How Compression and Approximation Affect Efficiency in String Distance Measures

Arun Ganesh*† Tomasz Kociumaka*‡ Andrea Lincoln*‡§ Barna Saha*‡

Abstract

Real-world data often comes in compressed form. Analyzing compressed data directly (without first decompressing it) can save space and time by orders of magnitude. In this work, we focus on fundamental sequence comparison problems and try to quantify the gain in time complexity when the underlying data is highly compressible. We consider grammar compression, which unifies many practically relevant compression schemes such as the Lempel–Ziv family, dictionary methods, and others. For two strings of total length N and total compressed size n, it is known that the edit distance and a longest common subsequence (LCS) can be computed exactly in time $\widetilde{O}(nN)$, as opposed to $O(N^2)$ for the uncompressed setting. Many real-world applications need to align multiple sequences simultaneously, and the fastest known exact algorithms for median edit distance and LCS of k strings run in $O(N^k)$ time, whereas the one for center edit distance has a time complexity of $O(N^{2k})$. This naturally raises the question if compression can help to reduce the running time significantly for $k \geq 3$, perhaps to $O(N^{k/2}n^{k/2})$ or, more optimistically, to $O(Nn^{k-1})$.[1]

Unfortunately, we show new lower bounds that rule out any improvement beyond $\Omega(N^{k-1}n)$ time for any of these problems assuming the Strong Exponential Time Hypothesis (SETH), where again N and n represent the total length and the total compressed size, respectively. This answers an open question of Abboud, Backurs, Bringmann, and Künnemann (FOCS'17).

In presence of such negative results, we ask if allowing approximation can help, and we show that approximation and compression together can be surprisingly effective for both multiple and two strings.

We develop an $\widetilde{O}(N^{k/2}n^{k/2})$-time FPTAS for the median edit distance of k sequences, leading to a saving of nearly half the dimensions for highly-compressible sequences. In comparison, no $O(N^{k-\Omega(1)})$-time PTAS is known for the median edit distance problem in the uncompressed setting. We obtain an improvement from $\widetilde{O}(N^{2k})$ to $\widetilde{O}(N^{k/2+o(k)}n^{k/2})$ for the center edit distance problem. For two strings, we get an $\widetilde{O}(N^{2/3}n^{4/3})$-time FPTAS for both edit distance and LCS; note that this running time is $o(N)$ whenever $n \ll N^{1/4}$. In contrast, for uncompressed strings, there is not even a subquadratic algorithm for LCS that has less than polynomial gap in the approximation factor. Building on the insight from our approximation algorithms, we also obtain several new and improved results for many fundamental distance measures including the edit, Hamming, and shift distances.

1 Introduction

With the information explosion, almost all real-world data comes in a compressed form. While compression is primarily intended to save storage space and transmission bandwidth, processing compressed data directly often provides an opportunity to reduce computation time and energy by several orders of magnitude. In this work, we focus on sequential data such as natural-language texts, biological sequences (nucleic acid sequences, including DNA, and amino acid sequences, including proteins), and computer codes. Sequential data often contains highly repetitive pattern. Modern technology (e.g., high-throughput sequencing) has led to an astonishingly rapid accumulation of such data, so much so that without proper data compression and algorithms over compressed data, it is not possible to utilize the wealth of information in them [BPS13, BDY16, GWH19, HPWO19].

Grammar compression represents strings as *straight-line programs* (SLPs), and provides a mathematically elegant way to unify algorithm design principles for processing compressed data [Loh12]. It is equivalent to many well-known compression schemes up to logarithmic factors and moderate constants [Ryt03, KP18, KK20] such as the celebrated LZ77 [ZL77] and RLBWT [BW94] schemes, and at least as strong as byte-pair encoding [Gag94], Re-Pair [LM00], Sequitur [NW97], further members of the Lempel–Ziv family [ZL78, Wel84], and many more

*University of California, Berkeley.

†Supported in part by an NSF 1816861 grant.

‡Supported in part by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.

§Supported in part by a Simons NTT Research Fellowship.

[1]In this paper, we assume that k is a constant; thus, the $O(\cdot)$ and $\Omega(\cdot)$ notation may hide factors with exponential dependence on k.

popular schemes (the list keeps growing). Therefore, following the lead of a large body of previous work (including [Tis15, Jeż15, ABBK17, BWK19, CKW20]), we work with grammar-compressed data.

In this work, we ask whether fundamental sequence similarity measures can be computed faster for compressed data. This research is motivated in part by the success of computing edit distance and longest common subsequence (LCS) of two strings [Gaw12, HLLW13, Tis15] much faster than the "decompress-and-solve" approach: If we let N denote the total length and n denote the total compressed size of the input strings, then the edit distance and the LCS length can be computed exactly in time $\widetilde{O}(nN)$ in contrast to $O(N^2)$ time for the uncompressed setting. Therefore, for highly compressible sequences where, say, $n = \operatorname{polylog} N$, the running time reduces to $\widetilde{O}(N)$. Abboud, Backurs, Bringmann, and Künnemann [ABBK17] asked whether it is possible to improve upon $\widetilde{O}(nN)$, noting that: "For example, an $O(n^2N^{0.1})$ bound could lead to major real-world improvements." In general, any sublinear dependency on N would be preferable; unfortunately, [ABBK17] shows that $\widetilde{O}(Nn)$ is essentially optimal under the Strong Exponential Time Hypothesis (SETH).

There are many real-world applications which deal with multiple sequences. A survey by Nature [NMN14] reports *multiple sequence alignment* as one of the most used modeling methods in biology, with [THG94] among the top-10 papers cited of all time (citation count 63105). Some of the basic measures for multiple sequence similarity include the LCS length and the cost of the median and center strings under edit distance. Abboud, Backurs, and V.-Williams [ABV15] showed that exact computation of k-LCS requires $\Omega(N^{k-o(1)})$ time (under SETH), and a similar result has been recently shown for both median and center k-edit distance [HBGT20]. A simple extension of the basic dynamic programming for two strings solves the median k-edit distance problems in $O(N^k)$ time whereas the best bound known for the center k-edit distance is $O(N^{2k})$ [NR05]. The two-string lower bound in the compressed setting leaves open the possibility of reducing the running times of the k-LCS and the median k-edit distance problems for compressed strings: It might be feasible to achieve runtimes of $O(N^{k/2}n^{k/2})$ or even $O(Nn^{k-1})$, and a substantial reduction of the exponent at N could lead to significant savings. This raises the following questions:

1. *Does compression allow for significantly reducing the running time for multi-sequence similarity problems?*
2. *For the case of two highly compressible strings, what relaxations of the LCS and the edit distance problems could allow circumventing the lower bounds and achieving sublinear dependency on N?*

Lower Bounds: Compression does not help with exact bounds much! Unfortunately, we show that computing the k-LCS, median k-edit distance, and center k-edit distance all require $\Omega((N^{k-1}n)^{1-o(1)})$ time under SETH. Therefore, the potential gain from compression becomes insignificant as k grows. Intuitively, SETH states that CNF-satisfiability requires $2^{n-o(n)}$ time [IP01]. Even more specifically, we use the k-Orthogonal Vectors problem (k-OV) [Vas18]. At a high level, k-OV takes as input a list L with n zero-one vectors of dimension d. We must return YES if there exist k vectors that, when multiplied element-wise, form the all zeros vector. The k-OV conjecture, which is implied by SETH, states that k-OV cannot be solved in $O(n^{k-\Omega(1)})$ time.

THEOREM 1.1. *If the k'-OV hypothesis is true for all constants k', then for any constant $\epsilon \in (0, 1]$ grammar-compressed k-LCS requires $\left(N^{k-1}n\right)^{1-o(1)}$ time when the alphabet size is $|\Sigma| = \Theta(k)$ and $n = N^{\epsilon \pm o(1)}$. Here, N denotes the total length of the k input strings and n is their total compressed size.*

We prove similar lower bounds for median and center k-edit distance (Theorem 6.2 and Theorem 6.8). Sections 6.2, 6.6, and 6.9 contain our lower bound results.

Abboud, Backurs, Bringmann, and Künnemann [ABBK17] left an open question whether their $\Omega((Nn)^{1-o(1)})$ lower bound for LCS also holds for computing the edit distance of two strings. We answer this question affirmatively and extend the argument to the k-string setting. Moreover, we note that for a seemingly simpler problem of computing the shift distance [AIK08, AIKH13, AGMP13, GKK+20], we show that compression does not help to reduce even a single dimension (Section 8).

Algorithms: Effectiveness of Approximation & Compression. In presence of such negative results, relaxing the median and center k-edit distance to circumvent the $\Omega((N^{k-1}n)^{1-o(1)})$ lower bound becomes even more important.

Can we use compression and approximation together to achieve much better approximation guarantees and, simultaneously, circumvent the exact computation lower bounds?

To the best of our knowledge, even for two strings, there is no previous work on approximating the edit distance of grammar-compressed strings. On the other hand, even after a long line of research in developing fast algorithms for approximate edit distance for the uncompressed setting (see e.g. [BEK+03, BJKK04, BES06, AKO10, AO12, CDG+18, GRS20, BR20, KS20, AN20]), the best approximation ratio achievable in truly subquadratic time is currently $3+\epsilon$ [GRS20], and the fastest constant-factor approximation algorithm runs in $O(n^{1+\epsilon})$ time [AN20] with an approximation factor that has doubly-exponential dependence on $\frac{1}{\epsilon}$. The situation is even worse for LCS approximation, where we do not know how to design a subquadratic algorithm with sub-polynomial approximation gap [HSSS19, RSSS19]. We are also unaware of any previous research on approximating LCS of two compressed strings.

In the case of multiple strings, there is a classic $O(N^2)$-time $(2-2/k)$-approximation for median edit distance and an $O(N^2)$-time 2-approximation for center edit distance [Gus97]. Combined with the results of [AN20], this yields an $O(N^{1+\epsilon})$-time constant-factor approximation for both versions. Nevertheless, a PTAS, that is, a $(1+\epsilon)$-approximation algorithm for every constant $\epsilon > 0$, would be much more desirable for practical applications.

Surprisingly, we show that already when an $(1+\epsilon)$-approximation is allowed for an arbitrary constant $\epsilon > 0$, the median k-edit distance computation time reduces to $\widetilde{O}(N^{k/2}n^{k/2})$ compared to the $\Omega((N^{k-1}n)^{1-o(1)})$ lower bound for exact algorithms. In other words, we can save $k/2$ dimensions by allowing approximation and compression. For $\epsilon = o(1)$, the running time of our algorithm increases by an $\epsilon^{-O(k)}$ factor, so we even obtain an FPTAS whereas no prior work in the uncompressed setting gives a $(1+\epsilon)$-approximation in $O(N^{k-\Omega(1)})$ time. The reduction in time for center k-edit distance is even more dramatic (and technically more complex) from exact $O(N^{2k})$ to $\widetilde{O}(N^{k/2+o(k)}n^{k/2})$ for a $(1+\epsilon)$ approximation.

For edit distance between two strings, we develop a more efficient FPTAS whose running time is $\widetilde{O}(N^{2/3}n^{4/3}\epsilon^{-1/3})$, which is sublinear in N as long as $n \ll N^{1/4}$. A slightly more sophisticated $\widetilde{O}(N^{2/3}n^{4/3}\epsilon^{-1/3})$-time algorithm also provides a $(1+\epsilon)$-approximation of the LCS length. In contrast, a comparable result for the uncompressed setting is an $O(N^{1.95})$-time algorithm of [RSSS19], which returns a common subsequence of length $\Omega(N/\lambda^4)$, providing an $O(\lambda^3)$-factor approximation. Even when the alphabet size is 2, so far, there does not exist any $(1+\epsilon)$ approximation in subquadratic time [RS20].

Improved Exact Algorithms in Compressed Setting. Interestingly, the insights behind our approximation algorithms also lead to new *exact algorithms*. In particular, we show that the edit distance can be computed in time $\widetilde{O}(n\sqrt{ND})$, where D is an upper bound on the edit distance. This improves upon the state-of-the-art bound of $\widetilde{O}(\min(nN, n+D^2))$ [Tis15, LV88, MSU97] whenever $D \gg N^{1/3}n^{2/3}$.

For this problem, the first improvements compared to the uncompressed settings were given in [Tis09, HLLW09]. Then, Tiskin [Tis15] obtained an $O(nN\log N)$-time algorithm and subsequent works [HLLW13, Gaw12] reduced the $O(\log N)$ factor. However, when the distance D is small, the edit distance can be computed in $O(N+D^2)$ time [LV88] in the uncompressed setting. The $O(N)$ term in the running time of the Landau–Vishkin algorithm [LV88] is solely needed to construct a data structure efficiently answering the Longest Common Extension (LCE) queries. However, already the results of Mehlhorn, Sundar, and Uhrig [MSU97] yield $\widetilde{O}(1)$-time LCE queries after $\widetilde{O}(n)$-time preprocessing of the grammars representing X and Y. This gives rise to an $\widetilde{O}(n+D^2)$-time algorithm computing the edit distance. With a more modern implementation of LCE queries in compressed strings by I [I17], the factor hidden within the $\widetilde{O}(\cdot)$ notation can be reduced to $O(\log N)$.

While the $\widetilde{O}(n+D^2)$-time algorithm is very fast if D is small, its efficiency quickly degrades with increasing D and the $\widetilde{O}(nN)$-time algorithm becomes more suitable already for $D \gg \sqrt{nN}$. With a time complexity of $\widetilde{O}(n\sqrt{ND})$, our algorithm improves upon the previous algorithms whenever $\sqrt[3]{n^2N} \ll \delta_E(X,Y) \ll N$. Nevertheless, the current lower bounds allow for a hypothetical holy-grail algorithm achieving the running time of $\widetilde{O}(\min(nD, n+D^2))$ which we leave as an interesting open question.

We also get improved results for the Hamming distance, which is a more basic measure trivially computable in $O(N)$ time. Here, we present an $O(n\sqrt{N})$-time algorithm which improves upon the $O(n^{1.41}N^{0.593})$ bound of [ABBK17]. Additionally, we note that natural generalizations to multiple strings (including the median Hamming distance) can be computed in $O(nN^{1-1/k})$ time.

2 Preliminaries

For two integers $i \le j$, we write $[i\,..\,j]$ to denote the set $\{i, \ldots, j\}$ and $[i\,..\,j)$ to denote $\{i, \ldots, j-1\}$. The notions $(i\,..\,j]$ and $(i\,..\,j)$ are defined analogously.

A *string* is a sequence of characters from a fixed alphabet Σ. We write Σ^* to denote the set of all strings over Σ, and we define $\Sigma^+ = \Sigma^* \setminus \{\gamma\}$, where γ denotes the empty string. The length of a string X is denoted by $|X|$ and, for a position $i \in [1\,..\,|X|]$ in X, the character of X at position i is denoted by $X[i]$. For an integer $N \ge 0$, the set of length-N strings over Σ is denoted by Σ^N.

For two positions $i \le j$ in X, we write $X[i\,..\,j]$ to denote the fragment of X starting at positions i and ending at position j; this fragment is an occurrence of $X[i] \cdots X[j]$ as a substring of X. The fragments $X[i\,..\,j)$, $X(i\,..\,j]$, and $X(i\,..\,j)$ are defined similarly.

A morphism is a function $f : \Sigma_1^* \to \Sigma_2^*$ such that $f(X) = \bigcirc_{i=1}^{|X|} f(X[i])$, where $\bigcirc$ denotes the concatenation operator. Note that every function mapping Σ_1 to Σ_2^* can be uniquely extended to a morphism.

2.1 Straight-Line Programs

A straight-line program is a tuple $\mathbf{G} = (\mathbf{S}, \Sigma, \mathsf{rhs}, S)$, where $\mathbf{S}$ is a finite sequence of *symbols*, $\Sigma \subseteq \mathbf{S}$ is a set of *terminal symbols*, $\mathsf{rhs} : (\mathbf{S} \setminus \Sigma) \to \mathbf{S}^*$ is the *production* (or *right-hand side*) function, and $S \in \mathbf{S}$ is the start symbol, and the symbols in $\mathbf{S}$ are ordered so that B precedes A if B occurs in $\mathsf{rhs}(A)$. We also write $A \to B_1 \cdots B_k$ instead of $\mathsf{rhs}(A) = B_1 \cdots B_k$.

The set $\mathbf{S} \setminus \Sigma$ of *non-terminals* is denoted by $\mathbf{N}$. The size of $\mathbf{G}$ is $|\mathbf{G}| := |\mathbf{S}| + \sum_{A \in \mathbf{N}} |\mathsf{rhs}(A)|$: the number of symbols plus the total length of productions. The *expansion* function $\exp : \mathbf{S} \to \Sigma^+$ is defined recursively:

$$\exp(A) = \begin{cases} A & \text{if } A \in \Sigma, \\ \bigcirc_{i=1}^{k} \exp(B_i) & \text{if } A \to \bigcirc_{i=1}^{k} B_i. \end{cases}$$

We say that $\mathbf{G}$ is a *grammar-compressed representation* of $\exp(S)$. The exp function naturally extends to a morphism $\exp : \mathbf{S}^* \to \Sigma^*$ with $\exp(\bigcirc_{i=1}^{m} A_i) = \bigcirc_{i=1}^{m} \exp(A_i)$.

For a symbol $A \in \mathbf{S}$, we denote $|A| = |\exp(A)|$. In this work, we assume a word RAM machine with machine words of $\Omega(\log |S|)$ bits. In this setting, one can compute $|A|$ for all $A \in \mathbf{S}$ in $O(|\mathbf{G}|)$ time. Consequently, we assume that $|A|$ is stored along with A in the straight-line programs given to our algorithms.

A straight-line program $\mathbf{G}$ is in *Chomsky normal form* if $|\mathsf{rhs}(A)| = 2$ for all $A \in \mathbf{N}$. Given an arbitrary straight-line program $\mathbf{G}$, an equivalent straight-line program $\mathbf{G}'$ in Chomsky normal form can be constructed in $O(|\mathbf{G}|)$ time; moreover, $|\mathbf{G}| = O(|\mathbf{G}'|)$. Thus, without loss of generality, we assume that all straight-line programs given to our algorithms are already in the Chomsky normal form.

3 FPTAS for Compressed Edit Distance of Two Strings

The edit distance $\delta_E(X, Y)$ of two strings $X, Y \in \Sigma^*$ is defined as the minimum number of character insertions, deletions, and substitutions needed to transform X into Y.

In this section, we prove the following result.

THEOREM 3.1. *Given a straight-line program $\mathbf{G}_X$ of size n generating a string X of length $N > 0$, a straight-line program $\mathbf{G}_Y$ of size m generating a string Y of length $M > 0$, and a parameter $\epsilon \in (0, 1]$, an integer between $\delta_E(X, Y)$ and $(1+\epsilon)\delta_E(X, Y)$ can be computed in $\widetilde{O}\left((nm(N+M))^{2/3}\epsilon^{-1/3}\right)$ time.*

Let $\$ \notin \Sigma$ and let $\cdot^{\$} : \Sigma^* \to (\Sigma \cup \{\$\})^*$ be a morphism defined with $a^{\$} = a\$$ for $a \in \Sigma$. Then, $\delta_E(X, Y) = \frac{1}{2}\delta_D(X^{\$}, Y^{\$})$ [Tis15]. Moreover, if X is represented by a straight line program $\mathbf{G}$, then $X^{\$}$ can be represented using a straight-line program of size $2|\mathbf{G}| + 1$. This reduction allows computing δ_D instead of δ_E.

DEFINITION 3.1. (ALIGNMENT GRAPH) *For two strings X and Y, the alignment graph $\mathbf{G}_{X,Y}$ is a weighted undirected graph with vertex set $\{v_{x,y} : x \in [0\,..\,|X|], y \in [0\,..\,|Y|]\}$ and edges:*

- *$v_{x,y-1} \leftrightarrow v_{x,y}$ of length 1, for $x \in [0\,..\,|X|]$ and $y \in [1\,..\,|Y|]$;*
- *$v_{x-1,y} \leftrightarrow v_{x,y}$ of length 1, for $x \in [1\,..\,|X|]$ and $y \in [0\,..\,|Y|]$;*
- *$v_{x-1,y-1} \leftrightarrow v_{x,y}$ of length 0, for $x \in [1\,..\,|X|]$ and $y \in [1\,..\,|Y|]$ such that $X[x] = Y[y]$.*

OBSERVATION 3.1. *Let d be the metric induced by $G_{X,Y}$. All $x, x' \in [0\,..\,|X|]$ and $y, y' \in [0\,..\,|Y|]$, satisfy*

$$d(v_{x,y}, v_{x',y'}) = \begin{cases} \delta_D(X(x\,..\,x'], Y(y\,..\,y']) & \text{if } x \le x' \text{ and } y \le y', \\ \delta_D(X(x'\,..\,x], Y(y'\,..\,y]) & \text{if } x' \le x \text{ and } y' \le y, \\ |x - x'| + |y - y'| & \text{otherwise.} \end{cases}$$

For two ranges $[x\,..\,x'] \subseteq [0\,..\,|X|]$ and $[y\,..\,y'] \subseteq [0\,..\,|Y|]$, the subgraph of $G_{X,Y}$ induced by $\{v_{\bar{x},\bar{y}} : \bar{x} \in [x\,..\,x'], \bar{y} \in [y\,..\,y']\}$ is denoted $G_{X,Y}^{[x\,..\,x'],[y\,..\,y']}$ and called a *block* in $G_{X,Y}$. For a block B, we distinguish the *input vertices* $\mathrm{in}^B = (v_{x',y}, v_{x'-1,y}, \ldots, v_{x,y}, v_{x,y+1}, \ldots, v_{x,y'})$ and the *output vertices* $\mathrm{out}^B = (v_{x',y}, v_{x',y+1}, \ldots, v_{x',y'}, v_{x'-1,y'}, \ldots, v_{x,y'})$; both sequences consist of $|B| := (x' - x) + (y' - y) + 1$ vertices. The $DIST_B$ table is a $|B| \times |B|$ matrix with entries $DIST_B[i,j] = d(\mathrm{in}_i^B, \mathrm{out}_j^B)$ for $i, j \in [1\,..\,|B|]$. The $DIST_B$ table satisfies the Monge property [Tis15]: $DIST_B[i,j] + DIST_B[i',j'] \le DIST_B[i,j'] + DIST_B[i,j']$ holds for all $i, i', j, j' \in [1\,..\,|B|]$ such that $i \le i'$ and $j \le j'$. For two strings $X, Y \in \Sigma^*$, we also define $DIST_{X,Y}$ to be $DIST_B$ for $B = G_{X,Y}^{[0\,..\,|X|],[0\,..\,|Y|]}$. By Observation 3.1, if $B = G_{X,Y}^{[x\,..\,x'],[y\,..\,y']}$, then $DIST_B = DIST_{X(x\,..\,x'],Y(y\,..\,y']}$.

Box decomposition For two strings $X, Y \in \Sigma^*$, the *box decomposition* $\mathbf{B}$ of the graph $G_{X,Y}$ is defined based on decompositions $X = X_1 \circ \cdots \circ X_{p_X}$ and $Y = Y_1 \circ \cdots \circ Y_{p_Y}$ into non-empty fragments, called *phrases*.

Let us define sets $\{b_0^X, \ldots, b_{p_X}^X\}$ and $\{b_0^Y, \ldots, b_{p_Y}^Y\}$ of *phrase boundaries* in X and Y, respectively, so that the phrases are $X_i = X(b_{i-1}^X\,..\,b_i^X]$ for $i \in [1\,..\,p_X]$ and $Y_j = Y(b_{j-1}^Y\,..\,b_j^Y]$ for $j \in [1\,..\,p_Y]$. A vertex $v_{x,y}$ is a *boundary vertex* if x is a phrase boundary in X or y is a phrase boundary in Y, and a *grid vertex* if both x is a phrase boundary in X and y is a phrase boundary in Y. The box decomposition $\mathbf{B}$ is an indexed family $(B_{i,j})_{i \in [1\,..\,p_X], j \in [1\,..\,p_Y]}$ of *boxes* $B_{i,j} := G_{X,Y}^{[b_{i-1}^X\,..\,b_i^X],[b_{j-1}^Y\,..\,b_j^Y]}$.

3.1 Portal-Respecting Walks

Hermelin et al. [HLLW13] applied a box decomposition obtained via an analogue of Corollary 3.1 to determine $\delta_D(X,Y)$ using a dynamic-programming procedure computing $\delta_D(X[1\,..\,x], Y[1\,..\,y])$ for all boundary vertices $v_{x,y}$. We reduce the number of DP states by considering only a selection $\mathbf{P}$ of boundary vertices, called *portals*. This allows improving the running time from $\widetilde{O}(\frac{NM}{\tau})$ to $\widetilde{O}(|\mathbf{P}|)$, but reduces the search space from the family of all walks $v_{0,0} \rightsquigarrow v_{x,y}$ to walks that cross box boundaries only at portals. Below, we formally define such portal-respecting walks and provide a construction suitable for approximating $\delta_D(X,Y)$.

DEFINITION 3.2. *Let $\mathbf{B}$ be a box decomposition of $G_{X,Y}$ and let $\mathbf{P}$ be a set* portals *(selected boundary vertices). We say that a walk W is a* portal-respecting (i,j)-walk *if W is a concatenation of walks W' and W'' such that:*

- *W'' starts at an input portal of $B_{i,j}$ and is entirely contained within $B_{i,j}$, and*
- *W' is the empty walk at $v_{0,0}$, a portal-respecting $(i-1,j)$-walk, or a portal-respecting $(i,j-1)$-walk.*

Let us fix a box decomposition $\mathbf{B}$ of $G_{X,Y}$, and a set of portals $\mathbf{P}$. For a box $B_{i,j} \in \mathbf{B}$, let $\mathbf{P}_{i,j} = \mathbf{P} \cap \mathrm{out}^{B_{i,j}}$ denote the output portals of $B_{i,j}$. Moreover, for a vertex $v_{x,y} \in B_{i,j}$, we denote $d_{x,y} = d(v_{0,0}, v_{x,y}) = \delta_D(X[1\,..\,x], Y[1\,..\,y])$ and let $D_{x,y}^{i,j}$ be the minimum length of a portal-preserving (i,j)-walk ending at $v_{x,y}$.

LEMMA 3.1. *Given a set of portals $\mathbf{P}$ for a box decomposition $\mathbf{B}$ of $G_{X,Y}$, the the length of the shortest portal-respecting (p_X, p_Y)-walk ending at $v_{|X|,|Y|}$ can be computed in $\widetilde{O}(|\mathbf{P}|)$ time provided $\widetilde{O}(1)$-time random access to the DIST matrices of all the boxes of $\mathbf{B}$.*

Proof. For each box $B_{i,j} \in \mathbf{B}$, our algorithm computes $D_{x,y}^{i,j}$ for all vertices $v_{x,y} \in \mathbf{P}_{i,j}$. For this, the boxes $B_{i,j}$ containing any output portal are processed in the order of non-decreasing values $i + j$.

If $(i,j) = (1,1)$, then Definition 3.2 and Observation 3.1 yield $D_{x,y}^{1,1} = d(v_{0,0}, v_{x,y})$, and this value can be retrieved from the $DIST_{B_{1,1}}$ matrix in $\widetilde{O}(1)$ time. Thus, we henceforth assume $(i,j) \neq (1,1)$.

Consider a portal-respecting (i,j)-walk W to a vertex $v_{x,y} \in \mathrm{out}^{B_{i,j}}$. By Definition 3.2, W is a concatenation of two walks W' and W'' such that W'' starts at a vertex $v_{x',y'} \in \mathbf{P} \cap \mathrm{in}^{B_{i,j}}$ and is entirely contained within $B_{i,j}$, whereas W' is a portal-respecting $(i,j-1)$-walk to $v_{x',y'}$ or a portal respecting $(i-1,j)$-walk to $v_{x',y'}$. Observe that, for a fixed portal $v_{x',y'} \in \mathbf{P} \cap \mathrm{in}^{B_{i,j}}$, the lengths of W' and W'' can be optimized independently. Consequently, by Observation 3.1,

$$D_{x,y}^{i,j} = \max\left(\max_{v_{x',y'} \in \mathbf{P}_{i-1,j} \cap \mathrm{in}^{B_{i,j}}} \left\{ D_{x',y'}^{i-1,j} + d(v_{x',y'}, v_{x,y}) \right\}, \max_{v_{x',y'} \in \mathbf{P}_{i,j-1} \cap \mathrm{in}^{B_{i,j}}} \left\{ D_{x',y'}^{i,j-1} + d(v_{x',y'}, v_{x,y}) \right\} \right).$$

A matrix (indexed by $v_{x,y} \in \mathbf{P}_{i,j}$ and all vertices $\mathbf{P}_{i-1,j} \cap \mathrm{in}^{B_{i,j}}$) containing the values $D^{i-1,j}_{x',y'} + d(v_{x',y'}, v_{x,y})$ can be obtained from a submatrix of the $DIST_{B_{i,j}}$ matrix by adding $D^{i-1,j}_{x',y'}$ to all entries in the column of $v_{x',y'}$. These modifications preserve the Monge property, so the resulting matrix is a Monge matrix with $\widetilde{O}(1)$-time random access. Consequently, the SMAWK algorithm [AKM+87] allows computing row-minima, i.e., the values $\max_{v_{x',y'} \in \mathbf{P}_{i-1,j} \cap \mathrm{in}^{B_{i,j}}} \left\{D^{i-1,j}_{x',y'} + d(v_{x',y'}, v_{x,y})\right\}$. A symmetric procedure allows computing the values $\max_{v_{x',y'} \in \mathbf{P}_{i,j-1} \cap \mathrm{in}^{B_{i,j}}} \left\{D^{i,j-1}_{x',y'} + d(v_{x',y'}, v_{x,y})\right\}$, which lets us derive the costs $D^{i,j}_{x,y}$ for all the vertices $v_{x,y} \in \mathbf{P}_{i,j}$. The SMAWK algorithm takes nearly linear time with respect to the sum of matrix dimensions, so the overall time complexity is $\widetilde{O}(|\mathbf{P} \cap B_{i,j}|)$.

Each vertex belongs to at most four boxes, so the overall running time is $\widetilde{O}(|\mathbf{P}|)$. ∎

LEMMA 3.2. *Let $\mathbf{B}$ be a box decomposition of the graph $G_{X,Y}$ for $X, Y \in \Sigma^+$ and let $\alpha > 0$ be a real number. Suppose that $\mathbf{P}$ consists of all the grid vertices and all the boundary vertices $v_{x,y}$ of $\mathbf{B}$ satisfying $|x-y| = \lfloor (1+\alpha)^r \rfloor$ for some integer r. Then, every vertex $v_{x,y} \in B_{i,j}$ satisfies $D^{i,j}_{x,y} \le (1+2\alpha)^{i+j} d_{x,y}$.*

Proof. We proceed by induction on $i+j$. The base case is trivially satisfied due to $D^{1,1}_{x,y} = d_{x,y}$ for $v_{x,y} \in B_{1,1}$. We henceforth fix $v_{x,y} \in B_{i,j}$ with $(i,j) \neq (1,1)$. By Observation 3.1, there is a shortest path from $v_{0,0}$ to $v_{x,y}$ contained within $G_{X[1..x],Y[1..y]}$. Let $v_{x',y'}$ be the first vertex of $B_{i,j}$ on this path. Observe that $v_{x',y'} \in \mathrm{in}^{B_{i,j}}$ and $d_{x,y} = d_{x',y'} + d(v_{x',y'}, v_{x,y})$. By symmetry, we may assume without loss of generality that $v_{x',y'} \in \mathrm{out}^{B_{i-1,j}}$.

Let us choose $v_{x',y''} \in \mathbf{P}_{i-1,j} \cap \mathrm{out}^{B_{i-1,j}}$ as close as possible to $v_{x',y'}$. Since grid vertices are portals, such $v_{x',y''}$ exists. Moreover, by the choice of the remaining portals, $d(v_{x',y'}, v_{x',y''}) \le \alpha |x' - y'| \le \alpha d_{x',y'}$. Let us construct a portal-respecting (i,j)-walk to $v_{x,y}$ by concatenating a shortest portal-respecting $(i-1,j)$-walk to $v_{x',y''}$ and a shortest path from $v_{x',y''}$ to $v_{x,y}$ (by Observation 3.1, we may assume that this path is contained in $B_{i,j}$). This proves $D^{i,j}_{x,y} \le D^{i-1,j}_{x,y} + d(v_{x',y''}, v_{x,y})$. The inductive assumption further yields $D^{i-1,j}_{x,y} \le (1+2\alpha)^{i+j-3} d_{x',y''}$, and thus $D^{i,j}_{x,y} \le (1+2\alpha)^{i+j-3} d_{x',y''} + d(v_{x',y''}, v_{x,y}) \le (1+2\alpha)^{i+j-3}(d_{x',y'} + d(v_{x',y'}, v_{x',y''})) + d(v_{x',y'}, v_{x',y''}) + d_{x,y} - d_{x',y'} \le (1+2\alpha)^{i+j-3}(d_{x',y'} + \alpha d_{x',y'}) + \alpha d_{x',y'} + d_{x,y} - d_{x',y'} \le (1+2\alpha)^{i+j-2} d_{x,y}$. ∎

3.2 A Grammar-Based Box Decomposition Hermelin et al. [HLLW13] presented an algorithm that, given two grammar-compressed strings $X, Y \in \Sigma^+$ and an integer parameter τ, constructs a box decomposition $\mathbf{B}$ of $G_{X,Y}$ with $p_X = O(\lceil \frac{1}{\tau}|X| \rceil)$ and $p_Y = O(\lceil \frac{1}{\tau}|Y| \rceil)$, along with an oracle providing random access to the $DIST_{B_{i,j}}$ matrices of all the boxes $B_{i,j}$. However, their construction costs $\Omega(|X| + |Y|)$ time, which is prohibitive in most of our applications. In this section, we achieve the same goal avoiding the linear dependency on the lengths of X and Y. The bottleneck of [HLLW13] is constructing appropriate decompositions of X and Y into phrases. In the following lemma, we implement an analogous step more efficiently by building a grammar-compressed representation of the sequence of phrases, with each phrase represented by a symbol in an auxiliary grammar.

