omega(\log^2 n/\epsilon)$ is known for computing $1+\epsilon$ approximation of $n-$ LCS due to the result of [58].

LIS and distance to monotonicity (DTM) are special cases of LCS and ED that are also studied in the streaming model [41, 64]. In these two problems, one of the strings is a length n string over the alphabet $[n]$ and the second string is the sorted permutation $\langle 1, 2, \ldots, n\rangle$. Therefore, any asymmetric streaming algorithm for LCS and ED also implies a classical streaming algorithm for LIS and DTM, since we can assume that we have random access to the sorted permutation $\langle 1, 2, \ldots, n\rangle$. As a result, our algorithms for ED and LCS can be seen as a generalization of [41, 64] on streaming LIS and distance to monotonicity.

More generally, we also study the memory requirements of ED, LCS, and LIS in the non-streaming model, where we have random access to all input strings. Even in this setting, classic algorithms that compute ED and LCS exactly using dynamic programming need linear space. A recent work [50] slightly improves the space complexity to $O(\frac{n\log^{1.5} n}{2^{\sqrt{\log n}}})$ while preserving a polynomial running time, however achieving strongly sublinear space (i.e., space $n^{1-\alpha}$ for some constant $\alpha > 0$) with a polynomial running time for ED and LCS still seems out of reach. Therefore, in this paper we turn to the relaxed goal of approximating ED and LCS using significantly smaller space while preserving a polynomial running time.

The only previous positive results in this setting we are aware of, are the previously mentioned works on streaming algorithms and the work of Saha [63]. Again, for ED the streaming results only work in the low-distance regime, except the work of Saks and Seshadhri [64] which gives a $(1+\epsilon)$ approximation of $n-$ LCS using $\tilde{O}_\epsilon(\sqrt{n})$ space. Additionally, [64] also gives a randomized algorithm that achieves an ϵn additive approximation of LCS in this model, using space $O(k\log^2 n/\epsilon)$ where k is the maximum number of times any symbol appears in y. [63] gives a small space algorithm for ED, although the result only works for constant alphabet size and only provides an additive approximation instead of multiplicative approximation. Further the space needed is at least $\Omega(n^{2/3})$. For LIS the situation is slightly better. In particular, the work of Gopalan, Jayram, Krauthgamer and Kumar [41] provides a deterministic streaming algorithm that approximates LIS to within a $1-\epsilon$ factor, using time $O(n\log n)$ and space $O(\sqrt{n/\epsilon}\log n)$; while a very recent work by Kiyomi, Ono, Otachi, Schweitzer and Tarui [51] obtains a deterministic algorithm that computes LIS *exactly* using $O(n^{1.5}\log n)$ time and $O(\sqrt{n}\log n)$ space. [63] in addition provides an algorithm for LIS using space $O(\frac{\log n}{\epsilon})$ that achieves an ϵn additive approximation. Thus here we seek to obtain an $1-\epsilon$ approximation of LIS using space much smaller than $\sqrt{n}$.

More broadly, the questions studied in this paper are closely related to the general question of *non-deterministic small space computation* vs. *deterministic small space computation*. Specifically, the decision versions of all three problems (ED, LCS, and LIS) can be easily shown to be in the class NL (i.e., non-deterministic log-space), and the question of whether NL = L (i.e., if non-deterministic log-space computation is equivalent to deterministic log-space computation) is a major open question in complexity theory. Note that if NL = L, this would trivially imply polynomial time algorithms for *exactly* computing ED, LCS, and LIS in logspace. However, although we know that NL $\subseteq$ P and NL $\subseteq$ SPACE$(\log^2 n)$ (by Savitch's theorem [65]), it is not known if every problem in NL can be solved simultaneously in polynomial time and polylog space, i.e., if NL $\subseteq$ SC where SC is Steve's class. In fact, it is not known if an NL-complete language (e.g., directed s-t connectivity) can be solved simultaneously in polynomial time and strongly sub linear space (i.e., space $n^{1-\alpha}$ for some fixed constant $\alpha > 0$). Thus, studying special problems such as ED, LCS, and LIS, and the relaxed version of approximation is a reasonable first step towards major open problems.

In the non-streaming setting, for all three problems ED, LCS, and LIS, we give efficient *deterministic* approximation algorithms that can achieve $1 \pm \epsilon$ approximation, using even polylog(n) space while maintaining a polynomial running time. By relaxing the space complexity to n^δ for any constant $\delta > 0$, we obtain algorithms whose running time is comparable to the standard dynamic programming approach. This is in sharp contrast to the time complexity of ED, LCS, where we only know how to beat the standard dynamic programming significantly by using *randomized* algorithms and with much worse approximation guarantees.

Last but not least, it turns out that we can also use our techniques developed for the non-streaming model to significantly reduce the running time of our algorithm in the asymmetric streaming model, resulting in a polynomial time algorithm.

1.1 Related work

Quadratic time solutions for ED and LCS have been known for many decades [55]. Recently, it has been shown that a truly subquadratic time solution for either ED or LCS refutes *Strong Exponential Time Hypothesis* (SETH), a conjecture widely believed in the community (see [14, 2, 25]). Therefore, much attention is given to approximation algorithms for the two problems. For edit distance, a series of works [54], [16], [17], and [13] improve the approximation factor culminating in the seminal work of Andoni, Krauthgamer, and Onak [11] that finally obtains a polylogarithmic approximation factor in near-linear time. More recently constant factor approximation algorithms with truly subquadratic runtimes are obtained for edit distance (a question which was open for a few decades): first a quantum algorithm [20], then a classic solution [26], and finally near linear time solutions are given [22, 52, 12]. LCS has also received tremendous attention in recent years [44, 60, 61, 1, 4, 30]. Only trivial solutions were known for LCS until very recently: a 2 approximate solution when the alphabet is 0/1 and an $O(\sqrt{n})$ approximate solution for general alphabets in linear time. Both these bounds are recently improved by [44] and Rubinstein and Song [60] (see also a recent approximation algorithms given by [61]).

Streaming algorithms for edit distance have been limited to the case that the distance between the two strings is *bounded* by a parameter k which is substantially smaller than n. In particular, [28] gives a randomized one-pass algorithm in a variant of the streaming model where one can scan the two strings s and $\bar{s}$ simultaneously in the coordinated fashion, using space $O(k^6)$ and time $O(n + k^6)$. This was later improved to space $O(k)$ and time $O(n + k^2)$ in [27, 18]. [18] further gives a randomized one-pass algorithm in the standard streaming model using space $O(k^8 \operatorname{polylog} n)$ and time $\tilde{O}(k^2 n)$. It is improved to $O(k^3 \operatorname{polylog} n)$ space recently in [49]. Note that in all these algorithms the space can be as large as n.

Table 1 The results for asymmetric streaming model of this paper along with previous work.

problem	approximation factor	memory	reference
ED	$O(2^{1/\delta})$	$\tilde{O}(n^\delta/\delta)$	Theorem 1
ED	$1+\epsilon$	$\tilde{O}_\epsilon(\sqrt{n})$	Theorem 2
LCS	$1-\epsilon$	$\tilde{O}_\epsilon(\sqrt{n})$	Theorem 2
LIS	$1-\epsilon$	$\tilde{O}_\epsilon(\sqrt{n})$	[41] (SODA'07)
n − LCS	$1+\epsilon$	$\tilde{O}_\epsilon(\sqrt{n})$	[64] (SODA'13)
DTM	$1+\epsilon$	$O_\epsilon(\log^2 n)$	[64, 58] (SODA'13, SODA'14)
DTM	$1+\epsilon$	$\tilde{O}_\epsilon(\sqrt{n})$	[41] (SODA'07)
DTM	2	$O(\log^2 n)$	[35] (SODA'08)
DTM	4	$O(\log^2 n)$	[41] (SODA'07)

1.2 Our Results

We summarize our main theorems in this section and in Table 1. In the next subsection we demonstrate our main techniques and general ideas in details. Formal proofs are deferred to the appendix. First we start with the following results for the asymmetric streaming model.

▶ **Theorem 1.** *Given online string s and offline string $\bar{s}$ both in Σ^n, for any constant $\delta > 0$, there exists a deterministic algorithm that, making one pass through s, outputs a $O(2^{1/\delta})$ approximation of $\mathsf{ED}(s, \bar{s})$ using $\tilde{O}(n^\delta/\delta)$ space and $\tilde{O}_\delta(n^4)$ time.*

▶ **Theorem 2.** *Given online string s and offline string $\bar{s}$ both in Σ^n, for any constant $\epsilon \in (0, 1)$, there is a deterministic algorithm that, making one pass through s, outputs a $(1-\epsilon)$-approximation of $\mathsf{LCS}(s, \bar{s})$ using $\tilde{O}(\frac{\sqrt{n}}{\epsilon})$ space, or $(1+\epsilon)$-approximation of $\mathsf{ED}(s, \bar{s})$ using $\tilde{O}(\sqrt{\frac{n}{\epsilon}})$ space, in polynomial time.*

Theorem 1 is surprising in the sense that by allowing random access to one string, we can design an efficient deterministic one pass streaming algorithm achieving a constant approximation of ED with space n^δ for any constant $\delta > 0$. Previously no such positive results with sublinear space are known in the full streaming model even for randomized algorithms. The proof of this theorem makes novel uses of triangle inequality, and combines the techniques we develop for the non-streaming model. Theorem 2 further shows that we can in fact achieve an $(1+\epsilon)$-approximation of ED, albeit using a larger space $(\tilde{O}(\sqrt{\frac{n}{\epsilon}}))$. The result also extends to give the first $(1-\epsilon)$-approximation of LCS with sublinear space in the asymmetric streaming model, and the space matches the lower bound implied from the lower bound for LIS in [37].

