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

Egor Gorbachev

□

□

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

Tomasz Kociumaka ⊠ ©

Max Planck Institute for Informatics, Saarland Informatics Campus, Germany

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 length-n strings in $\mathcal{O}(n^2)$ time, and one of the early 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 $\mathcal{O}(n+k^2)$ time, which is optimal (up to sub-polynomial factors and conditioned on OVH) as a function of n and k.

The dynamic version of the edit distance problem 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 $\widetilde{\mathcal{O}}(k^2)$ time, where $\widetilde{\mathcal{O}}(\cdot)$ hides poly log n factors. Recently, Charalampopoulos, Kociumaka, and Mozes [CPM'20] observed that a framework of Tiskin [SODA'10] yields a dynamic algorithm with an update time of $\widetilde{\mathcal{O}}(n)$. This is optimal in terms of n: significantly faster updates would improve upon the static $\mathcal{O}(n^2)$ -time algorithm and violate OVH. Nevertheless, the state-of-the-art update time of $\widetilde{\mathcal{O}}(\min\{n,k^2\})$ raised an exciting open question of whether $\widetilde{\mathcal{O}}(k)$ is possible. We answer this question in the affirmative: $\widetilde{\mathcal{O}}(k)$ worst-case update time can be achieved with a deterministic algorithm.

Surprisingly, our solution relies on tools introduced in the context of *static* algorithms for *weighted* edit distance, where the weight of each edit depends on the edit type and the characters involved. The textbook dynamic programming natively supports weights, but the Landau–Vishkin approach is restricted to the unweighted setting, and, for many decades, a simple $\mathcal{O}(nk)$ -time version of the dynamic programming procedure remained the fastest known algorithm for *bounded* weighted edit distance. Only recently, Das, Gilbert, Hajiaghayi, Kociumaka, and Saha [STOC'23] provided an $\mathcal{O}(n+k^5)$ -time algorithm; shortly afterward, Cassis, Kociumaka, and Wellnitz [FOCS'23] presented an $\widetilde{\mathcal{O}}(n+\sqrt{nk^3})$ -time solution and proved this runtime optimal for $\sqrt{n} \leq k \leq n$ (up to sub-polynomial factors and conditioned on the All-Pairs Shortest Paths Hypothesis).

In this paper, we focus on the special case of integer edit weights between 0 and W, and we show that the weighted edit distance can then be computed in $\widetilde{\mathcal{O}}(n+Wk^2)$ time and maintained dynamically in $\widetilde{\mathcal{O}}(W^2k)$ time per update. In the practically meaningful case of constant W, our complexities match the OVH-based conditional lower bounds for unweighted edit distance. To address the case of large W, we show that our static algorithm can be implemented in $\widetilde{\mathcal{O}}(n+k^{2.5})$ time, which is faster than all known alternatives if $k \leq \min\{W^2, \sqrt{n}\}$.

Our key novel insight is a combinatorial lemma that allows stitching optimal alignments between two pairs of strings into an optimal alignment between their concatenations: It suffices to naively concatenate the two partial alignments and then fix the result within a small (in terms of a certain compressibility measure) neighborhood of the stitching point. To exploit compressibility in dynamic edit distance maintenance, we apply balanced straight-line programs of Charikar, Lehman, Liu, Panigrahy, Prabhakaran, Sahai, and Shelat [STOC'02]. Our support of integer weights relies on the algorithm of Russo [SPIRE'10] for efficient (min, +)-multiplication of Monge matrices with small core. For large integer weights, we develop a novel procedure that substantially reduces the core of a Monge matrix while truthfully preserving all entries not exceeding a specified threshold.

Funding 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).

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

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 edit distance (also called Levenshtein distance [Lev65]) is among the most popular measures of string (dis)similarity. For two strings X and Y, the edit distance ed(X,Y) is the minimum number of character insertions, deletions, and substitutions (jointly called edits) needed to transform X into Y. The textbook dynamic programming algorithm, which takes $\mathcal{O}(n^2)$ time to compute the edit distance of two strings of length at most n, was independently discovered several times more than five decades ago [Vin68, NW70, Sel74, WF74]. Despite substantial efforts, its quadratic runtime has been improved only by a poly-logarithmic factor [MP80, Gra16]. Only in the last decade, fine-grained complexity theory provided a satisfactory explanation for this quadratic barrier: any polynomial-factor improvement would violate the Orthogonal Vectors Hypothesis [ABW15, BK15, AHWW16, BI18] and thus also a more famous Strong Exponential Time Hypothesis [IP01, IPZ01].

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 [Ukk85] and Myers [Mye86], Landau and Vishkin [LV88] presented an elegant $\mathcal{O}(n+k^2)$ -time algorithm for bounded edit distance. The fine-grained hardness of the unbounded version prohibits any polynomial-factor improvements upon this runtime: a hypothetical $\mathcal{O}(n+k^{2-\epsilon})$ -time algorithm, even restricted to instances satisfying $k = \Theta(n^{\kappa})$ for some constant $\frac{1}{2} < \kappa \le 1$, would immediately yield an $\mathcal{O}(n^{2-\epsilon})$ -time edit distance algorithm, violating the Orthogonal Vectors Hypothesis. Although the decades-old solution is conditionally optimal for the bounded edit distance problem, the last few years brought numerous exciting developments across multiple models of computation. This includes sketching and streaming algorithms [BZ16, JNW21, KPS21, BK23, KS24], sublinear-time approximation algorithms [GKS19, KS20, GKKS22, BCFN22a, BCFK24], algorithms for preprocessed strings [GRS20, BCFN22b], algorithms for compressed input [GKLS22], and quantum algorithms [GJKT24], just to mention a few settings. Multiple papers have also studied generalizations of the bounded edit distance problem, including those to weighted edit distance [GKKS23, DGH⁺23, CKW23], tree edit distance [Tou07, AJ21, DGH⁺22, DGH⁺23], and Dyck edit distance [BO16, FGK⁺23, Dür23, DGH⁺23].

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

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 [HHS22] 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 [MSU97], longest common prefix queries [MSU97, ABR00, GKK⁺18], text indexing [ABR00, GKK⁺15, NII⁺20, KK22], approximate pattern matching [CKW20, CGK⁺22, CKW22], suffix array maintenance [AB20, KK22], longest common substring [AB18, CGP20], longest increasing subsequence [MS20, KS21, GJ21], Lempel–Ziv factorization [NII⁺20], detection of repetitions [ABCK19], and last but not least edit distance [Tis08, CKM20, KMS23] (see [Koc22] 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 approach [LV88] using a dynamic strings implementation that can efficiently test equality between substrings. With the data structure of Mehlhorn, Sundar, and Uhrig [MSU97], one can already achieve the worst-case update time of $\tilde{\mathcal{O}}(k^2)$. Modern optimized alternatives [GKK+18, KK22] yield $\mathcal{O}(k^2 \log n)$ update time with high probability and $\mathcal{O}(k^2 \log^{1+o(1)} n)$ update time deterministically. Unfortunately, these results are not meaningful for $k \geq \sqrt{n}$: then, it is better to recompute the edit distance from scratch, in $\mathcal{O}(n+k^2) = \mathcal{O}(k^2)$ 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 with updates only at the endpoints of the maintained strings [LMS98, KP04, IIST05, Tis08]. The most general of these results is an algorithm by Tiskin [Tis08] that works in $\mathcal{O}(n)$ time per update subject to edits at all endpoints of both strings. More recently, Charalampopoulos, Kociumaka, and Mozes [CKM20] applied Tiskin's toolbox [Tis08, Tis15] in a dynamic edit distance algorithm that supports arbitrary updates in $\mathcal{O}(n\log^2 n)$ time. Any significantly better update time of $\mathcal{O}(n^{1-\epsilon})$ would immediately yield an $\mathcal{O}(n^{2-\epsilon})$ -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 $\widetilde{\mathcal{O}}(\min\{n,k^2\})$ time motivates the following tantalizing open question, explicitly posed in [Koc22]:

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

The first main result of this work is an affirmative answer to this question:

■ **Theorem 1.1.** There exists a deterministic dynamic algorithm that maintains strings $X, Y \in \Sigma^*$ subject to edits and, upon every update, computes $k := \operatorname{ed}(X,Y)$ in $\mathcal{O}(k \log^6 n)$ time, where n := |X| + |Y|. The algorithm can be initialized in $\mathcal{O}(n \log^{o(1)} n + k^2 \log^6 n)$ time and, along with $\operatorname{ed}(X,Y)$, it also outputs an optimal sequence of edits transforming X into Y.

We have not optimized the poly $\log n$ factors in our running times, and we believe they can be significantly reduced. At the same time, the fine-grained lower bounds prohibit improving the update time to $\widetilde{\mathcal{O}}(k^{1-\epsilon})$ for any $\epsilon > 0$, even for instances restricted to $k = \Theta(n^{\kappa})$ for any constant $0 < \kappa < 1$.

As we discuss next, our dynamic algorithm supports weighted edit distance with constant integer weights. In that case, not even a static $\widetilde{\mathcal{O}}(n+k^2)$ -time algorithm was known prior to our work.

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. Some early works [Sel74, WF74, Sel80] and many application-oriented textbooks [Wat95, Gus97, JM09, MBCT15] 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

¹ The $\widetilde{\mathcal{O}}(\cdot)$ notation hides factors poly-logarithmic in the input size n, that is, $\log^c n$ for any constant c.

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

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

 $a, b \in \Sigma$, 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 $ed^w(X,Y)$ of two strings X and Y is the minimum total weight of edits in an alignment of X onto Y. Consistently with previous works, we assume that w(a,a)=0 and $w(a,b)\geq 1$ hold for every $a,b\in \overline{\Sigma}$ with $a\neq b$. As a result, $\operatorname{\sf ed}^w(X,X)=0$ and $\operatorname{ed}(X,Y) \leq \operatorname{ed}^w(X,Y)$ hold for all strings $X,Y \in \Sigma^*$.

As shown in [Sel74, WF74], the textbook $\mathcal{O}(n^2)$ -time dynamic-programming algorithm natively supports arbitrary weights. A simple optimization, originating from [Ukk85], allows for an improved running time of $\mathcal{O}(nk)$ if $k := ed^w(X,Y)$ 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 [DGH⁺23] managed to improve upon the $\mathcal{O}(nk)$ time, albeit only for $k \leq \sqrt[4]{n}$: their algorithm runs in $\mathcal{O}(n+k^5)$ time. Soon afterward, Cassis, Kociumaka, and Wellnitz [CKW23] developed an $\widetilde{\mathcal{O}}(n+\sqrt{nk^3})$ -time algorithm; the runtime of this solution exceeds neither $\widetilde{\mathcal{O}}(n+k^3)$ nor $\widetilde{\mathcal{O}}(nk)$, and it interpolates smoothly between $\widetilde{\mathcal{O}}(n)$ for $k \leq \sqrt[3]{n}$ and $\widetilde{\mathcal{O}}(nk)$ for $k \geq n$. Unexpectedly, the running time of $\mathcal{O}(n+\sqrt{nk^3})$ is optimal for $\sqrt{n} \le k \le n$: any polynomial-factor improvement would violate the All-Pairs Shortest Paths Hypothesis [CKW23]. This result provides a strict separation between the weighted and the unweighted variants of the bounded edit distance problem. Besides settling the complexity for $\sqrt[3]{n} \le k \le \sqrt{n}$, where the conditional lower bound degrades to $(n+k^{2.5})^{1-o(1)}$, the most important open question posed in [CKW23] is the following one:

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

The only class systematically studied before this work consists of *uniform* weight functions that assign a fixed integer weight W_{indel} to all insertions and another fixed integer weight W_{sub} to all substitutions. Tiskin [Tis08] observed that the $\mathcal{O}(n+k^2)$ running time can be generalized from the unweighted case of $W_{\text{indel}} = W_{\text{sub}} = 1$ to $W_{\text{indel}}, W_{\text{sub}} = \mathcal{O}(1)$. In a more fine-grained study, Goldenberg, Kociumaka, Krauthgamer, and Saha [GKKS23] adapted the Landau-Vishkin algorithm [LV88] so that it runs in $\mathcal{O}(n+k^2/W_{\text{indel}})$ time for arbitrary integers $W_{\text{indel}}, W_{\text{sub}} \geq 1$.

In this work, we consider a much more general class of integer weight functions $w: \overline{\Sigma}^2 \to [0..W]$. Notably, weight functions arising in practice, such as those originating from the BLOSUM [HH92] and PAM [DSO78] substitution matrices commonly used for amino-acids, are equivalent (after normalization) to weight functions with small integer values (below 20). Moreover, small integer weights are already sufficient to violate the monotonicity property that the Landau-Vishkin algorithm [LV88] hinges on.⁵ We nevertheless show that a different approach still yields the following result:

Theorem 1.2. Given strings $X, Y \in \Sigma^{\leq n}$ and oracle access to a weight function $w : \overline{\Sigma}^2 \to [0..W]$, the weighted edit distance $k := ed^w(X, Y)$ can be computed in $\mathcal{O}(n + k^2 \min\{W, \sqrt{k} \log n\} \log^5 n)$ time. The algorithm also outputs a w-optimal sequence of edits transforming X into Y.

For $W = \mathcal{O}(1)$, the running time is $\widetilde{\mathcal{O}}(n+k^2)$, and any polynomial-factor improvement would violate the Orthogonal Vectors Hypothesis. Our algorithm also includes optimizations targeted at large integer weights: the resulting runtime of $\widetilde{\mathcal{O}}(n+k^{2.5})$ improves upon the upper bound of $\widetilde{\mathcal{O}}(n+\sqrt{nk^3})$ if $k \leq \sqrt{n}$ but does not circumvent the lower bound of [CKW23], which is $(n+k^{2.5})^{1-o(1)}$ for $k \leq \sqrt{n}$.

Our final result is that the dynamic algorithm of Theorem 1.1 generalizes to the setting of small integer weights. In particular, we achieve the optimal update time of $\mathcal{O}(k)$ if $W = \mathcal{O}(1)$.

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.

4 For $i, j \in \mathbb{Z}$, we write $[i..j) \coloneqq \{i, i+1, \ldots, j-1\}$ and $[i..j] \coloneqq \{i, i+1, \ldots, j\}$; we define (i..j) and (i..j] analogously.

5 As already observed in $[\mathrm{DGH}^{+}23]$, we have $2 = \mathrm{ed}^{w}(\mathrm{ab}, \mathrm{c}) < \mathrm{ed}^{w}(\mathrm{a}, \varepsilon) = 3$ if $w(\mathrm{b}, \varepsilon) = w(\mathrm{a}, \mathrm{c}) = 1$ and $w(\mathrm{a}, \varepsilon) = 3$.

Theorem 1.3. 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 $k := \operatorname{ed}^w(X,Y)$ in $\mathcal{O}(W^2k\log^6 n)$ time, where n := |X| + |Y|. The algorithm can be initialized in $\mathcal{O}(n\log^{o(1)} n + Wk^2\log^6 n)$ and, along with $\operatorname{ed}^w(X,Y)$, it also outputs a w-optimal sequence of edits transforming X into Y.

Open Questions

In the unweighted setting, the worst-case deterministic update time $\widetilde{\mathcal{O}}(k)$ of Theorem 1.1 does not give much room for improvement: the most pressing challenge is to reduce the $\mathcal{O}(\log^6 n)$ -factor overhead. For small integer weights, the dependency on the largest weight W remains to be studied. The linear dependency in the running time $\widetilde{\mathcal{O}}(n+Wk^2)$ of Theorem 1.2 feels justified, but we hope that the quadratic dependency in the update time $\widetilde{\mathcal{O}}(W^2k)$ of Theorem 1.3 can be reduced. An exciting open question is to establish the optimality of our $\widetilde{\mathcal{O}}(n+k^{2.5})$ -time static algorithm for large integer weights (for $k \leq \min\{W^2, \sqrt{n}\}$). This runtime matches the lower bound of [CKW23], but the hard instances constructed there crucially utilize fractional weights with denominators $\Omega(k)$. The complexity of dynamic bounded edit distance also remains widely open for large weights: we are only aware of how to achieve $\widetilde{\mathcal{O}}(k^{2.5})$ and $\widetilde{\mathcal{O}}(k^3)$ 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 some extra effort, the techniques developed in this paper should be able to improve upon these results for updates that do not change $\operatorname{ed}^w(X,Y)$ too much. We do not know, however, how to improve upon the aforementioned simple approach for updates that change $\operatorname{ed}^w(X,Y)$ by a constant factor (of the larger among the distances before and after the update).

2 Technical Overview

In this section, we provide an overview of our algorithms, highlighting the limitations of existing tools as well as novel techniques that we developed to address these challenges.

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 and a weight function $w : (\Sigma \cup \{\varepsilon\})^2 \to \mathbb{R}_{\geq 0}$, the alignment graph $AG^w(X,Y)$ is a directed grid graph with vertex set $[0..|X|] \times [0..|Y|]$ and the following edges:

- $(x,y) \to (x,y+1)$ of weight $w(\varepsilon,Y[y])$, representing an insertion of Y[y];
- $= (x,y) \rightarrow (x+1,y)$ 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].

Every alignment \mathcal{A} mapping X onto Y, denoted $\mathcal{A}: X \leadsto Y$, can be interpreted as a path in $AG^w(X,Y)$ from the top-left corner (0,0) to the bottom-right corner (|X|,|Y|), and the cost of the alignment, denoted $\operatorname{ed}_{\mathcal{A}}^w(X,Y)$ is the length of the underlying path. Consequently, the edit distance $\operatorname{ed}^w(X,Y)$ 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 $BM^w(X,Y)$ that stores the distance from every input vertex on the top-left boundary to every output vertex on the bottom-right boundary. To be more precise, the input vertices form a sequence $(0,|Y|), (0,|Y|-1), \ldots, (0,1), (0,0), (1,0), \ldots, (|X|-1,0), (|X|,0)$, whereas the output vertices form a sequence $(0,|Y|), (1,|Y|), \ldots, (|X|-1,|Y|), (|X|,|Y|), (|X|,|Y|-1)$,

..., (|X|, 1), (|X|, 0), and the distance from the *i*th input vertex to the *j*th input vertex is stored at $M_{i,j}$ if $M = BM^w(X, Y)$. Crucially, the planarity of $AG^w(X, Y)$ implies that M satisfies the *Monge property*, i.e., $M_{i,j} + M_{i+1,j+1} \leq M_{i,j+1} + M_{i+1,j}$ holds whenever all four entries are well-defined.⁶

Dynamic Algorithm for Unbounded Unweighted Edit Distance

Our solutions build upon the algorithm of Charalampopoulos, Kociumaka, and Mozes [CKM20], so we start with an overview of their approach. The key insight, originally due to Tiskin [Tis08, Tis15], is that the unweighted boundary distance matrix BM(X,Y) can be represented in $\mathcal{O}(|X|+|Y|)$ space and constructed in $\mathcal{O}(|X|+|Y|)$ time from $BM(X_L,Y)$ and $BM(X_R,Y)$ if $X=X_LX_R$ or from $BM(X, Y_L)$ and $BM(X, Y_R)$ if $Y = Y_L Y_R$. If we assume for simplicity that |X| = |Y| = n is a power of two, then BM(X,Y) can be constructed in $\widetilde{\mathcal{O}}(n^2)$ 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 $\mathcal{O}(n/2^s)$, and each of them is constructed in $\mathcal{O}(2^s)$ time; this yields a total update time of $\mathcal{O}(n)$. To support insertions and deletions as updates, the algorithm of [CKM20] 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 sufficient care, only $\mathcal{O}(1)$ fragments at each scale need to be changed upon each update, and, for s>0, each fragment at scale s is the concatenation of $\mathcal{O}(1)$ fragments at scale s-1. Thus, the simple strategy of recomputing all the affected boundary matrices still takes $\mathcal{O}(n)$ time per update.

Dynamic Bounded Edit Distance with Updates Restricted to Substitutions

Let us henceforth focus on a simplified task of maintaining the value $\operatorname{ed}(X,Y)$ under a promise that it never exceeds a fixed threshold k. If we restrict updates to substitutions, then achieving $\widetilde{\mathcal{O}}(k)$ update time is fairly easy. Consider a decomposition $X = \bigcup_{i=0}^{m-1} X_i$ of the string X into phrases $X_i = X[x_i..x_{i+1})$ of length $\Theta(k)$ each. Moreover, for each $i \in [0..m)$, 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 $\operatorname{AG}(X,Y)$ induced by $[x_i..x_{i+1}] \times [y_i..y'_{i+1}]$. Observe that the union G of all subgraphs G_i contains every vertex (x,y) with $|x - y| \le k$, so the promise $\operatorname{ed}(X,Y) \le k$ guarantees $\operatorname{ed}(X,Y) = \operatorname{dist}_G((0,0),(|X|,|Y|))$.

For each $i \in [0..m]$, let us 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 . Note that every $(0,0) \leadsto (|X|,|Y|)$ path in G visits all sets V_i for subsequent indices $i \in [0..m]$. Thus, if we define $D_{i,j}$ as the matrix of distances (in G) between vertices in V_i and V_j , then $D_{0,m} = \bigotimes_{i=0}^{m-1} D_{i,i+1}$, where \bigotimes denotes the (min, +) product of matrices. To maintain $D_{0,m}$ (and its only entry distG((0,0),(|X|,|Y|))), we can use a standard dynamic data structure for aggregation with respect to an associative operator. In other words, we store $D_{\ell,r}$ for every dyadic interval $[\ell..r)$. Even though the matrices $D_{\ell,r}$ are of size $\mathcal{O}(k) \times \mathcal{O}(k)$, just like the boundary distance matrices, they can be stored in $\mathcal{O}(k)$ space and multiplied in $\widetilde{\mathcal{O}}(k)$ time using [Tis08, Tis15].

 $^{^{6}}$ 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.

⁷ Dyadic intervals are of the form $[i \cdot 2^s ... \min\{(i+1) \cdot 2^s, m\})$ for $s \in [0..[\log m]]$ and $i \in [0...[m/2^s])$.

The complete algorithm maintains, for each $i \in [0..m)$, the boundary distance matrix $\mathrm{BM}(X_i,Y_i)$ using the algorithm of $[\mathrm{CKM20}]$ and, for each dyadic interval $[\ell..r)$, the matrix $D_{\ell,r}$. Each position of X is contained in a single fragment X_i and each position of Y is contained in constantly many fragments Y_i . Consequently, upon every update, we need to retrieve the updated matrices $\mathrm{BM}(X_i,Y_i)$ for constantly many indices $i \in [0..m)$, and then recompute $D_{\ell,r}$ for every dyadic interval $[\ell..r)$ containing i. For $[\ell..r) = \{i\}$, we simply retrieve an appropriate submatrix of $\mathrm{BM}(X_i,Y_i)$; otherwise, we compute $D_{\ell,r} = D_{\ell,q} \otimes D_{q,r}$ for $q \in (\ell..r)$ such that $[\ell..q)$ and [q..r) are both dyadic intervals. If we use the techniques of $[\mathrm{Tis}08, \mathrm{Tis}15]$ for storing and multiplying all distance matrices, the total update time is $\tilde{O}(k)$, whereas initialization takes $\tilde{O}(mk^2) = \tilde{O}(nk)$ time.

Challenges of Supporting Insertions and Deletions as Updates

Having seen how to maintain $\operatorname{ed}(X,Y)$ subject to substitutions, one would naturally ask if this approach can be generalized to handle insertions and deletions (collectively called *indels*) as updates. To a limited extent, this is indeed possible: if the algorithm needs to support at most k indels throughout its lifetime, we can initialize $Y_i = Y[y_i..y'_{i+1})$ with $y_i = \max\{x_i - 3k, 0\}$ and $y'_{i+1} = \min\{x_{i+1} + 3k, |Y|\}$ so that, even after k indels, the graph G is still guaranteed to contain every $(0,0) \leadsto (|X|,|Y|)$ path in $\operatorname{AG}(X,Y)$ of cost at most k. If each fragment X_i is initially of length at least 3k, then its length remains at least 2k, so we can still guarantee that each update in X affects a single fragment X_i and each update in Y affects constantly many fragments Y_i .

Unfortunately, after exceeding the limit of k indels, we would need to restart the whole algorithm and pay the initialization cost of $\tilde{\Theta}(nk)$. Amortized over the k edits, this is $\tilde{\Theta}(n)$ per update – no improvement over [CKM20]. A specific scenario that illustrates this challenge is when X stays unchanged, the k characters at the beginning of Y are deleted, and k other characters are inserted at the end of Y. In this case, the part of AG(X,Y) meaningful for computing ed(X,Y) before the 2k updates is disjoint from the one meaningful for computing ed(X,Y) after the 2k updates, and both contain $\Theta(nk)$ vertices that our initialization algorithm needs to visit.

At this point, we could try improving the initialization time beyond $\tilde{\mathcal{O}}(nk)$, ideally to $\tilde{\mathcal{O}}(k^2)$. If we knew that some matrix $D_{\ell,r}$ will not change throughout the algorithm's lifetime, we could construct it directly (using [CKW22, Lemma 8.23]) in time $\tilde{\mathcal{O}}(k^2)$, which improves upon $\tilde{\mathcal{O}}((r-\ell)\cdot k^2)$. Nevertheless, k edits may affect k distinct phrases X_i , and we need to construct $\mathrm{BM}(X_i,Y_i)$ for all of these affected phrases. This already takes $\tilde{\Theta}(k^3)$ time, which is $\tilde{\Theta}(k^2)$ amortized time per update – no improvement over [LV88, MSU97]. Consequently, overcoming the $\mathcal{O}(\min\{n,k^2\})$ update-time barrier poses a significant challenge even if the sequence of updates is known in advance.

Self-Edit Distance to the Rescue

Surprisingly, the techniques that allowed us to break through the aforementioned barrier originate from the recent *static* algorithm for *weighted* bounded edit distance [CKW23]. At a very high level, the novel insight in [CKW23] is that the problem of computing $\operatorname{ed}^w(X,Y) \leq k$ can be reduced to instances satisfying $\operatorname{self-ed}(X) \leq k$. The value $\operatorname{self-ed}(X)$, called the *self edit distance* of X, is the distance from (0,0) to (|X|,|X|) in the alignment graph $\operatorname{AG}(X,X)$ with the edges $(x,x) \to (x+1,x+1)$ on the main diagonal removed. Self edit distance can also be interpreted as a compressibility measure; in particular, the size of the Lempel–Ziv factorization [ZL77] of X satisfies $|\operatorname{LZ}(X)| \leq 2 \cdot \operatorname{self-ed}(X)$. This is because X can be factorized into $\operatorname{self-ed}(X)$ single characters and $\operatorname{self-ed}(X)$ fragments X[x, x'] with previous

⁸ For this, y_i and y'_{i+1} deviate from their initial definitions as the algorithm handles subsequent indels.

occurrences at most self-ed(X) positions earlier, i.e., such that X[x..x') = X[x-d..x'-d) for some $d \in [1...self-ed(X)]$. Based on the latter property, a subroutine of [CKW23] produces a decomposition $X = \bigoplus_{i=0}^{m-1} X_i$ into phrases $X_i = X[x_i, x_{i+1})$ of lengths in [3k, 6k) such that $X_i = X_{i-1}$ holds for all but $\mathcal{O}(k)$ indices i. Furthermore, if we define $Y_i = Y[y_i, y'_{i+1})$ as above, with $y_i = \max\{x_i - 3k, 0\}$ and $y'_{i+1} = \min\{x_{i+1} + 3k, |Y|\}$, then (due to $ed(X,Y) \leq k$) also $Y_i = Y_{i-1}$ holds for all but $\mathcal{O}(k)$ indices i. In particular, there is a set $F \subseteq [0..m)$ of size $\mathcal{O}(k)$ such that graphs G_i and G_{i-1} are isomorphic for $i \in [0..m) \setminus F$.

An $\widetilde{\mathcal{O}}(n+k^3)$ -time algorithm for weighted edit distance simply builds $\mathrm{BM}^w(X_i,Y_i)$ and $D_{i,i+1}$ for all $i \in F$ and then computes $D_{0,m} = \bigotimes_{i=0}^{m-1} D_{i,i+1}$ exploiting the fact that $D_{i,i+1} = D_{i-1,i}$ for $i \notin F$. The same approach can be used in a dynamic algorithm for the unweighted edit distance: as we compute the matrices $D_{\ell,r}$ for dyadic intervals $[\ell..r)$, we can benefit from the fact that $D_{\ell,r}$ is equal to $D_{2\ell-r,\ell}$ for the preceding dyadic interval $[2\ell-r,\ell]$ of the same length if $(2\ell-r,r)\cap F=\emptyset$. Consequently, even though the hierarchy of dyadic intervals consists of $\Theta(m)$ intervals, it can be stored using $\mathcal{O}(k \log m)$ memory locations provided that a single location may represent multiple intervals corresponding to isomorphic subgraphs of AG(X,Y). Since the dynamic algorithm of [CKM20] is fully persistent (that is, it allows queries and updates to all previous versions of the maintained pair of strings), we can start with $BM(X_i, Y_i)$ for $i \in F$ in the leaves of the hierarchy and then, when X_i or Y_i changes, create a new leaf with the updated $BM(X_i,Y_i)$ instead of modifying the original leaf. Using Tiskin's [Tis08, Tis15] toolkit for distance matrices, we achieve $\tilde{\mathcal{O}}(k)$ time per update after $\widetilde{\mathcal{O}}(k^3)$ -time initialization. Since the lifetime of the algorithm is restricted to k updates, the amortized update time is $\widetilde{\mathcal{O}}(k^2)$ – still no improvement over [LV88, MSU97]. The hope is not lost, however: due to self-ed(X) $\leq k$, the strings X_{i-1} and X_i typically have a lot in common even if they are not equal. If we look at the aforementioned factorization of X (into single characters and fragments with occurrences at most self-ed(X) positions earlier) and denote by k_i the number of factors with a non-empty intersection with X_i , then $\sum_{i \in F} k_i = \mathcal{O}(k)$. A similar property holds for Y_i because one can show that $\operatorname{self-ed}(Y) \leq \operatorname{self-ed}(X) + 2 \cdot \operatorname{ed}(X,Y) \leq 3k$ and each position of Y is contained in constantly many fragments Y_i . Our goal is to transform a dynamic data structure maintaining $\mathrm{BM}(X_{i-1},Y_{i-1})$ into one maintaining $\mathrm{BM}(X_i,Y_i)$ using just $\widetilde{\mathcal{O}}(k_i)$ operations implemented in $\widetilde{\mathcal{O}}(k)$ time each. This way, the total initialization will become $\tilde{O}(k^2)$, allowing for an amortized update time of $\mathcal{O}(k)$. Unfortunately, the necessary operations are of more powerful types than those supported in [CKM20]: to build $X_{i-1}X_i$ from X_{i-1} , we need a substring copy-paste operation (that appends a copy of a substring at the end of the maintained string), and then we need a prefix removal operation to extract X_i ; similarly, these two operation types are sufficient to obtain Y_i from Y_{i-1} .

