

Bounded Weighted Edit Distance


Dynamic Algorithms and Matching Lower Bounds

Itai Boneh 

Reichman University, Herzliya, Israel
University of Haifa, Israel

Egor Gorbachev 

Saarland University and Max Planck Institute for Informatics, Saarland Informatics Campus,
Saarbrücken, Germany

Tomasz Kociumaka 

Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

The edit distance $\text{ed}(X, Y)$ of two strings $X, Y \in \Sigma^*$ is the minimum number of character edits (insertions, deletions, and substitutions) needed to transform X into Y . Its weighted counterpart $\text{ed}^w(X, Y)$ minimizes the total cost of edits, where the costs of individual edits, depending on the edit type and the characters involved, are specified using a function w , normalized so that each edit costs at least one. The textbook dynamic-programming procedure, given strings $X, Y \in \Sigma^{\leq n}$ and oracle access to w , computes $\text{ed}^w(X, Y)$ in $\mathcal{O}(n^2)$ time. Nevertheless, one can achieve better running times if the computed distance, denoted k , is small: $\mathcal{O}(n + k^2)$ for unit weights [Landau and Vishkin; JCSS'88] and $\tilde{\mathcal{O}}(n + \sqrt{nk^3})^1$ for arbitrary weights [Cassis, Kociumaka, Wellnitz; FOCS'23].

In this paper, we study the dynamic version of the weighted edit distance problem, where the goal is to maintain $\text{ed}^w(X, Y)$ for strings $X, Y \in \Sigma^{\leq n}$ that change over time, with each update specified as an edit in X or Y . Very recently, Gorbachev and Kociumaka [STOC'25] showed that the *unweighted* distance $\text{ed}(X, Y)$ can be maintained in $\tilde{\mathcal{O}}(k)$ time per update after $\tilde{\mathcal{O}}(n + k^2)$ -time preprocessing; here, k denotes the *current* value of $\text{ed}(X, Y)$. Their algorithm generalizes to small integer weights, but the underlying approach is incompatible with large weights.

Our main result is a dynamic algorithm that maintains $\text{ed}^w(X, Y)$ in $\tilde{\mathcal{O}}(k^{3-\gamma})$ time per update after $\tilde{\mathcal{O}}(nk^\gamma)$ -time preprocessing. Here, $\gamma \in [0, 1]$ is a real trade-off parameter and $k \geq 1$ is an integer threshold *fixed* at preprocessing time, with ∞ returned whenever $\text{ed}^w(X, Y) > k$. We complement our algorithm with conditional lower bounds showing fine-grained optimality of our trade-off for $\gamma \in [0.5, 1]$ and justifying our choice to fix k .

We also generalize our solution to a much more robust setting while preserving the fine-grained optimal trade-off. Our full algorithm maintains $X \in \Sigma^{\leq n}$ subject not only to character edits but also substring deletions and copy-pastes, each supported in $\tilde{\mathcal{O}}(k^2)$ time. Instead of dynamically maintaining Y , it answers queries that, given any string Y specified through a sequence of $\mathcal{O}(k)$ arbitrary edits transforming X into Y , in $\tilde{\mathcal{O}}(k^{3-\gamma})$ time compute $\text{ed}^w(X, Y)$ or report that $\text{ed}^w(X, Y) > k$.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases Edit distance, dynamic algorithms, conditional lower bounds

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.45

Related Version *Full Version:* <https://arxiv.org/abs/2507.02548v1> [6]

Funding *Itai Boneh:* supported by Israel Science Foundation grant 810/21.

Egor Gorbachev: This 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).

¹ Henceforth, the $\tilde{\mathcal{O}}(\cdot)$ notation hides factors poly-logarithmic in n .



© Itai Boneh, Egor Gorbachev, and Tomasz Kociumaka;
licensed under Creative Commons License CC-BY 4.0

33rd Annual European Symposium on Algorithms (ESA 2025).

Editors: Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman; Article No. 45; pp. 45:1–45:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Among the most fundamental string processing problems is the task of deciding whether two strings are similar to each other. A classic measure of string (dis)similarity is the *edit distance*, also known as the *Levenshtein distance* [22]. The (unit-cost) edit distance $\text{ed}(X, Y)$ of two strings X and Y is defined as the minimum number of character edits (insertions, deletions, and substitutions) needed to transform X into Y . In typical use cases, edits model naturally occurring modifications, such as typographical errors in natural-language texts or mutations in biological sequences. Some of these changes occur more frequently than others, which motivated introducing *weighted edit distance* already in the 1970s [33]. In this setting, each edit has a cost that may depend on its type and the characters involved (but nothing else), and the goal is to minimize the total cost of edits rather than their quantity. The costs of individual edits can be specified through a weight function $w: \bar{\Sigma}^2 \rightarrow \mathbb{R}_{\geq 0}$, where $\bar{\Sigma} = \Sigma \cup \{\varepsilon\}$ denotes the alphabet extended with a symbol ε representing the lack of a character. For $a, b \in \Sigma$, 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 $w(a, b)$. (Note that the cost of an edit is independent of the position of the edited character in the string.) Consistently with previous works, we assume that w is normalized: $w(a, b) \geq 1$ and $w(a, a) = 0$ hold for every distinct $a, b \in \bar{\Sigma}$. The unweighted edit distance constitutes the special case when $w(a, b) = 1$ holds for $a \neq b$.

The textbook dynamic-programming algorithm [32, 25, 26, 33] takes $\mathcal{O}(n^2)$ time to compute the edit distance of two strings of length at most n , and some of its original formulations incorporate weights [26, 33, 27]. Unfortunately, there is little hope for much faster solutions: for any constant $\delta > 0$, an $\mathcal{O}(n^{2-\delta})$ -time algorithm would violate the Orthogonal Vectors Hypothesis [1, 7, 2, 5], and hence the Strong Exponential Time Hypothesis [16, 17]. The lower bound holds already for unit weights and many other fixed weight functions [7].

Bounded Edit Distance. On the positive side, the quadratic running time can be improved when the computed distance is small. For unweighted edit distance, the algorithm by Landau and Vishkin [21], building upon the ideas of [30, 24], computes $k := \text{ed}(X, Y)$ in $\mathcal{O}(n + k^2)$ time for any two strings $X, Y \in \Sigma^{\leq n}$. This running time is fine-grained optimal: for any $\delta > 0$, a hypothetical $\mathcal{O}(n + k^{2-\delta})$ -time algorithm, even restricted to instances satisfying $k = \Theta(n^\kappa)$ for some constant $\kappa \in (0.5, 1]$, would violate the Orthogonal Vectors Hypothesis.

As far as the bounded *weighted* edit distance problem is concerned, a simple optimization by Ukkonen [30] improves the quadratic time complexity of the classic dynamic-programming algorithm to $\mathcal{O}(nk)$, where $k := \text{ed}^w(X, Y)$. Recently, Das, Gilbert, Hajiaghayi, Kociumaka, and Saha [12] developed an $\mathcal{O}(n + k^5)$ -time solution and, soon afterwards, Cassis, Kociumaka, and Wellnitz [8] presented an $\tilde{\mathcal{O}}(n + \sqrt{nk^3})$ -time algorithm. Due to the necessity to read the entire input, the latter running time is optimal for $k \leq \sqrt[3]{n}$. More surprisingly, there is a tight conditional lower bound for $\sqrt{n} \leq k \leq n$ [8], valid already for edit costs in the real interval $[1, 2]$. For any constants $\kappa \in [0.5, 1]$ and $\delta > 0$, an $\mathcal{O}(\sqrt{nk^{3-\delta}})$ -time algorithm restricted to instances satisfying $k = \Theta(n^\kappa)$ would violate the All-Pairs Shortest Paths Hypothesis [31]. The optimal complexity remains unknown for $\sqrt[3]{n} < k < \sqrt{n}$, where the upper bound is $\tilde{\mathcal{O}}(\sqrt{nk^3})$, yet the lower-bound construction allows for an $\tilde{\mathcal{O}}(n + k^{2.5})$ -time algorithm.

Dynamic Edit Distance. Over the last decades, the edit distance problem has been studied in numerous variants, including the dynamic ones, where the input strings change over time. The most popular dynamic (weighed) edit distance formulation is the following one [9, 15].