LEMMA 3.3. *Given a straight-line program $\mathbf{G}$ generating a string X and an integer $\tau \ge 1$, in $O(|\mathbf{G}|)$ time one can construct straight-line programs $\mathbf{G}^+$ and $\mathbf{G}^P$ of size $O(|\mathbf{G}|)$ such that:*

- *the terminal symbols of $\mathbf{G}^P$ are the symbols A of $\mathbf{G}^+$ satisfying $|A| \le \tau$,*
- *$\mathbf{G}^P$ generates a string P such that $\exp_{\mathbf{G}^+}(P) = X$ and $|P| \le \lceil \frac{3}{\tau}|X| \rceil$.*

Proof. If $\tau \ge |X|$, then we simply set $\mathbf{G}^+ = \mathbf{G}$ and set $\mathbf{G}^P$ to be a grammar with no non-terminals whose starting symbol is the starting symbol of $\mathbf{G}$; this construction clearly satisfies the required conditions.

We henceforth assume that $\tau < |X|$. The grammar $\mathbf{G}^+$ is constructed by adding new non-terminals to $\mathbf{G}$. As for $\mathbf{G}^P$, we include as terminals all symbols A of $\mathbf{G}^+$ with $|A| \le \tau$, and we add further symbols as non-terminals. For every symbol A of $\mathbf{G}$ with $|A| > \tau$, we introduce three new non-terminals:

- $L(A)$ and $R(A)$ to $\mathbf{G}^+$, satisfying $|L(A)| \le \tau$ and $|R(A)| \le \tau$,
- $M(A)$ to $\mathbf{G}^P$.

The productions for $L(A)$, $R(A)$, and $M(A)$ are determined based on the production $A \to B_L B_R$:

1. If $|B_L| \le \tau$ and $|B_R| \le \tau$, then $L(A) \to B_L$, $R(A) \to B_R$, and $M(A) \to \gamma$.
2. If $|B_L| > \tau$ and $|B_R| > \tau$, then $L(A) \to L(B_L)$, $R(A) \to R(B_R)$, and $M(A) \to M(B_L)R(B_L)L(B_R)M(B_R)$.
3. If $|B_L| > \tau$ and $|B_R| \le \tau$, then $L(A) \to L(B_L)$ and:
   (a) $R(A) \to R(B_L)B_R$ and $M(A) \to M(B_L)$ if $|R(B_L)| + |B_R| \le \tau$,

(b) $R(A) \to B_R$ and $M(A) \to M(B_L)R(B_L)$ otherwise.

4. If $|B_L| \le \tau$ and $|B_R| > \tau$, then $R(A) \to R(B_R)$ and:
   (a) $L(A) \to B_L L(B_R)$ and $M(A) \to M(B_R)$ if $|B_L| + |L(B_R)| \le \tau$,
   (b) $L(A) \to B_L$ and $M(A) \to L(B_R)M(B_R)$ otherwise.

Additionally, for the starting symbol S of $\mathbf{G}$, we add a starting symbol $S^P \to L(S)M(S)R(S)$ to $\mathbf{G}^P$.

A simple inductive argument shows that every symbol A of $\mathbf{G}$ with $|A| > \tau$ satisfies

$$\exp_{\mathbf{G}}(A) = \exp_{\mathbf{G}^+}(L(A) \circ \exp_{\mathbf{G}^P}(M(A)) \circ R(A)).$$

In particular, $P = \exp_{\mathbf{G}^P}(S^P)$ satisfies $\exp_{\mathbf{G}^+}(P) = \exp_{\mathbf{G}}(S) = X$.

It remains to prove that $|P| < \frac{3}{\tau}|X|$. For this, we inductively show that every symbol A of $\mathbf{G}$ with $|A| > \tau$ satisfies $|L(A)| + |R(A)| + \tau(|M(A)| + 2) < 3|A|$. To prove this claim, we analyze the cases based on the production $A \to B_L B_R$.

1. If $|B_L| \le \tau$ and $|B_R| \le \tau$, then
$$|L(A)| + |R(A)| + \tau(|M(A)| + 2) = |A| + 2\tau < 3|A|.$$
2. If $|B_L| > \tau$ and $|B_R| > \tau$, then
$$|L(A)| + |R(A)| + \tau(|M(A)| + 2) = |L(B_L)| + |R(B_R)| + \tau(|M(B_L)| + 2 + |M(B_R)| + 2) <$$
$$|L(B_L)| + |R(B_L)| + \tau(|M(B_L)| + 2) + |L(B_R)| + |R(B_R)| + \tau(|M(B_R)| + 2) < 3|B_L| + 3|B_R| = 3|A|.$$
3. If $|B_L| > \tau$, $|B_R| \le \tau$, then
   - If $|R(B_L)| + |B_R| \le \tau$, then
   $$|L(A)| + |R(A)| + \tau(|M(A)| + 2) = |L(B_L)| + |R(B_L)| + |B_R| + \tau(|M(B_L)| + 2) <$$
   $$3|B_L| + |B_R| < 3|A|.$$
   - Otherwise,
   $$|L(A)| + |R(A)| + \tau(|M(A)| + 2) = |L(B_L)| + |B_R| + \tau(|M(B_L)| + 3) <$$
   $$|L(B_L)| + |B_R| + \tau(|M(B_L)| + 2) + |R(B_L)| + |B_R| < 3|B_L| + 2|B_R| < 3|A|.$$
4. The case involving $|B_L| \le \tau$ and $|B_R| > \tau$ is symmetric to the previous one.

In particular, this claim holds for $A = S$, so $|S^P| = |M(S)| + 2 < \frac{1}{\tau}(3|S| - |L(S)| - |R(S)|) < \frac{3}{\tau}|S| = \frac{3}{\tau}|X|$. ∎

As for constructing the $DIST$ matrices, we use the original implementation from [HLLW13].

LEMMA 3.4. ([HLLW13, SECTION 5]) *Given straight-line programs $\mathbf{G}_X$ and $\mathbf{G}_Y$ and an integer $\tau \ge 1$, in $\widetilde{O}(|\mathbf{G}_X||\mathbf{G}_Y|\tau)$ time one can construct a data structure that provides $\widetilde{O}(1)$-time random access to the $DIST_{\exp(A_X),\exp(A_Y)}$ matrices for all symbols A_X of $\mathbf{G}_X$ and A_Y of $\mathbf{G}_Y$ satisfying $|A_X| \le \tau$ and $|A_Y| \le \tau$.*

Combining Lemmas 3.3 and 3.4, we complete our construction.

COROLLARY 3.1. *Given a straight-line program $\mathbf{G}_X$ of size n generating a string X of length $N > 0$, a straight-line program $\mathbf{G}_Y$ of size m generating a string Y of length $M > 0$, and an integer $\tau \in [1\,..\,N + M]$, one can in $\widetilde{O}(\frac{N+M}{\tau} + nm\tau)$ time construct a box decomposition $\mathbf{B} = (B_{i,j})_{i\in[1\,..\,p_X],j\in[1\,..\,p_Y]}$ of $G_{X,Y}$ with $p_X = O(\lceil\frac{N}{\tau}\rceil)$ and $p_Y = O(\lceil\frac{M}{\tau}\rceil)$, along with an oracle providing $\widetilde{O}(1)$-time random access to the $DIST_{B_{i,j}}$ matrices.*

Proof. First, we use Lemma 3.3 to obtain grammars $\mathbf{G}_X^+$ and $\mathbf{G}_X^P$. The string P_X represented by $\mathbf{G}_X^P$ satisfies $X = \exp_{\mathbf{G}_X^+}(P_X)$, so it can be interpreted as a decomposition of X into $p_X := |P_X|$ phrases, with the ith phrase X_i being an occurrence of $\exp_{\mathbf{G}_X^+}(P_X[i])$. The decomposition of Y is obtained in the same way based on grammars $\mathbf{G}_Y^+$ and $\mathbf{G}_Y^P$ constructed for Y.

The box decomposition $\mathbf{B}$ is based on these decompositions of X and Y. Note that each box $B_{i,j}$ satisfies

$$DIST_{B_{i,j}} = DIST_{\exp_{\mathbf{G}_X^+}(P_X[i]),\exp_{\mathbf{G}_Y^+}(P_Y[j])}.$$

Due to $|P_X[i]| \le \tau$ and $|P_Y[j]| \le \tau$, Lemma 3.4 applied to $\mathbf{G}_X^+$ and $\mathbf{G}_Y^+$ provides $\widetilde{O}(1)$-time oracle access to all these matrices. Storing P_X and P_Y, we can point to $DIST_{B_{i,j}}$ in $O(1)$ time given i, j. ∎

3.3 Algorithm

PROPOSITION 3.1. *Given a straight-line program $\mathbf{G}_X$ of size n generating a string X of length $N > 0$, a straight-line program $\mathbf{G}_Y$ of size m generating a string Y of length $M > 0$, and a parameter $\epsilon \in (0, 1]$, a $(1+\epsilon)$-approximation of $\delta_D(X, Y)$ can be computed in $\widetilde{O}\left((nm(N+M))^{2/3}\epsilon^{-1/3}\right)$ time.*

Proof. The algorithm uses Corollary 3.1 and Lemma 3.1 with the set of portals $\mathbf{P}$ defined as in Lemma 3.2, where $\alpha = \Omega(\frac{\epsilon}{p_X+p_Y-2}) = \Omega(\frac{\epsilon\tau}{N+M})$ is chosen so that $(1+2\alpha)^{p_X+p_Y-2} = 1+\epsilon$. Lemma 3.2 guarantees that the resulting value is a $(1+\epsilon)$-approximation of $\delta_D(X, Y)$. The number of portals is $O\left(\frac{NM}{\tau^2} + \frac{N}{\tau}\log_{1+\alpha} M + \frac{M}{\tau}\log_{1+\alpha} N\right) = \widetilde{O}\left(\frac{(N+M)^2}{\epsilon\tau^2}\right)$, so the overall running time is $\widetilde{O}\left(nm\tau + \frac{(N+M)^2}{\epsilon\tau^2}\right)$. Optimizing $\tau \in [1\,..\,N+M]$, we get $\widetilde{O}(nm + \epsilon^{-1} + (nm(N+M))^{2/3}\epsilon^{-1/3})$ time. If the first term dominates, then $nm \geq (nm(N+M))^{2/3}\epsilon^{-1/3} \geq (N+M)^2\epsilon^{-1}$. However, $O(NM) = O((N+M)^2\epsilon^{-1})$ time is enough to compute $\delta_D(X, Y)$ exactly without compression. If the second term dominates, then $\epsilon^{-1} \geq (nm(N+M))^{2/3}\epsilon^{-1/3} \geq nm(N+M)$. However, $\widetilde{O}(\sqrt{nm}(N+M)) = \widetilde{O}(nm(N+M))$ time is enough to compute $\delta_D(X, Y)$ exactly using Proposition 3.2 with $D = N+M$. ∎

Theorem 3.1 follows through the reduction from δ_E to δ_D.

3.4 Exact Output-Sensitive Algorithm

In this section we prove Theorem 3.2:

THEOREM 3.2. *Given a straight-line program $\mathbf{G}_X$ of size n generating a string X of length $N > 0$ and a straight-line program $\mathbf{G}_Y$ of size m generating a string Y of length $M > 0$, the edit distance $\delta_E(X, Y)$ can be computed in $\widetilde{O}\left(\sqrt{(1+\delta_E(X,Y))nm(N+M)}\right)$ time.*

The algorithm behind Theorem 3.2 reduces the problem to a decision version (asking whether $\delta_E(X, Y) \leq D$ for a threshold D) and then uses the same scheme with all boundary vertices (x, y) satisfying $|x - y| \leq D$ selected as portals.

LEMMA 3.5. *Let $\mathbf{B}$ be a box decomposition of the graph $G_{X,Y}$ for $X, Y \in \Sigma^+$ and let $D \geq 0$ be an integer. Suppose that $\mathbf{P}$ consists of all the boundary vertices $v_{x,y}$ of $\mathbf{B}$ satisfying $|x - y| \leq D$. Then, every vertex $v_{x,y} \in B_{i,j}$ with $d_{x,y} \leq D$ satisfies $D^{i,j}_{x,y} = d_{x,y}$.*

Proof. We proceed by induction on $i + j$. The base case is trivially satisfied due to $D^{1,1}_{x,y} = d_{x,y}$ for $v_{x,y} \in B_{1,1}$. We henceforth fix $v_{x,y} \in B_{i,j}$ with $(i, j) \neq (1, 1)$ and $d_{x,y} \leq D$. By Observation 3.1, there is a shortest path from $v_{0,0}$ to $v_{x,y}$ contained within $G_{X[1\,..\,x],Y[1\,..\,y]}$. Let $v_{x',y'}$ be the first vertex on this path that belongs to $B_{i,j}$. Observe that $v_{x',y'} \in \text{in}^{B_{i,j}}$ and $d_{x,y} = d_{x',y'} + d(v_{x',y'}, v_{x,y})$. Consequently, $|x' - y'| \leq d_{x',y'} \leq d_{x,y} \leq D$, so $v_{x',y'} \in \mathbf{P}_{i-1,j} \cup \mathbf{P}_{i,j-1}$. By symmetry, we may assume without loss of generality that $v_{x',y'} \in \mathbf{P}_{i-1,j}$.

Let us construct a portal-respecting (i, j)-walk to $v_{x,y}$ by concatenating a shortest portal-respecting $(i-1, j)$-walk to $v_{x',y'}$ and a shortest path from $v_{x',y'}$ to $v_{x,y}$ (by Observation 3.1 applied to $G_{X(x'\,..\,x],Y(y'\,..\,y]}$, we may assume that this path is contained in $B_{i,j}$). This proves $D^{i,j}_{x,y} \leq D^{i-1,j}_{x',y'} + d(v_{x',y'}, v_{x,y}) = D^{i,j}_{x,y} \leq D^{i-1,j}_{x',y'} + d_{x,y} - d_{x',y'}$. The inductive assumption yields $D^{i-1,j}_{x',y'} = d_{x',y'}$, and thus $D^{i,j}_{x,y} \leq d_{x,y}$ holds as claimed. ∎

PROPOSITION 3.2. *Given a straight-line program $\mathbf{G}_X$ of size n generating a string X of length $N > 0$, a straight-line program $\mathbf{G}_Y$ of size m generating a string Y of length $M > 0$, and an integer $D \in [1\,..\,N+M]$, one can in $\widetilde{O}\left(\sqrt{nmD(N+M)}\right)$ time compute $\delta_D(X, Y)$ or certify that $\delta_D(X, Y) > D$.*

Proof. The algorithm uses Corollary 3.1 and Lemma 3.1 with the set of portals $\mathbf{P}$ defined as in Lemma 3.5. The latter lemma guarantees that the resulting value is $\delta_D(X, Y)$ provided that $\delta_D(X, Y) \leq D$. Otherwise, the resulting value exceeds D, certifying that $\delta_D(X, Y) > D$.

The number of portals is $O(D \cdot \frac{N+M}{\tau})$, so the overall running time is $\widetilde{O}(nm\tau + D \cdot \frac{N+M}{\tau})$. Optimizing $\tau \in [1\,..\,N+M]$, we get $\widetilde{O}(nm + D + \sqrt{nmD(N+M)})$ time. Since $D \leq N+M$, the second term is dominated by the third one. If the first term dominates, then $nm > D(N+M)$, and thus $\sqrt{nmD(N+M)} \geq D(N+M)$. However, $\widetilde{O}(N+M+D^2) = \widetilde{O}(D(N+M))$ time suffices solve the problem for uncompressed strings [LV88]. ∎

Theorem 3.2 follows through exponential search and the reduction from δ_E to δ_D.

3.5 LCS Approximation In this section we prove Theorem 3.3:

THEOREM 3.3. *Given a straight-line program $\mathbf{G}_X$ of size n generating a string X of length $N > 0$, a straight-line program $\mathbf{G}_Y$ of size m generating a string Y of length $M > 0$, and a parameter $\epsilon \in (0, 1]$, a $(1+\epsilon)$-approximation of $\mathrm{LCS}(X, Y)$ can be computed in $\widetilde{O}\left((nm(N+M))^{2/3}\epsilon^{-1/3}\right)$ time.*

The algorithm behind Theorem 3.3 is essentially the same as that of Theorem 3.1, and this is why the running times coincide. The main difference is that the output portals of the box $B_{i,j}$ are chosen *adaptively* while the dynamic-programming algorithm processes $B_{i,j}$.

As for LCS approximation, our choice of portals is *adaptive*. For a box $B_{i,j} \in \mathbf{B}$, let $\mathbf{P}_{i,j} = \mathbf{P} \cap \mathrm{out}^{B_{i,j}}$. Observe that the value $D^{i,j}_{x,y}$ for $v_{x,y} \in B_{i,j}$ depends only on $\mathbf{P}_{i',j'}$ with $i' + j' < i + j$. Hence, except for the grid vertices (all included in $\mathbf{P}$), we may select the portals $\mathbf{P}_{i,j}$ based on the values $D^{i,j}_{x,y}$ for $v_{x,y} \in \mathrm{out}^{B_{i,j}}$.

For $v_{x,y} \in B_{i.j}$, let $\ell_{x,y} = \frac{1}{2}(|X| + |Y| - d_{x,y}) = \mathrm{LCS}(X[1\,..\,x], Y[1\,..\,y])$ and $L^{i,j}_{x,y} = \frac{1}{2}(|X| + |Y| - D^{i,j}_{x,y})$.

LEMMA 3.6. *Let $\mathbf{B}$ be a box decomposition of the graph $G_{X,Y}$ for $X, Y \in \Sigma^+$ and let $\alpha > 0$ be a real number. Suppose that $\mathbf{P}$ consists of all the grid vertices and, for each box $B_{i,j} \in \mathbf{B}$, all vertices $v_{x,y} \in \mathrm{out}^{B_{i,j}}$ such that:*

- *$v_{x-1,y} \in \mathrm{out}^{B_{i,j}}$ and $\lfloor \log_{1+\alpha} L^{i,j}_{x,y} \rfloor > \lfloor \log_{1+\alpha} L^{i,j}_{x-1,y} \rfloor$, or*
- *$v_{x,y-1} \in \mathrm{out}^{B_{i,j}}$ and $\lfloor \log_{1+\alpha} L^{i,j}_{x,y} \rfloor > \lfloor \log_{1+\alpha} L^{i,j}_{x,y-1} \rfloor$.*

Then, for each vertex $v_{x,y} \in B_{i,j}$, we have $L^{i,j}_{x,y} \geq (1+\alpha)^{2-i-j}\ell_{x,y}$.

Proof. We proceed by induction on $i + j$. The base case is trivially satisfied due to $L^{1,1}_{x,y} = \ell_{x,y}$ for $v_{x,y} \in B_{1,1}$. Thus, we henceforth fix a vertex $v_{x,y} \in B_{i,j}$ with $(i, j) \neq (1, 1)$. By Observation 3.1, there is a shortest path from $v_{0,0}$ to $v_{x,y}$ contained within $G_{X[1\,..\,x],Y[1\,..\,y]}$. Let $v_{x',y'}$ be the first vertex on this path that belong to $B_{i,j}$. Observe that $v_{x',y'} \in \mathrm{in}^{B_{i,j}}$ and $\ell_{x,y} = \ell_{x',y'} + \mathrm{LCS}(X(x'\,..\,x], Y(y'\,..\,y])$. By symmetry, we may assume without loss of generality that $v_{x',y'} \in \mathrm{out}^{B_{i-1,j}}$.

Consider the largest value $y'' \in [1\,..\,y']$ such that $v_{x',y''} \in \mathbf{P}_{i-1,j}$. Since grid vertices are portals, such $v_{x',y''}$ exits. Moreover, by the choice of the remaining portals, $L^{i-1,j}_{x',y'} \leq (1+\alpha)L^{i-1,j}_{x',y''}$. Let us construct a portal-respecting (i, j) walk to $v_{x,y}$ by concatenating a shortest portal-respecting $(i-1, j)$-walk to $v_{x',y''}$ and a shortest path from $v_{x',y''}$ to $v_{x,y}$ (by Observation 3.1, we may assume that this path is contained $B_{i,j}$). This proves that $D^{i,j}_{x,y} \leq D^{i-1,j}_{x',y''} + d(v_{x',y''}, v_{x,y}) \leq D^{i-1,j}_{x',y''} + y' - y'' + d(v_{x',y'}, v_{x,y})$, i.e., $L^{i,j}_{x,y} \geq L^{i-1,j}_{x',y''} + \mathrm{LCS}(X(x'\,..\,x], Y(y'\,..\,y]) \geq (1+\alpha)^{-1}L^{i-1,j}_{x',y'} + \ell_{x,y} - \ell_{x',y'}$. The inductive assumption further yields $L^{i-1,j}_{x',y'} \geq (1+\alpha)^{3-i-j}\ell_{x',y'}$, and thus $L^{i,j}_{x,y} \geq (1+\alpha)^{2-i-j}\ell_{x',y'} + \ell_{x,y} - \ell_{x',y'} \geq (1+\alpha)^{2-i-j}\ell_{x,y}$ holds as claimed. ∎

LEMMA 3.7. *Given $\widetilde{O}(1)$-time random access to the $DIST_{B_{i,j}}$ matrix, the values $D^{i-1,j}_{x',y'}$ for all vertices $v_{x',y'} \in \mathbf{P}_{i-1,j}$ (if $i > 1$), and the values $D^{i,j-1}_{x',y'}$ for all vertices $v_{x',y'} \in \mathbf{P}_{i,j-1}$ (if $j > 1$), the values $D^{i,j}_{x,y}$ for any q query vertices $v_{x,y} \in \mathrm{out}^{B_{i,j}}$ can be computed in $\widetilde{O}(q + |\mathbf{P} \cap \mathrm{in}^{B_{i,j}}|)$ time.*

Proof. If $(i, j) = (1, 1)$, then Definition 3.2 and Observation 3.1 yield $D^{1,1}_{x,y} = d(v_{0,0}, v_{x,y})$, and this value can be retrieved from the $DIST_{B_{1,1}}$ matrix in $\widetilde{O}(1)$ time. Thus, we henceforth assume $(i, j) \neq (1, 1)$.

Consider a portal-respecting (i, j)-walk W to a vertex $v_{x,y} \in \mathrm{out}^{B_{i,j}}$. By Definition 3.2, W is a concatenation of two walks W' and W'' such that W'' starts at a vertex $v_{x',y'} \in \mathbf{P} \cap \mathrm{in}^{B_{i,j}}$ and is entirely contained within $B_{i,j}$, whereas W' is a portal-respecting $(i, j-1)$-walk to $v_{x',y'}$ or a portal respecting $(i-1, j)$-walk to $v_{x',y'}$. Observe that, for a fixed portal $v_{x',y'} \in \mathbf{P} \cap \mathrm{in}^{B_{i,j}}$, the lengths of W' and W'' can be optimized independently. Consequently, by Observation 3.1,

$$D^{i,j}_{x,y} = \max\left(\max_{v_{x',y'} \in \mathbf{P}_{i-1,j} \cap \mathrm{in}^{B_{i,j}}} \left\{D^{i-1,j}_{x',y'} + d(v_{x',y'}, v_{x,y})\right\}, \max_{v_{x',y'} \in \mathbf{P}_{i,j-1} \cap \mathrm{in}^{B_{i,j}}} \left\{D^{i,j-1}_{x',y'} + d(v_{x',y'}, v_{x,y})\right\}\right).$$

A matrix (indexed by the query vertices $v_{x,y} \in \mathrm{out}^{B_{i,j}}$ and all vertices $\mathbf{P}_{i-1,j} \cap \mathrm{in}^{B_{i,j}}$) containing the values $D^{i-1,j}_{x',y'} + d(v_{x',y'}, v_{x,y})$ can be obtained from a submatrix of the $DIST_{B_{i,j}}$ matrix by adding $D^{i-1,j}_{x',y'}$ to all entries in the column of $v_{x',y'}$. These modifications preserve the Monge property, so the resulting matrix is a Monge matrix with $\widetilde{O}(1)$-time random access. Consequently, the SMAWK algorithm [AKM+87] allows computing row-minima,

i.e., the values $\max_{v_{x',y'} \in \mathbf{P}_{i-1,j} \cap \text{in}^{B_{i,j}}} \{D^{i-1,j}_{x',y'} + d(v_{x',y'}, v_{x,y})\}$. A symmetric procedure allows computing the values $\max_{v_{x',y'} \in \mathbf{P}_{i,j-1} \cap \text{in}^{B_{i,j}}} \{D^{i,j-1}_{x',y'} + d(v_{x',y'}, v_{x,y})\}$, which lets us derive the costs $D^{i,j}_{x,y}$ for all the query vertices $v_{x,y} \in \text{out}^{B_{i,j}}$. The SMAWK algorithm takes nearly linear time with respect to the sum of matrix dimensions, so the overall time complexity is $\widetilde{O}(q + |\mathbf{P} \cap \text{in}^{B_{i,j}}|)$. ∎

LEMMA 3.8. *Given a box decomposition* $\mathbf{B}$ *of* $G_{X,Y}$*, a parameter* $\epsilon \in (0, 1]$*, and* $\widetilde{O}(1)$*-time random access to the DIST matrices of all the boxes of* $\mathbf{B}$*, a* $(1+\epsilon)$*-approximation of* $\text{LCS}(X, Y)$ *can be computed in* $\widetilde{O}(\epsilon^{-1}(p_X + p_Y)^2)$ *time.*

Proof. Let us choose $\alpha = \Omega(\frac{\epsilon}{p_X + p_Y - 2})$ so that $(1+\alpha)^{p_X+p_Y-2} = 1 + \epsilon$. We process boxes $B_{i,j} \in \mathbf{B}$ in the order of non-decreasing values $i + j$, constructing the output portals $P_{i,j}$ according to Lemma 3.6 and computing the values $L^{i,j}_{x,y}$ for all $v_{x,y} \in \mathbf{P}_{i,j}$. By Lemma 3.6, the value $L^{p_X,p_Y}_{|X|,|Y|}$ is guaranteed to be a $(1+\epsilon)$-approximation of $\text{LCS}(X, Y)$.

The ordering of boxes lets us compute the values $L^{i,j}_{x,y}$ for any q vertices $v \in \text{out}^{B_{i,j}}$ in $\widetilde{O}(q + |\mathbf{P} \cap \text{in}^{B_{i,j}}|)$ time. By symmetry, we may focus without loss of generality on the right boundary of $B_{i,j}$, i.e., vertices $v_{x,y}$ with $x = b^X_i$ and $y \in [b^Y_{j-1} \,..\, b^Y_j]$. Note that the corresponding values $L^{i,j}_{x,y}$ are non-decreasing: $D^{i,j}_{x,y} \le D^{i,j}_{x,y-1} + 1$ implies $L^{i,j}_{x,y} \ge L^{i,j}_{x,y-1}$. First, we apply Lemma 3.7 to derive $L^{i,j}_{x,y}$ for the two extreme values $y \in \{b^Y_{j-1}, b^Y_j\}$. Next, for each value $r \in [\lfloor \log_{1+\alpha} L^{i,j}_{x,b^Y_{j-1}} \rfloor \,..\, \lfloor \log_{1+\alpha} L^{i,j}_{x,b^Y_j} \rfloor]$, we binary search for the smallest $y \in [b^Y_{j-1} \,..\, b^Y_j]$ such that $\log_{1+\alpha} L^{i,j}_{x,y} \ge r$, and include $v_{x,y}$ in $\mathbf{P}_{i,j}$. The binary searches are executed in parallel, with Lemma 3.7 applied to determine $L^{i,j}_{x,y}$ for all the current pivots. This way, the algorithm is implemented in $\widetilde{O}(1 + \log_{1+\alpha} L^{i,j}_{x,b^Y_j} - \log_{1+\alpha} L^{i,j}_{y,b^Y_{j-1},y} + |\mathbf{P} \cap \text{in}^{B_{i,j}}|)$ time. Due to $L^{i,j}_{y,b^Y_{j-1}} \ge L^{i,j-1}_{y,b^Y_{j-1}}$, the first term sums up to $\widetilde{O}(\log_{1+\alpha} |Y|) = \widetilde{O}(\epsilon^{-1}(p_X + p_Y))$ across $j \in [1 \,..\, p_Y]$, and to $\widetilde{O}(\epsilon^{-1}(p_X + p_Y)^2)$ across $B_{i,j} \in \mathbf{B}$. This also bounds the number of portals created, so the second term, which sums up to $|\mathbf{P}|$ across all boxes, is also $\widetilde{O}(\epsilon^{-1}(p_X + p_Y)^2)$. ∎

Proof. [Proof of Theorem 3.3] The algorithm uses Corollary 3.1 and Lemma 3.8. Due to $p_X + p_Y = \frac{N+M}{\tau}$, the overall running time is $\widetilde{O}\big(nm\tau + \frac{(N+M)^2}{\epsilon\tau^2}\big)$. Optimizing τ, we get the running time of $\widetilde{O}(nm + \epsilon^{-1} + (nm(N+M))^{2/3}\epsilon^{-1/3})$.

If the first term dominates, then $nm \ge (nm(N+M))^{2/3}\epsilon^{-1/3} \ge (N+M)^2\epsilon^{-1}$. However, $O(NM) = O((N+M)^2\epsilon^{-1})$ time is enough to compute $\text{LCS}(X, Y)$ exactly without compression. If the second term dominates, then $\epsilon^{-1} \ge (nm(N+M))^{2/3}\epsilon^{-1/3} \ge nm(N+M)$. However, $\widetilde{O}(\sqrt{nm}(N+M)) = \widetilde{O}(nm(N+M))$ time is enough to compute $\text{LCS}(X, Y)$ exactly using Proposition 3.2 with $D = N + M$. ∎

4 FPTAS For Compressed Median k-Edit Distance

The median k-edit distance is defined as below.

DEFINITION 4.1. *The (median) edit distance* $\delta_E(X_1, \ldots, X_k)$ *of* k *strings* $X_1, \ldots, X_k$ *is the minimum total number of edits (insertions, deletions, and substitutions) needed to make all strings* X_i *equal some string* X^**. That is,* $\delta_E(X_1, \ldots, X_k) = \min_{X^*} \sum_{i=1}^{k} \delta_E(X_i, X^*)$.

For the (median) edit distance between k strings, we show that allowing $(1+\epsilon)$-approximation gives an algorithm circumventing the bound in Theorem 6.2:

THEOREM 4.1. *Given* $k = O(1)$ *straight-line programs* $\mathbf{G}_{X_i}$ *of total size* n *generating strings* X_i *of total length* $N > 0$ *and a parameter* $\epsilon \in (0, 1]$*, an integer between* $\delta_E(X_1, \ldots, X_k)$ *and* $(1+\epsilon)\delta_E(X_1, \ldots, X_k)$ *can be computed in* $\widetilde{O}\left(\epsilon^{-O(k)} n^{k/2} N^{k/2}\right)$ *time.*

To prove the above theorem, we use a different set of techniques than in the two-string case. Most approaches for speeding up the textbook DP algorithm for two (compressible) strings, including the aforementioned results in this paper, rely on the ability to perform computations involving DIST matrices efficiently. These computations crucially depend on the fact that DIST matrices satisfy the Monge property. However, for the natural high-dimensional generalization of DIST matrices, we do not know of any analog of the Monge property they satisfy that allows us to perform similar computations even for three-string similarity problems. Indeed, most natural

generalizations of the Monge property seem to not hold even in the three-string setting (see Section 9 for more details). Thus, it appears unlikely that, for example, an algorithm that partitions the DP table into boxes and computes the DP values on the boundary of each box using computations involving DIST matrices would be substantially more efficient than the textbook edit-distance algorithm, even in the three-string setting.

This motivates us to instead use the window-respecting alignment scheme that has appeared in approximation algorithms for edit distance (e.g., [CDG+18, GRS20]).

4.1 Window-Respecting Alignments We will assume that $\delta_E(X_1, \dots, X_k)$ lies between D and $2D$ for some known D at the loss of a $\log N$ factor in the runtime. We partition X_1 into $\lceil |X_1|/\tau \rceil$ disjoint windows $W_{1,1}$ to $W_{1,N/\tau}$ each of length τ (without loss of generality; we could always e.g. pad each string with an equal amount of a new dummy character to ensure $|X_1|$ is a multiple of τ, without asymptotically affecting their size or compression size). That is, $W_{1,j} = X_1[(j-1)\tau + 1, j\tau]$.

We define for $X_2, \dots, X_k$ possibly overlapping windows indexed by (i) Δ, a guess for the (signed) difference between the length of W and the corresponding window in X_1 and (ii) the starting position p of the window. More formally, the windows are indexed by $W_{i,\Delta,p}$. Throughout the section, let $\sigma := \max\{\lfloor \epsilon D\tau/|X_1| \rfloor, 1\}$ and $R_d(x)$ denote x rounded down to the nearest multiple of σ. Then $W_{i,\Delta,p} = X_i[p\sigma + 1 \,..\, R_d(p\sigma + \tau + \Delta)]$ (or is the empty string "starting" at position $p\sigma + 1$ if $R_d(\min\{p\sigma + \tau + \Delta, |X_i|\}) < p\sigma + 1$). If $R_d(p\sigma + \tau + \Delta) > |X_1|$, $W_{i,\Delta,p}$ is not included in our set of windows. We will define this window for:

- All Δ in $\{0, 1, -1, \lfloor (1+\epsilon) \rfloor, -\lfloor (1+\epsilon) \rfloor, \lfloor (1+\epsilon)^2 \rfloor, \dots \lfloor (1+\epsilon)^{\lceil \log_{1+\epsilon} 2\tau/\epsilon^2 \rceil} \rfloor\} \cup \{-\tau\}$ for which $\tau + \Delta \geq 0$,
- All p from 0 to $\lfloor |X_i|/\sigma \rfloor$.