More Powerful Updates using Balanced Straight Line Programs

Recall that the algorithm of [CKM20] maintains hierarchical decompositions of X and Y and stores the boundary distance matrix for every pair of fragments at the same level of the hierarchy. The substring copy-paste operation on X extracts a substring of X and appends its copy at the end of X. Even though this adds a lot of new fragments to the hierarchical decomposition of X, 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 of X not as a tree, but rather as a directed

Our bounds on the initialization time assume $\widetilde{\mathcal{O}}(1)$ -time substring equality queries (implemented using a dynamic strings data structure [MSU97, GKK⁺18, KK22], which does not need to be periodically reconstructed). Using these queries, the subroutines of [CKW23] construct the decomposition $X = \bigodot_{i=0}^{m-1} X_i$ and the set F in $\widetilde{\mathcal{O}}(k^2)$ time.

acyclic graph, where isomorphic subtrees of the tree are glued together. In the context of data compression, such a representation is called a straight-line grammar or, if the underlying tree is binary, 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 (ordered) sequence of outgoing edges. These properties are sufficient to guarantee that the language associated with each symbol consists of exactly one string called the *expansion* of the symbol.

Established tools [Ryt03, CLL+05] can be used to grow an SLP subject to operations that add a symbol A 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 $\mathcal{O}(\log \operatorname{len}(A))$ auxiliary symbols, where $\operatorname{len}(A)$ is the length of the expansion of A. Unfortunately, unlike the hierarchical decompositions of [CKM20], 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 specifically use the weight-balanced SLPs of [CLL⁺05] that satisfy an additional property: for every non-terminal A with production $A_L A_R$ (i.e., outgoing edges to A_L and A_R), we have $\frac{1}{3} \leq \text{len}(A_L)/\text{len}(A_R) \leq 3$. Now, it suffices 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 $\frac{1}{4} \leq$ $len(A)/len(B) \le 4$. Whenever we need to construct this matrix and $len(A) \ge len(B)$ without loss of generality, the symbols A_L , A_R in the production of A satisfy $\frac{1}{4} \text{len}(B) \leq \frac{1}{4} \text{len}(A) \leq \text{len}(A_L)$, $\text{len}(A_R) \leq \frac{1}{4} \text{len}(A) \leq \frac{1}{4}$ $len(A) \leq 4len(B)$, so the matrices for (A_L, B) and (A_R, B) are already available. Furthermore, if we make sure to prune unused symbols from the SLP of Y, then, for each symbol A in the SLP of X, the number of symbols B in the SLP of Y satisfying $\frac{1}{4} \leq \text{len}(A)/\text{len}(B) \leq 4$ is $\mathcal{O}(|Y|/\text{len}(A))$. For each of them, the construction of the boundary distance matrix (using [Tis08, Tis15]) takes $\widetilde{\mathcal{O}}(\mathsf{len}(A) + \mathsf{len}(B)) = \widetilde{\mathcal{O}}(\mathsf{len}(A))$ time, for a total of $\widetilde{\mathcal{O}}(|Y|)$ time across all relevant symbols B. Each operation on X creates $\mathcal{O}(\log |X|)$ symbols, so it can be implemented in $\mathcal{O}(|Y|\log |X|) = \mathcal{O}(n)$ time.

Divide and Conquer: Reduction to Small Self Edit Distance

The application of balanced SLPs improves initialization time to $\mathcal{O}(k^2)$ and amortized update time to $\widetilde{\mathcal{O}}(k)$, but these guarantees hold under an extra assumption that $\mathsf{self-ed}(X) \leq k$. Although [CKW23] provides a static reduction to this case, it is not clear at all how to make this reduction dynamic. To understand why, let us recall the divide-and-conquer recursive procedure behind this reduction. It partitions $X = X_L X_R$ into two halves of the same length (up to ± 1) and finds the longest suffix X_L^* of X_L satisfying self-ed $(X_L^*) \leq 11k$ and the longest prefix X_R^* of X_R satisfying self-ed $(X_R^*) \leq 11k$. Then, the algorithm constructs $X^* = X_L^* X_R^*$ and a substring Y^* obtained from Y by cutting off a prefix of length $|X_L| - |X_L^*|$ and a suffix of length $|X_R| - |X_R^*|$. The key observation are that $\operatorname{ed}(X^*,Y^*) \leq \operatorname{ed}(X,Y) \leq k$ and a vertex $(|X_L|,y)$ of $\operatorname{AG}^w(X,Y)$ belongs to an optimal alignment $X^* \rightsquigarrow Y^*$ if and only if $(|X_L|, y)$ also belongs to an optimal alignment $X \rightsquigarrow Y$. Consequently, given an optimal alignment $X^* \rightsquigarrow Y^*$, which can be computed efficiently due to $\operatorname{self-ed}(X^*) \leq \operatorname{self-ed}(X_L^*) + \operatorname{self-ed}(X_R^*) \leq 22k$ (and because the procedure described above can be extended to report an $\mathcal{O}(k)$ -size representation of an optimal alignment), we can find a decomposition $Y = Y_L Y_R = Y[0..y)Y[y..|Y|)$ such that $ed(X,Y) = ed(X_L,Y_L) + ed(X_R,Y_R)$. It then suffices to recurse on (X_L, Y_L) and (X_R, Y_R) and simply stitch the resulting alignments.¹⁰ Unfortunately, an update in X_L may affect the partition of Y, and this has cascading effects in the recursive call (X_R, Y_R) . To eliminate this pitfall, we design another reduction based on novel combinatorial insight.

 $^{^{10}}$ Recursion is more complex because we do not know a priori how to partition the budget k among the recursive calls.

Our approach relies on an approximately optimal alignment of cost $\mathcal{O}(k)$ that we can afford to maintain and that remains relatively stable as the algorithm handles subsequent updates: If we find a cost- $\mathcal{O}(k)$ alignment \mathcal{A} at initialization time (e.g., using the Landau-Vishkin algorithm [LV88]), then the cost of \mathcal{A} (minimally adjusted to take updated characters into account) remains $\mathcal{O}(k)$ for k subsequent updates. Thus, we henceforth assume that our dynamic algorithm has access to an alignment $A: X \leadsto Y$ of cost $\mathcal{O}(k)$ and needs to maintain an optimal alignment $\mathcal{O}: X \leadsto Y$ subject to k updates. As noted in [CKW23], self edit distance quantifies how often the two alignments (viewed as paths in AG(X,Y)) need to intersect. Specifically, if (x,y) and (x',y') are two subsequent intersection points, then self-ed $(X[x..x']) \le 2ed_{\mathcal{A}}(X[x..x'], Y[y..y'])$, where $ed_{\mathcal{A}}(X[x..x'], Y[y..y'])$ denotes the cost that \mathcal{A} pays for the subpath from (x,y) to (x',y'). Informally, the optimal alignment \mathcal{O} deviates from \mathcal{A} only in small (in terms of self edit distance) regions around edits in \mathcal{A} .

Keeping track of all such regions at once is a tedious task, so we resort to a divide-and-conquer strategy again. We still partition $X = X_L X_R$ into two halves of equal length (up to ± 1), but the partition $Y = Y_L Y_R$ is simply determined by \mathcal{A} so that $\operatorname{ed}_{\mathcal{A}}(X,Y) = \operatorname{ed}_{\mathcal{A}}(X_L,Y_L) + \operatorname{ed}_{\mathcal{A}}(X_R,Y_R)$. With recursive calls used to compute optimal alignments $\mathcal{O}_L: X_L \rightsquigarrow Y_L$ and $\mathcal{O}_R: X_R \rightsquigarrow Y_R$, we are left with the task of combining \mathcal{O}_L and \mathcal{O}_R into an optimal alignment $\mathcal{O}: X \leadsto Y$. Unlike in the original divide-and-conquer algorithm, we cannot simply stitch \mathcal{O}_L and \mathcal{O}_R because we have no guarantee that $ed(X,Y) = ed(X_L,Y_L) + ed(X_R,Y_R)$. Nevertheless, since \mathcal{O}_L , \mathcal{O}_R , and \mathcal{O} intersect \mathcal{A} frequently, the result of stitching needs to be improved only in a neighborhood of the stitching point with self edit distance $\mathcal{O}(k)$. As we show in the following proposition of independent interest, which we use for $(x_m, y_m) = (X_L, Y_L)$, it suffices to find two appropriate points (x_ℓ, y_ℓ) and (x_r, y_r) on \mathcal{A} , compute an optimal alignment \mathcal{O}_M between them, exploiting the fact that $\mathsf{self-ed}(X[x_\ell..x_r)) = \mathcal{O}(k)$, and then combine \mathcal{O}_L , \mathcal{O}_M , and \mathcal{O}_R in a very straightforward way: Follow \mathcal{O}_L until an intersection point with \mathcal{O}_M , then follow \mathcal{O}_M until an intersection point with \mathcal{O}_R , and finally follow \mathcal{O}_R .

Proposition 6.7. Consider a normalized weight function $w: \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$, strings $X, Y \in \Sigma^*$, an alignment $(x_i, y_i)_{i=0}^t =: \mathcal{A}: X \leadsto Y$, and indices $0 \leq \ell \leq m \leq r \leq t$ such that

$$\mathsf{self-ed}(X[x_\ell..x_m)) > 4 \cdot \mathsf{ed}_{\mathcal{A}}^w(X,Y) \quad or \quad \ell = 0, \quad and \quad \mathsf{self-ed}(X[x_m..x_r)) > 4 \cdot \mathsf{ed}_{\mathcal{A}}^w(X,Y) \quad or \quad r = t.$$

$$If \ \mathcal{O}_L : X[0..x_m) \leadsto Y[0..y_m), \ \mathcal{O}_M : X[x_\ell..x_r) \leadsto Y[y_\ell..y_r), \ and \ \mathcal{O}_R : X[x_m..|X|) \leadsto Y[y_m..|Y|)$$

$$are \ w\text{-optimal alignments, then} \ \mathcal{O}_L \cap \mathcal{O}_M \neq \emptyset \neq \mathcal{O}_R \cap \mathcal{O}_M, \ and \ all \ points \ (x_L^*, y_L^*) \in \mathcal{O}_L \cap \mathcal{O}_M \ and$$

$$(x_R^*, y_R^*) \in \mathcal{O}_R \cap \mathcal{O}_M \ satisfy$$

$$\operatorname{ed}^w(X,Y) = \operatorname{ed}^w_{\mathcal{O}_L}(X[0..x_L^*),Y[0..y_L^*)) + \operatorname{ed}^w_{\mathcal{O}_M}(X[x_L^*..x_R^*),Y[y_L^*..y_R^*)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[y_R^*..|Y|)).$$

This novel combinatorial insight guarantees that the recursive call for (X_R, Y_R) is not affected by updates within (X_L, Y_L) and vice versa. Consequently, each update affects only $\mathcal{O}(\log n)$ recursive calls in total (a single path in the recursion tree) and can be implemented in $\widetilde{\mathcal{O}}(k)$ time. Assigning the local threshold k for each recursive call based on the local cost of A and using standard deamortization tools to account for occasional rebuilding of \mathcal{A} and changes of the local thresholds, we achieve the worst-case update time of $\widetilde{\mathcal{O}}(k)$ after $\widetilde{\mathcal{O}}(k^2)$ -time initialization.

Incorporating Small Integer Weights

Our dynamic algorithm for unweighted edit distance utilizes many insights borrowed from a static weighted edit distance algorithm, so a natural question is whether it can be generalized to the weighted setting. Even though our solution combines many steps, there are only three points where we utilized the fact that we deal with unweighted edit distance:

- 1. The toolbox of Tiskin [Tis08, Tis15] for representing distance matrices and computing their (min, +)-products relies on the unit-Monge property that holds in the unweighted case only.
- **2.** In the initialization phase, we used the (inherently unweighted) Landau–Vishkin algorithm [LV88] to compute an approximate alignment $A: X \rightsquigarrow Y$.
- 3. Our strategy relied on the fact that the cost of \mathcal{A} grows by at most one unit per update.

The last issue is the most serious one: in general, our approach does not seem to offer any advantage over the naive strategy (run an $\tilde{\mathcal{O}}(k^3)$ -time static algorithm from scratch upon every update) if a single update can change $\operatorname{ed}_{\mathcal{A}}^w(X,Y)$ from $\mathcal{O}(k)$ to $\omega(k)$. Nevertheless, if we assume that the weight of every edit is bounded by some value W, then $\operatorname{ed}_{\mathcal{A}}^w(X,Y)$ remains $\mathcal{O}(k)$ for at least $\Omega(k/W)$ edits, so we only incur an $\mathcal{O}(W)$ -factor overhead in the update time.

As the lower bounds of [CKW23] indicate, the toolbox of Tiskin [Tis08, Tis15] cannot be generalized to the weighted case without significant overheads. Moreover, since the underlying hard instances use weight functions with values in $\{0\} \cup [1,2]$, the upper bound W is not helpful either. It turns out, however, that the $\tilde{\mathcal{O}}(k)$ bound on the $(\min, +)$ -product of $\mathcal{O}(k) \times \mathcal{O}(k)$ distance matrices for unweighted edit distance can be generalized to $\tilde{\mathcal{O}}(kW)$ in the case of integer weights in [0..W]. A lesser-known work by Russo [Rus12] generalizes Tiskin's unit-Monge $(\min, +)$ -product algorithm and shows that the $(\min, +)$ -product of Monge matrices can be computed in time near-linear in the dimensions and the core sizes of the involved matrices, where the core of a Monge matrix M consists of all pairs (i, j) such that $M_{i,j} + M_{i+1,j+1} < M_{i,j+1} + M_{i+1,j}$. It is not hard to prove that an $\mathcal{O}(k) \times \mathcal{O}(k)$ Monge matrix with integer values between 0 and $\mathcal{O}(kW)$ has core of size $\mathcal{O}(kW)$, which lets us implement all the necessary distance-matrix operations with an $\tilde{\mathcal{O}}(W)$ -factor overhead compared to the unweighted case (this comes on top of the overhead from rebuilding \mathcal{A} more often).

The final missing piece is a static $\mathcal{O}(n+k^2)$ -time algorithm for efficient initialization of our dynamic scheme. In fact, we only need to compute a constant-factor approximation of the w-optimal alignment: the divide-and-conquer strategy that we developed readily improves such an approximately w-optimal alignment into a truly w-optimal one in $\widetilde{\mathcal{O}}(k^2W)$ time. 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 $\mathcal{O}(\log W)$ times, we arrive at the problem of computing an unweighted alignment, which we can solve in $\mathcal{O}(n+k^2)$ using [LV88]. Overall, we obtain both an $\widetilde{\mathcal{O}}(n+k^2W)$ -time static algorithm and a dynamic algorithm with $\widetilde{\mathcal{O}}(kW^2)$ update time.

Faster Static Algorithm for Large Integer Weights

Our $\widetilde{\mathcal{O}}(n+k^2W)$ -time algorithm for integer weights in [0..W] can be seen as a relatively low-effort application of our techniques, but it is a very interesting result on its own: even when the Landau–Vishkin algorithm [LV88] fails beyond repair, its running time can be recovered with little overheads for small integer weights. This has not been known even for very small W such as 2 or 3. The lower bound of [CKW23] highlights that integrality was crucial here (recall that hard instances have weights in $\{0\}\cup[1,2]$), so it is natural to ask what happens for integer weights of potentially unbounded magnitude. The bottleneck here is that the (min, +)-product of $m \times m$ integer Monge matrices takes $\widetilde{\Theta}(m^2)$ time, 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. We develop a novel procedure of independent interest that, given an $m \times m$ Monge matrix M with non-negative integer entries, constructs an $m \times m$ Monge matrix M whose core is of size $\mathcal{O}(m\sqrt{k})$ yet $\min\{M_{i,j}, k+1\} = \min\{M'_{i,j}, k+1\}$ holds for every entry. Applying it to all distance matrices, we get an $\widetilde{\mathcal{O}}(n+k^{2.5})$ -time static algorithm, which is faster than all known alternatives when $k \leq \min\{W^2, \sqrt{n}\}$.

3 Preliminaries

We closely follow the narration of [CKW23] in the preliminaries.

Strings

A string $X = X[0] \cdots X[n-1] \in \Sigma^n$ is a sequence of |X| = n characters over an alphabet Σ ; |X| is the *length* of |X|. For a *position* $i \in [0..n)$, we say that X[i] is the *i*-th character of X. We denote the empty string over Σ by ε . Given indices $0 \le i \le j \le |X|$, we say that $X[i..j) := X[i] \cdots X[j-1]$ is a *fragment* of X. We may also write X[i..j-1], X(i-1..j-1], or X(i-1..j) for the fragment X[i..j).

We say that X occurs as a substring of a string Y, if there are $0 \le i \le j \le |Y|$ such that X = Y[i..j).

Alignments and (Weighted) Edit Distances

We start with the crucial notion of an *alignment*, which gives us a formal way to describe a sequence of edits to transform a string into another.

■ **Definition 3.1** (Alignment, [DGH+23, Definition 2.3]). A sequence $\mathcal{A} = (x_t, y_t)_{t=0}^m$ is an alignment of X[x..x') onto Y[y..y'), denoted by $\mathcal{A} : X[x..x') \rightsquigarrow Y[y..y')$ if $(x_0, y_0) = (x, y)$, $(x_m, y_m) = (x', y')$, and $(x_{t+1}, y_{t+1}) \in \{(x_t + 1, y_t), (x_t, y_t + 1), (x_t + 1, y_t + 1)\}$ for all $t \in [0..m)$.

We write $\mathbf{A}(X[x..x'), Y[y..y'))$ for the set of all alignments of X[x..x') onto Y[y..y').

For an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^m \in \mathbf{A}(X[x..x'), Y[y..y'))$ and an index $t \in [0..m)$, we say that

- A deletes $X[x_t]$ if $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t)$;
- $A inserts Y[y_t] if (x_{t+1}, y_{t+1}) = (x_t, y_t + 1);$
- $A \text{ aligns } X[x_t] \text{ to } Y[y_t], \text{ denoted by } X[x_t] \rightsquigarrow Y[y_t] \text{ if } (x_{t+1}, y_{t+1}) = (x_t + 1, y_t + 1);$
- \mathcal{A} matches $X[x_t]$ with $Y[y_t]$ if $X[x_t] \rightsquigarrow Y[y_t]$ and $X[x_t] = Y[y_t]$;
- A substitutes $X[x_t]$ for $Y[y_t]$ if $X[x_t] \rightsquigarrow Y[y_t]$ but $X[x_t] \neq Y[y_t]$.

Insertions, deletions, and substitutions are jointly called (character) edits.

Given $\mathcal{A} = (x_t, y_t)_{t=0}^m \in \mathbf{A}(X, Y)$, we define the *inverse alignment* as $\mathcal{A}^{-1} := (y_t, x_t)_{t=0}^m \in \mathbf{A}(Y, X)$.

Given an alphabet Σ , we set $\overline{\Sigma} := \Sigma \cup \{\varepsilon\}$. We call w a weight function if $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_{\geq 0}$, and for $a, b \in \overline{\Sigma}$, 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 write $\operatorname{ed}_{\mathcal{A}}^{w}(X[x..x'), Y[y..y'))$ for the *cost* of an alignment $\mathcal{A} \in \mathbf{A}(X[x..x'), Y[y..y'))$ with respect to a weight function w, that is, for the total cost of edits made by \mathcal{A} , where

- the cost of deleting X[x] is $w(X[x], \varepsilon)$,
- the cost of inserting Y[y] is $w(\varepsilon, Y[y])$,
- the cost of aligning X[x] with Y[y] is w(X[x], Y[y]).

We define the weighted edit distance of strings $X, Y \in \Sigma^*$ with respect to a weight function w as $ed^w(X,Y) := \min_{A \in \mathbf{A}(X,Y)} ed^w_A(X,Y)$. For an integer $k \ge 0$, we also define a capped version

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

- **Definition 3.2** (Alignment Graph, [CKW23, Definition 3.2]). For strings $X, Y \in \Sigma^*$ and a weight function $w : \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$, we define the alignment graph $AG^w(X,Y)$ as follows. $AG^w(X,Y)$ has vertices $[0..|X|] \times [0..|Y|]$,
- horizontal edges $(x,y) \to (x+1,y)$ of cost $w(X[x],\varepsilon)$ for $(x,y) \in [0..|X|) \times [0..|Y|]$,
- vertical edges $(x,y) \to (x,y+1)$ of cost $w(\varepsilon,Y[y])$ for $(x,y) \in [0..|X|] \times [0..|Y|)$, and
- diagonal edges $(x,y) \rightarrow (x+1,y+1)$ of cost w(X[x],Y[y]) for $(x,y) \in [0,|X|) \times [0,|Y|)$.

We visualize the alignment graph $AG^w(X,Y)$ as a grid graph with |X|+1 columns and |Y|+1 rows. We think of the vertex (0,0) as the top left vertex of the grid, and a vertex (x,y) in the x-th column and y-th row.

Observe that we can interpret $\mathbf{A}(X[x..x'), Y[y..y'))$ as the set of $(x,y) \rightsquigarrow (x',y')$ paths in $G := \mathrm{AG}^w(X,Y)$. Moreover, $\mathrm{ed}_{\mathcal{A}}^w(X[x..x'), Y[y..y'))$ is the cost of \mathcal{A} interpreted as a path in G, and thus $\mathrm{ed}^w(X[x..x'), Y[y..y')) = \mathrm{dist}_G((x,y), (x',y'))$.

If for every $a, b \in \overline{\Sigma}$ we have that w(a, b) = 1 if $a \neq b$ and w(a, b) = 0 otherwise, then $\operatorname{\sf ed}^w(X, Y)$ corresponds to the standard $\operatorname{\it unweighted}$ edit distance (also known as Levenshtein distance [Lev65]). For this case, we drop the subscript w in $\operatorname{\sf ed}^w$, $\operatorname{\sf ed}^w_{\mathcal{A}}$, $\operatorname{\sf ed}^w_{\leq k}$, and AG^w .

We say that an alignment $A \in \mathbf{A}(X,Y)$ is w-optimal if $\mathrm{ed}_{A}^{w}(X,Y) = \mathrm{ed}^{w}(X,Y)$.

A weight function w is called normalized if $w(a,b) \geq 1$ holds for all $a,b \in \overline{\Sigma}$ with $a \neq b$.

Fact 3.3 (Slight Generalization of [DGH⁺23, Proposition 2.16]). Given strings $X, Y \in \Sigma^*$, an integer $k \geq 1$, and (oracle access to) a normalized weight function $w : \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$, the value $\operatorname{ed}_{\leq k}^w(X,Y)$ can be computed in $\mathcal{O}(\min\{|X|+1,|Y|+1\}\cdot k)$ time. Furthermore, if $\operatorname{ed}^w(X,Y) \leq k$, the algorithm returns a w-optimal alignment of X onto Y.

Proof. Consider any w-optimal alignment \mathcal{A} . If $\operatorname{ed}^w(X,Y) \leq k$, \mathcal{A} may contain at most k vertical and diagonal edges as each one of them costs at least one. Therefore, all vertices (x,y) of \mathcal{A} satisfy $|x-y| \leq k$. These vertices lie on 2k+1 diagonals of $\operatorname{AG}^w(X,Y)$. Each such diagonal consists of at most $\min\{|X|+1,|Y|+1\}$ vertices. Therefore, the subgraph of $\operatorname{AG}^w(X,Y)$ induced by such vertices has size $\mathcal{O}(\min\{|X|+1,|Y|+1\}\cdot k)$ and can be computed in the same time complexity. Hence, finding $\operatorname{ed}^w(X,Y)$ if $\operatorname{ed}^w(X,Y) \leq k$ is equivalent to finding the shortest path from (0,0) to (|X|,|Y|) in such a subgraph. As the graph is acyclic, we can find such a shortest path in $\mathcal{O}(\min\{|X|+1,|Y|+1\}\cdot k)$ time along with the distance. If this distance is at most k, we return it along with the path we found. Otherwise, we return ∞ .

The breakpoint representation of an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^m \in \mathbf{A}(X, Y)$ 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 $2 + \operatorname{ed}_{\mathcal{A}}(X, Y)$ and that it can be used to retrieve the entire alignment: for any two consecutive elements (x', y'), (x, y) of the breakpoint representation, it suffices to add $(x - \delta, y - \delta)$ for $\delta \in (0. \max(x - x', y - y'))$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^m \in \mathbf{A}(X, Y)$, for every $\ell, r \in [0..m]$ with $\ell \leq r$ we say that \mathcal{A} aligns $X[x_\ell..x_r)$ onto $Y[y_\ell..y_r)$ and denote it by $X[x_\ell..x_r) \rightsquigarrow Y[y_\ell..y_r)$. We denote the cost of the induced alignment of $X[x_\ell..x_r)$ onto $Y[y_\ell..y_r)$ by $\operatorname{ed}_{\mathcal{A}}^w(X[x_\ell..x_r), Y[y_\ell..y_r))$.

Given $\mathcal{A}: X \leadsto Y$ and a fragment $X[x_{\ell}..x_r)$ of X, we write $\mathcal{A}(X[x_{\ell}..x_r))$ for the fragment $Y[y_{\ell}..y_r)$ of Y where

$$y_{\ell} \coloneqq \min\{y \mid (x_{\ell}, y) \in \mathcal{A}\}$$
 and $y_r \coloneqq \begin{cases} |Y| & \text{if } x_r = |X|, \\ \min\{y \mid (x_r, y) \in \mathcal{A}\} & \text{otherwise.} \end{cases}$

Intuitively, $\mathcal{A}(X[x_{\ell}..x_r))$ is the fragment that \mathcal{A} aligns $X[x_{\ell}..x_r)$ onto.

- Fact 3.4 (Triangle Inequality, [DGH⁺23, Fact 2.5]). Consider strings $X, Y, Z \in \Sigma^*$ as well as alignments $A: X \leadsto Y$ and $B: Y \leadsto Z$. There exists a composition alignment $B \circ A: X \leadsto Z$ satisfying the following properties for all $x \in [0..|X|)$ and $z \in [0..|Z|)$:
- $\mathcal{B} \circ \mathcal{A}$ aligns X[x] to Z[z] if and only if there exists $y \in [0..|Y|)$ such that \mathcal{A} aligns X[x] to Y[y] and \mathcal{B} aligns Y[y] to Z[z].
- $\mathcal{B} \circ \mathcal{A}$ deletes X[x] if and only if \mathcal{A} deletes X[x] or there exists $y \in [0..|Y|)$ such that \mathcal{A} aligns X[x] to Y[y] and \mathcal{B} deletes Y[y].
- $\mathcal{B} \circ \mathcal{A}$ inserts Z[z] if and only if \mathcal{B} inserts Z[z] or there exists $y \in [0..|Y|)$ such that \mathcal{A} inserts Y[y] and \mathcal{B} aligns Y[y] to Z[z].

If a weight function w satisfies the triangle inequality, that is, $w(a,b) \leq w(a,c) + w(c,b)$ holds for all $a,b,c \in \overline{\Sigma}$, then $\operatorname{ed}_{\mathcal{B} \circ A}^w(X,Z) \leq \operatorname{ed}_{\mathcal{A}}^w(X,Y) + \operatorname{ed}_{\mathcal{B}}^w(Y,Z)$.

Corollary 3.5. There is an algorithm that given strings $X, Y, Z \in \Sigma^*$ as well as the breakpoint representations of alignments $A: X \rightsquigarrow Y$ and $B: Y \rightsquigarrow Z$, in time $\mathcal{O}(\operatorname{ed}_{\mathcal{A}}(X,Y) + \operatorname{ed}_{\mathcal{B}}(Y,Z) + 1)$ builds the breakpoint representation of $B \circ A$.

Proof. Consider some character X[x]. If \mathcal{A} matches X[x] to some Y[y], and \mathcal{B} matches Y[y] to some Z[z], then X[x] = Y[y] = Z[z], and $\mathcal{C} := \mathcal{B} \circ \mathcal{A}$ matches X[x] to Z[z]. Hence, if some vertex (x,z) of AG(X,Z) is a part of the breakpoint representation of \mathcal{C} , then either (x,y) for some y is a part of the breakpoint representation of \mathcal{A} , or \mathcal{A} matches X[x] to some Y[y], and (y,z) is a part of the breakpoint representation of \mathcal{B} . Therefore, by scanning the breakpoint representations of \mathcal{A} and \mathcal{B} using two pointers, we can construct the breakpoint representation of \mathcal{C} in time $\mathcal{O}(\operatorname{ed}_{\mathcal{A}}(X,Y) + \operatorname{ed}_{\mathcal{B}}(Y,Z) + 1)$.

Even though composition alignment does not necessarily satisfy triangle inequality, we can still formulate a fact similar to Fact 3.4 in the general case.