► **Problem 1.1** (Dynamic Weighted Edit Distance). *Given an integer n and oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, maintain strings $X, Y \in \Sigma^{\leq n}$ subject to updates (character edits in X and Y) and report $\text{ed}^w(X, Y)$ after each update.*²

The time complexity of Problem 1.1 is already well understood if it is measured solely in terms of the string length n . In case of unit weights, the algorithm of Charalampopoulos, Kociumaka, and Mozes [9] (building upon the techniques by Tiskin [28]) is characterized by $\tilde{O}(n)$ update time and $\mathcal{O}(n^2)$ preprocessing time; any polynomial-factor improvement would violate the lower bound for the static edit distance problem. As far as arbitrary weights are concerned, the approach presented in [9, Section 4] achieves $\tilde{O}(n\sqrt{n})$ update time after $\tilde{O}(n^2)$ -time preprocessing, and there is a matching fine-grained lower bound [8, Section 6] conditioned on the All-Pairs Shortest Paths Hypothesis.

In this work, we aim to understand Problem 1.1 for small distances:

How fast can one dynamically maintain $\text{ed}^w(X, Y)$ when this value is small?

Very recently, Gorbachev and Kociumaka [15] settled the parameterized complexity of Problem 1.1 in the *unweighted* case. Their solution achieves $\tilde{O}(n + k^2)$ preprocessing time and $\tilde{O}(k)$ update time; improving either complexity by a polynomial factor, even for instances satisfying $k = \Theta(n^\kappa)$ for some $\kappa \in (0, 1]$, would violate the Orthogonal Vectors Hypothesis.

Interestingly, the approach of [15] builds upon the static procedure for bounded *weighted* edit distance [8] and, as a result, the dynamic algorithm seamlessly supports *small integer* weights, achieving $\mathcal{O}(W^2k)$ update time for edit costs in $[1..W]$. Unfortunately, [15] does not provide any improvements for arbitrary weights; in that case, the results of [8] yield a solution to Problem 1.1 with $\tilde{O}(n)$ preprocessing time and $\tilde{O}(\min(k^3, \sqrt{nk^3}))$ update time. This is far from satisfactory: for $k \geq \sqrt[3]{n}$, the algorithm simply recomputes $\text{ed}^w(X, Y)$ from scratch after every update, and for $k < \sqrt[3]{n}$, it does not store (and reuse) anything beyond a dynamic data structure for checking the equality between fragments of X and Y .

Our Results: Lower Bounds

The reason why the approach of [15] is incompatible with large weights, e.g., $W = \Omega(k)$, is that their presence allows a single update to drastically change $\text{ed}^w(X, Y)$. Our first result strengthens the lower bound of [8] and shows that the $\tilde{O}(n + \sqrt{nk^3})$ -time static algorithm is fine-grained optimal already for instances that can be transformed from a pair of equal strings using four edits. The ideas behind the following theorem are presented in Section 5.

► **Theorem 1.2.** *Let $\kappa \in [0.5, 1]$ and $\delta > 0$ be real parameters. Assuming the APSP Hypothesis, there is no algorithm that, given two strings $X, Y \in \Sigma^{\leq n}$ satisfying $\text{ed}(X, Y) \leq 4$, a threshold $k \leq n^\kappa$, and oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, in time $\mathcal{O}(n^{0.5+1.5\kappa-\delta})$ decides whether $\text{ed}^w(X, Y) \leq k$.*

As a simple corollary, we conclude that significantly improving upon the naïve update time of $\tilde{O}(\sqrt{nk^3})$ for $\sqrt{n} \leq k \leq n$ requires large preprocessing time.

► **Corollary 1.3.** *Suppose that Problem 1.1 admits a solution with preprocessing time $T_P(n, \text{ed}^w(X, Y))$ and update time $T_U(n, \text{ed}^w(X, Y))$ for some non-decreasing functions T_P and T_U . If $T_P(n, 0) = \mathcal{O}(n^{0.5+1.5\kappa-\delta})$ and $T_U(n, n^\kappa) = \mathcal{O}(n^{0.5+1.5\kappa-\delta})$ hold for some real parameters $\kappa \in [0.5, 1]$ and $\delta > 0$, then the APSP Hypothesis fails.*

² For clarity, the introduction focuses on the version of the problem where the value $\text{ed}^w(X, Y)$ is reported after every update. In Theorem 1.7, we address a more general setting where queries may occur less frequently, allowing us to distinguish between update time and query time.

Proof idea. For an instance (X, Y) of the static problem of Theorem 1.2, we initialize the dynamic algorithm with (X, X) and transform one copy of X into Y using four updates. ◀

As discussed in the full version of the paper [6], instead of initializing the dynamic algorithm with (X, X) , we can initialize it with a pair of empty strings and gradually transform this instance to (X, X) using $\mathcal{O}(n)$ updates while keeping the weighted edit distance between 0 and 2 at all times. Hence, as long as the preprocessing of $(\varepsilon, \varepsilon)$ takes $n^{\mathcal{O}(1)}$ time and is not allowed to access the entire weight function (e.g., weights are revealed online when an update introduces a fresh character for the first time), we can replace $T_P(n, 0) = \mathcal{O}(n^{0.5+1.5\kappa-\delta})$ with $T_U(n, 2) = \mathcal{O}(n^{1.5\kappa-0.5-\delta})$ in the statement of Corollary 1.3.

Overall, we conclude that, in order to support updates with relatively large distances faster than naively (i.e., by computing $\text{ed}^w(X, Y)$ from scratch using a static algorithm), one needs to pay a lot for both preprocessing and updates while $\text{ed}^w(X, Y)$ is still very small. In particular, we should decide in advance how large distances need to be supported efficiently. This justifies a simplified variant of Problem 1.1, where the parameter k is fixed at preprocessing time. Here, instead of $\text{ed}^w(X, Y)$, the algorithm reports the following quantity:

$$\text{ed}_{\leq k}^w(X, Y) = \begin{cases} \text{ed}^w(X, Y) & \text{if } \text{ed}^w(X, Y) \leq k, \\ \infty & \text{otherwise.} \end{cases}$$

► **Problem 1.4** (Dynamic Weighted Edit Distance with Fixed Threshold). *Given integers $1 \leq k \leq n$ and oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, maintain strings $X, Y \in \Sigma^{\leq n}$ subject to updates (character edits in X and Y) and report $\text{ed}_{\leq k}^w(X, Y)$ upon each update.*

Corollary 1.3 can be rephrased in terms of Problem 1.4, but its major limitation is that it only applies to $k \geq \sqrt{n}$. To overcome this issue, we compose multiple hard instances from Theorem 1.2. This results in the following theorem, discussed in Section 5.

► **Theorem 1.5.** *Suppose that Problem 1.4 admits a solution with preprocessing time $T_P(n, k)$ and update time $T_U(n, k)$ for some non-decreasing functions T_P and T_U . If $T_P(n, n^\kappa) = \tilde{\mathcal{O}}(n^{1+\kappa\gamma})$ and $T_U(n, n^\kappa) = \mathcal{O}(n^{\kappa\cdot(3-\gamma)-\delta})$ hold for some parameters $\gamma \in [0.5, 1)$, $\kappa \in (0, 1/(3-2\gamma))$, and $\delta > 0$, then the APSP Hypothesis fails.*

Theorem 1.5 indicates that (conditioned on the APSP Hypothesis and up to subpolynomial-factor improvements), for a dynamic algorithm with preprocessing time $\tilde{\mathcal{O}}(nk^\gamma)$ for $\gamma \in [0.5, 1)$, the best possible update time is $\tilde{\mathcal{O}}(k^{3-\gamma})$. The lower bound does not extend to $\kappa > 1/(3-2\gamma)$; precisely then, the $\tilde{\mathcal{O}}(\sqrt{nk^3})$ -time static algorithm improves upon the $\tilde{\mathcal{O}}(k^{3-\gamma})$ update time.

Our Results: Algorithms

Our main positive result is an algorithm that achieves the fine-grained-optimal trade-off.

► **Theorem 1.6.** *There exists an algorithm that, initialized with an extra real parameter $\gamma \in [0, 1]$, solves Problem 1.4 with preprocessing time $\tilde{\mathcal{O}}(nk^\gamma)$ and update time $\tilde{\mathcal{O}}(k^{3-\gamma})$.*

As a warm-up, in Section 3, we prove Theorem 1.6 for $\gamma = 1$. Similarly to [15], the main challenge is that updates can make X and Y slide with respect to each other. If we interleave insertions at the end of Y and deletions at the beginning of Y , then, after $\Omega(k)$ updates, for every character $X[i]$, the fragment $Y[i-k..i+k]$ containing the plausible matches of $X[i]$ changes completely. The approach of [15] was to restart the algorithm every $\Theta(k)$ updates.

Our preprocessing time is too large for this, so we resort to a trick originating from dynamic edit distance approximation [20]: we maintain a data structure for aligning X with itself. Instead of storing Y , we show how to compute $\text{ed}_{\leq k}^w(X, Y)$ for any string Y specified using $\mathcal{O}(k)$ edits (of arbitrary costs) transforming X into Y . To find such edits in the original setting of Problem 1.4, one can use a dynamic *unweighted* edit distance algorithm. Already the folklore approach, based on [21, 23], with $\tilde{\mathcal{O}}(k^2)$ time per update is sufficiently fast.