Next we bring our results for the (non-streaming) small space model.

▶ **Theorem 3.** *Given any strings x and y each of length n, there are deterministic algorithms that approximate $\mathsf{ED}(x, y)$ with the following parameters:*

1. *For any constants $\delta \in (0, \frac{1}{2})$ and $\epsilon \in (0, 1)$, an algorithm that outputs a $1+\epsilon$ approximation of $\mathsf{ED}(x, y)$ using $\tilde{O}_{\epsilon,\delta}(n^\delta)$ space and $\tilde{O}_{\epsilon,\delta}(n^2)$ time.*
2. *An algorithm that outputs a $1 + O(\frac{1}{\log\log n})$ approximation of $\mathsf{ED}(x, y)$ using $O(\frac{\log^4 n}{\log\log n})$ space and $n^{7+o(1)}$ time.*

Note that our first algorithm for ED uses roughly the same running time as the standard dynamic programming, but much smaller space. Indeed, we can use space n^δ for any constant $\delta > 0$. This also significantly improves the previous result of [64], which needs to use space $\Omega(\sqrt{n}\log n)$ and only provides a $2+\epsilon$ approximation for standard ED. With a larger (but still polynomial) running time, we can achieve space complexity $O(\frac{\log^4 n}{\log\log n})$.

▶ **Theorem 4.** *Given any strings x and y each of length n, there are deterministic algorithms that approximate $\mathsf{LCS}(x, y)$ with the following parameters:*

1. *For any constants $\delta \in (0, \frac{1}{2})$ such that $\frac{1}{\delta}$ is an integer, and $\epsilon \in (0, 1)$,an algorithm that outputs a $1-\epsilon$ approximation of $\mathsf{LCS}(x, y)$ using $\tilde{O}_{\epsilon,\delta}(n^\delta)$ space and $\tilde{O}_{\epsilon,\delta}(n^{3-\delta})$ time.*
2. *An algorithm that outputs a $1 - O(\frac{1}{\log\log n})$ approximation of $\mathsf{LCS}(x, y)$, using $O(\frac{\log^4 n}{\log\log n})$ space and $n^{6+o(1)}$ time.*

To the best of our knowledge, Theorem 4 (together with Theorem 2) is the first $1-\epsilon$ approximation of LCS using truly sub-linear space, and in fact we can achieve space n^δ for any constant $\delta > 0$ with only a slightly larger running time than the standard dynamic programming approach. We can achieve space $O(\frac{\log^4 n}{\log\log n})$ with an even larger (but still polynomial) running time. For LIS, we have a similar theorem.

▶ **Theorem 5.** *Given any string x of length n, there are deterministic algorithms that approximate $\mathsf{LIS}(x)$ with the following parameters:*

1. *For any constants $\delta \in (0, \frac{1}{2})$ such that $\frac{1}{\delta}$ is an integer, and $\epsilon \in (0,1)$, an algorithm that computes a $1-\epsilon$ approximation of $\mathsf{LIS}(x)$ using $\tilde{O}_{\epsilon,\delta}(n^{\delta})$ space and $\tilde{O}_{\epsilon,\delta}(n^{2-2\delta})$ time.*
2. *An algorithm that outputs a $1 - O(\frac{1}{\log\log n})$ approximation of $\mathsf{LIS}(x)$ using $O(\frac{\log^4 n}{\log\log n})$ space and $n^{5+o(1)}$ time.*

2 Preliminaries

Let Σ be an alphabet. For a string $s \in \Sigma^n$, we use $s[i]$ to denote the i^{th} character in s. We use $s[i,j]$ to denote the substring of s from the i^{th} character to the j^{th} character. We also use $s[i,j)$ to denote the substring of s from the i^{th} character to $(j-1)^{\text{th}}$ character ($s[i,i)$ is an empty string). We denote the concatenation of two strings x and y by $x \circ y$. We use two special symbols ∞ and $-\infty$ with $\infty > i$ and $-\infty < i$ for any character $i \in \Sigma$.

Given two strings s and $\bar{s}$ in Σ^n, the longest common subsequence (LCS) of s and $\bar{s}$ is a string t with the maximum length such that t is a subsequence of both s and $\bar{s}$. In other words, t can be obtained from both s and $\bar{s}$ by removing some of the characters. We use $\mathsf{LCS}(s,\bar{s})$ to denote the length of the LCS of two strings s and $\bar{s}$. The edit distance (ED) between two strings s and $\bar{s}$, denoted by $\mathsf{ED}(s,\bar{s})$, is the minimum number of character insertions, deletions, and substitutions needed to transform one string to the other string. The longest increasing subsequence (LIS) problem is defined as follows. We assume there is a given total order on the alphabet set Σ. We say the string $x \in \Sigma^t$ is an *increasing subsequence* of $s \in \Sigma^n$ if there exists indices $1 \le i_1 < i_2 < \cdots < i_t \le n$ such that $x = s_{i_1} s_{i_2} \cdots s_{i_t}$ and $s_{i_1} < s_{i_2} < \cdots < s_{i_t}$. We denote the length of the longest increasing subsequence of string s by $\mathsf{LIS}(s)$.

Asymmetric streaming model. Throughout this paper, we assume that the input of the algorithm consists of two strings $\bar{s}$ and s. We assume for simplicity and without loss of generality that the two strings have equal length n. We call the string $\bar{s}$ the *offline* string and assume that the algorithm has random access to the characters of $\bar{s}$ via making a query. The other string s arrives as a stream of characters. We call s the *online* string.

3 Constant Approximation for Streaming Edit distance

Our main results in the asymmetric streaming setting is an algorithm that given any constant $\delta > 0$ finds a constant approximation of the edit distance using $\tilde{O}(n^{\delta})$ memory. As we discussed in the previous section, instead of directly solving the edit distance, we aim to find a substring of s such that its edit distance is smallest to $\bar{s}$. We can this problem *Closest Substring*. It is defined as follows. It takes two inputs: an offline string $\bar{s}$ and an online string s. The goal is to find two indices l, r and $\mathsf{ED}(\bar{s}[l,r],s)$ such that $\mathsf{ED}(\bar{s}[l,r],s) \le \mathsf{ED}(\bar{s}[i,j],s)$ for every $1 \le i \le j \le n$.

We first show that how solving the closest substring problem can give us a good approximation of the edit distance. Let $\bar{s}[l,r]$ be the substring of $\bar{s}$ with the minimum edit distance to s. We know by the definition of edit distance that it satisfies the *triangle inequality*[1].

[1] $\mathsf{ED}(s_1,s_3) \le \mathsf{ED}(s_1,s_2) + \mathsf{ED}(s_2,s_3)$ for any strings s_1, s_2, s_3.

Therefore, we have

$$\mathsf{ED}(\bar{s}, s) \leq \mathsf{ED}(\bar{s}, \bar{s}[l, r]) + \mathsf{ED}(\bar{s}[l, r], s)\,. \tag{1}$$

We also have,

$$\begin{aligned}
&\mathsf{ED}(\bar{s}, \bar{s}[l, r]) + \mathsf{ED}(\bar{s}[l, r], s) \\
&\quad \leq \mathsf{ED}(\bar{s}, s) + \mathsf{ED}(\bar{s}[l, r], s) + \mathsf{ED}(\bar{s}[l, r], s) && \text{By the triangle inequality.} \\
&\quad \leq 3\mathsf{ED}(\bar{s}, s) && \text{Since } \bar{s}[l, r] \text{ has the minimum ED to } s.
\end{aligned} \tag{2}$$

It follows from (1) and (2) that $\mathsf{ED}(\bar{s}, \bar{s}[l, r]) + \mathsf{ED}(\bar{s}[l, r], s)$ is a 3-approximation of the edit distance between $\bar{s}$ and s. Therefore, if we design a streaming algorithm that finds $\bar{s}[l, r]$ and its edit distance from s, we can then estimate the edit distance of s and $\bar{s}$ by computing $\mathsf{ED}(\bar{s}, \bar{s}[l, r]) + \mathsf{ED}(\bar{s}[l, r], s)$. In the following theorem which directly implies from Savitch's theorem [65], we show that $\mathsf{ED}(\bar{s}, \bar{s}[l, r]) + \mathsf{ED}(\bar{s}[l, r], s)$ can be computed using a poly-logarithmic memory. In specific, we show that the edit distance between any two substrings of the offline string can be computed using a very small memory of $O(\log^2 n)$. The proof is deferred to the full version [36]