It suffices to consider windows of size at most $2\tau/\epsilon^2$ by the following lemma:

LEMMA 4.1. *Given $X_1, X_2, \dots, X_k$ and a parameter τ, for $J = |X_1|/\tau$, let X^* be the string such that $\delta_E(X_1, X_2, \dots X_n) = \sum_i \delta_E(X_i, X^*)$. There exists a partition of each X_1 into substrings $\{X_{1,j}\}_{j\in[J]}$, disjoint substrings of the other X_i, $\{X_{i,j}\}_{j\in[J]}$, and a partition of X^* into substrings $\{X^*_j\}_{j\in[J]}$ such that:*

- *$|X_{1,j}| = \tau$ for all j.*
- *For any j and $j < j'$, $X_{i,j}$ appears before $X_{i,j'}$ in X_i.*
- *$\max_{i,j} |X_{i,j}| \leq 2\tau/\epsilon^2$.*
- $$\sum_{j\in[J]} \delta_E(X_{ij}, X^*_j) + |X_i| - \sum_{j\in[J]} |X_{i,j}| \leq (1+3\epsilon)\delta_E(X_i, X^*),$$

  Which implies:

  $$\sum_{j\in[J]} \delta_E(X_{1,j}, X_{2,j}, \dots, X_{k,j}) + \sum_{i>1}(|X_i| - \sum_{j\in[J]} |X_{i,j}|) \leq (1+3\epsilon)\delta_E(X_1, X_2, \dots X_k).$$

  That is, the cost of the alignment that aligns $X_{1,j}$ with each $X_{i,j}$, and then deletes all characters in X_2 to X_k that are unaligned with some $X_{1,j}$ is at most $(1+3\epsilon)\delta_E(X_1, X_2, \dots X_k)$.

Effectively, Lemma 4.1 says that there is a near-optimal alignment that aligns the windows of X_1 to substrings of the other strings that are not more than $1/\epsilon^2$ times larger.

Proof. We partition X_1 into substrings of length τ, $\{X_{1,j}\}_{j\in[J]}$. X^* can be partitioned into substrings $\{\tilde{X}^*_j\}_{j\in[J]}$ such that $\delta_E(X_1, X^*) = \sum_{j\in[J]} \delta_E(X_{1,j}, \tilde{X}^*_j)$.

First, we will "realign" X_1 and X^* to ensure no $\tilde{X}^*_j$ is much larger than $X_{1,j}$. Call a contiguous subsequence of $[J]$, $[j \,..\, j'] := \{j, j+1, \dots j'\}$, "skewed" if $\sum_{m\in[j\,..\,j']} |X_{1,m}| < \frac{\epsilon}{2} \sum_{m\in[j\,..\,j']} |\tilde{X}^*_m|$. Let us take a "maximal" set S of disjoint skewed contiguous subsequences, i.e. a set S such that (i) all the subsequences in S are disjoint (ii) for every contiguous subsequence s in S, there is no skewed contiguous subsequence s' such that $s \subset s'$ and (iii) there is no skewed contiguous subsequence that is not in S but also is completely disjoint from every element of S.

For each skewed contiguous subsequence $[j \,..\, j']$ in S, note that $j' + 1$ does not appear in any element of S (otherwise, $[j \,..\, j']$ and this element can be combined to form a longer skewed contiguous subsequence, violating (ii)), and $[j \,..\, j' + 1]$ is not skewed (again, $[j \,..\, j' + 1]$ being skewed would violate (ii) since $[j \,..\, j'] \subset [j \,..\, j' + 1]$). Take S and replace each $[j \,..\, j']$ with $[j \,..\, j' + 1]$ to get S'. For each $[j \,..\, j' + 1] \in S'$, we have:

$$\frac{\epsilon}{2}\sum_{m\in[j\,..\,j'+1]}|\tilde{X}^*_m| \le \sum_{m\in[j\,..\,j'+1]}|X_{1,m}| \le 2\sum_{m\in[j\,..\,j']}|X_{1,m}| < \epsilon\sum_{m\in[j\,..\,j'+1]}|\tilde{X}^*_m|. \tag{4.1}$$

The right hand side of (4.1) implies:

$$\delta_E(\bigcirc_{m\in[j\,..\,j'+1]}X_{1,m}, \bigcirc_{m\in[j\,..\,j'+1]}\tilde{X}^*_m) \ge (1-\epsilon)|\bigcirc_{m\in[j\,..\,j'+1]}\tilde{X}^*_m|.$$

It also implies that for any partition of $\bigcirc_{m\in[j\,..\,j'+1]}\tilde{X}^*_m$ into substrings $\{X^*_m\}_{m\in[j\,..\,j'+1]}$, we have:

$$\sum_{m\in[j\,..\,j'+1]}\delta_E(X_{1,m}, X^*_m) \le (1+\epsilon)|\bigcirc_{m\in[j\,..\,j'+1]}\tilde{X}^*_m|.$$

And so if ϵ is sufficiently small:

$$\sum_{m\in\cup_{e\in S'}e}\delta_E(X_{1,m}, X^*_m) \le \frac{1+\epsilon}{1-\epsilon}\sum_{m\in\cup_{e\in S'}e}\delta_E(X_{1,m}, \tilde{X}^*_m) \le (1+3\epsilon)\sum_{m\in\cup_{e\in S'}e}\delta_E(X_{1,m}, \tilde{X}^*_m)$$

In particular, because of the left hand side of (4.1), we can choose the partition of $\bigcirc_{m\in[j\,..\,j'+1]}\tilde{X}^*_m$ that splits it into substrings $\{X^*_m\}_{m\in[j\,..\,j'+1]}$, each of length at most $2\tau/\epsilon$.

Now if we set $X^*_m = \tilde{X}^*_m$ for any m not in a subsequence in S', we trivially have:

$$\sum_{m\notin\cup_{e\in S'}e}\delta_E(X_{1,m}, X^*_m) \le \sum_{m\notin\cup_{e\in S'}e}\delta_E(X_{1,m}, \tilde{X}^*_m)$$

And also $X^*_m \le 2\tau/\epsilon$ for all such m (otherwise, m should appear in some subsequence in S' by condition (iii)). So we've found a partition of X^* into substrings $\{X^*_j\}_{j\in[J]}$ such that $|X^*_j| \le \frac{1}{\epsilon}|X_{1,j}|$ for all j, and:

$$\sum_{j\in[J]}\delta_E(X_{1,j}, X^*_j) \le (1+3\epsilon)\sum_{j\in[J]}\delta_E(X_{1,j}, \tilde{X}^*_j).$$

Now, we will use this partition to determine $\{X_{i,j}\}_{i>1,j\in[J]}$. For each i, X_i can be partitioned into substrings $X'_{i,j}$ such that $\delta_E(X_i, X^*) = \sum_{j\in[J]}\delta_E(X'_{i,j}, X^*_j)$. If $|X'_{i,j}| \le 2\tau/\epsilon^2$, we set $X_{i,j} = X'_{i,j}$. If any $X'_{i,j}$ has length larger than $2\tau/\epsilon^2 > |X^*_j|/\epsilon$, then $\delta_E(X'_{i,j}, X^*_j) \ge (1-\epsilon)|X'_{i,j}|$. On the other hand:

$$\delta_E(\gamma, X^*_j) + |X'_{i,j}| \le (1+\epsilon)|X'_{i,j}| \le \frac{1+\epsilon}{1-\epsilon}\delta_E(X'_{i,j}, X^*_j) \le (1+3\epsilon)\delta_E(X'_{i,j}, X^*_j)$$

for the empty string γ. So we can now choose $X_{i,j}$ to be any empty substring of $X'_{i,j}$. These choices of $X_{i,j}$ give the properties of the lemma, completing the proof. ∎

Let $\mathcal{W}_1$ be the set of all windows we partition X_1 into, and $\mathcal{W}_i$ be the set of windows we define for X_i. Let $\mathtt{s}(W)$ denote the index of the first character in W, and $\mathtt{e}(W)$ denote the index of the last character. For k strings, we define a window-respecting alignment as follows:

DEFINITION 4.2. *A window respecting alignment is a function $f : \mathcal{W}_1 \to \mathcal{W}_1 \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_k$ with the following properties:*

- *For all $W \in \mathcal{W}_1$, $f(W)_1 = W$.*
- *For any $j < j'$ and any i, $\mathtt{e}(f(W_{1,j})_i) < \mathtt{s}(f(W_{1,j'})_i)$.*

Let $r_i(f)$ denote the number of characters in X_i that are not contained in $f(W)_i$ for any $W \in \mathcal{W}_1$. The cost of a window-respecting alignment is defined as follows:

$$\delta_E(f) := \sum_{j\in[J]}\delta_E(f(W_{1,j})) + \sum_i r_i(f).$$

Let $\mathcal{F}$ be the set of all window-respecting alignments. The following lemma shows that window-respecting alignments approximate normal alignments:

LEMMA 4.2.

$$\delta_E(X_1, X_2, \dots X_k) \le \min_{f \in \mathcal{F}} \delta_E(f) \le (1 + 13\epsilon k)\delta_E(X_1, X_2, \dots X_k).$$

Proof. The first inequality follows because for any f, there is an alignment that for all j exactly aligns $W_{1,j}$ with the windows in $f(W_{i,j})$ at cost at most $\delta_E(f(W_{i,j}))$, and uses $\sum_i r_i(f)$ deletions to handle the remaining characters in each string.

Next, we show that there exists $f \in \mathcal{F}$ such that $\delta_E(f) \le (1+\epsilon k)\delta_E(X_1, X_2, \dots X_k)$. Let us take the substrings $X_{i,j}$ given by Lemma 4.1. Note that $W_{1,j} = X_{1,j}$.

If $|X_{i,j}| \le \epsilon\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + 3(\sigma - 1)$, let $W_{i,j}$ be the empty window "starting" at index $\mathsf{e}(W_{i,j-1}) + 1$, or if $j = 1$, at index 1. Then we have $|X_{i,j}| - |W_{i,j}| \le \epsilon\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + 3(\sigma - 1)$.

Otherwise, let $W_{i,j}$ be the longest window $W_{i,\Delta,p}$ that is a substring of $X_{i,j}$. Note that $|X_{i,j}|$ and $|W_{1,j}|$ differ by at most $\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j})$ and $|W_{i,j}| \le 2\tau/\epsilon^2$ for all i, j. If ϵ is a sufficiently small constant, this implies there is a choice of $W_{i,j}$ such that $|X_{i,j}| - |W_{i,j}| \le \epsilon\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + 3(\sigma - 1)$. We can identify $W_{i,j}$ as follows: Take $X_{i,j}$ and delete at most $\sigma - 1$ characters from the beginning until it starts at $p\sigma + 1$ for some integer p to get $\tilde{X}_{i,j}$. We have $|X_{i,j}| - |\tilde{X}_{i,j}| \le \sigma - 1$, and so $|\tilde{X}_{i,j}|$ and $|W_{1,j}|$ differ by at most $\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + (\sigma - 1)$ characters. Choose Δ such that $\frac{1}{1+\epsilon}(\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + (\sigma - 1)) \le \Delta \le \delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + (\sigma - 1)$. $W_{i,\Delta,p}$ is a prefix of $\tilde{X}_{i,j}$ containing all but at most the last $\epsilon(\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + (\sigma - 1)) + (\sigma - 1)$ characters of $\tilde{X}_{i,j}$. In turn, if ϵ is sufficiently small we have $|X_{i,j}| - |W_{i,j}| \le \epsilon\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + 3(\sigma - 1)$.

In turn, by triangle inequality and since $\sigma - 1 \le \epsilon k D\tau/|X_1|$:

$$\delta_E(W_{1,j}, W_{2,j} \dots W_{k,j}) \le (1 + \epsilon k)\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + 3\epsilon k D\tau/|X_1|.$$

We now choose $f(W_{1,j}) = (W_{1,j}, W_{2,j} \dots W_{k,j})$. We also have that the number of characters f does not align within $X_{2,j}, X_{3,j} \dots X_{k,j}$ is at most $\epsilon k\delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + 3\epsilon k D\tau/|X_1|$.

Putting it all together and using Lemma 4.1 we get:

$$\begin{aligned}
\delta_E(f) :=& \sum_{j \in [J]} \delta_E(f(W_{1,j})) + \sum_i r_i(f) \\
\le& (1 + \epsilon k) \sum_j \delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + \frac{|X_1|}{\tau} \cdot \frac{3\epsilon k D\tau}{|X_1|} \\
&+ \epsilon k \sum_j \delta_E(W_{1,j}, X_{2,j} \dots X_{k,j}) + \frac{|X_1|}{\tau} \cdot \frac{3\epsilon k D\tau}{|X_1|} + \sum_i (|X_i| - \sum_{j \in [J]} |X_{i,j}|) \\
\le& (1 + 2\epsilon k) \sum_j \delta_E(X_{1,j}, X_{2,j}, \dots X_{k,j}) + 6\epsilon k\delta_E(X_1, X_2, \dots, X_k) + \sum_i (|X_i| - \sum_{j \in [J]} |X_{i,j}|) \\
\le& (1 + 13\epsilon k)\delta_E(X_1, X_2, \dots X_k).
\end{aligned}$$

□

4.2 An Efficient Algorithm for Window-Respecting Alignments Our algorithm, denoted k-ED-ALG, is as follows:

1. Let $\mathcal{D} := \{1, 2, 4, \dots, 2k\tau/\epsilon^2\}$. For X_1 and each d in $\mathcal{D}$, identify a set $\tilde{\mathcal{W}}_{1,d}$ of "representative" strings such that (i) $|\tilde{\mathcal{W}}_{1,d}| = O(n\tau/\epsilon d)$ and (ii) for every window $W_{1,i}$, there is some string $\mathtt{shift}_d(W_{1,i}) \in \tilde{\mathcal{W}}_{1,d}$ in such that $\delta_E(W_{i,\Delta,p}, \mathtt{shift}_d(W_{i,\Delta,p})) \le \epsilon d$.
2. For each other string X_i,each value of d in $\mathcal{D}$, and each value of Δ, identify a set of "representative" length $\tau + \Delta$ strings $\tilde{\mathcal{W}}_{i,d,\Delta}$ such that (i) $|\tilde{\mathcal{W}}_{i,d,\Delta}| = O(n(\tau + \Delta)/\epsilon d)$, and (ii) for every window $W_{i,\Delta,p}$, there is some string $\mathtt{shift}_d(W_{i,\Delta,p}) \in \tilde{\mathcal{W}}_{i,d,\Delta}$ such that $\delta_E(W_{i,\Delta,p}, \mathtt{shift}_d(W_{i,\Delta,p})) \le \epsilon d$.
3. Let $\tilde{\mathcal{W}}_{i,d} = \cup_\Delta \tilde{\mathcal{W}}_{i,d,\Delta}$. For each $d \in \mathcal{D}$ and every k-tuple of strings $\tilde{W}_{1,d}, \tilde{W}_{2,d} \dots \tilde{W}_{k,d}$ in $\tilde{\mathcal{W}}_{1,d} \times \tilde{\mathcal{W}}_{2,d} \times \cdots \times \tilde{\mathcal{W}}_{k,d}$, compute the median distance of this k-tuple if it is less than d. Store this as $\tilde{\delta_E}(\tilde{W}_{1,d}, \tilde{W}_{2,d} \dots \tilde{W}_{k,d}) + \epsilon kd$. If the true median distance of these windows is greater than d, store $\tilde{\delta_E}(\tilde{W}_{1,d}, \tilde{W}_{2,d} \dots \tilde{W}_{k,d}) = \infty$ instead.

4. Our algorithm solves the following dynamic program:

$$c(x_1, x_2, \dots x_k) = \min \begin{cases} \min_{i \neq 1} c(x_1, x_2, \dots, x_i - \sigma, \dots, x_k) + \sigma \\ \min_{W_1, \dots W_k \in \mathcal{W}_1 \times \dots \times \mathcal{W}_k : \forall_i \mathtt{e}(W_i) = x_i, W_1 = \dots = W_k} [c(\mathtt{s}(W_1) - 1, \dots, \mathtt{s}(W_k) - 1)] \\ \min_{W_1, \dots W_k \in \mathcal{W}_1 \times \dots \times \mathcal{W}_k : \forall_i \mathtt{e}(W_i) = x_i} [c(\mathtt{s}(W_1) - 1, \dots, \mathtt{s}(W_k) - 1) \\ + \min_{d \in \mathcal{D}} \tilde{\delta_E}(\mathtt{shift}_d(W_1), \dots, \mathtt{shift}_d(W_k))] \end{cases}$$

For every k-tuple such that x_1 is a multiple of τ, $x_2, \dots, x_k$ are all multiples of σ, and such that $|x_i - x_1| \leq 4D$ for all i.
The base case is $c(0, 0, \dots, 0) = 0$, and our final output is $c(|X_1|, R_u(N_2), \dots R_u(N_k))$, where $R_u(x)$ denotes x rounded up to the nearest multiple of σ.

At a high-level, in steps 1 and 2 of k-ED-Alg we exploit the compression of the input strings to identify a small set of "representative" strings for each X_i, such that for each window in X_i there is a representative string within small edit distance of that window. In step 3, we then compute the median distance between k-tuples of representative strings (instead of between all k-tuples of windows). Since all windows are within a small distance of some representative string, this also gives for all k-tuples of windows a reasonable approximation of their median distance. Step 4 of k-ED-Alg uses these approximations to solve a natural DP for finding an optimal window-respecting alignment. This DP is the same as the standard DP for edit distance, but instead of matching characters we are only allowed to match windows, at cost equal to (the approximation of) their median distance.

We first bound the runtime of k-ED-Alg. The following lemmas show that Steps 1 and 2 of k-ED-Alg can be performed efficiently (as well as their correctness):

Lemma 4.3. *Given a straight-line program $\mathbf{G}$ of size n that generates a string X of size n, a length parameter τ, and a parameter $\delta_{\max} \leq \tau$, there exists an algorithm that in time $O(|X|)$ finds (an implicit representation of) a set S of $O(n\tau/\delta_{\max})$ substrings of length at most τ such that for every length τ substring of X, x, there is a string $\mathtt{shift}(x)$ in S such that $\delta_E(x, \mathtt{shift}(x)) \leq \delta_{\max}$. We can also construct a data structure that identifies $\mathtt{shift}(x)$ given the starting location of x in X using $O(|X|)$ preprocessing time and $O(1)$ query time.*

Proof. If $\delta_{\max} \geq \tau$, we can trivially choose S that only contains the empty substring, and the data structure just returns the empty substring for any query. So assume $\delta_{\max} < \tau$.

Given that $\mathbf{G}$ has size n, the optimal LZ77 factorization of X has size at most n [Ryt03]. We will first show the existence of S for any X that has an LZ77 factorization of size at most n. For brevity, we will not go into the details of LZ77 factorization here. The key property we need is that a string X that has a LZ77 factorization of size n can be written as $X_1 \circ X_2 \circ X_3$, where X_1 is a string with LZ77 factorization of size $n-1$, X_2 is a substring of X_1, and X_3 is a single character. Moreover, the factorization gives the location of X_2 in X_1.

Inductively, suppose we have constructed S, a set of at most $3(n-1)\tau/\delta_{\max}$ substrings that has the desired properties for X_1. For all "good" indices $i < |X_2| - \tau$, the length τ substring starting at the ith character in X_2 is fully contained in X_2, and thus is a substring in X_1. This leaves at most $\tau + 1$ "bad" indices where the length τ substring starting at these indices may not have a nearby string in S: those starting at indices $|X_1| - \tau + 1$ to $|X_1|$ of X_1, and the substring starting at index $|X_2| - \tau + 2$ of X_2. Consider the length τ substring starting at every $(\delta_{\max}/2)$-th position in indices $|X_1| = \tau + 1$ to $|X_1|$ of X_1, as well as the length τ substring starting at index $|X_2| - \tau + 2$ of X_2. This set of strings has size at most $2\tau/\delta_{\max} + 1 \leq 3\tau/\delta_{\max}$, and every length τ substring starting at a bad index is within edit distance $\delta_{\max}$ of some string in this set. So adding these strings to S gives that S now has size at most $3n\tau/\delta_{\max}$ and has the desired properties.

For an efficient implementation of this procedure, we can compute the optimal LZ77 factorization in $O(|X|)$ time [RPE81]. Given the LZ77 factorization, we decompose X into $X_1 \circ X_2 \circ X_3$ as before, and recursively compute an array A for indices in $[1 \,..\, |X_1| - \tau + 1]$ and set B with the following property: the length τ substrings starting at indices i and $A[i]$ are within edit distance $\delta_{\max}$, and A has at most $3(n-1)\tau/\delta_{\max}$ distinct values, which are exactly the values in B.

Since the LZ77 factorization gives us the position of X_2 in X_1, we can fill in A for the "good" indices in X_2 in time linear in the number of good indices. We can also fill in the values of A for the bad indices, in time linear in the number of bad indices, and add these values to B. Overall, the algorithm takes linear time to compute A, B. A now serves as the desired efficient data structure, and B as our implicit representation of S. ∎

There are $O(\log N)$ values of d and $\tilde{O}(\log N/\epsilon)$ values of Δ, so we can do Steps 1 and 2 in time $\tilde{O}(N/\epsilon)$ time. We also show that Step 3 can be performed efficiently:

LEMMA 4.4. *Given strings $X_1, X_2, \dots X_k$, there exists a data structure that can be computed in $O(\sum_i |X_i|)$ time that can answer queries of the following form in $O(d^k)$ time: Given indices $s_1, s_2, \dots, s_k$ and $e_1, e_2, \dots, e_k$, if $\delta_E(X_1[s_1 \dots e_1], X_2[s_2 \dots e_2], \dots X_k[s_k \dots e_k]) \le d$, output $\delta_E(X_1[s_1 \dots e_1], X_2[s_2 \dots e_2], \dots X_k[s_k \dots e_k])$, otherwise output ∞.*

Proof. Given $X_1, X_2, \dots X_k$, let $\texttt{Slide}_{d_2,d_3,\dots d_k}(j) = \max\{q : X_1[j \dots q] = X_2[j+d_2 \dots q+d_2] = X_3[j+d_3 \dots q+d_3] = \dots = X_k[j+d_k \dots q+d_k]\}$. We can rewrite $\texttt{Slide}_{d_2,d_3,\dots d_k}(j)$ as $\min_{i\in\{2,3,\dots,k\}} \max\{q : X_1[j \dots j+q] = X_i[j+d_i \dots q+d_i]\}$. Section 2.3 of [LMS98] shows that we can compute $\max\{q : X_1[j \dots j+q] = X_i[j+d_i; q+d_i]\}$ for any i, j, d_i in $O(1)$ time after $O(|X_1| + |X_i|)$ preprocessing time. So we can compute $\texttt{Slide}_{d_2,d_3,\dots d_k}(i)$ in $O(1)$ time after $O(\sum_i |X_i|)$ preprocessing time (recall that $k = O(1)$).

Let $L^h(d_2, d_3, \dots d_k)$ be the largest value of j such that $\delta_E(X_1[1 \dots j], X_2[1 \dots j+d_2], X_3[1 \dots j+d_3], \dots, X_k[1 \dots j+d_k]) \le h$. We have the following recurrence relation:

$$L^h(d_2, d_3 \dots, d_k) = \texttt{Slide}\left(\max\left\{\begin{array}{l} L^{h-1}(d_2+1, d_3+1 \dots, d_k+1) \\ \max_i L^{h-1}(d_2, d_3, \dots, d_i-1, \dots, d_k) \\ \max_{e\in\{0,1\}^k} L^{h-w}(d_2-e_2+e_1, d_3-e_2+e_1, \dots, d_k-e_k+e_1) \\ \qquad \text{for } w = \min_{i:e_i=1} |\{j \ne i : X_i[x_i] \ne X_j[x_j] \vee e_j = 0\}| \end{array}\right\}\right)$$

The first case considers deleting from X_1, the second case considers deleting a character from any of $X_2, X_3, \dots X_k$, and the third case considers inserting characters into some subset of the strings (for which $e_i = 0$), and then matching the inserted characters with a character in the remaining strings (for which $e_i = 1$), such that we use at most w insertions or substitutions.

Each $L^h(\cdot)$ only depends on $O(1)$ other values, and so we can compute each value in $O(1)$ time. In turn, we can compute the values $L^h(d_2, d_3, \dots d_k)$ for all $0 \le h \le d, 0 \le d_2 + d_3 + \dots + d_k \le d$ in $O(d^k)$ time. Our output for the edit distance is the smallest h such that $L^h(|X_2| - |X_1|, |X_3| - |X_1|, \dots, |X_k| - |X_1|) \ge |X_1|$, or ∞ if $L^d(|X_2| - |X_1|, |X_3| - |X_1|, \dots, |X_k| - |X_1|) < |X_1|$. ∎

The total number of strings in any $\tilde{\mathcal{W}}_{i,d}$ is $\sum_{\Delta\in\mathcal{D}} O(n(\tau+\Delta)/\epsilon d) = O(n\tau/\epsilon^3 d)$. In turn, combined with Lemma 4.4, the total time needed to compute $\tilde{\delta_E}$ for all k-tuples in $\tilde{\mathcal{W}}_{1,d} \times \tilde{\mathcal{W}}_{2,d} \times \dots \times \tilde{\mathcal{W}}_{k,d}$ is $O(n^k\tau^k/\epsilon^{3k})$. There are $O(\log N)$ choices of d, so in total this step takes time $\tilde{O}(n^k\tau^k/\epsilon^{3k})$.

For Step 4, it takes $O(1)$ time to process the first case in the recurrence relation. For the second and third case, there are $O(\log N)$ values of d, $O(\log N/\epsilon)$ values of Δ, and for each i, x_i, Δ there is 1 window $W_{i,\Delta,p}$ such that $\tilde{e}(W_i) = x_i$. Lemma 4.4 gives an $O(1)$-time method to determine if $W_1 = W_2 = \dots = W_k$ in the second case, and we have precomputed all the necessary values in the third case. So, the time to compute each $c(x_1, x_2, \dots x_k)$ is $O((\log^2 N/\epsilon)^k)$.

The number of tuples $x_1, x_2, \dots x_k$ such that $\sum_{i\ne 1} |x_i - x_1| \le D$ is $O(ND^{k-1})$. Of these, fraction $O(\frac{1}{\lceil \epsilon D\tau/N\rceil^{k-1}\tau})$ satisfy that x_1 is a multiple of τ and $x_2 \dots x_k$ are multiples of $\lceil \epsilon D\tau/N\rceil$. So the number of entries we need to compute is $O(N^k/\epsilon^k\tau^k)$, and the total time to compute all these entries is $\tilde{O}(N^k/\epsilon^{2k}\tau^k)$.

Putting it all together, Steps 3 and 4 dominate the runtime with total runtime $\tilde{O}(N^k/\epsilon^{2k}\tau^k + n^k\tau^k/\epsilon^{3k})$. Setting $\tau = (N/n\epsilon)^{1/2}$, we get an overall runtime of $\tilde{O}(N^{k/2}n^{k/2}\epsilon^{-5k/2})$.

We complete our analysis by showing that the final value computed by k-ED-ALG is close to $\delta_E(X_1, X_2, \dots X_k)$.

LEMMA 4.5. *k-ED-ALG outputs $\tilde{D}$ such that*

$$\delta_E(X_1, X_2, \dots X_k) \le \tilde{D} \le (1 + 19\epsilon k)\delta_E(X_1, X_2, \dots X_k).$$

Proof. Consider any window-respecting alignment for which $f(W_{1,j}) = (W_{1,j}, W_{2,j}, \dots, W_{k,j})$. If $\delta_E(W_{1,j}, W_{2,j}, \dots, W_{k,j}) > 0$, let d_j be the smallest value in $\mathcal{D}$ such that $d_j \ge \delta_E(W_{1,j}, W_{2,j}, \dots, W_{k,j}) + \epsilon k d_j$. By Lemma 4.3, for every j and if ϵ is sufficiently small, by triangle inequality we have:

$$\begin{aligned}\delta_E(W_{1,j}, W_{2,j}, \ldots, W_{k,j}) &\leq [\tilde{\delta_E}(\texttt{shift}_{d_j}(W_{1,j}), \texttt{shift}_{d_j}(W_{2,j}), \ldots, \texttt{shift}_{d_j}(W_{k,j}))] + \epsilon k d_j \\ &\leq \delta_E(W_{1,j}, W_{2,j}, \ldots, W_{k,j}) + 2\epsilon k d_j \\ &\leq (1 + 5\epsilon k)\delta_E(W_{1,j}, W_{2,j}, \ldots, W_{k,j}).\end{aligned}$$

The first inequality implies that any path through the DP table for c has total cost at least that of some window-respecting alignment, which by Lemma 4.2 gives the first inequality in the lemma statement. The second inequality implies that for the best window-respecting alignment, there is a path through the DP table such that the cost of the path through the DP table is no more than $(1 + 5\epsilon)$ times the cost of the window-respecting alignment. Furthermore, this path only goes through points in the DP table such that $|x_i - x_1| \leq 4D$ for all i, i.e. is considered by k-ED-Alg. Combined with Lemma 4.2 this gives the second inequality in the lemma statement if ϵ is sufficiently small. □

We can now compute a $(1 + \epsilon)$-approximation of the edit distance by rescaling ϵ appropriately and running k-ED-Alg for all D that are powers of 2, giving Theorem 4.1. One could also extract the alignment achieving this edit distance by using standard techniques to retrieve a path through the DP table, and applying these same techniques to the DP tables used in invocations of Lemma 4.4 as a subroutine; we omit the details here.

5 FPTAS For Center Distance

The center distance problem is defined as follows:

Definition 5.1. *The center (edit) distance $\delta_{CE}(X_1, \ldots, X_k)$ of k strings $X_1, \ldots, X_k$ is defined as $\delta_{CE}(X_1, \ldots, X_k) = \min_{X^*} \max_i \delta_E(X_i, X^*)$. That is, it is the smallest value D such that by making at most D edits to each X_i, we can transform them all into the same string X^*.*

In this section we prove Theorem 5.1:

Theorem 5.1. *Given $k = O(1)$ straight-line programs $\mathbf{G}_{X_i}$ of total size n generating strings X_i of total length $N > 0$ and a parameter $\epsilon \in (0, 1]$, an integer between $\delta_{CE}(X_1, \ldots, X_k)$ and $(1+\epsilon)\delta_{CE}(X_1, \ldots, X_k)$ can be computed in $O\left(\epsilon^{-O(k)} n^{k/2} N^{k/2+o(1)}\right)$ time.*

Prior to our work, the best known algorithm result for the center distance problem was the exact $O(N^{2k})$-time algorithm of [NR05]. Our framework for the algorithm is similar to the framework from the previous section which uses window-respecting alignments.

Our algorithm will actually solve a more general problem of computing an approximation of a set of values which we call the *edit tuples*. We again assume $\delta_{CE}(X_1, \ldots, X_k)$ lies between D and $2D$ for some known (power of 2) D.

Definition 5.2. *Given strings $X_1, X_2, \ldots, X_k$, an edit tuple of these strings is a vector $v \in \mathbb{Z}^k_{\geq 0}$ such that there exists X^* for which $\delta_E(X_i, X^*) \leq v_i$ for all i. We denote the set of all edit tuples in $\{0, 1, \ldots, D\}^k$ of $X_1, X_2, \ldots X_n$ by $\boldsymbol{tup}_D(X_1, X_2, \ldots X_k)$.*

We say that S is a Δ-approximation of $\boldsymbol{tup}_D(X_1, X_2, \ldots X_k)$ if for each $v \in S$, there is a vector $v' \in \boldsymbol{tup}_D(X_1, X_2, \ldots X_k)$ such that $v' \leq v$, and for each $v \in \boldsymbol{tup}_D(X_1, X_2, \ldots X_k)$, there is a vector $v' \in S$ such that $v' \leq v + \Delta \cdot \mathbf{1}$. Here $a \leq b$ denotes $a_i \leq b_i$ for all i and $\mathbf{1}$ denotes the all ones vector.

We will use again use the window-respecting alignment framework. However, our algorithm is now recursive, and thus we need to be careful about choosing the windows to operate with in each level of recursion. Let $\ell = O(\log \log N)$ be a parameter and $\tau_0 = N > \tau_1 > \cdots > \tau_\ell = N^{1/\log \log N}$ be a sequence such that for all $i < \ell$, $\tau_m/\tau_{m-1} = \Theta(N^{1/\log \log N})$ and is integer (that is, these ratios are not necessarily the same but are all within a constant factor of $N^{1/\log \log N}$). We will also eventually choose a sequence of error parameters for each level $\epsilon_0, \epsilon_1, \ldots, \epsilon_\ell$. Let $\mathcal{D} = \{1, 2, 4, \ldots, N\}$, and for each $d \in \mathcal{D}$ let $\sigma_m(d) := \max\{\frac{\epsilon_{m+1} d \tau_{m+1}}{\tau_m} + \frac{\epsilon_{m+1} D \tau_{m+1}}{|X_1|}, 1\}$ rounded down to the nearest power of 2. For each $i > 0$, for τ_m, each $d \in \mathcal{D}$, the corresponding ϵ_m, $\sigma_m(d)$, and R_d defined

as the function rounding down to the nearest multiple of $\sigma_m(d)$, we define windows in each string just as in Section 4. In particular, for X_1 we have windows $W_{1,m,p}$ that are again just a partition of X_1 into substrings of length τ_m, and for $X_2, \dots, X_k$ we have windows $W_{i,m,d,\Delta,p} = X_i[p\sigma_m(d)+1\,..\,R_d(p\sigma_m(d)+\tau_m+\Delta)]$, where the set of possible Δ is defined by ϵ_m and τ_m. We will refer to these as the windows at level m. Note that we are using the same guess D to define windows at all levels of recursion, even though at lower levels of recursion the center distance between the substrings we consider is likely to be much smaller even if our guess is accurate at the first level.

