Bounded Edit Distance: Optimal Static and Dynamic Algorithms for Small Integer Weights*

Egor Gorbachev†
Saarland University & Max Planck Institute for Informatics
Saarland Informatics Campus, Saarbrücken, Germany
egorbachev@cs.uni-saarland.de

Tomasz Kociumaka‡
Max Planck Institute for Informatics
Saarland Informatics Campus, Saarbrücken, Germany
tomasz.kociumaka@mpi-inf.mpg.de

Abstract

The *edit distance* (also known as the Levenshtein distance) of two strings is the minimum number of character insertions, deletions, and substitutions needed to transform one string into the other. The textbook algorithm determines the edit distance of two length-n strings in $O(n^2)$ time, and one of the foundational results of fine-grained complexity is that any polynomial-factor improvement upon this quadratic runtime would violate the Orthogonal Vectors Hypothesis. In the *bounded* version of the problem, where the complexity is parameterized by the value k of the edit distance, the classic algorithm of Landau and Vishkin [JCSS'88] achieves $O(n+k^2)$ time, which is optimal (up to sub-polynomial factors and conditioned on OVH) as a function of n and k.

While the Levenshtein distance is a fundamental theoretical notion, most practical applications use *weighted edit distance*, where the weight (cost) of each edit can be an arbitrary real number in $[1,\infty)$ that may depend on the edit type and the characters involved. Unfortunately, the Landau–Vishkin algorithm does not generalize to the weighted setting and, for many decades, a simple $O(nk)$-time dynamic programming procedure remained the state of the art for *bounded weighted edit distance*. Only recently, Das, Gilbert, Hajiaghayi, Kociumaka, and Saha [STOC'23] provided an $O(n+k^5)$-time algorithm; shortly afterward, Cassis, Kociumaka, and Wellnitz [FOCS'23] presented an $\widetilde{O}(n+\sqrt{nk^3})$-time solution (where $\widetilde{O}(\cdot)$ hides poly $\log n$ factors) and proved this runtime optimal for $\sqrt{n} \le k \le n$ (up to sub-polynomial factors and conditioned on the All-Pairs Shortest Paths Hypothesis).

Notably, the hard instances constructed to establish the underlying conditional lower bound use fractional weights with large denominators, reaching poly(n), which stands in contrast to weight functions used in practice (e.g., in bioinformatics) that, after normalization, typically attain small integer values. Our first main contribution is a surprising discovery that the $\widetilde{O}(n+k^2)$ running time of the Landau–Vishkin algorithm can be recovered if the weights are small integers (e.g., if some specific weight function is fixed in a given application). In general, our solution takes $\widetilde{O}(n+\min\{W,\sqrt{k}\}\cdot k^2)$ time for integer weights not exceeding a threshold W. Despite matching time complexities, we do not use the Landau–Vishkin framework; instead, we build upon the recent techniques for arbitrary weights. For this, we exploit extra structure following from integer weights and avoid further bottlenecks using several novel ideas to give faster and more robust implementations of multiple steps of the previous approach.

Next, we shift focus to the *dynamic* version of the *unweighted* edit distance problem, which asks to maintain the edit distance of two strings that change dynamically, with each update modeled as a single edit (character insertion, deletion, or substitution). For many years, the best approach for dynamic edit distance combined the Landau–Vishkin algorithm with a dynamic strings implementation supporting efficient substring equality queries, such as one by Mehlhorn, Sundar, and Uhrig [SODA'94]; the resulting solution supports updates in $\widetilde{O}(k^2)$ time. Recently, Charalampopoulos, Kociumaka, and Mozes [CPM'20] observed that a framework of Tiskin [SODA'10] yields a dynamic algorithm with an update time of $\widetilde{O}(n)$. This is optimal in terms of n: significantly faster updates would improve upon the static $O(n^2)$-time algorithm and violate OVH. With the state-of-the-art update time at $\widetilde{O}(\min\{n,k^2\})$, an exciting open question is whether $\widetilde{O}(k)$ update time is possible. Our second main contribution is an affirmative answer to this question: we present a deterministic dynamic algorithm that maintains the edit distance in $\widetilde{O}(k)$ worst-case time per update. Surprisingly, this result builds upon our new static solution and hence natively supports small integer weights: if they do not exceed a threshold W, the update time becomes $\widetilde{O}(W^2k)$.

CCS Concepts

• **Theory of computation** → **Pattern matching**; Divide and conquer.

Keywords

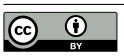
edit distance, dynamic algorithms, fine-grained complexity

ACM Reference Format:
Egor Gorbachev and Tomasz Kociumaka. 2025. Bounded Edit Distance: Optimal Static and Dynamic Algorithms for Small Integer Weights. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing (STOC '25), June 23–27, 2025, Prague, Czechia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3717823.3718168

1 Introduction

Quantifying similarity between strings (sequences, texts) is a crucial algorithmic task applicable across many domains, with the most profound role in bioinformatics and computational linguistics.

*The full version of the paper is available at https://arxiv.org/abs/2404.06401 [40].
†The work is part of the project TIPEA that has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No. 850979).
‡Partially funded by the Ministry of Education and Science of Bulgaria's support for INSAIT, Sofia University "St. Kliment Ohridski", as part of the Bulgarian National Roadmap for Research Infrastructure.



STOC '25, Prague, Czechia

ACM ISBN 979-8-4007-1510-5/25/06
https://doi.org/10.1145/3717823.3718168

The *edit distance* (Levenshtein distance [61]) is among the most popular measures of string (dis)similarity. For two strings X and Y, the edit distance $\mathrm{ed}(X, Y)$ is the minimum number of character insertions, deletions, and substitutions (jointly called *edits*) needed to transform X into Y. The textbook algorithm [67, 71, 77, 78] takes $O(n^2)$ time to compute the edit distance of two strings of length at most n. Despite substantial efforts, its runtime has been improved only by a poly-logarithmic factor [41, 63]. Fine-grained complexity provided an explanation for this quadratic barrier: any polynomial-factor improvement would violate the Orthogonal Vectors Hypothesis [1, 2, 9, 17] and thus also the Strong Exponential Time Hypothesis [45, 46].

An established way of circumventing this lower bound is to consider the *bounded* edit distance problem, where the running time is expressed in terms of not only the length n of the input strings but also the value k of the edit distance. Building upon the ideas of Ukkonen [76] and Myers [66], Landau and Vishkin [60] presented an elegant $O(n + k^2)$-time algorithm for bounded edit distance. The fine-grained hardness of the unbounded version prohibits any polynomial-factor improvements upon this runtime: a hypothetical $O(n + k^{2-\epsilon})$-time algorithm, even restricted to instances satisfying $k = \Theta(n^\kappa)$ for some constant $\frac{1}{2} < \kappa \le 1$, would violate the Orthogonal Vectors Hypothesis. Although bounded edit distance admits a decades-old optimal solution, the last few years brought numerous exciting developments across multiple models of computation. This includes sketching and streaming algorithms [11, 12, 48, 55, 58], sublinear-time approximation algorithms [13, 14, 36, 38, 56], algorithms for preprocessed strings [15, 39], algorithms for compressed input [30], and quantum algorithms [35], just to mention a few settings. Multiple papers have also studied generalizations of the bounded edit distance problem, including weighted edit distance [18, 25, 37], tree edit distance [4, 25, 26, 75], and Dyck edit distance [10, 25, 28, 29].

In this work, we provide conditionally optimal algorithms for bounded edit distance with *small integer weights* and settle the complexity of the *dynamic* version of the bounded edit distance problem.

1.1 Weighted Edit Distance

Although the theoretical research on the edit distance problem has predominantly focused on the *unweighted* Levenshein distance, most practical applications require a more general *weighted* edit distance, where each edit is associated with a cost depending on the edit type and the characters involved. Many early works [71, 72, 78] and application-oriented textbooks [42, 49, 62, 79] introduce edit distance already in the weighted variant. Formally, it can be conveniently defined using a weight function $w : \overline{\Sigma}^2 \to \mathbb{R}$, where $\overline{\Sigma} = \Sigma \cup \{\varepsilon\}$ denotes the alphabet extended with a special symbol ε representing the empty string (or the lack of a character). For every $a, b \in \Sigma$, the cost of inserting b is $w(\varepsilon, b)$, the cost of deleting a is $w(a, \varepsilon)$, and the cost of substituting a for b is $w(a, b)$. The weighted edit distance $\mathrm{ed}^w(X, Y)$ of two strings X and Y is the minimum total weight of edits transforming X into Y. Consistently with previous works, we assume that the weight function is normalized so that $w(a, a) = 0$ and $w(a, b) \ge 1$ hold for every $a, b \in \overline{\Sigma}$ with $a \ne b$.[1] As a result, $\mathrm{ed}^w(X, X) = 0$ and $\mathrm{ed}(X, Y) \le \mathrm{ed}^w(X, Y)$ hold for all strings $X, Y \in \Sigma^*$.