▶ **Theorem 6.** *Suppose that we have random access to two given strings s and $\bar{s}$ of length n. Then $\mathsf{LCS}(s, \bar{s})$ and $\mathsf{ED}(s, \bar{s})$ can be computed using $O(\log^2 n)$ memory.*

Therefore, by finding the substring that has the minimum edit distance to s, we can get a good approximation of the edit distance. Nonetheless, we do not know any streaming algorithm with the memory of $O(n^\delta)$ for finding closest substring, and our algorithm only finds an approximate solution for this problem. In other words, it finds a substring of $\bar{s}$ such that its approximate edit distance to s is close to the minimum. In the rest of the section, we show that how we can approximately solve the closest substring problem with the memory of $\tilde{O}(n^\delta)$. Given an online string, we divide the online string into $n^{1-\delta}$ windows of size n^δ. Our algorithm (formally as Algorithm 1), then recursively finds substrings of $\bar{s}$ that have the minimum edit distance from each of these windows. Note that for each window we can store the result of solving the closest substring problem in $O(\log n)$ (We can store only three numbers which are the start and the end of the interval and the approximate edit distance to the online string). Therefore, by the end of all recursive calls our algorithm needs to store $O(n^\delta)$ values.

In order to find the solution of the closest substring problem using these partial solutions, our algorithm considers all different substrings $\bar{s}[l, r]$ of $\bar{s}$ and all different mappings between the windows of the s and the substrings of $\bar{s}[l, r]$. Then, for any mapping it estimates the edit distance between a window of s and its mapped substring of $\bar{s}[l, r]$ using the solution of the closest substring problem that we have found in the recursive call.

In order to analyze our algorithm, we first show that finding any approximation of the closest substring problem, can yield us an approximation for the edit distance. We first define an approximate version of the closest substring problem as follows.

▶ **Definition 7.** *Given an offline string $\bar{s}$ and online string s, we say that the substring $\bar{s}[l, r]$ along with its approximate edit distance d is an α-approximation for the closest substring problem if for any substring $\bar{s}[l^*, r^*]$ we have*

$$\mathit{ED}(\bar{s}[l, r], s) \leq d \leq \alpha \cdot \mathit{ED}(\bar{s}[l^*, r^*], s)\,. \tag{3}$$

Algorithm 1 Algorithm APPROXIMATE-CLOSEST-SUBSTR for approximating ED.

Data: An offline string $\bar{s}$ of length n, a stream of characters of the online string s, and a parameter $\delta > 0$.

if $|s| \leq n^{\delta}$ **then**
  Store all characters of s in the memory.
  Find a substring of $\bar{s}$ that has the minimum edit distance to s. Let $\bar{s}[l, r]$ be this substring and d be its edit distance.
  return l, r and d.
else
  $\xi \leftarrow n^{\delta}$.
  Divide s into ξ windows $s_1^*, s_2^*, \ldots, s_{\xi}^*$ of size $|s|/\xi$.
  for $i \in [\xi]$ **do**
    Recursively find the closest substring of $\bar{s}$ from s_i^* . Let l_i, r_i be the start and the end of this substring respectively, and d_i be the approximate edit distance of this substring to s_i^*.
  $min\_dist \leftarrow \infty$.
  for $1 \leq p_0 \leq p_1 \leq \ldots \leq p_{\xi} \leq n + 1$ **do**
    $dist = \sum_{i=1}^{\xi} d_i + \mathsf{ED}\big(\bar{s}[p_{i-1}, p_i), \bar{s}[l_i, r_i]\big)$.
    if $dist < min\_dist$ **then**
      $min\_dist \leftarrow dist$.
      $l \leftarrow p_0$.
      $r \leftarrow p_{\xi} - 1$.
  return l, r and min_dist.

In the following claim we show that we can use any α-approximation of the closest substring problem to get a $O(\alpha)$-approximation for the edit distance.

▷ Claim 8. Let $\bar{s}[l, r]$ be an α approximation of the closest substring problem and let d be its approximate edit distance to s. Then for any substring $\bar{s}[l^*, r^*]$, $d + \mathsf{ED}\big(\bar{s}[l, r], \bar{s}[l^*, r^*]\big)$ is a $(2\alpha + 1)$-approximation for the edit distance between $\bar{s}[l^*, r^*]$ and s.

Proof. First we show that $\big(d + \mathsf{ED}\big(\bar{s}[l, r], \bar{s}[l^*, r^*]\big)\big)$ is not less than the edit distance between $\bar{s}[l^*, r^*]$ and s.

$$\begin{aligned} d + \mathsf{ED}\big(\bar{s}[l, r], \bar{s}[l^*, r^*]\big) &\geq \mathsf{ED}(\bar{s}[l, r], s) + \mathsf{ED}\big(\bar{s}[l, r], \bar{s}[l^*, r^*]\big) && \text{By (3).} \\ &\geq \mathsf{ED}\big(\bar{s}[l^*, r^*], s\big)\,. && \text{By the triangle inequality.} \end{aligned}$$

We now show that the value of $\big(d + \mathsf{ED}\big(\bar{s}[l, r], \bar{s}[l^*, r^*]\big)\big)$ is at most $(2\alpha + 1) \cdot \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big)$. Thus, it gives us a $(2\alpha + 1)$-approximation of the edit distance. We have

$$\begin{aligned} d + \mathsf{ED}\big(\bar{s}[l, r], \bar{s}[l^*, r^*]\big) &\leq d + \mathsf{ED}\big(s, \bar{s}[l, r]\big) + \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big) && \text{By the triangle inequality.} \\ &\leq d + \alpha \cdot \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big) + \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big) && \text{By (3).} \\ &= d + (\alpha + 1) \cdot \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big) \\ &\leq \alpha \cdot \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big) + (\alpha + 1) \cdot \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big) && \text{By (3).} \\ &= (2\alpha + 1) \cdot \mathsf{ED}\big(s, \bar{s}[l^*, r^*]\big)\,, \end{aligned}$$

which completes the proof of the claim. ◁

Based on our discussion above, we design an algorithm that finds a constant approximation of the edit distance using $\tilde{O}(n^\delta)$ memory for any $\delta > 0$. The algorithm first divides the online string into n^δ windows with the equal length. Therefore, the length of each window is $n^{1-\delta}$. It then finds an approximate solution of the closest substring problem for each window recursively. By Claim 8, we can use the approximate solution of the closest substring problem for each window, to find its edit distance from every other substring of the offline string. The algorithm uses these approximate solutions to approximate the edit distance between the entire online string and any substring of the offline string.

Note that by each recursive call the length of the online string will get smaller by a multiplicative factor of $n^{-\delta}$. Therefore, when the depth of the recursive calls becomes $1/\delta$, the length of the remaining online string is bounded by $O(n^\delta)$ and we can store all of this remaining online string in the memory and find the exact solution of the closest substring problem. Thus, the depth of the recursion is bounded by $O(1/\delta)$. In the following theorem we show that the approximation ratio of our algorithm is $O(2^{1/\delta})$.

▶ **Theorem 9.** *Given an offline string $\bar{s}$, an online string s and any constant $\delta > 0$, let n be the length of the offline string and n^γ be the length of the online string where $\gamma > 0$. Then, Algorithm 1 finds a $O\left(2^{\gamma/\delta}\right)$ approximation for the closest substring problem.*

The full proof of Theorem 9 is deferred to the full version [36]. Now we are ready to prove a version of Theorem 1 without the time complexity bound. We introduce how to achieve polynomial runtime in Section 5.2.

Proof of Theorem 1 without the time complexity bound. By Theorem 9, Algorithm 1 finds a $O(2^{1/\delta})$ approximation of the closest substring problem. Recall that by Theorem 6, we can find the edit distance of any two substrings of $\bar{s}$ using a very small memory. Therefore by Claim 8, we can find a $O(2^{1/\delta})$ approximation of the edit distance between s and $\bar{s}$.