In Section 4, we discuss how to generalize our dynamic algorithm to arbitrary $\gamma \in [0, 1]$. This is our most difficult result – it requires combining the techniques of [8] and their subsequent adaptations in [15] with several new ideas, including a novel *shifted* variant of self-edit distance. We derive Theorem 1.6 as a straightforward corollary of the following result, which demonstrates that the same fine-grained optimal trade-off can be achieved in a much more robust setting compared to the one presented in Problem 1.4.

► **Theorem 1.7.** *There is a dynamic algorithm that, initialized with a real parameter $\gamma \in [0, 1]$, integers $1 \leq k \leq n$, and $\mathcal{O}(1)$ -time oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, maintains a string $X \in \Sigma^{\leq n}$ and, after $\tilde{\mathcal{O}}(nk^\gamma)$ -time preprocessing, supports the following operations (updates and queries):*

- *Apply a character edit, substring deletion, or copy-paste to X in $\tilde{\mathcal{O}}(k^2)$ time.*
- *Given u edits transforming X into a string Y , compute $\text{ed}_{\leq k}^w(X, Y)$ in $\tilde{\mathcal{O}}(u + k^{3-\gamma})$ time.*

We remark that our query algorithm for $\gamma = 1$ extends a subroutine of [11], where the string X is static. In [11], this result is used for approximate pattern matching with respect to weighted edit distance. If the pattern is far from periodic, which is arguably true in most natural instances, the trade-off of Theorem 1.7 yields a faster pattern matching algorithm.

Open Problems

Recall that Theorem 1.5 proves fine-grained optimality of Theorem 1.6 only for $\gamma \in (0.5, 1)$. The omission of $\gamma \in [0, 0.5)$ stems from a gap between algorithms and lower bounds for the underlying static problem. Hence, we reiterate the open question of [8] asking for the complexity of computing $\text{ed}^w(X, Y)$ when $\sqrt[3]{n} < \text{ed}^w(X, Y) < \sqrt{n}$. Since our lower bounds arise from Theorem 1.2, it might be instructive to first try designing faster algorithms for the case of $\text{ed}(X, Y) \leq 4$. Before this succeeds, one cannot hope for faster dynamic algorithms.

Wider gaps in our understanding of Problem 1.4 remain in the regime of large preprocessing time. It is open to determine the optimal update time for $\gamma = 1$ and to decide whether $\gamma > 1$ can be beneficial. As far as $\gamma = 1$ is concerned, we believe that [9, Section 4] can be used to obtain $\tilde{\mathcal{O}}(k\sqrt{n})$ update time, which improves upon Theorem 1.6 for $k \geq \sqrt{n}$. Moreover, by monotonicity, the lower bound of [8, Section 6] excludes update times of the form $\mathcal{O}(k^{1.5-\delta})$.

Finally, we remark that all our lower bounds exploit the fact that $\text{ed}^w(X, Y)$ can change a lot as a result of a single update. It would be interesting to see what can be achieved if the weights are small (but fractional) or when the update time is parameterized by the change in $\text{ed}^w(X, Y)$. In these cases, we hope for non-trivial applications of the approach of [15].

2 Preliminaries³

A string $X \in \Sigma^n$ is a sequence of $|X| := n$ characters over an alphabet Σ . We denote the set of all strings over Σ by Σ^* , we use ε to represent the empty string, and we write $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$. For a *position* $i \in [0..n)$, we say that $X[i]$ is the i -th character of X . We

³ In this section, we partially follow the narration of [8, 15].

say that a string X occurs as a *substring* of a string Y , if there exist indices $i, j \in [0..|Y|]$ satisfying $i \leq j$ such that $X = Y[i] \cdots Y[j-1]$. The occurrence of X in Y specified by i and j is a *fragment* of Y denoted by $Y[i..j)$, $Y[i..j-1]$, $Y(i-1..j-1)$, or $Y(i-1..j)$.

Weighted Edit Distances and Alignment Graphs

Given an alphabet Σ , we set $\bar{\Sigma} := \Sigma \cup \{\varepsilon\}$. We call w a *normalized weight function* if $w: \bar{\Sigma} \times \bar{\Sigma} \rightarrow \mathbb{R}_{\geq 1} \cup \{0\}$ and, for $a, b \in \bar{\Sigma}$, we have $w(a, b) = 0$ if and only if $a = b$. Note that w does not need to satisfy the triangle inequality nor does w need to be symmetric.

We interpret weighted edit distances of strings as distances in alignment graphs.

► **Definition 2.1** (Alignment Graph, [8]). *For strings $X, Y \in \Sigma^*$ and a normalized weight function $w: \bar{\Sigma}^2 \rightarrow \mathbb{R}_{\geq 0}$, we define the alignment graph $\text{AG}^w(X, Y)$ to be a weighted directed graph with vertices $[0..|X|] \times [0..|Y|]$ and the following edges:*

- a horizontal edge $(x, y) \rightarrow (x+1, y)$ of cost $w(X[x], \varepsilon)$ representing a deletion of $X[x]$, for every $(x, y) \in [0..|X|] \times [0..|Y|]$,
- a vertical edge $(x, y) \rightarrow (x, y+1)$ of cost $w(\varepsilon, Y[y])$ representing an insertion of $Y[y]$, for every $(x, y) \in [0..|X|] \times [0..|Y|]$, and
- a diagonal edge $(x, y) \rightarrow (x+1, y+1)$ of cost $w(X[x], Y[y])$ representing a substitution of $X[x]$ into $Y[y]$ or, if $X[x] = Y[y]$, a match of $X[x]$ with $Y[y]$, for every $(x, y) \in [0..|X|] \times [0..|Y|]$.

We visualize the graph $\text{AG}^w(X, Y)$ as a grid graph with $|X|+1$ columns and $|Y|+1$ rows, where $(0, 0)$ and $(|X|, |Y|)$ are the top-left and bottom-right vertices, respectively.

Alignments between fragments of X and Y can be interpreted as paths in $\text{AG}^w(X, Y)$.

► **Definition 2.2** (Alignment). *Consider strings $X, Y \in \Sigma^*$ and a normalized weight function $w: \bar{\Sigma}^2 \rightarrow \mathbb{R}_{\geq 0}$. An alignment \mathcal{A} of $X[x..x')$ onto $Y[y..y')$, denoted $\mathcal{A}: X[x..x') \rightsquigarrow Y[y..y')$, is a path from (x, y) to (x', y') in $\text{AG}^w(X, Y)$, often interpreted as a sequence of vertices. The cost of \mathcal{A} , denoted by $\text{ed}_{\mathcal{A}}^w(X[x..x'), Y[y..y'])$, is the total cost of edges belonging to \mathcal{A} .*

The *weighted edit distance* of strings $X, Y \in \Sigma^*$ with respect to a weight function w is $\text{ed}^w(X, Y) := \min \text{ed}_{\mathcal{A}}^w(X, Y)$ where the minimum is taken over all alignments \mathcal{A} of X onto Y . An alignment \mathcal{A} of X onto Y is *w-optimal* if $\text{ed}_{\mathcal{A}}^w(X, Y) = \text{ed}^w(X, Y)$.

We say that an alignment \mathcal{A} of X onto Y aligns fragments $X[x..x')$ and $Y[y..y')$ if $(x, y), (x', y') \in \mathcal{A}$. In this case, we also denote the cost of the *induced alignment* (the subpath of \mathcal{A} from (x, y) to (x', y')) by $\text{ed}_{\mathcal{A}}^w(X[x..x'), Y[y..y'])$.

A normalized weight function satisfying $w(a, b) = 1$ for distinct $a, b \in \bar{\Sigma}$ gives the *unweighted* edit distance (Levenshtein distance [22]). In this case, we drop the superscript w .

► **Definition 2.3** (Augmented Alignment Graph, [15]). *For strings $X, Y \in \Sigma^*$ and a weight function $w: \bar{\Sigma}^2 \rightarrow [0, W]$, the augmented alignment graph $\overline{\text{AG}}^w(X, Y)$ is obtained from $\text{AG}^w(X, Y)$ by adding, for every edge of $\text{AG}^w(X, Y)$, a back edge of weight $W + 1$.⁴*

The following fact shows several properties of $\overline{\text{AG}}^w(X, Y)$. Importantly, $\text{ed}^w(X[x..x'), Y[y..y'])$ is the distance from (x, y) to (x', y') in $\overline{\text{AG}}^w(X, Y)$.