As shown in [71, 78], the textbook $O(n^2)$-time dynamic-programming algorithm natively supports arbitrary weights. A simple optimization, originating from [76], allows for an improved running time of $O(nk)$ if $k := \mathrm{ed}^w(X, Y)$ does not exceed n. For almost four decades, this remained the best running time for bounded weighted edit distance. Only in 2023, Das, Gilbert, Hajiaghayi, Kociumaka, and Saha [25] managed to improve upon the $O(nk)$ time, albeit only for $k \ll \sqrt[4]{n}$: their algorithm runs in $O(n + k^5)$ time. Soon afterward, Cassis, Kociumaka, and Wellnitz [18] developed an $\widetilde{O}(n + \sqrt{nk^3})$-time[2] algorithm; the runtime of this solution exceeds neither $\widetilde{O}(n + k^3)$ nor $\widetilde{O}(nk)$, and it interpolates smoothly between $\widetilde{O}(n)$ for $k = \sqrt[3]{n}$ and $\widetilde{O}(nk)$ for $k = n$. Unexpectedly, the running time of $\widetilde{O}(n + \sqrt{nk^3})$ is optimal for $\sqrt{n} \le k \le n$: any polynomial-factor improvement would violate the All-Pairs Shortest Paths Hypothesis [18]. This result provides a strict separation between the weighted and the unweighted variants of the bounded edit distance problem. Along with settling the complexity for $\sqrt[3]{n} \le k \le \sqrt{n}$, where the conditional lower bound degrades to $n + k^{2.5-o(1)}$, the most important open question posed in [18] is the following one:

Can the $\sqrt{nk^{3-o(1)}}$ lower bound be circumvented for some natural weight function classes?

Arguably, the weight functions devised to establish the lower bound in [18] have a rather atypical structure: in particular, they use fractional costs with large denominators (reaching $\Theta(n^c)$ for a constant $c \gg 1$). In contrast, weight functions arising in practice, such as those originating from the BLOSUM [44] and PAM [27] substitution matrices commonly used for amino-acids, are equivalent (after normalization) to weight functions with small integer values (below 30). Unfortunately, the literature does not provide specialized solutions for this case, and small integer values are already sufficient to violate the monotonicity property that the Landau–Vishkin algorithm [60] hinges on.[3] The first main contribution of our work is that, up to a modest polylogarithmic-factor overhead, the time complexity of the Landau–Vishkin algorithm (optimal under the Orthogonal Vectors Hypothesis) can be recovered for small integer weights. In the most basic version, our result reads as follows:

Theorem 1.1. *Fix an alphabet Σ and a weight function $w : \overline{\Sigma}^2 \to \mathbb{Z}_{\ge 0}$. Given strings $X, Y \in \Sigma^{\le n}$, the weighted edit distance $k := \mathrm{ed}^w(X, Y)$ can be computed in $O(n + k^2 \log^2 n)$ time.*

Remark 1.2. We note that $\mathbb{Z}_{\ge 0}$ in Theorem 1.1 can be replaced with $\mathbb{Q}_{\ge 0}$: In order to reduce to the integer case, it suffices to scale all the weights up by lowest common multiple of all the denominators. Since we assume the weight function to be fixed (rather than a part of the input), this scaling factor is constant, so the time complexity is not affected even though the value k grows.

[1] Further weight functions w' can be handled by setting $w'(a, b) = \alpha(a) + \beta(b) + \gamma \cdot w(a, b)$ for appropriate parameters $\alpha, \beta : \overline{\Sigma} \to \mathbb{R}$ and $\gamma \in \mathbb{R}_{>0}$. This normalization step does not change the set of optimal alignments.

[2] The $\widetilde{O}(\cdot)$ notation hides factors poly-logarithmic in the input size n, that is, $\log^c n$ for any constant c.

[3] As already observed in [25], we have $2 = \mathrm{ed}^w(\mathsf{ab}, \mathsf{c}) < \mathrm{ed}^w(\mathsf{a}, \varepsilon) = 3$ if $w(\mathsf{b}, \varepsilon) = w(\mathsf{a}, \mathsf{c}) = 1$ and $w(\mathsf{a}, \varepsilon) = 3$.

As discussed in the technical overview (Section 2), the algorithm behind Theorem 1.1 builds upon the approach of [18], with the better running time enabled by the extra structure arising from the small integer weights. Nevertheless, we also need several novel ideas of independent interest (applicable to arbitrary weights) to eliminate the bottlenecks that pop up as we improve the time complexity to $\widetilde{O}(n + k^2)$. The resulting algorithm is very robust: it seamlessly supports integer weight functions given as a part of the input (with a modest $O(W)$-factor overhead, where W is the maximum cost of a single edit), and it outputs not only the weighted edit distance, but also the underlying optimal alignment (an optimal sequence of edits transforming one string into the other).

Theorem 1.3. *Given strings $X, Y \in \Sigma^{\le n}$ and oracle access to a weight function $w : \overline{\Sigma}^2 \to [0..W]$,[4] the weighted edit distance $k := \mathrm{ed}^w(X, Y)$ can be computed in $O(n + W \cdot k^2 \log^2 n)$ time. The algorithm also outputs a w-optimal sequence of edits transforming X into Y.*

Even though the time complexity of Theorem 1.3 is optimal (up to sub-polynomial factors and conditioned on OVH) for $W = n^{o(1)}$, the algorithm of [18] for arbitrary weights becomes faster when W is large, e.g., $W \ge k$. In order to address this issue, we present a variant of our solution with an extra subroutine that lets us effectively cap the dependency on W at $O(\sqrt{k} \log n)$: the resulting runtime of $\widetilde{O}(n + k^{2.5})$ improves upon the upper bound of $\widetilde{O}(n + \sqrt{nk^3})$ from [18] if $k \ll \sqrt{n}$.

Theorem 1.4. *Given strings $X, Y \in \Sigma^{\le n}$ and oracle access to a weight function $w : \overline{\Sigma}^2 \to \mathbb{Z}_{\ge 0}$, the weighted edit distance $k := \mathrm{ed}^w(X, Y)$ can be computed in $O(n+k^{2.5} \log^3 n)$ time. The algorithm also outputs a w-optimal sequence of edits transforming X into Y.*

Remark 1.5. As a corollary of independent interest, for an arbitrary normalized weight function $w' : \overline{\Sigma}^2 \to \mathbb{R}_{\ge 0}$ and a real parameter $\epsilon \in (0, 1]$, a $(1+\epsilon)$-factor approximation of $k := \mathrm{ed}^{w'}(X, Y)$ can be computed in $O(n + (k/\epsilon)^{2.5} \log^3 n)$ time. For this, it suffices to use Theorem 1.4 with $w(a, b) := \lfloor w'(a, b)/\epsilon \rfloor$.

Related Results. The only weight function class systematically studied before this work consists of *uniform* weight functions that assign an integer weight W_{indel} to all insertions and deletions and another integer weight W_{sub} to all substitutions. Tiskin [73] observed that the $O(n+k^2)$ running time can be generalized from the unweighted case of $W_{\mathsf{indel}} = W_{\mathsf{sub}} = 1$ to $W_{\mathsf{indel}}, W_{\mathsf{sub}} = O(1)$. In a more fine-grained study, Goldenberg, Kociumaka, Krauthgamer, and Saha [37] adapted the Landau–Vishkin algorithm [60] so that it runs in $O(n + k^2/W_{\mathsf{indel}})$ time for any integers $W_{\mathsf{indel}}, W_{\mathsf{sub}} \ge 1$.

1.2 Dynamic Edit Distance

Dynamic algorithms capture a natural scenario when the input data changes frequently, and the solution needs to be maintained upon every update. Although the literature on dynamic algorithms covers primarily graph problems (see [43] for a recent survey), there is a large body of work on dynamic strings. The problems studied in this setting include testing equality between substrings [64], longest common prefix queries [5, 34, 64], text indexing [5, 33, 50, 68], approximate pattern matching [21, 22, 24], suffix array maintenance [7, 50], longest common substring [6, 19], longest increasing subsequence [32, 57, 65], Lempel–Ziv factorization [68], detection of repetitions [8], and last but not least edit distance [20, 54, 73] (see [53] for a survey talk).

A popular model of dynamic strings, and arguably the most natural one in the context of edit distance, is where each update is a single edit, i.e., a character insertion, deletion, or substitution in one of the input strings. A folklore dynamic edit-distance algorithm can be obtained from the Landau–Vishkin framework [60] using a dynamic strings implementation that can efficiently test equality between substrings. With the data structure of Mehlhorn, Sundar, and Uhrig [64], one can already achieve the worst-case update time of $\widetilde{O}(k^2)$. Modern optimized alternatives [34, 50] yield $O(k^2 \log n)$ update time with high probability and $O(k^2 \log^{1+o(1)} n)$ update time deterministically. Unfortunately, these results are not meaningful for $k \ge \sqrt{n}$: then, it is better to recompute the edit distance from scratch, in $O(n + k^2) = O(k^2)$ time, upon every update. In particular, it remained open if sub-quadratic update time can be achieved for the unbounded version of dynamic edit distance.

Early works contributed towards answering this question by studying a restricted setting allowing updates only at the endpoints of the maintained strings [47, 51, 59, 73]. The most general of these results is an algorithm by Tiskin [73] that works in $O(n)$ time per update subject to edits at both endpoints of both strings. More recently, Charalampopoulos, Kociumaka, and Mozes [20] applied Tiskin's toolbox [73, 74] in a dynamic edit distance algorithm that supports arbitrary updates in $O(n \log^2 n)$ time. Any significantly better update time of $O(n^{1-\epsilon})$ would immediately yield an $O(n^{2-\epsilon})$-time static algorithm and thus violate the Orthogonal Vectors Hypothesis. The fine-grained lower bound, however, does not prohibit improvements for $k \ll n$, and the state-of-the-art update time of $\widetilde{O}(\min\{n, k^2\})$ motivates the following tantalizing open question, explicitly posed in [53]:

Is there a dynamic edit distance algorithm that supports updates in $\widetilde{O}(k)$ time?