Now we show that the memory of Algorithm 1 is at most $\tilde{O}(n^\delta/\delta)$. While the length of the online string is larger than n^δ, Algorithm 1 divides the online string into n^δ windows and recursively solves the closest substring problem for each window. Therefore, by each recursive call the length of the online string will decrease by a multiplicative factor of $n^{-\delta}$. Thus, the maximum depth of the recursive calls is bounded by $O(1/\delta)$. At each call the algorithm acquires a memory of $\tilde{O}(n^\delta)$ which is the memory needed for storing the result of the recursive calls and iterating over all possible mappings. Therefore, the memory of the algorithm is bounded by $\tilde{O}(n^\delta/\delta)$. ◀

4 $1+\epsilon$-Approximation of Streaming ED

A previous work by Saks and Seshadri [64] studied the approximation algorithm for deletion distance in the asymmetric streaming model. Here, the deletion distance (DD) is defined as the minimum number of insertions and deletions required to transform s to $\bar{s}$. Thus, assuming $|s| = |\bar{s}| = n$, $\mathsf{DD}(s, \bar{s}) = 2(n - \mathsf{LCS}(s, \bar{s}))$. [64] gives an algorithm that outputs a $(1+\epsilon)$ approximation of $\mathsf{DD}(s, \bar{s})$ for any constant $\epsilon > 0$ with $\tilde{O}_\epsilon(\sqrt{n})$ space.

▶ **Theorem 10** ([64])**.** *Given an offline string $\bar{s}$, an online string s, both with length n, and any constant $\epsilon \in (0, 1]$, there is a deterministic algorithm that outputs a $1+\epsilon$ approximation of DD$(\bar{s}, s)$ with $O(\sqrt{n \log n/\epsilon})$ space in polynomial time.*

Although deletion distance is not equivalent to edit distance we study in this paper, there is a simple transformation that reduces edit distance to deletion distance. The transformation can be found in previous works (see example 6.9 of [67] for example). More specifically,

for any given string s, we can obtain a new string s' by prepending a special symbol \$ (\$ is not the in the alphabet set) to each character of s. Thus, say $s = s_1 s_2 \cdots s_n$, we let $s' = \$s_1\$s_2 \cdots \$s_n \in (\Sigma \cup \{\$\})^{2n}$. We can obtain a string $\bar{s}'$ from $\bar{s}$ with the same transformation. The following lemma shows that the above transformation reduces computing edit distance to computing deletion distance.

▶ **Lemma 11.** *$DD(s', \bar{s}') = 2ED(s, \bar{s})$.*

Proof. We first show that $2\mathsf{ED}(s, \bar{s}) \geq \mathsf{DD}(s', \bar{s}')$. Assume we can transform s to $\bar{s}$ with d edit operations (insertion, deletion, and substitution), we show that it is possible to transform s' to $\bar{s}'$ with $2d$ insdel operations (insertion and deletion). To see this, if we delete one symbol from s, we delete the corresponding symbol in s' together with the \$ prepended to it. If we insert one symbol to s, we insert the corresponding symbol to s' together with a \$ prepended to it. If we substitute one symbol with another in s, we can first delete the corresponding symbol in s' and insert the new symbol at the same position.

We now show that $2\mathsf{ED}(s, \bar{s}) \leq \mathsf{DD}(s', \bar{s}')$. We consider an LCS between s' and $\bar{s}'$, and consider each pair of adjacent matches of \$ in this LCS. Without loss of generality, we can assume that in at least one side (say s'), this pair looks like $\$a\$$ where a is a symbol (we can also append a \$ at the end if needed), because otherwise if both sides have at least two symbols between the \$'s, then there is another pair of \$'s in the middle that can be matched and added to the LCS. Suppose now in $\bar{s}'$ there are t non-\$ symbols between the two \$'s. We have two cases: 1. If in the LCS, a is matched to one of the t symbols in $\bar{s}$. Then the number of insdel operations between s' and $\bar{s}'$ we need for this part is $2t - 2$, while for the corresponding part of s and $\bar{s}$, we only need $t - 1$ deletions. 2. If in the LCS a is not matched to any of the t symbols in $\bar{s}$. Then the number of insdel operations between s' and $\bar{s}'$ we need for this part is $2t$, while for s and $\bar{s}$ we need $t - 1$ deletions and one substitution (we can substitute a for any of the t symbols in $\bar{s}$), so that's t operations. Thus, if we can transform s' to $\bar{s}'$ with d insdel operations, we can transform s to $\bar{s}$ with $\frac{d}{2}$ edit operations. ◀

Note that the reduction can be implemented in a streaming manner, thus the following theorem is a direct result of the reduction and Theorem 10.

▶ **Theorem 12.** *Given an offline string $\bar{s}$, an online string s, both with length n, and any constant $\epsilon > 0$, there is a deterministic algorithm that outputs a $1 + \epsilon$ approximation of $ED(\bar{s}, s)$ with $\tilde{O}(\sqrt{\frac{n}{\epsilon}})$ space in polynomial time.*

5 Space Efficient Algorithms for ED in the Non-Streaming Model

5.1 Space Efficient algorithms for ED

In this section, we present our algorithm for approximating ED with small space. The pseudocode for our algorithm is given in Algorithm 2. Our algorithm is based on recursion. In each level of recursion, we use an idea from [45] to approximate the edit distance between certain pairs of substrings. We start by giving a brief description of the algorithm in [45].

Let x and y be two input strings such that $|x| = n$ and $|y| = m$. We assume a value Δ is given to us. If Δ is a $(1+\epsilon)$-approximation of $\mathsf{ED}(x, y)$, then the algorithm will output a good approximation of $\mathsf{ED}(x, y)$. Otherwise, the algorithm will output a value at least $\mathsf{ED}(x, y)$. Thus, we can try every $\Delta = \lceil (1 + \epsilon)^i \rceil \leq n$ for some integer i and take the minimum. This only increases the running time by a $\log_{1+\epsilon} n$ factor.

Given such a Δ, we first divide x into b blocks each of length $\frac{n}{b}$. [45] showed that, for each block of x that is not matched to a too large or too small interval in y, there is a way to choose $O(\frac{b}{\epsilon} \log_{1+\epsilon} n) = O(\frac{b}{\epsilon^2} \log n)$ candidate intervals such that one of them is close enough to the

Algorithm 2 Algorithm SpaceEfficientApproxED.

Data: Two strings x and y, parameters $b \leq \sqrt{n}$ and $\epsilon \in (0,1)$

```
if |x| ≤ b then
    compute ED(x, y) exactly.
    return ED(x, y).
ed ← ∞.
set n = |x| and m = |y|.
divide x into b block each of length at most ⌈n/b⌉ such that x = x^1 ∘ x^2 ∘ ··· ∘ x^b.
for all Δ = 0 or ⌈(1+ε)^j⌉ for some integer j and Δ ≤ max{|x|, |y|} do
    for i = 1 to b do
        for all (a, b) ∈ CandidateSet(n, m, (l_i, r_i), ε, Δ) do
            M(i, (a, b)) ← SpaceEfficientApproxED(x^i, y[a, b], b, ε).
    e ← min{e, DPEditDistance(n, m, b, ε, Δ, M)}.
return e.
```

optimal alignment. We compute the edit distance between each block and all of its candidate intervals, which gives $O(\frac{b^2}{\epsilon^2}\log n)$ values. After this, a simple dynamic programming gives $(1+O(\epsilon))$-approximation of the edit distance if Δ is a $(1+\epsilon)$-approximation of $\mathsf{ED}(x,y)$. The dynamic programming algorithm takes $O(\frac{b^3\log n}{\epsilon^3})$ time.

Since each block has length $\frac{n}{b}$, computing the edit distance of each block with one of its candidate intervals in y takes at most $O(\frac{n}{b}\log n)$ space (we assume each symbol can be stored with space $O(\log n)$). We can run this algorithm sequentially and reuse the space for each computation. Storing the edit distance of each pair takes $O(\frac{b^2}{\epsilon^2}\log^2 n)$ bits of space. Thus, if we take $b = n^{1/3}$, the above algorithm uses a total of $\tilde{O}_\epsilon(n^{2/3})$ space.

We now run the above algorithm recursively to further reduce the space required. Algorithm 2 takes four inputs: two strings $x, y \in \Sigma^n$, two parameters b, ϵ such that $b \leq \sqrt{n}$ and $\epsilon \in (0,1)$. The goal is to output a good approximation of $\mathsf{ED}(x,y)$ with small space (related to parameters b and ϵ). Similarly, we first divide x into b blocks. We try every Δ that is equal to $\lceil(1+\epsilon)^i\rceil$ for some integer i, and for each Δ there is a set of candidate intervals for each block of x. Then, for each block of x, we use a sub algorithm called *CandidateSet* to return all its $O(\frac{b}{\epsilon^2}\log n)$ candidate intervals. Instead of computing the edit distance between each block of x and its candidate intervals exactly, we recursively call our space efficient approximation algorithm with this pair as input, while keeping b and ϵ unchanged. After getting the results from the recursive calls, we combine these results with a dynamic programming (the sub algorithm *DPEditDistance*). We argue that if the recursive call outputs a $(1+\gamma)$-approximation of the actual edit distance, the output of the dynamic programming increases by at most a $(1+\gamma)$ factor. Thus if Δ is a $(1+\epsilon)$-approximation of $\mathsf{ED}(x,y)$, the output of the dynamic programming is guaranteed to be a $(1+O(\epsilon))(1+\gamma)$-approximation. The recursion stops whenever one of the input strings has length smaller than b, where we compute the edit distance exactly with $O(b\log n)$ space.