We note some properties of our recursion that motivate this choice of windows: In the ith level of recursion, if our subproblems' input is $X'_1, \dots, X'_k$, then we will have the guarantee that X'_1 is a window in X_1 of length τ_m and $X'_2, \dots, X'_k$ are one of the windows in $X_2, \dots, X_k$ corresponding to τ_m. When we are solving a subproblem involving a length τ_m substring of X_1, we will use the windows defined by $\tau = \tau_{m+1}$. In addition, when we are solving this subproblem, by our requirement that all $\sigma_m(d)$ be a power of 2, we have the following property: the windows defined on the full strings $X_1, \dots X_k$ for τ_{m+1} that are contained within $X'_1, \dots, X'_k$, are equivalent to the windows we would define within $X'_1, \dots, X'_k$ if we used the same choice of parameters τ_{m+1}, σ_{m+1}. We will refer to this set of windows as the windows at level $m+1$ restricted to $X'_1, \dots, X'_k$.

To give some intuition behind the choice of $\sigma_m(d)$, which is crucial for our analysis: The term with d is a "local" term. It contributes to the approximation error locally, only adding error proportional to our center distance estimate for the current tuple of windows, and also helps us keep the number of entries in the DP table within one call small. The term with D is a "global" term. It contributes to the approximation error globally; across all recursive calls, the final approximation error accumulated at the top level due to this term will be something like ϵD. It also keeps the number of windows across all recursive calls small.

Now, for a fixed level m and the corresponding windows, we can define window respecting alignments of $X_1, \dots, X_k$ identically to Definition 4.2. If we are considering a window-respecting alignment of substrings $X'_1, \dots, X'_k$ instead of the full strings, we simply restrict to the windows contained within these substrings, and then define window-respecting alignments of $X'_1, \dots, X'_k$ as before using these sets of windows. We define the edit tuples of a window-respecting alignment f, $\texttt{tup}_D(f)$, to be:

$$[\circledast_{j\in[J]}\texttt{tup}_{\max_i |f(W_{1,j})_i|}(f(W_{1,j})) \circledast \{r(f)\}] \cap \{0,1,\dots D\}^k$$

Where $\circledast$ is the convolution of sets of vectors, i.e. $\circledast_i S_i = \{\sum_i v_i | v_i \in S_i \forall i\}$, and $r(f)$ is the vector whose ith entry is $r_i(f) = |X_i| - \sum_j |f(W_{1,j})_i|$, i.e., the number of characters in X_i not in any window. Similarly to Lemma 4.2, we can show window-respecting alignments approximate the best standard alignment.

LEMMA 5.1. *Let d be any value in $\mathcal{D}$. Let $X'_1, \dots, X'_k$ be windows in $X_1, \dots, X_k$ at the same level m. Let $\mathcal{F}$ be the set of window-respecting alignments of $X'_1, \dots, X'_k$, using the windows at level $m+1$ parametrized by d, restricted to $X'_1, \dots, X'_k$. Then we have that $\cup_{f\in\mathcal{F}} \texttt{tup}_{3d}(f)$ is a $(13\epsilon_{m+1}kd + 6\epsilon_{m+1}D\tau_m/|X_1|)$-approximation of $\texttt{tup}_{2d}(X'_1, X'_2, \dots X'_k)$.*

Proof. First, we will show that for any f and $v \in \texttt{tup}_{3d}(f)$, v is also an edit tuple of $X'_1, X'_2, \dots, X'_k$. Let $J = \tau_m/\tau_{m+1}$. For $v \in \texttt{tup}_{3d}(f(W_{1,j}))$, it can be decomposed as $\sum_{j\in[J]} v_j + r(f)$, where v_j is an edit tuple of $f(W_{1,j})$. By deleting the $r_i(f)$ characters in each X_i that are not in any $W_{i,j}$, we get the string $\bigcirc_j W_{i,j}$ for each i, and $\sum_{j\in[J]} v_j$ is clearly a valid edit tuple for these strings. So v is an edit tuple of $X'_1, X'_2, \dots, X'_k$.

It now suffices to show that for any edit tuple v of $X'_1, \dots X'_k$ in $\{0,1,\dots,2d\}^k$, there exists f and v' in $\texttt{tup}_{3d}(f)$ such that $v' \le v + (9\epsilon_{m+1}kd + 6\epsilon_{m+1}D\tau_m/|X_1|)\cdot \mathbf{1}$. Fix any such v. We partition X'_1 into substrings of length τ_{m+1}, $\{X_{1,j}\}_{j\in[J]}$. Let X^* be the string such that $\delta_E(X'_i, X^*) \le v[i]$ for all i. Using the same procedure as in Lemma 4.1, we can find a partition of X^* into substrings $\{X^*_j\}$ such that each X^*_j has length at most $2\tau_{m+1}/\epsilon_{m+1}$ and:

$$\sum_{j\in[J]} \delta_E(X_{1,j}, X^*_j) \le (1+3\epsilon_{m+1})v[1] \le v[1] + 6\epsilon_{m+1}d.$$

Given this partition, again using the same procedure as in Lemma 4.1, we can find disjoint substrings of X_i, $X_{i,j}$, for all $i > 1$ such that each $X_{i,j}$ has length at most $2\tau/\epsilon^2_{m+1}$ and

$$\sum_{j\in[J]} \delta_E(X_{i,j}, X^*_j) + |X_i| - \sum_{j\in[J]} |X_{i,j}| \le (1+3\epsilon_{m+1})v[i] \le v[i] + 6\epsilon_{m+1}d.$$

Now, let $W_{1,j} = X_{1,j}$ for all j. Similarly to Lemma 4.2, for each i, if $|X_{i,j}| \leq 2\epsilon_{m+1} \max_{i'} \delta_E(X_{i',j}, X_j^*) + 6\epsilon_{m+1} D\tau_{m+1}/|X_1|$, let $W_{i,j}$ be the empty window "starting" at index $\mathtt{e}(W_{i,j-1}) + 1$ (or index 1 if $j = 1$). Otherwise, let $W_{i,j}$ be the longest window $W_{i,m+1,\Delta,p}$ that is a substring of $X_{i,j}$. Note that $|X_{i,j}|$ and $|W_{i,j}|$ differ by at most $2\max_{i'} \delta_E(X_{i',j}, X_j^*)$. If ϵ_{m+1} is a sufficiently small constant, similarly to the proof of Lemma 4.2, this implies there is a choice of $W_{i,j}$ such that $|X_{i,j}| - |W_{i,j}| \leq \epsilon_{m+1} \max_{i'} \delta_E(X_{i',j}, X_j^*) + 3(\sigma_m(d) - 1) \leq \epsilon_{m+1} \max_{i'} \delta_E(X_{i',j}, X_j^*) + 6[\frac{\epsilon_{m+1} d\tau_{m+1}}{\tau_m} + \frac{\epsilon_{m+1} D\tau_{m+1}}{|X_1|}]$. Note that $\sum_{j\in[J]} \max_i \delta_E(X_{i,j}, X_j^*) \leq k\,||v||_\infty \leq 2kd$. This implies $r_i(f) - (|X_i| - \sum_{j\in[J]} |X_{i,j}|)$ is at most $5\epsilon_{m+1}kd$. We also have by triangle inequality that:

$$\delta_E(W_{i,j}, X_j^*) \leq \delta_E(X_{i,j}, X_j^*) + \epsilon \max_{i'} \delta_E(X_{i',j}, X_j^*) + 6[\frac{\epsilon_{m+1} d\tau_{m+1}}{\tau_m} + \frac{\epsilon_{m+1} D\tau_{m+1}}{|X_1|}].$$

Now consider the alignment that chooses $f(W_{1,j}) = (W_{1,j}, W_{2,j}, \ldots, W_{k,j})$. For each j, by the above inequalities, one edit tuple for $f(W_{1,j}) = (W_{1,j}, W_{2,j}, \ldots, W_{k,j})$ arising from a window-respecting alignment is element-wise at most:

$$\begin{aligned}
&(\delta_E(X_{1,j}, X_j^*),\\
&\delta_E(X_{2,j}, X_j^*) + \epsilon_{m+1} \max_i \delta_E(X_{i,j}, X_j^*) + 6[\frac{\epsilon_{m+1} d\tau_{m+1}}{\tau_m} + \frac{\epsilon_{m+1} D\tau_{m+1}}{|X_1|}],\\
&\ldots,\\
&\delta_E(X_{k,j}, X_j^*) + \epsilon_{m+1} \max_i \delta_E(X_{i,j}, X_j^*) + 6[\frac{\epsilon_{m+1} d\tau_{m+1}}{\tau_m} + \frac{\epsilon_{m+1} D\tau_{m+1}}{|X_1|}])
\end{aligned}$$

So summing up these edit tuples, and adding $r(f)$, we get a vector arising from a window-respecting alignment that is at element-wise at most $v + (\epsilon_{m+1}k\,||v||_\infty + 11\epsilon_{m+1}kd + 6\epsilon_{m+1}D\tau_m/|X_1|) \cdot \mathbf{1} \leq v + (13\epsilon_{m+1}kd + 6\epsilon_{m+1}D\tau_m/|X_1|) \cdot \mathbf{1}$. ∎

We are now ready to state our algorithm. Our recursive algorithm for computing a sparse approximation of $\mathtt{tup}_D(X_1, X_2, \ldots X_k)$, denoted k-CED-Alg, is defined as follows:

k-CED-Alg$(X_1', X_2', \ldots, X_k', d, m)$:

Let $\mathcal{W}_i$ denote the windows at level $m+1$ parametrized by d restricted to $X_1', X_2', \ldots, X_k'$, and $\mathtt{s}, \mathtt{e}$ be the functions that take a window and gives its starting/ending index in the corresponding X_i'. We solve the following dynamic program:

$$\begin{aligned}
c(x_1, x_2, \ldots x_k) = &(\cup_{i>1} c(x_1, x_2, \ldots, x_i - \sigma_{m+1}, \ldots, x_k) \circledast \{(0, 0, \ldots, \sigma_{m+1}, \ldots, 0)\}) \cup\\
&(\cup_{W_1, W_2, \ldots W_k \in \mathcal{W}_1 \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_k : \forall i, \mathtt{e}(W_i) = x_i} [c(\mathtt{s}(W_1) - 1, \mathtt{s}(W_2) - 1, \ldots, \mathtt{s}(W_k) - 1)\\
&\circledast \cup_{d' \in \mathcal{D} : d' \leq 2d} k\text{-CED-Alg}(W_1, W_2, \ldots, W_k, d', m+1)])
\end{aligned}$$

For every k-tuple such that x_1 is a multiple of τ_{m+1}, $x_2, \ldots, x_k$ are all multiples of σ_{m+1}, and such that $|x_i - x_1| \leq 3d + \frac{\epsilon_m D\tau_m}{|X_1|}$. The base case for the dynamic program is $c(0, 0, \ldots, 0) = \{(0, 0, \ldots, 0)\}$.

After computing each entry $c(x_1, x_2, \ldots x_k)$, we remove all elements of $c(x_1, x_2, \ldots x_k)$ not in $\{0, 1, \ldots 3d + \lfloor \frac{\epsilon_m D\tau_m}{|X_1|} \rfloor\}^k$. After getting a set of edit tuples from a call to k-CED-Alg, we round each coordinate of each vector up to the nearest multiple of $\sigma_m(d)$ before taking the convolution.

Our final output is $c(|X_1'|, |X_2'|, \ldots, |X_k'|)$, and then return this set of vectors.

Our base case will be when $m = \ell$, and we have that $|X_1'| = N^{1/\log\log N}$ and all $|X_i'|$ are at most $2N^{1/\log\log N}/\epsilon_\ell^2$. To handle the base case, we will enumerate all substrings of length at most $2N^{1/\log\log N}/\epsilon_\ell^2$ of each of $X_1, \ldots, X_k$, and compute their edit tuples using, e.g., the exact algorithm of [NR05]. Our top-level recursive call is to k-CED-Alg$(X_1, \ldots, X_k, D, 0)$.

To keep the algorithm's description consistent across levels, in addition to assuming X_1's length is a multiple of τ_1, we will assume that $X_2, \ldots, X_k$ are multiples of $\sigma_1(D)$; we can enforce this assumption by padding each of $X_2, \ldots, X_k$ with at most $\sigma_1(D)$ copies of a new dummy character. This cannot decrease the center distance and

the total increase in center distance due to this padding is at most $\sigma_1(D)$, which contributes an additive $o(\epsilon_0 D)$ to our approximation factor, only at the top level of recursion. By construction, at lower levels of the recursion each X'_i will have length that is a multiple of $\sigma_m(d)$ for $i > 1$, so by this assumption we no longer need to worry about rounding the indices of the value in DP table we output at any level.

5.1 Approximation Guarantee We first prove the approximation guarantee of k-CED-ALG, as it will be necessary for our runtime analysis to specify what choice of ϵ_0 to ϵ_ℓ is needed for the desired approximation guarantee.

LEMMA 5.2. *Let the sequence $\epsilon_0, \ldots, \epsilon_\ell$ satisfy $\epsilon_0 \leq 1$ and $\epsilon_{m+1} = \frac{1}{16k}\epsilon_m$ for all m. Then at level m of the recursion, each invocation of k-CED-ALG$(X'_1, X'_2, \ldots, X'_k, d, m)$ returns a set of edit tuples that is a $(\epsilon_m d + \frac{\epsilon_m D \tau_m}{|X_1|})$-approximation of $\mathtt{tup}_{2d}(X'_1, \ldots, X'_k)$.*

Proof. We proceed by induction. Clearly the guarantee holds for the base case $m = \ell$, since we solve the base cases using exact algorithms.

Inductively, assuming at level $m+1$, any edit tuple generated returned by k-CED-ALG is element-wise greater than some edit tuple of the corresponding windows, by an argument similar to the first part of the proof of Lemma 5.1 the same property holds at level m. So we just need to show that each edit tuple returned by k-CED-ALG is not too large an overestimate of some edit tuple of its input strings.

Take any edit tuple v for any window-respecting alignment f. Assume the approximation guarantee holds for calls made at level $m+1$. We show that for the corresponding path through the DP table for c, there is a vector close to v in the edit tuples generated by this path. v can be decomposed as $\sum_{j\in[J]} v_j + r(f)$ where v_j is an edit tuple of $f(W_{1,j})$. Let d_j be the smallest value in $\mathcal{D}$ such that $d_j \geq ||v_j||_\infty$. By our inductive hypothesis, for each j we get a $(\epsilon_{m+1}d_j + \frac{\epsilon_{m+1}D\tau_{m+1}}{|X_1}$-approximation of the edit tuples of $\mathtt{tup}_{d_j}(f(W_{1,j}))$ from the call to k-CED-ALG$(f(W_{1,j}), d_j, m+1)$, which includes a vector v'_j that is element-wise at most $v_j + (\epsilon_{m+1}d_j + \frac{\epsilon_{m+1}D\tau_{m+1}}{|X_1|}) \cdot \mathbf{1} \leq v_j + (2\epsilon_{m+1}\,||v_j||_\infty + \frac{\epsilon_{m+1}D\tau_{m+1}}{|X_1|})$ if ϵ is sufficiently small. In addition, the sum of the vectors contributed by the first case in the recurrence relation for c is $r(f)$. So there is an edit tuple computed by our algorithm for this path that is element-wise less than:

$$\sum_j [v_j + (2\epsilon_{m+1}\,||v_j||_\infty + \frac{\epsilon_{m+1}D\tau_{m+1}}{|X_1|}) \cdot \mathbf{1}] + r(f) \leq v + \left(4\epsilon_{m+1}kd + \frac{\epsilon_{m+1}D\tau_m}{|X_1|}\right) \cdot \mathbf{1}.$$

After accounting for the approximation error of window-respecting alignments due to Lemma 5.1 and the rounding step, the additive error is increased to at most $\left(16\epsilon_{m+1}kd + 8\frac{\epsilon_{m+1}D\tau_m}{|X_1|}\right) \cdot \mathbf{1} \leq (\epsilon_m d + \frac{\epsilon_m D\tau_m}{|X_1|}) \cdot \mathbf{1}$ as desired.

Finally, note that since we only remove vectors with values larger than $3d + \frac{\epsilon_m D\tau_m}{|X_1|}$ and assume ϵ_0 (and thus all ϵ_m) is at most 1, we do not remove any vector that would be in a $(\epsilon_m d + \frac{\epsilon_m D\tau_m}{|X_1|})$-approximation of $\mathtt{tup}_{2d}(X'_1, \ldots, X'_k)$. ∎

If we set $\epsilon_0 = \epsilon - o(1)$, then after accounting for the $o(\epsilon_0 D)$ error introduced by padding X_2 to X_k, the smallest ℓ_∞-norm of any vector in the output of k-CED-ALG$(X_1, \ldots, X_k, D, 0)$ gives a $(1+\epsilon)$-multiplicative approximation of the center distance as desired.

5.2 Runtime Analysis We now bound the runtime of k-CED-ALG, completing the proof of Theorem 5.1.

LEMMA 5.3. *For the choice of $\epsilon_0, \ldots, \epsilon_\ell$ given in Lemma 5.2, we can compute the output of k-CED-ALG$(X_1, \ldots, X_k, D, 0)$ in time $O(n^{k/2} \cdot N^{k/2+o(k)}/\epsilon^{O(k)})$.*

Proof. Throughout the analysis, we will use the fact that for all m, $1/\epsilon_m \leq \log^{O(\log k)} N/\epsilon$.

We first bound the time spent on base cases. Since each X'_j at the bottom level of recursion has size at most $2N^{1/\log\log N}/\epsilon_\ell^2 = O(N^{o(1)}/\epsilon^2)$ by construction, we can compute each base case's edit tuples and round them in $N^{o(k)}$ time. There are $O(\log d)$ choices of d and $O(\log_{1+\epsilon_\ell}(2\tau_\ell/\epsilon_\ell^2)) = O(\log^{O(1)}(N)/\epsilon^2)$ possible sizes for each choice of d, so there are $O(\log^{O(k)}(N)/\epsilon^{2k})$ different tuples of possible window sizes to consider at this level.

The proof of Lemma 4.3 implies that for any string X generated by an SLP of size n, the number of distinct substrings of length τ is $O(n\tau)$ (in particular, in that proof when $\delta = 1$ we are simply taking every substring into S). Combining these facts, we conclude there are $O(n^k N^{o(k)}/\epsilon^{O(k)})$ distinct base cases, and thus by amortizing the work for base cases, the total time spent on base cases is $O(n^k N^{o(k)}/\epsilon^{O(k)})$.

Besides base-cases, the only work our algorithm does is rounding and convolutions. We can perform the recursion in an amortized fashion. That is, we never make multiple calls to k-CED-Alg on the same k-tuple of strings with the same choice of d. Similarly, for each d and each level $m+1$ call to k-CED-Alg, we only round that call's output's coordinates to the nearest multiple of $\sigma_m(d)$ once. The time spent rounding a set of vectors is proportional to its size, and the final set of vectors that we round was produced by a convolution that took time at least the size of the set of vectors. For this convolution, with amortization we only need to round its output at most $\log N$ times, once per value of d in $\mathcal{D}$. Thus, the time spent on rounding is bounded by the time spent on convolutions times $O(\log N)$.

We now just need to bound the time spent on convolutions. Fix a level m of the recursion and a choice of d in the input. We will bound the total work across all calls at level m and with d as input; there are $O(\log N)$ levels and $O(\log \log N)$ levels, so our final bound on time spent on convolutions will be within logarithmic factors of the bound for one choice of m and d.

The time spent on convolutions in any call is bounded by a constant factor times the time spent on convolutions in the second case in the recurrence relation, i.e., convolutions involving recursive calls. We perform these convolutions on tuples in $\{0, 1, \ldots, 3d\}$ whose coordinates are multiples of $\sigma_m(d)$, i.e., have size at most $O((d/\sigma_m(d))^k) = O((\tau_m/\epsilon_{m+1}\tau_{m+1})^k) = O(N^{o(k)}/\epsilon^k)$. Using FFT, we can thus perform these convolutions in $O(N^{o(k)}/\epsilon^k)$ time (e.g., we could divide all entries by $\sigma_m(d)$, take the convolution, and then multiply by $\sigma_m(d)$). In each call to k-CED-Alg, by the same argument as in Section 4, there are $O((\tau_m/\tau_{m-1}) \cdot (d/\sigma_m(d))^{k-1}) = O((\tau_m/\epsilon_{m+1}\tau_{m+1})^k) = O(N^{o(k)}/\epsilon^{O(k)})$ entries to compute, and for each entry we need to do $O((\log^2 N/\epsilon)^k)$ convolutions. So the time spent on convolutions per call to k-CED-Alg is $N^{o(k)}/\epsilon^{O(k)}$ as well.

We now just need to bound the number of calls made to k-CED-Alg, and our final runtime will be within an $N^{o(k)}/\epsilon^{O(k)}$ factor of this. We will show for each choice of m and d, the number of calls made is $O(n^{k/2} \cdot N^{k/2+o(k)}/\epsilon^{O(k)})$, which gives the desired runtime bound. We bound the number of calls at each level in two ways. The first way is again using the fact that for any string X generated by an SLP of size n, the number of distinct substrings of length τ is $O(n\tau)$, and that at each level there are $O(\log^{O(k)}(N)/\epsilon^{2k})$ tuples of possible lengths for the strings in the input, each at most τ_m/ϵ_m^2. Putting these facts together, there are at most $O(n^k \tau_m^k N^{o(k)}/\epsilon^{O(k)})$ distinct calls to k-CED-Alg at level m with parameter d.

The second way is exactly what we did in Section 4 to bound the number of coordinates in the DP table: For every k-tuple of windows we call k-CED-Alg on at level m with parameter d, the window X_1' ends at an index in X_1 that is a multiple of τ_m, and the other windows end at indices in $X_2, \ldots, X_k$ that are multiples of $\sigma_m(d)$. Furthermore, these entries are distance at most $O(D)$ from the diagonal. So the total number of possible tuples of ending indices for these windows is $O((N/\tau_m) \cdot (D/\sigma_m(d))^{k-1}) = O(N^k/\tau_m^k \epsilon^{O(k)})$. For each tuple of ending indices, there are $N^{o(k)}/\epsilon^{O(k)}$ possible tuples of windows that end at those indices. So we get a bound of $O(N^{k+o(k)}/\tau_m^k \epsilon^{O(k)})$ different calls for each choice of m and d. The desired bound of $O(n^{k/2} \cdot N^{k/2+o(k)}/\epsilon^{O(k)})$ calls follows by taking the geometric mean of the first and second bound, which is at least the smaller of the two. □

6 Lower Bounds

We will start with a summary and overview of the techniques.

6.1 Lower Bound Overview

We will start with the definitions of our hypotheses, then we will describe the results of the lower bound sections.

Hypotheses We use two hypotheses from fine-grained complexity to generate our lower bounds. We use the strong exponential time hypothesis (SETH) and the k-OV hypothesis. Note that SETH implies k-OV [Wil07].

Definition 6.1. *The k-CNF Satisfiability (k-SAT) problem takes as input a formula ϕ with m clauses and n variables. The formula is in conjunctive normal form (CNF) which requires that the formula be the and of m*

clauses. Each clause is the or of at most k variables. Return true if ϕ has a satisfying assignment and false otherwise.

DEFINITION 6.2. (THE STRONG EXPONENTIAL TIME HYPOTHESIS (SETH) [IP01]) *For all constants $\epsilon > 0$ there is some constant k such that k-SAT requires $\omega(2^{n(1-\epsilon)})$ time.*

We can re-frame this as k-SAT requiring $2^{n(1-o(1))}$ time, as long as k is an arbitrarily large constant. Next we define the k-OV problem.

DEFINITION 6.3. (k-OV [Wil07]) *Take as input a list, L, of n zero one vectors of dimension $d = n^{o(1)}$. Return true if there are k vectors $v_i \in L$ for $i \in [1, k]$ such that for all j $v_1[j] \cdot v_2[j] \cdots v_k[j] = 0$.*

The k-OV hypothesis states that for constant k, k-OV requires $n^{k-o(1)}$ time. The k-OV hypothesis is implied by SETH.

We use the k-OV hypothesis to generate our lower bounds. As the k-OV hypothesis is implied by SETH, SETH also implies our lower bounds.

k-LCS lower bound Assuming the well-studied Strong Exponential Time Hypothesis (SETH), in Section 6.2 we show a lower bound for the k-LCS problem in the compressed setting. Intuitively, SETH states that CNF-satisfiability requires $2^{n-o(n)}$ time [IP01]. Even more specifically, we use the k-Orthogonal Vectors problem (k-OV) [Vas18]. At a high level, k-OV takes as input a list L with n zero-one vectors of dimension d. We must return YES if there exist k vectors that, when multiplied element-wise, form the all zeros vector. The k-OV conjecture, which is implied by SETH, states that k-OV cannot be solved in $O(n^{k-\Omega(1)})$ time.

REMINDER OF THEOREM 1.1. *If the k'-OV hypothesis is true for all constants k', then for any constant $\epsilon \in (0, 1]$ grammar-compressed k-LCS requires $\left(M^{k-1}m\right)^{1-o(1)}$ time when the alphabet size is $|\Sigma| = \Theta(k)$ and $m = M^{\epsilon \pm o(1)}$. Here, M denotes the total length of the k input strings and m is their total compressed size.*

Our lower bound relies on two primary tools. First, we use a very compressible representation of a-OV instances. Specifically, given a list L of n zero-one vectors of dimension d, consider a new list $\mathbf{List}(L)_a$ of n^a zero-one vectors of dimension d, with every vector in $\mathbf{List}(L)_a$ representing the element wise multiplication of a vectors from L. Formally, $\mathbf{List}(L)_a$ is indexed by a-tuples of indices from $[1\,..\,n]$, and each vector $\vec{v} = \mathbf{List}(L)_a[j_1][j_2]\cdots[j_a]$ is defined, for every coordinate $i \in [1\,..\,d]$, with:

$$\mathbf{List}(L)_a[j_1][j_2]\cdots[j_a][i] = \vec{v}[i] = L[j_1][i] \cdot L[j_2][i] \cdots L[j_a][i]$$

Notably, $\mathbf{List}(L)_a$ contains an all zeros vector if and only if L is a YES-instance of the a-OV problem.

In the 2-LCS lower bound of [ABBK17], an $(a + 2b)$-OV instance L is first transformed into $A = \mathbf{List}(L)_a$, $B = \mathbf{List}(L)_b$, and $C = \mathbf{List}(L)_b$. Then, the following strings are defined for every $\vec{v_b} \in B$ and $\vec{v_c} \in C$:

$$x_{\vec{v_b}} = \underbrace{\vec{v_{a_1}}[1]\vec{v_b}[1]\vec{v_{a_2}}[1]\vec{v_b}[1]\cdots\vec{v_{a_{n^a}}}[1]\vec{v_b}[1]}_{\text{first bit}}\cdots\underbrace{\vec{v_{a_1}}[d]\vec{v_b}[d]\vec{v_{a_2}}[d]\vec{v_b}[d]\cdots\vec{v_{a_{n^a}}}[d]\vec{v_b}[d]}_{d\text{th bit}},$$

$$y_{\vec{v_c}} = \vec{v_c}[1]\underbrace{000000}_{n^a-1}\vec{v_c}[2]\cdots\underbrace{000000}_{n^a-1}\vec{v_c}[d].$$

The string $x_{\vec{v_b}}$ that interleaves $\vec{v_b}$ with bits of n^a vectors $\vec{v_{a_i}} \in A$, referred to as "interleaved" representation, is highly compressible, to an SLP of size $O(nd)$. Moreover, if there exists a vector $\vec{v_{a_i}} \in A$ such that $(\vec{v_{a_i}}, \vec{v_b}, \vec{v_c})$ is orthogonal, Abboud et al. [ABBK17] show (using the structural alignment gadget of [BK15]) how to perfectly align $(\vec{v_{a_i}}[l], \vec{v_b}[l], \vec{v_c}[l])$ for all $l \in [1\,..\,d]$. Finally, the gadgets $x_{\vec{v_b}}$ for all $\vec{v_b} \in B$ are concatenated with extra padding to generate X_B, and the gadgets $y_{\vec{v_c}}$ for all $\vec{v_c} \in C$ are concatenated with extra padding to generate Y_C. This leads to the $(Mm)^{1-o(1)}$ lower bound since the uncompressed and compressed lengths of X_B and Y_C are (roughly) $O(n^{a+b})$ and $O(n^b)$, respectively, and we are solving an $(a + 2b)$-OV instance.

We may extend the above construction to the compressed k-LCS setting by transforming an $(a + kb)$-OV instance L into lists $A = \mathbf{List}(L)_a$, $B = \mathbf{List}(L)_b$, and $C_h = \mathbf{List}(L)_b$ for $h \in [1\,..\,k-1]$. We then create X_B and Y_{C_h} for $h \in [1\,..\,k-1]$. Since the strings Y_{C_h} are zero-padded, we can easily adapt the same structural

alignment gadget of 2-LCS from [BK15] to ensure a perfect alignment. However, this only leads to a lower bound of $(m^{k-1}M)^{1-o(1)}$ since the uncompressed and compressed lengths of the strings remain (roughly) $O(n^{a+b})$ and $O(n^b)$, respectively, and we are solving an $(a+kb)$-OV instance: $Mm^{k-1} = O(n^{a+kb})$. To get a much stronger lower bound of $(mM^{k-1})^{1-o(1)}$, we need to solve a much higher OV instance. In particular, we will solve an $(a(k-1)+kb)$-OV instance by taking $A_h = \mathbf{List}(L)_a$, $B_h = \mathbf{List}(L)_b$ for $h \in [1\,..\,k-1]$, and $C = \mathbf{List}(L)_b$. We then create strings X_{B_h} from A_h and B_h for each $h \in [1\,..\,k-1]$, and Y_C. That is, we now have $(k-1)$ interleaved strings and only one zero-padded string. This makes generalizing the structural alignment gadget substantially more intricate since we may have to deal with $k-1$ different offsets. In fact, without any zero-padded string, we are not able to show any perfect alignment gadget. Because we are now solving an $(a(k-1)+kb)$-OV instance, we get our desired lower bound by noting $M^{k-1}m = O(n^{a(k-1)+kb})$.

Easy k-Median Edit Distance lower bounds via LCS reduction As a first lower bound for edit distance, we can reduce from LCS to both median k-edit distance and center k-edit distance. Suppose, we are given a k-LCS instance with strings $S_1, \ldots, S_k$ all of length M and let γ denote the empty string. It can be shown that

$$\delta_E(S_1, \ldots, S_k, \underbrace{\gamma, \ldots, \gamma}_{(k-1)}) = Mk - \mathrm{LCS}(S_1, \ldots, S_k).$$

This increases k since we add $(k-1)$ empty strings, but it does not increase the size of the problem, or the compression size. Using the above relation, we can prove the following theorem.

Reminder of Theorem 6.6. *Given an instance of k-median edit distance on strings of lengths $M_1 \le M_2 \le \cdots \le M_k$ where these strings can all be compressed into a SLP of size $m = |\sum_i M_i|^{\delta \pm o(1)}$ for any constant $\delta \in (0,1]$. Then, an algorithm for k-median edit distance that runs in $((M_2+1)\cdots(M_k+1)\cdot m)^{1-\epsilon}$ time for constant $\epsilon > 0$ violates SETH.*

We can get a similar lower bound for center k-edit distance from k-LCS by adding a single empty string.

Theorem 6.1. *Given an instance of k-center edit distance on strings of lengths $M_1 \le M_2 \le \cdots \le M_k$ where these strings can all be compressed into a SLP of size $m = |\sum_i M_i|^{\delta \pm o(1)}$ for any constant $\delta \in (0,1]$, then, an algorithm for k-center edit distance that runs in time $((M_2+1)\cdots(M_k+1)\cdot m)^{1-\epsilon}$ time for constant $\epsilon > 0$ violates SETH.*

These reductions are convenient for propagating results from k-LCS to k-Edit Distance generically. However, because they add empty strings, they don't prove hardness for some of the most commonly studied cases such as where all strings are of the same length and for median k-edit distance with even k. To get lower bounds for all k and when all strings are of the same length, we use a reduction directly from SETH, instead of going through k-LCS.

Stronger k-Median Edit Distance Lower Bounds directly from SETH We get a lower bound for median k-edit distance and center k-edit distance over compressed strings from SETH. When $k=2$ this resolves the second open problem suggested by Abboud et al [ABBK17]. We also generalize the lower bound for all $k \ge 2$. There are many difficulties introduced by trying to get lower bounds for median k-edit distance when $k \ge 2$. We can use some of the ideas from the k-LCS reduction. Specifically, the notion of the compressed interleaved strings remains. Notably, we need to allow any choice of $\Delta_1, \ldots, \Delta_{k-1}$ offsets; however, if these offsets are more similar we have many characters that match on all but one string. For k-LCS we still need to delete these characters, but, in median k-edit distance we can simply insert a character in one string. This creates an artificial pressure to make all the Δ_i values the same. To overcome this, we can use some of the ideas from the recent paper that gives lower bounds for the uncompressed case for k-edit distance [HBGT20]. There is still an issue, they build their alignment gadget with the crucial use of empty 'fake gadgets'. However, we need to guarantee that $\Delta_i \in [0, n^a - 1]$, and these fake gadgets allow for values of Δ_i outside of this range. To overcome this we incentivize a match up of the real gadgets, which then forces a restriction on valid Δ_i values.

Specifically, we need to add a gadget, which we call a selector gadget. This gadget causes characters lined up inside it to have a low edit distance if they all match, and otherwise have a higher edit distance that is unchanged by exactly how well they match. The selector gadget looks like this: $SCSG_i(c) = \%^{ix} c^y \%^{(k-i)x}$. We

have gadgets $SCSG_1(c_1), \ldots, SCSG_k(c_k)$ such that we can either try to match the characters c_i, or try to line up the % characters. If we line up the % characters, the edit distance is ky. If we line up the c_i characters and they all match ($c_i = c_j \ \forall i, j$), the edit distance is $xk^2/4$ if k is even and $x(k^2-1)/4$ if k is odd. If the characters don't match the edit distance is at least $xk^2/4 + y$ if k is even and $x(k^2-1)/4 + y$ if k is odd. Consider the case of k even, we can choose integer values of x and y such that $xk^2/4 < yk \leq xk^2/4 + y$. By doing so, if all the characters match, then the median k-edit distance is $xk^2/4$, otherwise it is yk. In some sense this gadget is causing characters to act like they do in k-LCS, where only a match across all strings gives us a benefit. Using these selector gadgets and ideas from the edit distance and LCS lower bounds, we get a lower bounds for both median k-edit distance and center k-edit distance from SETH.