The second main result of our work is an affirmative answer to this question.

Theorem 1.6. *There exists a deterministic dynamic algorithm that maintains strings $X, Y \in \Sigma^*$ subject to edits and, upon every update, computes $k := \mathrm{ed}(X, Y)$ in $O(k \log^2 n)$ time, where $n := |X| + |Y|$. The algorithm can be initialized in $O(n \log^{o(1)} n + k^2 \log^2 n)$ time and, along with $\mathrm{ed}(X, Y)$, it also outputs an optimal sequence of edits transforming X into Y.*

We note that the fine-grained lower bounds for the (unbounded) static edit distance problem prohibit improving the update time to $\widetilde{O}(k^{1-\epsilon})$ for any $\epsilon > 0$, even for instances restricted to $k = \Theta(n^\kappa)$ for any constant $0 < \kappa \le 1$.[5]

[4] For $a, b \in \mathbb{R}$, we write $[a..b]$ for the set of all integers c satisfying $a \le c \le b$.

[5] To see this, consider a static edit distance instance $(\bar{X}, \bar{Y})$ with $\mathrm{ed}(\bar{X}, \bar{Y}) = \Theta(n^\kappa)$ and $|\bar{X}| + |\bar{Y}| = \Theta(n^\kappa)$. Initialize a dynamic edit distance algorithm with $X = Y = \mathtt{0}^n$ and perform $\Theta(n^\kappa)$ insertions so that $X = \mathtt{0}^n \bar{X}$ and $Y = \mathtt{0}^n \bar{Y}$. With an update time of $\widetilde{O}(k^{1-\epsilon})$, we could compute $\mathrm{ed}(\bar{X}, \bar{Y}) = \mathrm{ed}(X, Y)$ in $\widetilde{O}(n^\kappa \cdot n^{\kappa(1-\epsilon)}) = \widetilde{O}(n^{\kappa(2-\epsilon)})$ time, violating the Orthogonal Vectors Hypothesis. Considering multiple instances $(\bar{X}, \bar{Y})$, one can further prove that the $\widetilde{O}(k^{1-\epsilon})$ update time cannot be achieved even after $O(n^c)$-time initialization for an arbitrarily large constant c.

Notably, even though our solution is significantly more complicated than the state-of-the-art dynamic algorithm for unbounded edit distance [20], we do not incur any additional polylogarithmic factors in the update time. What is perhaps more surprising, our algorithm natively supports small integer weights and, unlike in the static case, we are not aware of any significantly simpler approach tailored for the unweighted version of the dynamic bounded edit distance problem.

Theorem 1.7. *Let $w : \overline{\Sigma}^2 \to [0..W]$ be a weight function supporting constant-time oracle access. There exists a deterministic dynamic algorithm that maintains strings $X, Y \in \Sigma^*$ subject to edits and, upon every update, computes $k := \mathrm{ed}^w(X, Y)$ in $O(W^2 k \log^2 n)$ time, where $n := |X| + |Y|$. The algorithm can be initialized in $O(n \log^{o(1)} n + Wk^2 \log^2 n)$ time and, along with $\mathrm{ed}^w(X, Y)$, it also outputs a w-optimal sequence of edits transforming X into Y.*

1.3 Open Questions

In the unweighted setting, the $O(k \log^2 n)$ update time of Theorem 1.6 does not give much room for improvement: the most pressing challenge is to reduce the $O(\log^2 n)$-factor overhead in [20]. For small integer weights, the dependency on the largest weight W remains to be studied. The linear dependency in the running time $\widetilde{O}(n + Wk^2)$ of Theorem 1.3 feels justified, but we hope that the quadratic dependency in the update time $\widetilde{O}(W^2 k)$ of Theorem 1.7 can be reduced. An exciting open question is to determine the optimality of our $\widetilde{O}(n + k^{2.5})$-time static algorithm for large integer weights (for $k \le \min\{W^2, \sqrt{n}\}$). This runtime matches the lower bound of [18], but the hard instances constructed there crucially utilize fractional weights. The complexity of dynamic bounded edit distance also remains widely open for large weights: we are only aware of how to achieve $\widetilde{O}(k^{2.5})$ and $\widetilde{O}(k^3)$ update time for integer and general weights, respectively, and these solutions simply combine static algorithms with a dynamic strings implementation supporting efficient substring equality tests. With extra effort, the techniques developed in this paper should lead to a faster support of updates that do not change $\mathrm{ed}^w(X, Y)$ too much. We do not know, however, how to improve upon the aforementioned simple approach for updates that drastically alter $\mathrm{ed}^w(X, Y)$.

2 Technical Overview

In this section, we provide an overview of our algorithms. We focus on the static algorithm behind Theorem 1.1, which already incorporates most novel ideas and techniques. Complete proofs of all theorems from Section 1 can be found in the full version of the paper [40]. In the overview, we assume for simplicity that the parameter k is an upper bound on $\mathrm{ed}^w(X, Y)$ known to the algorithms.

Basic Concepts. Consistently with most previous work, we interpret the edit distance problem using the *alignment graph* of the input strings. For strings $X, Y \in \Sigma^*$ and a weight function $w : \overline{\Sigma}^2 \to \mathbb{R}_{\ge 0}$, the alignment graph $\mathrm{AG}^w(X, Y)$ is a directed grid graph with vertex set $[0..|X|] \times [0..|Y|]$ and the following edges (see Figure 1):

- $(x, y) \to (x, y + 1)$ of weight $w(\varepsilon, Y[y])$, representing an insertion of $Y[y]$;
- $(x, y) \to (x + 1, y)$ of weight $w(X[x], \varepsilon)$, representing a deletion of $X[x]$;
- $(x, y) \to (x+1, y+1)$ of weight $w(X[x], Y[y])$, representing a match (if $X[x] = Y[y]$) or a substitution of $X[x]$ for $Y[y]$.

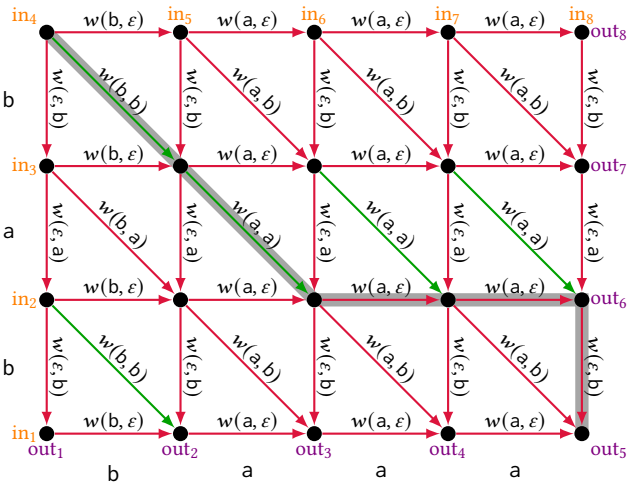


Figure 1: The alignment graph $\mathrm{AG}^w(X, Y)$ for $X = \mathtt{baaa}$ and $Y = \mathtt{bab}$. In the unweighted case, the green edges have weight 0, and the red ones have weight 1. In the weighted case, the red edges have some weights defined by the weight function w. An optimal alignment for a weight function w satisfying $w(\mathtt{a}, \varepsilon) = w(\varepsilon, \mathtt{b}) = 1$ and $w(\mathtt{a}, \mathtt{b}) = w(\mathtt{b}, \mathtt{a}) = w(\varepsilon, \mathtt{a}) = w(\mathtt{b}, \varepsilon) = 3$ is given in gray. The input vertices for the boundary matrix $\mathrm{BM}^w(X, Y)$ have orange labels, and the output vertices have violet labels.

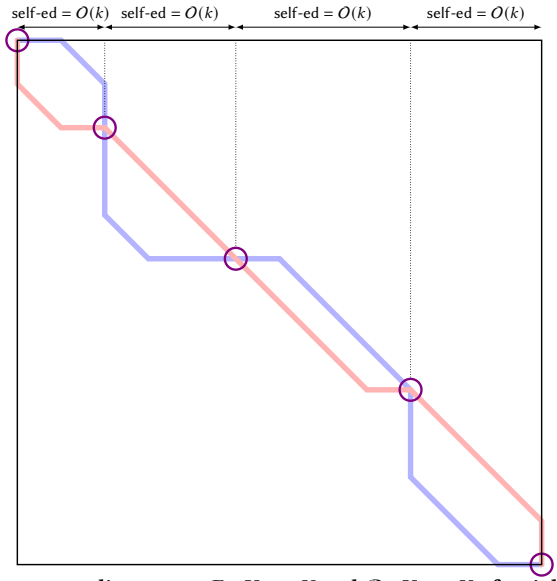
Every alignment $\mathcal{A}$ mapping X onto Y, denoted $\mathcal{A} : X \rightsquigarrow\!\!\!\!\!\to Y$, can be interpreted as a path in $\mathrm{AG}^w(X, Y)$ from the top-left corner $(0, 0)$ to the bottom-right corner $(|X|, |Y|)$, and the cost of the alignment, denoted $\mathrm{ed}^w_{\mathcal{A}}(X, Y)$ is the length of the underlying path. Consequently, the edit distance $\mathrm{ed}^w(X, Y)$ is simply the distance from $(0, 0)$ to $(|X|, |Y|)$ in the alignment graph.