Notice that at each level of the recursion, the first input string is divided into b blocks if it has length larger than b. Thus the length of first input string at the i-th level of recursion is at most $\frac{n}{b^{i-1}}$. The depth of recursion is at most $d = \log_b n$.

At the d-th level, Algorithm 2 computes the edit distance exactly. Using this as a base case, we can show that the output of the i-th level of recursion is a $(1+O(\epsilon))^{d-i}$ approximation of the edit distance by induction on i from d to 1. Thus, the output in the first level is guaranteed to be a $(1+O(\epsilon))^d$-approximation. Since $d \leq \log_b n$, we get a $(1+O(\epsilon))^d = 1+O(\epsilon d) = 1+O(\epsilon\log_b n)$ approximation of $\mathsf{ED}(x,y)$.

To analyze the time and space complexity, we study the recursion tree in our algorithm. Notice that for each block x^i and each choice of Δ, we consider $O(\frac{b}{\epsilon^2}\log n)$ candidate intervals. Since there are b blocks and $O(\log_{1+\epsilon} n)$ choices of Δ, we need to solve $O(\frac{b^2}{\epsilon^3}\log^3 n)$ subproblems by recursion. Thus, the degree of the recursion tree is $O(\frac{b^2}{\epsilon^3}\log^3 n)$.

The dynamic programming at each level can be divided into b steps. At the j-th step, the inputs are the edit distances between block x^j and each of its candidate intervals. The information we need to maintain is an approximation of edit distances between the first $j-1$ blocks of x and the substrings $y[1,l]$ of y, where l is chosen from the set of starting points of the candidate intervals of x^j. There are $O(\frac{b}{\epsilon})$ choices for l and we query the approximated edit distance between x^j and each of its candidate intervals by recursively applying our algorithm. Thus, we only need to maintain $O(\frac{b}{\epsilon})$ values at any time for the dynamic programming.

At the i-th level of recursion, we either invoke one more level of recursion and maintain $O(\frac{b}{\epsilon})$ values where each value takes $O(\log n)$ space, or do an exact computation of edit distance when one of the input strings has length at most b, which takes $O(b\log n)$ space. Hence, the space used at each level is bounded by $O(\frac{b}{\epsilon}\log n)$. There are at most $d=\log_b n$ levels. The aggregated space used by our recursive algorithm is still $O(\frac{b\log^2 n}{\epsilon\log b})$.

We compute the running time by counting the number of nodes in the recursion tree. Notice that the number of nodes at level i is at most $(O(\frac{b^2}{\epsilon^3}\log^3 n))^{i-1}$. For each leaf node, we do exact computation with time $O(\frac{b^2}{\epsilon})$, and the number of leaf nodes is bounded by $(O(\frac{b^2}{\epsilon^3}\log^3 n))^{d-1}$. For each inner node, we run the dynamic programming $O(\log_{1+\epsilon} n)$ times (the number of choices of Δ) which takes $O(\frac{b^3\log^2 n}{\epsilon^4})$ time, and the number of inner nodes is bounded by $(d-1)(O(\frac{b^2}{\epsilon^3}\log^3 n))^{d-2}$.

If we take $b=\log n$ and $\epsilon=\frac{1}{\log n}$, we get a $(1+O(\frac{1}{\log\log n}))$-approximation using $O(\frac{\log^4 n}{\log\log n})$ space and $n^{7+o(1)}$ time. If we take $b=n^\delta$ for $\delta\in(0,\frac{1}{2})$ and ϵ a small constant, we get a $(1+O(\epsilon))$-approximation using $\tilde{O}_{\epsilon,\delta}(n^\delta)$ space and $\tilde{O}_{\epsilon,\delta}(n^2)$ time.

5.2 Improving the Runtime of Streaming ED

We can now use our techniques for small space approximation of ED to reduce the runtime of our algorithm for ED in the asymmetric streaming model proposed in Section 3. Recall that the most time consuming part of our streaming algorithm is finding the optimal window-compatible solution given a series of windows. More specifically, this can be formulated as follows. Suppose we divide the online string s into $b=n^\delta$ windows $\{s^i\}$, and in the recursion our algorithm already finds b windows $\{\bar{s}[l_i,r_i]\}$ in the offline string $\bar{s}$, that are approximately the closest to $\{s^i\}$. We now need to use the windows $\{\bar{s}[l_i,r_i]\}$ to find a substring in $\bar{s}$ that is approximately the closest to s. Previously, this is done by a brute force approach: we try all possible $1\le p_0\le p_1\le\ldots\le p_b\le n+1$ to find the set of p_i's that minimizes $\sum_{i=1}^b \mathsf{ED}(\bar{s}[p_{i-1},p_i),\bar{s}[l_i,r_i])$. Let the optimal set be $\{p_i^*\}$ and record the substring $\bar{s}[l,r]$ of $\bar{s}$ where $l=p_0^*$ and $r=p_b^*-1$. The exponential running time comes from two parts: First, the step of trying all possible $1\le p_0\le p_1\le\ldots\le p_b\le n+1$ needs to examine $\binom{n}{n^\delta+1}$ such choices. Second, when computing $\mathsf{ED}(\bar{s}[p_{i-1},p_i),\bar{s}[l_i,r_i])$, the $O(\log^2 n)$ algorithm from Savitch's theorem uses quasi-polynomial time.

Our main observation now is that, the step of trying all possible $1\le p_0\le p_1\le\ldots\le p_b\le n+1$ to find the set of p_i's that minimizes $\sum_{i=1}^b \mathsf{ED}(\bar{s}[p_{i-1},p_i),\bar{s}[l_i,r_i])$, is equivalent to finding the substring of $\bar{s}$ that minimizes the edit distance to the concatenation of $\bar{s}[l_i,r_i]$ from $i=1$ to b. Thus, instead of trying all possible p_i's, we can try all substrings of y (there

are only $O(n^2)$ such substrings) and compute the edit distance between each substring and the concatenation of the $\bar{s}[l_i, r_i]$'s. Furthermore, instead of an exact computation which either uses $\Omega(n)$ space or $2^{\Omega(\log^2 n)}$ time, we can use our $(1+\epsilon)$-approximation for ED with $\tilde{O}(n^\delta)$ space and $\tilde{O}(n^2)$ time. The approximation factor is now increased to $O((2+\epsilon)^{\frac{1}{\delta}})$, which is still $O(2^{\frac{1}{\delta}})$ if we take ϵ to be small enough. But the running time decreases to $\tilde{O}(n^4)$, and the space complexity remains $\tilde{O}(n^\delta/\delta)$.

6 $1-\epsilon$-Approximation of Streaming LCS

In this section, we design a streaming algorithm for finding a $(1-\epsilon)$ approximation of the LCS using $\tilde{O}(\sqrt{n}/\epsilon)$ memory. We first define the LCSPosition function as below. Given the online string s and the offline string $\bar{s}$, it takes a position p in $\bar{s}$ and a non-negative integer k as inputs. The output is the smallest position q such that $\mathsf{LCS}(\bar{s}[p,q], s[l,r]) \geq k$. If no such q exists, the output is ∞.

For a position p in $\bar{s}$, a substring $s[l,r]$ of s, and a non-negative integer k, we use $\text{LCSPosition}_{l,r}(p,k)$ to denote the result of the mentioned function.[2] It is easy to verify that the LCS of two strings $\bar{s}$ and s is equal to the largest k such that $\text{LCSPosition}_{1,n}(1,k) < \infty$. Therefore, instead of solving the LCS problem, we can solve the $\text{LCSPosition}_{1,n}$ problem and report the largest k such that $\text{LCSPosition}_{1,n}(1,k) < \infty$.

Algorithm 3 Algorithm for approximating the LCS in streaming model.

Data: An offline string $\bar{s}$ of length n, an online string s, and an $\epsilon^* > 0$.
1: Divide s into $\sqrt{n}$ windows $s^*_1, s^*_2, \ldots, s^*_{\sqrt{n}}$ of size $\sqrt{n}$.
2: $D \leftarrow$ an array of size $\lfloor \log_{1+\epsilon^*} n \rfloor$ initially containing ∞ in all cells.
3: **for** $i \in \lceil \sqrt{n} \rceil$ **do**
4: $\quad T \leftarrow$ an array of size $\lfloor \log_{1+\epsilon^*} n \rfloor$ initially containing ∞ in all cells.
5: $\quad$ **for** $0 \leq k \leq \lfloor \log_{1+\epsilon^*} n \rfloor$ **do**
6: $\quad\quad T[k] \leftarrow \text{LCSPosition}_{(i-1)\sqrt{n}+1, i\sqrt{n}}\left(1, \lfloor (1+\epsilon^*)^k \rfloor\right)$.
7: $\quad\quad$ **for** $0 \leq k_1 \leq k$ **do**
8: $\quad\quad\quad$ **if** $D[k_1] < \infty$ **then**
9: $\quad\quad\quad\quad$ Find $\text{LCSPosition}_{(i-1)\sqrt{n}+1, i\sqrt{n}}\left(D[k_1]+1, \lfloor (1+\epsilon^*)^k \rfloor - \lfloor (1+\epsilon^*)^{k_1} \rfloor\right)$ using any offline algorithm. Let q be this result.
10: $\quad\quad\quad\quad T[k] \leftarrow \min\{T[k], q\}$.
11: $\quad D \leftarrow T$.
12: **return** The largest value $\lfloor (1+\epsilon^*)^k \rfloor$ such that $D[k] < \infty$.
13: $\quad$ **return** 0 if no such k exists.