⁴ These edges ensure that $\overline{\text{AG}}^w(X, Y)$ is strongly connected, and thus the distance matrices used throughout this work are Monge matrices (see Definition 2.11) rather than so-called partial Monge matrices.

► **Fact 2.4** ([15, Lemma 5.2]). Consider strings $X, Y \in \Sigma^*$ and a weight function $w: \bar{\Sigma}^2 \rightarrow [0, W]$. Every two vertices (x, y) and (x', y') of the graph $\overline{\text{AG}}^w(X, Y)$ satisfy the following properties:

Monotonicity. Every shortest path from (x, y) to (x', y') in $\overline{\text{AG}}^w(X, Y)$ is (weakly) monotone in both coordinates.

Distance preservation. If $x \leq x'$ and $y \leq y'$, then

$$\text{dist}_{\overline{\text{AG}}^w(X, Y)}((x, y), (x', y')) = \text{dist}_{\text{AG}^w(X, Y)}((x, y), (x', y')).$$

Path irrelevance. If $(x \leq x'$ and $y \geq y')$ or $(x \geq x'$ and $y \leq y')$, then every path from (x, y) to (x', y') in $\overline{\text{AG}}^w(X, Y)$ monotone in both coordinates is a shortest path between these two vertices.

Consistently with many edit-distance algorithms, we will often compute not only the distance from $(0, 0)$ to $(|X|, |Y|)$ in $\overline{\text{AG}}^w(X, Y)$ but the entire *boundary distance matrix* $\text{BM}^w(X, Y)$ that stores the distances from every *input* vertex on the top-left boundary to every *output* vertex on the bottom-right boundary of the graph $\overline{\text{AG}}^w(X, Y)$.

► **Definition 2.5** (Input and Output Vertices of an Alignment Graph, [15, Definition 5.5]). Let $X, Y \in \Sigma^*$ be two strings and $w: \bar{\Sigma}^2 \rightarrow [0, W]$ be a weight function. Furthermore, for fragments $X[x..x']$ and $Y[y..y']$, let $R = [x..x'] \times [y..y']$ be the corresponding rectangle in $\overline{\text{AG}}^w(X, Y)$. We define the input vertices of R as the sequence of vertices of $\overline{\text{AG}}^w(X, Y)$ on the left and top boundary of R in the clockwise order. Analogously, we define the output vertices of R as a sequence of vertices of $\overline{\text{AG}}^w(X, Y)$ on the bottom and right boundary of R in the counterclockwise order. Furthermore, the input and output vertices of $X \times Y$ are the corresponding boundary vertices of the whole graph $\overline{\text{AG}}^w(X, Y)$.

► **Definition 2.6** (Boundary Distance Matrix, [15, Definition 5.6]). Let $X, Y \in \Sigma^*$ be two strings and $w: \bar{\Sigma}^2 \rightarrow [0, W]$ be a weight function. The boundary distance matrix $\text{BM}^w(X, Y)$ of X and Y with respect to w is a matrix M of size $(|X| + |Y| + 1) \times (|X| + |Y| + 1)$, where $M_{i,j}$ is the distance from the i -th input vertex to the j -th output vertex of $X \times Y$ in $\overline{\text{AG}}^w(X, Y)$.

The following fact captures the simple dynamic programming algorithm for bounded weighted edit distance.

► **Fact 2.7** ([15, Fact 3.3]). Given strings $X, Y \in \Sigma^*$, an integer $k \geq 1$, and $\mathcal{O}(1)$ -time oracle access to a normalized weight function $w: \bar{\Sigma}^2 \rightarrow \mathbb{R}_{\geq 0}$, the value $\text{ed}_{\leq k}^w(X, Y)$ can be computed in $\mathcal{O}(\min(|X| + 1, |Y| + 1) \cdot \min(k, |X| + |Y| + 1))$ time. Furthermore, if $\text{ed}^w(X, Y) \leq k$, the algorithm returns a w -optimal alignment of X onto Y .

The proof of Fact 2.7 relies on the fact that a path of length k can visit only a limited area of the alignment graph $\text{AG}^w(X, Y)$. This property can be formalized as follows:

► **Fact 2.8** ([15, Lemma 5.3]). Let $X, Y \in \Sigma^*$ be two strings, k be an integer, and $w: \bar{\Sigma}^2 \rightarrow \mathbb{R}_{\geq 0}$ be a normalized weight function. Consider a path P of cost at most k connecting (x, y) to (x', y') in $\overline{\text{AG}}^w(X, Y)$. All vertices $(x^*, y^*) \in P$ satisfy $|(x - y) - (x^* - y^*)| \leq k$ and $|(x' - y') - (x^* - y^*)| \leq k$.

The *breakpoint representation* of an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^m$ of X onto Y is the subsequence of \mathcal{A} consisting of pairs (x_t, y_t) such that $t \in \{0, m\}$ or \mathcal{A} does not match $X[x_t]$ with $Y[y_t]$. Note that the size of the breakpoint representation is at most $2 + \text{ed}_{\mathcal{A}}(X, Y)$ and that it can be used to retrieve the entire alignment and its cost: for any two consecutive

elements $(x', y'), (x, y)$ of the breakpoint representation, it suffices to add $(x - \delta, y - \delta)$ for $\delta \in (0, \max(x - x', y - y'))$. We will also talk of *breakpoint representations* of paths in $\overline{\text{AG}}^w(X, Y)$ that are weakly monotone in both coordinates. If such a path starts in (x, y) and ends in (x', y') with $x \leq x'$ and $y \leq y'$, then the breakpoint representation of such a path is identical to the breakpoint representation of the corresponding alignment. Otherwise, if $x > x'$ or $y > y'$, we call the breakpoint representation of such a path a sequence of all its vertices; recall that all edges in such a path have strictly positive weights.

Planar Graphs, the Monge Property, and Min-Plus Matrix Multiplication

As alignment graphs are planar, we can use the following tool of Klein.

► **Fact 2.9** (Klein [19]). *Given a directed planar graph G of size n with non-negative edge weights, we can construct in $\mathcal{O}(n \log n)$ time a data structure that, given two vertices $u, v \in V(G)$ at least one of which lies on the outer face of G , computes the distance $\text{dist}_G(u, v)$ in $\mathcal{O}(\log n)$ time. Moreover, the data structure can report the shortest $u \rightsquigarrow v$ path P in $\mathcal{O}(|P| \log \log \Delta(G))$ time, where $\Delta(G)$ is the maximum degree of G .*

The following lemma specifies a straightforward variation of Klein's algorithm tailored for path reconstruction in alignment graphs.

► **Lemma 2.10.** *Given strings $X, Y \in \Sigma^+$ and $\mathcal{O}(1)$ -time access to a normalized weight function $w: \Sigma^2 \rightarrow [0, W]$, we can construct in $\mathcal{O}(|X| \cdot |Y| \log^2(|X| + |Y|))$ time a data structure that, given two vertices $u, v \in \overline{\text{AG}}^w(X, Y)$ at least one of which lies on the outer face of $\overline{\text{AG}}^w(X, Y)$, computes the distance $\text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v)$ in $\mathcal{O}(\log(|X| + |Y|))$ time. Moreover, the breakpoint representation of a shortest $u \rightsquigarrow v$ path can be constructed in $\mathcal{O}((1 + \text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v)) \log^2(|X| + |Y|))$ time.*

Proof. The data structure we build is recursive. We first build the data structure of Klein's algorithm (Fact 2.9) for $\overline{\text{AG}}^w(X, Y)$, and then if $|X| > 1$, recurse onto $(X[0.. \lfloor |X|/2 \rfloor], Y)$ and $(X[\lfloor |X|/2 \rfloor.. |X|], Y)$. Preprocessing costs $\mathcal{O}(|X| \cdot |Y| \log^2(|X| + |Y|))$ time. For distance queries, the top-level instance of Klein's algorithm is sufficient. It remains to answer path reconstruction queries. We reconstruct the shortest $u \rightsquigarrow v$ path recursively. First, in $\mathcal{O}(\log(|X| + |Y|))$ time we query $\text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v)$. If the distance is 0, we return the breakpoint representation of the trivial shortest $u \rightsquigarrow v$ path. Otherwise, we make further recursive calls. If u and v both lie in $X[0.. \lfloor |X|/2 \rfloor] \times Y$ or both lie in $X[\lfloor |X|/2 \rfloor.. |X|] \times Y$, by the monotonicity property of Fact 2.4 we can recurse onto the corresponding small recursive data structure instance. Otherwise, every $u \rightsquigarrow v$ path contains a vertex (x, y) with $x = \lfloor |X|/2 \rfloor$. Furthermore, due to Fact 2.8, there are $\mathcal{O}(\text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v))$ possible values for y . In $\mathcal{O}(\text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v) \log(|X| + |Y|))$ time we query $\text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, (x, y))$ and $\text{dist}_{\overline{\text{AG}}^w(X, Y)}((x, y), v)$ using the smaller recursive data structure instances and find some y satisfying $\text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v) = \text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, (x, y)) + \text{dist}_{\overline{\text{AG}}^w(X, Y)}((x, y), v)$. We then recursively find the breakpoint representations of the optimal $u \rightsquigarrow (x, y)$ path and the optimal $(x, y) \rightsquigarrow v$ path in the smaller recursive data structure instances. There are $\mathcal{O}(\log(|X| + 1))$ recursive levels, on each one of them our total work is $\mathcal{O}((1 + \text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v)) \log(|X| + |Y|))$; thus, the total time complexity is $\mathcal{O}((1 + \text{dist}_{\overline{\text{AG}}^w(X, Y)}(u, v)) \log^2(|X| + |Y|))$. ◀