Many edit-distance algorithms compute not only the distance from $(0, 0)$ to $(|X|, |Y|)$ but the entire *boundary distance matrix* $\mathrm{BM}^w(X, Y)$ that stores the distances from every *input* vertex on the top-left boundary to every *output* vertex on the bottom-right boundary of the alignment graph $\mathrm{AG}^w(X, Y)$. Crucially, the planarity of $\mathrm{AG}^w(X, Y)$ implies that $M = \mathrm{BM}^w(X, Y)$ satisfies the *Monge property*, i.e., $M_{i,j} + M_{i+1,j+1} \le M_{i,j+1} + M_{i+1,j}$ holds whenever all four entries are finite.[6]

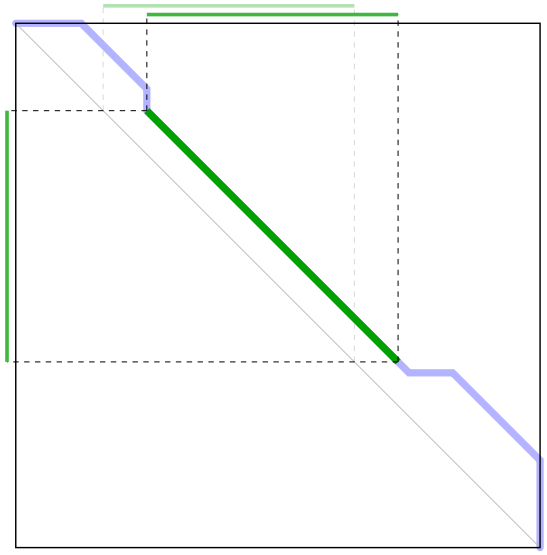
High-Level Algorithm Structure: Divide and Conquer.

Self-Edit Distance. The most important insight from [18] is that the problem of computing the value $\mathrm{ed}^w(X, Y) \le k$ can be reduced to instances satisfying $\text{self-ed}(X) = O(k)$. The value $\text{self-ed}(X)$, called the *self-edit distance* of X, is the distance from $(0, 0)$ to $(|X|, |X|)$ in the unweighted alignment graph $\mathrm{AG}(X, X)$ with the edges $(x, x) \to (x + 1, x + 1)$ on the main diagonal removed. There are two main properties of self-edit distance that we employ. The

[6] In this overview, we ignore the fact that some entries of M are infinite. Our actual algorithms augment the alignment graph with backward edges so that the finite distances are preserved and the infinite distances become finite.



(a) For any two alignments $\mathcal{A} : X \rightsquigarrow Y$ and $\mathcal{B} : X \rightsquigarrow Y$ of weights at most k, the self-edit distances of fragments between consecutive intersection points of $\mathcal{A}$ and $\mathcal{B}$ are bounded by $O(k)$.



(b) Illustration of a repetitive factor of a string X induced by the small self-edit distance of X. A self-alignment of X is given in blue. The green diagonal segment consists of weight-0 edges and corresponds to two matching fragments in X that are at most self-ed(X) positions apart. The earlier of these fragments is drawn second time for clarity.

Figure 2: Two main properties of self-edit distance.

first one claims that, for any two alignments $\mathcal{A} : X \rightsquigarrow Y$ and $\mathcal{B} : X \rightsquigarrow Y$ of weights at most k, the self-edit distance of any fragment between two consecutive intersection points of $\mathcal{A}$ and $\mathcal{B}$ is bounded by $O(k)$; see Figure 2a. To see that, let $\hat{X}$ be the fragment of X between two consecutive intersection points of $\mathcal{A}$ and $\mathcal{B}$. Restrictions of $\mathcal{A}$ and $\mathcal{B}$ onto $\hat{X}$ are edge-disjoint. By transforming $\hat{X}$ into some $\hat{Y}$ using $\mathcal{A}$, and then inverting this transformation using $\mathcal{B}$, we get a self-alignment of $\hat{X}$ onto itself of cost at most $\mathrm{ed}^w_{\mathcal{A}}(\hat{X}, \hat{Y}) + \mathrm{ed}^w_{\mathcal{B}}(\hat{X}, \hat{Y}) = O(k)$.

The second property of self-edit distance we use is that strings of small self-edit distance are highly repetitive. More precisely, a string X of self-edit distance at most k can be factorized into at most k individual characters (that the optimal self-alignment deletes or substitutes) and at most $k+1$ factors with an earlier occurrence at most k positions earlier (that the self-alignment matches perfectly); see Figure 2b. Using this property, Cassis, Kociumaka, and Wellnitz [18] show that there is a decomposition $X = \bigodot_{i=0}^{m-1} X_i$ of the string X into phrases $X_i = X[x_i \,..\, x_{i+1})$ of length $\Theta(k)$ each such that all but $O(k)$ phrases X_i satisfy $X_i = X_{i-1}$.

Previous Divide-and-Conquer Scheme. We now sketch the reduction from the problem of computing the value $\mathrm{ed}^w(X, Y) \le k$ in the general case to instances satisfying self-ed$(X) = O(k)$ given in [18]. At a high level, the reduction follows a divide-and-conquer scheme: the strings are partitioned into two *halves* $X = X_L X_R$ and $Y = Y_L Y_R$, the values $\mathrm{ed}^w(X_L, Y_L)$ and $\mathrm{ed}^w(X_R, Y_R)$ are computed recursively, and then the distance $\mathrm{ed}^w(X, Y)$ is derived. The original approach from [18], also applied in a quantum algorithm for the unweighted edit distance [35], partitions X so that the two halves have the same length (up to ± 1) and Y so that $\mathrm{ed}^w(X, Y) = \mathrm{ed}^w(X_L, Y_L) + \mathrm{ed}^w(X_R, Y_R)$. For the latter, it computes a w-optimal alignment between appropriate fragments of self-edit distance $O(k)$ taken from the middles of X and Y. Intuitively, this is sufficient because whether a given vertex $(x, y) \in [0..|X|] \times [0..|Y|]$ belongs to a w-optimal alignment of cost at most k depends only on *contexts* of self-edit distance $O(k)$ around positions x in X and y in Y (due to the first main property of self-edit distance). Unfortunately, the resulting scheme is unable to tell how to split the budget k between the two recursive calls, and thus both calls utilize exponential search to estimate the local weighted edit distance, causing significant complications and, in case of [35], also a large polylogarithmic-factor overhead on top of the $\widetilde{O}(k^2)$ time complexity. Moreover, this scheme is incompatible with the dynamic setting because a single update may change the partition of Y and thus affect both recursive calls.

Technical Contribution 1: Simple and Robust Divide-and-Conquer Scheme. We circumvent the aforementioned issues using a novel divide-and-conquer approach that relies on an *approximately optimal* alignment $\mathcal{A} : X \rightsquigarrow Y$ of (weighted) cost $\mathrm{ed}^w_{\mathcal{A}}(X, Y) = O(k)$. In the context of Theorem 1.1, we can pick $\mathcal{A}$ to be an optimal *unweighted* alignment whose cost satisfies $\mathrm{ed}^w_{\mathcal{A}}(X, Y) = O(\mathrm{ed}(X, Y)) = O(\mathrm{ed}^w(X, Y)) = O(k)$. In the dynamic setting, it suffices to occasionally rebuild $\mathcal{A}$.

The procedure described next and illustrated in Figure 3a constructs a w-optimal alignment $\mathcal{B} : X \rightsquigarrow Y$ using an arbitrary alignment $\mathcal{A} : X \rightsquigarrow Y$ of cost $O(k)$. We partition X into two halves of the same length (up to ± 1) and Y so that $(x_m, y_m) := (|X_L|, |Y_L|) \in \mathcal{A}$, that is, $\mathrm{ed}^w_{\mathcal{A}}(X, Y) = \mathrm{ed}^w_{\mathcal{A}}(X_L, Y_L) + \mathrm{ed}^w_{\mathcal{A}}(X_R, Y_R)$.

Then, we recursively build w-optimal alignments $\mathcal{B}_L : X_L \rightsquigarrow\!\!\!\!\!\rightarrow Y_L$ and $\mathcal{B}_R : X_R \rightsquigarrow\!\!\!\!\!\rightarrow Y_R$. Now, the recursive calls have predictable structure (guided by $\mathcal{A}$) with local budgets given by the local costs of $\mathcal{A}$. Unfortunately, we are no longer guaranteed that $\mathrm{ed}^w(X, Y) = \mathrm{ed}^w(X_L, Y_L) + \mathrm{ed}^w(X_R, Y_R)$, so $\mathcal{B}$ cannot be obtained by just concatenating $\mathcal{B}_L$ and $\mathcal{B}_R$. Nevertheless, it turns out that the resulting alignment needs to be fixed only in a small (in terms of self-edit distance) neighborhood of (x_m, y_m). To be precise, we find a context X_M of position x_m in X and compute the w-optimal alignment $\mathcal{B}_M : X_M \rightsquigarrow\!\!\!\!\!\rightarrow Y_M$, where Y_M is the image of X_M under $\mathcal{A}$. If the context has a sufficiently large self-edit distance (on both sides of x_m), then the alignment $\mathcal{B}_M$ intersects both $\mathcal{B}_L$ and $\mathcal{B}_R$, and the sought w-optimal alignment $\mathcal{B} : X \rightsquigarrow\!\!\!\!\!\rightarrow Y$ can be obtained from the concatenation of $\mathcal{B}_L$ and $\mathcal{B}_R$ by following $\mathcal{B}_M$ between the two intersection points. The recursion terminates whenever X is very short (when a naive $O(nk)$-time algorithm is used) or $\mathcal{A}$ perfectly matches X with Y (when $\mathcal{A}$ is already w-optimal).