We now describe our algoirthm. The pseudocode is given in algorithm 3. It first divides the online string into $\sqrt{n}$ windows of equal sizes. We assume w.l.o.g., that length of the strings is divisible by $\sqrt{n}$. Otherwise we can always pad offline and online strings with different characters that are not in Σ such that their new length get divisible by $\sqrt{n}$. The algorithm divides s into $\sqrt{n}$ windows $s^*_1, s^*_2, \cdots, s^*_{\sqrt{n}}$ each with the size of $\sqrt{n}$ where s^*_i is the substring $s[(i-1)\sqrt{n}+1, i\sqrt{n}]$. Given an $\epsilon^* > 0$, the algorithm keeps an array D of the size $\lfloor \log_{1+\epsilon^*} n \rfloor$ where $D[k]$ is an estimation of $\text{LCSPosition}(1, \lfloor (1+\epsilon^*)^k \rfloor)$ in the subsequence of the online string that has arrived so far in the stream. Specifically, after arrival of the window

[2] We also define $\text{LCSPosition}_{l,r}(p,0)$ to be $p-1$.

s_i^* in the stream, the algorithm keeps an estimation of $\text{LCSPosition}_{1,i\sqrt{n}}(1, \lfloor(1+\epsilon^*)^k\rfloor)$ in $D[k]$. Please see the full version [36] for more details on how we update the array D upon arrival of a new window and why we can achieve the approximation guarantee. We have the following Theorem. The full proof is deferred to [36].

▶ **Theorem 13.** *There exists a single-pass deterministic streaming algorithm that finds a* $(1-\epsilon)$ *approximation of the LCS between* $\bar{s}$ *and* s *using* $\tilde{O}(\sqrt{n}/\epsilon)$ *memory and polynomial time.*

7 Space Efficient Algorithms for LIS in the Non-Streaming Model

We now consider approximating the LIS of a string $x \in \Sigma^n$ where the alphabet Σ has a total order. We assume each symbol in Σ can be stored with $O(\log n)$ space. Let ∞ and $-\infty$ be two special symbols such that for any symbol $\sigma \in \Sigma$, $-\infty < \sigma < \infty$. We denote the length of the longest increasing subsequence of x by $\mathsf{LIS}(x)$.

Again our algorithm is a recursive one, and in each recursion we use an approach similar to the deterministic streaming algorithm from [41] that gives a $1-\epsilon$ approximation of $\mathsf{LIS}(x)$ with $O(\sqrt{n/\epsilon}\log n)$ space. Before describing their approach, we first give a brief introduction to a classic algorithm for LIS, known as *PatienceSorting*. The algorithm initializes a list P with n elements such that $P[i] = \infty$ for all $i \in [n]$, and then scans the input sequence x from left to right. When reading a new symbol x_i, we find the smallest index l such that $P[l] \geq x_i$ and set $P[l] = x_i$. After processing the string x, for each i such that $P[i] < \infty$, we know $\sigma = P[i]$ is the smallest possible character such there is an increasing subsequence in x of length i ending with σ. Finally the algorithm returns the largest index l such that $P[l] < \infty$. This computes LIS exactly in $O(n \log n)$ time and $O(l \log n)$ space (see [5] for more details).

In the streaming algorithm from [41], we maintain a set S and a list Q, such that, $Q[i]$ is stored only for $i \in S$ and $S \subseteq [n]$ is a set of size $O(\sqrt{n})$. We use S and Q as an approximation to the list P in *PatienceSorting* in the sense that for each $s \in S$, there is an increasing subsequence in x of length s ending with $Q[s]$. More specifically, we can generate a list P' from S and Q such that $P'[i] = Q[j]$ for the smallest $j \geq i$ that lies in S. For i larger than the maximum element in S, we set $P'[i] = \infty$. Each time we read a new element, we update Q and S accordingly, which is equivalent to doing *PatienceSorting* on the list P'. When S gets larger than $2\sqrt{n}$, we do a cleanup to S by only keeping $\sqrt{n}/\epsilon$ evenly picked values from 1 to $\max S$ and storing $Q[s]$ for $s \in S$. This loses at most $\frac{\epsilon}{\sqrt{n}}\mathsf{LIS}(x)$ in the length of the longest increasing subsequence detected. Since we only do $O(\sqrt{n})$ cleanups, we are guaranteed an increasing subsequence of length at least $(1-\epsilon)\mathsf{LIS}(x)$.

We now modify the above algorithm into another form. This time we first divide x evenly into many small blocks. Meanwhile, we also maintain a set S and a list Q. We now process x from left to right, and update S and Q each time we have processed one block of x. If the number of blocks in x is small, we can get the same approximation as in [41] with S and Q having smaller size. For example, we can divide x into $n^{1/3}$ blocks each of size $n^{2/3}$, and we update S and Q once after processing each block. If we do exact computation within each block, we only need to maintain the set S and the list Q of size $O(\frac{n^{1/3}}{\epsilon})$. We can still get a $(1-\epsilon)$ approximation, because we do $n^{1/3}$ cleanups and for each cleanup, we lose about $\frac{\epsilon}{n^{1/3}}\mathsf{LIS}(x)$ in the length of the longest increasing subsequence detected.

This almost already gives us an $\tilde{O}_\epsilon(n^{1/3})$ space algorithm, except the exact computation within each block needs $O(n^{2/3}\log n)$ space. A natural idea to reduce the space complexity is to replace the exact computation with an approximation. When each block x^i has size $n^{2/3}$, running the approximation algorithm from [41] takes $O(\frac{n^{1/3}}{\epsilon}\log n)$ space and thus we

can hope to reduce the total space required to $O(\frac{n^{1/3}}{\epsilon}\log n)$. However, a problem here is that the approximation algorithm on each block x^i only gives us an approximation of $\mathsf{LIS}(x^i)$. This alone does not give us enough information on how to update S and Q. Also, for a longest increasing subsequence of x, say τ, the subsequence of τ that lies in the block x^i may be much shorter than $\mathsf{LIS}(x^i)$. This subsequence of τ may be ignored if we run the approximation algorithm instead of using exact computation.

We now give some intuition of our approach to fix these issues. Let us consider a longest increasing subsequence τ of x such that τ can be divided into many parts, where the i-th part τ^i lies in x^i. We denote the length of τ^i by d_i. Let the first symbol of τ^i be α_i and the last symbol be β_i if τ^i is not empty. When we process the block x^i, we want to make sure that our algorithm can detect an increasing subsequence of length very close to d_i in x^i, where the first symbol is at least α_i and the last symbol is at most β_i. We can achieve this by running a bounded version of the approximation algorithm which only considers increasing subsequences no longer than d_i. Since we do not know α_i or d_i in advance, we can guess α_i by trying every symbol in $Q[s]$ where one of them is close enough to α_i. For d_i, we can try $O(\log_{1+\epsilon} n)$ different values of l such that one of them is close enough to d_i. In this way, we are guaranteed to detect a good approximation of τ_i.

Based on the above intuition, we now introduce our space-efficient algorithm for LIS called **ApproxLIS**. The pseudocode is given in Algorithm 4. It takes three inputs, a string $x \in \Sigma^*$, two parameters b and ϵ. We also introduce a slightly modified version of **ApproxLIS** called **ApproxLISBound**. It takes an additional input l, which is an integer at most n. We want to guarantee that if the string x has an increasing subsequence of length l ending with $\alpha \in \Sigma$, then **ApproxLISBound**(x, b, ϵ, l) can detect an increasing subsequence of length close to l ending with some symbol no larger than α. The idea is to run **ApproxLIS** but only consider increasing subsequence of length at most l. **ApproxLISBound** has the same space and time complexity as **ApproxLIS**.

We now describe **ApproxLIS**. When the input string x has length at most b^2, we compute an $(1-\epsilon)$-approximation of LIS using the algorithm in [41] with $O(\frac{b}{\epsilon}\log n)$ space. Otherwise, we divide the input string into b blocks each of length $\frac{n}{b}$. Similar to the streaming algorithm in [41], we maintain two sets S and Q of size $O(\frac{b}{\epsilon})$ as an approximation of the list P when running *PatienceSorting*. We will show that it is enough to use $O(\frac{b}{\epsilon}\log n)$ bits for S and Q, because we only update them b times and we lose about $O(\frac{\epsilon}{b})\,\mathsf{LIS}(x)$ after each update. Initially, S contains only one element 0 and $Q[0] = -\infty$. We update S and Q after processing each block of x as follows.