Lemma 3.6. Consider strings $X,Y,Z \in \Sigma^*$, alignments $A: X \leadsto Y$ and $B: Y \leadsto Z$, and a weight function $w: \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$ such that $w(a,b) \leq W$ for all $a,b \in \overline{\Sigma}$ for some value W. The composition alignment $C = \mathcal{B} \circ A$ satisfies $\operatorname{ed}_{\mathcal{C}}^w(X,Z) \leq \operatorname{ed}_{\mathcal{A}}^w(X,Y) + W \cdot \operatorname{ed}_{\mathcal{B}}(Y,Z)$ and $\operatorname{ed}_{\mathcal{C}}^w(X,Z) \leq W \cdot \operatorname{ed}_{\mathcal{A}}(X,Y) + \operatorname{ed}_{\mathcal{B}}^w(Y,Z)$.

Proof. We first claim that triangle inequality of Fact 3.4 works in a more general case. That is, given alignments $\mathcal{A}: X \leadsto Y$ and $\mathcal{B}: Y \leadsto Z$ and three weight functions w_1, w_2 , and w_3 such that $w_1(a,b) \leq w_2(a,c) + w_3(c,b)$ holds for all $a,b,c \in \overline{\Sigma}$, we have $\operatorname{ed}_{\mathcal{B} \circ \mathcal{A}}^{w_1}(X,Z) \leq \operatorname{ed}_{\mathcal{A}}^{w_2}(X,Y) + \operatorname{ed}_{\mathcal{B}}^{w_3}(Y,Z)$. It follows from the properties of \circ described in Fact 3.4. Any edit $a \mapsto b$ of $\mathcal{B} \circ \mathcal{A}$ for $a \in X \cup \{\varepsilon\}$ and $b \in Z \cup \{\varepsilon\}$ can be decomposed into an edit $a \mapsto c$ of \mathcal{A} and an edit $c \mapsto b$ of \mathcal{B} for some $c \in Y \cup \{\varepsilon\}$. By the triangle inequality, we have $w_1(a,b) \leq w_2(a,c) + w_3(c,b)$. Furthermore, different edits of $\mathcal{B} \circ \mathcal{A}$ correspond to different edits of \mathcal{A} and \mathcal{B} . Therefore, by summing up these inequalities over all edits, we obtain $\operatorname{ed}_{\mathcal{B} \circ \mathcal{A}}^{w_1}(X,Z) \leq \operatorname{ed}_{\mathcal{A}}^{w_2}(X,Y) + \operatorname{ed}_{\mathcal{B}}^{w_3}(Y,Z)$.

We now use this fact to prove the lemma. We define $w_1 := w$, $w_2 := w$, and $w_3(a, a) = 0$ for any $a \in \overline{\Sigma}$ and $w_3(a, b) = W$ for any $a \neq b \in \overline{\Sigma}$. We claim that $w_1(a, b) \leq w_2(a, c) + w_3(c, b)$ holds for all $a, b, c \in \overline{\Sigma}$. If $c \neq b$, the claim holds as $w_1(a, b) = w(a, b) \leq W = w_3(c, b) \leq w_2(a, c) + w_3(c, b)$. On the other hand, if c = b, the claim holds as $w_1(a, b) = w(a, b) = w_2(a, b) = w_2(a, c) \leq w_2(a, c) + w_3(c, b)$.

By the extension of Fact 3.4 for three different weight functions, we obtain $\operatorname{\sf ed}_{\mathcal{B} \circ \mathcal{A}}^w(X,Z) = \operatorname{\sf ed}_{\mathcal{B} \circ \mathcal{A}}^{w_1}(X,Z) \leq \operatorname{\sf ed}_{\mathcal{A}}^{w_2}(X,Y) + \operatorname{\sf ed}_{\mathcal{B}}^{w_3}(Y,Z) = \operatorname{\sf ed}_{\mathcal{A}}^w(X,Y) + W \cdot \operatorname{\sf ed}_{\mathcal{B}}(Y,Z)$, thus proving the first inequality from the lemma statement.

The second inequality can be obtained analogously by swapping w_2 and w_3 .

Corollary 3.7. Consider strings $X, Y, X', Y' \in \Sigma^*$, alignments $A : X \leadsto Y$, $B : X \leadsto X'$, and $C : Y \leadsto Y'$, and a weight function $w : \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$ such that $w(a,b) \leq W$ for all $a,b \in \overline{\Sigma}$ for some value W. The composition alignment $D = C \circ (A \circ B^{-1})$ satisfies $\operatorname{ed}_{\mathcal{D}}^w(X',Y') \leq \operatorname{ed}_{\mathcal{A}}^w(X,Y) + W \cdot (\operatorname{ed}_{\mathcal{B}}(X,X') + \operatorname{ed}_{\mathcal{C}}(Y,Y'))$.

Proof. We first consider the alignment $\mathcal{D}' := \mathcal{A} \circ \mathcal{B}^{-1}$. Lemma 3.6 implies that $\operatorname{ed}_{\mathcal{D}'}^w(X',Y) \leq W \cdot \operatorname{ed}_{\mathcal{B}^{-1}}(X',X) + \operatorname{ed}_{\mathcal{A}}^w(X,Y) = W \cdot \operatorname{ed}_{\mathcal{B}}(X,X') + \operatorname{ed}_{\mathcal{A}}^w(X,Y)$. As $\mathcal{D} = \mathcal{C} \circ \mathcal{D}'$, Lemma 3.6 implies that $\operatorname{ed}_{\mathcal{D}}^w(X',Y') \leq \operatorname{ed}_{\mathcal{D}'}^w(X',Y) + W \cdot \operatorname{ed}_{\mathcal{C}}(Y,Y') \leq W \cdot \operatorname{ed}_{\mathcal{B}}(X,X') + \operatorname{ed}_{\mathcal{A}}^w(X,Y) + W \cdot \operatorname{ed}_{\mathcal{C}}(Y,Y')$, thus proving the claim.

Fact 3.8 ([CKW23, Lemma 3.7]). Let $X, Y \in \Sigma^*$ denote strings and let $A \in \mathbf{A}(X, Y)$ denote an alignment. For every $(x, y) \in A$, we have

$$\operatorname{ed}^w(X,Y) \leq \operatorname{ed}^w(X[0..x),Y[0..y)) + \operatorname{ed}^w(X[x..|X|),Y[y..|Y|)) \leq \operatorname{ed}^w_{\mathcal{A}}(X,Y).$$

Balanced Straight Line Programs

For a context-free grammar \mathcal{G} , we denote by $\mathcal{N}_{\mathcal{G}}$ and $\Sigma_{\mathcal{G}}$ the set of non-terminals and the set of terminals, respectively. The set of symbols is $\mathcal{S}_{\mathcal{G}} := \Sigma_{\mathcal{G}} \cup \mathcal{N}_{\mathcal{G}}$. A straight-line program (SLP) is a context-free grammar \mathcal{G} such that:

- each non-terminal $A \in \mathcal{N}_{\mathcal{G}}$ has a unique production $A \to \text{rhs}_{\mathcal{G}}(A)$, where $\text{rhs}_{\mathcal{G}}(A) = BC$ consists of two symbols $B, C \in \mathcal{S}_{\mathcal{G}}$,
- the set $\mathcal{S}_{\mathcal{G}}$ admits a partial order \prec such that $B \prec A$ and $C \prec A$ if $\mathrm{rhs}_{\mathcal{G}}(A) = BC$.

The expansion function $\exp_{\mathcal{G}}: \mathcal{S}_G \to \Sigma_{\mathcal{G}}^*$ is defined as follows:

$$\exp_{\mathcal{G}}(A) = \begin{cases} A & \text{if } A \in \Sigma_{\mathcal{G}}, \\ \exp_{\mathcal{G}}(B) \exp_{\mathcal{G}}(C) & \text{if } A \in \mathcal{N}_{\mathcal{G}} \text{ with } \text{rhs}_{\mathcal{G}}(A) = BC. \end{cases}$$

Moreover, $\exp_{\mathcal{G}}$ is lifted to $\exp_{\mathcal{G}}: \mathcal{S}_{\mathcal{G}}^* \to \Sigma_{\mathcal{G}}^*$ with $\exp_{\mathcal{G}}(A_1 \cdots A_a) = \exp_{\mathcal{G}}(A_1) \cdots \exp_{\mathcal{G}}(A_a)$ for $A_1 \cdots A_a \in \mathcal{S}_{\mathcal{G}}^*$. When the grammar \mathcal{G} is clear from context, we omit the subscript.

The parse tree $\mathcal{T}(A)$ of a symbol $A \in \mathcal{S}_{\mathcal{G}}$ is a rooted ordered tree with each node ν associated to a symbol symb $(\nu) \in \mathcal{S}_{\mathcal{G}}$. The root of $\mathcal{T}(A)$ is a node ρ with symb $(\rho) = A$. If $A \in \Sigma_{\mathcal{G}}$, then ρ has no children. If $A \in \mathcal{N}_{\mathcal{G}}$ and rhs(A) = BC, then ρ has two children, and the subtrees rooted in these children are (copies of) $\mathcal{T}(B)$ and $\mathcal{T}(C)$, respectively. The root of $\mathcal{T}(B)$ is called the left child of the root of $\mathcal{T}(A)$, and the root of $\mathcal{T}(C)$ is called the right child of the root of $\mathcal{T}(A)$. Furthermore, we say that B is the left child of A, and C is the right child of A.

The weight of a symbol $A \in \mathcal{S}_{\mathcal{G}}$ is defined as $\operatorname{len}(A) := |\exp(A)|$; we assume that all algorithms store this weight along with the symbol A. We say that an SLP \mathcal{G} is weight-balanced if every nonterminal $A \in \mathcal{N}_{\mathcal{G}}$ with $\operatorname{rhs}(A) = BC$ satisfies $\frac{1}{3} \leq \operatorname{len}(B)/\operatorname{len}(C) \leq 3$.

Theorem 3.9 (see [CLL+05, Section VII.E] and [Gaw11, Lemma 8]). A weight-balanced SLP \mathcal{G} can be extended (while remaining weight-balanced) using the following operations: Merge(B, C): Given symbols $B, C \in \mathcal{S}_{\mathcal{G}}$, in $\mathcal{O}(1 + |\log(\ln(B)/\ln(C))|)$ time returns a symbol A

such that $\exp(A) = \exp(B) \cdot \exp(C)$.

¹¹Weight-balanced SLPs from [CLL⁺05] allow for $\alpha/(1-\alpha) \le \text{len}(B)/\text{len}(C) \le (1-\alpha)/\alpha$ for any balancedness parameter $\alpha \le 1 - \sqrt{2}/2 \approx 0.29$. We use $\alpha = 0.25$.

Power(A, ℓ): Given a symbol $A \in \mathcal{S}_{\mathcal{G}}$ and an integer $\ell \geq 1$, in $\mathcal{O}(1 + \log \ker(A) + \log \ell)$ time returns a symbol B such that $\exp(B) = \exp(A)^{\infty}[0, \ell]$.

The algorithms implementing these operations represent each symbol along with its weight $len(\cdot)$ and, for non-terminals, the production $rhs(\cdot)$, and make no other assumptions about the implementation of $\mathcal G$. In particular, every symbol present in the parse trees of the output symbols is either newly added to $\mathcal G$ or present in the parse trees of the input symbols. The number of newly added symbols is bounded by the time complexity of each operation.

Proof. The implementations of Merge(B, C) and $Substring(A, \ell, r)$ are provided in $[CLL^+05]$, Section VII.E]. The original analysis focuses on the number of symbols added to \mathcal{G} , but several subsequent works, including [Gaw11], noted that the running time can be bounded in the same way. Our implementation of $Power(A, \ell)$ is based on the proof of [Gaw11], Lemma 8]. Since the original description is stated in the context of a particular application (rather than in an abstract form), we repeat the simple algorithm and its analysis below.

Denote $t := \max\{0, \lceil \log_2 \frac{\ell}{\operatorname{len}(A)} \rceil\}$. We create a sequence of symbols $B_0, B_1, B_2, \ldots, B_t$ such that $B_0 = A$ and $\operatorname{rhs}(B_i) = B_{i-1}B_{i-1}$ for $i \in [1..t]$. If $\operatorname{len}(B_t) = \ell$, we return B_t . Otherwise, we retrieve $B = \operatorname{Substring}(B_t, 0, \ell)$ and return B as the answer.

The children of B_i are perfectly balanced, so \mathcal{G} remains a weight-balanced SLP. At the same time $\exp(B_i) = \exp(B_{i-1})^2 = \exp(A)^{2^i}$. The correctness follows from the facts that $\exp(B_t)$ is a prefix of $\exp(A)^{\infty}$ and $\operatorname{len}(B_t) \geq \ell$. The time complexity is $\mathcal{O}(t+1)$ for the creation of B_0, \ldots, B_t and $\mathcal{O}(\log \operatorname{len}(B_t))$ for the final Substring operation. Since $\operatorname{len}(B_t) \leq 2 \max\{\operatorname{len}(A), \ell\}$ and $t \leq \log_2 \operatorname{len}(B_t)$, this is $\mathcal{O}(1 + \log \operatorname{len}(A) + \log \ell)$.

The PILLAR Model

Charalampopoulos, Kociumaka, and Wellnitz [CKM20] introduced the PILLAR model. The PILLAR model provides an abstract interface to a set of primitive operations on strings which can be efficiently implemented in different settings. Thus, an algorithm developed using the PILLAR interface does not only yield algorithms in the standard setting, but also directly yields algorithms in diverse other settings, for instance, fully compressed, dynamic, etc.

In the PILLAR model we are given a family \mathcal{X} of strings to preprocess. The elementary objects are fragments $X[\ell..r)$ of strings $X \in \mathcal{X}$. Initially, the model gives access to each $X \in \mathcal{X}$ interpreted as X[0..|X|). Other fragments can be retrieved via an Extract operation:

■ Extract (S, ℓ, r) : Given a fragment S and positions $0 \le \ell \le r \le |S|$, extract the fragment $S[\ell, r]$, which is defined as $X[\ell' + \ell, \ell' + r]$ if $S = X[\ell', r]$ for $X \in \mathcal{X}$.

Moreover, the PILLAR model provides the following primitive operations for fragments S and T [CKM20]:

- = LCP(S,T): Compute the length of the longest common prefix of S and T.
- **LCP** $^R(S,T)$: Compute the length of the longest common suffix of S and T.
- Access(S, i): Assuming $i \in [0, ... |S|)$, retrieve the character S[i].
- Length(S): Retrieve the length |S| of the string S.

Observe that in the original definition [CKM20], the PILLAR model also includes an IPM operation to find all (internal) exact occurrences of one fragment in another. We do not need the IPM operation in this work.¹²

We use the following version of the classic Landau-Vishkin algorithm [LV88] in the PILLAR model.

■ Fact 3.10 ([CKW20, Lemma 6.1]). There is a PILLAR algorithm that, given strings $X, Y \in \Sigma^*$, in time $\mathcal{O}(k^2)$ computes $k = \operatorname{ed}(X, Y)$ along with the breakpoint representation of an optimal unweighted alignment of X onto Y.

Monge Matrix Toolbox

In this paper, we extensively use distance matrices in an alignment graph of two strings. As alignment graphs are planar, such matrices are Monge.

- **Definition 3.11.** A matrix A of size $p \times q$ is called Monge if for $i \in [1..p)$ and $j \in [1..q)$, we have $A_{i,j} + A_{i+1,j+1} \leq A_{i,j+1} + A_{i+1,j}$.
- **Fact 3.12** ([FR06, Section 2.3]). Consider a directed planar graph G with non-negative edge weights. For vertices $u_1, \ldots, u_p, v_q, \ldots, v_1$ lying (in this cyclic order) on the outer face of G, define a $p \times q$ matrix D with $D_{i,j} = \operatorname{dist}_G(u_i, v_j)$. If all entries of D are not equal to ∞ , D is Monge. ■

When talking about distance matrices in a graph, combining paths corresponds to the min-plus product of such matrices.

Definition 3.13. Given a matrix A of size $p \times q$ and a matrix B of size $q \times r$, define their min-plus product $A \otimes B$ as a matrix C of size $p \times r$, where $C_{i,k} = \min_j A_{i,j} + B_{j,k}$ for $i \in [1..p], k \in [1..r]$.

Furthermore, if matrices A and B are Monge, then their min-plus product $A \otimes B$ is also Monge.

Fact 3.14 ([Tis15, Theorem 2]). Let A, B, and C be matrices, such that $A \otimes B = C$. If A and B are Monge, then C is also Monge.

One of the most celebrated results concerning Monge matrices is the SMAWK algorithm of [AKM⁺87].

Fact 3.15 ([AKM⁺87]). There is an algorithm that, given random access to the entries of a $p \times q$ Monge matrix, computes its row and column minima and their positions in time $\mathcal{O}(p+q)$.

When talking about Monge matrices, notions of matrix core and density matrix are of fundamental importance.

■ **Definition 3.16.** For a matrix A of size $p \times q$, define its density matrix A^{\square} of size $(p-1) \times (q-1)$, where $A_{i,j}^{\square} = A_{i,j+1} + A_{i+1,j} - A_{i,j} - A_{i+1,j+1}$ for $i \in [1..p), j \in [1..q)$.

We define the core of a matrix A as $\operatorname{core}(A) := \{(i, j, A_{i,j}^{\square}) \mid i \in [1..p), j \in [1..q), A_{i,j}^{\square} \neq 0\}$. We denote the size of the core $\delta(A) := |\operatorname{core}(A)|$.

Note that for a Monge matrix A, all entries of its density matrix are non-negative, and thus core(A) consists of triples (i, j, v) with some positive values v.

For any matrix A of size $p \times q$ we write A[a..b)[c..d) for any $a, b \in [1..p]$ with a < b and $c, d \in [1..q]$ with c < d to denote a contiguous submatrix of A consisting of all entries on the intersection of rows [a..b) and columns [c..d) of A.

¹²We still use the name PILLAR, and not PLLAR, though.

Efficient Algorithms for $(\min, +)$ -Products of Monge Matrices

As discussed in Section 3, we will require tools to work with Monge matrices and their min-plus products. SMAWK algorithm ([AKM⁺87]) allows min-plus multiplication of Monge matrices in near-linear time. However, it is still not fast enough for our applications. The Monge matrices we will be working with are "sparse" in the sense that they have small cores. We would like to develop algorithms that operate on Monge matrices, and their time complexities depend on the core sizes, not on matrix sizes. Such tools were developed by Russo in [Rus12]. We present slight generalizations of the main results from [Rus12]. In this section, whenever we speak about matrices, we fix some positive integer N and assume that the dimensions of all the matrices are bounded by N.

- **Fact 4.1** (Generalization of [Rus12, Lemma 12]). Let A and B denote Monge matrices of sizes $p \times q$ and $q \times r$, respectively. There is an algorithm that, given T(N)-time random access to the entries of A and B, in time $\mathcal{O}(N \cdot T(N) \log N + \delta(A) \log^3 N + \delta(A) T(N) \log^2 N)$ creates a data structure $\mathcal{MT}_{A\otimes B}$ that provides $\mathcal{O}(\log N\cdot T(N))$ -time random access to the entries of $A\otimes B$.
- Fact 4.2 (Straightforward corollary of [Rus12, Lemma 13]). Let A denote a Monge matrix of size $p \times q$. There is an algorithm that, given T(N)-time random access to the entries of A, in time $\mathcal{O}((N + \delta(A) \log N) \cdot T(N))$ computes $\operatorname{core}(A)$.

However, note that if we apply Fact 4.1 repeatedly, the access time to the resulting matrices will increase by a factor of $\log N$ each time. To address this issue and enrich the interface, we design a new data structure that extends the abilities of \mathcal{MT} .

Lemma 4.3. There exists an algorithm that, given T(N)-time random access to the entries of the first row and the first column of a Monge matrix A of size $p \times q$ as well as all entries of core(A)explicitly, in time $\mathcal{O}(N \cdot T(N) + \delta(A) \log N)$ builds a core-based Monge matrix oracle data structure $\mathsf{CMO}(A)$ that provides the following interface.

Core range query: given $a, b \in [1..p]$ such that a < b and $c, d \in [1..q]$ such that c < d, in time $\mathcal{O}(\log N)$ returns $\mathrm{sum}(A^{\square}[a..b)[c..d)) \coloneqq \sum_{i \in [a..b)} \sum_{j \in [c..d)} A_{i,j}^{\square}$. **Random access:** given $i \in [1..p]$ and $j \in [1..q]$, in time $\mathcal{O}(\log N)$ returns $A_{i,j}$.

Full core listing: in time $\mathcal{O}(1 + \delta(A))$ returns $\operatorname{core}(A)$.

Furthermore, we develop the following tools to operate with CMO:

- **Given CMO**(A) and CMO(B) for Monge matrices A and B, compute CMO(A \otimes B) in time $\widetilde{\mathcal{O}}(N+\delta(A)+\delta(A\otimes B)).$
- = Given CMO(A) for a Monge matrix A, compute CMO(A^T) in time $\widetilde{\mathcal{O}}(N + \delta(A))$.
- \blacksquare Given $\mathsf{CMO}(A)$ for a Monge matrix A, compute $\mathsf{CMO}(C)$ for any contiguous submatrix C of A in time $\mathcal{O}(N + \delta(A))$.
- Given CMO(A) and CMO(B) for Monge matrices A and B, compute CMO(A+B) in time $\mathcal{O}(N + \delta(A) + \delta(B)).$
- Given $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B)$ for Monge matrices A and B, compute $\mathsf{CMO}((A \mid B))^{13}$ (if $(A \mid B)$ is Monge) in time $\tilde{\mathcal{O}}(N + \delta(A) + \delta(B))$.

In the regime of small integer edit weights, we will prove that distance matrices we work with have small cores. However, if we do not assume edit weights to be small, matrix cores may be proportional to the sizes of the matrices, thus neglecting the advantage our algorithms have over SMAWK.

¹³ For matrices A of size $p \times q$ and B of size $p \times r$, we denote by $(A \mid B)$ the matrix C of size $p \times (q+r)$ such that $C_{i,j} = A_{i,j}$ for $j \leq q$ and $C_{i,j} = B_{i,j-q}$ for j > q. We call this operation "stitching" of matrices A and B.

To address this issue, we define the notion of k-equivalence.

■ **Definition 4.4.** Let a and b be two non-negative integers. For a positive k, we call a and b k-equivalent and write $a \stackrel{k}{=} b$ if $\min\{a, k+1\} = \min\{b, k+1\}$. Furthermore, let A and B be matrices of sizes $p \times q$ with non-negative entries. We call these two matrices k-equivalent and write $A \stackrel{k}{=} B$ if $A_{i,j} \stackrel{k}{=} B_{i,j}$ for all $i \in [1..p]$ and $j \in [1..q]$.

As we calculate $\operatorname{\sf ed}_{\leq k}^w$, if some values in distance matrices are larger than k, their exact values do not serve any purpose to us. Hence, instead of considering the exact distance matrix A, it will suffice to maintain its "capped" version: any matrix A' that is k-equivalent to A. As it turns out, if we allow such a relaxation, it becomes possible to find such a matrix A' with small core.

We observe that k-equivalence is an equivalence relation, and it is preserved under some basic operations with matrices.

Observation 4.5. The following properties of k-equivalence hold for any valid matrices A, B, C, D.

```
■ A \stackrel{k}{=} A.

■ If A \stackrel{k}{=} B, then B \stackrel{k}{=} A.

■ If A \stackrel{k}{=} B and B \stackrel{k}{=} C, then A \stackrel{k}{=} C.

■ If A \stackrel{k}{=} B, then A^T \stackrel{k}{=} B^T.

■ If A \stackrel{k}{=} B, then A[a..b][c..d) \stackrel{k}{=} B[a..b][c..d) for any a < b and c < d.

■ If A \stackrel{k}{=} B and C \stackrel{k}{=} D, then (A \mid C) \stackrel{k}{=} (B \mid D).

■ If A \stackrel{k}{=} B and C \stackrel{k}{=} D, then A + C \stackrel{k}{=} B + D. 14

■ If A \stackrel{k}{=} B and C \stackrel{k}{=} D, then A \otimes C \stackrel{k}{=} B \otimes D.
```

We extend the list of tools for CMO with "capped" matrix multiplication. That is, we develop an algorithm that given a positive integer k and $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B)$ for Monge matrices A and B, in time $\widetilde{\mathcal{O}}(\delta(A) + N\sqrt{k})$ computes $\mathsf{CMO}(C')$ for some Monge matrix C' that is k-equivalent to $A \otimes B$, such that $\delta(C') = \mathcal{O}(N\sqrt{k})$.

4.1 Simple Implementations

We first describe how to implement CMO. For that, we require an efficient static weighted orthogonal range counting data structure.

■ Fact 4.6 (Weighted Orthogonal Range Counting [Wil85, Theorem 3]). There exists an algorithm that, given a collection of n points $(x_i, y_i)_{i \in [1...n]}$ with integer weights $(w_i)_{i \in [1...n]}$, in time $\mathcal{O}(n \log n)$ builds a data structure that supports the following queries in $\mathcal{O}(\log n)$ time: given an orthogonal range $R = [x, x'] \times [y, y']$, compute $\sum_{i:(x_i, y_i) \in R} w_i$.

Proof of Lemma 4.3. We implement CMO(A) for some Monge matrix A of size $p \times q$ in the following way. We store the entries $A_{1,j}$ for $j \in [1..q]$ and $A_{i,1}$ for $i \in [1..p]$ explicitly. Furthermore, we explicitly store all elements of core(A). Finally, we store the weighted orthogonal range counting

¹⁴ Note that in contrast to the other properties of k-equivalence, this one and the next one rely on the fact that the entries of A and B are required to be non-negative.

data structure of Fact 4.6 over the elements of core(A) that in logarithmic time provides query access to the sum of the values of core elements of A (and thus, values of A^{\square}) on a rectangle.

To build such a data structure, in time $\mathcal{O}(N \cdot T(N))$ we query and store all entries of A of form $A_{1,j}$ for $j \in [1..q]$ and $A_{i,1}$ for $i \in [1..p]$. We also explicitly store all elements of the core of A. Furthermore, we build the weighted orthogonal range counting data structure over the elements of core(A) in time $\mathcal{O}(\delta(A)\log N)$ using Fact 4.6. Hence, the algorithm for building CMO works in time $\mathcal{O}(N \cdot T(N) + \delta(A) \log N)$.

Core range queries. Note that elements of core(A) are exactly all non-zero entries of A^{\square} . Hence, range queries to the entries of core(A) using the weighted orthogonal range counting data structure of Fact 4.6 provides $\mathcal{O}(\log N)$ -time queries for the sum of the elements of A^{\square} in a range.

Random access. Say, $A_{i,j}$ is queried. Note that $A_{i,j} = A_{1,j} + A_{i,1} - A_{1,1} - \sup(A^{\square}[1..i)[1..j))$ by the definition of A^{\square} . We store $A_{1,j}, A_{i,1}$, and $A_{1,1}$ explicitly, and sum $(A^{\square}[1..i)[1..j))$ can be computed in $\mathcal{O}(\log N)$ time using a single core range query.

Full core listing. To answer the full core listing query, we just return core(A) that we store explicitly.

We now show the promised functionality of CMO. First, we show how to perform some basic operations with matrices using CMO.

- **Lemma 4.7.** The following facts hold.
- There is an algorithm that given CMO(A) for some Monge matrix A, computes $CMO(A^T)$ in time $\mathcal{O}(N \log N + \delta(A) \log N)$.
- There is an algorithm that given CMO(A) for some Monge matrix A, computes CMO(C) in time $\mathcal{O}(N \log N + \delta(A) \log N)$ for any contiguous submatrix C of A.
- There is an algorithm that given CMO(A) and CMO(B) for some Monge matrices A and B of sizes $p \times q$, computes $\mathsf{CMO}(A+B)$ in time $\mathcal{O}(N\log N + \delta(A)\log N + \delta(B)\log N)$.
- There is an algorithm that given CMO(A) and CMO(B) for Monge matrices A of size $p \times q$ and B of size $p \times r$ such that $(A \mid B)$ is Monge, computes $\mathsf{CMO}((A \mid B))$ in time $\mathcal{O}(N \log N +$ $\delta(A) \log N + \delta(B) \log N$.

Proof. Note that $\mathsf{CMO}(A)$ provides $\mathcal{O}(\log N)$ -time random access to the entries of A^T and any contiguous submatrix C of A. Furthermore, given core(A), one can calculate $core(A^T)$ and core(C)in time $\mathcal{O}(\delta(A))$. Moreover, A^T and C are Monge. Thus, we can build $\mathsf{CMO}(A^T)$ and $\mathsf{CMO}(C)$ in time $\mathcal{O}(N \log N + \delta(A) \log N)$ using Lemma 4.3, thus proving the first two claims.

Note that $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B)$ provide $\mathcal{O}(\log N)$ -time random access to the entries of A+B. Furthermore, note that $(A+B)^{\square} = A^{\square} + B^{\square}$. Thus, $\delta(A+B) < \delta(A) + \delta(B)$, and $\operatorname{core}(A+B)$ can be computed in time $\mathcal{O}((\delta(A) + \delta(B)) \log N)$ by merging $\operatorname{core}(A)$ and $\operatorname{core}(B)$. Moreover, A + B is Monge. Thus, we can build CMO(A+B) in time $O(N \log N + (\delta(A) + \delta(B)) \log N)$ using Lemma 4.3, thus proving the third claim.

Finally, $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B)$ provide $\mathcal{O}(\log N)$ -time random access to the entries of $(A \mid B)$. Furthermore, $core((A \mid B))$ consists of the core entries of A, the core entries of B (shifted), and potentially some core elements on the boundary between A and B that can be found in time $\mathcal{O}(N \log N)$ by accessing the entries of the last column of A and the first column of B through CMO. Thus,

 $\delta((A \mid B)) \leq \delta(A) + \delta(B) + N$, and in time $\mathcal{O}(\delta(A) + \delta(B) + N \log N)$ we can compute $\operatorname{core}((A \mid B))$. Therefore, if $(A \mid B)$ is Monge, we can build $\operatorname{CMO}((A \mid B))$ in time $\mathcal{O}((N + \delta(A) + \delta(B)) \log N)$ using Lemma 4.3, thus proving the fourth claim.