THEOREM 6.2. *If the k'-OV hypothesis is true for all constants k', then for all constant $\epsilon \in (0, 1]$ grammar-compressed k-median edit distance requires $\left(M^{k-1}m\right)^{1-o(1)}$ time when the alphabet size is $|\Sigma| = \Theta(k)$ and $m = M^{\epsilon \pm o(1)}$. Here, M and m denote the total uncompressed and compressed length of the k input strings respectively.*

The lower bound for median k-edit distance immediately implies a lower bound for center k-edit distance following [HBGT20].

REMINDER OF THEOREM 6.8 . *We are given k strings of length M with a SLP of size m. The center k-edit distance problem on these strings requires $\left(M^{k-1}m\right)^{1-o(1)}$ time if SETH is true.*

Given these lower bounds for the case of compressed k-LCS, median k-edit distance and center k-edit distance, we want to consider not just compression but also approximation.

6.2 Lower Bound with LCS In this section we will argue that if we have k strings each of length M and they have a SLP compression of size m then the problem requires $M^{k-1-o(1)}m^{1-o(1)}$ if SETH is true. In the next section we use these hardness results for k-LCS to prove hardness for k'-Edit Distance.

The core of this section is building a generalized "perfect alignment" gadget. This is a gadget that causes substrings to be aligned with no skips or merges. We use this generalized alignment gadget to generalize the work of [ABBK17]. The main idea for this perfect alignment gadget is that between every string we want to align, we add symbols $\$_1\$_2 \ldots \$_k$. Additionally, at the end of each string S_i in our gadget, we add many copies of these characters, excepting $\$_i$. That is, we add $\$_1 \ldots \$_{i-1}\$_{i+1} \ldots \$_k$. Via this construction, any valid perfect alignment will match all available copies of $\$_i$ for all i. Any alignment that isn't perfect (for example it skips matching some sub-string in the middle of S_i) will miss out on one of these $\$_i$ characters in S_i, thus lowering the value of a potential k-LCS.

Recall that $\text{LCS}(S_1, \ldots, S_k)$ is a function that returns the k-LCS of the strings $S_1, \ldots, S_k$. Recall that $\delta_D(S_1, \ldots, S_k) = \sum_{i \in [1,k]}(S_i - \text{LCS}(S_1, \ldots, S_k))$. That is, the count of all unmatched characters.

6.3 Representations of Many Lists at Once The key idea is going to be different ways to represent many lists of OV instances at once. This representation comes from [ABBK17].

DEFINITION 6.4. *Let L be the list of vectors to a k-OV instance. Let $|L| = n$.*

The list representation of ℓ copies of L is made up of n^ℓ vectors $\vec{v} = \mathbf{List}_\ell(L)[j_1][j_2]...[j_\ell]$.

$$\mathbf{List}(L)_\ell[j_1][j_2]...[j_\ell][i] = \vec{v}[i] = L[j_1][i] \cdot L[j_2][i] \cdots L[j_\ell][i]$$

As a convenience of notation we will allow indexing with a single index into $\mathbf{List}(L)$:

$$\mathbf{List}(L)_\ell \left[\sum_{i=1}^{\ell} j_i n^{i-1} \right] = \mathbf{List}(L)_\ell[j_1][j_2]...[j_\ell]$$

When writing down this list of vectors into a string there are two ways to do it. The serial way of writing out each vector in order, or the interleaving way. The serial way of writing vectors is in many ways easier to use for gadgets. However, the interleaved version is easier to compress. We will describe both and use both in our gadgets.

DEFINITION 6.5. *Let L be the list of vectors to a k-OV instance. Let $|L| = n$.*

We define the serial version as:

$$\mathbf{String}_{\mathbf{B}\ell}(L) = \bigcirc_{j_1 \in [1,n] \ldots, j_\ell \in [1,n]} \left(\bigcirc_{i=1}^{d} L[j_1][i] \cdot L[j_2][i] \cdots L[j_\ell][i] \right).$$

Note that this is equivalent to

$$\mathbf{String}_{\mathbf{B}\ell}(L) = \bigcirc_{j \in [1,n^\ell]} \bigcirc_{i \in [1,d]} \mathbf{List}(L)[j][i].$$

We define the interleaving version as:

$$\mathbf{String}_{\mathbf{I}\ell}(L) = \bigcirc_{i=1}^{d} \left(\bigcirc_{j_1 \in [1,n] \ldots, j_\ell \in [1,n]} L[j_1][i] \cdot L[j_2][i] \cdots L[j_\ell][i] \right).$$

Note that this is equivalent to

$$\mathbf{String}_{\mathbf{I}\ell}(L) = \bigcirc_{i \in [1,d]} \bigcirc_{j \in [1,n^\ell]} \mathbf{List}(L)[j][i].$$

So the difference between these versions is really just what order we represent the vectors. But crucially if there is a particular vector in $\mathbf{List}(L)$ that is of interest, this will appear in different places. In $\mathbf{String}_{\mathbf{B}\ell}(L)$ a vector $\vec{v} = \mathbf{List}(L)[i]$ appears as bits $[i \cdot d, (i+1) \cdot d - 1]$. Where as in $\mathbf{String}_{\mathbf{I}\ell}(L)$ the vector $\vec{v} = \mathbf{List}(L)[i]$ appears as bits $i, i + n^k, \ldots, i + (d-1)n^k$.

We give one final version that merges a single vector with the interleaved representation.

DEFINITION 6.6. *We will expand the previous definition of an interleaved string to allow a merge with a single other vector. Recall that*

$$\mathbf{String}_{\mathbf{I}\ell}(L) = \bigcirc_{i \in [1,d]} \bigcirc_{j \in [1,n^\ell]} \mathbf{List}(L)[j][i].$$

Recall that for a vector $u = \mathbf{List}(L)[j]$ it is represented in bits $j, j + n^k, \ldots, j + (d-1)n^k$ in $\mathbf{String}_{\mathbf{I}\ell}(L)$.

We will define

$$\mathbf{VecS}_{\mathbf{I}\ell}(L, v) = \bigcirc_{i \in [1,d]} \bigcirc_{j \in [1,n^\ell]} \mathbf{List}(L)[j][i]v[i].$$

Note that now if we take bits $j, j+n^\ell, \ldots, j+(d-1)n^\ell$ we give a vector w such that $w[i] = u[i]v[i]$ where $u = \mathbf{List}[j]$.

6.4 Intuition for our Reduction We will describe at a high level the reduction of [ABBK17] and the idea for generalizing it. In this section we will informally explain how to use the serial and interleaved representations of the vectors to build a reduction from SETH to compressed k-LCS. We hope to build understanding for what the different levels of alignment gadgets are doing through small examples and intuition.

6.4.1 Why We Care About Lining up the Strings Lets say we have a representation $\mathbf{String}_{\mathbf{I}\ell}(L)$ and we have a single vector, v of length d. We create a new vector $\hat{v}$ where $\hat{v}[i \cdot n^\ell] = v[i]$ and otherwise $\hat{v}$ is zero. $\hat{v}$ will have length $n^\ell(d-1)+1$.

Now we will note the following: the locations of the bits in $\hat{v}$ have exactly the offsets that single vectors do in $\mathbf{String}_{\mathbf{I}\ell}(L)$! So, if we consider sub-string $\mathbf{String}_{\mathbf{I}\ell}(L)[i, i+n^\ell(d-1)]$ then v forms an orthogonal $\ell+1$ tuple with the vectors represented by $\mathbf{List}(L)_\ell[i]$ if $\hat{v}$ is orthogonal to $\mathbf{String}_{\mathbf{I}\ell}(L)[i, i+n^\ell(d-1)]$. This is why we care about offsets. The next few subsections will simply be building the gadgets necessary to get this "perfect alignemnt" and the gadgets needed to represent k-OV coordinates in the edit distance setting.

6.4.2 The Case of LCS With Two Strings How did all of this work in [ABBK17]? Start with k-OV. Now consider a k_1 and k_2 that have this property: $k = k_1 + 2k_2$. They then create three sets: A represents k_1 vectors at once, B represents k_2 vectors at once and C represents k_2 vectors at once. We will give a text explanation and then give a small example.

For C they create its string S_C by taking $\mathbf{List}(L)_{k_2}$ and making $Y[i] = \mathbf{List}'_{k_2}(L)[i]$. That is they pad the vector with $n^{k_1} - 1$ zeros after each entry in the original vector.

The $\mathbf{String}_{\mathbf{I}k_1}(L)$ representation of A and the zeros are all very compressible with straight line programs. For B they create its string S_B by basically merging each vector $b \in \mathbf{List}_{k_2}(L)$ with A which is structured like $\mathbf{String}_{\mathbf{I}k_1}(L)$.

So, while the length of each string is $n^{k_2+k_1}$ the compressions are of size n^{k+1}. We need a gadget that forces our representation to align the two strings with no gaps in the LCS. If we do so, we can then check if an OV exists.

Now let us work through a small example. Let $k_1 = 2$ and $k_2 = 1$.

$$v_1 = < 0, 1, 1, 1 > \tag{6.2}$$

$$v_2 = < 1, 1, 0, 1 > \tag{6.3}$$

$$v_3 = < 1, 0, 1, 1 > \tag{6.4}$$

$$v_4 = < 0, 1, 1, 0 > \tag{6.5}$$

$$L = \{v_1, v_2, v_3, v_4\} \tag{6.6}$$

For both B and C we form lists that are concatenations of vectors.

$$B = C = v_1, v_2, v_3, v_4$$

For A we first we want to generate all the vectors $v_{i,j}[p] = v_i[p] \cdot v_j[p]$.

$$v_{1,2} = < 0, 1, 0, 1 > \quad v_{1,3} = < 0, 0, 1, 1 > \quad v_{1,4} = < 0, 1, 0, 0 > \tag{6.7}$$

$$v_{2,3} = < 1, 0, 0, 1 > \quad v_{2,4} = < 0, 1, 0, 0 > \tag{6.8}$$

$$v_{3,4} = < 0, 0, 1, 0 > \tag{6.9}$$

Then we form A by taking the first bit of each of these vectors then the second bits, etc. For this example, we put a semicolon in between the first bits and second bits. We do this here for making it easier to read.

$$A = 0, 0, 0, 1, 0, 0; 1, 0, 1, 0, 1, 0; 0, 1, 0, 0, 0, 1; 1, 1, 0, 1, 0, 0$$

Note that if we take the bits $p, p+6, p+12, p+18$ these correspond to a single vector $v_{i,j}$. We want the ability to merge A and a single vector v_i. For this, if there is $v_i[p] = 0$ then we replace all those bits with zeros, otherwise we leave the bits of A as is.

$$(A\&v_1) = 0, 0, 0, 0, 0, 0; 1, 0, 1, 0, 1, 0; 0, 1, 0, 0, 0, 1; 1, 1, 0, 1, 0, 0 \tag{6.10}$$

$$(A\&v_2) = 0, 0, 0, 1, 0, 0; 1, 0, 1, 0, 1, 0; 0, 0, 0, 0, 0, 0; 1, 1, 0, 1, 0, 0 \tag{6.11}$$

$$(A\&v_3) = 0, 0, 0, 1, 0, 0; 0, 0, 0, 0, 0, 0; 0, 1, 0, 0, 0, 1; 1, 1, 0, 1, 0, 0 \tag{6.12}$$

$$(A\&v_4) = 0, 0, 0, 0, 0, 0; 1, 0, 1, 0, 1, 0; 0, 1, 0, 0, 0, 1; 0, 0, 0, 0, 0, 0 \tag{6.13}$$

We also want to generate the padded vectors for S_C. These padded vectors have 'real' vector values at locations $0, 6, 12, 18$. We want this because it means that if we line up one of these padded vectors, $(0\&v_i)$, against a vector mixed with A, $(A\&v_j)$, the 'real' values correspond to a vector in A.

$$(0\&v_1) = 0, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 1 \tag{6.14}$$

$$(0\&v_2) = 1, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0; 1 \tag{6.15}$$

$$(0\&v_3) = 1, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 1 \tag{6.16}$$

$$(0\&v_4) = 0, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 0 \tag{6.17}$$

Specifically, if we line up $(0\&v_i)$ and $(A\&v_j)$ with an offset of Δ every lined up set of entries has a zero if there are $k_1 + 2k_2$ vectors that are orthogonal. Lets consider this example:

$$(A\&v_1) = 0, 0, 0, 0, 0, 0; 1, 0, 1, 0, 1, 0; 0, 1, 0, 0, 0, 1; 1, 1, 0, 1, 0, 0 \tag{6.18}$$

$$(0\&v_2) = \quad 1, 0, 0, 0, 0, 0; 1, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0; 1 \tag{6.19}$$

$$\text{This alignment} = \quad 0; 0, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0 \tag{6.20}$$

By picking this alignment, $\Delta = 5$, we are picking the sixth vector that went into A which is $v_{3,4}$. So, this alignment is checking the orthogonality of v_1, v_2, v_3, v_4. Lets look at a set of non-orthogonal vectors to compare. The vectors

v_1, v_2, v_1, v_4 are not orthogonal. The vector $v_{1,4}$ corresponds to $\Delta = 2$.

$$(A\&v_1) = 0,0,0,0,0,0;1,0,1,0,1,0;0,1,0,0,0,1;1,1,0,1,0,0 \tag{6.21}$$

$$(0\&v_2) = \quad 1,0,0,0,0,0;1,0,0,0,0,0,0;0,0,0,0,0,0;1 \tag{6.22}$$

$$\text{This alignment} = \quad 0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0 \tag{6.23}$$

This is why we want 'perfect alignments'. We want to build up representations of these strings and allow for any choice of Δ, but no skipping characters or merging gadgets. Previous work generates a perfect alignment gadget such that the output gadgeted strings $T_{(A\&v_i)}$ and $T_{(0\&v_j)}$ have a low LCS if there is a Δ such that $(A\&v_i)$ and $(0\&v_j)$ have an all zeros alignment with offset Δ (like $(A\&v_1)$ and $(0\&v_2)$ with $\Delta = 5$ in our example). We will now show what the perfect alignment gadget looks like. We can not use zeros and ones directly to solve OV (see Lemma 6.1). We want gadgets such that $0,0$ and $0,1$ have a low LCS and higher LCS for $1,1$. We add the characters \$ and 5. The 5 characters make sure we don't skip any symbols from $T_{(0\&v_i)}$. The \$ characters make sure we don't skip any symbols from $T_{(A\&v_j)}$.

$$T_{(A\&v_1)} \approx \$05\$05\$05\$05\$05\$05\$15\$05\$15\$05\$15\$05\$05\$15\$05\$05\$05\$15\$15\$15\$05\$15\$05\$0\$ \tag{6.24}$$

$$T_{(0\&v_2)} \approx \$ \quad \$ \quad \$ \quad \$ \quad \$ \quad \$15\$05\$05\$05\$05\$15\$05\$05\$05\$05\$05\$05\$05\$05\$05\$05\$1\$^6 \tag{6.25}$$

The extra dollar signs at the ends of the strings allow all the dollar signs in $T_{(A\&v_1)}$ can be matched regardless of the offset Δ. The 5 symbols make skipping zero or one characters also skip at least one 5 character. This set of characters (at a high level) form the perfect alignment gadget of [ABBK17].

To form the string S_B we want to basically concatenate $T_{(A\&v_1)}, T_{(A\&v_2)}, T_{(A\&v_3)}, T_{(A\&v_4)}$. To form the string S_C we will basically concatenate $T_{(0\&v_1)}, T_{(0\&v_2)}, T_{(0\&v_3)}, T_{(0\&v_4)}$. These strings are not really concatenated, but instead have an alignment gadget wrapped around them. This alignment gadget guarantees a low LCS if a pair of strings $T_{(A\&v_i)}$ and $T_{(0\&v_j)}$ have a low LCS, and otherwise has a high LCS. In total, this means the strings S_B and S_C have low LCS if there exist i, j, Δ $(A\&v_i)$ and $(0\&v_j)$ have an all zeros alignment with offset Δ. Such a zero alignment existing implies a $(k_1 + 2k_2)$-OV exists (a 4-OV in our example).

6.4.3 How to Generalize This (Intuition) What we want generically for k-LCS is to have k sets of lists that act like B and C, and ℓ sets of lists that act like A. If we make an efficient reduction with these parameters, then we get a lower bound of $(M^\ell m^{k-\ell})^{1-o(1)}$.

To get the easy generalization we set $\ell = 1$, and we have one "A type" set of lists. Lets call the "B" and "C" type lists $B_1, \ldots, B_k$. We create strings $S_1, \ldots, S_k$ and merge B_1 and A into S_1 using the method from [ABBK17]. For S_i where $i > 1$ we instead use the padding with zeros method. Now we have a situation where we want a gadget that forces the zero padded strings to line up exactly and they are both on some offset of i from the strings in S_1. This as it turns out is easy. The strings are of the same length and just copying the construction used for C in [ABBK17] will get us what we want here. Specifically, the string S_1 will be roughly[2] a concatenation of $T_{(A\&v_i)}$ strings. The S_i strings for $i > 1$ are instead roughly[2] the concatenation of $T_{(0\&v_i)}$ strings. If we are given a set of k strings $T_{(A\&v_1)}, T_{(0\&v_2)}, \ldots, T_{(0\&v_k)}$ we want to allow any offset Δ, but that same Δ should be shared across all the $T_{(0\&v_i)}$ strings. As a result, we can just use the same $T_{(0\&v_i)}$ strings from the two string case for all the strings $i > 1$. The $T_{(0\&v_i)}$ strings are the same in every location except for the d representations of the bits in the vector v_i. The structure of the 5 characters forces all of the $T_{(0\&v_i)}$ strings to line up together to match all the 5 characters. The \$ characters force any high LCS to not skip any of the zeros or ones in the $T_{(A\&v_i)}$ representation of $(A\&v_i)$. On a high level this reduction is easy because we still have only one offset Δ that we need to deal with.

What needs to happen if $\ell > 1$? The primary hurdle is coming up with a setup where two long strings of the B type from the original construction can be forced to have their optimal setup line them up exactly with no skips when they have two different offsets from the zero padded strings. To get a sense of the difficulty consider how many \$ characters should be at the start and end of those strings to allow all \$ characters to be matched regardless of offsets. As we grow the number on one string we have to grow the number on the other. So we need different symbols $\$_1, \ldots, \$_k$ for each string.

[2]Once again, it isn't really a concatenation. Instead these strings are wrapped in an alignment gadget. However, these alignment gadgets are basically concatenations of the strings but with characters in-between the strings.

For convenience let 0 and 1 be stand-ins for the strings we use in LCS reductions from OV (there are longer strings that have the property we want where the LCS of $110, 101, 011, 100, 010, \ldots, 000$ are all equal). Now, if we want to compare k strings where $X \in \{0,1\}^m$ and $Y_1, \ldots, Y_{k-1} \in \{0,1\}^n$ where $n < m$ and we want the Ys to line up exactly and we want them to compare to some substring of X then we can add a special character \$. Let $Z = \$^c$ where c will be a constant in terms of k that is larger than the full length of string representations of 0 and 1.

$$S_X = ZX[0]ZX[1]Z \ldots ZX[m]Z \tag{6.26}$$

$$S_{Y_i} = Z^{m-n}Y_i[0]ZY_i[1]Z \ldots ZY_i[m]Z^{m-n} \tag{6.27}$$

Now, if there is a sub-string of X that is orthogonal to $Y_1, \ldots, Y_{k-1}$ then the optimal LCS will match all the Zs in X and match each character $Y_i[j]$ with some character in X. If we spread out our matches of Y_i and don't match to a sub-string of X but instead to a subsequence, we will miss out on some Z characters. So, if there isn't a match that corresponds to an OV we will lose out. And this works with many strings at once.

This gadget is forcing not just any alignment of the underlying strings in $X_1, \ldots, X_{k-1}, Y$, but a *perfect* alignment. We will use this structure to build a perfect alignment gadget.

6.5 Reduction We will prove lemmas building up the gadgets for this construction. We will describe the details of our gadgets and reductions here. The intuition described above of both why we care about perfect alignment and how to achieve it is used in the next subsection on our alignment gadget.

6.5.1 Alignment Gadget Now we will prove that the alignment gadget works as desired. First let us define what an alignment and perfect alignment are.

Definition 6.7. *We will generalize the Structured Alignment Cost definition of previous work [ABBK17]. We are given as input k lists of strings $X_1, \ldots, X_{k-1}, Y$. Where $|X_i| = n$, $|Y| = m$, and $m < n$.*

An alignment, Λ is a list of t k-tuples:

$$((i_{1,1}, i_{1,2}, \ldots, i_{1,k}), \ldots, (i_{t,1}, i_{t,2}, \ldots, i_{t,k}))$$

where $i_{j,p} < i_{j',p}$ if $j < j'$. We call an alignment perfect if $i_{j,p} + 1 = i_{j+1,p}$ and $t = m$.

Now we will create some gadgets to maintain alignment. We will define them here and then below prove the various properties we care about.

Definition 6.8. (Perfect Alignment Gadget) *We are given as input k lists of strings $X_1, \ldots, X_{k-1}, Y$. Where $|X_i| = n$, $|Y| = m$, and $m < n$.*

We will add k new symbols $\$_1, \ldots, \$_k$. We will define $A = \$_1^{2\ell} \circ \ldots \circ \$_k^{2\ell}$ where $\circ$ is the concatenation operator. Let $A_{-i} = \$_1^{2\ell} \circ \ldots \circ \$_{i-1}^{2\ell} \circ \$_{i+1}^{2\ell} \circ \ldots \circ \$_k^{2\ell}$. Note that this is just A with all the $\$_i$ symbols removed. We also add a character % which we use to pad out our strings to give them more value. Let $B = \%^{2\ell}$ (note that the % character is serving the purpose of the 5 character in our earlier example). We define $f = n - m$. Then the generalized structured alignment gadget would produce strings:

$$AG_i(X_i) = A_{-i}^f \circ A \circ X_i[1] \circ B \circ A \circ X_i[2] \circ B \circ \ldots \circ A \circ X_i[n] \circ B \circ A_{-i}^f \tag{6.28}$$

$$AG_y(Y) = A_{-k}^f \circ A \circ Y[1] \circ B \circ A \circ Y[2] \circ B \circ \ldots \circ A \circ Y_m \circ B \circ A_{-k}^f \tag{6.29}$$

We also want gadgets to be a selector around the alignment gadget. We add a new character @. We will leave D unset for now. We define our SAG gadgets:

$$SAG_i(X_i) = @^{D-1}AG_i(X_i)$$

$$SAG_y(Y) = AG_y(Y)@^{D-1}.$$

We will now prove that these work as perfect alignment gadgets. This setup requires that we are trying to detect if there is a perfect alignment in which all strings match as much as they can.

THEOREM 6.3. *We are given as input k lists of strings $X_1, \ldots, X_{k-1}, Y$. Where $|X_i| = n$, $|Y| = m$, and $m < n$. Furthermore all strings $S \in X_i$ and $S' \in Y$ have length $|S| = |S'| = \ell$.*

Additionally, given any set of k strings $S_i \in X_i$ and $S_y \in Y$ the LCS distance is either z or $z+1$ for some constant z.

Let Λ be a perfect alignment which is a list of m k-tuples: $(i_{1,1}, i_{1,2}, \ldots, i_{1,k})$. Where $i_{j,h} + 1 = i_{j+1,h}$.

Then, if there is a perfect alignment Λ in which there are exactly m k-tuples such that

$$\text{LCS}(X_1[i_{j,1}], \ldots, X_{k-1}[i_{j,k-1}], X_y[i_{j,k}]) = z + 1,$$

then

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D,$$

if all perfect alignments have less than m k-tuples with a LCS *of $z+1$ then*

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D - 1.$$

These strings use an additional alphabet of size $O(k)$. The total length of strings is $O(\ell n)$. The value of is $D = 2\ell(2m + (k-1)n) + (z+1)m$.

Proof. In the SAG gadgets note that if we match any @ character we have a maximum LCS of $D-1$, and we can always achieve this. If we match any characters from AG_i, then we can match no @ symbols. Thus, what remains to be proven is that if there is a perfect alignment Λ in which there are m k-tuples such that if we have m matches where

$$\text{LCS}(X_1[i_{j,1}], \ldots, X_{k-1}[i_{j,k-1}], X_y[i_{j,k}]) = z + 1,$$

then

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), SAG_y(Y)) = D,$$

otherwise

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), SAG_y(Y)) \leq D - 1.$$

Recall $D = 2\ell(2m + (k-1)n) + (z+1)m$. Now we have three cases to argue.

Case 1 [There are m "good" k-tuples, lower bound]: Consider aligning the strings in the perfect alignment Λ which has m k-tuples of strings which have a LCS of $z+1$. Now, we can match m copies of B. What about $\$_i$ symbols? There are $2\ell n$ copies of the $\$_i$ symbol in $AG_i(X_i)$, they only appear in the copies of A (they don't appear in A_{-i}). If we are matching up with a perfect alignment we can match all $2\ell n$ of these symbols. They either line up with copies of A in other strings, or copies of A_{-j}. Finally, there are $2\ell m$ copies of $\$_y$ in $AG_y(Y)$. In a perfect alignment these symbols will all get matched to symbols that appear in copies of A in other strings. So, in total

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), AG_y(Y)) \geq 2\ell m + 2\ell(m + (k-1)n) + (z+1)m \geq D.$$

Case 2 [There are m "good" k-tuples, upper bound]: Across all % and $\$_i$ symbols the maximum number of matches is $2\ell(m + m + n(k-1))$. What if we don't match the strings in a perfect alignment manner? There are two ways to do this. One is to skip matching some strings in Y (e.g. merging $Y[j]$ and $Y[j+1]$ and matching that to some single string somewhere else, or simply skipping over a string in Y). If this happens we miss out on the characters in at least one B. The advantage gleaned for every skipped string in Y is at most $|Y[j]| = \ell$, but skipping out on B is worse, we lose 2ℓ matches. The next case is skipping strings in X_i. That is, matching $Y[j]$ with $X_i[j']$ but matching $Y[j]$ with $X_i[j' + 1 + \Delta]$ for some $\Delta \geq 1$. This looses at least 2ℓ $\$_i$ characters. Any k-tuple of strings has, by assumption in the lemma, a k-LCS of at most $z+1$. So, this causes the k-LCS less than D. Finally, any time we match multiple strings in X_i with a single string in Y, $Y[j]$, can increase the match in $Y[j]$ by at most ℓ. However, we loose at least 2ℓ symbols $\$_i$ in X_i. This means the k-LCS of AG_i strings is at most $D - \ell$.

Case 3 [There are less than m "good" k-tuples]: Now, if there are less than m matches with a k-LCS of $z+1$ then what is the maximum k-LCS? Similarly to case 2, if we skip as string in Y or merge $Y[j]$ and $Y[j+1]$ we loose at least one B. Because the B copies are in-between every pair of adjacent Y strings. If we matching some symbol in each $Y[j]$, then the maximum value if we match one string from each X_i to each string in Y is $D-2\ell$ because we must skip some 2ℓ characters $\$_i$, so the maximum match would be at most $D-2\ell$. Finally, we could merge multiple strings from X_i and match with a single string from Y, in this setting we could potentially get $D-2\ell+\ell$. While we can potentially match all ℓ characters in Y, we must miss out on at least 2ℓ $\$_i$ characters.

So, we have proven the result that if we have m matches where

$$\text{LCS}(X_1[i_{j,1}], \ldots, X_{k-1}[i_{j,k-1}], X_y[i_{j,k}]) = z+1,$$

then

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), SAG_y(Y)) = D,$$

otherwise

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), AG_y(Y)) \leq D-1.$$

Then, because of the @ symbols if there is a perfect alignment Λ in which there are exactly m k-tuples such that

$$\text{LCS}(X_1[i_{j,1}], \ldots, X_{k-1}[i_{j,k-1}], X_y[i_{j,k}]) = z+1,$$

then

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D,$$

if all perfect alignments have less than m k-tuples then

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D-1.$$

Proving our desired result.

□

Now, we will also want to use this gadget for regular alignment. In this case we will care about distinguishing between a single k-tuple with high value versus no strings having high value.

THEOREM 6.4. *We are given as input k lists of strings $X_1, \ldots, X_{k-1}, Y$. Where $|X_i| = n$, $|Y| = m$, and $m < n$. Furthermore all strings $S \in X_i$ and $S' \in Y$ have length $|S| = |S'| = \ell$.*

Additionally, given any set of k strings $S_i \in X_i$ and $S_y \in Y$ the LCS distance is either z or $z+1$ for some constant z.

Finally define $\hat{X}_i$ as a list that is simply two copies of X_i. That is $\hat{X}_i[j] = \hat{X}_i[n+j] = X_i[j]$.

In the first case there is exactly one k-tuple where

$$\text{LCS}(X_1[i_{1,1}], \ldots, X_{k-1}[i_{1,k-1}], X_y[i_{1,k}]) = z+1,$$

and there exists a perfect alignment that can align this k-tuple. That is $n - i_{1,j} \geq m - i_{1,k}$ and $i_{1,j} \leq i_{1,k}$. In this first case we want:

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D.$$

In the second case there are zero k-tuples that have an LCS of $z+1$ then we want:

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D-1.$$

These strings use an additional alphabet of size $O(k)$. The total length of strings is $O(\ell n)$. The value of is $D = 2\ell(2m + (k-1)n) + zm + 1$.

Additionally, let c be the size of an SLP that gives a single variable for all $(k-1)n+m$ strings $X_i[j]$ and $Y[j]$. Then there is a SLP representation of all the strings $AG_1(X_1), \ldots, AG_{k-1}(X_{k-1}), AG_y(Y)$ of size $O(c + \lg(\ell)k + \lg(n) + kn + m)$.

Proof. As in the above theorem: in the SAG gadgets note that if we match any @ character we have a maximum LCS of $D-1$, and we can always achieve this. If we match any characters from AG_i, then we can match no @ symbols. Thus, what remains to be proven is that if there is an alignment with exactly one k-tuple such that

$$\text{LCS}(X_1[i_{j,1}], \ldots, X_{k-1}[i_{j,k-1}], X_y[i_{j,k}]) = z+1,$$

then

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), SAG_y(Y)) = D,$$

if there are no k-tuples with an LCS of $z+1$ then:

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), SAG_y(Y)) \leq D-1.$$

Case 1 [There is 1 "good" k-tuple, lower bound]: Consider aligning the strings where the k-tuples of strings which have a LCS of $z+1$ line up. Now, we can match m copies of B. What about $\$_i$ symbols? There are $2\ell n$ copies of the $\$_i$ symbol in $AG_i(X_i)$, they only appear in the copies of A (they don't appear in A_{-i}). If we are matching up with a perfect alignment we can match all $2\ell n$ of these symbols. They either line up with copies of A in other strings, or copies of A_{-j}. Finally, there are $2\ell m$ copies of $\$_y$ in $AG_y(Y)$. In a perfect alignment these symbols will all get matched to symbols that appear in copies of A in other strings. So, in total

$$\text{LCS}(AG_1(X_1), AG_2(X_2), \ldots, AG_{k-1}(X_{k-1}), AG_y(Y)) \geq 2\ell m + 2\ell(m+(k-1)n) + zm + 1 = D.$$

Case 2 [There is 1 "good" k-tuple, upper bound]: If there is a singular "good" k-tuple we can not re-arrange the strings to get a larger alignment. If we skip or merge any strings in Y we loose 2ℓ symbols from B at least, giving a maximum LCS of $D-2\ell$. If we skip or merge strings in X_i then we loose at least 2ℓ symbols from $\$_i$ characters. We gain at most ℓ matches, giving a maximum LCS if we merge or skip of $D-\ell$. Thus, the largest LCS possible is D.

Case 3 [There are zero "good" k-tuples, upper bound]: In this case, if we follow a perfect alignment we achieve an LCS of $2\ell m + 2\ell(m+(k-1)n) + zm = D-1$. If we skip a string in Y we miss out on 2ℓ characters. If we match a single string in Y to multiple strings in X_i we loose at least 2ℓ characters and match an additional ℓ characters at most for a total LCS of at most $D-\ell$. Finally, if we skip over strings in X_i, we miss out on 2ℓ characters $\$_i$, for no benefit. All k-tuples have a value of z regardless, so the maximum LCS is $D-2\ell$.

From all these cases we can say that if there is exactly one "good" k-tuple, and it is reachable in a perfect alignment then

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D.$$

If there are no "good" k-tuples

$$\text{LCS}(SAG_1(X_1), SAG_2(X_2), \ldots, SAG_{k-1}(X_{k-1}), SAG_y(Y)) = D-1.$$

For the compression, $D < \ell n$. So we add an additional $\lg(n) + \lg(\ell)$. So if c is the size of an SLP that gives a single variable for all $(k-1)n+m$ strings $X_i[j]$ and $Y[j]$. Then there is a SLP representation of all the strings $AG_1(X_1), \ldots, AG_{k-1}(X_{k-1}), AG_y(Y)$ of size $O(c + \lg(\ell)k + \lg(n) + kn + m)$. Unchanged. ∎

6.5.2 Zero and One Strings First we will use the gadgets for representing zeros and ones from [ABV15].

LEMMA 6.1. (ZERO AND ONE STRINGS [ABV15]) *There are strings $CG_i(0), CG_i(1)$ such that:*

$$k - LCS(CG_1(b_1), \ldots, CG_k(b_k)) = \begin{cases} C & \text{if } b_1 \cdots b_k = 0 \\ C+1 & \text{if } \forall b_1 \cdots b_k = 1 \end{cases}$$

for some positive integer C that is a function of k. Note this corresponds to our desired relationship from k-OV. If we have one "zero string" then we get a small k-LCS, if there are all "one strings" then we get a larger k-LCS. These strings use an alphabet of size $O(1)$. If k is a constant the length of these strings is $O(1)$.