For simplicity, we denote S and Q after processing the t-th block by S_t and Q_t. To see how S and Q are updated, we take the t-th update as an example. Given S_{t-1} and Q_{t-1}, we first determine the length of the LIS in $x^1 \circ \cdots \circ x^t$ that can be detected based on S_{t-1} and Q_{t-1}. We denote this length by k_t. Notice that, for each $s \in S_{t-1}$, we know there is an increasing subsequence in $x^1 \circ \cdots \circ x^{t-1}$ of length s ending with $Q_{t-1}[s] \in \Sigma$. This gives us $|S_{t-1}|$ increasing subsequences. The idea is to find the best extension of these increasing subsequences in the block x^t and see which one gives us the longest increasing subsequence of $x^1 \circ \cdots \circ x^t$. Since each block is of size $\frac{n}{b}$, we cannot always afford to do exact computation. Thus we compute an approximation of the LIS by recursively calling **ApproxLIS** itself. For each $s \in S_{t-1}$, we run **ApproxLIS**(z^s, b, ϵ) where z^s is the subsequence of x^t with only symbols larger than $Q_{t-1}[s]$. Finally, we let $k_t = \max_{s\in S_{t-1}}(s + \textbf{ApproxLIS}(z^s, b, \epsilon))$. Given k_t, we then set S_t to be the $\frac{b}{\epsilon}$-th evenly picked integers from 0 to k_t.

The next step is to compute Q_t. We first set $Q_t[s] = \infty$ for all $s \in S_t$ except $s = 0$, and we set $Q_t[0] = -\infty$. Then, for each $s \in S_{t-1}$ and $l = 1, 1+\epsilon, (1+\epsilon)^2, \ldots, k_t - s$, we run **ApproxLISBound**(z^s, b, ϵ, l). For each $s' \in S_t$ such that $s \le s' \le s + l$, we update $Q_t[s']$

Algorithm 4 Algorithm **ApproxLIS** (**ApproxLISBound**) for approximating LIS.

Data: A string x , parameters b and ϵ. And an additional parameter l for **ApproxLISBound**

if $|x| \leq b^2$ **then**
  compute an $(1-\epsilon)$-approximation of $\mathsf{LIS}(x)$ with the streaming algorithm from [41] using $O(\frac{b}{\epsilon}\log n)$ space. (For **ApproxLISBound**, we only consider LIS of length at most l.)
  return
divide x evenly into b blocks such that $x = x^1 \circ x^2 \circ \cdots \circ x^b$. ▷ $|x^i| \leq \lceil n/b \rceil$
initialize $S = \{0\}$ and $Q[0] = -\infty$.
for $i = 1$ **to** b **do**
  $k = 0$.
  for all $s \in S$ **do**
    let z be the subsequence of x^i by only considering the elements larger than $Q[s]$.
    $d = \textbf{ApproxLIS}(z, b, \epsilon)$.
    $k = \max\{k, s+d\}$. (For **ApproxLISBound**, we let $k = \min\{l, \max\{k, s+d\}\})$.
  if $k \leq b/\epsilon$ **then**
    let $S' = \{0, 1, 2, \ldots, k\}$.
  else
    let $S' = \{0, \frac{\epsilon}{b}k, 2\frac{\epsilon}{b}k, \ldots, k\}$. ▷ *evenly pick* $b/\epsilon + 1$ *integers from* 0 *to* k *(including* 0 *and* k*).*
  $Q'[s] = \infty$ for all $s' \in S'$ except $Q'[0] = -\infty$.
  for all $s \in S$ **do**
    let z be the subsequence of x^i by only considering the elements larger than $Q[s]$.
    for all $l = 1, 1+\epsilon, (1+\epsilon)^2, \ldots, k-s$ **do**
      $\tilde{S}, \tilde{Q} \leftarrow \textbf{ApproxLISBound}(z, b, \epsilon, l)$.
      for each $s' \in S'$ such that $s \leq s' \leq s+l$, let $\tilde{s}$ be the smallest element in $\tilde{S}$ that is larger than $s'-s$ and set $Q'[s'] = \min\{\tilde{Q}[\tilde{s}], Q'[s']\}$.
  $S \leftarrow S'$, $Q \leftarrow Q'$.
return $\max S$. (for **ApproxLISBound**, we return the set S and Q)

if **ApproxLISBound**(z^s, b, ϵ, l) detects an increasing subsequence of length at least $s'-s$ ending with a symbol smaller than the old $Q_t[s']$. The intuition is that, with the bound l, we may be able to find a smaller symbol in Σ such that there is an increasing subsequence of length l ending with it. This information can be easily ignored if l is a lot smaller than the actual length of LIS in x^t. To see why this is important, let τ be a longest increasing subsequence of x, and let τ^t be the part of τ that lies in the block x^t. The length of τ^t may be much smaller than the length of LIS in x^t. When the bound l is close to $|\tau^t|$, we will be able to detect a good approximation of τ^t by running **ApproxLISBound**(z^s, b, ϵ, l) on z^s for each $s \in S_{t-1}$. Since we do not know the length of τ^t, we will guess it by trying $O_\epsilon(\log n)$ values of l and always record the optimal $Q_t[s]$ for $s \in S_t$.

Continue doing this, we get S_b and Q_b. **ApproxLIS** outputs the largest element in S_b.

ApproxLIS is recursive. We denote the depth of recursion by d, and it can be seen that d is at most $\log_b n - 1$. To see the correctness of our algorithm, given fixed b, ϵ, we show the output at the r-th recursive level is a $(1 - O((d-r)\epsilon))$-approximation. Thus, the final output will be a $(1 - O(\epsilon \log_b n))$-approximation to $\mathsf{LIS}(x)$.

The proof is by induction on r from d to 1. For the base case of $r = d$, the statement follows from [41]. Now consider the computation at the r-th level. For convenience, we denote the input string by x, which has length at most $\frac{n}{b^{r-1}}$. Let τ be an increasing subsequence of x where τ^i lies in x^i. For our analysis, let P'_t be the list generated by S_t and Q_t in the following way: for every i let $P'_t[i] = Q_t[j]$ for the smallest $j \geq i$ that lies in S_t. If no such j exists, set $P'_t[i] = \infty$. Correspondingly, P_t is the list P after running *PatienceSorting* with input $x^1 \circ x^2 \circ \cdots \circ x^t$.

Let $h_t = \sum_{j=1}^{t} |\tau^j|$ and $k_t = \max S_t$, our main observation is the following inequality:

$$P'_t[(1 - 3(d - (r+1)\epsilon - \epsilon)h_t - 2t\frac{\epsilon}{b}k_t] \leq P_t[h_t] \tag{4}$$

Note that $h_b = |\tau| = \mathsf{LIS}(x)$. We have $P_b[h_b] < \infty$ by the correctness of *PatienceSorting*. If inequality 4 holds, then by $k_t \leq h_t$, there must exist an element in S_b larger than $(1 - 3(d-r)\epsilon)\,\mathsf{LIS}(x)$ which gives the correctness of the computation at the r-th level.

We prove inequality 4 by induction on t. The intuition is that, at the t-th update, if inequality 4 holds for $t-1$, we know that there must exist an $s \in S_{t-1}$ that is close to h_{t-1} and $Q_{t-1}[s] \leq P_{t-1}[h_{t-1}] = \beta_{t-1} < \alpha_t$. By trying $l = 1, 1+\epsilon, (1+\epsilon), \ldots, k_t - s$, one l is close enough to $|\tau^t|$. Thus we are guaranteed to detect a good approximation of τ_t in x^t and the inequality also holds for t.

At each recursive level, we need to maintain the sets S and Q with space $O(\frac{b}{\epsilon}\log n)$. Thus the total space is $O(d\frac{b}{\epsilon}\log n) = O(\frac{b\log^2 n}{\epsilon \log b})$. The analysis of time complexity is similar to the case of edit distance, where we analyze the recursion tree and bound the number of nodes.

If we take $b = \log n$ and $\epsilon = \frac{1}{\log n}$, we get a $(1 - O(\frac{1}{\log\log n}))$-approximation algorithm using $O(\frac{\log^4 n}{\log\log n})$ space and $O(n^{5+o(1)})$ time. Let $\delta \in (0, \frac{1}{2})$ be a constant such that $\frac{1}{\delta}$ is an integer. If we take $b = n^\delta$ and ϵ a small constant, we get a $(1 + O(\epsilon))$-approximation algorithm using $\tilde{O}_{\epsilon,\delta}(n^\delta)$ space and $\tilde{O}_{\epsilon,\delta}(n^{2-2\delta})$ time.