Next, we show how to obtain $CMO(A \otimes B)$ from CMO(A) and CMO(B).

Lemma 4.8. There is an algorithm that given $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B)$ for Monge matrices A of size $p \times q$ and B of size $q \times r$, builds $\mathsf{CMO}(A \otimes B)$ in time $\mathcal{O}(N \log^2 N + \delta(A) \log^3 N + \delta(A \otimes B) \log^3 N)$.

Proof. CMO(A) and CMO(B) provide $\mathcal{O}(\log N)$ -time random access to the entries of A and B. Applying Lemma 4.1, we get $\mathcal{O}(\log^2 N)$ -time random access to the entries of $C := A \otimes B$ in time $\mathcal{O}(N\log^2 N + \delta(A)\log^3 N)$. Applying Lemma 4.2, we calculate $\operatorname{core}(C)$ in time $\mathcal{O}(N\log^2 N + \delta(C)\log^3 N)$. Applying Lemma 4.3, we get CMO(C) in time $\mathcal{O}(N\log^2 N + \delta(C)\log N)$. The overall time complexity is $\mathcal{O}(N\log^2 N + \delta(A)\log^3 N + \delta(C)\log^3 N)$.

4.2 Capped Multiplication

We now describe the more complicated procedure of "capped" min-plus multiplication. Note that given two capped matrices A and B, Fact 4.1 provides random access to the entries of $C := A \otimes B$ in time that is independent of $\delta(C)$. However, if $\delta(C)$ is large, we cannot afford to apply Fact 4.2 to directly access $\operatorname{core}(C)$. Thus, we would like to compute only such elements of the core of C that are somehow "necessary" under k-equivalence. To achieve it, we first prove the following combinatorial claim.

Lemma 4.9. Consider some positive integers p,q, and s, and a set $P \subseteq [1..p] \times [1..q]$, such that if $(i,j) \in P$, then there are at most s elements $(k,\ell) \in P$ (excluding (i,j)), such that $k \ge i$ and $\ell \le j$. Then, $|P| = \mathcal{O}(\sqrt{pqs})$.

Proof. We call all elements of $[1..p] \times [1..q]$ cells. Say that a cell $(i,j) \in [1..p] \times [1..q]$ dominates another cell $(k,\ell) \in [1..p] \times [1..q]$ if $k \geq i$ and $\ell \leq j$. Denote by D the set of all cells that are either in P or dominated by some cell from P (in particular, if some cell (i,j) is in D, then all cells dominated by (i,j) are also in D). For each row r of $[1..p] \times [1..q]$, denote by x_r the maximum number, such that (r,x_r) is in D ($x_r = 0$ if no cells in the r-th row are in D). Note that $(x_r)_{r \in [1..p]}$ is a non-decreasing sequence. Denote $R_r := [r..p] \times [1..x_r]$ for each $r \in [1..p]$. Since (r,x_r) is in D, this cell is either in P or is dominated by some cell (i,j) from P. Thus, such a cell (i,j) from P dominates all cells in R_r , so there are at most s+1 cells from P in each R_r . Hence, $\sum_r |R_r \cap P| \leq p \cdot (s+1)$.

Denote $\#_c := |\{i \in [1..p] \mid (i,c) \in P\}|$ for every $c \in [1..q]$. Say that if we order the elements of $\{(i,j) \in P \mid j=c\}$ in the increasing order of i, we get a sequence $(r_1,c), (r_2,c), \ldots, (r_{\#_c},c)$. As $(r_1,c) \in P$, we get that $x_{r_1} \ge c$. As $(x_r)_r$ is a non-decreasing sequence, $x_j \ge c$ for every $j \in [r_1..p]$ follows. Thus, for any $i \in [1..\#_c]$, the cell (r_i,c) lies in at least $r_i - r_1 + 1 \ge i$ rectangles R_j for $j \in [r_1..r_i]$. Consequently, all cells of P from the c-th column lie in at least $1 + 2 + \cdots + \#_c = {\#_c+1 \choose 2}$ rectangles R_j in total (if two cells lie in the same rectangle, it is counted twice). Note that $\sum_r |R_r \cap P|$ is equal to the sum over all elements of P of the number of rectangles they lie in. Thus, we get that

$$\sum_{c \in [1..q]} {\#_c + 1 \choose 2} \le \sum_r |R_r \cap P| \le p \cdot (s+1).$$

As $\binom{\#_c+1}{2} \ge \#_c^2/2$, we get that $\sum_{c \in [1..q]} \#_c^2 \le 2p \cdot (s+1)$. By the inequality between the arithmetic mean and the quadratic mean, we get $\sum_{c \in [1..q]} \#_c^2 \ge (\sum_{c \in [1..q]} \#_c)^2/q$. Thus, $|P| = \sum_{c \in [1..q]} \#_c \le \sqrt{q \cdot 2p \cdot (s+1)} = \mathcal{O}(\sqrt{pqs})$.

We now use this combinatorial claim to show that if we have access to the entries of some Monge matrix C, we can compute $\mathsf{CMO}(C')$ for some $C' \stackrel{k}{=} C$, where C' has a small core, by filtering the core elements of C. We first prove this fact for two restricted classes of matrices.

■ **Lemma 4.10.** There is an algorithm that, given some positive integer k and T(N)-time random access to the entries of a Monge matrix $C \in \mathbb{Z}_{\geq 0}^{p \times q}$, where all entries in the last row and the first column of C are zeros, in time $\mathcal{O}(N\sqrt{k} \cdot T(N)\log N)$ builds $\mathsf{CMO}(C')$ for some Monge matrix $C' \in \mathbb{Z}_{\geq 0}^{p \times q}$ such that $\delta(C') = \mathcal{O}(N\sqrt{k})$ and $C' \stackrel{k}{=} C$.

Proof. Matrix C is Monge, thus for any $i \in [1..p]$ and $j \in [1..q]$ we have $C_{i,j} = C_{i,1} + C_{p,j} - C_{p,1} + \text{sum}(C^{\square}[i..p)[1..j)) = 0 + 0 - 0 + \text{sum}(C^{\square}[i..p)[1..j)) = \text{sum}(C^{\square}[i..p)[1..j))$ by the definition of C^{\square} .

Define C' as a Monge matrix, in which values in the last row and the first column are zeros, and $\operatorname{core}(C')$ is a subset of $\operatorname{core}(C)$, such that an element $(i,j,v) \in \operatorname{core}(C)$ is in $\operatorname{core}(C')$ if and only if $\operatorname{sum}(C^{\square}[i..p)[1..j]) - v \leq k$. Hence, $C'_{i,j} = \operatorname{sum}(C'^{\square}[i..p)[1..j))$ for all $i \in [1..p]$ and $j \in [1..q]$, where C'^{\square} is defined via $\operatorname{core}(C')$.

We prove that $C' \stackrel{k}{=} C$. If $C'_{i,j} \neq C_{i,j}$, there is some core entry in $[i..p) \times [1..j)$ that we deleted. Consider the leftmost (and the bottommost out of leftmost ones) such deleted core entry of C in $[i..p) \times [1..j)$. We delete a core entry only if the sum of other core entries of C to the left and down of it is $\geq k+1$. As it is the leftmost and bottommost deleted core entry, all these core entries of C to the left and down are also in $\operatorname{core}(C')$, and they all lie inside $[i..p) \times [1..j)$. Hence, both $C_{i,j}$ and $C'_{i,j}$ are at least k+1. Consequently, $\min\{C'_{i,j}, k+1\} = k+1 = \min\{C_{i,j}, k+1\}$, thus proving $C' \stackrel{k}{=} C$.

Furthermore, any core entry of C' has at most k other core entries of C' to the left and down of it as otherwise it would be deleted. Hence, the set $\{(i,j) \mid (i,j,v) \in \operatorname{core}(C')\}$ satisfies the conditions of Lemma 4.9 for s=k, and thus $\delta(C')=\mathcal{O}(N\sqrt{k})$.

We now obtain the core entries of C' column-by-column from left to right by saving the appropriate core values of C. Fix the j-th column. We design a recursive procedure that generates all the core entries of C' within the given subcolumn of the j-th column. It terminates if either C has no core entries within the subcolumn or the sum of all core elements to the left and down of the subcolumn is larger than k. These conditions can be checked in time $\mathcal{O}(T(N))$ as $C^{\square}[a..b][c..d) = C_{a,d} + C_{b,c} - C_{a,c} - C_{b,d}$ for any $1 \le a < b \le p$ and $1 \le c < d \le q$, thus by querying four elements of C, we may get the sum of core elements of C in any rectangle. Otherwise, the procedure recurses into two halves of the given subcolumn or, if the subcolumn is just a single entry, generates a core element of C' corresponding to this one entry.

Note that if we terminate early, there are either no core elements of C at all in the current subcolumn, or for each one of them, the sum of other core entries of C to the left and down of it is larger than k, and thus such a core entry does not belong to core(C'). Hence, such a procedure indeed computes core(C').

Furthermore, we claim that if given some subcolumn, we make further recursive calls, then this subcolumn contains either at least one core element of C' or the bottommost core element of C in this column that is not in $\operatorname{core}(C')$. If all entries of $\operatorname{core}(C)$ from the j-th column are in $\operatorname{core}(C')$, the claim is clear: we recurse only if there are some entries of $\operatorname{core}(C)$ in the current subcolumn, and thus they are entries of $\operatorname{core}(C')$. Otherwise, denote by ℓ the number, such that $(\ell, j, v) \in \operatorname{core}(C)$ for some v is the lowest core entry of C in the j-th column that is not a core entry of C'. If a subcolumn does not contain (ℓ, j) and does not contain any core entries of C' in the j-th column, it is either completely above (ℓ, j) , and then we terminate because the sum of the core entries to the left and

bottom of it is larger than k; or it is completely below (ℓ, j) , and then does not contain any core entries of C, and thus we also terminate.

Each element of $\operatorname{core}(C)$ that we consider lies in a logarithmic number of potential recursive calls, thus we compute $\operatorname{core}(C')$ in time $\mathcal{O}((\delta(C')+q)\cdot T(N)\log N)$.

Knowing $\operatorname{core}(C')$, we can calculate all entries in the first row of C' in $\mathcal{O}(\delta(C')\log N)$ time. We now may apply Lemma 4.3 to build $\operatorname{CMO}(C')$ in time $\mathcal{O}(N+\delta(C')\log N)$. The total time complexity of the algorithm is $\mathcal{O}((\delta(C')+q)\cdot T(N)\log N+\delta(C')\log N)=\mathcal{O}(N\cdot T(N)\log N+N\sqrt{k}\cdot T(N)\log N)=\mathcal{O}(N\sqrt{k}\cdot T(N)\log N)$.

■ Lemma 4.11. There is an algorithm that, given some positive integer k and T(N)-time random access to the entries of a Monge matrix $C \in \mathbb{Z}_{\geq 0}^{p \times q}$ such that row and column minima in C are equal to zero, in time $\mathcal{O}(N\sqrt{k} \cdot T(N)\log N)$ builds $\mathsf{CMO}(C')$ for some Monge matrix $C' \in \mathbb{Z}_{\geq 0}^{p \times q}$ such that $\delta(C') = \mathcal{O}(N\sqrt{k})$ and $C' \stackrel{k}{=} C$.

Proof. Define z_i for $i \in [1..p]$ as the leftmost zero in the i-th row of C. Define a matrix $F \in \mathbb{Z}_{\geq 0}^{p \times q}$, where $F_{i,j} = C_{i,j}$ if $j > z_i$, and $F_{i,j} = 0$ otherwise. Define $G \in \mathbb{Z}_{\geq 0}^{p \times q}$, where $G_{i,j} = C_{i,j}$ if $j < z_i$, and $G_{i,j} = 0$ otherwise. Note that for any $i \in [1..p]$ and $j \in [1..q]$, $\min\{F_{i,j}, G_{i,j}\} = 0$ and $F_{i,j} + G_{i,j} = \max\{F_{i,j}, G_{i,j}\} = C_{i,j}$.

 \Box Claim 4.12. All entries in the first column and the last row of F are zeros, and all entries in the first row and the last column of G are zeros. Furthermore, matrices F and G are Monge.

Proof. We prove the lemma for matrix F. The proof for matrix G is analogous.

By the definition of matrix F, we have $F_{i,j} = C_{i,j}$ if $j > z_i$, and $F_{i,j} = 0$ otherwise. As $z_i \ge 1$ for any i, for $F_{i,1}$, the second defining equality applies. Thus, all entries in the first column of F are zeros.

Define z_i' for $i \in [1..p]$ as the rightmost zero in the i-th row of C. We claim that all entries between z_i -th and z_i' -th in the i-th row of C are zeros. Assume for the sake of contradiction that there is some $i \in [1..p]$ and $j \in (z_i..z_i')$ such that $C_{i,j} > 0$. Matrix C is Monge, thus the topmost column minima positions are non-decreasing. $C_{i,z_i'} = 0$, so it is a column minimum for the z_i' -th column. Thus, the topmost minimum position in this column is $\leq i$, and thus the topmost minimum position (i',j) in the j-th column satisfies $i' \leq i$ (and $C_{i',j} = 0$ as all column minima are zeros). As $C_{i,j} \neq 0$, we have i' < i. Therefore, we arrive at a contradiction as $C_{i',j} + C_{i,z_i} = 0 < C_{i,j} \leq C_{i,j} + C_{i',z_i}$ violates the fact that C is Monge. Hence, indeed all entries between z_i -th and z_i' -th in the i-th row of C are zeros.

Furthermore, we claim that $z_{i+1} \leq z_i' + 1$. Assume for the sake of contradiction that there is some $i \in [1..p)$, such that $z_{i+1} > z_i' + 1$. We have $C_{i+1,z_{i+1}} = 0$, thus the topmost column minimum in the z_{i+1} -st column is in a row with index at most i+1. The topmost minima positions are non-decreasing, thus topmost minimum in the $z_i' + 1$ -st column appears in row $i' \leq i+1$ ($C_{i',z_{i+1}} = 0$ as all column minima are zeros). As both i-th and i+1-st rows do not contain zero in the $z_i' + 1$ -st column, we have $i' \leq i-1$. Therefore, we arrive at a contradiction as $C_{i',z_{i+1}} + C_{i,z_i} = 0 < C_{i,z_{i+1}} \leq C_{i,z_{i+1}} + C_{i',z_i}$ violates the fact that C is Monge. Hence, indeed $z_{i+1} \leq z_i' + 1$ for all i.

Column minimum in the last column is zero, thus there is some i, for which $z_i' = q$. The sequence z' is non-decreasing as C is Monge. Therefore, $z_p' = q$ follows. By the definition of matrix F, $F_{i,j} = C_{i,j}$ if $j > z_i$, and $F_{i,j} = 0$ otherwise. As all entries in the i-th row between z_i -th and z_i' -th are zeros, we may write an alternative definition: $F_{i,j} = C_{i,j}$ if $j > z_i'$, and $F_{i,j} = 0$ otherwise. We have $z_p' = q$, so for any $F_{p,j}$ the second defining equality applies. Thus, all entries in the last row of F are zeros.

It remains to show that F is Monge. It is sufficient to check Monge property for all contiguous 2×2 submatrices of F, so $F_{i,j} + F_{i+1,j+1} \leq F_{i,j+1} + F_{i+1,j}$ for all $i \in [1..p)$ and $j \in [1..q)$. If all the summands in the inequality are zeros or all are equal to the entries from C, the Monge property clearly holds. Thus, it remains to check it for submatrices, in which some but not all non-zero values from C are preserved in F. That is, there is at least one non-zero entry in this submatrix in C that is preserved and at least one that is not preserved. These two entries can not be neighbors in the same row as between preserved and not preserved positive values in a row there should be at least one zero element. Analogously, they can not be neighbors in the same column. So these two elements lie on the same diagonal. Then the remaining two elements should be zeros in C, as otherwise they would be either preserved or not preserved, but, in any case, two adjacent elements, where one is preserved and the other is not, would exist, which is impossible, as we have already discussed. If entries $C_{i,j}$ and $C_{i+1,j+1}$ are zeros, then in any case Monge property is still valid in F. If entries $C_{i,j+1}$ and $C_{i+1,j}$ are zeros, then Monge property would be violated in C. Thus, all contiguous 2×2 submatrices of F satisfy Monge property, and so F is Monge.

In time $\mathcal{O}(N \cdot T(N))$ we can find values z_i for all $i \in [1..p]$ using Fact 3.15. Random access in time T(N) to the entries of C and values z_i provide $\mathcal{O}(T(N))$ -time random access to the entries of F and G. We now solve the problem for matrices F (using Lemma 4.10) and G (using Lemma 4.10 for G^T and Lemma 4.7 after that) to obtain $\mathsf{CMO}(F')$ and $\mathsf{CMO}(G')$ for some appropriate $F' \stackrel{k}{=} F$ and $G' \stackrel{k}{=} G$ in time $\mathcal{O}(N\sqrt{k} \cdot T(N)\log N)$. As C = F + G, we get that $C' := F' + G' \stackrel{k}{=} C$ due to Observation 4.5. And thus, $\mathsf{CMO}(C')$ can be obtained in time $\mathcal{O}((N + \delta(F') + \delta(G'))\log N) = \mathcal{O}(N\log N + N\sqrt{k}\log N)$ time by Lemma 4.7.

The overall time complexity is $\mathcal{O}(N \cdot T(N) + N\sqrt{k} \cdot T(N) \log N + N \log N + N\sqrt{k} \log N) = \mathcal{O}(N\sqrt{k} \cdot T(N) \log N)$. Furthermore, $\delta(C') \leq \delta(F') + \delta(G') = \mathcal{O}(N\sqrt{k})$.

We now prove the same claim in the general case.

Lemma 4.13. There is an algorithm that, given some positive integer k and T(N)-time random access to the entries of a Monge matrix $C \in \mathbb{Z}_{\geq 0}^{p \times q}$, in time $\mathcal{O}(N\sqrt{k} \cdot T(N) \log N)$ builds $\mathsf{CMO}(C')$ for some Monge matrix $C' \in \mathbb{Z}_{\geq 0}^{p \times q}$ such that $\delta(C') = \mathcal{O}(N\sqrt{k})$ and $C' \stackrel{k}{=} C$.

Proof. We apply Fact 3.15 to compute row-minima r_i of C in time $\mathcal{O}(N \cdot T(N))$. Define a Monge matrix D, such that $D_{i,j} := C_{i,j} - r_i$ for any $i \in [1..p]$ and $j \in [1..q]$. By storing r_i and having T(N)-time query access to the entries of C, we get $\mathcal{O}(T(N))$ -time random access to the entries of D. We now use this oracle access and Fact 3.15 to compute column-minima c_j of D in time $\mathcal{O}(N \cdot T(N))$. Define a Monge matrix E, such that $E_{i,j} := D_{i,j} - c_j = C_{i,j} - r_i - c_j$. Analogously, we have $\mathcal{O}(T(N))$ -time random access to the entries of E. We use Lemma 4.11 to build $\mathsf{CMO}(E')$ for some Monge matrix $E' \in \mathbb{Z}_{\geq 0}^{p \times q}$, such that $\delta(E') = \mathcal{O}(N\sqrt{k})$ and $E' \stackrel{k}{=} E$ in time $\mathcal{O}(N\sqrt{k} \cdot T(N) \log N)$. Define matrix C', such that $C'_{i,j} := E'_{i,j} + r_i + c_j$. Note that $\mathsf{core}(C') = \mathsf{core}(E')$, and thus $\delta(C') = \mathcal{O}(N\sqrt{k})$. Furthermore, we can obtain the entries of the first row and the first column of C' in $\mathcal{O}(N\log N)$ time by accessing $\mathsf{CMO}(E')$. Hence, using Lemma 4.3 we can build $\mathsf{CMO}(C')$ in time $\mathcal{O}(N+\delta(C')\log N) = \mathcal{O}(N\sqrt{k}\log N)$. Finally, as $E' \stackrel{k}{=} E$, we have $C' \stackrel{k}{=} C$ due to Observation 4.5 (C = E + M) and C' = E' + M for matrix M with $M_{i,j} = r_i + c_j$. The overall time complexity is $\mathcal{O}(N\sqrt{k} \cdot T(N)\log N)$.

Finally, we apply Lemma 4.13 to compute the capped product of matrices.

■ Lemma 4.14. There is an algorithm that, given CMO(A) and CMO(B) for Monge matrices $A \in \mathbb{Z}_{\geq 0}^{p \times q}$ and $B \in \mathbb{Z}_{\geq 0}^{q \times r}$ and some positive integer k, in time $\mathcal{O}(\delta(A)\log^3 N + N\sqrt{k}\log^3 N)$ builds CMO(C'), where $C' \in \mathbb{Z}_{\geq 0}^{p \times r}$ is some Monge matrix such that $\delta(C') = \mathcal{O}(N\sqrt{k})$ and $C' \stackrel{k}{=} A \otimes B$.

Proof. Applying Lemma 4.1, we get $\mathcal{O}(\log^2 N)$ -time random access to the entries of $C := A \otimes B$ in time $\mathcal{O}(N \log^2 N + \delta(A) \log^3 N)$. Applying Lemma 4.13, we get $\mathsf{CMO}(C')$ in time $\mathcal{O}(N \sqrt{k} \log^3 N)$. The total time complexity is $\mathcal{O}(N \log^2 N + \delta(A) \log^3 N + N \sqrt{k} \log^3 N) = \mathcal{O}(\delta(A) \log^3 N + N \sqrt{k} \log^3 N)$.

4.3 Monge Stitching

Note that all operations in Lemma 4.7 except for "stitching" are applicable for any input Monge matrices. That is, given two Monge matrices A and B, matrix $(A \mid B)$ is not necessarily Monge. In practice, if we can ensure that $(A \mid B)$ is Monge, Lemma 4.7 is applicable. However, if we are given some $A' \stackrel{k}{=} A$ and $B' \stackrel{k}{=} B$, it is not guaranteed that $(A' \mid B')$ is Monge even if $(A \mid B)$ is. To address this issue, we develop the following lemma. Note that it does not cover all such situations, but only a very specific one that we will encounter.

Lemma 4.15. Let $C ∈ \mathbb{Z}_{\geq 0}^{(n_1+n_2)\times m}$ be a Monge matrix, and let k be a positive integer. Let $N = n_1 + n_2 + m$. Let $A := C[1..n_1][1..m]$ and $B := C(n_1..n_1 + n_2][1..m]$. Let $\bar{\jmath} ∈ [1..m]$ be such that $C_{i,j} ≤ C_{i-1,j}$ for all $(i,j) ∈ (1..n_1+1]\times[\bar{\jmath}..m]$ and $C_{i,j} ≤ C_{i,j+1}$ for all $(i,j) ∈ [1..n_1+1]\times[\bar{\jmath}..m)$. Let B' be a Monge matrix that is k-equivalent to B such that $B'_{1,j} = B_{1,j}$ for all $j ≤ \bar{\jmath}$. There is an algorithm that given k, CMO(A), CMO(B'), and $\bar{\jmath}$, in time $O((N + \delta(A) + \delta(B'))\log N)$ builds CMO(C') for some Monge matrix $C' ∈ \mathbb{Z}_{\geq 0}^{(n_1+n_2)\times m}$ such that $C' \stackrel{k}{=} C$, $\delta(C') = O(N + \delta(A) + \delta(B'))$, and $C'_{i,1} = C_{i,1}$ for $i ∈ [1..n_1]$.

Proof. Consider $\bar{C} = (A^T \mid B'^T)^T$. Note that $\bar{C} \stackrel{k}{=} C$. Furthermore, all entries of \bar{C}^{\square} in all rows except for n_1 -st correspond to some entries of A^{\square} and B'^{\square} , and thus are non-negative. Furthermore, we can find all entries in the n_1 -st row of \bar{C}^{\square} in time $\mathcal{O}(m \log N)$ by accessing all elements in the last row of A and the first row of B' through $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B')$. If all these entries are non-negative, we define $C' := \bar{C}$ and use Lemma 4.7 to build $\mathsf{CMO}(\bar{C})$ in time $\mathcal{O}((N + \delta(A) + \delta(B')) \log N)$ as $\bar{C} = (A^T \mid B'^T)^T$. It is clear that such a matrix C' satisfies all the required conditions.

Otherwise, let j^* be the smallest index in [1..m) such that $\bar{C}_{n_1,j^*}^{\square} < 0$. Note that $j^* \geq \bar{\jmath}$ as $B_{1,j} = B'_{1,j}$ for all $j \leq \bar{\jmath}$. As we have access to $\operatorname{core}(A)$ and $\operatorname{core}(B')$ via $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B')$, we can build $\operatorname{core}(\bar{C})$ in time $\mathcal{O}(\delta(A) + \delta(B') + N)$.

Furthermore, in time $\mathcal{O}(\delta(A) + \delta(B') + N)$ we can build

$$S \coloneqq \{(i, j, v) \in \operatorname{core}(\bar{C}) \mid i \neq n_1 \text{ or } j < j^*\} \cup \{(n_1, j^*, k + 1 + \bar{C}_{n_1 + n_2, 1})\}.$$

We claim that for any $(i,j,v) \in S$, we have $v \geq 0$. Note that it holds for $(n_1,j^*,k+1+\bar{C}_{n_1+n_2,1})$. It remains to show it for all entries of $\operatorname{core}(\bar{C})$ that are in S. As all entries of \bar{C}^{\square} in all rows except for n_1 -st correspond to some entries of A^{\square} and B'^{\square} , they are non-negative as A and B' are Monge. Furthermore, by the definition of j^* , all entries $(n_1,j,v)\in\operatorname{core}(\bar{C})$ with $j< j^*$ satisfy $v\geq 0$ as j^* is the smallest index, for which $\bar{C}_{n_1,j^*}^{\square}$ is negative. Thus, indeed all elements $(i,j,v)\in S$ satisfy $v\geq 0$.

We now consider a Monge matrix $C' \in \mathbb{Z}^{(n_1+n_2)\times m}$ such that $C'_{i,1} = \bar{C}_{i,1}$ for $i \in [1..n_1 + n_2]$, $C'_{n_1+n_2,j} = \bar{C}_{n_1+n_2,j}$ for $j \in [1..m]$, and $\operatorname{core}(C') = S$. That is, we have

$$C'_{i,j} = \bar{C}_{i,1} + \bar{C}_{n_1 + n_2, j} - \bar{C}_{n_1 + n_2, 1} + \sum_{(a,b,v) \in S, \text{ s.t. } a \ge i, b < j} v$$

for any $i \in [1..n_1 + n_2]$ and $j \in [1..m]$. Such a matrix is Monge as $v \ge 0$ for all $(i, j, v) \in S$. Note that we can calculate $C'_{n_1+n_2,j} = \bar{C}_{n_1+n_2,j} = B'_{n_2,j}$ for all $j \in [1..m]$ in time $\mathcal{O}(m \log N)$ using $\mathsf{CMO}(B')$. Thus, using the formula from the definition of C', we can calculate $C'_{1,j}$ for all $j \in [1..m]$ in time $\mathcal{O}((m+|S|)\log N) = \mathcal{O}((\delta(A)+\delta(B')+N)\log N)$. Furthermore, we have $\operatorname{core}(C') = S$ explicitly (in particular, $\delta(C') = \mathcal{O}(N + \delta(A) + \delta(B'))$. Therefore, we can use Lemma 4.3 to build CMO(C') in time $\mathcal{O}(N + (\delta(A) + \delta(B') + N) \log N)$. The whole algorithm takes time $\mathcal{O}((\delta(A) + \delta(B') + N) \log N)$.

It remains to show that $C'_{i,1} = C_{i,1}$ for $i \in [1..n_1]$, all entries of C' are non-negative, and $C' \stackrel{k}{=} C$. The first claim obviously follows as $C'_{i,1} = \bar{C}_{i,1} = A_{i,1} = C_{i,1}$ for all $i \in [1..n_1]$. As the first column and the last row of C' coincide with the first column and the last row of \bar{C} , and as core entries of C' and \bar{C} outside of $[1..n_1] \times [j^*..m]$ coincide, we have that $C'_{i,j} = \bar{C}_{i,j} \geq 0$ for all $(i,j) \notin [1..n_1] \times [j^* + 1..m]$. Furthermore, we have $C'_{i,j} = \bar{C}_{i,j} \stackrel{k}{=} C_{i,j}$.

Therefore, it remains to prove the claim for $C'_{i,j}$ for $(i,j) \in [1..n_1] \times [j^* + 1..m]$. Note that in this case $n_1 \ge i$, $j^* < j$, and $(n_1, j^*, k + 1 + \bar{C}_{n_1 + n_2, 1}) \in S$. Therefore, by the definition of C', we have

$$\begin{split} C'_{i,j} &= \bar{C}_{i,1} + \bar{C}_{n_1 + n_2, j} - \bar{C}_{n_1 + n_2, 1} + \sum_{(a,b,v) \in S, \text{ s.t. } a \geq i, b < j} v \\ &\geq \bar{C}_{i,1} + \bar{C}_{n_1 + n_2, j} - \bar{C}_{n_1 + n_2, 1} + (k + 1 + \bar{C}_{n_1 + n_2, 1}) \\ &= \bar{C}_{i,1} + \bar{C}_{n_1 + n_2, j} + k + 1 \\ &\geq k + 1. \end{split}$$

Furthermore, we claim that $C_{i,j} \geq k+1$ holds for all $(i,j) \in [1..n_1] \times [j^*+1..m]$. Note that $\bar{C}_{n_1,j^*}^{\square} < 0$ by the definition of j^* . Hence, $\bar{C}_{n_1,j^*}^{\square} \neq C_{n_1,j^*}^{\square}$ as C is Monge. Therefore, one of the entries of \bar{C} in $[n_1...n_1+1]\times[j^*...j^*+1]$ is not equal to the corresponding entry of C. Let this entry be (i',j'). As $\bar{C} \stackrel{k}{=} C$, it follows that $\bar{C}_{i',j'}, C_{i',j'} \geq k+1$. By the definition of $\bar{\jmath}$, we have $C_{i,j} \leq C_{i-1,j}$ for all $(i,j) \in (1..n_1+1] \times [\bar{\jmath}..m]$ and $C_{i,j} \leq C_{i,j+1}$ for all $(i,j) \in [1..n_1+1] \times [\bar{\jmath}..m)$. We have $i' \leq n_1 + 1$ and $j' \geq j^* \geq \bar{\jmath}$. Therefore, we obtain $C_{i,j} \geq C_{i',j'} \geq k + 1$ for all $(i,j) \in [1..i'] \times [j'..m]$. In particular, it holds for all $(i,j) \in [1..n_1] \times [j^* + 1..m]$ as $n_1 \leq i'$ and $j^* + 1 \geq j'$.