The scheme above reduces computing $\mathrm{ed}^w(X, Y)$ to the case when $\text{self-ed}(X) = O(k)$ with an $O(\log n)$-factor overhead due to the depth of recursion. In the full version [40], we provide an optimized implementation that avoids this overhead by partitioning X so that the cost of $\mathcal{A}$ is split equally between the two halves. We also discuss computing $\mathcal{A}$ in the contexts of Theorems 1.3 and 1.4. A similar instantiation of this scheme can also be applied to significantly simplify the results of [18].

The Case of Small Self-Edit Distance. The remaining task is to compute $\mathrm{ed}^w(X, Y)$ under the assumption that $\mathrm{ed}^w(X, Y) \le k$ and $\text{self-ed}(X) = O(k)$ hold for a known parameter k. On a high level, we follow an $\widetilde{O}(n + k^3)$-time algorithm from [18], whose setup we recall next and illustrate in Figure 3b. Due to the second main property of self-edit distance, the optimal self-alignment $X \rightsquigarrow\!\!\!\!\!\rightarrow X$ yields a decomposition $X = \bigodot_{i=0}^{m-1} X_i$ of the string X into phrases $X_i = X[x_i .. x_{i+1})$ of length $\Theta(k)$ so that all but $O(k)$ phrases X_i satisfy $X_i = X_{i-1}$. For each phrase, we define a fragment $Y_i = Y[y_i .. y'_{i+1})$, where $y_i = \max\{x_i - k, 0\}$ and $y'_{i+1} = \min\{x_{i+1} + k, |Y|\}$, and a subgraph G_i of $\mathrm{AG}^w(X, Y)$ induced by $[x_i .. x_{i+1}] \times [y_i .. y'_{i+1}]$. The union G of subgraphs G_i contains every vertex (x, y) with $|x - y| \le k$, so the promise $\mathrm{ed}^w(X, Y) \le k$ guarantees $\mathrm{ed}^w(X, Y) = \mathrm{dist}_G((0, 0), (|X|, |Y|))$: every alignment pays at least 1 to move between diagonals, so the optimal alignment cannot deviate by more than k from the main diagonal. Furthermore, there are $O(k)$ indices i such that $(X_i, Y_i) \ne (X_{i-1}, Y_{i-1})$ and, the results of [18] let us efficiently construct the set F of such indices i along with the underlying fragments X_i and Y_i.

For each $i \in [0 .. m]$, we define $V_i = \{x_i\} \times [y_i .. y'_i]$, where we set $y_m = |Y|$ and $y'_0 = 0$ so that $V_0 = \{(0, 0)\}$ and $V_m = \{(|X|, |Y|)\}$. For $i \in (0 .. m)$, the set V_i consists of the vertices shared by G_{i-1} and G_i. Thus, if we define $D_{i,j}$ as the matrix of distances (in G) between vertices in V_i and V_j, then the sought value $\mathrm{ed}^w(X, Y)$ is the only entry of $D_{0,m} = \bigotimes_{i=0}^{m-1} D_{i,i+1}$, where $\bigotimes$ denotes the $(\min, +)$ product of matrices. Since $D_{i,i+1} = D_{i-1,i}$ holds for all $i \notin F$, it suffices to construct $D_{i,i+1}$ for all $i \in F$, raise each matrix to an appropriate power, and then multiply all the $O(k)$ matrix powers. Overall, this requires constructing $O(k)$ individual matrices $D_{i,i+1}$ and performing $O(k \log n)$ products of the form $D_{p,r} = D_{p,q} \otimes D_{q,r}$. The algorithm of [18] performs each of these operations individually using Klein's planar multiple-source shortest path algorithm [52] to construct $D_{i,i+1}$ in $O(k^2 \log k)$ time and the SMAWK method [3] for $O(k^2)$-time $(\min, +)$ Monge matrix multiplication. Both steps contribute to an $\widetilde{O}(k^3)$-time bottleneck.

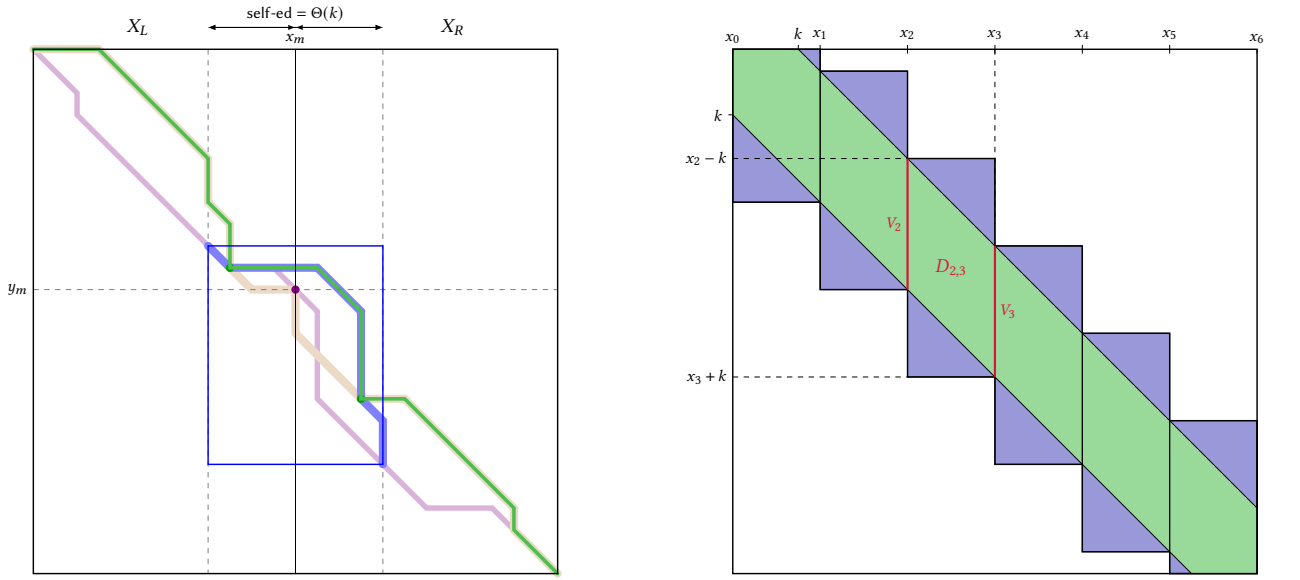
Technical Contribution 2: Application of Core-Sparse Monge Matrix Multiplication. The key advantage of small integer weights is the *bounded difference* property [16] satisfied by the underlying distance matrices: if the edit costs are in $[0 .. W]$, then the differences between adjacent entries in any distance matrix belong to the integer range $[-W .. W]$. We observe that an $O(s) \times O(s)$ Monge matrix M satisfying this property has just $O(Ws)$ so-called *core elements*: pairs (i, j) such that $M_{i,j} + M_{i+1,j+1} \ne M_{i,j+1} + M_{i+1,j}$. As shown by Russo [69], such matrices can be stored in $\widetilde{O}(Ws)$ space and their $(\min, +)$ product can be computed in $\widetilde{O}(Ws)$ time. We apply this result (recently improved in [31]) in the full version [40], which provides a complete framework for *core-sparse* Monge matrices. Using this framework, the distance matrices can be stored in $\widetilde{O}(k)$ space and their $(\min, +)$ products can be computed in $\widetilde{O}(k)$ time, thus circumventing one of the two bottlenecks.

Technical Contribution 3: Exploiting Shared Structures between Graphs G_i. With the improved multiplication time, the remaining bottleneck is the construction of individual distance matrices. Building one such matrix requires $k^{2-o(1)}$ time (because it stores weighted edit distance of length-$\Theta(k)$ strings), so we have to exploit some shared structure between the graphs G_i for $i \in F$. Combining core-sparse Monge matrix multiplication with insights behind the $O(n + \sqrt{nk^3})$-time algorithm of [18] lets us construct these matrices in $\widetilde{O}(k^{2.5})$ total time, which is still unsatisfactory.

Our improved solution shares the following key observation with [18]: due to $\text{self-ed}(X) = O(k)$, the string X can be factorized into $O(k)$ individual characters (that the optimal self-alignment edits) and $O(k)$ factors with an earlier occurrence $O(k)$ positions earlier (that the self-alignment matches perfectly). Using this factorization, we show that distinct phrases $X_{i-1} \ne X_i$ are still very similar *on average*: for each $i \in F$, we issue a sequence of k_i operations extending X_{i-1} to $X_{i-1}X_i$ so that $\sum_{i \in F} k_i = \widetilde{O}(k)$ and each operation extends the current string with a single character (an insertion) or a substring of the current string (a copy-paste). This follows from the second main property of self-edit distance: the string X can be decomposed into $O(k)$ individual characters and $O(k)$ fragments with earlier occurrences in X that we can copy-paste from. Finally, X_i is obtained from $X_{i-1}X_i$ with a prefix deletion. Similar $\widetilde{O}(k)$ updates transform Y_{i-1} to Y_i as we iterate over all $i \in F$.