As discussed in [13, Section 2.3] and [15, Fact 3.7 and Section 5.1], the planarity of alignment graphs implies that the $\text{BM}^w(X, Y)$ matrices satisfy the following *Monge property*:

► **Definition 2.11** (Monge Matrix). *A matrix A of size $p \times q$ is a Monge matrix if, for all $i \in [0..p-1]$ and $j \in [0..q-1]$, we have $A_{i,j} + A_{i+1,j+1} \leq A_{i,j+1} + A_{i+1,j}$.*

In the context of distance matrices, concatenating paths corresponds to the min-plus product of matrices; this operation preserves Monge property and can be evaluated fast.

► **Fact 2.12** (Min-Plus Matrix Product, SMAWK Algorithm [3], [29, Theorem 2]). *Let A and B be matrices of size $p \times q$ and $q \times r$, respectively. Their min-plus product is a matrix $A \otimes B := C$ of size $p \times r$ such that $C_{i,k} = \min_j A_{i,j} + B_{j,k}$ for all $i \in [0..p)$ and $k \in [0..r)$.*

If A and B are Monge matrices, then C is also a Monge matrix. Moreover, C can be constructed in $\mathcal{O}(pr + \min(pq, qr))$ time assuming $\mathcal{O}(1)$ -time random access to A and B .

3 $\tilde{\mathcal{O}}(k^2)$ -Time Updates after $\tilde{\mathcal{O}}(nk)$ -Time Preprocessing

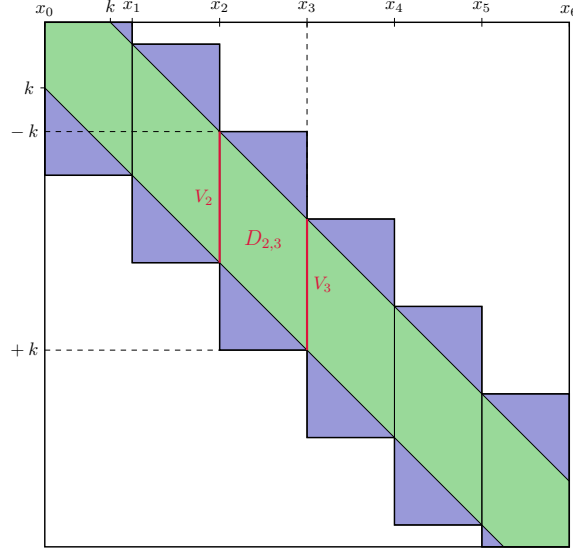
In this section, we present a dynamic algorithm that maintains $\text{ed}_{\leq k}^w(X, Z)$ for two dynamic strings $X, Z \in \Sigma^{\leq n}$ with $\tilde{\mathcal{O}}(nk)$ preprocessing time and $\tilde{\mathcal{O}}(k^2)$ update time. As all vertical and horizontal edges cost at least 1, the relevant part of the graph $\overline{\text{AG}}^w(X, Z)$ has size $\mathcal{O}(nk)$ (i.e., a band of width $\Theta(k)$ around the main diagonal), and thus we can afford to preprocess this part of $\overline{\text{AG}}^w(X, Z)$. Nevertheless, the curse of dynamic algorithms for bounded weighted edit distance is that, after $\Theta(k)$ updates (e.g., $\Theta(k)$ character deletions from the end of Z and $\Theta(k)$ character insertions at the beginning of Z), the preprocessed part of the original graph $\overline{\text{AG}}^w(X, Z)$ can be completely disjoint from the relevant part of the updated graph $\overline{\text{AG}}^w(X, Z)$, which renders the original preprocessing useless. To counteract this challenge, we use the idea from [20]: we dynamically maintain information about $\overline{\text{AG}}^w(X, X)$ instead of $\overline{\text{AG}}^w(X, Z)$. Every update to X now affects both dimensions of $\overline{\text{AG}}^w(X, X)$, so the relevant part of the graph does not shift anymore, and hence our preprocessing does not expire. Tasked with computing $\text{ed}_{\leq k}^w(X, Z)$ upon a query, we rely on the fact that, unless $\text{ed}_{\leq k}^w(X, Z) = \infty$, the graph $\overline{\text{AG}}^w(X, Z)$ is similar to $\overline{\text{AG}}^w(X, X)$.

Consistently with the previous works [8, 15], we cover the relevant part of $\overline{\text{AG}}^w(X, X)$ with $m = \mathcal{O}(n/k)$ rectangular subgraphs G_i of size $\Theta(k) \times \Theta(k)$ each; see Figure 1. We preprocess each subgraph in $\tilde{\mathcal{O}}(k^2)$ time. A single update to X alters a constant number of subgraphs G_i , so we can repeat the preprocessing for such subgraphs in $\tilde{\mathcal{O}}(k^2)$ time. The string Z does not affect any subgraph G_i ; we only store it in a dynamic strings data structure supporting efficient substring equality queries for the concatenation $X \cdot Z$.

Let V_i for $i \in [1..m)$ be the set of vertices in the intersection of G_i and G_{i+1} ; see Figure 1. Furthermore, let $V_0 := \{(0, 0)\}$ and $V_m := \{(|X|, |X|)\}$. Let G be the union of all graphs G_i . Let $D_{i,j}$ for $i, j \in [0..m]$ with $i < j$ denote the matrix of pairwise distances from V_i to V_j in G . Note that $D_{a,c} = D_{a,b} \otimes D_{b,c}$ holds for all $a, b, c \in [0..m]$ with $a < b < c$.

Upon a query for $\text{ed}^w(X, Z)$, we note that only $\mathcal{O}(\text{ed}(X, Z))$ subgraphs G_i in $\overline{\text{AG}}^w(X, X)$ differ from the corresponding subgraphs in $\overline{\text{AG}}^w(X, Z)$. (If $\text{ed}(X, Z) > k$, then $\text{ed}_{\leq k}^w(X, Z) = \infty$, so we can assume that $\text{ed}(X, Z) \leq k$.) We call such subgraphs *affected*. Let G'_i , V'_i , and $D'_{i,j}$ be the analogs of G_i , V_i , and $D_{i,j}$ in $\overline{\text{AG}}^w(X, Z)$. Note that if $\text{ed}^w(X, Z) \leq k$, then $\text{ed}^w(X, Z)$ is the only entry of the matrix $D'_{0,m}$. Furthermore, we have $D'_{0,m} = D'_{0,1} \otimes \dots \otimes D'_{m-1,m}$. We compute this product from left to right. For unaffected subgraphs G_i , we have $D'_{i-1,i} = D_{i-1,i}$, and we can use precomputed information about $\overline{\text{AG}}^w(X, X)$. To multiply by $D'_{i-1,i}$ for affected subgraphs G_i , we use the following lemma, the proof of which can be found in the full version of the paper [6].

► **Lemma 3.1** (Generalization of [11, Claim 4.2]). *Let $X, Y \in \Sigma^+$ be nonempty strings and $w: \Sigma^2 \rightarrow [0, W]$ be a weight function. Given $\mathcal{O}(1)$ -time oracle access to w , the strings X and Y can be preprocessed in $\mathcal{O}(|X| \cdot |Y| \log^2(|X| + |Y|))$ time so that, given a string $Y' \in \Sigma^*$ and a vector v of size $|X| + |Y'| + 1$, the row $v^T \otimes \text{BM}^w(X, Y')$ can be computed in*



■ **Figure 1** The decomposition of the alignment graph of X onto X into subgraphs G_i of size $\Theta(k) \times \Theta(k)$ that cover the whole **stripe** of width $\Theta(k)$ around the main diagonal. Each matrix $D_{i,i+1}$ represents the pairwise distances from V_i to V_{i+1} .