As
$$C'_{i,j} \geq k+1$$
 and $C_{i,j} \geq k+1$, we obtain $C'_{i,j} \stackrel{k}{=} C_{i,j}$, thus proving the claim.

4

Hierarchical Alignment Graph Decompositions

In our algorithms, we use SLPs to create hierarchical decompositions of strings X and Y. We calculate the edit distance of X and Y in a divide-and-conquer manner. That is, we use edit distances of smaller substrings to calculate edit distances of larger ones.

Composable Alignment Graph Representation 5.1

In this section we develop a tool that allows given distance information for some X_L and Y and some X_R and Y, to compute distance information for $X := X_L \cdot X_R$ and Y. This tool provides oracle access to the distances between the vertices of the outer face of the alignment graph $AG^w(X,Y)$. Note that some pairs of such vertices of $AG^w(X,Y)$ are not reachable from each other. To circumvent this issue, we define a strongly connected extension of $AG^w(X,Y)$ that we call an augmented alignment graph.

■ Definition 5.1 (Augmented alignment graph). For any two strings $X, Y \in \Sigma^*$ and a weight function $w : \overline{\Sigma}^2 \to [0..W]$, define a directed graph $\overline{AG}^w(X,Y)$ obtained from the alignment graph $AG^w(X,Y)$ by adding, for every edge of $AG^w(X,Y)$, a back edge of weight W+1.

We also use $\overline{\mathrm{AG}}^w(A,B)$ for symbols A and B of some SLP $\mathcal G$ to denote $\overline{\mathrm{AG}}^w(\exp(A),\exp(B))$.

Note that, in contrast to $AG^w(X,Y)$, any vertex of $\overline{AG}^w(X,Y)$ is reachable from any other vertex of $\overline{AG}^w(X,Y)$. We further show that $\overline{AG}^w(X,Y)$ preserves distances from $AG^w(X,Y)$ between reachable pairs of vertices, and for all other relevant pairs of vertices, shortest paths between them behave predictably.

■ Lemma 5.2. Consider strings $X, Y \in \Sigma^*$ and a weight function $w : \overline{\Sigma}^2 \to [0..W]$. Any two vertices (x,y) and (x',y') of the graph $\overline{\operatorname{AG}}^w(X,Y)$ satisfy the following properties:

Monotonicity. Every shortest path $(x,y) = (x_0,y_0) \to (x_1,y_1) \to \cdots \to (x_m,y_m) = (x',y')$ from (x,y) to (x',y') in $\overline{\operatorname{AG}}^w(X,Y)$ is (non-strictly) monotone in both coordinates $(x_i)_{i\in[0..m]}$ and $(y_i)_{i\in[0..m]}$.

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

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

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

Proof. Monotonicity. Consider any shortest path $(x,y) = (x_0,y_0) \to (x_1,y_1) \to \cdots \to (x_m,y_m) = (x',y')$ from (x,y) to (x',y') in $\overline{\mathrm{AG}}^w(X,Y)$. If its first and second coordinates are monotone, we are done. Otherwise, without loss of generality assume that $(x_i)_{i\in[0..m]}$ is not monotone. Every two adjacent elements in this sequence differ by at most one, thus if such a sequence is not monotone, there exist two indices i and j such that i < j - 1, $x_i = x_j$, and $x_k \neq x_i$ for all $k \in (i..j)$. Here j is the first index that breaks monotonicity, and i is the last index before j with $x_i = x_j$. We claim that by replacing the part of the path between (x_i,y_i) and (x_j,y_j) with a straight segment between these two points, we strictly decrease the weight of the path, thus the initial path could not be a shortest one. We consider two cases.

- 1. $y_i \geq y_j$. In this case the straight segment $(x_i, y_i) \to (x_i, y_i 1) \to \cdots \to (x_i, y_j + 1) \to (x_i, y_j) = (x_j, y_j)$ uses only back edges and has weight $(W+1) \cdot (y_i y_j)$. On the other hand, any path from (x_i, y_i) to (x_j, y_j) should decrease the second coordinate of the current vertex at least $y_i y_j$ times, and decreasing a coordinate is possible only using a back edge, thus any such path has at least $y_i y_j$ back edges. Consequently, its weight is at least $(W+1) \cdot (y_i y_j)$, hence, the straight-line path is not longer. Furthermore, the old path should contain more than $y_i y_j$ edges, because the only path between (x_i, y_i) and (x_i, y_j) of that length is a straight segment. All edges except for forward diagonal ones have strictly positive weights, and thus if there are any more of them, the weight of the old path is strictly larger than $(W+1) \cdot (y_i y_j)$, thus proving the claim. On the other hand, if there is a forward diagonal edge, it increases the second coordinate of the current vertex by one, thus there should be at least $y_i y_j + 1$ back edges of total weight at least $(W+1) \cdot (y_i y_j + 1)$, thus proving the claim. Hence, indeed the old path is strictly more expensive than the straight segment path.
- 2. $y_i < y_j$. The path $(x_i, y_i) \to (x_{i+1}, y_{i+1}) \to \cdots \to (x_j, y_j)$ goes from a vertex with the second coordinate equal to y_i to a vertex with the second coordinate equal to y_j . Thus, for every $\bar{y} \in [y_i..y_j)$, there exists some $k_{\bar{y}} \in [i..j)$ such that $y_{k_{\bar{y}}} = \bar{y}$ and $y_{k_{\bar{y}}+1} = \bar{y} + 1$. Furthermore, let

c be the number of back edges on the path $(x_i, y_i) \to (x_{i+1}, y_{i+1}) \to \cdots \to (x_j, y_j)$. The cost of this path is at least $(W+1) \cdot c + \sum_{\bar{y} \in [y_i \dots y_j)} w((x_{k_{\bar{y}}}, y_{k_{\bar{y}}}), (x_{k_{\bar{y}}+1}, y_{k_{\bar{y}}+1}))$ because all edges $(x_{k_{\bar{y}}}, y_{k_{\bar{y}}}) \to (x_{k_{\bar{y}}+1}, y_{k_{\bar{y}}+1})$ are distinct and forward. Furthermore, edges of form $(x_{k_{\bar{y}}}, y_{k_{\bar{y}}}) \to (x_{k_{\bar{y}}+1}, y_{k_{\bar{y}}+1})$ are either diagonal if $x_{k_{\bar{y}}+1} = x_{k_{\bar{y}}} + 1$ or vertical if $x_{k_{\bar{y}}+1} = x_{k_{\bar{y}}}$. For a vertical edge, its weight is equal to the weight of the edge $(x_i, y_{k_{\bar{y}}}) \to (x_i, y_{k_{\bar{y}}} + 1)$ from the straight segment path. For a diagonal edge, it increases the first coordinate of the current vertex on the path by one, and as $x_i = x_j$, the number of such increases is bounded by the number of back edges c on this path because only they decrease the first coordinate of the current vertex on the path. Thus, there are $\leq c$ edges $(x_i, y_{k_{\bar{y}}}) \to (x_i, y_{k_{\bar{y}}} + 1)$ of the straight-line path that correspond to diagonal edges in the old path. Each such edge has weight $\leq W$. From this, it follows that the straight-line path is not longer than the old path.

Furthermore, $c \ge 1$ because the only path between (x_i, y_i) and (x_i, y_j) without back edges is a straight line segment. Thus, the sum of all diagonal edges and back edges in the old path is at least $c \cdot (W+1)$, while there are at most c corresponding edges in the new path, each having length $\le W$, hence having total length $\le c \cdot W < c \cdot (W+1)$. Thus, indeed the old path is strictly more expensive than the straight segment path.

As no path that is not monotone in both coordinates can be a shortest one, we prove the first claim of the lemma.

Distance preservation. Distance preservation follows from monotonicity as if $x \le x'$ and $y \le y'$, both coordinates on any shortest path are non-decreasing, and thus there are no back edges on this path. Hence, the same path exists in $AG^w(X,Y)$. Furthermore, clearly any path from $AG^w(X,Y)$ exists in $\overline{AG}^w(X,Y)$, thus $\operatorname{dist}_{AG^w(X,Y)}((x,y),(x',y')) = \operatorname{dist}_{\overline{AG}^w(X,Y)}((x,y),(x',y'))$.

Path irrelevance. Without loss of generality assume that $x \leq x'$ and $y \geq y'$. Consider some path $(x,y) = (x_0,y_0) \to (x_1,y_1) \to \cdots \to (x_m,y_m) = (x',y')$ from (x,y) to (x',y') in $\overline{\mathrm{AG}}^w(X,Y)$ that is monotone in both coordinates. Such a path has $(x_i)_{i \in [0..m]}$ non-decreasing and $(y_i)_{i \in [0..m]}$ non-increasing. All edges that are monotone in this direction are horizontal forward edges and vertical back edges. Every such path consists of y - y' vertical back edges of weight W + 1 each and a single horizontal forward edge of form $(\bar{x}, \bar{y}) \to (\bar{x} + 1, \bar{y})$ for each $\bar{x} \in [x..x')$ for some \bar{y} . All horizontal edges of this form have equal weights for a fixed \bar{x} , thus the weight of the whole path is independent of the specific path we choose.

We will talk of breakpoint representations of paths in $\overline{\mathrm{AG}}^w(X,Y)$ that are monotone in both coordinates. If such a path starts at (x,y) and ends at (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. Note that it is a consistent definition as all edges in such a path have nonzero weights.

We now show that a fact similar to the one used to restrict the search graph in Fact 3.3 also holds for augmented alignment graphs.

Lemma 5.3. Let $X,Y \in \Sigma^*$ be two strings, k be an integer, and $w: \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$ be a normalized weight function. Let (x,y) and (x',y') be two vertices of $\overline{\operatorname{AG}}^w(X,Y)$. Let P be a path from (x,y) to (x',y') in $\overline{\operatorname{AG}}^w(X,Y)$ of weight at most k. We have that all nodes $(x^*,y^*) \in P$ satisfy $|(x-y)-(x^*-y^*)| \leq k$ and $|(x'-y')-(x^*-y^*)| \leq k$.

Proof. Consider the prefix of the path P from (x,y) to (x^*,y^*) . It has weight at most k and thus contains at most k non-zero edges. Note that each edge of weight zero is a diagonal edge. Hence, while going from (x,y) to (x^*,y^*) along P, the difference $\hat{x}-\hat{y}$ between the coordinates of the current vertex (\hat{x},\hat{y}) change at most k times. Furthermore, one edge changes this difference by at most one. Therefore, we obtain $|(x-y)-(x^*-y^*)| \leq k$. Analogously, we get $|(x'-y')-(x^*-y^*)| \leq k$.

We now precisely define what the distance oracle between the vertices of the outer face of $\overline{\mathrm{AG}}^w(X,Y)$ is.

■ Definition 5.4. For a rectangle $[a..b] \times [c..d]$, we define its perimeter as $\mathcal{P}([a..b] \times [c..d]) \coloneqq (b-a) + (d-c) + 1$.

Furthermore, for two strings X and Y, we define the perimeter of their product as $\mathcal{P}(X \times Y) := \mathcal{P}([0..|X|] \times [0..|Y|]) = |X| + |Y| + 1.$

■ **Definition 5.5.** Let $X,Y \in \Sigma^*$ be two strings and $w: \overline{\Sigma}^2 \to [0..W]$ be a weight function. Furthermore, let $R = [a..b] \times [c..d]$ for $0 \le a \le b \le |X|$ and $0 \le c \le d \le |Y|$ be some rectangle in $\overline{AG}^w(X,Y)$.

Let the sequence of input vertices in(R) of R be a sequence $(p_i)_i$ of vertices of $\overline{AG}^w(X,Y)$ of length $\mathcal{P}(R)$, where $p = ((a,d),(a,d-1),\ldots,(a,c),(a+1,c),\ldots,(b,c))$. Let the left input vertices of R be the prefix of vertices of in(R) with the first coordinate equal to a, and let the top input vertices of R be the suffix of vertices of in(R) with the second coordinate equal to c. Note that the left and the top input vertices of R intersect by one vertex (a,c).

Analogously, let the sequence of output vertices out(R) of R be a sequence $(q_i)_i$ of vertices of $\overline{\mathrm{AG}}^w(X,Y)$ of length $\mathcal{P}(R)$, where $q=((a,d),(a+1,d),\ldots,(b,d),(b,d-1),\ldots,(b,c))$. Let the bottom output vertices of R be the prefix of vertices of out(R) with the second coordinate equal to d, and let the right output vertices of R be the suffix of vertices of d with the first coordinate equal to d. Note that the bottom and the right output vertices of d intersect by one vertex d d.

Furthermore, we define $in(X \times Y) := in([0..|X|] \times [0..|Y|])$ and $out(X \times Y) := out([0..|X|] \times [0..|Y|])$. These are called the input and the output vertices of the augmented alignment graph $\overline{AG}^w(X,Y)$.

■ **Definition 5.6.** Let $X, Y \in \Sigma^*$ be two strings and $w : \overline{\Sigma}^2 \to [0..W]$ be a weight function. The boundary distance matrix $BM^w(X,Y)$ of these two strings is a matrix M of size $\mathcal{P}(X \times Y) \times \mathcal{P}(X \times Y)$ defined as follows: $M_{i,j} = \operatorname{dist}_{\overline{AG}^w(X,Y)}(p_i,q_j)$, where $p := \operatorname{in}(X \times Y)$ and $q := \operatorname{out}(X \times Y)$.

We also use $BM^w(A, B)$ for symbols A and B of some $SLP \mathcal{G}$ to denote $BM^w(\exp(A), \exp(B))$.

Note that $\mathrm{BM}^w(X,Y)$ is Monge due to Fact 3.12. Furthermore, if X[a..b) and Y[c..d) are two fragments of X and Y, then $\mathrm{BM}^w(X[a..b),Y[c..d))$ is a distance matrix from the input vertices of $[a..b]\times[c..d]$ in $\overline{\mathrm{AG}}^w(X,Y)$ to the output vertices of $[a..b]\times[c..d]$ as due to Lemma 5.2, no such path exits $[a..b]\times[c..d]$.

We now limit the size of the core of $BM^w(X,Y)$ in terms of the maximum weight of edits. As this fact will be later used in other contexts, it is phrased in a more general way.

Lemma 5.7. Let $X,Y \in \Sigma^*$ be two strings and $w: \overline{\Sigma}^2 \to [0..W]$ be a weight function. Let G be some induced subgraph of $\overline{\operatorname{AG}}^w(X,Y)$ such that G is connected and V(G) is orthogonally $\operatorname{convex^{15}}$. Say, $(p_i)_{i \in [1..n]}$ and $(q_i)_{i \in [1..m]}$ are two sequences of nodes of G such that matrix M of size

¹⁵ That is, every horizontal or vertical line crosses V(G) either by an empty set or by a segment of vertices of $\overline{\mathrm{AG}}^w(X,Y)$.

 $n \times m$ defined as $M_{i,j} = \operatorname{dist}_G(p_i, q_j)$ is Monge. Furthermore, for some positive integer t, we have $d_M(p_i, p_j) \le t$ for any $i, j \in [1..n]$ and $d_M(q_i, q_j) \le t$ for any $i, j \in [1..m]$, where d_M is Manhattan distance. Then $\delta(M) = \mathcal{O}(W \cdot t)$.

Proof. Note that as G is connected and V(G) is orthogonally convex, between any two vertices u and v in G there is a path that is monotone in both dimensions. As all edges in $\overline{AG}^{w}(X,Y)$ (and thus in G) have weights at most W+1, we have $\operatorname{dist}_G(v,u) \leq d_M(v,u) \cdot (W+1)$. Hence, $\operatorname{dist}_G(p_i, p_j) \leq t \cdot (W+1)$ for any $i, j \in [1..n]$ and $\operatorname{dist}_G(q_i, q_j) \leq t \cdot (W+1)$ for any $i, j \in [1..m]$. By triangle inequality, we have

$$\begin{split} M_{a,b} &= \mathrm{dist}_G(p_a, q_b) \leq \mathrm{dist}_G(p_a, p_c) + \mathrm{dist}_G(p_c, q_d) + \mathrm{dist}_G(q_d, q_b) \\ &\leq t \cdot (W+1) + \mathrm{dist}_G(p_c, q_d) + t \cdot (W+1) = M_{c,d} + 2t \cdot (W+1). \end{split}$$

Hence, $|M_{a,b} - M_{c,d}| = \mathcal{O}(W \cdot t)$ for any $a, c \in [1..n]$ and $b, d \in [1..m]$. The sum of the values of the core elements of M is equal to

$$M_{1,m} + M_{n,1} - M_{1,1} - M_{n,m} = (M_{1,m} - M_{1,1}) + (M_{n,1} - M_{n,m}) = \mathcal{O}(W \cdot t).$$

As all entries of M are integers, all entries of M^{\square} are also integers. Furthermore, they are nonnegative as M is Monge. Therefore, each core element of M has a value of at least one. Hence, $\delta(M) = \mathcal{O}(W \cdot t).$ 4

We now show a helper lemma.

Lemma 5.8. Consider strings $X, Y \in \Sigma^{\leq n}$ and a weight function $w : \overline{\Sigma}^2 \to [0..W]$. Furthermore, consider a sequence $(r_i)_{i \in [1..m]} =: (x_i, y_i)_{i \in [1..m]}$ of distinct vertices of $\overline{\mathrm{AG}}^w(X, Y)$, where $(x_i)_{i \in [1..m]}$ is non-decreasing and $(y_i)_{i \in [1..m]}$ is non-increasing. Let $(p_i)_{i \in [1..m_p]}$ and $(q_i)_{i \in [1..m_q]}$ be two contiguous subsequences of $(r_i)_{i \in [1..m]}$. Let M be a matrix of size $m_p \times m_q$ such that $M_{i,j} = \operatorname{dist}_{\overline{AG}^w(X,Y)}(p_i,q_j)$ for all $i \in [1..m_p]$ and $j \in [1..m_q]$.

Matrix M is Monge with $\delta(M) = \mathcal{O}((x_m - x_1) + (y_1 - y_m))$. Furthermore, there is an algorithm that in time $\mathcal{O}(((x_m - x_1) + (y_1 - y_m) + 1) \log n)$ builds $\mathsf{CMO}(M)$.

Proof. Note that every pair (p_i, q_i) satisfies the path irrelevance condition of Lemma 5.2. Therefore, all paths between them that are monotone in both dimensions are the shortest ones. Fix some such path P from r_1 to r_m going through all r_i . We calculate the distances $\operatorname{dist}_{\overline{AG}^w(X,Y)}(p_i,q_j)$ along this path (and its inverse). The fact that M is Monge follows directly from Fact 3.12 as all nodes p_i and q_i lie in the right order on the outer face of the subgraph of $\overline{AG}^w(X,Y)$ induced by P.

Consider node p_1 . Distances from it to all nodes q_i can be found in time $\mathcal{O}((x_m-x_1)+(y_1-y_m)+1)$ by going along P in both directions from p_1 . Analogously, we can find distances from all nodes p_i to q_1 . Therefore, we find all entries of M in the first row and the first column.

We now compute the core of M. Consider the i-th row of M. We find all entries $(i',j',v) \in$ core(M) with i'=i. Let p_i be the k-th element of the sequence r. Therefore, $p_{i+1}=r_{k+1}$. Consider the difference $M_{i+1,j} - M_{i,j} = \operatorname{dist}_{\overline{\mathrm{AG}}^w(X,Y)}(p_{i+1},q_j) - \operatorname{dist}_{\overline{\mathrm{AG}}^w(X,Y)}(p_i,q_j)$ for some fixed $j \in [1..m_q]$. Let q_j be the ℓ_j -th element of the sequence r. If $\ell_j \leq k$, we have $M_{i+1,j} - M_{i,j} = \ell_j$ $\operatorname{dist}_{\overline{AG}^w(X,Y)}(p_{i+1},p_i)$ as the path along P becomes longer by this fragment. Otherwise, if $\ell_j \geq k+1$, we have $M_{i+1,j} - M_{i,j} = -\operatorname{dist}_{\overline{\mathrm{AG}}^w(X,Y)}(p_i, p_{i+1})$ as the path along P becomes shorter by this fragment. Note that these differences do not depend on j. Therefore, if $\ell_j \geq k+1$ for all $j \in [1..m_q]$ or $\ell_j \leq k$ for all $j \in [1..m_q]$, then $A_{i,j}^{\square} = 0$ for all $j \in [1..m_q]$, and thus there are no entries $(i',j',v) \in \operatorname{core}(M)$ with i'=i. Otherwise, let j^* be the last index in $[1..m_q)$ such that $\ell_{j^*} \leq k$. We have that $M_{i,j}^{\square} = 0$ for all $j \neq j^*$ and $M_{i,j^*}^{\square} = \operatorname{dist}_{\overline{AG}^w(X,Y)}(p_{i+1},p_i) + \operatorname{dist}_{\overline{AG}^w(X,Y)}(p_i,p_{i+1})$. Hence, $(i,j^*,\operatorname{dist}_{\overline{AG}^w(X,Y)}(p_{i+1},p_i) + \operatorname{dist}_{\overline{AG}^w(X,Y)}(p_i,p_{i+1}))$ is the only entry $(i',j',v) \in \operatorname{core}(M)$ with i' = i.

By going through rows of M in the increasing order we can maintain j^* on the go using two pointers. Hence, we can compute $\operatorname{core}(M)$ in time $\mathcal{O}((x_m - x_1) + (y_1 - y_m) + 1)$. Furthermore, $\delta(M) \leq m_p - 1 \leq m - 1 \leq (x_m - x_1) + (y_1 - y_m)$. Moreover, as we have all entries of the first row and the first column of M and $\operatorname{core}(M)$, we may apply Lemma 4.3 to build $\operatorname{CMO}(M)$ in time $\mathcal{O}(((x_m - x_1) + (y_1 - y_m) + 1) \log n)$.

We are now ready to describe the composition procedure of BM^w . This procedure has two variants: one that computes all distances exactly, and a relaxed version of it that computes k-equivalent values.

Lemma 5.9. There is an algorithm that given strings X and Y, oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, and $\mathsf{CMO}(A)$ and $\mathsf{CMO}(B)$, where $A = \mathsf{BM}^w(X[a..b), Y[c..d))$ and $B = \mathsf{BM}^w(X[a..b), Y[d..e))$ for some $0 \le a < b \le |X|$ and $0 \le c < d < e \le |Y|$, in time $\mathcal{O}(m\log^3 m + \delta(A)\log^3 m + \delta(B)\log m + \delta(C)\log^3 m)$ for $m = \mathcal{P}([a..b] \times [c..e])$ builds $\mathsf{CMO}(C)$, where $C := \mathsf{BM}^w(X[a..b), Y[c..e))$.

Furthermore, there is an algorithm that, given a positive integer k, $\mathsf{CMO}(A')$ and $\mathsf{CMO}(B')$ for $A' \stackrel{k}{=} \mathsf{BM}^w(X[a..b), Y[c..d))$ and $B' \stackrel{k}{=} \mathsf{BM}^w(X[a..b), Y[d..e))$, in time $\mathcal{O}(\delta(A')\log^3 m + \delta(B')\log m + m\sqrt{k}\log^3 m)$ builds $\mathsf{CMO}(C')$ for some matrix C' such that $C' \stackrel{k}{=} \mathsf{BM}^w(X[a..b), Y[c..e))$ and $\delta(C') = \mathcal{O}(m\sqrt{k})$.

The same statements hold for two rectangles adjacent horizontally, not vertically.

Proof. We prove the lemma for rectangles adjacent vertically. The claim for rectangles adjacent horizontally is analogous.

We first prove the first statement of the lemma. Consider matrix D of distances in $\overline{\mathrm{AG}}^w(X,Y)$ between the left input vertices of $[a..b] \times [d+1..e]$ and the output vertices of $[a..b] \times [c..d]$. Due to Lemma 5.8, matrix D is Monge, $\delta(D) = \mathcal{O}(m)$, and we can build $\mathsf{CMO}(D)$ in time $\mathcal{O}(m \log m)$.

By applying Lemma 4.7, we can build $\mathsf{CMO}(E)$ for matrix $E \coloneqq (D^T \mid A^T)^T$ of distances in $\overline{\mathsf{AG}}^w(X,Y)$ from the input vertices of $[a..b] \times [c..e]$ to the output vertices of $[a..b] \times [c..d]$ in time $\mathcal{O}(m\log m + \delta(A)\log m)$, where $(D^T \mid A^T)$ is Monge due to Fact 3.12. The size of the core of E is $\mathcal{O}(\delta(A) + m)$.

Consider matrix F of distances in $\overline{\mathrm{AG}}^w(X,Y)$ from the input vertices of $[a..b] \times [c..e]$ to the left input vertices of $[a..b] \times [d+1..e]$. Due to Lemma 5.8, matrix F is Monge, $\delta(F) = \mathcal{O}(m)$, and we can build $\mathsf{CMO}(F)$ in time $\mathcal{O}(m\log m)$.

By applying Lemma 4.7 we can build CMO(G) for matrix $G := (F \mid E)$ of distances in $\overline{\mathrm{AG}}^w(X,Y)$ from the input vertices of $[a..b] \times [c..e]$ to the union of left input vertices of $[a..b] \times [d+1..e]$ and the output vertices of $[a..b] \times [c..d]$ in time $\mathcal{O}(m \log m + (\delta(A) + m) \log m)$, where $(F \mid E)$ is Monge due to Fact 3.12. The size of the core of G is $\mathcal{O}(\delta(A) + m)$.

By an analogous procedure starting from matrix B, in time $\mathcal{O}(m \log m + \delta(B) \log m)$ we can build $\mathsf{CMO}(H)$ for matrix H of distances in $\overline{\mathsf{AG}}^w(X,Y)$ from the union of left input vertices of $[a..b] \times [d+1..e]$ and the output vertices of $[a..b] \times [c..d]$ to the output vertices of $[a..b] \times [c..e]$, where $\delta(H) = \mathcal{O}(\delta(B) + m)$.

We claim that $G \otimes H = C$. It follows from the fact that vertices of the union of left input vertices of $[a..b] \times [d+1..e]$ and the output vertices of $[a..b] \times [c..d]$ are a separator between the input and the output vertices of $[a..b] \times [c..e]$ in the subgraph of $\overline{AG}^w(X,Y)$ induced by $[a..b] \times [c..e]$, and thus, every path between them inside $[a..b] \times [c..e]$ goes through this separator. Lemma 5.2

implies that every shortest path between boundary vertices of $[a..b] \times [c..e]$ indeed lies inside $[a..b] \times [c..e]$. As G contains distances from the input vertices of $[a..b] \times [c..e]$ to this separator in $\overline{\overline{AG}}^w(X,Y)$ and H contains distances from this separator to the output vertices of $[a.b] \times [c.e]$ in $\overline{AG}^w(X,Y)$, we get that $G\otimes H$ is a matrix of distances from the input to the output vertices of $[a..b] \times [c..e]$. Given $\mathsf{CMO}(G)$ and $\mathsf{CMO}(H)$, we apply Lemma 4.8 to get $\mathsf{CMO}(C)$ in time $\mathcal{O}(m\log^2 m + \delta(G)\log^3 m + \delta(G\otimes H)\log^3 m) = \mathcal{O}(m\log^2 m + (\delta(A) + m)\log^3 m + \delta(C)\log^3 m).$ Thus, the total time complexity is $\mathcal{O}((m\log m + \delta(A)\log m) + (m\log m + \delta(B)\log m) + (m\log^2 m) + (m\log^2$ $(\delta(A) + m)\log^3 m + \delta(C)\log^3 m) = \mathcal{O}(m\log^3 m + \delta(A)\log^3 m + \delta(B)\log m + \delta(C)\log^3 m).$

We now prove the second statement of the lemma. We proceed the same way as in the first statement but emphasize the differences. First, we build CMO(D) using Lemma 5.8 in time $O(m \log m)$. We note that all entries of $E[1..e-d+1][1..\mathcal{P}([a..b]\times[c..d])]$ are monotonically decreasing by rows and monotonically increasing by columns due to the definition of E as all left input vertices of $[a..b] \times [d..e]$ are to the left and down of all output vertices of $[a..b] \times [c..d]$. Furthermore, $A_{1,1} = 0$ as it represents the distance in $\overline{\mathrm{AG}}^w(X,Y)$ from (a,d) to (a,d). As $A' \stackrel{k}{=} A$, we have $A'_{1,1} = A_{1,1} = 0$. Therefore, as $E = (D^T \mid A^T)^T$, by applying Lemma 4.15 for k, $\mathsf{CMO}(D)$, $\mathsf{CMO}(A')$, and $\bar{\jmath} = 1$, in time $\mathcal{O}((m+\delta(A'))\log m)$ we can obtain $\mathsf{CMO}(E')$ for some E' such that $E' \stackrel{k}{=} E, \, \delta(E') = \mathcal{O}(\delta(A') + m)$, and $E'_{i,1} = E_{i,1}$ for all $i \in [1..e-d]$.