6.5.3 Interleave Gadget Now we will build the gadget that checks if our interleave representations represent a yes instance of orthogonal vectors. For this next Lemma recall Definition 6.6, where we define $\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)$.

LEMMA 6.2. *Let L be a list of n $\{0,1\}$ vectors each of dimension $d = n^{o(1)}$. Let $v_1, \ldots, v_k$ be $\{0,1\}$ vectors each of dimension d. Given the lists, We produce strings: $IVG_1(L, \ell, v_1), \ldots, IVG_{k-1}(L, \ell, v_{k-1}), EIVG(v_k)$ such that*

$$\text{LCS}(IVG_1(L, \ell, v_1), \ldots, IVG_{k-1}(L, \ell, v_{k-1}), EIVG(v_k)) = C$$

if there are $(k-1)\ell$ vectors in L such that are orthogonal with $v_1, \ldots, v_k$. If there do not exist $(k-1)\ell$ vectors in L that are orthogonal with $v_1, \ldots, v_k$ then

$$\text{LCS}(IVG_1(L, \ell, v_1), \ldots, IVG_{k-1}(L, \ell, v_{k-1}), EIVG(v_k)) = C - 1.$$

These strings have length at most $O(n^\ell d)$ and an alphabet of size $O(k)$.

Additionally we can compress x strings $IVG_i(L, \ell, v_1), \ldots, IVG_i(L, \ell, v_x)$ or $EIVG(v_1), \ldots, EIVG(v_x)$ with a total compression size of $O(xd + nd + \ell \lg(n))$.

Proof. Consider the $k-1$ lists $\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)$ for $i \in [1, k-1]$. Additionally, let

$$Vec_E(v_k)[j] = \begin{cases} v_k[h] & \text{if } j = h \cdot n^\ell \\ 0 & \text{else.} \end{cases}$$

Where the total length is $|Vec_E(v_k)| = n^\ell \cdot (d-1) + 1$.

Now create the lists

$$X_i[j] = CG_i(\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)[j])$$
$$Y[j] = CG_k(Vec_E(v_k)[j]).$$

Finally create the strings

$$IVG_i(L, \ell, v_1) = SAG_i(X_i)$$
$$EIVG(v_k) = SAG_y(Y).$$

Now, note that by construction the k-LCS of coordinate gadgets $CG_i(\cdot)$ is either some value z or $z+1$.

Now, note that $Vec_E(v_k)$ has zeros in every location except $h \cdot n^\ell$ for $h \in [1, d]$. If we perfectly align $Vec_E(v_k)$ with the $(k-1)$ vectors $\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)$, then these locations correspond with a vector in each! That is, as mentioned in Definition 6.6, the bits in locations $j, j+n^\ell, \ldots, j+(d-1)n^\ell$ correspond to a vector w where $w[h] = u[h]v_i[h]$ and $u = List[h]$. If there are $(k-1)\ell$ vectors in L that are orthogonal with $v_1, \ldots, v_k$, then there should be a perfect alignment of these vectors such that in every alignment location there is at least one zero. That is, an alignment where every aligned k-tuple has a LCS of $z+1$ as opposed to z.

Let $c = |CG_i(\cdot)|$. Let $z+1$ be the value of $\delta_L CS(CG_1(b_1), \ldots, CG_k(b_k))$ if $b_1 \cdots b_k = 0$.
So $\delta_L CS(CG_1(b_1), \ldots, CG_k(b_k)) = z$ if $b_1 \cdots b_k = 0$.

So, by Theorem 6.3, if there are $(k-1)\ell$ vectors in L that are orthogonal with $v_1, \ldots, v_k$ then

$$\text{LCS}(IVG_1(L, \ell, v_1), \ldots, EIVG(v_k)) = C = 2c(2((d-1)n^\ell + 1) + (k-1)dn^\ell) + (z+1)((d-1)n^\ell + 1).$$

Otherwise,

$$\text{LCS}(IVG_1(L, \ell, v_1), \ldots, IVG_{k-1}(L, \ell, v_{k-1}), EIVG(v_k)) = C - 1.$$

Now we are going to argue that we can compress x strings $IVG_i(L, \ell, v_1), \ldots, IVG_i(L, \ell, v_x)$ or $EIVG(v_1), \ldots, EIVG(v_x)$ with a total compression size of $O(xd + nd)$. First, note that A^f_{-i} can be represented with an SLP of size $O(\lg(\ell) + \lg(n))$. Now the rest of our string is a series of entries that look like $A \cdot CG_i(b) \cdot B$. We can create a SLP for both $S_0 = A \cdot CG_i(0) \cdot B$ and $S_1 = A \cdot CG_i(1) \cdot B$ with size $O(\lg(\ell))$. We give the names of S_0, S_1 to simplify the notation.

Next, if we are compressing x different $EIVG(v_i)$ gadgets, first we want to compress $S_0^{n^\ell - 1}$ which appears repeatedly. We can do this with size $O(\ell \lg(n))$. Finally, we need to add the bits that correspond with each vector. We can do this with size dx. This gives a total size of $O(\ell \lg(n) + dx)$.

Finally, consider the case of compressing x different $IVG(L, \ell, v_j)$ gadgets. This part of the proof, where interleaves are very compressible, borrows very heavily from [ABBK17]. For this we note first that $\mathbf{VecS_I}_\ell(L, v_j)$ has d sections of size n^ℓ that correspond to the d dimensions of the vectors. Any section that corresponds to a h where $v_j[h] = 0$ has n^ℓ copies of S_0. This can be represented with size $O(\ell \lg(n))$. When $v_j[h] = 1$ the section instead corresponds to $\bigcirc_{j\in[1,n^\ell]}\mathbf{List}[j][h]$. By using Definition 6.4 we can re-write this as

$$\mathbf{SubList}[L, k] = \bigcirc_{j_1\in[1,n]} \bigcirc_{j_2\in[1,n]} \bigcirc_{j_3\in[1,n]} \cdots \bigcirc_{j_k\in[1,n]} L[j_1][i] \cdot L[j_2][i] \cdots L[j_k][i].$$

Now note that $\mathbf{SubList}[L, 1]$ is simply a string of n bits and thus has a compression of size n. Now note that $\mathbf{SubList}[L, g]$ is simply a string formed by appending either $\mathbf{SubList}[L, g-1]$ or n^{g-1} zeros. We can represent n^{g-1} zeros with $O(g \lg(n))$ variables. So, if $\mathbf{SubList}[L, g]$ has an SLP $f(g)$ then $\mathbf{SubList}[L, g+1]$ has an SLP $f(g+1) = n + f(g) + g \lg(n)$. We have that $f(1) = n$, so $f(k) = kn + k^2 \lg(n) = O(n)$. Thus, the total size of compressing x different $IVG(L, \ell, v_j)$ gadgets is $O(\ell \lg(n) + dx + dn)$. ∎

6.5.4 Putting it all Together Now that we have an interleave gadget, we want to put many of these gadgets one after each other. However, we want to line up these gadgets. So, we put our interleave gadgets into the perfect alignment gadget.

LEMMA 6.3. *Let a, b, and k be positive constant integers. Let L be a list of n vectors of length $d = n^{o(1)}$.*

There are k strings $GOV_i(a, b, k, L)$ such that:

$$\text{LCS}(GOV_1(a, b, k, L), \ldots, GOV_k(a, b, k, L)) = \begin{cases} D & \text{if there is a } (bk + a(k-1))\text{-OV in } L \\ D-1 & \text{else} \end{cases}$$

The length of each of these strings is $O(n^{a+b+o(1)})$ with an alphabet of size $O(k)$.

Proof. Unique $(bk + a(k-1))$-OV is equivalent to the normal detection problem of $(bk + a(k-1))$-OV, via a randomized reduction [folklore][Vas18]. So, we can consider the case where a single $(bk + a(k-1))$-tuple of vectors are orthogonal.

Let $L_b = \mathbf{List}(L)_b$ as defined in Definition 6.4. Now we will use these to define lists of vectors. For all $i \in [1, k-1]$ let

$$X_i[j] = IVG_i(L, a, L_b[j]),$$
$$Y[j] = EIVG(L_b[j]).$$

So all X_i and Y have length n^b. The strings inside the gadgets is $n^{a+o(1)}$. All k-tuples of these strings have LCS values of either C or $C-1$. We basically want to wrap an alignment gadget around these strings. However, we want to allow any k-tuple to be compared so we will double all the X_i lists:

$$\hat{X}_i[j] = \hat{X}_i[j + n^b] = X_i[j].$$

Now, for any k tuple $j_1, \ldots, j_k$ where $j_i \in [0, n^b - 1]$ there is some offset $\Delta_i \in [0, n^b - 1]$ for all $\hat{X}_i$ such that $j_k = j_1 + \Delta_i \mod n^b$. So, we can align the $\hat{X}_i$ strings with Y and get any k-tuple lined up. So, we can now build our gadgets. For all $i \in [1, k-1]$:

$$GOV_i(a, b, k, L) = SAG_i(\hat{X}_i) \tag{6.30}$$
$$GOV_k(a, b, k, L) = SAG_y(Y) \tag{6.31}$$

Now, if there is a single $(bk + a(k-1))$-OV in our unique OV instance then there is a single $k-$tuple $X_1[j_1], \ldots, Y[j_k]$ of strings that have a LCS of C. Otherwise, they will all have a LCS of $C-1$. By Theorem 6.4 we have that in the first case where an $(bk + a(k-1))$-OV exists

$$\text{LCS}(GOV_1(a, b, k, L), \ldots, GOV_k(a, b, k, L)) = D$$

otherwise

$$\text{LCS}(GOV_1(a, b, k, L), \ldots, GOV_k(a, b, k, L)) = D - 1.$$

Note that having only two possible LCS values hinges crucially on using a unique $(bk + a(k-1))$-OV instance.

The total size of the alphabet is $O(k)$ for the interleaves and another $O(k)$ for the SAG gadgets, with a total alphabet size of $O(k)$.

The total length of the IVG gadgets is $n^{a+o(1)}$. We have n^b copies of these gadgets. Giving a total length of $n^{a+b+o(1)}$. ∎

We will now bound the size of the compressed length of these gadgets.

LEMMA 6.4. *Let k be a constant integer and let $d = n^{o(1)}$. Given the k strings $GOV_i(a, b, k, L)$ defined in Lemma 6.3 the size of the compression is $O(n^{b+o(1)} + n^{1+o(1)})$.*

Proof. We will start by describing the compression size of $CG_i(0), CG_i(1)$. These strings have length $O(1)$, thus the total size of the compression is at most $O(1)$. There are $2k$ of these strings and k is a constant. So we can have $2k$ variables, one for each string and still the total size of the compression will be $O(1)$.

Next, we need to analyze the size of the compression of the kn^b interleave gadgets $IVG_i(L, \ell, v_i)$. These include the zero and one bit representations, and then are wrapped in a perfect alignment gadget. By Lemma 6.2 we have that the total size of the compression of these strings is $O(kd(n + n^b))$. Because we have n^b distinct copies of v_i we are generating the IVG strings with, so the x of the lemma is n^a in our case. Note that $kd = n^{o(1)}$. So the total size of these compressions is $O(n^{1+o(1)} + n^{b+o(1)})$

Next, we need to analyze the size of the compression of the entire string. We use the compression of the interleave gadgets and the coordinate gadgets. In addition to this we are wrapping our interleave gadgets in an alignment gadget (from Lemma 6.4). This adds an additional $O(\lg(n) + n^b)$ variables to the SLP.

This gives an SLP of total size of $O(n^{b+o(1)} + n^{1+o(1)})$. ∎

Now we will now combine the previous lemmas to give the hardness of k-LCS with compression.

REMINDER OF THEOREM 1.1. *If the k'-OV hypothesis is true for all constants k', then for any constant $\epsilon \in (0, 1]$ grammar-compressed k-LCS requires $\left(M^{k-1}m\right)^{1-o(1)}$ time when the alphabet size is $|\Sigma| = \Theta(k)$ and $m = M^{\epsilon \pm o(1)}$. Here, M denotes the total length of the k input strings and m is their total compressed size.*

Proof. Use the gadgets from Lemma 6.3. Call the strings $S_1, \ldots, S_k$ Consider positive integers a and b.

These have an alphabet of size $O(k)$ and length $M = O(n^{a+b+o(1)})$ by Lemma 6.3. These have a compression of total size $m = O(n^{b+o(1)} + n^{1+o(1)})$ by Lemma 6.4.

The size of the alphabet of the reduction is $O(k)$, so if the alphabet is allowed to be size $\Theta(k)$, then this lower bound applies.

So $M^{k-1}m = O(n^{(k-1)a+kb+o(1)})$. If $(bk + a(k-1))$-OV requires $n^{bk+a(k-1)-o(1)}$ time, then this corresponds to a lower bound of $\left(M^{k-1}m\right)^{1-o(1)}$ for SLP compressed $k-$LCS.

Consider a contradiction to our theorem statement. There would be an algorithm running in $(M^{k-1}m)^{1-\gamma}$ time to solve grammar compressed k-LCS when $m = M^{\epsilon \pm o(1)}$ and $\epsilon \in (0, 1]$. In the easiest case we can pick an a, b such that $b/(a + b) = \epsilon$, in this case we are done. For irrational ϵ we need to approximate and then pad the strings. Choose an a and b such that $\epsilon \leq b/(a + b) < \epsilon(1 + \gamma/2)$. Such a, b exist that are in $O(\frac{1}{\epsilon\gamma})$. We add a new character 3. Let $S'_i = S_i 3^x$, where we will set $x \in [M, M^2]$ later. Note that $\text{LCS}(S'_1, \ldots, S'_k) = \text{LCS}(S_1, \ldots, S_k) + x$. Note that the compression of these strings is $m' = m + \lg(x) = \Theta(m)$ where as the length is $M' = M + x = \Theta(x)$. Choose $x = m^{1/\epsilon \pm o(1)} = M^{b/(a+b)\cdot 1/\epsilon}$. So now $m' = M'^{1/\epsilon \pm o(1)}$. Note that $1 \leq b/(a + b) \cdot 1/\epsilon \leq (1 + \gamma/2)$. Now consider running the claimed fast algorithm on our new $S'_1, \ldots, S'_k$ instance. The running time is

$$(M'^{(k-1)}m')^{1-\gamma} = O((M^{(1+\gamma/2)(k-1)}m)^{1-\gamma-o(1)}).$$

This running time can be simplified to $O(M^{(k-1)}m)^{(1+\gamma/2)(1-\gamma-o(1))}$. Note that $(1 + \gamma/2)(1 - \gamma - o(1))$ is less than $1 - o(1)$. This algorithm violates the lower bound for the original $S_1, \ldots, S_k$ instance. This is a contradiction.

So any algorithm running in $(M^{k-1}m)^{1-\gamma}$ time to solve grammar compressed k-LCS when $m = M^{\epsilon \pm o(1)}$ and $\epsilon \in (0, 1]$ violates k'-OV for some k' that depends on ϵ, γ. This implies our theorem statement. ∎

6.6 Easy Edit Distance Lower Bounds from LCS In this section we will prove that k-median edit distance is hard from k'-LCS. We take a k'-LCS instance and add various numbers of empty strings. This pushes the k-median edit distance problem towards deletions. So, we increase the number of strings, but we don't increase the total uncompressed or compressed length of the input.

Nicolas and Rivals show NP-hardness for k-edit distance through k'-LCS for large k and k' [NR05]. We take inspiration from their reduction to build our own, removing some of their restrictions, and making it fine-grained efficient. We then use the hardness results we have for k'-LCS to get lower bounds for k-edit distance. We will be focusing on a version of edit distance where the strings are allowed to be of very different sizes. We will give an explicit definition now.

DEFINITION 6.9. *Given k strings $S_1, \ldots, S_k$ of lengths $M_1, M_2, \ldots, M_k$ the k-edit distance (or k-median edit distance) of those strings is the minimum sum across all strings of edits needed to make all strings equal some new string S'. The allowed edits are deleting a character, adding a character and replacing a character (Levenshtein distance).*

More formally: Recall that $\delta_E(S_i, S')$ denotes the edit distance of S_i and S'. Recall that the k-median distance is:

$$\delta_E(S_1, S_2, \ldots, S_k) = \min_{S' \in All\ Strings} \left(\sum_{i=[1,k]} \delta_E(S_i, S') \right).$$

We can use inspiration from [NR05] to get lower bounds for the center version of this problem as well. Let us remind the definition of k-center edit distance problem.

DEFINITION 6.10. *Given k strings $S_1, \ldots, S_k$ of lengths $M_1, M_2, \ldots, M_k$. We define the k-center edit distance of those strings is the minimum of the maximum of the distances of those strings to a string S'. The allowed edits are deleting a character, adding a character and replacing a character (Levenshtein distance).*

More formally: Let $\delta_E(S_i, S')$ be the edit distance of S_i and S'. Now the k-center distance is:

$$\delta_{CE}(S_1, S_2, \ldots, S_k) = \min_{S' \in All\ Strings} \left(\max_{i=[1,k]} \delta_E(S_i, S') \right).$$

6.7 Median Edit Distance

THEOREM 6.5. *We are given a k-LCS instance with strings $S_1, \ldots, S_k$ all of length M. Let the k-LCS of these strings be denoted by $\mathrm{LCS}(S_1, \ldots, S_k)$. The $(2k-1)$-median edit distance on S_1 through S_k and $k-1$ copies of the empty string γ is related to the k-LCS of S_1 through S_k:*

$$\delta_E(S_1, \ldots, S_k, \gamma, \ldots, \gamma) = Mk - \mathrm{LCS}(S_1, \ldots, S_k).$$

Proof. First, let us prove that $\delta_E(S_1, \ldots, S_k, \gamma, \ldots, \gamma) \leq Mk - \mathrm{LCS}(S_1, \ldots, S_k)$. Let T be the target string of $\mathrm{LCS}(S_1, \ldots, S_k) = |T|$. Then, the sum of edit distances to T is $k(n - |T|) + (k-1)|T| = kn - |T|$.

Second, let us prove that $\delta_E(S_1, \ldots, S_k, \gamma, \ldots, \gamma) \geq Mk - \mathrm{LCS}(S_1, \ldots, S_k)$. Let T' be a target string. Now let d_j, i_j, b_j be the number of deletions, insertions and substitutions to go from S_j to T'. Let $e_j = d_j + i_j + b_j$ be the edit distance of S_j to T'. Now note that $e_j \geq M - |T| + 2i_j + b_j$. Additionally, note that the distance from γ to T' is $|T'|$. So the total distance is

$$kM - k|T'| + \sum_{j=1}^{k} 2i_j + b_j + (k-1)|T'| = kM - |T'| + \sum_{j=1}^{k} 2i_j + b_j.$$

So, $\delta_E(S_1, \ldots, S_k, \gamma, \ldots, \gamma)$ can only be less if $|T'| > |T|$. Note, that $\sum_{j=1}^{k} 2i_j + b_j \geq |T'| - |T|$. The target T is the longest string to be achieved with only deletions. Any change from this T (notably added characters) must involve at least one substitution or an insertion. So we can say that the total distance is

$$kM - |T'| + \sum_{j=1}^{k} 2i_j + b_j \geq kM - |T'| + |T'| - |T| = kM - |T|.$$

So, $\delta_E(S_1, \ldots, S_k, \gamma, \ldots, \gamma) \geq Mk - \text{LCS}(S_1, \ldots, S_k)$.

Thus, $\delta_E(S_1, \ldots, S_k, \gamma, \ldots, \gamma) = Mk - \text{LCS}(S_1, \ldots, S_k)$. ∎

Now that we have a tight relationship between the edit distance and LCS, we can use this to get a lower bound from SETH through LCS.

THEOREM 6.6. *Given an instance of k-median edit distance on strings of lengths $M_1 \leq M_2 \leq \cdots \leq M_k$ where these strings can all be compressed into a SLP of size m. Then, an algorithm for k-median edit distance that runs in $((M_2+1)\cdots(M_{2k-1}+1)\cdot m)^{1-\epsilon}$ time for constant $\epsilon > 0$ violates SETH.*

Proof. We will use Theorem 1.1 and Theorem 6.5.

Say we are given an instance of k-LCS with strings $S_1, \ldots, S_k$ of length M and a SLP compression of all strings of size m. Then, by Theorem 6.5 we can solve this with an instance of $(2k-1)$-median edit distance on k strings $S_1, \ldots, S_k, \gamma, \ldots, \gamma$. We can compress these k strings with a compression of size $m' = m + O(k)$ (we need only compress the empty string in addition).

k-LCS requires $(M^{k-1}m)^{1-o(1)}$ if SETH is true. Note that for our chosen strings $M^{k-1} = M_2 \cdots M_k$. Now note that our compression is of size $m' = O(m)$. The reduction takes constant time (simply append the empty string and make a call to k-median edit distance). So k-median edit distance requires $((M_2+1)\cdots(M_{2k-1}+1)\cdot m)^{1-o(1)}$ time if SETH is true. We can re-state this as an algorithm running time $((M_2+1)\cdots(M_{2k-1}+1)\cdot m)^{1-\epsilon}$ time for constant $\epsilon > 0$ violates SETH. ∎

Next we will use similar ideas to show hardness for center edit distance.

6.8 Center Edit Distance Nicolas and Rivals present a very simple reduction from a specific version of $(k-1)$-LCS to k-Center Edit Distance. This reduction simply adds the empty string as the last string. The same concept works here. We can distinguish between the case where a $(k-1)$-LCS is greater than or equal to some constant c. Because, if all the strings in the k-LCS are of length M adding a single empty string distinguishes between the $(k-1)$-LCS less than $M/2$ or greater than or equal to it. Why? Because, if the k-LCS at least $M/2$ then every string is $M/2$ deletions away from the target string *and* the new empty string is as well! Otherwise, if the LCS is less than $M/2$, we are more than $M/2$ edits away provably. By adding characters to our $(k-1)-$LCS strings we can artificially increase the match (adding a large number of matching characters to each string), or artificially decrease it (add a large number of not-matching characters). By doing this we can go from our $(k-1)$-LCS being c to our $(k-1)$-LCS being $M'/2$, for our new length of strings.

LEMMA 6.5. *Assume a k-LCS instance over k strings of length exactly M. If deciding whether the k-LCS distance is equal to $M/2$ over an alphabet of size $|\Sigma|$ can be done in $T(M)$ time, then we can decide whether the k-LCS distance is equal to C over an alphabet of size $|\Sigma| + k + 1$ for any constant C in time $O(T(M) + kM)$.*

Proof. Let $S_1, \ldots, S_k$ be an instance of k-LCS where we want to decide if the distance is exactly C. Let the k-LCS be $\text{LCS}(S_1, \ldots, S_k)$.

For integers a and b let

$$S_i' = @^a S_i \#_i^b.$$

That is, we append a @ symbols at the start and b $\#_i$ symbols at the end of each string. The $\#_i$ strings can not be matched. The @ symbols can be trivially matched. So we have that $|S_i'| = M' = M + a + b$ and

$$\text{LCS}(S_1', \ldots, S_k') = \text{LCS}(S_1, \ldots, S_k) + a = C + a.$$

We simply want to choose values of a and b such that $2(C+a) = n + a + b$. This simplifies to $a = M + b - 2C$. If $2C > M$ then $b = 2C - M$ and $a = 0$. If $2C < M$ then $b = 0$ and $a = M - 2C$.

The length of these strings is $M' = 2C$ or $M' = 2M - 2C$, both are less than $2M$. So, in $O(kM)$ time we can produce new strings of length M' where determining if the k-LCS is exactly $M'/2$ determines if the original instance had distance exactly C. ∎

Now we will show that k-center edit distance solves $(k-1)$-LCS.

THEOREM 6.7. *We are given a k-LCS instance with strings $S_1, \ldots, S_k$ all of length M where k-LCS of these strings is denoted by* $\text{LCS}(S_1, \ldots, S_k)$. *The $(k+1)$-center edit distance of $S_1, \ldots, S_k$ and emptystring γ and k-LCS are related as follows.*

$$\delta_{CE}(S_1, \ldots, S_k, \gamma) \begin{cases} = M/2 & \text{if } \text{LCS}(S_1, \ldots, S_k) \geq M/2, \\ > M/2 & \text{if } \text{LCS}(S_1, \ldots, S_k) < M/2. \end{cases}$$

Proof. Consider first, what's the length of a target string for $\delta_{CE}(S_1, \ldots, S_k, \gamma) = M/2$. Call this target central string T. If $|T| > M/2$ then the distance from γ to T is greater than $M/2$. If $|T| < M/2$ then the strings S_i must have more than $M/2$ deletions, giving a distance greater than $M/2$. So, to hit $M/2$ the target string must have length $M/2$ exactly.

Next note that for the empty string to reach length $M/2$ it must simply have $M/2$ insertions. For any of the S_i strings to go down to $M/2$ they must simply have $M/2$ deletions.

Note that $\text{LCS}(S_1, \ldots, S_k)$ is defined as M minus the number of deletions needed in each string to reach the minimal target. Thus, with this addition of an empty string

$$\delta_{CE}(S_1, \ldots, S_k, \gamma) \begin{cases} = M/2 & \text{if } \text{LCS}(S_1, \ldots, S_k) \geq M/2 \\ > M/2 & \text{if } \text{LCS}(S_1, \ldots, S_k) < M/2 \end{cases}.$$

□

Now we will apply the above Lemma 6.5 and Theorem 6.7 to get a k-center edit distance lower bound from SETH.

REMINDER OF THEOREM 6.1. *Given an instance of k-center edit distance on strings of lengths $M_1 \leq M_2 \leq \cdots \leq M_k$ where these strings can all be compressed into a SLP of size m, then, an algorithm for k-center edit distance that runs in time $((M_2+1)\cdots(M_k+1)\cdot m)^{1-\epsilon}$ time for constant $\epsilon > 0$ violates SETH.*

Proof. We will use Theorem 1.1, Lemma 6.5 and Theorem 6.7.

Say we are given an instance of k-LCS with strings $S_1, \ldots, S_k$ of length M and an SLP compression of all strings of size m. Determining if the $k-$LCS is some integer C requires $(M^{k-1}m)^{1-o(1)}$ time if SETH is true by Theorem 1.1.

Then Lemma 6.5 simply appends at most M symbols @ or $\#_i$ to each string making a new problem $S'_1, \ldots, S'_k$ of length M'. Note that the size of the compression is now $m' = m + O(k \lg(M))$. Now determining if the $k-$LCS is $M'/2$ requires $\left((M')^{k-1}m'\right)^{1-o(1)}$ if SETH is true.

Now we will apply Theorem 6.7. We can produce an instance of $(k+1)$-center edit distance that has strings $S'_1, \ldots, S'_k, \gamma$ that distinguishes between the k-LCS of $S'_1, \ldots, S'_k$ being $M'/2$ or not. Now note that $M_i = |S'_i|$ and $M_{k+1} = 0$. So $(M_2+1)\cdots(M_{k+1}+1) = \Theta((M')^{k-1})$. The compression of this empty string means that the new compression has size $m'' = m' + O(1) = m + O(k \lg(M))$.

We can run this a second time where we add two characters to each string: $S'_i = S_i\%_i\%_i$. These characters are unmatchable. Also, if the LCS used to be at least $M'/2+1$ it will still be at least half the length of the strings. So, we can distinguish the exact value. Similarly, the compression of these strings is of size at most $m'' + O(k) = m'' + O(1)$.

So, we get that an algorithm for $(k+1)$-median edit distance that runs in time $((M_2+1)\cdots(M_{k+1}+1)\cdot m)^{1-\epsilon}$ time for constant $\epsilon > 0$ violates SETH. □

6.9 Edit Distance Lower Bounds from SETH In this section we show a better lower bound for k-edit-distance by reducing from SETH directly. A recent paper has given $M^{k-o(1)}$ lower bounds for Edit Distance from SETH where M is the length of the strings [HBGT20]. In this section we show a $M^{k-1-o(1)}m$ lower bound for compressed k-edit-distance where m is the size of the SLP describing the strings. Our reduction uses the ideas from the SETH lower bound for k-edit-distance to achieve this. We will use the same ideas and list structures that we used in the k-LCS lower bound. We use many of the same notions of gadgets, however, to distinguish between them, we add ED to the end of the name of the gadgets (for Edit Distance). Note that the structure of this proof mirror almost exactly the k-LCS lower bound. However, due to the different distance measures we need to generate different gadgets.

The main takeaway of this section is that in order to build an interleave gadget for edit distance we need to generate a selector gadget that has one value if all values match, and another if not all values match.

The primary difficulty in generalizing this lower bound comes from the variable costs of partial matches. That is, if we have the edit distance of $\delta_E(a,a,a,a,b) = 1$, where as $\delta_E(a,a,b,b,c) = 3$. By contrast, the LCS of both is $\text{LCS}(a,a,a,a,b) = \text{LCS}(a,a,b,b,c) = 0$. So, the overall structure needs to account for this in some way. We want to re-create a perfect alignment gadget, but for Edit-Distance. This will give us two results. First we generalize the 2-LCS lower bound into a 2-edit distance lower bound, answering an explicit open problem given by [ABBK17].

We will use the pre-existing coordinate gadgets and alignment gadgets from [HBGT20]. So, we have two primary tasks. We need to generate and prove the correctness of *perfect* alignment gadgets. Additionally, we need to analyze the size of the compression of both our gadgets and the [HBGT20] gadgets.

6.10 Selection Gadgets We want an additional gadget. A selector gadget. These allow us to say either strings $A_1, \ldots, A_k$ are compared or strings $B_1, \ldots, B_k$ are compared but not both. We will use a version that works for single characters.

LEMMA 6.6. *There exist single character selection gadgets $SCSG_i(\cdot)$ such that the length is polynomial in k and they add a single character to the alphabet. The k-edit distance of k $SCSG_i(c_i)$ strings is either some constant Q if all characters match or $Q + v$ (where v is a positive integer) if one character does not match.*

Proof. First let us define the gadget in terms of two free variables we will set later, a and b:

$$SCSG_i(c) = \#^{ib} c^a \#^{(k-1-i)b}$$

Now note that if we match the characters c together we must fail to match many # characters. Specifically these induce an edit distance of:

$$\begin{cases} bk^2/4 & \text{if } k \text{ is even} \\ b(k^2-1)/4 & \text{if } k \text{ is odd} \end{cases}$$

Now note that if we instead match the # characters then we have an edit distance of ak, as we have to delete the characters input to the gadget.

Also note that if we match the characters c and one or more symbols are off the edit distance will be **at least**:

$$\begin{cases} bk^2/4 + a & \text{if } k \text{ is even} \\ b(k^2-1)/4 + a & \text{if } k \text{ is odd} \end{cases}$$

So if we can choose an a and b such that:

$$\begin{cases} bk^2/4 < ak \leq bk^2/4 + a & \text{if } k \text{ is even} \\ b(k^2-1)/4 < ak \leq b(k^2-1)/4 + a & \text{if } k \text{ is odd} \end{cases}$$

then, if all characters match we get an edit distance of $bk^2/4$, otherwise, we get an edit distance of ak. ∎

Next we will note the existence of coordinate gadgets from previous work. Then we will combine the coordinate gadgets with our selector gadgets to make interleave gadgets.

6.11 Coordinate Gadgets We will use the coordinate gadgets from the [HBGT20] in our reduction.

LEMMA 6.7. (COORDINATE GADGET LEMMA FROM [HBGT20]) *Let $b_1, \ldots, b_k$ be in $\{0,1\}$. Let $C^- = 2(k-1)^2$ and let $C^+ = C^- + k - 1 = (2k-1)(k-1)$. Then,*

$$\delta_E(CGED_1(b_1), \ldots, CGED_k(b_k)) = \begin{cases} C^- & \text{if } b_1 \cdots b_k = 0 \\ C^+ & \text{otherwise} \end{cases}$$

We will use these inside our interleave vector gadgets.

6.12 Interleave Vector Gadget We are given a $(a(k-1)+bk)$-OV with a list of n vectors each of length d. We want to take every vector $v_j = \mathbf{List}(L)_b[j]$ for $j \in [0, n^b]$ and combine them with the interleave representation of a lists. Recall that in Definition 6.6 we define $\mathbf{VecS}_{\mathbf{I}a}(L, v_j)$ as the explicit distribution of the interleave representation of a lists mixed with a single vector. So, we want to have $k-1$ strings that hold representations of $\mathbf{VecS}_{\mathbf{I}a}(L, v_j)$ for all $j \in [0, n^b]$. Finally, we need one string that is full of representations of vectors v_j for all $j \in [0, n^b]$, padded with many zeros. If we do this and we can force a perfect alignment of these gadgets. We will use an altered version of the sliding pyramids from previous work [HBGT20] (see Figure 1).

LEMMA 6.8. *Treat k, ℓ as constants. We are given as input a list L of n vectors each of length d. Where $d = n^{o(1)}$. Let $v_1, \ldots, v_k$ be $\{0,1\}$ vectors each of dimension d. Then there are gadgets $IED'_i(L, v_i)$ and $EED'(v_k)$ such that:*

$$\delta_E(IED'_1(L, \ell, v_1), \ldots, IED'_{k-1}(L, \ell, v_{k-1}), EED'(v_k)) = C$$

if there are $(k-1)\ell$ vectors in L such that are orthogonal with $v_1, \ldots, v_k$. If there do not exist $(k-1)\ell$ vectors in L that are orthogonal with $v_1, \ldots, v_k$ then

$$\delta_E(IED'_1(L, \ell, v_1), \ldots, IED'_{k-1}(L, \ell, v_{k-1}), EED'(v_k)) \geq C + 1.$$

Additionally we can compress x strings $IED'_i(L, \ell, v_1), \ldots, IED'_i(L, \ell, v_x)$ or $EED'(v_1), \ldots, EED'(v_x)$ with a total compression size of $O(xd + nd + \lg(n))$.