As explained below, we will store boundary distance matrices using a dynamic algorithm so that $\mathrm{BM}^w(X_i, Y_i)$ can be obtained from $\mathrm{BM}^w(X_{i-1}, Y_{i-1})$ using just $O(k_i)$ operations implemented in $\widetilde{O}(k)$ time each, for a total of $\widetilde{O}(k^2)$ time across all $i \in F$. The techniques described so far to compute $\mathrm{ed}^w(X, Y)$ in the case of small self-edit distance are presented in the full version [40].

Dynamic Algorithm for Unbounded Unweighted Edit Distance. To maintain $\mathrm{BM}^w(X, Y)$, we build upon a dynamic algorithm of Charalampopoulos, Kociumaka, and Mozes [20], which supports arbitrary edits as updates in $\widetilde{O}(|X| + |Y|)$ time each. Although the original formulation is limited to *unweighted* edit distance, it



(a) The approximate alignment $\mathcal{A}$ is drawn in violet. The two optimal alignments $\mathcal{B}_L$ and $\mathcal{B}_R$ computed recursively are given in brown. The optimal alignment $\mathcal{B}_M$ in the blue central area of small self-edit distance is given in blue. The globally optimal green alignment $\mathcal{B}$ is a combination of $\mathcal{B}_L$, $\mathcal{B}_M$, and $\mathcal{B}_R$.

(b) The decomposition of the alignment graph of X and Y into subgraphs G_i of size $\Theta(k) \times \Theta(k)$ that cover the whole stripe of width $O(k)$ around the main diagonal. Each matrix $D_{i,i+1}$ represents the distances between the vertices of V_i and V_{i+1}.

Figure 3: Our algorithms' setup for the general case and the case of small self-edit distance, respectively.

can be easily adapted to a fixed integer weight function using our core-sparse Monge matrix framework instead of Tiskin's *simple unit-Monge* matrices [73, 74].

To describe the algorithm, let us first assume for simplicity that $|X| = |Y| = n$ is a power of two. Then, $\mathrm{BM}^w(X, Y)$ can be constructed in $\widetilde{O}(n^2)$ time by decomposing, for each scale $s \in [0..\log n]$, the strings X and Y into fragments of length 2^s and building the boundary distance matrix for each pair of fragments of length 2^s. For $s = 0$, this matrix can be easily constructed in constant time; for $s > 0$, we can combine four boundary distance matrices of the smaller scale. This approach readily supports dynamic strings subject to substitutions: upon each update, it suffices to recompute all the affected boundary matrices. For each scale s, the number of such matrices is $O(n/2^s)$, and each of them is constructed in $\widetilde{O}(2^s)$ time; this yields a total update time of $\widetilde{O}(n)$. To support insertions and deletions as updates, the algorithm of [20] relaxes the hierarchical decomposition so that the lengths of fragments at each scale s may range from 2^{s-2} to 2^{s+1}. With appropriate rebalancing, the simple strategy of recomputing all the affected boundary matrices still takes $\widetilde{O}(n)$ time per update.

Technical Contribution 4: Supporting Copy-Pastes with Balanced Straight Line Programs. Even with the extension to small integer weights, the algorithm of [20] falls short of our requirements as it only supports character edits as updates, while we also need the copy-paste operation that extracts a substring and appends its copy at the end of the string. Consider the hierarchical decomposition that the algorithm of [20] maintains for X. Although a copy-paste on X adds a lot of new fragments to the decomposition, we expect most of them to match fragments of the pre-existing decomposition of X; for such fragments, no new boundary distance matrices need to be computed. To keep track of matching fragments, it is convenient to represent the hierarchical decomposition as a directed acyclic graph obtained by gluing together isomorphic subtrees. In data compression, such a representation of a binary tree is called *a straight-line program* (SLP). Each node of the DAG can then be interpreted as a symbol in a context-free grammar, with leaves (sink nodes) corresponding to terminals and the remaining nodes to non-terminals. Each non-terminal has exactly one production corresponding to the pair of outgoing edges. These properties guarantee that the language associated with each symbol consists of one string: the *expansion* of the symbol.

Established tools [23, 70] can be used to maintain an SLP with operations that create a symbol whose expansion is the concatenation of the expansions of two existing symbols or a prescribed substring of the expansion of an existing symbol. Crucially, these operations add only logarithmically many auxiliary symbols. Unfortunately, unlike the hierarchical decompositions of [20], these SLPs lack a structure of levels, so it is not clear for which pairs of symbols the boundary distance matrix should be stored. To address this issue, we rely on an additional invariant maintained in [23]: for every non-terminal A with production $A \to A_L A_R$, the lengths of expansions of A_L and A_R are within a constant factor of each other. Based on this, we choose to store the boundary distance matrix for every pair of symbols (A, B) (originating from the SLP of X and the SLP of Y, respectively) such that the expansions of A and B are also within a constant factor of each other. This way, when a new symbol A is added to the SLP of X, we need to build the boundary distance matrix for $O(|Y|/\mathrm{len}(A))$ symbols B in the SLP

of Y, where $\mathrm{len}(A)$ denotes the length of the expansion of A, and each matrix is constructed in $\widetilde{O}(\mathrm{len}(A))$ time, for a total of $\widetilde{O}(|Y|)$. The details are described in the full version of the paper [40].

Faster Static Algorithm for Large Integer Weights. The overview above explains how to achieve $\widetilde{O}(n+k^2)$ runtime for a fixed integer weight function, but the underlying techniques yield an $\widetilde{O}(n + W \cdot k^2)$-time solution if a weight function $w : \overline{\Sigma}^2 \to [0..W]$ is given through an oracle. The only major challenge arising in this generalization is to construct an approximate alignment for the divide-and-conquer procedure. A simple but effective trick is to compute a w'-optimal alignment for a weight function w' defined as $w'(a,b) = \lceil w(a,b)/2 \rceil$. Since w and w' are within a factor of 2 from each other, this yields a 2-approximate w-optimal alignment. Moreover, repeating this reduction $\lceil \log W \rceil$ times, we arrive at the problem of computing an unweighted alignment, which we solve in time $O(n+k^2)$ using the Landau–Vishkin algorithm [60].

If the weights are large, the $(\min,+)$-product of $s \times s$-sized integer-valued Monge matrices takes $\widetilde{\Theta}(s^2)$ time to be computed, just like in the fractional case. In our application, however, we care about distances at most k, and thus only small entries need to be computed truthfully. In the full version [40], we develop a novel procedure of independent interest that, given an $s \times s$ Monge matrix M with non-negative integer entries, constructs an $s \times s$ Monge matrix M' whose core is of size $O(s\sqrt{k})$ yet $\min\{M_{i,j}, k+1\} = \min\{M'_{i,j}, k+1\}$ holds for every entry. Applying this reduction to all distance matrices that our algorithm constructs, we get an $\widetilde{O}(n + k^{2.5})$-time static algorithm, which is faster than all known alternatives when $k \ll \min\{W^2, \sqrt{n}\}$.

Dynamic Algorithm for Bounded Edit Distance. In order to turn the procedure behind Theorem 1.1 into a dynamic algorithm, we maintain the strings X and Y in a standard dynamic data structure [34, 50, 64] that provides $\widetilde{O}(1)$-time random access and equality tests (whether two fragments match perfectly) at the cost of an $\widetilde{O}(1)$-time additive overhead for each update. Below, we highlight further major adaptations.

The Case of Small Self-Edit Distance. The central component of our solution is a subroutine that, given a threshold k, maintains $\mathrm{ed}^w(X,Y)$ for $O(k)$ updates, under a promise that $\text{self-ed}(X) = O(k)$ and $\mathrm{ed}^w(X,Y) = O(k)$. Its high-level structure resembles the static counterpart illustrated in Figure 3b, but the subgraphs G_i are slightly taller so that their union G covers a sufficiently wide stripe even after $O(k)$ updates. At the initialization time, we still identify $O(k)$ indices $i \in [0..m)$ such that $(X_i, Y_i) \neq (X_{i-1}, Y_{i-1})$, and we derive the corresponding matrices $D_{i,i+1}$ from the boundary distance matrices $\mathrm{BM}^w(X_i, Y_i)$. The key difference is that we rely on the dynamic algorithm employed to construct boundary distance matrices being *fully persistent*: each update creates a new instance of the algorithm and one can still issue updates to the original instance. As a result, $\mathrm{BM}^w(X_i, Y_i)$ (and hence $D_{i,i+1}$) can be updated in $\widetilde{O}(k)$ time whenever an update affects X_i or Y_i. The $(\min,+)$-product $D_{0,m} = \bigotimes_{i=0}^{m-1} D_{i,i+1}$ also needs to be maintained dynamically: for this, we employ a perfectly balanced binary tree with leaves storing the subsequent matrices $D_{i,i+1}$ and internal nodes maintaining the product of its children's matrices. In order to secure $\widetilde{O}(k^2)$-time initialization, identical subtrees are glued together at initialization time (so that each run of identical matrices $D_{i,i+1}$ requires creating just $O(\log m)$ nodes), and the updates are implemented in a functional way (we create a new copy of each modified node since the original may still be linked from elsewhere). With these modifications, in the full version [40], we achieve $\widetilde{O}(k^2)$-time initialization and $\widetilde{O}(k)$-time updates.