$\mathcal{O}((\text{ed}(Y, Y') + 1) \cdot (|X| + |Y|) \log(|X| + |Y|))$ time. Furthermore, given an input vertex a and an output vertex b of $X \times Y'$, the shortest path from a to b in $\overline{\text{AG}}^w(X, Y')$ can be computed in the same time complexity.

There are $\text{ed}(X, Z)$ edits that transform $\overline{\text{AG}}^w(X, X)$ into $\overline{\text{AG}}^w(X, Z)$, and each edit affects a constant number of subgraphs G_i . Therefore, applying Lemma 3.1 for all affected subgraphs takes time $\tilde{\mathcal{O}}(k^2)$ in total. Between two consecutive affected subgraphs G_i and G_j , we have $D'_{i,j-1} = D_{i,j-1}$, and thus it is sufficient to store some range composition data structure over $(D_{i,i+1})_{i=0}^{m-1}$ to quickly propagate between subsequent affected subgraphs.

As the information we maintain dynamically regards $\overline{\text{AG}}^w(X, X)$ and does not involve the string Z , we may generalize the problem we are solving. All we need upon a query is an alignment of X onto Z of *unweighted* cost at most k . Such an alignment can be obtained, for example, from a dynamic bounded unweighted edit distance algorithm [15] in $\tilde{\mathcal{O}}(k)$ time per update. Alternatively, it is sufficient to have $\tilde{\mathcal{O}}(1)$ -time equality tests between fragments of X and Z . In this case, we can run the classical Landau–Vishkin [21] algorithm from scratch after every update in $\tilde{\mathcal{O}}(k^2)$ time to either get an optimal unweighted alignment or learn that $\text{ed}^w(X, Z) \geq \text{ed}(X, Z) > k$. Efficient fragment equality tests are possible in a large variety of settings [10] including the dynamic setting [23, 4, 14, 18], so we will not focus on any specific implementation. Instead, we assume that the string Z is already given as a sequence of $u \leq k$ edits of an unweighted alignment transforming X into Z , and our task is to find a w -optimal alignment. We are now ready to formulate the main theorem of this section.

► **Theorem 3.2.** *There is a dynamic data structure that, given a string $X \in \Sigma^*$, a threshold $k \geq 1$, and $\mathcal{O}(1)$ -time oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, can be initialized in $\mathcal{O}((|X| + 1)k \log^2(|X| + 2))$ time and allows for the following operations:*

- *Apply a character edit to X in $\mathcal{O}(k^2 \log^2(|X| + 2))$ time.*
- *Given $u \leq k$ edits transforming X into a string $Z \in \Sigma^*$, compute $\text{ed}_{\leq k}^w(X, Z)$ in $\mathcal{O}(k \cdot (u + \log(|X| + 2)) \cdot \log(|X| + 2))$ time. Furthermore, if $\text{ed}^w(X, Z) \leq k$, the algorithm returns the breakpoint representation of a w -optimal alignment of X onto Z .*

Proof Sketch. At the initialization phase, we decompose $\overline{AG}^w(X, X)$ into subgraphs G_i and, for each subgraph G_i , initialize the algorithm of Lemma 3.1 and compute $D_{i-1,i}$ using Lemma 2.10. We also build a dynamic range composition data structure over $(D_{i,i+1})_{i=0}^{m-1}$, implemented as a self-balancing binary tree. Given a range $[a..b)$, in $\mathcal{O}(\log n)$ time, this data structure returns (pointers to) $\ell = \mathcal{O}(\log n)$ matrices S_1, \dots, S_ℓ such that $D_{a,b} = S_1 \otimes \dots \otimes S_\ell$.

When an update to X comes, it affects a constant number of subgraphs G_i , which we locally adjust and recompute the initialization for in $\tilde{\mathcal{O}}(k^2)$ time.

Upon a query, we are tasked with computing $D'_{0,1} \otimes \dots \otimes D'_{m-1,m}$. We locate the $\mathcal{O}(u)$ affected subgraphs G_i and process them one-by-one. We use Lemma 3.1 to multiply the currently computed row $v^T = D'_{0,1} \otimes \dots \otimes D'_{i-2,i-1}$ by $D'_{i-1,i}$. The algorithm then uses the range composition data structure over $(D_{i,i+1})_{i=0}^{m-1}$ to obtain $\ell = \mathcal{O}(\log n)$ matrices S_1, \dots, S_ℓ that correspond to the segment of matrices between the current affected subgraph G_i and the next affected subgraph G_j . We use SMAWK algorithm (Fact 2.12) to multiply v^T by matrices S_1, \dots, S_ℓ . After processing all the matrices, we obtain $D'_{0,1} \otimes \dots \otimes D'_{m-1,m}$, the only entry of which is equal to $\text{ed}^w(X, Z)$ if $\text{ed}_{\leq k}^w(X, Z) \neq \infty$. The applications of Lemma 3.1 work in $\mathcal{O}(ku \log n)$ time in total. Each of the $\mathcal{O}(u)$ blocks of unaffected subgraphs is treated by invoking SMAWK algorithm (Fact 2.12) $\mathcal{O}(\log n)$ times for a total of $\mathcal{O}(ku \log n)$ time.

See the full version of the paper [6] for a formal proof including the alignment reconstruction procedure. \blacktriangleleft

We complete this section with a simple corollary improving the query time if $\text{ed}^w(X, Z) \ll k$.

Corollary 3.3. *There is a dynamic data structure that, given a string $X \in \Sigma^*$, a threshold $k \geq 1$, and $\mathcal{O}(1)$ -time oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$ can be initialized in $\mathcal{O}((|X| + 1)k \log^2(|X| + 2))$ time and allows for the following operations:*

- *Apply a character edit to X in $\mathcal{O}(k^2 \log^2(|X| + 2))$ time.*
- *Given $u \leq k$ edits transforming X into a string $Z \in \Sigma^*$, compute $\text{ed}_{\leq k}^w(X, Z)$ in $\mathcal{O}(1 + (u + d) \cdot (u + \log(|X| + 2)) \cdot \log(|X| + 2))$ time, where $d := \min(\text{ed}^w(X, Z), k)$. Furthermore, if $\text{ed}^w(X, Z) \leq k$, the algorithm returns the breakpoint representation of a w -optimal alignment of X onto Z .*

Proof. We maintain the data structures of Theorem 3.2 for thresholds $1, 2, 4, \dots, 2^{\lceil \log k \rceil}$. Upon a query, we query the data structures for subsequent thresholds starting from $2^{\lceil \log u \rceil}$ until we get a positive result or reach $2^{\lceil \log k \rceil}$. \blacktriangleleft

As discussed earlier, in a variety of settings, upon a query, in $\tilde{\mathcal{O}}(k^2)$ time we can compute an optimal unweighted alignment of X onto Z or learn that $\text{ed}^w(X, Z) \geq \text{ed}(X, Z) > k$. Therefore, we may assume that whenever Corollary 3.3 is queried, we have $u = \text{ed}(X, Z) \leq k$. In this case, the query time complexity can be rewritten as $\mathcal{O}(1 + \min(\text{ed}^w(X, Z), k) \cdot (\text{ed}(X, Z) + \log(|X| + 2)) \cdot \log(|X| + 2))$.

Due to the robustness of the setting of Corollary 3.3, there are many problems beyond dynamic weighted edit distance that can be solved using Corollary 3.3. For example, given strings $X, Y_0, \dots, Y_{m-1} \in \Sigma^{\leq n}$, we can find $\text{ed}_{\leq k}^w(X, Y_i)$ for all $i \in [0..m)$ in $\tilde{\mathcal{O}}(nm + nk + mk^2)$ time, compared to $\tilde{\mathcal{O}}(m \cdot (n + \sqrt{nk^3}))$ time arising from m applications of the optimal static algorithm [8].

4 Trade-Off Algorithm: Technical Overview

In the full version of the paper [6], we generalize the result of Theorem 3.2 in two ways. First, we extend the set of updates the string X supports from character edits to *block edits*, which include substring deletions and copy-pastes. More importantly, rather than having $\tilde{\mathcal{O}}(nk)$

preprocessing time and $\tilde{O}(k^2)$ query time, we allow for a complete trade-off matching the lower bound of Theorem 1.5, i.e., $\tilde{O}(nk^\gamma)$ preprocessing and $\tilde{O}(k^{3-\gamma})$ update time for any $\gamma \in [0, 1]$.

A major challenge for $\gamma < 1$ is that we cannot preprocess the whole width- $\Theta(k)$ band around the main diagonal of $\overline{\text{AG}}^w(X, X)$. In the proof of Theorem 3.2, we cover the band with $\Theta(k) \times \Theta(k)$ -sized subgraphs G_i and preprocess each of them in $\tilde{O}(k^2)$ time for a total of $\tilde{O}(nk)$. To achieve $\tilde{O}(nk^\gamma)$ -time preprocessing, we are forced to instead use $\Theta(k^{2-\gamma}) \times \Theta(k^{2-\gamma})$ -sized subgraphs G_i and still spend just $\tilde{O}(k^2)$ preprocessing time per subgraph.