Proof. As in the k-LCS Lemma 6.2 consider the $k-1$ lists $\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)$ for $i \in [1, k-1]$. Additionally, let

$$Vec_E(v_k)[j] = \begin{cases} v_k[h] & \text{if } j = h \cdot n^\ell \\ 0 & \text{else.} \end{cases}$$

We will build our gadgets IED' and EED' from these lists. Every entry in these lists is either a zero or a one. We want to force a prefect alignment and check orthogonality of the perfect alignment. That is, we want to hit one value if there exists a $\Delta_1, \ldots, \Delta_{k-1}$ such that

$$\sum_{j=0}^{n^\ell(d-1)+1} Vec_E(v_k)[j] \cdot \mathbf{VecS}_{\mathbf{I}\ell}(L, v_1)[j + \Delta_1] \cdots \mathbf{VecS}_{\mathbf{I}\ell}(L, v_{k-1})[j + \Delta_{k-1}] = 0.$$

To do this we will use the very convenient coordinate gadgets, but alter them with a selector. We want the selector gadget to force an alignment of the true correct values. We add a new character 2, this character is just there to be matched in these gadgets. We add another new character 3, which encourages lining up coordinate gadgets. We will set $x = 100|CGED_i(b_i)|$. We want to have enough copies of the $SCSG$ gadgets that lining up real gadgets with each-other is optimal. We set $y = 100x|SCSG_i(2)|$, we want enough copies of 3 to force coordinate alignments to be optimal. Finally, our updated coordinate gadgets are below

$$CGED'_i(b_i) = 3^y \circ (SCSG_i(2))^x \circ CGED_i(b_i).$$

Next we need to generate "fake" coordinate gadgets to fill out space, so that any offset will be valid. We add the characters $\%_i$ for $i \in [1, k]$. The character $\%_i$ will only appear in the i^{th} string. This will guarantee these characters are unmatched. A fake gadget will have a selector gadget wrapped around one of these unmatchable characters and a coordinate gadget of a zero:

$$F_i = 3^y \circ (SCSG_i(\%_i))^x \circ CGED_i(0).$$

Let $f = |\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)| - |Vec_E(v_k)| = n^\ell - 1$. Now, we will define the three parts of the strings. The section of real gadgets, the section of fake gadgets, and the section of unmatchable characters. We add k new characters $\#_i$, with the intention of making them unmatchable. Let $z = |CGED'_i(b_i)| = |F_i|$. See Figure 1 for a visual depiction.

$$\begin{aligned} REAL_i &= \bigcirc_{j \in [1, dn^\ell]} (CGED'_i(\mathbf{VecS}_{\mathbf{I}\ell}(L, v_i)[j])) \qquad \text{when } i \in [1, k-1] \\ REAL_k &= \bigcirc_{j \in [1, n^\ell(d-1)+1]} (CGED'_k(Vec_E(v_k)[j])) \\ FAKE_i &= F_i^{f(k+1-i)} \\ UNMA_i &= \#_i^{(k+i+1)zf}. \end{aligned}$$

Now that we have defined these useful parts we can define the overall gadgets:

$$IED'_i(L,\ell,v_i) = UNMA_i \circ FAKE_i \circ REAL_i \circ FAKE_i \circ UNMA_i$$
$$EED'(v_k) = UNMA_k \circ FAKE_k \circ REAL_k \circ FAKE_k \circ UNMA_k$$

Now let us consider what happens if there are $(k-1)\ell$ vectors that are orthogonal to $v_1,\ldots,v_k$ then we want to compute the edit distance. Note that all characters in $UNMA_i$ will either be deleted or substituted which has an edit distance of one per character. This gives a total edit distance cost of $zfk(k+5)/2$. Let $p_i = \delta_E(F_1,\ldots,F_i)$. Now, on any valid alignment we have $2f$ fake gadgets completely unmatched, $2f$ fake gadgets matched with one other fake gadget, $2f$ fake gadgets matched with two other fake gadgets, etc. Until we get to $(k-1)$ fake gadgets matched together at which point we have $3f$ of these. When the fake gadgets are matched with each other they are also "matched" with the unmatchable characters. Those characters will simply substitute/delete to equal the output string. We have enough unmatchable characters that their length is longer than the overhanging fake characters. So we can be assured no insertions will need to happen. So the edit distance contribution of these is

$$fp_{k-1} + \sum_{i=1}^{k-1} 2fp_i.$$

Now, we have $2f$ fake gadgets which line up with a mix of fake and real gadgets. Each of these k tuples of lined up gadgets have a contribution of $x(Q+v)$ from their $SCSG$ gadgets, and the $CGED$ gadgets contribute C^- edit distance (the 3 symbols match perfectly and thus have no contribution to the edit distance). So these give a contribution of $2f(x(Q+v)+C^-)$. Finally, we have $M=(d-1)n^\ell+1$ real gadgets which line up with other real gadgets and these contribute $xQ+C^-$ edit distance each. So our total edit distance is

$$C = zfk(k+5)/2 + fp_{k-1} + \sum_{i=1}^{k-1} 2fp_i + 2f(x(Q+v)+C^-) + M(xQ+C^-).$$

What happens if we don't have a set of $(k-1)\ell$ vectors which are orthogonal to $v_1,\ldots,v_k$? If we similarly line up gadgets in a valid way, as above, then we have at least one of the real k-tuples of $CGED'$ gadgets where the internal $CGED$ gadgets contribute C^+, increasing the above cost by $C^+ - C^-$. What if we instead don't do a valid alignment? If we skip aligning a coordinate gadget we skip some 3^y symbols, these then cost an additional y in the edit distance. If we instead align some of the M real gadgets of string k to fake gadgets we miss out on matches of the $SCSG$ gadgets, costing xv in the edit distance. So if there is no set of $(k-1)\ell$ vectors which are orthogonal to $v_1,\ldots,v_k$ then the edit distance is higher than C.

First, we can generate the SLP for $FAKE_i$ and $UNMA_i$. The size of F_i and $\%_i$ are both $O(1)$ (assuming k is a constant). So, we simply need to handle many repetitions. This requires an SLP of size $\lg(f(k+1-i))$ and one of size $\lg((i+1)zf)$ for each $i\in[1,k]$. Luckily for us, in total this SLP will have size $O(\ell\lg(n))$. We additionally need to represent the various values for $REAL_i$. First note that $CGED'_i$ gadgets can be represented with size $O(1)$ SLPs (when k is constant). For $REAL_k$ this is easy from this point on. There are long strings of zero gadgets with only d instances of non zeros. The total representation is $O(d+\ell\lg(n))$. So we just need $REAL_i$ for $i\in[1,k-1]$. Now note that we can use the same structure we used in the k-LCS SLP for these interleave gadgets. We can build the structures for different values of ℓ.

For convenience let $REAL_i^\ell$ be the real gadget for $IED'_i(L,\ell,v_i)$. Now note that $REAL_i^1$ has an SLP of size $O(dn)$ trivially, it only has length $O(dn)$ in the first place. Now consider separating out the parts that correspond to each of the d dimensions of the vector. Next note that we can form these d parts of $REAL_i^{j+1}$ by concatenating n instances of the parts of either $REAL_i^j$ or n^j zero coordinate gadgets. So, given an SLP for $REAL_i^j$ we can create an SLP for $REAL_i^{j+1}$ with an additional size of $n+j\lg(n)$. This gives a total size of $O(\ell^2\lg(n)+\ell dn)$, as ℓ is a constant we have have that the size is $O(dn)$.

So the total size of the SLP will be $O(dn+dx+lg(n))$. ∎

Notice that we can get away without having the same type of $\$_i$ interleaved symbols that we used in k-LCS. This is due to the cost of edits varying even when only some subset of the k strings match on a symbol. We can guarantee we don't skip characters because it will cost us in edits, even if those characters could only be matched

up to one other string. However, we aren't quite done. We want to wrap this so that the value is either a match or one higher than a match. We don't want to have the final interleave gadgets give variable outputs depending on how orthogonal vectors are. We want the same value no matter what.

LEMMA 6.9. *We are given as input a list L of n vectors each of length d, where $d = n^{o(1)}$. Let ℓ be a constant. Let $v_1, \ldots, v_k$ be $\{0,1\}$ vectors each of dimension d. Then there are gadgets $IED_i(L, v_i)$ and $EED(v_k)$ such that for some constants D and w:*

$$\delta_E(IED_1(L, \ell, v_1), \ldots, IED_{k-1}(L, \ell, v_{k-1}), EED(v_k)) = D$$

if there are $(k-1)\ell$ vectors in L such that are orthogonal with $v_1, \ldots, v_k$. If there do not exist $(k-1)\ell$ vectors in L that are orthogonal with $v_1, \ldots, v_k$ then

$$\delta_E(IED_1(L, \ell, v_1), \ldots, IED_{k-1}(L, \ell, v_{k-1}), EED(v_k)) = D + w.$$

Additionally we can compress x strings $IED_i(L, \ell, v_1), \ldots, IED_i(L, \ell, v_x)$ or $EED(v_1), \ldots, EED(v_x)$ with a total compression size of $O(xd + nd + x\lg(n))$.

The strings IED and EED have length $n^{\ell+o(1)}$.

Proof. Let u be an all zero vector of length d. Let v_i' be the vector v_i but with an added last index $v_i'[d+1] = 1$ if $i \in [1, k]$. Let u' be the all zeros vector except for an added last index $u[d+1] = 0$. Let v_k^* be the vector v_k but an added last index $v_k^*[d+1] = 0$. Now we add additional characters 5 and 4. We add copies where $p = 100|IED_i'(L, \ell, v_i')|$ and $q = 10p$. Now we generate the following:

$$IED_i(L, \ell, v_i) = 5^q 4^p IED_i'(L, \ell, v_i') 4^p 5^q$$
$$EED(v_k) = 5^q EED'(v_k^*) 4^p EED'(u') 5^q.$$

Here we match up the 5 characters. Finally, we must match the 4^q symbols. There will be q unmatched 4 symbols. Finally, we will have $|EED'(v_k^*)| = |EED'(u')|$ unmatched symbols no matter what. Now, how much comes from matching the IED' and EED' gadgets? If there are $(k-1)\ell$ vectors are orthogonal to $v_1, \ldots, v_k$ then the cost is C. If there aren't then the cost of matching the symbols to $EED'(u')$ instead is $C + C^+ - C^-$. So, $D = C + q + |EED'(v_k^*)|$, and $w = C^+ - C^-$.

For the size of the SLP we are doubling the number of EED' gadgets, and we are adding in the 5 and 4 symbols. So the total size should be $O(2xd + nd + \ell\lg(n) + 7x(\lg(p) + \lg(q)))$. Given the size of p and q this gives: $O(xd + nd + x\lg(n))$.

For the length of the strings we have at most $O(dn^\ell)$ coordinate gadgets and the number of unmatached symbols is $O(dn^\ell)$. Note that the size of coordinate gadgets is constant when k is constant and $d = n^{o(1)}$. So the total length of our generated strings IED and EED is $O(n^{\ell+o(1)})$. ∎

Now that we have generated interleave vector gadgets we will put multiple copies of them and align them. We want to set this up using the same 'sliding pyramid' set up as we used for the interleave gadgets.

6.13 Aligning Interleave Vector Gadgets For aligning our gadgets we generalize the idea from [HBGT20] for aligning gadgets. First, we create a fake list of vectors F that is n vectors of dimension d where every entry of the vectors is 1. Then we create "fake" versions of the alignment and empty vector gadgets, build from F instead of L. We concatenate the "real" IED' and EED' gadgets with many matchable symbols in between. We surround these real gadgets with our "fake" gadgets. We also build a "pyramid" that allows the strings to have any valid alignment of the real gadgets while having the same number of matches of the fake gadgets. Around these we put an additional number of unmatchable characters (characters that are unique to each set). See Figure 1 for a depiction. These fake gadgets allow for any choice of alignment of the real gadgets to have the same value of matches outside of the k gadgets we are matching with the alignment.

We start by proving we can create strings such that k-edit distance can be used to detect $(a(k-1)+bk)$-OVs.

LEMMA 6.10. *Let a, b, and k be positive constant integers. Let L be a list of n length d vectors, where $d = n^{o(1)}$.*

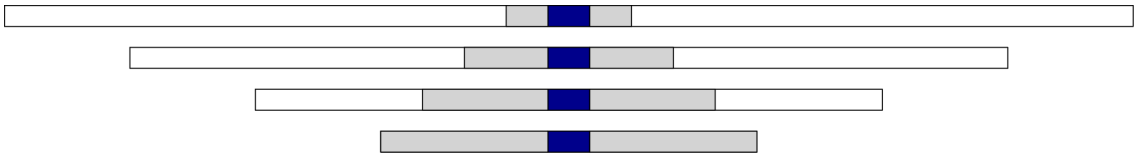


Figure 1: A visual depiction of the structure of our alignment. This is using the ideas from [HBGT20]. The dark blue section is a depiction are the real gadgets. The light-gray section are the "fake" gadgets. The white sections are unmatchable characters (a distinct character in each string).

There are k strings $EDOV_i(a,b,k,L)$ such that:

$$\delta_E(EDOV_1(a,b,k,L),\ldots,EDOV_k(a,b,k,L)) \begin{cases} \geq E & \text{if there is a } (a(k-1)+bk)\text{-OV in } L \\ \leq E-1 & \text{else} \end{cases}$$

The length of each of these strings is $O(n^{a+b+o(1)})$ with an alphabet of size $O(k)$.

Proof. We are given as input a $(a(k-1)+bk)-$OV instance. Say the list is L. It contains n zero one vectors of length d.

We define some new symbols. We add a new symbol \$, we will use this to encourage matching in lined up sections. We will also add $\%_i$ symbols for $i \in [1,k]$. A symbol $\%_i$ appears only in string i, thus it can not be matched, it must be deleted or substituted.

Let us now define the real sections of these strings (the blue section at the center of the strings in Figure 1). First we will define this for $i \in [1,k-1]$:

$$REAL_i = \bigcirc_{j\in[1,n^b]} IED'(L, \mathbf{List}(L)_b[j], a)$$

and for the last string

$$REAL_k = \bigcirc_{j\in[1,n^b]} EED'(\mathbf{List}(L)_b[j]).$$

Next let us define the fake gadget part of these strings. First a single fake gadget is generated by plugging in $\hat{L}_F$, a list of n all ones vectors all of length d. And the vector $\hat{u}_F$, a length d vector of all ones. For $i \in [1,k-1]$

$$F_i = IED'(\hat{L}_F, \hat{u}_F, a)$$

and then for $i = k$

$$F_k = EED'(\hat{u}_F).$$

We can now define our fake gadgets, the gray parts of Figure 1:

$$FAKE_i = F_i^{in^b}$$

Now we will define the unmatched symbol sections. We will add new symbols $\&_i$ for $i \in [1,k]$. Note that $\&_i$ appears only in string i.

$$UNMA_i = \&_i^{|F_i|n^b(2k-i)}$$

Now let us define our gadgets for $i \in [1,k-1]$:

$$EDOV_i(a,b,k,L) = UNMA_i \circ FAKE_i \circ REAL_i \circ FAKE_i \circ UNMA_i.$$

Now note that if there is a $(a(k-1)+bk)$-OV this corresponds to a k-tuple of $IED'_1,\ldots,IED'_{k-1},EED'$ gadgets in this construction having an edit distance of C (smaller than $C+w$). Additionally, note that if there is no $(a(k-1)+bk)$-OV then all k-tuple of $IED'_1,\ldots,IED'_{k-1},EED'$ gadgets have an edit distance of $C+w$.

Now, if there is a $(a(k-1)+bk)$-OV then, we can align the gadgets and give an upper bound on total cost. First, all unmatched characters will cost 1 so they have total cost of $|F_i|n^b(2k^2-k(k+1)/2)$. Next, consider the fake gadgets that overhang and interact with fewer than k other gadgets. Let $p_i = \delta_E(F_k,\ldots,F_i)$. There are $2n$

fake gadgets that are aligned with i total gadgets and otherwise aligned with the unmatchable characters. The cost for these is $\sum_{i\in[1,k]} 2np_i$. Finally, there are $3n$ gadgets which line up with a full k other gadgets. If m of these represent underlying orthogonal vectors then all m tuples will have a cost of D, the rest will have a cost of $D+w$ by Lemma 6.9. This means if there is at least one match then the cost is at most:

$$E = |F_i|n^b(2k^2 - k(k+1)/2) + \sum_{i\in[1,k]} 2np_i + 3n(D+w) - w.$$

What if there are no $(a(k-1)+bk)$-OVs? Well, any valid alignment (where we skip no characters and have the same size of overhangs) will cost at least $E+w$. If we don't align gadgets then at some point we are skipping 5^q symbols, this could potentially allow us to improve our edit distance by $2|IED'_i| + p$, however we set $q = 50p + 500|IED'_i|$ in Lemma 6.9. These skipped symbols increase the cost due to the unmatchable characters. When we foreshorten our string by skipping these 5^q symbols we sill need to pay the cost in the unmatchable characters as deletions (instead of substitutions) but we also need to pay for the deletion of the 5^q characters. So, if there is no valid alignment our cost is at least $E+w$, in fact it is exactly $E+w$.

The length of the generated strings is $O(n^b|IED'_i|) = O(n^b n^a d)$. Because $d = n^{o(1)}$ we can simplify this to $O(n^{b+a+o(1)})$. ∎

Next, we show that the strings we produced compress well.

LEMMA 6.11. *Let k, a, b be constant integers. Let $d = n^{o(1)}$. Given the k strings $EDOV_i(a,b,k,L)$ defined in Lemma 6.3 the size of the compression is $O(n^{b+o(1)} + n^{1+o(1)})$.*

Proof. We want to represent $O(n^b)$ instances of EED' and IED' gadgets. By Lemma 6.9 we have there is an SLP to represent these of size $O(n^b d + nd + n^b a \lg(n))$. This can be simplified to $n^{b+o(1)}n^{1+o(1)}$.

We additionally want to represent the unmatchable characters. These have a total length of $n^{b+a+o(1)}$, so there is an SLP to represent these of size $(b+a+o(1))\lg(n) = n^{o(1)}$.

So the total size of the SLP is $n^{b+o(1)}n^{1+o(1)}$. ∎

6.14 Putting it all Together Now that we have proven the above lemmas, we can prove our desired result.

REMINDER OF THEOREM 6.2. *If the k'-OV hypothesis is true for all constants k', then for all constant $\epsilon \in (0,1]$ grammar-compressed k-median edit distance requires $\left(M^{k-1}m\right)^{1-o(1)}$ time when the alphabet size is $|\Sigma| = \Theta(k)$ and $m = M^{\epsilon\pm o(1)}$. Here, M and m denote the total uncompressed and compressed length of the k input strings respectively.*

Proof. We will use Lemma 6.10 and Lemma 6.11. Given an instance of $(bk+a(k-1))$-OV we can produce strings $EDOV_1(a,b,k,L), \ldots, EDOV_k(a,b,k,L)$ such that they have length $M = n^{a+b+o(1)}$ and $m = n^{b+o(1)} + n^{1+o(1)} = n^{b+o(1)}$ when $b \geq 1$.

Our alphabet is of size $|\Sigma| = O(k)$, so this lower bound applies as long as the size of the alphabet is $\Theta(k)$.

Now note that $M^{k-1}m = n^{(k-1)a+(k-1)b+b} = n^{(k-1)a+kb}$. So, an algorithm that runs in $(M^{k-1}m)^{1-\epsilon}$ time $\epsilon > 0$ implies an algorithm for $(bk+a(k-1))$-OV that violates our assumption. Thus, k-edit distance requires $\left(M^{k-1}m\right)^{1-o(1)}$ time given the assumption on $(bk+a(k-1))$-OV.

Now we consider a contradiction to our theorem statement. There would be an algorithm running in $(M^{k-1}m)^{1-\gamma}$ time to solve grammar compressed k-median edit distance when $m = M^{\epsilon\pm o(1)}$ and $\epsilon \in (0,1]$. In the easiest case we can pick an a, b such that $b/(a+b) = \epsilon$, in this case we are done. For irrational ϵ we need to approximate and then pad the strings. Choose an a and b such that $\epsilon \leq b/(a+b) < \epsilon(1+\gamma/2)$. Such a, b exist that are in $O(\frac{1}{\epsilon\gamma})$. We add a new character 3 to our alphabet. Let $S'_i = S_i 3^x$, where we will set $x \in [M, M^2]$ later. Note that $\delta_E(S_1, \ldots, S_k) = \delta_E(S'_1, \ldots, S'_k)$. Note that the compression of these strings is $m' = m + \lg(x) = \Theta(m)$ where as the length is $M' = M + x = \Theta(x)$. Choose $x = m^{1/\epsilon\pm o(1)} = M^{b/(a+b)\cdot 1/\epsilon}$. So now $m' = M'^{1/\epsilon\pm o(1)}$. Note that $1 \leq b/(a+b)\cdot 1/\epsilon \leq (1+\gamma/2)$. Now consider running the claimed fast algorithm on our new $S'_1, \ldots, S'_k$ instance. The running time is

$$(M'^{(k-1)}m')^{1-\gamma} = O((M^{(1+\gamma/2)(k-1)}m)^{1-\gamma-o(1)}).$$

This running time can be simplified to $O(M^{(k-1)}m)^{(1+\gamma/2)(1-\gamma-o(1))}$. Note that $(1+\gamma/2)(1-\gamma-o(1))$ is less than $1-o(1)$. This algorithm violates the lower bound for the original $S_1, \ldots, S_k$ instance. This is a contradiction.

So any algorithm running in $(M^{k-1}m)^{1-\gamma}$ time to solve grammar compressed k-median edit distance when $m = M^{\epsilon \pm o(1)}$ and $\epsilon \in (0,1]$ violates k'-OV for some k' that depends on ϵ, γ. This implies our theorem statement. □

Finally we will apply this same lower bound to k-center edit distance using a reduction from [HBGT20].

6.15 k-Center Edit Distance In Section 3 of [HBGT20] they present a reduction from median k-edit distance to k-center distance [HBGT20]. We will restate their reduction here.

Say we are given a k-median edit distance instance with k strings $X_1, \ldots, X_k$ where $|X_i| = N$. Then, as [HBGT20] suggest, construct the following strings:

$$\begin{aligned} Y_1 &= X_1\ \$^N\ X_2\ \$^N\ \cdots\ \$^N\ X_{k-1}\ \$^N\ X_k \\ Y_2 &= X_2\ \$^N\ X_3\ \$^N\ \cdots\ \$^N\ X_k\ \ \$^N\ X_1 \\ &\vdots \\ Y_k &= X_k\ \$^N\ X_1\ \$^N\ \cdots\ \$^N\ X_{k-2}\ \$^N\ X_{k-1} \end{aligned}$$

Claim of Section 3 in [HBGT20]: $\delta_{CE}(Y_1, Y_2, \ldots, Y_k) = \delta_E(X_1, X_2, \ldots, X_k)$.

As a quick intuition for this claim, we have to match the $\$^N$ sections. First note that we can achieve this bound by taking a string T which is one of the median edit distance minimizing strings of the X_i and creating a center string for the Y_i strings $C = T\ \$^N\ T\ \$^N\ \ldots\ \$^N\ T$. Now note that the distance to this string from all Y_i is $\delta_E(X_1, X_2, \ldots, X_k)$. Thus, $\delta_{CE}(Y_1, Y_2, \ldots, Y_k) \leq \delta_E(X_1, X_2, \ldots, X_k)$. For the other side of the inequality note that the center string C should have shape $C = T_1\ \$^N\ T_2\ \$^N\ \ldots\ \$^N\ T_k$ for some strings $T_1, \ldots, T_k$. Now note that for all j

$$\sum_{i \in [1,k]} \delta_E(T_j, X_i) \geq \delta_E(X_1, X_2, \ldots, X_k).$$

Because, the k-median edit distance minimizes this sum over all possible strings, and T_j is simply an instantiation of a string. Now note that

$$\sum_{j \in [1,k]} \sum_{i \in [1,k]} \delta_E(T_j, X_i) \geq k\delta_E(X_1, X_2, \ldots, X_k),$$

which implies

$$\sum_{j \in [1,k]} \delta_E(Y_j, C) \geq k\delta_E(X_1, X_2, \ldots, X_k).$$

So, the max over all j of $\delta_E(Y_j, C) \geq \delta_E(X_1, X_2, \ldots, X_k)$. This is the definition of the center edit distance, so we have shown both sides of the inequality.

THEOREM 6.8. *We are given k strings of length M with a SLP of size m. The k-center-edit-distance problem on these strings requires $\left(M^{k-1}m\right)^{1-o(1)}$ time if SETH is true.*

Proof. By Theorem 6.2, given k strings, $X_1, \ldots, X_k$, of length N which all compress to length n, k-edit distance requires $\left(N^{k-1}n\right)^{1-o(1)}$ time if SETH is true.

We use the transformation of [HBGT20] and produce strings $Y_1, \ldots, Y_k$. These strings have length $M = kN$ and an SLP of size $m = kn + k + \lg(N)$. As a result $M^{k-1}m = O\left(N^{k-1}(n + \lg(N))\right)$. Thus, an algorithm that runs in $\left(M^{k-1}m\right)^{1-\epsilon}$ time for k-center edit distance for some constant $\epsilon > 0$ implies a violation of SETH. Thus, k-center edit distance requires $\left(M^{k-1}m\right)^{1-o(1)}$ time. □

7 Hamming Distance and Beyond

Given k equal-length strings $X_1, \ldots, X_k$ with $X_i \in \Sigma_i^N$, we define a string $X = \bigotimes_{i=1}^k X_i \in (\bigtimes_{i=1}^k \Sigma_i)^N$ with $X[j] = (X_1[j], \ldots, X_k[j])$ for $j \in [1\,..\,N]$. In this section, we show that if each string X_i can be represented using a straight-line program of size n_i, then X can be represented using a straight-line program of size $O((\prod_{i=1}^k n_i)^{1/k} N^{1-1/k})$. Next, we apply this construction for computing Hamming distance of two grammar-compressed strings and propose several generalizations for $k = O(1)$ grammar-compressed strings.

PROPOSITION 7.1. *Given $k = O(1)$ straight-line programs $\mathbf{G}_i$ of sizes n_i representing strings X_i of the same length $N > 0$, a straight-line program $\mathbf{G}$ of size $O((\prod_{i=1}^k n_i)^{1/k} N^{1-1/k})$ representing $X = \bigotimes_{i=1}^k X_i$ can be constructed in time $O((\prod_{i=1}^k n_i)^{1/k} N^{1-1/k})$.*

Proof. We proceed based on a threshold $\tau \in [1\,..\,N]$ to be fixed later. For each grammar $\mathbf{G}_i$, we first use Lemma 3.3 to derive grammars $\mathbf{G}_i^+$ and $\mathbf{G}_i^P$ of size $O(n_i)$.

Next, we consider *relevant tuples* $\mathbf{F} = (F_1, \ldots, F_k)$ such that:

- each F_i is a fragment of $\exp(A_i)$ for a symbol A_i of $\mathbf{G}_i^+$ satisfying $|A_i| \le \tau$,
- $|F_1| = \cdots = |F_k|$,
- there exists $i_p \in [1\,..\,k]$ such that F_{i_p} is a prefix of $\exp(A_{i_p})$,
- there exists $i_s \in [1\,..\,k]$ such that F_{i_s} is a suffix of $\exp(A_{i_s})$.

The number of relevant tuples does not exceed $\tau^{k-1} \cdot k \cdot \prod_{i=1}^k n_i$, because each $\mathbf{F}$ is uniquely determined by the choices of symbols A_i, the choice of i_p, and the starting positions of F_i in $\exp(A_i)$ for $i \neq i_p$. (The common length $|F_1| = \cdots = |F_k|$ is uniquely determined due to the constraint that at least one F_i is a suffix of $\exp(A_i)$.)

For each relevant tuple $\mathbf{F}$, we add to $\mathbf{G}$ a symbol $A_{\mathbf{F}}$ aiming at $\exp(A_{\mathbf{F}}) = \bigotimes_{i=1}^k F_i$. The symbols $A_{\mathbf{F}}$ are ordered consistently with the lexicographic order of tuples $(A_1, \ldots, A_k)$ based on the order of symbols A_i within each grammar $\mathbf{G}_i$.

If each A_i is a terminal of $\mathbf{G}_i$, then $F_i = \exp(A_i) = A_i$ holds for each i, and we set $A_{\mathbf{F}} = (A_1, \ldots, A_k)$ to be a terminal of $\mathbf{G}$. Otherwise, we set $A_{\mathbf{F}}$ to be a non-terminal, and we need to specify $\mathsf{rhs}(A_{\mathbf{F}})$. For this, let us fix an arbitrary index j such that A_j is a non-terminal of $\mathbf{G}_j$ and $A_j \to A'_j A''_j$. We then consider three cases:

1. F_j is contained within the prefix $\exp(A'_j)$ of $\exp(A_j)$. In this case, we set $A_{\mathbf{F}} \to A_{\mathbf{F}'}$, where F'_j is the fragment of $\exp(A'_j)$ corresponding to F_j and $F'_i = F_i$ for $i \neq j$. Note that F_j cannot be a suffix of $\exp(A_j)$ and, if F_j is a prefix of $\exp(A_j)$, then F'_j is a prefix of $\exp(A'_j)$. Thus, $\mathbf{F}'$ is a relevant tuple.
2. F_j is contained within the suffix $\exp(A''_j)$ of $\exp(A_j)$. In this case, we set $A_{\mathbf{F}} \to A_{\mathbf{F}''}$, where F''_j is the fragment of $\exp(A''_j)$ corresponding to F_j and $F''_i = F_i$ for $i \neq j$. Note that F_j cannot be a prefix of $\exp(A_j)$ and, if F_j is a suffix of $\exp(A_j)$, then F''_j is a suffix of $\exp(A''_j)$. Thus, $\mathbf{F}''$ is a relevant tuple.
3. F_j overlaps both the prefix $\exp(A'_j)$ and the suffix $\exp(A''_j)$ of $\exp(A_j)$. In this case, we set $A_{\mathbf{F}} \to A_{\mathbf{F}'} A_{\mathbf{F}''}$, where F'_j is the suffix of $\exp(A'_j)$ overlapping F_j, F''_j is the prefix of $\exp(A''_j)$ overlapping F_j, and for every $j \neq i$ we have $F_i = F'_i F''_i$ with $|F'_i| = |F'_j|$ and $|F''_i| = |F''_j|$.
   Note that F'_j is a suffix of $\exp(A'_j)$ and F''_j is a prefix of $\exp(A''_j)$. Moreover, if F_j is a prefix of $\exp(A_j)$, then F'_j is a prefix of $\exp(A'_j)$, and if F_j is a suffix of $\exp(A_j)$, then F''_j is a suffix of $\exp(A''_j)$. Finally, for $i \neq j$, if F_i is a prefix of $\exp(A_i)$, then F'_i is a prefix of $\exp(A_i)$, and if F_i is a suffix of $\exp(A_i)$, then F''_i is a suffix of $\exp(A_i)$. Thus, both $\mathbf{F}'$ and $\mathbf{F}''$ are relevant tuples.

Next, for each i, we interpret the string P_i generated by $\mathbf{G}_i^P$ as a decomposition of X_i into $|P_i| = O(\frac{N}{\tau})$ phrases. Each phrase is of the form $\exp(A)$ for a symbol A of $\mathbf{G}_i^+$ satisfying $|A| \le \tau$. Let B_i be the set of phrase boundaries of this decomposition of X_i (i.e., $B_i = \{|\exp(P_i[1\,..\,j])| : i \in [0\,..\,|P_i|]\}$), and let $B = \bigcup_{i=1}^k B_i$.

For each string X_i, let us construct another partition $X_i = X_{i,1} \circ \cdots \circ X_{i,r}$ with phrase boundaries B (if $0 = b_0 < \cdots < b_r = N$ are the elements of B, then $X_{i,j} = X_i(b_{j-1}\,..\,b_j]$). Since $B_i \subseteq B$, each phrase $X_{i,j}$ can be represented as a fragment of $\exp(A_{i,j})$ for a symbol $A_{i,j}$ of $\mathbf{G}_i^+$ satisfying $|A_{i,j}| \le \tau$. Moreover, for each j, there exists $i_p \in [1\,..\,k]$ such that $X_{i_p,j}$ is a prefix of $\exp(A_{i_p,j})$ and $i_s \in [1\,..\,k]$ such that $X_{i_s,j}$ is a suffix of $\exp(A_{i_s,j})$. (This is because $b_{j-1} \in B_{i_p}$ and $b_j \in B_{i_s}$ holds for some i_p and i_s.) Hence, for each $j \in [1\,..\,r]$, there exists a relevant tuple $\mathbf{F}_j = (X_{i,j})_{i=1}^k$. Thus, it suffices to add to $\mathbf{G}$ a starting symbol $S \to \bigcirc_{j=1}^r A_{\mathbf{F}_j}$.