Dynamic Divide-and-Conquer Implementation. In its basic version, our implementation of the divide-and-conquer algorithm is given a threshold k and maintains $\mathrm{ed}^w(X,Y)$ for $O(k)$ updates under a promise that $\mathrm{ed}^w(X,Y) = O(k)$. At the initialization time, we use the $\widetilde{O}(k^2)$-time static algorithm to build an optimal alignment $\mathcal{A}$ that will guide our recursive decomposition: we partition $X = X_L X_R$ and $Y = Y_L Y_R$ so that $\mathrm{ed}^w_{\mathcal{A}}(X_L, Y_L) + \mathrm{ed}^w_{\mathcal{A}}(X_R, Y_R) = \mathrm{ed}^w_{\mathcal{A}}(X,Y)$ and recursively maintain $\mathrm{ed}^w(X_L, Y_L)$ and $\mathrm{ed}^w(X_R, Y_R)$. Crucially, the recursion has negligible overhead because each update affects only one of these subroutines. As in the static case, the optimal alignments $\mathcal{B}_L : X_L \rightsquigarrow Y_L$ and $\mathcal{B}_R : X_R \rightsquigarrow Y_R$ are combined into an optimal alignment $\mathcal{B} : X \rightsquigarrow Y$ using an optimal alignment $\mathcal{B}_M : X_M \rightsquigarrow Y_M$ for an appropriate fragment X_M around the midpoint of X and its image Y_M under $\mathcal{A}$ (see Figure 3a). In the dynamic case, we need to be more generous when initializing X_M since updates may increase the cost of $\mathcal{A}$ and decrease the self-edit distances. Nevertheless, we can still maintain $\mathrm{ed}^w(X_M, Y_M)$ using the subroutine from the previous subsubsection.

The remaining challenge is to make sure that, despite occasional rebuilding, the update time is $\widetilde{O}(k)$ in the worst case rather than just amortized. Such a deamortization is needed at each level of the recursive procedure, so standard techniques would yield a polylogarithmic-factor penalty. In the full version [40], we propose a tailor-made approach whose (rather technical) complexity analysis proves that we incur just a constant-factor overhead, and hence achieve $O(k \log^2 n)$ update time for any fixed weight function, matching the results of [20] for $k = \Theta(n)$ and unweighted edit distance.

Acknowledgments

The second author would like to thank Itai Boneh (Reichman University) for helpful discussions.

References

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. Tight Hardness Results for LCS and Other Sequence Similarity Measures. In *FOCS 2015*. IEEE Computer Society, 59–78. https://doi.org/10.1109/FOCS.2015.14

[2] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. 2016. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *STOC 2016*. ACM, 375–388. https://doi.org/10.1145/2897518.2897653

[3] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. 1987. Geometric Applications of a Matrix-Searching Algorithm. *Algorithmica* 2 (1987), 195–208. https://doi.org/10.1007/BF01840359

[4] Shyan Akmal and Ce Jin. 2021. Faster Algorithms for Bounded Tree Edit Distance. In *ICALP 2021 (LIPIcs, Vol. 198)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:15. https://doi.org/10.4230/LIPIcs.ICALP.2021.12

[5] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. 2000. Pattern matching in dynamic texts. In *SODA 2000*. ACM/SIAM, 819–828. http://dl.acm.org/citation.cfm?id=338219.338645

[6] Amihood Amir and Itai Boneh. 2018. Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms. In *CPM 2018 (LIPIcs, Vol. 105)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 11:1–11:13. https://doi.org/10.4230/LIPICS.CPM.2018.11

[7] Amihood Amir and Itai Boneh. 2020. Update Query Time Trade-Off for Dynamic Suffix Arrays. In *ISAAC 2020 (LIPIcs, Vol. 181)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 63:1–63:16. https://doi.org/10.4230/LIPICS.ISAAC.2020.63

[8] Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. 2019. Repetition Detection in a Dynamic String. In *ESA 2019 (LIPIcs, Vol. 144)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 5:1–5:18. https://doi.org/10.4230/LIPIcs.ESA.2019.5

[9] Arturs Backurs and Piotr Indyk. 2018. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). *SIAM J. Comput.* 47, 3 (2018), 1087–1097. https://doi.org/10.1137/15M1053128

[10] Arturs Backurs and Krzysztof Onak. 2016. Fast Algorithms for Parsing Sequences of Parentheses with Few Errors. In *PODS 2016*. ACM, 477–488. https://doi.org/10.1145/2902251.2902304

[11] Djamal Belazzougui and Qin Zhang. 2016. Edit Distance: Sketching, Streaming, and Document Exchange. In *FOCS 2016*. IEEE Computer Society, 51–60. https://doi.org/10.1109/FOCS.2016.15

[12] Sudatta Bhattacharya and Michal Koucký. 2023. Locally Consistent Decomposition of Strings with Applications to Edit Distance Sketching. In *STOC 2023*. ACM, 219–232. https://doi.org/10.1145/3564246.3585239

[13] Karl Bringmann, Alejandro Cassis, Nick Fischer, and Tomasz Kociumaka. 2024. Faster Sublinear-Time Edit Distance. In *SODA 2024*, Vol. abs/2312.01759. SIAM, 3274–3301. https://doi.org/10.1137/1.9781611977912.117

[14] Karl Bringmann, Alejandro Cassis, Nick Fischer, and Vasileios Nakos. 2022. Almost-optimal sublinear-time edit distance in the low distance regime. In *STOC 2022*. ACM, 1102–1115. https://doi.org/10.1145/3519935.3519990

[15] Karl Bringmann, Alejandro Cassis, Nick Fischer, and Vasileios Nakos. 2022. Improved Sublinear-Time Edit Distance for Preprocessed Strings. In *ICALP 2022 (LIPIcs, Vol. 229)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 32:1–32:20. https://doi.org/10.4230/LIPIcs.ICALP.2022.32

[16] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. 2019. Truly Subcubic Algorithms for Language Edit Distance and RNA Folding via Fast Bounded-Difference Min-Plus Product. *SIAM J. Comput.* 48, 2 (2019), 481–512. https://doi.org/10.1137/17M112720X

[17] Karl Bringmann and Marvin Künnemann. 2015. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. In *FOCS 2015*. IEEE Computer Society, 79–97. https://doi.org/10.1109/FOCS.2015.15

[18] Alejandro Cassis, Tomasz Kociumaka, and Philip Wellnitz. 2023. Optimal Algorithms for Bounded Weighted Edit Distance. In *FOCS 2023*. IEEE, 2177–2187. https://doi.org/10.1109/FOCS57990.2023.00135

[19] Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. 2020. Dynamic Longest Common Substring in Polylogarithmic Time. In *ICALP 2020 (LIPIcs, Vol. 168)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 27:1–27:19. https://doi.org/10.4230/LIPIcs.ICALP.2020.27

[20] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. 2020. Dynamic String Alignment. In *CPM 2020 (LIPIcs, Vol. 161)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 9:1–9:13. https://doi.org/10.4230/LIPIcs.CPM.2020.9

[21] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. 2020. Faster Approximate Pattern Matching: A Unified Approach. In *FOCS 2020*. IEEE, 978–989. https://doi.org/10.1109/FOCS46700.2020.00095

[22] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. 2022. Faster Pattern Matching under Edit Distance: A Reduction to Dynamic Puzzle Matching and the Seaweed Monoid of Permutation Matrices. In *FOCS 2022*. IEEE, 698–707. https://doi.org/10.1109/FOCS54457.2022.00072

[23] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Trans. Inf. Theory* 51, 7 (2005), 2554–2576. https://doi.org/10.1109/TIT.2005.850116

[24] Raphaël Clifford, Paweł Gawrychowski, Tomasz Kociumaka, Daniel P. Martin, and Przemysław Uznański. 2022. The Dynamic k-Mismatch Problem. In *CPM 2022 (LIPIcs, Vol. 223)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 18:1–18:15. https://doi.org/10.4230/LIPIcs.CPM.2022.18

[25] Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, and Barna Saha. 2023. Weighted Edit Distance Computation: Strings, Trees, and Dyck. In *STOC 2023* (Orlando, FL, USA) *(STOC 2023)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/3564246.3585178

[26] Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, Barna Saha, and Hamed Saleh. 2022. $\tilde{O}(n + \mathrm{poly}(k))$-time Algorithm for Bounded Tree Edit Distance. In *FOCS 2022*. IEEE, 686–697. https://doi.org/10.1109/FOCS54457.2022.00071

[27] Margaret Dayhoff, R Schwartz, and B Orcutt. 1978. A model of evolutionary change in proteins. *Atlas of protein sequence and structure* 5 (1978), 345–352.