To remedy the shortage of preprocessing time, we use the ideas behind the recent static algorithms [8, 15]. At the expense of a logarithmic-factor slowdown, they allow us to answer queries only for “repetitive” fragments of X rather than the whole X . More precisely, we are tasked to answer queries involving fragments \hat{X} of X satisfying $\text{self-ed}(\hat{X}) \leq k$, where the *self-edit distance* $\text{self-ed}(\hat{X})$ is the minimum unweighted cost of an alignment of \hat{X} onto itself that does not match any character of \hat{X} with itself. We can use this to our advantage: the subgraphs G_i corresponding to more “repetitive” fragments X_i allow for more thorough preprocessing in $\tilde{O}(k^2)$ time since we can reuse some computations. On the other hand, upon a query, we can afford to spend more time on subgraphs G_i with less “repetitive” fragments X_i as their number is limited due to $\text{self-ed}(\hat{X}) \leq k$.

This approach requires the repetitiveness measure to be super-additive: $\sum_i \text{self-ed}(X_i) \leq \text{self-ed}(\hat{X})$. Unfortunately, self-edit distance is sub-additive: $\sum_i \text{self-ed}(X_i) \geq \text{self-ed}(\hat{X})$. To circumvent this issue, we introduce a better-suited notion of repetitiveness we call *k-shifted self-edit distance* $\text{self-ed}^k(\hat{X})$: a relaxed version of $\text{self-ed}(\hat{X})$ allowed to delete up to k first characters of \hat{X} and insert up to k last characters of \hat{X} for free. In contrast to self-edit distance, the k -shifted variant self-ed^k satisfies $\sum_i \text{self-ed}^k(X_i) \leq \text{self-ed}(\hat{X})$ if $\text{self-ed}(\hat{X}) \leq k$.

By using the value of $\text{self-ed}^k(X_i)$ to determine the level of preprocessing performed on G_i , upon a query, similarly to Theorem 3.2, we have $\mathcal{O}(k)$ “interesting” subgraphs G_i (subgraphs that are either affected by the input edits or satisfy $\text{self-ed}^k(X_i) > 0$) that we process one-by-one in time $\tilde{O}((u_i + \text{self-ed}^k(X_i)) \cdot k^{2-\gamma})$ each using a generalization of Lemma 3.1. The super-additivity property of self-ed^k guarantees that this sums up to a total of $\tilde{O}(k^{3-\gamma})$. The remaining subgraphs G_i are not affected by the input edits and satisfy $\text{self-ed}^k(X_i) = 0$, which means that X_i has a period of at most k . For such subgraphs G_i , we can afford the complete preprocessing and, upon a query, processing such subgraphs in chunks in $\tilde{O}(k^2)$ time in total precisely as in the proof of Theorem 3.2. Furthermore, similarly to Theorem 3.2, all the information the data structure stores is either local or cumulative, and thus block edits can be supported straightforwardly. This rather simple high-level approach runs into several low-level technical complications that are discussed in detail in the full version of the paper [6].

5 Lower Bounds: Technical Overview

As discussed in Section 1, our conditional lower bounds stem from the following result:

► **Theorem 1.2.** *Let $\kappa \in [0.5, 1]$ and $\delta > 0$ be real parameters. Assuming the APSP Hypothesis, there is no algorithm that, given two strings $X, Y \in \Sigma^{\leq n}$ satisfying $\text{ed}(X, Y) \leq 4$, a threshold $k \leq n^\kappa$, and oracle access to a normalized weight function $w: \overline{\Sigma}^2 \rightarrow \mathbb{R}_{\geq 0}$, in time $\mathcal{O}(n^{0.5+1.5\kappa-\delta})$ decides whether $\text{ed}^w(X, Y) \leq k$.*

The reduction provided in [8] is insufficient because the instances it produces satisfy $\text{ed}(X, Y) = \Theta(k)$ rather than $\text{ed}(X, Y) \leq 4$. Still, we reuse hardness of the following problem:

► **Problem 5.1** (Batched Weighted Edit Distance [8]). *Given a batch of strings $X_1, \dots, X_m \in \Sigma^x$, a string $Y \in \Sigma^y$, a threshold $k \in \mathbb{R}_{\geq 0}$, and oracle access to a weight function $w : \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, decide if $\min_{i=1}^m \text{ed}^w(X_i, Y) \leq k$.*

The hard instances of Problem 5.1, which cannot be solved in $\mathcal{O}(x^{2-\delta}\sqrt{m})$ time for any $\delta > 0$ assuming the APSP Hypothesis, satisfy many technical properties, including $h := \max_{i=1}^{m-1} \text{hd}(X_i, X_{i+1}) = \mathcal{O}(x/m)$, where $\text{hd}(X_i, X_{i+1})$ is the Hamming distance, i.e., the number of positions in which X_i differs from X_{i+1} .

The approach of [8, Section 7] is to construct strings \bar{X} and \bar{Y} that take the following form if we ignore some extra gadgets: $\bar{X} = X_1 Y X_2 \dots X_{m-1} Y X_m$ and $\bar{Y} = X_0^\perp Y X_1^\perp \dots X_{m-1}^\perp Y X_m^\perp$, where $X_i^\perp \in \Sigma^x$ is chosen so that $\text{ed}^w(X_i^\perp, X_{i-1}) = \text{ed}^w(X_i^\perp, X_i) = h$. The ignored gadgets let us focus on alignments $\mathcal{A}_i : \bar{X} \rightsquigarrow \bar{Y}$ that satisfy the following properties for $i \in [1..m]$:

- the fragment X_i is aligned with the copy of Y in \bar{Y} located between X_{i-1}^\perp and X_i^\perp ;
- the $m-1$ copies of Y in \bar{X} are matched with the remaining $m-1$ copies of Y in \bar{Y} ;
- all the characters within the fragments X_{i-1}^\perp and X_i^\perp of \bar{Y} are inserted;
- for $j \neq i$, the fragment X_j is aligned with X_{j-1}^\perp if $j < i$ and with X_j^\perp if $j > i$.

This ensures that the cost of \mathcal{A}_i is equal to a baseline cost (independent of i) plus $\text{ed}^w(X_i, Y)$.

Our reduction uses three types of “ X gadgets”: X_i , X_i^\perp , and X_i^\top , with the latter two playing similar roles. We also introduce a new symbol $\$$ that is extremely expensive to delete, insert, or substitute with any other symbol. Modulo some extra gadgets, our strings are of the following form:

$$\begin{aligned}\tilde{X} &= X_0^\top Y X_0^\perp Y \$ \bigodot_{i=1}^{m-1} \left(X_i Y X_i^\top Y X_i^\perp Y \right) X_m Y X_m^\top Y X_m^\perp \$, \\ \tilde{Y} &= \$ X_0^\top Y X_0^\perp Y \bigodot_{i=1}^{m-1} \left(X_i Y X_i^\top Y X_i^\perp Y \right) X_m Y X_m^\top \$ Y X_m^\perp.\end{aligned}$$

Notice that removing the two $\$$ symbols from \tilde{X} or from \tilde{Y} results in the same string, so we have $\text{ed}(\tilde{X}, \tilde{Y}) \leq 4$. Due to the expensive cost for editing a $\$$, any optimal alignment must match the two $\$$ symbols in \tilde{X} with the two $\$$ symbols in \tilde{Y} , deleting a prefix of \tilde{X} and a suffix of \tilde{Y} for some predictable cost. Thus, we can focus our analysis on $\text{ed}^w(\hat{X}, \hat{Y})$, where

$$\hat{X} = X_1 Y X_1^\top Y X_1^\perp Y \dots X_m Y X_m^\top Y X_m^\perp \quad \text{and} \quad \hat{Y} = X_0^\top Y X_0^\perp Y X_1 Y \dots X_{m-1}^\perp Y X_m Y X_m^\top.$$

Observe that both \hat{X} and \hat{Y} consists of “ X gadgets” interleaved with Y , and that \hat{Y} contains one more copy of Y and one more “ X gadget”. Analogously to [8], the ignored extra gadgets let us focus on the alignments $\mathcal{A}_i : \hat{X} \rightsquigarrow \hat{Y}$ satisfying the following properties for $i \in [1..m]$:

- the fragment X_i in \hat{X} is aligned with the copy of Y in \hat{Y} located between X_{i-1}^\perp and X_{i-1}^\top ;
- the $3m-1$ copies of Y in \hat{X} are matched with the remaining $3m-1$ copies of Y in \hat{Y} ;
- all characters within the fragments X_{i-1}^\perp and X_{i-1}^\top of \hat{Y} are inserted;
- the remaining “ X gadgets” in \hat{X} are aligned with the remaining “ X gadgets” in \hat{Y} .