We now build CMO(F) in time $O(m \log m)$ using Lemma 5.8. We note that all entries of $G[e-d+1.\mathcal{P}([a.b]\times[c.e])][1.e-d+1]$ are monotonically increasing by rows and monotonically decreasing by columns due to the definition of G as all input vertices of $[a..b] \times [c..d]$ are to the right and up of all left input vertices of $[a..b] \times [d..e]$. Furthermore, $E'_{i,1} = E_{i,1}$ for all $i \in [1..e-d]$. Moreover, $E_{e-d+1,1} = 0$ as it represents the distance in $\overline{AG}^w(X,Y)$ from (a,d) to (a,d). As $E' \stackrel{k}{=} E$, we have $E'_{e-d+1,1} = E_{e-d+1,1} = 0$. Finally, we can obtain $CMO(F^T)$ and $CMO(E'^T)$ using Lemma 4.7 in time $\mathcal{O}((m+\delta(A'))\log m)$. Therefore, as $G=(F\mid E)$, by applying Lemma 4.15 for k, CMO (F^T) , $\mathsf{CMO}(E'^T)$, and $\bar{\jmath} = e - d + 1$, in time $\mathcal{O}((m + \delta(A'))\log m)$ we can obtain $\mathsf{CMO}(G'')$ for some G'' such that $G'' \stackrel{k}{=} G^T$ and $\delta(G'') = \mathcal{O}(\delta(A') + m)$. Using Lemma 4.7, we can build $\mathsf{CMO}(G')$ for $G' := G''^T$ in time $\mathcal{O}((\delta(A')+m)\log m)$. We have $G'\stackrel{k}{=}G$. The whole procedure of building $\mathsf{CMO}(G')$ takes $\mathcal{O}((m+\delta(A'))\log m)$ time.

By an analogous procedure starting from matrix B', in time $\mathcal{O}((m+\delta(B'))\log m)$ we can build $\mathsf{CMO}(H')$ for some H' such that $H' \stackrel{k}{=} H$ and $\delta(H') = \mathcal{O}(\delta(B') + m)$. We now use Lemma 4.14 to calculate $\mathsf{CMO}(C')$ for some C' such that $C' \stackrel{k}{=} G' \otimes H' \stackrel{k}{=} G \otimes H = C$ and $\delta(C') = \mathcal{O}(m\sqrt{k})$ in time $\mathcal{O}((\delta(A')+m)\log^3 m + m\sqrt{k}\log^3 m)$. The overall time complexity is $\mathcal{O}((m+\delta(A'))\log m + (m+1)\log m)$ $\delta(B')\log m + (\delta(A') + m)\log^3 m + m\sqrt{k}\log^3 m) = \mathcal{O}(\delta(A')\log^3 m + \delta(B')\log m + m\sqrt{k}\log^3 m).$

Hierarchical Alignment Data Structure

The algorithm of Lemma 5.9 allows us to compute BM^w in a divide-and-conquer manner: to compute $\mathrm{BM}^w(X,Y)$, we first compute $\mathrm{BM}^w(X_1,Y)$ and $\mathrm{BM}^w(X_2,Y)$, where $X=X_1\cdot X_2$, and then use Lemma 5.9 to obtain $BM^w(X,Y)$. We use SLPs to obtain these hierarchical decompositions of strings. We will maintain $BM^w(A, B)$ for some pairs of symbols A and B of the underlying SLP. As decompositions provided by an SLP are not trees, we will utilize the fact that some computations can be reused. To track what nodes of the computation are already computed, we define the following notions.

Definition 5.10. Let \mathcal{G} be an SLP. We call a set $Q \subseteq \mathcal{S}^2_{\mathcal{G}}$ closed if it satisfies the following properties for every pair $(A, B) \in Q$:

 $\mathcal{O}(\log |Q|)$ time.

```
if len(A) ≥ len(B) and rhs(A) = A<sub>L</sub>A<sub>R</sub> for some A<sub>L</sub>, A<sub>R</sub> ∈ S<sub>G</sub>, then (A<sub>L</sub>, B), (A<sub>R</sub>, B) ∈ Q;
if len(B) > len(A) and rhs(B) = B<sub>L</sub>B<sub>R</sub> for some B<sub>L</sub>, B<sub>R</sub> ∈ S<sub>G</sub>, then (A, B<sub>L</sub>), (A, B<sub>R</sub>) ∈ Q.
Moreover, the closure cl(A, B) of a pair (A, B) ⊆ S<sup>2</sup><sub>G</sub> is defined as the minimal closed superset of {(A, B)} or, equivalently, as the intersection of all closed sets containing (A, B).
```

Note that cl(A, B) is exactly the set of all pairs of symbols (C, D) such that $BM^w(C, D)$ will be recursively computed when starting a divide-and-conquer computation procedure for $BM^w(A, B)$.

We now describe a data structure that maintains all computed boundary distance matrices and uses them to retrieve optimal alignments. We describe two variants of this data structure: one that computes all distances exactly, and a relaxed version of it that computes k-equivalent values.

Proposition 5.11. Let \mathcal{G} be a balanced SLP that may grow over time and let $w: \overline{\Sigma}^2 \to [0..W]$ be a weight function accessible through an $\mathcal{O}(1)$ -time oracle. There exists a hierarchical alignment data structure D that maintains an (initially empty) closed set $Q \subseteq \mathcal{S}^2_{\mathcal{G}}$ subject to the following operations: Insertion: Given $(A, B) \in \mathcal{S}^2_{\mathcal{G}}$, set $Q := Q \cup \operatorname{cl}(A, B)$. This operation takes $\mathcal{O}((L+1) \cdot W \log^3 n)$ time, where $L = \sum_{(C,D) \in \operatorname{cl}(A,B) \setminus Q} (\operatorname{len}(C) + \operatorname{len}(D))$ and $n = \operatorname{len}(A) + \operatorname{len}(B) + |Q \cup \operatorname{cl}(A,B)|$. Matrix retrieval: Given $(A, B) \in Q$, retrieve a pointer to CMO(BM^w(A, B)). This operation takes

Alignment retrieval: Given $(A, B) \in Q$ and vertices p, q of $\overline{AG}^w(A, B)$ such that p is an input vertex and q is an output vertex of $\overline{AG}^w(A, B)$, compute the breakpoint representation of a shortest path from p to q. This operation takes $\mathcal{O}((d+1)\log^2 n)$ time, where $d = \operatorname{dist}_{\overline{AG}^w(A,B)}(p,q)$ and $n = \operatorname{len}(A) + \operatorname{len}(B) + |Q|$.

Additionally, for every positive integer $k \in \mathbb{Z}_{>0}$, there is a relaxed hierarchical alignment data structure D^k that supports insertions in $\mathcal{O}((L+1)\sqrt{k}\log^3 n)$ time, but, for matrix retrieval queries, it is only guaranteed to return a pointer to $\mathsf{CMO}(M)$ for some M such that $M \stackrel{k}{=} \mathsf{BM}^w(A,B)$, and for alignment retrieval queries, it fails (reports an error) whenever d > k.

Proof. The data structure D consists of a dynamic dictionary that maps each pair $(A, B) \in Q$ to a pointer to $\mathsf{CMO}(\mathsf{BM}^w(A, B))$. In the relaxed version D^k , the pointer is to $\mathsf{CMO}(M)$ for some M such that $M \stackrel{k}{=} \mathsf{BM}^w(A, B)$ and $\delta(M) = \mathcal{O}((\mathsf{len}(A) + \mathsf{len}(B))\sqrt{k})$. Even though faster dynamic dictionaries exist, we simply use a balanced BST that supports insertions and retrievals in $\mathcal{O}(\log |Q|)$ time.

Insertion. The insertion algorithm first uses the dictionary to check if $(A, B) \in Q$; in that case, there is nothing to do. Otherwise, the algorithm considers three cases:

- 1. If len(A) = len(B) = 1, it constructs $BM^w(A, B)$ and, based on it, $CMO(BM^w(A, B))$ using oracle access to w.
- 2. If $\operatorname{len}(A) \geq \operatorname{len}(B)$ and $\operatorname{rhs}(A) = A_L A_R$, the algorithm recursively inserts (A_L, B) and (A_R, B) , both of which belong to $\operatorname{cl}(A, B)$. In case of D, we use Lemma 5.9 to construct $\operatorname{CMO}(\operatorname{BM}^w(A, B))$ using pointers to $\operatorname{CMO}(\operatorname{BM}^w(A_L, B))$ and $\operatorname{CMO}(\operatorname{BM}^w(A_R, B))$. In case of D^k , we use Lemma 5.9 to construct $\operatorname{CMO}(M)$ for some M such that $M \stackrel{k}{=} \operatorname{BM}^w(A, B)$ and $\delta(M) = \mathcal{O}((\operatorname{len}(A) + \operatorname{len}(B))\sqrt{k})$ using pointers to $\operatorname{CMO}(M_L)$ for some M_L such that $M_L \stackrel{k}{=} \operatorname{BM}^w(A_L, B)$ and $\delta(M_L) = \mathcal{O}((\operatorname{len}(A_L) + \operatorname{len}(B))\sqrt{k})$ and to $\operatorname{CMO}(M_R)$ for some M_R such that $M_R \stackrel{k}{=} \operatorname{BM}^w(A_R, B)$, and $\delta(M_R) = \mathcal{O}((\operatorname{len}(A_R) + \operatorname{len}(B))\sqrt{k})$.
- 3. If len(B) > len(A) and $rhs(B) = B_L B_R$, the algorithm proceeds symmetrically. In all cases, the algorithm concludes by inserting (A, B) into the dictionary.

Let us proceed with the running time analysis. For this, we call a recursive call meaningful if it discovered that (A, B) is not in the dictionary. Let us first analyze the local cost of each meaningful call, excluding recursive calls and the operations on the dictionary. If len(A) = len(B) = 1, it takes $\mathcal{O}(1)$ time to build $CMO(BM^w(A, B))$ using oracle access to w. Otherwise, the running time is dominated by the application of Lemma 5.9. Denoting m = len(A) + len(B), the cores of the matrices involved are of size $\mathcal{O}(m \cdot W)$ for D (by Lemma 5.7) and $\mathcal{O}(m \cdot \sqrt{k})$ for D^k (by Lemma 5.9), so these running times are $\mathcal{O}(m \cdot W \log^3 m)$ or $\mathcal{O}(m\sqrt{k}\log^3 m)$, respectively. Across all recursive calls, the values m sum up to L and are bounded from above by n, so the total cost of meaningful calls, excluding operations on the dictionary, is $\mathcal{O}(L \cdot W \log^3 n)$ or $\mathcal{O}(L\sqrt{k}\log^3 n)$, respectively. It remains to bound the total cost of operations on the dictionary; in particular, these operations dominate the cost of non-meaningful calls. For each call, these operations take $\mathcal{O}(\log |Q|)$ time, which can be charged to the meaningful parent call (except for the root call). Consequently, the total cost of dictionary operations is $\mathcal{O}((1+L)\log |Q|) = \mathcal{O}((1+L)\log n)$.

Matrix retrieval. The matrix retrieval operation is implemented trivially: it suffices to perform a dictionary lookup and return the obtained pointer. This costs $\mathcal{O}(\log |Q|)$ time.

Alignment retrieval. While describing the alignment retrieval algorithm, we focus on the case of D^k . The query algorithm for D can be obtained by setting $k = (W+1) \cdot (\mathsf{len}(A) + \mathsf{len}(B))$ below so that $k \geq d$.

Let $m := \operatorname{len}(A) + \operatorname{len}(B)$. As $(A, B) \in Q$, we have a pointer to $\mathsf{CMO}(M)$ for some M such that $M \stackrel{k}{=} \mathsf{BM}^w(A, B)$. Note that $d = \operatorname{dist}_{\overline{\mathsf{AG}}^w(A, B)}(p, q)$ is one of the entries of $\mathsf{BM}^w(A, B)$. We read the corresponding entry of M using the random access functionality of M; if the value exceeds k, we conclude that d > k and report a failure. Otherwise, the corresponding entry of M contains exactly d < k. Overall, we can retrieve d in $\mathcal{O}(\log n)$ time.

If d=0, there is a trivial shortest path from p to q always taking diagonal edges of weight zero. We can return the breakpoint representation of such a path in constant time. We henceforth assume d>0 and, furthermore, $\mathsf{len}(A) \ge \mathsf{len}(B)$; the case $\mathsf{len}(A) < \mathsf{len}(B)$ is analogous. If $\mathsf{len}(A) = \mathsf{len}(B) = 1$, then the path can be constructed in $\mathcal{O}(1)$ time using oracle access to w. Otherwise, $\mathsf{rhs}(A) = A_L A_R$ for some $A_L, A_R \in \mathcal{S}_{\mathcal{G}}$. Since Q is closed, we have $(A_L, B), (A_R, B) \in Q$. Let $R_L := [0..\mathsf{len}(A_L)] \times [0..\mathsf{len}(B)]$ and $R_R := [\mathsf{len}(A_L)..\mathsf{len}(A)] \times [0..\mathsf{len}(B)]$ be rectangles in $\overline{\mathsf{AG}}^w(A,B)$.

If $p,q \in R_L$, then we interpret p,q as vertices of $\overline{AG}^w(A_L,B)$ and recursively compute the breakpoint representation of a shortest path in $\overline{AG}^w(A_L,B)$; by Lemma 5.2, the shortest path stays within R_L . Similarly, if $p,q \in R_R$, then we recursively compute the breakpoint representation of a shortest path in $\overline{AG}^w(A_R,B)$. Otherwise, if $p \in R_R$ and $q \in R_L$, then, by Lemma 5.2, any monotone path from p to q is a shortest one. Thus, we pick any and construct its breakpoint representation. This takes $\mathcal{O}(d)$ time because the path does not contain any cost-0 edges.

In the remaining case when $p \in R_L$ and $Q \in R_R$, we consider a set $S := out(R_L) \cap in(R_R)$ of vertices of $\overline{\mathrm{AG}}^w(A,B)$. Observe that S is a separator between R_L and R_R , so the shortest path from p to q goes through some vertex $v \in S$, and we have $\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,q) = \mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,v) + \mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(v,q)$. In particular, $\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,v)$, $\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(v,q) \leq \mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,q) = d \leq k$.

By Lemma 5.3, we can identify $\mathcal{O}(d)$ vertices $v \in S$ for which $\operatorname{dist}_{\overline{AG}^w(A,B)}(p,v) \leq d$ may hold. For each such vertex, we can compute $\min\{k+1,\operatorname{dist}_{\overline{AG}^w(A,B)}(p,v)\}$. Due to Lemma 5.2, an optimal path from p to v lies completely inside R_L , and thus the distance from p to v is an entry of $\operatorname{BM}^w(A_L,B)$. The matrix retrieval operation provides pointers to $\operatorname{CMO}(M_L)$ for some

 $M_L \stackrel{k}{=} \mathrm{BM}^w(A_L, B)$. Hence, by accessing M_L using $\mathrm{CMO}(M_L)$ in time $\mathcal{O}(\log n)$, we can find some value that is k-equivalent to $\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,v)$. If such a value is larger than k, we cap it with k+1, and if such a value is at most k, it is equal to $\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,v)$. Analogously, in time $\mathcal{O}(\log n)$ we can compute $\min\{k+1,\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(v,q)\}$. Overall, in $\mathcal{O}(d\log n)$ time we are able to identify a vertex $v \in S$ such that $\mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,q) = \mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(p,v) + \mathrm{dist}_{\overline{\mathrm{AG}}^w(A,B)}(v,q)$.

vertex $v \in S$ such that $\operatorname{dist}_{\overline{AG}^w(A,B)}(p,q) = \operatorname{dist}_{\overline{AG}^w(A,B)}(p,v) + \operatorname{dist}_{\overline{AG}^w(A,B)}(v,q)$. We recursively compute the breakpoint representation of an optimal path from the input vertex of $\overline{AG}^w(A_L,B)$ corresponding to p to the output vertex of $\overline{AG}^w(A_L,B)$ corresponding to v, and the breakpoint representation of an optimal path from the input vertex of $\overline{AG}^w(A_R,B)$ corresponding to v to the output vertex of $\overline{AG}^w(A_R,B)$ corresponding to v. We then reinterpret these breakpoint representations such that they represent a path from v to v and a path from v to v in $\overline{AG}^w(A,B)$. We combine them to obtain the breakpoint representation of an optimal path from v to v.

The overall running time can be expressed using recursion $T(m,0) = \mathcal{O}(\log n)$ and $T(m,d) = \mathcal{O}(d\log n) + T(m_L, d_L) + T(m_R, d_R)$ for $d \ge 1$, where $m_L, m_R \le 0.9m$ (because the SLP is balanced) and $d_L + d_R = d$, so the total time complexity of the recursive procedure is $\mathcal{O}((d+1)\log n\log m)$.

We now describe an alternative specification of closure.

Lemma 5.12. Consider an SLP \mathcal{G} . For every pair of symbols $(A, B) \in \mathcal{S}^2_{\mathcal{G}}$, the set cl(A, B) consists precisely of pairs (symb(α), symb(β)) taken over all nodes $\alpha \in \mathcal{T}(A)$ and $\beta \in \mathcal{T}(B)$ such that

$$\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \geq \mathsf{len}(\beta) \ \ or \ \ \mathsf{plen}(\alpha) \geq \mathsf{len}(\beta) > \mathsf{len}(\alpha), \ \ where \ \ \mathsf{plen}(\nu) = \begin{cases} \mathsf{len}(\mu) & \textit{if } \nu \ \textit{has parent } \mu, \\ \infty & \textit{if } \nu \ \textit{is the root.} \end{cases}$$

Proof. Let us first prove that $(\mathsf{symb}(\alpha), \mathsf{symb}(\beta)) \in \mathsf{cl}(A, B)$ holds for every pair (α, β) of nodes satisfying $\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta) > \mathsf{len}(\alpha)$. We proceed by induction on the depths of α and β and consider three main cases:

- 1. If $plen(\alpha) = plen(\beta) = \infty$, then $(symb(\alpha), symb(\beta)) = (A, B)$ belongs to cl(A, B) by definition.
- 2. If $\mathsf{plen}(\alpha) \ge \mathsf{plen}(\beta) \ne \infty$, we claim that the parent β' of β satisfies $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta') > \mathsf{len}(\alpha)$. The first equality follows directly from $\mathsf{plen}(\alpha) \ge \mathsf{plen}(\beta)$. The second one follows from $\mathsf{plen}(\beta) > \mathsf{len}(\alpha)$ or $\mathsf{len}(\beta) > \mathsf{len}(\alpha)$, depending on which of the two conditions (α, β) satisfies. In either case, the inductive assumption implies $(\mathsf{symb}(\alpha), \mathsf{symb}(\beta')) \in \mathsf{cl}(A, B)$. Since $\mathsf{len}(\beta') > \mathsf{len}(\alpha)$ and β is a child of β' , the definition of closed sets implies $(\mathsf{symb}(\alpha), \mathsf{symb}(\beta)) \in \mathsf{cl}(A, B)$.
- 3. If $\mathsf{plen}(\beta) > \mathsf{plen}(\alpha)$, we claim that the parent α' of α satisfies $\mathsf{plen}(\beta) > \mathsf{len}(\alpha') \ge \mathsf{len}(\beta)$. The first equality follows directly from $\mathsf{plen}(\beta) > \mathsf{plen}(\alpha)$. The second one follows from $\mathsf{len}(\alpha) \ge \mathsf{len}(\beta)$ or $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta)$, depending on which of the two conditions (α, β) satisfies. In either case, the inductive assumption implies $(\mathsf{symb}(\alpha'), \mathsf{symb}(\beta)) \in \mathsf{cl}(A, B)$. Since $\mathsf{len}(\alpha') \ge \mathsf{len}(\beta)$ and α is a child of α' , the definition of closed sets implies $(\mathsf{symb}(\alpha), \mathsf{symb}(\beta)) \in \mathsf{cl}(A, B)$.

It remains to prove that every element of $\mathsf{cl}(A,B)$ is of the form $(\mathsf{symb}(\alpha),\mathsf{symb}(\beta))$ for a pair (α,β) of nodes satisfying $\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta)$ or $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta) > \mathsf{len}(\alpha)$. The claim is certainly true for the pair (A,B) itself, which is of the form $(\mathsf{symb}(\alpha),\mathsf{symb}(\beta))$ for the roots of $\mathcal{T}(A)$ and $\mathcal{T}(B)$, respectively. For the remaining pairs, we proceed by induction on $\mathsf{len}(A) + \mathsf{len}(B)$ and consider three cases:

- 1. If len(A) = len(B) = 1, then $cl(A, B) = \{(A, B)\}$, so there is nothing more to prove.
- 2. If $\operatorname{len}(A) \ge \operatorname{len}(B)$ and $\operatorname{rhs}(A) = A_L A_R$, then $\operatorname{cl}(A,B) = \{(A,B)\} \cup \operatorname{cl}(A_L,B) \cup \operatorname{cl}(A_R,B)$. By the inductive assumption, every element of $\operatorname{cl}(A_L,B) \cup \operatorname{cl}(A_R,B)$ is of the form $(\operatorname{symb}(\alpha'),\operatorname{symb}(\beta))$ for a pair (α',β) of nodes satisfying $\operatorname{plen}(\beta) > \operatorname{len}(\alpha') \ge \operatorname{len}(\beta)$ or $\operatorname{plen}(\alpha') \ge \operatorname{len}(\beta) > \operatorname{len}(\alpha')$. The corresponding node α in $\mathcal{T}(A)$ satisfies $\operatorname{symb}(\alpha) = \operatorname{symb}(\alpha')$, $\operatorname{len}(\alpha) = \operatorname{len}(\alpha')$, and $\operatorname{plen}(\alpha) = \operatorname{len}(\alpha')$.

- $\min\{\mathsf{len}(A),\mathsf{plen}(\alpha')\}$. Since $\mathsf{len}(A) \ge \mathsf{len}(B) \ge \mathsf{len}(\beta)$, we conclude that $\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta)$ or $plen(\alpha) \ge len(\beta) > len(\alpha)$ holds as claimed.
- 3. If len(B) > len(A) and $rhs(B) = B_L B_R$, then $cl(A, B) = \{(A, B)\} \cup cl(A, B_L) \cup cl(A, B_R)$. By the inductive assumption, every element of $cl(A, B_L) \cup cl(A, B_R)$ is of the form $(symb(\alpha), symb(\beta'))$ for a pair (α, β') of nodes satisfying $\mathsf{plen}(\beta') > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta')$ or $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta') > \mathsf{len}(\alpha)$. The corresponding node β in $\mathcal{T}(B)$ satisfies $\mathsf{symb}(\beta) = \mathsf{symb}(\beta')$, $\mathsf{len}(\beta) = \mathsf{len}(\beta')$ and $\mathsf{plen}(\beta) =$ $\min\{\mathsf{len}(B),\mathsf{plen}(\beta')\}$. Since $\mathsf{len}(B) > \mathsf{len}(A) \ge \mathsf{len}(\alpha)$, we conclude that $\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta)$ or $plen(\alpha) > len(\beta) > len(\alpha)$ holds as claimed.

We now prove a lemma that helps bound the time complexity of the insertion operation of Proposition 5.11 if some "similar" pair of symbols is already present in the set Q of D.

Corollary 5.13. Let \mathcal{G} be a balanced SLP and let $A, A', B, B' \in \mathcal{S}_{\mathcal{G}}$. If the parse tree $\mathcal{T}(A)$ contains exactly u symbols that do not occur in $\mathcal{T}(A')$, then

$$\sum_{(C,D)\in \mathsf{cl}(A,B)\backslash \mathsf{cl}(A',B)} \left(\mathsf{len}(C) + \mathsf{len}(D) \right) = \mathcal{O}\left((u+1)(\mathsf{len}(A) + \mathsf{len}(B)) + \mathsf{len}(B) \log \left(1 + \frac{\mathsf{len}(B)}{\mathsf{len}(A)} \right) \right).$$

Analogously, if the parse tree $\mathcal{T}(B)$ contains exactly u symbols that do not occur in $\mathcal{T}(B')$, then

$$\sum_{(C,D)\in \mathsf{cl}(A,B)\backslash \mathsf{cl}(A,B')} \left(\mathsf{len}(C) + \mathsf{len}(D) \right) = \mathcal{O}\left((u+1)(\mathsf{len}(A) + \mathsf{len}(B)) + \mathsf{len}(A)\log\left(1 + \frac{\mathsf{len}(A)}{\mathsf{len}(B)}\right) \right).$$

Proof. We focus on the first part of the claim; the proof of the second part is analogous. Let $\mathcal{U} \subseteq \mathcal{S}_{\mathcal{G}}$ be the set of u symbols that occur in $\mathcal{T}(A)$ but do not occur in $\mathcal{T}(A')$. Moreover, let $\mathcal{V} \subseteq \mathcal{S}_{\mathcal{G}}$ consist of A as well as all symbols that occur in rhs(U) for $U \in \mathcal{U} \cap \mathcal{N}_{\mathcal{G}}$. Observe that $|\mathcal{V}| \leq 2u + 1$ since $|\operatorname{rhs}(U)| \leq 2 \text{ holds for every } U \in \mathcal{N}_{\mathcal{G}}.$

Claim 5.14. Every $(C, D) \in cl(A, B) \setminus cl(A', B)$ satisfies $C \in \mathcal{V}$.

Proof. For a proof by contradiction, suppose that $C \notin \mathcal{V}$. By Lemma 5.12, there are nodes α and β in $\mathcal{T}(A)$ and $\mathcal{T}(B)$, respectively, such that $C = \mathsf{symb}(\alpha)$, $D = \mathsf{symb}(\beta)$, and either $\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta)$ or $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta) > \mathsf{len}(\alpha)$. Observe that α is not the root of $\mathcal{T}(A)$ because $C \neq A$. Let E denote the symbol of the parent of α so that $\mathsf{plen}(\alpha) = \mathsf{len}(E)$. The assumption $C \notin \mathcal{V}$ implies $E \notin \mathcal{U}$, so there is a node in $\mathcal{T}(A')$ with symbol E. Since C occurs in rhs(E), this node has a child α' such that $symb(\alpha') = C$ and $plen(\alpha') = len(E) = plen(\alpha)$. Consequently, $plen(\beta) > len(\alpha') > len(\beta)$ or $plen(\alpha') > len(\beta) > len(\alpha')$ follows from analogous conditions on α and β . Finally, Lemma 5.12 implies that $(C,D) = (\mathsf{symb}(\alpha'), \mathsf{symb}(\beta)) \in \mathsf{cl}(A',B)$, contradicting the choice of (C, D).

Next, we bound the contribution of every $C \in \mathcal{V}$.

Claim 5.15. Every $C \in \mathcal{S}_{\mathcal{G}}$ satisfies

$$\sum_{D:(C,D)\in\mathsf{cl}(A,B)} \left(\mathsf{len}(C) + \mathsf{len}(D)\right) = \begin{cases} \mathcal{O}\left(\mathsf{len}(A) + \mathsf{len}(B) \cdot \log\left(1 + \frac{\mathsf{len}(B)}{\mathsf{len}(A)}\right)\right) & \text{if } C = A \\ \mathcal{O}(\mathsf{len}(A) + \mathsf{len}(B)) & \text{otherwise.} \end{cases}$$

Proof. Let us fix a node α of $\mathcal{T}(A)$ that satisfies $\mathsf{symb}(\alpha) = C$ and maximizes $\mathsf{plen}(\alpha)$. By Lemma 5.12, we have $(C, D) \in \mathsf{cl}(A, B)$ if and only if there is a node β of $\mathcal{T}(B)$ that satisfies $\mathsf{symb}(\beta) = D$ and either $plen(\beta) > len(\alpha) \ge len(\beta)$ or $plen(\alpha) \ge len(\beta) > len(\alpha)$.

Each root-to-leaf path in $\mathcal{T}(B)$ contributes at most one node β satisfying $\mathsf{plen}(\beta) > \mathsf{len}(\alpha) \ge \mathsf{len}(\beta)$. Excluding the root of $\mathcal{T}(B)$, each such node β satisfies $\mathsf{len}(\alpha) + \mathsf{len}(\beta) < \mathsf{plen}(\beta) + \mathsf{len}(\beta) \le 5 \cdot \mathsf{len}(\beta)$ because \mathcal{G} is balanced. Consequently, the total contribution of $\mathsf{len}(\alpha) + \mathsf{len}(\beta)$ across all such non-root node β does not exceed $5 \cdot \mathsf{len}(B)$. The contribution of the root of $\mathcal{T}(B)$ does not exceed $\mathsf{len}(C) + \mathsf{len}(B)$.

It remains to focus on the nodes β satisfying $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta) > \mathsf{len}(\alpha)$. Let us first assume that $C \ne A$ so that α is not the root of $\mathcal{T}(A)$ and $\mathsf{plen}(\alpha) \le 4 \cdot \mathsf{len}(\alpha)$ holds because \mathcal{G} is balanced. Since the lengths of subsequent nodes on a single root-to-leaf path are smaller by a factor of at least $\frac{4}{3}$, the number of such nodes β on the path does not exceed $\lceil \log_{4/3}(\mathsf{plen}(\alpha)/\mathsf{len}(\alpha)) \rceil \le \lceil \log_{4/3} 4 \rceil = 5$. Each such node β satisfies $\mathsf{len}(\alpha) + \mathsf{len}(\beta) < 2 \cdot \mathsf{len}(\beta)$, so the total contribution of $\mathsf{len}(\alpha) + \mathsf{len}(\beta)$ across all such nodes does not exceed $10 \cdot \mathsf{len}(B)$. Overall, if $C \ne A$, then the total contribution of all nodes β is at most $\mathsf{len}(C) + 15 \cdot \mathsf{len}(B) = \mathcal{O}(\mathsf{len}(A) + \mathsf{len}(B))$.