8 Space Efficient ALgorithms for LCS in the Non-Streaming Model

Our algorithm for LCS is based on a reduction to LIS. Given input strings x and y, for each $i \in [n]$ let $b^i \in [m]^*$ be the sequence consisting of all distinct indices j in $[m]$ such that $x_i = y_j$, arranged in descending order. Note that b^i may be empty. Let $z = b^1 \circ b^2 \circ \cdots \circ b^n$, which has length $O(mn)$ since each sequence b^i is of length at most m. We claim that $\mathsf{LIS}(z) = \mathsf{LCS}(x, y)$. This is because for every increasing subsequence of z, say $t = t_1 t_2 \cdots t_d$, the corresponding subsequence $y_{t_1} y_{t_2} \cdots y_{t_d}$ of y also appears in x. Conversely, for every common subsequence of x and y, we can find an increasing subsequence in z with the same length. We call this procedure ReduceLCStoLIS. Note that in our algorithms, z need not be stored, since we can compute each element in z as necessary in logspace by querying x and y. Thus our reduction is a logspace reduction.

Based on the reduction, we can use similar techniques as we used for LIS. Specifically, let $z = \textbf{ReduceLCStoLIS}(x, y)$. We call our space efficient algorithm for LCS **ApproxLCS**. If one input string is shorter than the parameter b, we know $\mathsf{LCS}(x, y) \leq b$ and we can compute $\mathsf{LIS}(z)$ using *PatienceSorting* with $O(b \log n)$ space. Otherwise, the goal is to compute an approximation of $\mathsf{LIS}(z)$. One difference here is, instead of dividing z evenly into b blocks, we divide z according to x. That is, we first divide x evenly into b blocks, and then divide z into b blocks such that $z^i = \text{ReduceLCStoLIS}(x^i, y)$. This gives us a slight improvement on running time over the naive approach of running our LIS algorithm on z. Note that $\mathsf{LIS}(z^i)$ is at most $\frac{n}{b}$ since the length of x^i is $\frac{n}{b}$. We compute an approximation of $\mathsf{LIS}(z^i)$ by recursively calling **ApproxLCS** with inputs x, y, b, ϵ. The input size to the next recursive level is decreased by a factor of b. The remaining analysis is similar to the case of LIS.

References

1 Amir Abboud and Arturs Backurs. Towards hardness of approximation for polynomial time problems. In *ITCS*, 2017.

2 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *FOCS*, 2015.

3 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. In *STOC*, 2016.

4 Amir Abboud and Aviad Rubinstein. Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds. In *ITCS*, 2018.

5 David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.

6 Carlos ER Alves, Edson N Cáceres, and Siang Wun Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45(3):301–335, 2006.

7 A. Andoni, M. Deza, A. Gupta, P. Indyk, and S. Raskhodnikova. Lower bounds for embedding edit distance into normed spaces. In *SODA*, 2003.

8 Alexandr Andoni, Assaf Goldberger, Andrew McGregor, and Ely Porat. Homomorphic fingerprints under misalignments: Sketching edit and shift distances. In *STOC*, 2013.

9 Alexandr Andoni and Robert Krauthgamer. The computational hardness of estimating edit distance [extended abstract]. In *FOCS*, 2007.

10 Alexandr Andoni and Robert Krauthgamer. The smoothed complexity of edit distance. In *ICALP*, 2008.

11 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *FOCS*, 2010.

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

13 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. In *STOC*, 2009.

14 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *STOC*, 2015.

15 Nikhil Bansal, Moshe Lewenstein, Bin Ma, and Kaizhong Zhang. On the longest common rigid subsequence problem. *Algorithmica*, 56(2):270–280, February 2010. doi:10.1007/s00453-008-9175-1.

16 Ziv Bar-Yossef, TS Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *FOCS*, 2004.

17 Tuğkan Batu, Funda Ergun, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *SODA*, 2006.

18 Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In *FOCS*, 2016.

19 Richard Bellman. Dynamic programming (dp), 1957.

20 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. In *SODA*, 2018.

21 Mahdi Boroujeni and Saeed Seddighin. Improved MPC algorithms for edit distance and Ulam distance. In *SPAA*, 2019.

22 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.

23 Karl Bringman and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *SODA*, 2018.

24 Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly sub-cubic algorithms for language edit distance and RNA-folding via fast bounded-difference min-plus product. In *FOCS*, 2016.

25 Karl Bringmann and Marvin Kunnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *FOCS*, 2015.

26 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucky, and Michael Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *FOCS*, 2018.

27 Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for computing edit distance without exploiting suffix trees. *arXiv preprint*, 2016. `arXiv:1607.03718`.

28 Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *STOC*, 2016.

29 Moses Charikar, Ofir Geri, Michael P Kim, and William Kuszmaul. On estimating edit distance: Alignment, dimension reduction, and embeddings. In *ICALP*, 2018.

30 Lijie Chen, Shafi Goldwasser, Kaifeng Lyu, Guy N Rothblum, and Aviad Rubinstein. Fine-grained complexity meets IP=PSPACE. In *SODA*, 2019.

31 Kuan Cheng, Zhengzhong Jin, Xin Li, and Yu Zheng. Space efficient deterministic approximation of string measures. *CoRR*, abs/2002.08498, 2020. `arXiv:2002.08498`.

32 Maxime Crochemore, Costas S Iliopoulos, Yoan J Pinzon, and James F Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.

33 Maxime Crochemore, Gad M Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003.

34 J Boutet de Monvel. Extensive simulations for longest common subsequences. *The European Physical Journal B-Condensed Matter and Complex Systems*, 7(2):293–308, 1999.

35 Funda Ergün and Hossein Jowhari. On distance to monotonicity and longest increasing subsequence of a data stream. In *SODA*, 2008.

36 Alireza Farhadi, MohammadTaghi Hajiaghayi, Aviad Rubinstein, and Saeed Seddighin. Streaming with oracle: New streaming algorithms for edit distance and LCS. *CoRR*, abs/2002.11342, 2020. `arXiv:2002.11342`.

37 Anna Gál and Parikshit Gopalan. Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence. In *FOCS*, 2007.

38 Minos Garofalakis and Amit Kumar. Correlating XML data streams using tree-edit distance embeddings. In *PODS*, 2003.

39 Omer Gold and Micha Sharir. Dynamic time warping and geometric edit distance: Breaking the quadratic barrier. In *ICALP*, 2017.

40 Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance. In *FOCS*, 2019.

41 Parikshit Gopalan, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Estimating the sortedness of a data stream. In *SODA*, 2007.

42 Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

43 Bernhard Haeupler, Aviad Rubinstein, and Amirbehshad Shahrasbi. Near-linear time insertion-deletion codes and $(1+\epsilon)$-approximating edit distance via indexing. In *STOC*, 2019.

44 MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating lcs in linear time: Beating the $\sqrt{n}$ barrier. In *SODA*, 2019.

45 MohammadTaghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively parallel approximation algorithms for edit distance and longest common subsequence. In *SODA*, 2019.

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

47 Piotr Indyk. Algorithmic applications of low-distortion geometric embeddings. In *FOCS*, 2001.

48 Rajesh Jayaram and Barna Saha. Approximating language edit distance beyond fast matrix multiplication: Ultralinear grammars are where parsing becomes hard! In *ICALP*, 2017.

49 Ce Jin, Jelani Nelson, and Kewen Wu. An improved sketching algorithm for edit distance. In *38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

50 Masashi Kiyomi, Takashi Horiyama, and Yota Otachi. Longest common subsequence in sublinear space. *arXiv preprint*, 2020. `arXiv:2009.08588`.

51 Masashi Kiyomi, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, and Jun Tarui. Space-efficient algorithms for longest increasing subsequence. *Theory of Computing Systems*, pages 1–20, 2018.

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

53 William Kuszmaul. Dynamic time warping in strongly subquadratic time: Algorithms for the low-distance regime and approximate evaluation. *arXiv preprint*, 2019. `arXiv:1904.09690`.

54 Gad M Landau, Eugene W Myers, and Jeanette P Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

55 Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, MA, 2001.

56 David Liben-Nowell, Erik Vee, and An Zhu. Finding longest increasing and common subsequences in streaming data. In *COCOON*, 2005.

57 William J Masek and Michael S Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

58 Timothy Naumovitz and Michael Saks. A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity. *SODA*, 2014.

59 Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. In *STOC*, 2005.

60 Aviad Rubinstein and Zhao Song. Reducing approximate longest common subsequence to approximate edit distance. *arXiv preprint*, 2019. `arXiv:1904.05451`.

61 Aviad Runbinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for LCS and LIS with truly improved running times. In *FOCS*, 2019.

62 Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *FOCS*, 2015.

63 Barna Saha. Fast & space-efficient approximations of language edit distance and RNA folding: An amnesic dynamic programming approach. In *FOCS*, 2017.

64 Michael Saks and C Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *SODA*, 2013.

65 Walter J Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.

66 Xiaoming Sun and David P. Woodruff. The communication and streaming complexity of computing the longest common and increasing subsequences. In *SODA*, 2007.

67 Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.