Due to the assumption that $k = O(1)$, the total running time and the size $|\mathbf{G}|$ are both $O(\frac{N}{\tau} + \tau^{k-1} \prod_{i=1}^k n_i)$. Optimizing for $\tau \in [1\,..\,N]$, this becomes $O(\prod_{i=1}^k n_i + (\prod_{i=1}^k n_i)^{1/k} N^{1-1/k})$. If the first term dominates, then $\prod_{i=1}^k n_i > N$. However, a trivial $O(N)$-size straight-line program representing X can be constructed in $O(N)$ time

by decompressing each string X_i. Thus, we can always construct a straight-line program representing X in time $O((\prod_{i=1}^{k} n_i)^{1/k} N^{1-1/k})$. ∎

COROLLARY 7.1. *Given $k = O(1)$ straight-line programs $\mathbf{G}_i$ of size n_i representing strings $X_i \in \Sigma_i^N$ of the same length $N > 0$ and a function $\delta : \bigtimes_{i=1}^{k} \Sigma_i \to \mathbb{R}$ that can be evaluated in $O(1)$ time, the value $\delta(X_1, \ldots, X_k) := \sum_{j=1}^{N} \delta(X_1[j], \ldots, X_k[j])$ can be computed in $O((\prod_{i=1}^{k} n_i)^{1/k} N^{1-1/k})$ time.*

Proof. Let $X = \bigotimes_{i=1}^{k} X_i$ and let $\mathbf{G}$ be a straight-line program representing X constructed using Proposition 7.1. For each symbol A of $\mathbf{G}$, we compute a value $\delta(A)$ defined as $\sum_{j=1}^{|A|} \delta(\exp(A)[j])$. Note that if $A = (A_1, \ldots, A_k)$ is a non-terminal, then $\delta(A) = \delta(A_1, \ldots, A_k)$ can be evaluated in $O(1)$ time. Otherwise, if $A \to \bigcirc_{\ell=1}^{r} B_\ell$, then $\delta(A) = \sum_{\ell=1}^{r} \delta(B_\ell)$, so $\delta(A)$ can be computed in $O(|\mathsf{rhs}(A)|)$ time. Consequently, constructing $\mathbf{G}$ and computing $\delta(A)$ for every symbol A of $\mathbf{G}$ costs $O((\prod_{i=1}^{k} n_i)^{1/k} N^{1-1/k})$ time in total. This allows retrieving $\delta(X_1, \ldots, X_k)$ as the value $\delta(S)$ for the starting symbol S of $\mathbf{G}$. ∎

In particular, we can set $\delta = \delta_H$ for $k = 2$ (defined for characters x, y with $\delta_H(x, y) = 0$ if $x = y$ and $\delta_H(x, y) = 1$ if $x \neq y$). Possible generalizations to an arbitrary number of strings include the following two definitions of $\delta(x_1, \ldots, x_k)$ for characters $x_1, \ldots, x_k$:

- $\delta(x_1, \ldots, x_k) = 0$ if $x_1 = \cdots = x_k$ and $\delta(x_1, \ldots, x_k) = 1$ otherwise (the straightforward generalization).
- $\delta(x_1, \ldots, x_k) = \min_{i=1}^{k} \sum_{j=1}^{k} \delta_H(x_i, x_j)$ (the generalization corresponding to the median string problem).

In either case, Corollary 7.1 allows computing $\delta(X_1, \ldots, X_k)$ in $O((\prod_{i=1}^{k} n_i)^{1/k} N^{1-1/k})$ time.

8 Shift Distance: Lower Bound & Upper Bound

In this section we will explore a problem where we can get tight upper and lower bounds, but there is no efficiency to be gained by having compressible strings in your input. The problem is k-Shift Distance. When $k = 2$ this problem is (basically) equivalent to the Hamming distance substring problem mentioned in [ABBK17]. This problem is a natural extension of pattern matching. This problem asks, given a set of k strings, how can we best line them up to maximize the number of matched characters? So, the alignment that minimizes the Hamming Distance between all the strings. This problem was studied in the average-case for $k = 2$ by [AIKH13]. They called the problem "shift finding". We give the natural generalization of this problem to k strings and present upper bounds and lower bounds. We also present an approximation algorithm. We present these results in part because they give an example of a k-string comparison problem where there is no efficiency to be gained from having a compressible input.

The core of this section is showing tight upper and lower bounds for this problem of finding the ideal alignment of strings that minimizes Hamming distance. We show that in cyclic shift there is no advantage to be gained from compression. The upper and lower bounds are tight and unchanged even with compression. We are also able to use FFT to get a fast algorithm for the problem of finding the best alignment with respect to Hamming distance.

We will now re-state the definition of k-Shift Distance, with more commentary.

REMINDER OF DEFINITION k-SHIFT DISTANCE (k-SD). *We are given k strings as input: $X_1, \ldots, X_k$. These strings have characters from the alphabet Σ. Each string has length N and compresses via SLP to a length of n. For convenience of notation let $X_j[i]$ when $i \notin [0, n-1]$ refer to $X_j[i']$ where $i' \in [0, n-1]$ and $i' \equiv i \mod n$ (so we let indices "wrap around").*

We must return the best alignment of the k strings. The alignment where in the maximum number of locations all strings have the same symbol. We will give a precise definition below. Let $\hat{\Delta} = \max(\Delta_1, \ldots, \Delta_{k-1})$. And let $[[\cdot]]$ be an operator that turns True to a 1 and False to a 0.

$$k\text{-}SD(X_1, \ldots, X_k) = \max_{\Delta_1, \ldots, \Delta_{k-1} \in [0, N-1]} \left(\sum_{i=1}^{N} \left[\left[X_1[i + \Delta_1] = X_2[i + \Delta_2] = \cdots = X_{k-1}[i + \Delta_{k-1}] = X_k[i] \right] \right] \right)$$

So, we want the offsets such that the maximum number of characters are all shared between all k strings.

We will also define the term of the offset score. Given strings $X_1, \ldots, X_k$ and a particular set of deltas $\Delta_1, \ldots, \Delta_{k-1}$ we will call the value:

$$\sum_{i=1}^{N} [X_1[i+\Delta_1] = X_2[i+\Delta_2] = \cdots = X_{k-1}[i+\Delta_{k-1}] = X_k[i]]$$

the offset score of the strings $X_1, \ldots, X_k$ and the deltas $\Delta_1, \ldots, \Delta_{k-1}$.

8.1 The Algorithm We will use FFT to get a fast algorithm here. We start by showing how to do this when $k = 2$.

LEMMA 8.1. (FROM [ABBK17]) *There is an $O(|\Sigma| N \lg(N|\Sigma|))$ algorithm for 2-SD with an alphabet Σ (k-SD when $k = 2$).*

We can then generalize to k by making calls to 2-SD.

THEOREM 8.1. *There is an $O(|\Sigma| N^{k-1} \lg(|\Sigma| N))$ algorithm for k-Shift Distance.*

Proof. Let our k input strings be: $X_1, \ldots, X_k$, each of length N.

We are going to reduce k-SD with alphabet Σ and strings of length N to 2-SD with alphabet $\Sigma \cup \{@\}$ and strings of length N. First, we will try all N^{k-2} possible offsets $\Delta_2, \ldots, \Delta_{k-1}$. Now, for each of these we will Create a new string Y which will be a "merge" of the strings $X_2, \ldots, X_k$. The string Y will have length N. The i^{th} bit of Y is:

$$Y[i] = \begin{cases} X_k[i] & \text{if } [X_2[i+\Delta_2] = \cdots = X_{k-1}[i+\Delta_{k-1}] = X_k[i]] \\ @ & \text{else} \end{cases}.$$

If all the strings agree given our choice of offset we set it to the agreed character. Otherwise, we use the new special character @ which does not appear in X_1 (as @ is not in Σ). Note that we can produce Y in kN time, so over all offsets we take N^{k-1} time to produce the inputs X_1, Y.

Now, we will make N^{k-2} calls to 2-SD. Each call takes time $(|\Sigma|+1)N \lg(N(|\Sigma|+1))$ time. Thus, we take time $O(|\Sigma| N^{k-1} \lg(N|\Sigma|))$.

This gives a total time of $O(|\Sigma| N^{k-1} \lg(N|\Sigma|))$. ∎

8.2 The Lower Bound We will now show that we can produce k strings of length $N = n^a$ that compress to length an such that an algorithm that runs faster than $n^{(k-1)a-o(1)} = N^{k-1-o(1)}$ violates SETH and the $(k-1)a-$OV hypothesis. This will give a tight lower bound. Additionally, it says that strings that compress from length N to length N^ϵ do not have faster algorithms than those that don't compress.

To do this we will use the interleaving representation defined previously. Recall that we defined the interleaving version as:

$$\mathbf{String}_{\mathbf{I}_\ell}(L) = \bigcirc_{i=1}^{d} \left(\bigcirc_{j_1 \in [1,n], \ldots, j_\ell \in [1,n]} L[j_1][i] \cdot L[j_2][i] \cdot \cdots \cdot L[j_\ell][i] \right).$$

Recall that this is equivalent to

$$\mathbf{String}_{\mathbf{I}_\ell}(L) = \bigcirc_{i \in [1,d]} \bigcirc_{j \in [1,n^\ell]} \mathbf{List}(L)[j][i].$$

Finally, recall that in $\mathbf{String}_{\mathbf{I}_\ell}(L)$ the vector $\vec{v} = \mathbf{List}(L)[i]$ appears as bits $i, i+n^k, \ldots, i+(d-1)n^k$.

THEOREM 8.2. *Let a and k be constants. Let N be the input string length for k-SD.*

If the $(a(k-1))-$OV hypothesis holds then k-SD requires $N^{k-1-o(1)}$ time even when the strings compress down to length $m = N^{1/a+o(1)}$ with an alphabet of size $O(1)$.

Proof. We take as input a $(a(k-1))$-OV instance with $(a(k-1))$ lists of n vectors each. Each vector has length d and $d = n^{o(1)}$. Recall that the $(a(k-1))$-OV hypothesis states that $(a(k-1))$-OV requires $n^{a(k-1)-o(1)}$ time.

We will use four characters $0, 1, \%, @, *_1, \ldots, *_{2^k}$. The zero and ones will be used to signify the zeros and ones of the OV instance. The @ and $*_i$ symbols will be used to force alignment in a way that is easy to prove. We

note that one can almost certainly prove the same result with a smaller alphabet. However, allowing this larger alphabet makes our proof much easier.

Given a $(a(k-1))$-OV instance split the lists of vectors up into $k-1$ groups each with a lists of vectors. Call these groups of a lists $L_1, \ldots, L_{k-1}$. We are going to form strings $X_1, \ldots, X_{k-1}$ by slight alterations to $\mathbf{String}_{\mathbf{I}_a}(L_1), \ldots, \mathbf{String}_{\mathbf{I}_a}(L_1)$. The final string X_k will remain constant regardless of the input instance of OV.

Let $\hat{X}_i = \mathbf{String}_{\mathbf{I}_a}(L_i)$ then $i \in [1, k-1]$. Let $\hat{X}_k$ be a string of all zeros except in positions $0, n^a, \ldots, (d-1)n^a$. Like the other strings we give a total length of dn^a for $\hat{X}_k$. Note that by choosing an offset for each string from $\hat{X}_k$ we are effectively choosing one vector from each list $L_1, \ldots, L_{k-1}$ to align with the ones in $\hat{X}_k$. We want to design ways to right out the zeros and ones that simultaneously: (1) force alignment and (2) have the same value if there is at least one zero and a lower match value if they are all ones. If we can do this, then the best alignment will be picking the "most orthogonal" set of $k-1$ vectors, which will let us find if any vectors are fully orthogonal.

We will now design $h_{1,i}$ and $h_{0,i}$ which will have the property that the offset score of $h_{b_1,1}, h_{b_2,2}, \ldots, h_{b_k,k}$ with all deltas zero is 0 if all $b_i = 1$ and is 1 otherwise. We will consider all strings in $\{0,1\}^k$. Let H be all 2^k of those strings in sorted order, with the all ones string last. Let $H[j]$ be the j^{th} string in H note that $H[2^k]$ is the all ones string (we will one index this list). Now

$$h_{b,i}[j] = \begin{cases} 1 & \text{if } H[j][i] = b \text{ and } j \neq 2^k \\ 0 & \text{if } i < k \text{ and the above does not apply .} \\ \% & \text{else} \end{cases}$$

So we get strings of length 2^k. Note that $h_{b,i}$ for $i \in [1, k-1]$ uses only $0, 1$ symbols, however, $h_{b,k}$ uses only $0, \%$ symbols. If we are aligning $h_{b_1,1}, h_{b_2,2}, \ldots, h_{b_k,k}$ we are simply counting locations where they are all 1. This only occurs in the location that is associated with the string in H $b_1 b_2 \cdots b_k$, if it is not the all ones string. So, the offset score of $h_{b_1,1}, h_{b_2,2}, \ldots, h_{b_k,k}$ with all deltas zero is 0 if $b_i = 1$ for all i and is 1 otherwise, as desired.

We will now design $T_{1,i}$ and $T_{0,i}$ that will force alignment. Let $\bigcirc$ represent concatenation:

$$T_{b,i} = \bigcirc_{j=0}^{2^k} *_j \, h_{b,i}[j].$$

Note that this wrapper is just adding special characters that force alignment of the bits in $h_{b,i}$ by making the only way to match the $*_j$ characters also force an alignment of the $h_{b,i}[j]$ characters. Note that $|T_{b,i}| = 2 \cdot 2^k = \ell$. Note that the offset score of $T_{b_1,1}, T_{b_2,2}, \ldots, T_{b_k,k}$ with all deltas zero is 2^k if $b_i = 1$ for all i and is $1 + 2^k$.

Let $S_{1,i}$ be the representation of a 1 in string X_i. Let $S_{0,i}$ be the representation of a 0 in string X_i. We will set

$$S_{1,i} = @^{\ell} T_{1,i} \quad \text{and} \quad S_{0,i} = @^{\ell} T_{0,i}.$$

Note that this wrapper adds these @ characters which further enforce alignment. Note that the offset score of $S_{b_1,1}, S_{b_2,2}, \ldots, S_{b_k,k}$ with all deltas zero is $2^k + \ell$ if $b_i = 1$ for all i and is $1 + 2^k + \ell$ otherwise.

Correctness Now, we want to claim that one of the best alignments of $X_1, \ldots, X_k$ will have deltas that are multiples of $|S_{b,i}| = 2\ell$. That is, the best alignment will align these representations of single bits. Consider if $\Delta_i \mod 2\ell = f$. If $f \neq 0 \mod 2\ell$ then the $*_j$ symbols can't be aligned with those in X_k. Additionally, at most $\ell - j$ of the @ characters will be matched. Giving a maximum match of: $\ell - j + 2^k$ (even if every 0, 1, and % characters were matched, which is of course unrealistic, we can't match 0 characters as none appear in the X_k string). This is worse than the worst alignments when Δ_is are multiples of 2ℓ.

So, the best alignment has all Δ_i as multiples of 2ℓ. Thus, the alignment of $X_1, \ldots, X_k$ is an alignment of $|\hat{X}_i|$ $S_{b,i}$ gadgets. Each gadget promises to return $2^k + \ell$ if $b_i = 1$ for all i and is $1 + 2^k + \ell$ otherwise.

Now, note that given our construction of $\hat{X}_1, \ldots, \hat{X}_k$, if we choose a set of deltas $\Delta_i = 2\ell\delta_i$ we are effectively picking $k-1$ vectors and comparing them because of how we structured $\hat{X}_k$. So, if there are an orthogonal $k-1$ vectors which are orthogonal in our list representation (which corresponds to $a(k-1)$ vectors in the original OV instance) then we get a score of: $|\hat{X}_1|(1 + 2^k + \ell)$. Otherwise, we get a score at least one less than that. This shows our reduction will give the correct answer.

Time So with k strings of length n^a and a constant sized alphabet ($|\Sigma| = O(2^k)$) we can solve $(a(k+1))-$OV. Notably $N = n^{a+o(1)}$. So an algorithm running in faster than $N^{k-1-o(1)}$ time will violate the $(a(k+1))-$OV hypothesis. This fulfills the statement in the theorem.

Compression Now we will argue that these strings are compress-able with SLP. We will mostly be using the same structure as [ABBK17]. First we can build variables in our SLP for all of our base characters with $O(2^k)$ variables. Next we can build $@^\ell$ with $\lg(\ell) = O(k)$ variables. Next we can build all $S_{b,i}$ for all i and b with at most $O(k2^k)$ variables. Next, we want to build our longer strings.

Now we will use the recursive structure of $\mathbf{String}_{\mathbf{I}a}(L)$. Let

$$\mathbf{String}_{\mathbf{I}\ell}(L)^{[i]} = \bigcirc_{j_1\in[1,n]\ldots,j_\ell\in[1,n]} L[j_1][i]\cdot L[j_2][i]\cdot\cdots\cdot L[j_\ell][i].$$

Note that

$$\mathbf{String}_{\mathbf{I}a}(L) = \bigcirc_{i=1}^{d}\left(\mathbf{String}_{\mathbf{I}a}(L)^{[i]}\right).$$

We are just pulling out the part related to the i^{th} bit of every vector. Now note that

$$\mathbf{String}_{\mathbf{I}a}(L)^{[i]} = \bigcirc_{j\in[1,n]}\begin{cases}\mathbf{String}_{\mathbf{I}a-1}(L)^{[i]} & \text{if } L[j][i]=1\\ 0^{(n^{a-1})} & \text{if } L[j][i]=0\end{cases}.$$

Where $0^{(n^{a-1})}$ is n^{a-1} zeros in a row.

Note that we can make SLP variables for all $0^{(n^i)}$ strings for $i\in[1,a]$ with $a\lg(n^a) = a^2\lg(n)$ variables. Next note that given an SLP variable for $\mathbf{String}_{\mathbf{I}a-1}(L)^{[i]}$ we can add n variables and form $\mathbf{String}_{\mathbf{I}a}(L)^{[i]}$. It takes n variables to form $\mathbf{String}_{\mathbf{I}1}(L)^{[i]}$. So, with an SLP with $an + a^2\lg(n)$ variables we can represent $\mathbf{String}_{\mathbf{I}a}(L)^{[i]}$. So, with an SLP with $d(an + a^2\lg(n))$ variables we can represent $\mathbf{String}_{\mathbf{I}a}(L)$. To replace all zeros with $S_{0,i}$ and all ones with $S_{1,i}$ requires an additional $O(2^k)$ variables.

So, we can compress all of our strings with $O(d(n+\lg(n)))$ variables. Given our restrictions on d we can write this as $n^{1+o(1)}$. So our compression has length $m = n^{1+o(1)}$. Our input to our k-SD instance is $N = n^{a+o(1)}$. So $N^{1/a+o(1)} = n^{1+ao(1)} = n^{1+o(1)}$. Fulfilling the statement of the theorem. ∎

8.3 Approximation Algorithm Let the k-SD distance be $k(N - k\text{-}SD(X_1,\ldots,X_k))$. In other words, the k-SD distance is the total number of unmatched characters.

THEOREM 8.3. *There is an $O\left(|\Sigma|N^{\lceil(k-1)/\ell\rceil}\lg(|\Sigma|N)\right)$ time algorithm to get an ℓ approximation of the k-Shift Distance distance for any integer $\ell\geq 2$.*

Proof. Partitions the $k-1$ of the strings into ℓ groups $G'_1,\ldots,G'_\ell$ which each contain as close to $(k-1)/\ell$ strings as possible, the maximum number of strings in each group is $\lceil(k-1)/ell\rceil$. Now, take the final string, S_k and add it to all the sets to make new sets $G_1,\ldots,G_\ell$, now the maximum number of strings in each group is $\lceil(k-1)/ell+1\rceil$.

On each of these partitions run the algorithm for k-Shift Distance. The time for this is $O\left(\ell|\Sigma|N^{\lceil(k-1)/\ell+1\rceil-1}\lg(|\Sigma|N)\right)$ and ℓ is a constant. Now, using the value of k-Shift Distance we can compute the k-Shift Distance distance. Let the distances of the sets of strings in $G_1,\ldots,G_\ell$ be $\Delta_1,\ldots,\Delta_\ell$. Now, note that these call be framed as distances to the last string X_k. So, the distance of all these strings together is at most $\Delta_1+\cdots+\Delta_\ell$ and is at least $\max(\Delta_1,\ldots,\Delta_\ell)$. Finally, note that

$$1\leq\frac{\Delta_1+\cdots+\Delta_\ell}{\max(\Delta_1,\ldots,\Delta_\ell)}\leq\ell.$$

As a result there is an approximation factor of ℓ and a running time $O\left(|\Sigma|N^{\lceil(k-1)/\ell\rceil}\lg(|\Sigma|N)\right)$. ∎

9 On High-Dimensional Generalizations of DIST Matrices

Many of the crucial properties of DIST matrices derived in, e.g., [Tis15] used for two-string algorithms rely on the Monge property. For LCS, the Monge property is that given two strings X, Y and the alignment graph $\mathbf{G}_{X,y}$ then letting $d(u,v)$ be the longest path from u to v, we have $d(v_{0,i}, v_{|X|,j}) + d(v_{0,i-1}, v_{|X|,j+1}) \leq d(v_{0,i-1}, v_{|X|,j}) + d(v_{0,i}, v_{|X|,j+1})$. For example, in this paper we used the ability to take min-plus products of unit Monge matrices efficiently, and our use of the SMAWK algorithm was enabled by the Monge property.

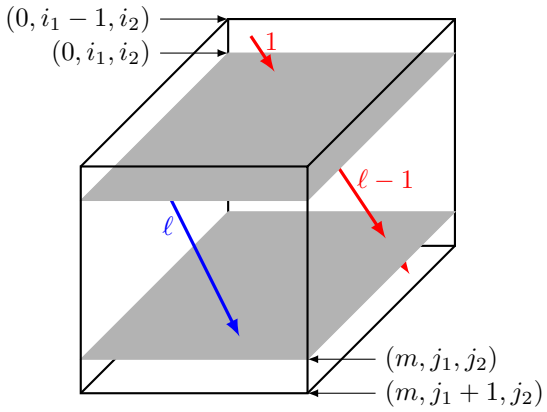
However, it appears no analogous property holds for even DIST "3-tensors", the three-string generalization of DIST matrices. Intuitively, this is because it is not possible to enforce that any path from v_1 to v_2 intersects any

path from v_3 to v_4 for four distinct vertices v_1, v_2, v_3, v_4, unlike in the two-dimensional alignment graph. We will use LCS as the metric for our examples here, but one can find similar examples for edit distance.

For example, let $A[i_1, i_2, j_1, j_2]$ be the longest path length from v_{0,i_1,i_2} to v_{n_1,j_1,j_2} in the three-dimensional alignment graph of three strings. An analog of the Monge property in three dimensions might be:

$$A(i_1, i_2, j_1, j_2) + A(i_1 - 1, i_2, j_1 + 1, j_2) \leq A(i_1 - 1, i_2, j_1, j_2) + A(i_1, i_2, j_1 + 1, j_2)$$

However, this does not seem true in general. Consider the following example, where there are two sets of length 1 edges. The first (in blue) has ℓ such edges, and is contained entirely between "layer" i_1 and j_1 of the DAG. The second (in red) has $\ell + 1$ edges, however two of these edges are outside the part of the DAG between $(0, i_1, i_2)$ and (m, j_1, j_2).



The two sets of length 1 edges are positioned such that one cannot use a "blue" and "red" edge in the same path. Now, we have that $A(i_1 - 1, i_2, j_1 + 1, j_2) = \ell + 1$ and all other terms in the above inequality are ℓ. So the above inequality would say $2\ell + 1 \leq 2\ell$, which is false. This can be generated by, e.g., the strings $X_1 = aabbb, X_2 = bbbaa, X_3 = baabb$; the LCS of the first two strings with $X_3[2\,..\,4], X_3[1\,..\,4], X_3[2\,..\,5]$ is aa, but the LCS of the first two strings with $X_3[1\,..\,5]$ is bbb.

While one can find other generalizations and even weakened versions of the Monge property which this example satisfies, for all the ones that we have considered there are three-string counterexamples that show they do not hold in general.

For example, the unit Monge property also says that given a DIST matrix, if we subtract every row from the next row and every column from the next column, we get a permutation matrix. In other words, each row and column only differs in behavior from the previous row/column by 1 entry. However, for DIST 3-tensors, consider the two-dimensional "slice" A for which $A[i, j]$ gives the path length between e.g. $(0, 0, i)$ and $(|X_1|, |X_2|, j)$. By looking at the DIST 3-tensors of even just three random strings of length roughly 100, we found that, e.g., for some sampled strings, A had a row that could be expressed as a linear function, but the next row of A was a piecewise linear function with six different pieces.

As another example, consider the following weaker "monotone" property: A is monotone if for any vector b, letting $m(i) = \arg\min_j A[i, j] + b[i]$ and choosing the lowest value of j to break ties, $m(i)$ is a monotonic function of i. This admits a divide and conquer algorithm for computing $\min_j A[i, j] + b[i]$ for all i in accesses to A near-linear in the number of i (as opposed to the SMAWK algorithm using linear accesses), a primitive that is useful in dynamic programming algorithms for two-string similarity. Informally, knowing $\arg\min_j A[i, j]$ lets us rule out a constant fraction of the possibilities for $\arg\min_j A[i', j]$ for $i' \neq i$. The 3-dimensional generalization of this primitive would be to compute $\arg\min_{i_1,i_2} A[i_1, i_2, j_1, j_2] + B[i_1, i_2]$ given access to entries of the DIST 3-tensor $A[i_1, i_2, j_1, j_2] = d(v_{0,i_1,i_2}, v_{|X_1|,j_1,j_2})$, and a matrix B. Put more simply, the rows of this slice have far less structural similarity to each other than the rows of a DIST matrix.

A weak generalization of the monotone property that would admit a similar divide and conquer algorithm for this problem is: knowing $i* = \arg\min_{i_1,i_2} A[i_1, i_2, j_1, j_2] + B[i_1, i_2]$ lets us eliminate possibilities for $\arg\min_{i_1,i_2} A[i_1, i_2, j_1', j_2'] + B[i_1, i_2]$ for (j_1', j_2') that are in a given "direction" from i^* if (j_1', j_2') is in a given "direction" from (j_1, j_2). Here, by in a given direction, we mean e.g. $j_1' \leq j_1$ and $j_2' \leq j_2$, or any of the four

possibilities given by reversing neither, one, or both of these inequalities. Unfortunately, even considering random strings of length 10, we found counterexamples to each of the variants of this property given by choosing any pair of directions to slot in to the definition.

10 Open Questions

We find many novel lower bounds and upper bounds in this paper. However, some of these are not tight. We give some open problems below whose resolution we think would be particularly interesting.

- For solving k-edit distance or k-LCS on strings where $k \geq 3$, we have a lower bound of $N^{k-1}n$ where N is the length of the strings and n is the size of the SLP. However, the best exact algorithms require $O(N^k)$ time. Can this gap be closed for any $k \geq 3$? Can this gap be closed for all constant k?
- There are no tight lower bounds for approximating k-LCS and k-edit distance. Can we give a tight lower bound?
- The lower bounds for *k-center* edit distance and the upper bounds do not match. Our lower bounds for *k-center* edit distance are the same as those for k-median edit distance. However, k-center edit distance has slower algorithms. For example in the uncompressed and exact case the k-center edit distance lower bounds are $\Omega(N^k)$ [HBGT20] but the best algorithm requires $\tilde{O}(N^{2k})$ time [NR05].

In general, the space of multiple string comparison seems under explored. We hope more work will happen in the space of algorithms and lower bounds for multiple string comparison. Specifically if there are efficient algorithms for the problem of comparing multiple strings with approximation for example, it will have significant impacts for multiple sequence alignment in biology.

References

[ABBK17] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017*, pages 192–203. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.26.

[ABV15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *56th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.

[AGMP13] Alexandr Andoni, Assaf Goldberger, Andrew McGregor, and Ely Porat. Homomorphic fingerprints under misalignments: sketching edit and shift distances. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing, STOC 2013*, pages 931–940. ACM, 2013. doi:10.1145/2488608.2488726.

[AIK08] Alexandr Andoni, Piotr Indyk, and Robert Krauthgamer. Earth mover distance over high-dimensional spaces. In Shang-Hua Teng, editor, *19th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008*, pages 343–352. SIAM, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347120.

[AIKH13] Alexandr Andoni, Piotr Indyk, Dina Katabi, and Haitham Hassanieh. Shift finding in sub-linear time. In Sanjeev Khanna, editor, *24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 457–465. SIAM, 2013. doi:10.1137/1.9781611973105.33.

[AKM+87] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987. doi:10.1007/BF01840359.

[AKO10] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2010*, pages 377–386. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.43.

[AN20] 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*. IEEE Computer Society, 2020. doi:10.1109/focs46700.2020.00096.

[AO12] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM Journal on Computing*, 41(6):1635–1648, 2012. doi:10.1137/090767182.

[BDY16] Bonnie Berger, Noah M Daniels, and Y William Yu. Computational biology in the 21st century: Scaling with compressive algorithms. *Communications of the ACM*, 59(8):72–80, 2016. doi:10.1145/2957324.

[BEK+03] Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. A sublinear algorithm for weakly approximating edit distance. In Lawrence L. Larmore and Michel X.

Goemans, editors, *35th Annual ACM Symposium on Theory of Computing, STOC 2003*, pages 316–324. ACM, 2003. doi:10.1145/780542.780590.

[BES06] Tugkan Batu, Funda Ergün, and Süleyman Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, pages 792–801. ACM Press, 2006. URL: http://dl.acm.org/citation.cfm?id=1109557.1109644.

[BJKK04] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *45th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2004*, pages 550–559. IEEE Computer Society, 2004. doi:10.1109/FOCS.2004.14.

[BK15] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *56th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.

[BPS13] Bonnie Berger, Jian Peng, and Mona Singh. Computational solutions for omics data. *Nature Reviews Genetics*, 14(5):333–346, 2013. doi:10.1038/nrg3433.

[BR20] Joshua Brakensiek and Aviad Rubinstein. Constant-factor approximation of near-linear edit distance in near-linear time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *52nd Annual ACM Symposium on Theory of Computing, STOC 2020*, pages 685–698. ACM, 2020. doi:10.1145/3357713.3384282.

[BW94] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

[BWK19] Karl Bringmann, Philip Wellnitz, and Marvin Künnemann. Few matches or almost periodicity: Faster pattern matching with mismatches in compressed texts. In Timothy M. Chan, editor, *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1126–1145. SIAM, 2019. doi:10.1137/1.9781611975482.69.

[CDG+18] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 979–990. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00096.

[CKW20] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. IEEE Computer Society, 2020. doi:10.1109/focs46700.2020.00095.

[Gag94] Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994. URL: https://dl.acm.org/doi/10.5555/177910.177914.

[Gaw12] Paweł Gawrychowski. Faster algorithm for computing the edit distance between SLP-compressed strings. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *19th International Symposium on String Processing and Information Retrieval, SPIRE 2012*, volume 7608 of *LNCS*, pages 229–236. Springer, 2012. doi:10.1007/978-3-642-34109-0_24.

[GKK+20] Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Przemyslaw Uznański. Improved circular k-mismatch sketches. In Jarosław Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020*, volume 176 of *LIPIcs*, pages 46:1–46:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.APPROX/RANDOM.2020.46.

[GRS20] Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. Does preprocessing help in fast sequence comparisons? In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *52nd Annual ACM Symposium on Theory of Computing, STOC 2020*, pages 657–670. ACM, 2020. doi:10.1145/3357713.3384300.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

[GWH19] Dan Greenfield, Vaughan Wittorff, and Michael Hultner. The importance of data compression in the field of genomics. *IEEE Pulse*, 10(2):20–23, 2019. doi:10.1109/mpuls.2019.2899747.

[HBGT20] Gary Hoppenworth, Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. The fine-grained complexity of median and center string problems under edit distance. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020*, volume 173 of *LIPIcs*, pages 61:1–61:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.61.

[HLLW09] Danny Hermelin, Gad M. Landau, Shir Landau, and Oren Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009*, volume 3 of *LIPIcs*, pages 529–540. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2009. doi:10.4230/LIPIcs.STACS.2009.1804.

[HLLW13] Danny Hermelin, Gad M. Landau, Shir Landau, and Oren Weimann. Unified compression-based acceleration of edit-distance computation. *Algorithmica*, 65(2):339–353, 2013. doi:10.1007/s00453-011-9590-6.

[HPWO19] Mikel Hernaez, Dmitri Pavlichin, Tsachy Weissman, and Idoia Ochoa. Genomic data compression. *Annual Review of Biomedical Data Science*, 2:19–37, 2019. doi:10.1146/annurev-biodatasci-072018-021229.

[HSSS19] MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating LCS in linear time: Beating the $\sqrt{n}$ barrier. In Timothy M. Chan, editor, *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1181–1200. SIAM, SIAM, 2019. doi:10.1137/1.9781611975482.72.

[I17] Tomohiro I. Longest Common Extensions with recompression. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

[Jeż15] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

[KK20] Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. IEEE Computer Society, 2020. doi:10.1109/focs46700.2020.00097.

[KP18] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: String attractors. In Monika Henzinger, editor, *50th Annual ACM Symposium on Theory of Computing, STOC 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

[KS20] Michal Koucký and Michael E. Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *52nd Annual ACM Symposium on Theory of Computing, STOC 2020*, pages 699–712. ACM, 2020. doi:10.1145/3357713.3384307.

[LM00] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000. doi:10.1109/5.892708.

[LMS98] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

[Loh12] Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012. doi:10.1515/gcc-2012-0016.

[LV88] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

[MSU97] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

[NMN14] Richard Van Noorden, Brendan Maher, and Regina Nuzzo. The top 100 papers. *Nature*, 514(7524):550–553, 2014. doi:10.1038/514550a.

[NR05] François Nicolas and Eric Rivals. Hardness results for the center and median string problems under the weighted and unweighted edit distances. *Journal of Discrete Algorithms*, 3(2-4):390–415, 2005. doi:10.1016/j.jda.2004.08.015.

[NW97] Craig G. Nevill-Manning and Ian H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–116, 1997. doi:10.1093/comjnl/40.2_and_3.103.

[RPE81] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981. doi:10.1145/322234.322237.

[RS20] Aviad Rubinstein and Zhao Song. Reducing approximate longest common subsequence to approximate edit distance. In Shuchi Chawla, editor, *31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 1591–1600. SIAM, 2020. doi:10.1137/1.9781611975994.98.

[RSSS19] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for LCS and LIS with truly improved running times. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 1121–1145. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00071.

[Ryt03] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

[THG94] 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(22):4673–4680, 1994. doi:10.1093/nar/22.22.4673.

[Tis09] Alexander Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158(5):759–769, 2009. doi:10.1007/s10958-009-9396-0.

[Tis15] Alexander Tiskin. Fast distance multiplication of unit-monge matrices. *Algorithmica*, 71(4):859–888, 2015. doi:10.1007/s00453-013-9830-z.

[Vas18] Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians, ICM 2018*, volume 3, pages 3431–3472. World Scientific, 2018. doi:10.1142/9789813272880_0188.

[Wel84] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984. doi:10.1109/MC.1984.1659158.

[Wil07] Ryan Williams. *Algorithms and resource requirements for fundamental problems*. PhD thesis, Carnegie Mellon University, 2007. URL: http://ra.adm.cs.cmu.edu/anon/2007/CMU-CS-07-147.pdf.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.