Next, suppose that C=A. In this case, the condition $\mathsf{plen}(\alpha) \ge \mathsf{len}(\beta) > \mathsf{len}(\alpha)$ is equivalent to $\mathsf{len}(B) \ge \mathsf{len}(\beta) > \mathsf{len}(A)$. Since the lengths of subsequent nodes on a single root-to-leaf path are smaller by a factor of at least $\frac{4}{3}$, the number of such nodes β on the path does not exceed $\lceil \log_{4/3}(\mathsf{len}(B)/\mathsf{len}(A)) \rceil$. Each such node β satisfies $\mathsf{len}(\alpha) + \mathsf{len}(\beta) < 2 \cdot \mathsf{len}(\beta)$, so the total contribution of $\mathsf{len}(\alpha) + \mathsf{len}(\beta)$ across all such nodes does not exceed $2 \cdot \mathsf{len}(B) \cdot \lceil \log_{4/3}(\mathsf{len}(B)/\mathsf{len}(A)) \rceil$. Overall, if C = A, then the total contribution of all nodes β is at most $\mathcal{O}(\mathsf{len}(A) + \mathsf{len}(B) \log(1 + \mathsf{len}(B)/\mathsf{len}(A)))$.

Combining the two claims with $|\mathcal{V}| = \mathcal{O}(u+1)$, we conclude that

$$\sum_{(C,D)\in \mathsf{cl}(A,B)\backslash \mathsf{cl}(A',B)} \left(\mathsf{len}(C) + \mathsf{len}(D) \right) = \mathcal{O}\left((u+1)(\mathsf{len}(A) + \mathsf{len}(B)) + \mathsf{len}(B) \log \left(1 + \frac{\mathsf{len}(B)}{\mathsf{len}(A)} \right) \right). \quad \blacksquare$$

6 Self-Edit Distance

One crucial ingredient of our algorithms is the notions of self-alignments and self edit distance introduced by Cassis, Kociumaka, and Wellnitz in [CKW23].

■ Definition 6.1 ([CKW23]). We say that an alignment $A: X \rightsquigarrow X$ is a self-alignment if A does not align any character X[x] to itself. We define the self edit distance of X as self-ed(X) := $\min_{A} \operatorname{ed}_{A}(X,X)$, where the minimization ranges over self-alignments $A: X \rightsquigarrow X$. In words, $\operatorname{self-ed}(X)$ is the minimum (unweighted) cost of a self-alignment.

We can interpret a self-alignment as a $(0,0) \leadsto (|X|,|X|)$ path in the alignment graph AG(X,X) that does not contain any edges of the main diagonal.

We list some fundamental properties of self-ed.

Lemma 6.2 (Properties of self-ed, extension of [CKW23, Lemma 4.2]). Let $X \in \Sigma^*$ denote a string. Then, all of the following hold:

Monotonicity. For any $\ell' \leq \ell \leq r \leq r' \in [0..|X|]$, we have $\operatorname{self-ed}(X[\ell..r)) \leq \operatorname{self-ed}(X[\ell'..r'))$. **Sub-additivity.** For any $m \in [0..|X|]$, we have $\operatorname{self-ed}(X) \leq \operatorname{self-ed}(X[0..m)) + \operatorname{self-ed}(X[m..|X|))$. **Triangle inequality.** For any $Y \in \Sigma^*$, we have $\operatorname{self-ed}(Y) \leq \operatorname{self-ed}(X) + 2\operatorname{ed}(X,Y)$. **Continuity.** For any $i \in (0..|X|)$, we have

$$self-ed(X[0..i)) \le self-ed(X[0..i+1)) \le self-ed(X[0..i)) + 1.$$

۵

Proof. The first three properties are proven in [CKW23, Lemma 4.2]. Hence, it remains to prove continuity. $\mathsf{self-ed}(X[0..i)) \leq \mathsf{self-ed}(X[0..i+1))$ follows from monotonicity of $\mathsf{self-ed}$. We now prove $\mathsf{self-ed}(X[0..i+1)) \leq \mathsf{self-ed}(X[0..i)) + 1$.

Consider an optimal self-alignment $(x_i, y_i)_{i=0}^t =: \mathcal{A} : X[0..i) \leadsto X[0..i)$. We have $(x_t, y_t) = (i, i)$. Note that $(x_{t-1}, y_{t-1}) \neq (i-1, i-1)$ as \mathcal{A} is a self-alignment. Therefore, $(x_{t-1}, y_{t-1}) = (i-1, i)$ or $(x_{t-1}, y_{t-1}) = (i, i-1)$. Without loss of generality assume that $(x_{t-1}, y_{t-1}) = (i, i-1)$. Consider an alignment $(x'_i, y'_i)_{i=0}^{t+1} =: \mathcal{B} : X[0..i+1) \leadsto X[0..i+1)$ where $(x'_i, y'_i) = (x_i, y_i)$ for $i \in [0..t-1]$, $(x'_t, y'_t) = (i+1, i)$, and $(x'_{t+1}, y'_{t+1}) = (i+1, i+1)$. We obtain

$$\begin{split} \mathsf{self-ed}(X[0..i+1)) &\leq \mathsf{ed}_{\mathcal{B}}(X[0..i+1), X[0..i+1)) \\ &= \mathsf{ed}_{(x_i,y_i)_{i=0}^{t-1}}(X[0..i), X[0..i-1)) + 2 \\ &= (\mathsf{ed}_{\mathcal{A}}(X[0..i), X[0..i)) - 1) + 2 \\ &= \mathsf{self-ed}(X[0..i)) + 1. \end{split}$$

Similarly to Fact 3.10, there is a PILLAR algorithm that computes self-ed(X) if this value is bounded by k, in time $\mathcal{O}(k^2)$.

Fact 6.3 ([CKW23, Lemma 4.5]). There is an $\mathcal{O}(k^2)$ -time PILLAR algorithm that, given a string $X \in \Sigma^*$ and an integer $k \in \mathbb{Z}_{>0}$ determines whether self-ed(X) ≤ k and, if so, retrieves the breakpoint representation of an optimal self-alignment $A: X \rightsquigarrow X$.

An important property of self-ed is that if self-ed(X) is large, then any two cheap alignments of X onto Y intersect.

- Fact 6.4 ([CKW23, Lemma 4.3]). Consider strings $X, Y \in \Sigma^*$. If two alignments $A, B \in X \rightsquigarrow Y$ intersect only at the endpoints (0,0) and (|X|,|Y|), then self-ed $(X) \leq \operatorname{ed}_{A}(X,Y) + \operatorname{ed}_{B}(X,Y)$. ■
- **Corollary 6.5.** Consider strings $X, Y \in \Sigma^*$ and alignments $A, B : X \rightsquigarrow Y$. If (x, y) and (x', y') are two subsequent intersection points in $A \cap B$, then self-ed $(X[x..x']) \leq \operatorname{ed}_A(X,Y) + \operatorname{ed}_B(X,Y)$.

Proof. Let X' = X[x..x') and Y' = Y[y..y'). Since $(x,y), (x',y') \in \mathcal{A} \cap \mathcal{B}$, restricting the alignments \mathcal{A} and \mathcal{B} to X' yields alignments $\mathcal{A}', \mathcal{B}' : X' \leadsto Y'$. Moreover, since (x,y), (x',y') are two subsequent points in the intersection $\mathcal{A} \cap \mathcal{B}$, we conclude that \mathcal{A}' and \mathcal{B}' intersect only at the endpoints. Hence, Fact 6.4 yields self-ed $(X') \leq \operatorname{ed}_{\mathcal{A}'}(X',Y') + \operatorname{ed}_{\mathcal{B}'}(X',Y')$, that is, self-ed $(X[x..x')) \leq \operatorname{ed}_{\mathcal{A}'}(X[x..x'),Y[y..y')) + \operatorname{ed}_{\mathcal{B}'}(X[x..x'),Y[y..y'))$. The final claim follows from $\operatorname{ed}_{\mathcal{A}'}(X',Y') \leq \operatorname{ed}_{\mathcal{B}}(X,Y)$ and $\operatorname{ed}_{\mathcal{B}'}(X',Y') \leq \operatorname{ed}_{\mathcal{B}}(X,Y)$.

We use this fact to derive our main combinatorial lemmas that describe how to obtain an optimal alignment of X onto Y given optimal alignments of some fragments of X and Y if these fragments intersect by parts of "sufficiently large" self edit distance.

Lemma 6.6. Consider a normalized weight function $w: \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$, strings $X, Y \in \Sigma^*$, an alignment $(x_i, y_i)_{i=0}^t =: \mathcal{A}: X \leadsto Y$, and indices $0 \leq \ell \leq r \leq t$ satisfying at least one of the following conditions:

$$\operatorname{self-ed}(X[x_{\ell}..x_r)) > 4 \cdot \operatorname{ed}_{A}^{w}(X,Y), \qquad \ell = 0, \quad or \quad r = t.$$

If $\mathcal{B}_L: X[0..x_r) \rightsquigarrow Y[0..y_r)$ and $\mathcal{B}_R: X[x_\ell..|X|) \rightsquigarrow Y[y_\ell..|Y|)$ are w-optimal alignments, then $\mathcal{B}_L \cap \mathcal{B}_R \neq \emptyset$ and every point $(x^*, y^*) \in \mathcal{B}_L \cap \mathcal{B}_R$ satisfies

$$\operatorname{ed}^w(X,Y) = \operatorname{ed}^w_{\mathcal{B}_L}(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w_{\mathcal{B}_R}(X[x^*..|X|),Y[y^*..|Y|)).$$

Proof. Let us first consider the case when self-ed($X[x_{\ell}..x_r)$) > 4k, where $k := \operatorname{ed}_{\mathcal{A}}^w(X,Y)$. Consider alignments \mathcal{C}_L obtained by concatenating \mathcal{B}_L with $\mathcal{A}_R := (x_i, y_i)_{i=r}^t$ and \mathcal{C}_R obtained by concatenating $\mathcal{A}_L := (x_i, y_i)_{i=0}^\ell$ with \mathcal{B}_R . Observe that

$$\begin{split} \operatorname{ed}_{\mathcal{C}_L}^w(X,Y) &= \operatorname{ed}_{\mathcal{B}_L}^w(X[0..x_r),Y[0..y_r)) + \operatorname{ed}_{\mathcal{A}}^w(X[x_r..|X|),Y[y_r..|Y|)) \\ &\leq \operatorname{ed}_{\mathcal{A}}^w(X[0..x_r),Y[0..y_r)) + \operatorname{ed}_{\mathcal{A}}^w(X[x_r..|X|),Y[y_r..|Y|)) \\ &= \operatorname{ed}_{\mathcal{A}}^w(X,Y) = k. \end{split}$$

A symmetric argument yields $\operatorname{ed}_{\mathcal{C}_B}^w(X,Y) \leq k$.

By Corollary 6.5, if (x, y), (x', y') are consecutive points in $\mathcal{C}_L \cap \mathcal{C}_R$, then $\mathsf{self-ed}(X[x_{\ell}..x')) \leq 2k$. Due to $\mathsf{self-ed}(X[x_{\ell}..x_r)) > 4k$, there exists a point $(x^*, y^*) \in \mathcal{C}_L \cap \mathcal{C}_R$ such that $x_{\ell} < x^* < x_r$ and $y_{\ell} \leq y^* \leq y_r$. In particular, $\mathcal{B}_L \cap \mathcal{B}_R \neq \emptyset$.

For the remainder of the proof, let us pick $(x^*, y^*) \in \mathcal{B}_L \cap \mathcal{B}_R$. Since $\mathsf{self-ed}(X[x_\ell..x_r)) > 4k$, subadditivity of $\mathsf{self-ed}$ implies that $\mathsf{self-ed}(X[x_\ell..x^*)) > 2k$ or $\mathsf{self-ed}(X[x^*..x_r)) > 2k$. By symmetry (up to reversal), it suffices to consider the case of $\mathsf{self-ed}(X[x^*..x_r)) > 2k$.

Consider a w-optimal alignment $\mathcal{O}: X \rightsquigarrow Y$. Since $\operatorname{ed}_{\mathcal{O}}^w(X,Y) \leq \operatorname{ed}_{\mathcal{A}}^w(X,Y) = k$, by Corollary 6.5, if (x,y) and (x',y') are two consecutive points in $\mathcal{C}_L \cap \mathcal{O}$, then $\operatorname{self-ed}(X[x..x')) \leq 2k$. Due to $\operatorname{self-ed}(X[x^*..x_r)) > 2k$, there exists a point $(\bar{x},\bar{y}) \in \mathcal{C}_L \cap \mathcal{O}$ such that $x^* < \bar{x} < x_r$ and $y^* \leq \bar{y} \leq y_r$. In particular, $(\bar{x},\bar{y}) \in \mathcal{B}_L \cap \mathcal{O}$. Since the alignments \mathcal{O} , \mathcal{B}_L , and \mathcal{B}_R are w-optimal, we conclude that

$$\begin{split} \operatorname{ed}^w(X,Y) &= \operatorname{ed}^w_{\mathcal{O}}(X,Y) \\ &= \operatorname{ed}^w_{\mathcal{O}}(X[0..\bar{x}),Y[0..\bar{y})) + \operatorname{ed}^w_{\mathcal{O}}(X[\bar{x}..|X|),Y[\bar{y}..|Y|)) \\ &= \operatorname{ed}^w_{\mathcal{B}_L}(X[0..\bar{x}),Y[0..\bar{y})) + \operatorname{ed}^w_{\mathcal{O}}(X[\bar{x}..|X|),Y[\bar{y}..|Y|)) \\ &= \operatorname{ed}^w_{\mathcal{B}_L}(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w_{\mathcal{B}_L}(X[x^*..\bar{x}),Y[y^*..\bar{y})) + \operatorname{ed}^w_{\mathcal{O}}(X[\bar{x}..|X|),Y[\bar{y}..|Y|)) \\ &= \operatorname{ed}^w_{\mathcal{B}_L}(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w(X[x^*..\bar{x}),Y[y^*..\bar{y})) + \operatorname{ed}^w(X[\bar{x}..|X|),Y[\bar{y}..|Y|)) \\ &\geq \operatorname{ed}^w_{\mathcal{B}_L}(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w(X[x^*..|X|),Y[y^*..|Y|)) \\ &= \operatorname{ed}^w(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w(X[x^*..|X|),Y[y^*..|Y|)) \\ &= \operatorname{ed}^w(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w(X[x^*..|X|),Y[y^*..|Y|)) \\ &\geq \operatorname{ed}^w(X,Y). \end{split}$$

It remains to consider the cases of $\ell = 0$ and r = t. By symmetry (up to reversal), we can solely focus on $\ell = 0$. Then, $(x_{\ell}, y_{\ell}) = (x_0, y_0) = (0, 0) \in \mathcal{B}_L \cap \mathcal{B}_R$. For the remainder of the proof, let us pick $(x^*, y^*) \in \mathcal{B}_L \cap \mathcal{B}_R$. Since the alignments $\mathcal{B}_L : X[0..x_r) \rightsquigarrow Y[0..y_r)$ and $\mathcal{B}_R : X \rightsquigarrow Y$ are w-optimal, we conclude that

$$\begin{split} \operatorname{ed}^w(X,Y) &= \operatorname{ed}^w_{\mathcal{B}_R}(X,Y) \\ &= \operatorname{ed}^w_{\mathcal{B}_R}(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w_{\mathcal{B}_R}(X[x^*..|X|),Y[y^*..|Y|)) \\ &= \operatorname{ed}^w(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w_{\mathcal{B}_R}(X[x^*..|X|),Y[y^*..|Y|)) \\ &= \operatorname{ed}^w_{\mathcal{B}_L}(X[0..x^*),Y[0..y^*)) + \operatorname{ed}^w_{\mathcal{B}_R}(X[x^*..|X|),Y[y^*..|Y|)). \end{split}$$

Proposition 6.7. Consider a normalized weight function $w: \overline{\Sigma}^2 \to \mathbb{R}_{\geq 0}$, strings $X, Y \in \Sigma^*$, an alignment $(x_i, y_i)_{i=0}^t =: \mathcal{A}: X \leadsto Y$, and indices $0 \leq \ell \leq m \leq r \leq t$ such that

$$\mathsf{self-ed}(X[x_\ell..x_m)) > 4 \cdot \mathsf{ed}_{\mathcal{A}}^w(X,Y) \quad or \ \ell = 0, \quad and \quad \mathsf{self-ed}(X[x_m..x_r)) > 4 \cdot \mathsf{ed}_{\mathcal{A}}^w(X,Y) \quad or \ r = t.$$

If $\mathcal{O}_L: X[0..x_m) \rightsquigarrow Y[0..y_m)$, $\mathcal{O}_M: X[x_\ell..x_r) \rightsquigarrow Y[y_\ell..y_r)$, and $\mathcal{O}_R: X[x_m..|X|) \rightsquigarrow Y[y_m..|Y|)$ are w-optimal alignments, then $\mathcal{O}_L \cap \mathcal{O}_M \neq \emptyset \neq \mathcal{O}_R \cap \mathcal{O}_M$, and all points $(x_L^*, y_L^*) \in \mathcal{O}_L \cap \mathcal{O}_M$ and $(x_R^*, y_R^*) \in \mathcal{O}_R \cap \mathcal{O}_M$ satisfy

$$\operatorname{ed}^w(X,Y) = \operatorname{ed}^w_{\mathcal{O}_L}(X[0..x_L^*),Y[0..y_L^*)) + \operatorname{ed}^w_{\mathcal{O}_M}(X[x_L^*..x_R^*),Y[y_L^*..y_R^*)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[y_R^*..|Y|)).$$

Proof. Let us first apply Lemma 6.6 for an alignment $(x_i, y_i)_{i=0}^r : X[0..x_r) \rightsquigarrow Y[0..y_r)$ and woptimal alignments $\mathcal{O}_L : X[0..x_m) \rightsquigarrow Y[0..y_m)$ and $\mathcal{O}_M : X[x_\ell..x_r) \rightsquigarrow Y[y_\ell..y_r)$. The assumptions are satisfied because $\ell = 0$ or self-ed $(X[x_\ell..x_m)) > 4 \cdot \operatorname{ed}_{\mathcal{A}}^w(X,Y) \ge 4 \cdot \operatorname{ed}_{\mathcal{A}}^w(X[0..x_r), Y[0..y_r))$. Consequently, $\mathcal{O}_L \cap \mathcal{O}_M \ne \emptyset$ and every $(x_L^*, y_L^*) \in \mathcal{O}_L \cap \mathcal{O}_M$ satisfies

$$\operatorname{ed}^{w}(X[0..x_{r}),Y[0..y_{r})) = \operatorname{ed}^{w}_{\mathcal{O}_{I}}(X[0..x_{L}^{*}),Y[0..y_{L}^{*})) + \operatorname{ed}^{w}_{\mathcal{O}_{M}}(X[x_{L}^{*}..x_{r}),Y[y_{L}^{*}..y_{r})).$$

Let $\mathcal{B}_L: X[0..x_r) \rightsquigarrow Y[0..y_r)$ be a w-optimal alignment that follows \mathcal{O}_L from (0,0) to (x_L^*, y_L^*) and then follows \mathcal{O}_M from (x_L^*, y_L^*) to (x_r, y_r) .

Next, we apply Lemma 6.6 for an alignment $\mathcal{A}: X \rightsquigarrow Y$ as well as w-optimal alignments $\mathcal{B}_L: X[0..x_r) \rightsquigarrow Y[0..y_r)$ and $\mathcal{O}_R: X[x_m..|X|) \rightsquigarrow Y[y_m..|Y|)$. The assumptions are satisfied because r = t or self-ed $(X[x_m..x_r]) > 4 \cdot \operatorname{ed}_{\mathcal{A}}^w(X,Y)$. The definition of \mathcal{B}_L implies that $\mathcal{B}_L \cap \mathcal{O}_R = \mathcal{O}_M \cap \mathcal{O}_R$. Consequently, $\mathcal{O}_M \cap \mathcal{O}_R = \mathcal{B}_L \cap \mathcal{O}_R \neq \emptyset$ and every $(x_R^*, y_R^*) \in \mathcal{B}_L \cap \mathcal{O}_R = \mathcal{O}_M \cap \mathcal{O}_R$ satisfies

$$\operatorname{ed}^{w}(X,Y) = \operatorname{ed}_{\mathcal{B}_{x}}^{w}(X[0..x_{R}^{*}),Y[0..y_{R}^{*})) + \operatorname{ed}_{\mathcal{O}_{R}}^{w}(X[x_{R}^{*}..|X|),Y[y_{R}^{*}..|Y|)).$$

Since $x_L^* \le x_m \le x_R^*$ and $y_L^* \le y_m \le y_R^*$, we have

$$\operatorname{ed}_{\mathcal{B}_L}^w(X[0..x_R^*),Y[0..y_R^*)) = \operatorname{ed}_{\mathcal{O}_L}^w(X[0..x_L^*),Y[0..y_L^*)) + \operatorname{ed}_{\mathcal{O}_M}^w(X[x_L^*..x_R^*),Y[y_L^*..y_R^*)).$$

Therefore,

$$\operatorname{ed}^w(X,Y) = \operatorname{ed}^w_{\mathcal{O}_L}(X[0..x_L^*),Y[0..y_L^*)) + \operatorname{ed}^w_{\mathcal{O}_M}(X[x_L^*..x_R^*),Y[y_L^*..y_R^*)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[y_R^*..|Y|)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[y_R^*..|X|)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[x_R^*..|X|)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[x_R^*..|X|)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),X[x_R^*..|X|)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|$$

4

holds as claimed.

7 Static Algorithm

In this section we describe our static algorithm that, given two strings $X,Y \in \Sigma^{\leq n}$ and a weight function $w: \overline{\Sigma}^2 \to [0..W]$, finds $k = \operatorname{ed}^w(X,Y)$ and a corresponding w-optimal alignment in time $\mathcal{O}(k^2 \log^5 n \cdot \min\{W, \sqrt{k} \log n\})$. For that, we use the two settings of the hierarchical alignment data structure of Proposition 5.11 and pick the one that runs faster. We first design an algorithm that works if X has small self edit distance, and then use it to design an algorithm for the general case.

7.1 The Case of Small Self Edit Distance

If the input strings are highly compressible, in the sense that their self edit distance is small, we use the following fact from [CKW23] to decompose them into a small number of different pieces.

■ Lemma 7.1 ([CKW23, Lemma 4.6 and Claim 4.11]). There is a PILLAR algorithm that, given a positive integer k and strings $X, Y \in \Sigma^*$ satisfying self-ed $(X) \le k \le |X|$ and ed $(X, Y) \le k$, in $\mathcal{O}(k^2)$ time builds a decomposition $X = \bigcirc_{i=0}^{m-1} X_i$ and a sequence of fragments $(Y_i)_{i=0}^{m-1}$ of Y such that For $i \in [0..m)$, each phrase $X_i = X[x_i..x_{i+1})$ is of length $x_{i+1} - x_i \in [k..2k)$ and each fragment $Y_i = Y[y_i..y'_{i+1})$ satisfies $y_i = \max\{x_i - k, 0\}$ and $y'_{i+1} = \min\{x_{i+1} + 3k, |Y|\}$.

■ There is a set $F \subseteq [0..m)$ of size $|F| = \mathcal{O}(k)$ such that $X[x_i..x_{i+1}) = X[x_{i-1}..x_i)$ and $Y[y_i..y'_{i+1}) = Y[y_{i-1}..y'_i)$ holds for each $i \in [0..m) \setminus F$ (in particular, $0 \in F$). The algorithm returns the set F and, for $i \in F$, the endpoints of $X_i = X[x_i..x_{i+1})$. 16

Proof. Since [CKW23, Claim 4.11] is not a stand-alone statement, we provide some details for completeness. A black-box application of [CKW23, Lemma 4.6] yields a decomposition $X = \bigoplus_{i=0}^{m-1} X_i$ into phrases of length $|X_i| \in [k..2k)$ and a set $F \subseteq [0..m)$ of size at most 2k such that $X_i = X_{i-1}$ holds for each $i \in [0..m) \setminus F$. The goal of [CKW23, Claim 4.11] is to add to F every index $i \in [0..m)$ such that $Y_{i-1} \neq Y_i$. For this, the algorithm constructs the breakpoint representation of an optimal (unweighted) alignment $A: X \leadsto Y$. As argued in the proof of [CKW23, Claim 4.11], the equality $Y_{i-1} = Y_i$ holds for each $i \in [7..m-8]$ such that $[i-6..i+7] \cap F = \emptyset$ and A does not make any edit within $X_{i-6}X_{i-5} \cdots X_{i+7}$. The number of indices violating at least one of these conditions does not exceed $14(1+|F|+k) \leq 14 \cdot 4k = 56k$

The application of [CKW23, Lemma 4.6] and the construction of the breakpoint representation of \mathcal{A} take $\mathcal{O}(k^2)$ time in the PILLAR model. The final step of extending F is implemented in $\mathcal{O}(k)$ time using a left-to-right scan of F and the breakpoint representation of \mathcal{A} to determine, for each edit in \mathcal{A} , the index i of the affected phrase X_i .

We now use the decomposition of Lemma 7.1 to build a hierarchical alignment data structure that provides access to boundary matrices for all pairs of corresponding fragments of X and Y from the decomposition.

- **Lemma 7.2.** Consider a weight function $w: \overline{\Sigma}^2 \to [0..W]$, a positive integer k, and two strings $X,Y \in \Sigma^{\leq n}$ such that $\text{self-ed}(X) \leq k \leq |X|$ and $\text{ed}(X,Y) \leq k$. Moreover, suppose that $X = \bigcup_{i=0}^{m-1} X_i$, $(Y_i)_{i=0}^{m-1}$, and $F \subseteq [0..m)$ satisfy the conditions in the statement of Lemma 7.1. There is an algorithm that, given oracle access to w, the integer k, the strings X,Y, the set F, and the endpoints of X_i for each $i \in F$, takes $\mathcal{O}(W \cdot k^2 \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations and constructs:
- = a weight-balanced SLP \mathcal{G} ,
- for each $i \in F$, a pair of symbols $(A_i, B_i) \in \mathcal{S}_{\mathcal{G}}^2$ such that $\exp(A_i) = X_i$ and $\exp(B_i) = Y_i$, and
- a hierarchical alignment data structure D representing a closed set Q with $\{(A_i, B_i) : i \in F\} \subseteq Q \subset S_c^2$.

A variant of this algorithm that, instead of D, builds a relaxed hierarchical alignment data structure D^k takes $\mathcal{O}(k^{2.5}\log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations.

Proof. We use Fact 6.3 to find the breakpoint representations of optimal self-alignments $\mathcal{A}: X \rightsquigarrow X$ and $\mathcal{B}: Y \rightsquigarrow Y$. We can assume that neither alignment contains points below the main diagonal because the parts below the main diagonal can be mirrored along the main diagonal without changing the cost of the alignment. Note that $\operatorname{ed}_{\mathcal{A}}(X,X) \leq k$ holds by our assumption and $\operatorname{ed}_{\mathcal{B}}(Y,Y) \leq \operatorname{self-ed}(X) + 2\operatorname{ed}(X,Y) \leq 3k$ follows from Lemma 6.2. This initial phase of the algorithm takes $\mathcal{O}(k^2)$ time and PILLAR operations.

We process indices $i \in F$ from left to right. We construct A_0 and B_0 by constructing perfectly balanced parse trees on top of X_0 and Y_0 , respectively, and using a fresh non-terminal for every internal node. Then, we insert (A_0, B_0) to Q using the interface of D or D^k . Since $|X_0| \le 2k$ and $|Y_0| \le |X_0| + 4k \le 6k$, the construction of (A_0, B_0) takes $\mathcal{O}(k)$ time and the parse trees $\mathcal{T}(A_0)$ and $\mathcal{T}(B_0)$ have $\mathcal{O}(k)$ distinct symbols. The cost of inserting (A_0, B_0) is, according to Proposition 5.11,

¹⁶ This determines the whole decomposition because $(x_i)_{i=\ell}^r$ is an arithmetic progression for every $(\ell..r] \subseteq (0..m] \setminus F$.

bounded in terms of $L = \sum_{(C,D) \in \mathsf{cl}(A_0,B_0)} (\mathsf{len}(C) + \mathsf{len}(D))$. For analysis, consider a terminal \$ that does not occur in X_0 so that $\mathsf{cl}(A_0,B_0)$ is disjoint with $\mathsf{cl}(\$,B_0)$. Now, Corollary 5.13 yields

$$L = \mathcal{O}\left((k+1)(\operatorname{len}(A_0) + \operatorname{len}(B_0)) + \operatorname{len}(B_0)\log\left(1 + \frac{\operatorname{len}(B_0)}{\operatorname{len}(A_0)}\right)\right) = \mathcal{O}(k^2 + k\log k) = \mathcal{O}(k^2).$$

Thus, the cost of constructing (A_0, B_0) and inserting it to Q is $\mathcal{O}(W \cdot k^2 \log^3 n)$ in case of D and $\mathcal{O}(k^{2.5} \log^3 n)$ in case of D^k .

For subsequent positions $i \in F \setminus \{0\}$, we assume that we already have symbols (A_j, B_j) for $j = \max(F \cap [0..i))$. Since $X_j = X_{j+1} = \cdots = X_{i-1}$ and $Y_j = Y_{j+1} = \cdots = Y_{i-1}$, we have $\exp(A_i) = X_{i-1}$ and $\exp(B_i) = Y_{i-1}$. Our next goal is to build A_i . For this, we first restrict the alignment \mathcal{A} to $\mathcal{A}_i: X_i \rightsquigarrow \mathcal{A}(X_i)$. The breakpoint representation of \mathcal{A}_i lets us decompose X_i into individual characters that A_i deletes or substitutes and fragments that A_i matches perfectly. We process them one by one maintaining a symbol $A \in \mathcal{S}_{\mathcal{G}}$ such that $\exp(A) = X[x_{i-1}..x)$, where $X[x_i, x]$ is the already processed prefix of X_i . Initially, $x = x_i$ and $A = A_i$. If X[x] is a deleted or substituted character, we use the Merge operation of Theorem 3.9 and set A := Merge(A, X[x]). The more interesting case is when X[x..x'] is matched perfectly to some X[x-d..x'-d]. Note that d>0 because A is a self-alignment that does not contain points below the main diagonal and $x-d \ge x$ - self-ed $(X) \ge x_i$ - self-ed $(X) \ge x_i - k \ge x_{i-1}$ due to Lemma 5.3. In this case, we use Substring $(A, x - d - x_{i-1}, x - x_{i-1})$ operation of Theorem 3.9 to retrieve a symbol A' with $\exp(A') = X[x-d..x)$, and then we set A := Merge(A, Power(A', x'-x)). Since X[x-d..x')has period d, the Power(A', x' - x) operation of Theorem 3.9 constructs a symbol with expansion $\exp(A')^{\infty}[0..x'-x)=X[x-d..x'-d)=X[x..x']$. Having processed the whole X_i , that is, when $x = x_{i+1}$, we use Substring $(A, x_i - x_{i-1}, x_{i+1} - x_{i-1})$ to retrieve A_i .