[28] Anita Dürr. 2023. Improved bounds for rectangular monotone Min-Plus Product and applications. *Inf. Process. Lett.* 181 (2023), 106358. https://doi.org/10.1016/j.ipl.2023.106358

[29] Dvir Fried, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Tatiana Starikovskaya. 2023. An Improved Algorithm for The k-Dyck Edit Distance Problem. *ACM Trans. Algorithms* (2023). https://doi.org/10.1145/3627539

[30] Arun Ganesh, Tomasz Kociumaka, Andrea Lincoln, and Barna Saha. 2022. How Compression and Approximation Affect Efficiency in String Distance Measures. In *SODA 2022*. SIAM, 2867–2919. https://doi.org/10.1137/1.9781611977073.112

[31] Paweł Gawrychowski, Egor Gorbachev, and Tomasz Kociumaka. 2024. Core-Sparse Monge Matrix Multiplication: Improved Algorithm and Applications. arXiv:2408.04613

[32] Paweł Gawrychowski and Wojciech Janczewski. 2021. Fully dynamic approximation of LIS in polylogarithmic time. In *STOC 2021*. ACM, 654–667. https://doi.org/10.1145/3406325.3451137

[33] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. 2015. Optimal Dynamic Strings. https://doi.org/10.48550/arXiv.1511.02612

[34] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. 2018. Optimal Dynamic Strings. In *SODA 2018*. SIAM, 1509–1528. https://doi.org/10.1137/1.9781611975031.99

[35] Daniel Gibney, Ce Jin, Tomasz Kociumaka, and Sharma V. Thankachan. 2024. Near-Optimal Quantum Algorithms for Bounded Edit Distance and Lempel-Ziv Factorization. In *SODA 2024*. SIAM, 3302–3332. https://doi.org/10.1137/1.9781611977912.118

[36] Elazar Goldenberg, Tomasz Kociumaka, Robert Krauthgamer, and Barna Saha. 2022. Gap Edit Distance via Non-Adaptive Queries: Simple and Optimal. In *FOCS 2022*. IEEE, 674–685. https://doi.org/10.1109/FOCS54457.2022.00070

[37] Elazar Goldenberg, Tomasz Kociumaka, Robert Krauthgamer, and Barna Saha. 2023. An Algorithmic Bridge Between Hamming and Levenshtein Distances. In *ITCS 2023 (LIPIcs, Vol. 251)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 58:1–58:23. https://doi.org/10.4230/LIPIcs.ITCS.2023.58

[38] Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. 2019. Sublinear Algorithms for Gap Edit Distance. In *FOCS 2019*. IEEE Computer Society, 1101–1120. https://doi.org/10.1109/FOCS.2019.00070

[39] Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. 2020. Does preprocessing help in fast sequence comparisons?. In *STOC 2020*. ACM, 657–670. https://doi.org/10.1145/3357713.3384300

[40] Egor Gorbachev and Tomasz Kociumaka. 2025. Bounded Edit Distance: Optimal Static and Dynamic Algorithms for Small Integer Weights. arXiv:2404.06401

[41] Szymon Grabowski. 2016. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Appl. Math.* 212 (2016), 96–103. https://doi.org/10.1016/J.DAM.2015.10.040

[42] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press. https://doi.org/10.1017/CBO9780511574931

[43] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2022. Recent Advances in Fully Dynamic Graph Algorithms — A Quick Reference Guide. *ACM J. Exp. Algorithmics* 27 (2022), 1.11:1–1.11:45. https://doi.org/10.1145/3555806

[44] Steven Henikoff and Jorja G. Henikoff. 1992. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci.* 89, 22 (Nov. 1992), 10915–10919. https://doi.org/10.1073/pnas.89.22.10915

[45] Russell Impagliazzo and Ramamohan Paturi. 2001. On the Complexity of k-SAT. *J. Comput. Syst. Sci.* 62, 2 (2001), 367–375. https://doi.org/10.1006/jcss.2000.1727

[46] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. 2001. Which Problems Have Strongly Exponential Complexity? *J. Comput. Syst. Sci.* 63, 4 (2001), 512–530. https://doi.org/10.1006/jcss.2001.1774

[47] Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. 2005. Fully Incremental LCS Computation. In *FCT 2005 (LNCS, Vol. 3623)*. Springer, 563–574. https://doi.org/10.1007/11537311_49

[48] Ce Jin, Jelani Nelson, and Kewen Wu. 2021. An Improved Sketching Algorithm for Edit Distance. In *STACS 2021 (LIPIcs, Vol. 187)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 45:1–45:16. https://doi.org/10.4230/LIPIcs.STACS.2021.45

[49] Dan Jurafsky and James H. Martin. 2009. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall, Pearson Education International. https://www.worldcat.org/oclc/315913020

[50] Dominik Kempa and Tomasz Kociumaka. 2022. Dynamic suffix array with polylogarithmic queries and updates. In *STOC 2022*. ACM, 1657–1670. https://doi.org/10.1145/3519935.3520061

[51] Sung-Ryul Kim and Kunsoo Park. 2004. A dynamic edit distance table. *J. Discrete Algorithms* 2, 2 (2004), 303–312. https://doi.org/10.1016/S1570-8667(03)00082-0

[52] Philip N. Klein. 2005. Multiple-source shortest paths in planar graphs. In *SODA 2005* (Vancouver, British Columbia) *(SODA '05)*. Society for Industrial and Applied Mathematics, USA, 146–155.

[53] Tomasz Kociumaka. 2022. Recent advances in dynamic string algorithms. A survey talk at the STOC 2022 satellite workshop "Dynamic Algorithms: Recent Advances and Applications". https://www.youtube.com/watch?v=cvFTTPlZX5U

[54] Tomasz Kociumaka, Anish Mukherjee, and Barna Saha. 2023. Approximating Edit Distance in the Fully Dynamic Model. In *FOCS 2023*. IEEE, 1628–1638. https://doi.org/10.1109/FOCS57990.2023.00098

[55] Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya. 2021. Small-space and streaming pattern matching with k edits. In *FOCS 2021*. IEEE, 885–896.

https://doi.org/10.1109/FOCS52979.2021.00090
[56] Tomasz Kociumaka and Barna Saha. 2020. Sublinear-Time Algorithms for Computing & Embedding Gap Edit Distance. In *FOCS 2020*. IEEE, 1168–1179. https://doi.org/10.1109/FOCS46700.2020.00112
[57] Tomasz Kociumaka and Saeed Seddighin. 2021. Improved dynamic algorithms for longest increasing subsequence. In *STOC 2021*. ACM, 640–653. https://doi.org/10.1145/3406325.3451026
[58] Michal Koucký and Michael E. Saks. 2024. Almost Linear Size Edit Distance Sketch. In *STOC 2024*. https://doi.org/10.1145/3618260.3649783
[59] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. 1998. Incremental String Comparison. *SIAM J. Comput.* 27, 2 (1998), 557–582. https://doi.org/10.1137/S0097539794264810
[60] Gad M. Landau and Uzi Vishkin. 1988. Fast string matching with k differences. *J. Comput. System Sci.* 37, 1 (1988), 63–78. https://doi.org/10.1016/0022-0000(88)90045-1
[61] Vladimir I. Levenshtein. 1965. Binary codes capable of correcting deletions, insertions and reversals. *Dokl. Akad. Nauk SSSR* 163 (1965), 845–848. http://mi.mathnet.ru/eng/dan31411
[62] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press. https://doi.org/10.1017/CBO9781139940023
[63] William J. Masek and Mike Paterson. 1980. A Faster Algorithm Computing String Edit Distances. *J. Comput. System Sci.* 20, 1 (1980), 18–31. https://doi.org/10.1016/0022-0000(80)90002-1
[64] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. 1997. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica* 17 (1997), 183–198. https://doi.org/10.1007/bf02522825
[65] Michael Mitzenmacher and Saeed Seddighin. 2020. Dynamic algorithms for LIS and distance to monotonicity. In *STOC 2020*. ACM, 671–684. https://doi.org/10.1145/3357713.3384240
[66] Eugene W. Myers. 1986. An $O(ND)$ Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. https://doi.org/10.1007/BF01840446
[67] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48, 3 (1970), 443–453. https://doi.org/10.1016/b978-0-12-131200-8.50031-9
[68] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2020. Dynamic index and LZ factorization in compressed space. *Discrete Appl. Math.* 274 (2020), 116–129. https://doi.org/10.1016/j.dam.2019.01.014
[69] Luís M. S. Russo. 2012. Monge properties of sequence alignment. *Theor. Comput. Sci.* 423 (2012), 30–49. https://doi.org/10.1016/J.TCS.2011.12.068
[70] Wojciech Rytter. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.* 302, 1-3 (2003), 211–222. https://doi.org/10.1016/S0304-3975(02)00777-6
[71] Peter H. Sellers. 1974. On the Theory and Computation of Evolutionary Distances. *SIAM J. Appl. Math.* 26, 4 (1974), 787–793. https://doi.org/10.1137/0126070
[72] Peter H. Sellers. 1980. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *J. Algorithms* 1, 4 (1980), 359–373. https://doi.org/10.1016/0196-6774(80)90016-4
[73] Alexander Tiskin. 2008. Semi-local String Comparison: Algorithmic Techniques and Applications. *Math. Comput. Sci.* 1, 4 (2008), 571–603. https://doi.org/10.1007/s11786-007-0033-3
[74] Alexander Tiskin. 2015. Fast Distance Multiplication of Unit-Monge Matrices. *Algorithmica* 71, 4 (2015), 859–888. https://doi.org/10.1007/S00453-013-9830-Z
[75] Hélène Touzet. 2007. Comparing similar ordered trees in linear-time. *J. Discrete Algorithms* 5, 4 (2007), 696–705. https://doi.org/10.1016/J.JDA.2006.07.002
[76] Esko Ukkonen. 1985. Finding Approximate Patterns in Strings. *J. Algorithms* 6, 1 (1985), 132–137. https://doi.org/10.1016/0196-6774(85)90023-9
[77] Taras K. Vintsyuk. 1968. Speech discrimination by dynamic programming. *Cybernetics* 4, 1 (1968), 52–57. https://doi.org/10.1007/BF01074755
[78] Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *J. ACM* 21, 1 (1974), 168–173. https://doi.org/10.1145/321796.321811
[79] Michael S. Waterman. 1995. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Taylor & Francis. https://doi.org/10.1201/9780203750131

Received 2024-11-04; accepted 2025-02-01