To perfectly mimic [8], we should ensure that all the “ X gadgets” that we can possibly align are at weighted edit distance h . We only managed to construct X_i^\perp and X_i^\top so that:

- $\text{ed}^w(X_i, X_{i-1}^\top) = \text{ed}^w(X_i, X_{i-1}^\perp) = \text{ed}^w(X_i^\top, X_i) = \text{ed}^w(X_i^\perp, X_i) = 2h$, and
- $\text{ed}^w(X_i^\top, X_{i-1}^\perp) = \text{ed}^w(X_i^\perp, X_i^\top) = 4h$.

Although we have two different distances between the relevant pairs of “ X gadgets”, it is easy to see the number of pairs of either type is constant across all the alignments \mathcal{A}_i . This is sufficient to guarantee that the cost of \mathcal{A}_i is equal to some baseline cost plus $\text{ed}^w(X_i, Y)$. Moreover, as in [8], the costs of alignments \mathcal{A}_i is $\mathcal{O}(x + mh) = \mathcal{O}(x)$ and, if we denote $n := |\tilde{X}| = |\tilde{Y}|$, then the lower bound derived from the lower bound for Problem 5.1 excludes running times of the form $\mathcal{O}(x^{2-\delta}\sqrt{m}) = \mathcal{O}(x^{2-\delta}\sqrt{n/x}) = \mathcal{O}(\sqrt{nx^{3-2\delta}})$.

Lower Bounds for Dynamic Weighed Edit Distance

From Theorem 1.2 we derive two dynamic lower bounds, each justifying a different limitation of our algorithm given in Theorem 1.6. Our first lower bound, formalized in the following lemma and proved in the full version of the paper [6], concerns our choice to fix a threshold k at the preprocessing phase. If, instead, we aimed for time complexities depending on the current value of $\text{ed}^w(X, Y)$, then, for sufficiently large values of $\text{ed}^w(X, Y)$, we could not achieve update times improving upon the static algorithm.

► **Lemma 5.2.** *Consider the following dynamic problem: Given an integer $n \geq 1$ and $\mathcal{O}(1)$ -time oracle access to a normalized weight function $w: \Sigma^2 \rightarrow \mathbb{R}_{\geq 0}$, maintain two initially empty strings $X, Y \in \Sigma^{\leq n}$ that throughout the lifetime of the algorithm satisfy $\text{ed}(X, Y) \leq 4$ subject to updates (character edits) and output $\text{ed}^w(X, Y)$ after every update. Suppose that there is an algorithm that solves this problem with $T_P(n)$ preprocessing and $T_U(n, \text{ed}^w(X, Y))$ update time, where T_U is non-decreasing. If $T_P(n) = \mathcal{O}(n^{0.5+1.5\kappa-\delta})$, $T_U(n, 2) = \mathcal{O}(n^{1.5\kappa-0.5-\delta})$, and $T_U(n, n^\kappa) = \mathcal{O}(n^{0.5+1.5\kappa-\delta})$ hold for some real parameters $\kappa \in [0.5, 1]$ and $\delta > 0$, then the APSP Hypothesis fails.*

Our second dynamic lower bound justifies the specific trade-off between the preprocessing and update time in Theorem 1.6. In simple words, we prove that, with preprocessing time $\tilde{\mathcal{O}}(nk^\gamma)$ for $\gamma \in [0.5, 1)$, the best possible update time is $\mathcal{O}(k^{3-\gamma-o(1)})$. For that, we note that Theorem 1.2 is based on a reduction from the Negative Triangle problem, which is *self-reducible*: solving m instances of bounded weighted edit distance from Theorem 1.2 is hence, in general, m times harder than solving a single instance. Given such m instances $(X_0, Y_0), \dots, (X_{m-1}, Y_{m-1})$, we initialize a dynamic algorithm with a pair of equal strings $\hat{X} = \hat{Y} = \bigodot_{i=0}^{m-1} (X_i \cdot \dagger)$, where \dagger is an auxiliary symbol that is very expensive to edit. For $i \in [0..m)$, in $\text{ed}(X_i, Y_i) = \mathcal{O}(1)$ updates, we can transform the fragment X_i of \hat{Y} into Y_i and retrieve $\text{ed}_{\leq k}^w(\hat{X}, \hat{Y}) = \text{ed}_{\leq k}^w(X_i, Y_i)$. By applying and reverting these updates for every $i \in [0..m)$, we can thus find $\text{ed}_{\leq k}^w(X_i, Y_i)$ for all i . If we pick $m := n/k^{3-2\gamma-2\delta}$ for an arbitrary small constant $\delta > 0$, then the static lower bound requires $m \cdot (\sqrt{(n/m)k^3})^{1-o(1)} \geq (nk^{\gamma+\delta})^{1-o(1)}$ total time. Our preprocessing time is asymptotically smaller, so, among $\mathcal{O}(m)$ updates, some must take $\Omega((\sqrt{(n/m)k^3})^{1-o(1)}) = \Omega(k^{3-\gamma-\delta-o(1)})$ time. See the full version of the paper [6] for a formal proof.

References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *FOCS 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.
- 2 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *STOC 2016*, pages 375–388. ACM, 2016. doi:10.1145/2897518.2897653.
- 3 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.
- 4 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA 2000*, pages 819–828. ACM/SIAM, 2000. URL: <http://dl.acm.org/citation.cfm?id=338219.338645>.
- 5 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

- 6 Itai Boneh, Egor Gorbachev, and Tomasz Kociumaka. Bounded weighted edit distance: Dynamic algorithms and matching lower bounds, 2025. [arXiv:2507.02548v1](#).
- 7 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *FOCS 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.
- 8 Alejandro Cassis, Tomasz Kociumaka, and Philip Wellnitz. Optimal algorithms for bounded weighted edit distance. In *FOCS 2023*, pages 2177–2187. IEEE, 2023. doi:10.1109/FOCS57990.2023.00135.
- 9 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. Dynamic string alignment. In *CPM 2020*, volume 161 of *LIPIcs*, pages 9:1–9:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.9.
- 10 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *FOCS 2020*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.
- 11 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Pattern matching under weighted edit distance. In *FOCS 2025*, 2025.
- 12 Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, and Barna Saha. Weighted edit distance computation: Strings, trees, and Dyck. In *STOC 2023*, STOC 2023, pages 377–390, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3564246.3585178.
- 13 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006. doi:10.1016/j.jcss.2005.05.007.
- 14 Paweł Gawrychowski, Adam Karczmarsz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In *SODA 2018*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.
- 15 Egor Gorbachev and Tomasz Kociumaka. Bounded edit distance: Optimal static and dynamic algorithms for small integer weights. In *57th Annual ACM Symposium on Theory of Computing, STOC 2025*, pages 2157–2166, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3717823.3718168.
- 16 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k -SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
- 17 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.
- 18 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *STOC 2022*, pages 1657–1670. ACM, 2022. doi:10.1145/3519935.3520061.
- 19 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *SODA*, pages 146–155. SIAM, 2005. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070454>.
- 20 Tomasz Kociumaka, Anish Mukherjee, and Barna Saha. Approximating edit distance in the fully dynamic model. In *FOCS 2023*, pages 1628–1638. IEEE, 2023. doi:10.1109/FOCS57990.2023.00098.
- 21 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. System Sci.*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.
- 22 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Dokl. Akad. Nauk SSSR*, 163:845–848, 1965. URL: <http://mi.mathnet.ru/eng/dan31411>.
- 23 Kurt Mehlhorn, Rajamani Sundar, and Christian Urig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17:183–198, 1997. doi:10.1007/bf02522825.
- 24 Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. doi:10.1007/BF01840446.

- 25 Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(3):443–453, 1970. doi:10.1016/b978-0-12-131200-8.50031-9.
- 26 Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26(4):787–793, 1974. doi:10.1137/0126070.
- 27 Peter H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980. doi:10.1016/0196-6774(80)90016-4.
- 28 Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Math. Comput. Sci.*, 1(4):571–603, 2008. doi:10.1007/s11786-007-0033-3.
- 29 Alexander Tiskin. Fast distance multiplication of unit-Monge matrices. *Algorithmica*, 71(4):859–888, 2015. doi:10.1007/S00453-013-9830-Z.
- 30 Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985. doi:10.1016/0196-6774(85)90023-9.
- 31 Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM*, 65(5):27:1–27:38, 2018. doi:10.1145/3186893.
- 32 Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968. doi:10.1007/BF01074755.
- 33 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.