The process of building B_i is very similar, with just minor differences stemming from the fact that Y_i overlaps Y_{i-1} . First, we restrict the alignment \mathcal{B} to $\mathcal{B}_i: Y_i' \leadsto \mathcal{B}(Y_i')$, where $Y_i' = Y[y_i'...y_{i+1}']$. The breakpoint representation of \mathcal{B}_i lets us decompose Y_i' into individual characters that \mathcal{B}_i deletes or substitutes and fragments that \mathcal{B}_i matches perfectly. We process them one by one maintaining a symbol $B \in \mathcal{S}_{\mathcal{G}}$ such that $\exp(B) = Y[y_{i-1}...y)$, where $Y[y_i'...y)$ is the already processed prefix of Y_i' . Initially, $y = y_i'$ and $B = B_j$. If Y[y] is a deleted or substituted character, we set B := Merge(B, Y[y]). If Y[y...y') is matched perfectly to Y[y - d...y' - d), then d > 0 because \mathcal{B} is a self-alignment that does not contain points below the main diagonal and $y - d \ge y_i' - \text{self-ed}(Y) \ge y_{i-1}' - 3k \ge y_{i-1}$; here, $y_{i-1}' - 3k \ge y_{i-1}$ follows from $y_{i-1}' \ne |Y|$, which we can assume because $y' > y \ge y_i' \ge y_{i-1}'$. In this case, we use $\text{Substring}(B, y - d - y_{i-1}, y - y_{i-1})$ to retrieve a symbol B' with $\exp(B') = B[y - d...y)$ and set B := Merge(B, Power(B', y' - y)). Having processed the whole Y_i , that is, when $y = y_{i+1}'$, we use $\text{Substring}(B, y_i - y_{i-1}, y_{i+1}' - y_{i-1})$ to retrieve Y_i . Finally, we insert (A_i, B_i) to Q using the interface of D or D^k .

The cost of processing $i \in F$ can be expressed in terms of the number k_i of edits in \mathcal{A}_i and \mathcal{B}_i . The breakpoint representations of these alignments can be retrieved in $\mathcal{O}(k)$ time from the breakpoint representations of \mathcal{A} and \mathcal{B} . The decomposition of X_i induced by \mathcal{A}_i has at most $2k_i + 1$ phrases. For each of these phrases, we perform $\mathcal{O}(1)$ weight-balanced SLP operations (Merge, Substring, and Power) on symbols whose expansions are substrings of the string $X_{i-1}X_i$ of length at most 4k. Each of these operations takes $\mathcal{O}(\log k)$ time and introduces $\mathcal{O}(\log k)$ symbols that were not present in the parse trees of its arguments. Except for A_j , all these arguments are terminals or symbols returned by previous operations performed while constructing A_i from A_j . Consequently, the whole process takes $\mathcal{O}((k_i + 1)\log k)$ time and $\mathcal{T}(A_i)$ contains $\mathcal{O}((k_i + 1)\log k)$ symbols that do not occur in $\mathcal{T}(A_j)$. Similarly, the decomposition of Y_i' induced by \mathcal{B}_i has at most $2k_i + 1$ phrases. For each of these phrases, we perform $\mathcal{O}(1)$ weight-balanced SLP operations on symbols whose expansions are

substrings of the string $Y_{i-1}Y_i'$ of length at most 8k. Each of these operations takes $\mathcal{O}(\log k)$ time and introduces $\mathcal{O}(\log k)$ symbols that were not present in the parse trees of its arguments. Except for B_j , all these arguments are terminals or symbols returned by previous operations performed while constructing B_i from B_j . Consequently, the whole process takes $\mathcal{O}((k_i+1)\log k)$ time and $\mathcal{T}(B_i)$ contains $\mathcal{O}((k_i+1)\log k)$ symbols that do not occur in $\mathcal{T}(B_j)$. The cost of inserting (A_i, B_i) to Q is, according to Proposition 5.11, bounded in terms of $L = \sum_{(C,D) \in \mathsf{cl}(A_i,B_i) \setminus Q} (\mathsf{len}(C) + \mathsf{len}(D))$. Prior to this operation, we already have $\mathsf{cl}(A_j, B_j) \in Q$, so

$$L \leq \sum_{(C,D) \in \mathsf{cl}(A_i,B_j) \backslash \mathsf{cl}(A_j,B_j)} (\mathsf{len}(C) + \mathsf{len}(D)) + \sum_{(C,D) \in \mathsf{cl}(A_i,B_i) \backslash \mathsf{cl}(A_i,B_j)} (\mathsf{len}(C) + \mathsf{len}(D)).$$

By Corollary 5.13, the two terms on the right-hand side can be bounded by

$$\mathcal{O}\left((k_i+1)\log k(\mathsf{len}(A_i)+\mathsf{len}(B_j))+\mathsf{len}(B_j)\log\left(1+\frac{\mathsf{len}(B_j)}{\mathsf{len}(A_i)}\right)\right)=\mathcal{O}((k_i+1)k\log k+k\log k),$$

and

$$\mathcal{O}\left((k_i+1)\log k(\operatorname{len}(A_i)+\operatorname{len}(B_i))+\operatorname{len}(A_i)\log\left(1+\frac{\operatorname{len}(A_i)}{\operatorname{len}(B_i)}\right)\right)=\mathcal{O}((k_i+1)k\log k+k\log k),$$

respectively. Hence, $L = \mathcal{O}((k_i + 1)k \log k)$, and the cost of constructing (A_i, B_i) and inserting it to Q is $\mathcal{O}(W \cdot (k_i + 1)k \log^4 n)$ in case of D and $\mathcal{O}((k_i + 1)k^{1.5} \log^4 n)$ in case of D^k .

To analyze the total time complexity, observe that

$$\sum_{i \in F} (k_i + 1) \leq \sum_{i \in F} k_i + |F| \leq \mathsf{self-ed}(X) + \mathsf{self-ed}(Y) + |F| = \mathcal{O}(k).$$

In case of D, we pay $\mathcal{O}(k^2)$ time plus $\mathcal{O}(k^2)$ PILLAR operations for preprocessing, $\mathcal{O}(W \cdot k^2 \log^3 n)$ time for i = 0, and $\mathcal{O}(W \cdot (1 + k_i)k \log^4 n)$ time for each $i \in F \setminus \{0\}$, for a total of $\mathcal{O}(W \cdot k^2 \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations. In case of D^k , we pay $\mathcal{O}(k^2)$ time plus $\mathcal{O}(k^2)$ PILLAR operations for preprocessing, $\mathcal{O}(k^{2.5} \log^3 n)$ time for i = 0, and $\mathcal{O}((1 + k_i)k^{1.5} \log^4 n)$ time for each $i \in F \setminus \{0\}$, for a total of $\mathcal{O}(k^{2.5} \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations.

We now use the hierarchical alignment data structure constructed by Lemma 7.2 to find a w-optimal alignment for the complete strings X and Y.

Lemma 7.3. There is an algorithm that, given oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, a positive integer k, and strings $X,Y \in \Sigma^{\leq n}$ such that self-ed(X) ≤ k and ed^w(X,Y) ≤ k, constructs the breakpoint representation of a w-optimal alignment of X onto Y and takes $\mathcal{O}(\min\{k^2W\log^4n,k^{2.5}\log^4n\})$ time plus $\mathcal{O}(k^2)$ PILLAR operations.

Proof. If |X| < k, we solve the problem using Fact 3.3. This algorithm takes $\mathcal{O}((|X|+1)k)$ time, which is $\mathcal{O}(k^2)$ since $|X| \le k$. Thus, we henceforth assume that $|X| \ge k$ and, in particular, $n \ge k$.

We first describe an algorithm that works in time $\mathcal{O}(k^2W\log^4 n)$, and later transform it into an algorithm that works in time $\mathcal{O}(k^{2.5}\log^4 n)$. By running these two algorithms in parallel and returning the answer of whichever one of them terminates first, we get the desired time complexity.

We follow a setup similar to the proof of [CKW23, Lemma 4.9] in PILLAR model for d := k. First, we run the algorithm of Lemma 7.1 for X, Y, and k, arriving at a decomposition $X = \bigcirc_{i=0}^{m-1} X_i$ and a sequence of fragments $(Y_i)_{i=0}^{m-1}$, such that $X_i = X[x_i..x_{i+1})$ and $Y_i = Y[y_i..y'_{i+1})$ for $y_i = \max\{x_i - k, 0\}$ and $y'_{i+1} = \min\{x_{i+1} + 3k, |Y|\}$ for all $i \in [0..m)$. The decomposition is represented using a set F of size $\mathcal{O}(k)$ such that $X_i = X_{i-1}$ and $Y_i = Y_{i-1}$ holds for each $i \in [0..m) \setminus F$ and the

endpoints of X_i for $i \in F$. To easily handle corner cases, we set $y'_0 = 0$ and $y_m = |Y|$, and we assume that $[0..m) \setminus F \subseteq [2..m-5]$; if the original set F does not satisfy this condition, we add the missing $\mathcal{O}(1)$ elements.

Having constructed (and possibly extended) F, we apply Lemma 7.2. It yields a weight-balanced SLP \mathcal{G} and, for each $i \in F$, a pair of symbols $(A_i, B_i) \in F$ such that $\exp(A_i) = X_i$ and $\exp(B_i) = Y_i$. Additionally, the underlying algorithm constructs a hierarchical alignment data structure D maintaining a closed set Q that satisfies $\{(A_i, B_i) : i \in F\} \subseteq Q \subseteq \mathcal{S}_{\mathcal{G}}^2$.

For each $i \in [0..m)$, consider a subgraph G_i of $\overline{\mathrm{AG}}^w(X,Y)$ induced by $[x_i..x_{i+1}] \times [y_i..y'_{i+1}]$. Denote by G the union of all subgraphs G_i . Note that G contains all vertices $(x,y) \in [0..|X|] \times [0..|Y|]$ with $|x-y| \leq k$. Therefore, as we know that $\operatorname{ed}^w(X,Y) \leq k$, Lemma 5.3 implies that the w-optimal alignment of X onto Y can go only through vertices (x,y) of $\overline{\mathrm{AG}}^w(X,Y)$ with $|x-y| \leq k$, and thus the distance from (0,0) to (|X|,|Y|) in $\overline{\mathrm{AG}}^w(X,Y)$ is the same as the distance from (0,0) to (|X|,|Y|) in G. For each $i \in [0..m]$, denote $V_i \coloneqq \{(x,y) \in V(G) \mid x=x_i,y \in [y_i..y'_i]\}$. Note that $V_0 = \{(0,0)\}$, $V_m = \{(|X|,|Y|)\}$, and $V_i = V(G_i) \cap V(G_{i-1})$ for $i \in (0..m)$. Moreover, for $i,j \in [0..m]$ with $i \leq j$, let $D_{i,j}$ denote the matrix of pairwise distances from V_i to V_j in G, where rows of $D_{i,j}$ represent vertices of V_i in the decreasing order of the second coordinate, and columns of $D_{i,j}$ represent vertices of V_i in the decreasing order of the second coordinate. By Lemma 5.2 the shortest paths between vertices of G_i stay within G_i , so $D_{i,i+1}$ is a contiguous submatrix of $\mathrm{BM}^w(X_i,Y_i)=\mathrm{BM}^w(\exp(A_i),\exp(B_i))$. Hence, for each $i \in F$, we can retrieve $\mathrm{CMO}(D_{i,i+1})$ using Lemma 4.7 and matrix retrieval operation of D for (A_i,B_i) .

 Γ Claim 7.4. Let $0 = j_0 < j_1 < \ldots < j_{|F|-1} < j_{|F|} = m$ be the elements of $F \cup \{m\}$. We have

$$D_{0,m} = \bigotimes_{i=0}^{|F|-1} D_{j_i,j_i+1}^{\otimes (j_{i+1}-j_i)}.$$

Proof. Note that, for every $a, i, b \in [0..m]$ with $a \le i \le b$, the set V_i is a separator in G between all vertices of V_a and V_b . Thus, every path from V_a to V_b in G crosses V_i , so $D_{a,b} = D_{a,i} \otimes D_{i,b}$. In general, this implies $D_{0,m} = \bigotimes_{i=0}^{m-1} D_{i,i+1}$.

It remains to prove that, for every $i \in [0..|F|)$, we have $D_{j_i,j_i+1} = D_{j_i+1,j_i+2} = \cdots = D_{j_{i+1}-1,j_{i+1}}$. If $j_{i+1} = j_i + 1$, this statement is obvious. Otherwise, the condition on F stipulated in Lemma 7.1, implies that the graphs $G_{j_i}, G_{j_{i+1}}, \ldots, G_{j_{i+1}-1}$ are isomorphic to $\overline{AG}^w(X_{j_i}, Y_{j_i})$. Moreover, since we assumed that $[0..m) \setminus F \subseteq [2..m-5]$, we have $1 \leq j_i < j_{i+1} \leq m-4$. Since each phrase is of length at least k and $|Y| \geq |X| - \operatorname{ed}^w(X,Y) \geq |X| - k$, for each $p \in [1..m-4]$, we have $x_p \geq x_{p-1} + k \geq k$ and $x_p \leq x_{p+4} - 4k \leq |X| - 4k \leq |Y| - 3k$. In particular, $[y_p, y_p'] = [x_p - k..x_p + 3k]$. Hence, for each $p \in (j_i..j_{i+1})$, not only G_p is isomorphic to G_{j_i} , but also V_p and V_{p+1} in this isomorphism correspond to V_{j_i} and $V_{j_{i+1}}$. Hence, we indeed have $D_{j_i,j_{i+1}} = D_{j_{i+1},j_{i+2}} = \cdots = D_{j_{i+1}-1,j_{i+1}}$. Therefore, $D_{j_i,j_{i+1}} = D_{j_i,j_{i+1}}^{\otimes (j_{i+1}-j_i)}$. We thus obtain $D_{0,m} = \bigotimes_{i=0}^{|F|-1} D_{j_i,j_{i+1}}^{\otimes (j_{i+1}-j_i)}$.

Note that for each $i \in [0..|F|)$, we already computed $\mathsf{CMO}(D_{j_i,j_i+1})$. Given $\mathsf{CMO}(D_{a,b})$ and $\mathsf{CMO}(D_{b,c})$ for any $a,b,c \in [0..m]$ with a < b < c, we can calculate $\mathsf{CMO}(D_{a,c})$ using Lemma 4.8 in time $\mathcal{O}(k\log^2 n + \delta(D_{a,b})\log^3 n + \delta(D_{a,c})\log^3 n) = \mathcal{O}(k\log^2 n + (W \cdot k)\log^3 n + (W \cdot k)\log^3 n) = \mathcal{O}(W \cdot k\log^3 n)$, where $\delta(D_{a,b}), \delta(D_{a,c}) = \mathcal{O}(W \cdot k)$ due to Lemma 5.7.

Therefore, using the idea of binary exponentiation, we can calculate $\mathsf{CMO}(M_i)$ where $M_i \coloneqq D_{j_i,j_i+1}^{\otimes (j_{i+1}-j_i)}$ in time $\mathcal{O}(W \cdot k \log^4 n)$. Doing it for all $i \in [0..|F|)$ requires time $\mathcal{O}(W \cdot k^2 \log^4 n)$. We then calculate $\mathsf{CMO}(D_{0,m})$ using the fact that $D_{0,m} = M_0 \otimes M_1 \otimes \cdots \otimes M_{|F|-1}$ in time $\mathcal{O}(|F| \cdot (W \cdot k \log^3 n)) = \mathcal{O}(W \cdot k^2 \log^3 n)$, where we calculate $M_0 \otimes M_1 \otimes \cdots \otimes M_{|F|-1}$ in a perfectly balanced binary tree manner.

It remains to reconstruct a w-optimal alignment $X \rightsquigarrow Y$; its cost $\operatorname{ed}^w(X,Y)$ is the only entry of $D_{0,m}$. For that, we additionally save all the CMO data structures that were computed throughout the time of the algorithm. We obtained $\operatorname{CMO}(D_{0,m})$ by a logarithmic-depth min-plus matrix multiplication sequence from $\operatorname{CMO}(D_{j_i,j_i+1})$ s. When two matrices are multiplied, a specific entry of the answer is equal to the minimum of the sum of $\mathcal{O}(k)$ pairs of entries from the input matrices as all matrices we work with have sizes $\mathcal{O}(k) \times \mathcal{O}(k)$. Hence, in time $\mathcal{O}(k \log n)$ we can reconstruct the entries of the input matrices, from which this value arose by accessing CMO data structures that we saved for all of them.

We use recursive backtracking similar to the one used in the alignment retrieval operation of Proposition 5.11. Whenever the current value we are trying to backtrack is zero, we terminate as there is a single path between any two vertices of $\overline{AG}^w(X,Y)$ that has weight zero. In each internal node of this process, we spend at most $\mathcal{O}(k \log n)$ time to backtrack the value to its child nodes. For each leaf, we apply alignment retrieval of Proposition 5.11 to reconstruct the w-optimal alignment inside G_i at the cost of at most $\mathcal{O}(k \log^2 n)$. As $\operatorname{ed}^w(X,Y) \leq k$ and the process has logarithmic depth, there are $\mathcal{O}(k \log n)$ internal nodes and $\mathcal{O}(k)$ leaves of this process with non-zero values. Therefore, the whole procedure takes time $\mathcal{O}(k \log n \cdot k \log n + k \cdot k \log^2 n) = \mathcal{O}(k^2 \log^2 n)$.

It remains to analyze the time complexity of the complete algorithm. We spend $\mathcal{O}(k^2)$ time and $\mathcal{O}(k^2)$ PILLAR operations for Lemma 7.1. The application of Lemma 7.2 takes time $\mathcal{O}(k^2 \cdot W \log^4 n)$ time and $\mathcal{O}(k^2)$ PILLAR operations. Computation of $D_{i,i+1}$ for $i \in F$ takes time $\mathcal{O}(|F| \cdot (k \log n + k \cdot W \log n)) = \mathcal{O}(k^2 \cdot W \log n)$ due to Lemma 5.7. Computation of $D_{0,m}$ then takes $\mathcal{O}(W \cdot k^2 \log^4 n)$ time. Reconstructing the alignment takes time $\mathcal{O}(k^2 \log^2 n)$. Thus, the whole algorithm works in $\mathcal{O}(k^2 \cdot W \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations.

We now show how to transform the presented algorithm into the one that takes $\mathcal{O}(k^{2.5}\log^4 n)$ time. The only thing we change is that we now use Lemma 4.14 instead of Lemma 4.8 whenever we multiply CMO and use the variant of the algorithm from Lemma 7.2 that builds a relaxed hierarchical alignment data structure D^k when we apply it. As basic operations over matrices preserve k-equivalence due to Observation 4.5, at the end we get some value that is k-equivalent to $\mathsf{ed}^w(X,Y)$. As we know that $\mathsf{ed}^w(X,Y) \leq k$, we find its exact value. The alignment reconstruction process works the same way as all values we backtrack to also have values at most k, and thus are computed correctly.

The application of Lemma 7.1 still takes $\mathcal{O}(k^2)$ time and $\mathcal{O}(k^2)$ PILLAR operations. Lemma 7.2 now takes $\mathcal{O}(k^{2.5}\log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations. Finally, we have $\mathcal{O}(k\log n)$ min-plus matrix multiplications in Claim 7.4, and thus it takes time $\mathcal{O}(k^{2.5}\log^4 n)$ to perform all of them using Lemma 4.14 as all the matrices we work with have cores of size $\mathcal{O}(k\sqrt{k})$. At the end, we reconstruct the alignment in time $\mathcal{O}(k^2\log^2 n)$. Hence, the entire algorithm works in $\mathcal{O}(k^{2.5}\log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations.

7.2 The General Case

In the general case, we first design a procedure that "improves" an alignment. That is, given some alignment $\mathcal{A}: X \leadsto Y$ (some approximation of a w-optimal alignment), finds a w-optimal alignment $\mathcal{B}: X \leadsto Y$. For that, we use divide-and-conquer combined with the combinatorial fact of Proposition 6.7: we recursively compute w-optimal alignments for two halves, and then use them and Lemma 7.3 to obtain a w-optimal alignment for the whole string.

Lemma 7.5. There is an algorithm that, given two strings $X, Y \in \Sigma^{\leq n}$, oracle access to a weight function $w : \overline{\Sigma}^2 \to [0..W]$, and the breakpoint representation of an alignment $A : X \rightsquigarrow Y$, finds the

```
1 ImproveAlignment (X, Y, w, A) begin
         k \leftarrow \operatorname{ed}_{\mathcal{A}}^{w}(X,Y);
         if k = 0 then
 3
              return A;
 4
         if k > |X| then
              return the breakpoint representation of a w-optimal alignment computed using
 6
                Fact 3.3:
         Denote (x_i, y_i)_{i=0}^t := \mathcal{A};
 7
         Pick some m such that x_m = \lfloor |X|/2 \rfloor;
 8
         \ell \leftarrow \min\{i \in [0..m] \mid \mathsf{self-ed}(X[x_i..x_m)) \leq 5k\};
 9
         r \leftarrow \max\{i \in [m..t] \mid \mathsf{self-ed}(X[x_m..x_i)) \le 5k\};
10
         \mathcal{A}_L \leftarrow (x_i, y_i)_{i=0}^m;
11
         \mathcal{A}_R \leftarrow (x_i, y_i)_{i=m}^t;
12
         \mathcal{B}_L \leftarrow \texttt{ImproveAlignment}(X[0..x_m), Y[0..y_m), w, \mathcal{A}_L);
13
         \mathcal{B}_R \leftarrow \texttt{ImproveAlignment}(X[x_m..|X|),Y[y_m..|Y|),w,\mathcal{A}_R);
14
15
         Compute breakpoint representation of a w-optimal alignment
          \mathcal{B}_M: X[x_\ell..x_r) \rightsquigarrow Y[y_\ell..y_r) using Lemma 7.3 for threshold 10k;
         Compute the leftmost intersection points (x_L^*, y_L^*) \in \mathcal{B}_L \cap \mathcal{B}_M and (x_R^*, y_R^*) \in \mathcal{B}_R \cap \mathcal{B}_M;
16
         return the breakpoint representation of an alignment \mathcal{B}: X \leadsto Y that follows \mathcal{B}_L from
17
          (0,0) to (x_L^*,y_L^*), follows \mathcal{B}_M from (x_L^*,y_L^*) to (x_R^*,y_R^*), and follows \mathcal{B}_R from (x_R^*,y_R^*) to
          (|X|, |Y|);
```

Algorithm 1 The algorithm from Lemma 7.5. Given strings X and Y, oracle access to a weight function w, and the breakpoint representation of an alignment A of X onto Y, the algorithm computes the breakpoint representation of a w-optimal alignment of X onto Y.

breakpoint representation of a w-optimal alignment $\mathcal{B}: X \leadsto Y$. The running time of the algorithm is $\mathcal{O}(\min\{k^2 \cdot W \log^5 n, k^{2.5} \log^5 n\})$ time plus $\mathcal{O}(k^2 \log^2 n)$ PILLAR operations, where $k = \operatorname{ed}_A^w(X, Y)$.

Proof. We develop a recursive procedure ImproveAlignment that satisfies the requirements of Lemma 7.5; see Algorithm 1.

We first find $k := ed_A^w(X,Y)$. Then, we handle some corner cases: if k = 0, we return A, and if k > |X|, we find an w-optimal alignment using Fact 3.3. If none of the corner cases are applicable, we proceed as follows. We denote $(x_i, y_i)_{i=0}^t := \mathcal{A}$ and find three indices $\ell, m, r \in [0, t]$ such that $x_m = |X|/2$, ℓ is the smallest possible such that $self-ed(X[x_\ell..x_m]) \le 5k$, and r is the largest possible such that self-ed $(X[x_m..x_r)) \leq 5k$. We recursively compute the breakpoint representation of a w-optimal alignment $\mathcal{B}_L: X[0..x_m) \rightsquigarrow Y[0..y_m)$ by improving upon $\mathcal{A}_L := (x_i, y_i)_{i=0}^m$. Analogously, we recursively compute the breakpoint representation of a w-optimal alignment $\mathcal{B}_R: X[x_m..|X|), Y[y_m..|Y|)$ by improving upon $\mathcal{A}_R := (x_i, y_i)_{i=m}^t$. We then use Lemma 7.3 to find the breakpoint representation of a w-optimal alignment $\mathcal{B}_M: X[x_\ell..x_r) \rightsquigarrow Y[y_\ell..y_r)$. We have $\operatorname{self-ed}(X[x_{\ell}..x_r)) \leq \operatorname{self-ed}(X[x_{\ell}..x_m)) + \operatorname{self-ed}(X[x_m..x_r)) \leq 5k + 5k = 10k$ and $\operatorname{ed}^w(X[x_\ell..x_r),Y[y_\ell..y_r)) \leq \operatorname{ed}^w_{\mathcal{A}}(X[x_\ell..x_r),Y[y_\ell..y_r)) \leq \operatorname{ed}^w_{\mathcal{A}}(X,Y) \leq k$, so it suffices to apply Lemma 7.3 with threshold 10k. We then compute the w-optimal alignment \mathcal{B} by combining $\mathcal{B}_L, \mathcal{B}_M$, and \mathcal{B}_R . We use two pointers to find the leftmost intersection point of \mathcal{B}_L and \mathcal{B}_M and the leftmost intersection point of \mathcal{B}_M and \mathcal{B}_R . We then create \mathcal{B} by first going along \mathcal{B}_L until the first intersection point with \mathcal{B}_M , then switching to \mathcal{B}_M and going along it until the first intersection point with \mathcal{B}_R , and finally switching to \mathcal{B}_R and going along it from there.

Let us prove the correctness of the algorithm. If k=0, then \mathcal{A} is already a w-optimal alignment as no alignment of negative weight can exist. If k>|X|, we use Fact 3.3, which returns a w-optimal alignment. Otherwise, the minimality of ℓ implies that $\ell=0$ or self-ed $(X[x_\ell..x_m))=5k>4\mathrm{ed}_{\mathcal{A}}^w(X,Y)$, whereas the maximality of r implies that r=t or self-ed $(X[x_m..x_r))=5k>4\mathrm{ed}_{\mathcal{A}}^w(X,Y)$, where equalities are due to the property of continuity of self-ed (see Lemma 6.2). Consequently, we can use Proposition 6.7 for the alignments $\mathcal{A}, \mathcal{B}_L, \mathcal{B}_M$, and \mathcal{B}_R to conclude that $\mathcal{B}_L \cap \mathcal{B}_M \neq \emptyset \neq \mathcal{B}_R \cap \mathcal{B}_M$, and all intersection points $(x_L^*, y_L^*) \in \mathcal{B}_L \cap \mathcal{B}_M$ and $(x_R^*, y_R^*) \in \mathcal{B}_R \cap \mathcal{B}_M$ satisfy

$$\operatorname{ed}^w(X,Y) = \operatorname{ed}^w_{\mathcal{B}_L}(X[0..x_L^*),Y[0..y_L^*)) + \operatorname{ed}^w_{\mathcal{B}_M}(X[x_L^*..x_R^*),Y[y_L^*..y_R^*)) + \operatorname{ed}^w_{\mathcal{B}_R}(X[x_R^*..|X|),Y[y_R^*..|Y|)).$$

Consequently, the constructed alignment \mathcal{B} is indeed w-optimal.

We now analyze the running time. First focus on a single execution of Algorithm 1, ignoring the time spent in the recursive calls. Denote n := |X| + |Y| and $k := \operatorname{ed}_{\mathcal{A}}^w(X,Y)$. Retrieving k from the breakpoint representation of \mathcal{A} takes $\mathcal{O}(k+1)$ time. If k=0, we then return \mathcal{A} in constant time. If k > |X|, we run Fact 3.3 that works in time $\mathcal{O}((|X|+1)k) = \mathcal{O}(k^2)$ since |X| < k. We henceforth assume that we are not in the corner cases. It takes $\mathcal{O}(k)$ time to find m. It then takes $\mathcal{O}(k^2 \log n)$ time and PILLAR operations to find ℓ and ℓ using Fact 6.3 and binary search. Constructing \mathcal{B}_M using Lemma 7.3 takes time $\mathcal{O}(\min\{k^2 \cdot W \log^4 n, k^{2.5} \log^4 n\})$ time and $\mathcal{O}(k^2)$ PILLAR operations. Combining the three ℓ -optimal alignments takes time ℓ -optimal does not exceed the cost of ℓ -optimal alignments takes time ℓ -optimal 1 takes time ℓ -optimal 2 takes time ℓ -optimal 3 takes time 4 takes time 3 takes time 4 takes time 5 takes 1 take

Now, let us consider the overall running time of the algorithm, taking into account the recursive calls. We recurse into two subproblems $(X_L, Y_L, w, \mathcal{A}_L)$ and $(X_R, Y_R, w, \mathcal{A}_R)$ such that $|X_L| + |X_R| = |X|, |Y_L| + |Y_R| = |Y|$, and $\operatorname{ed}_{\mathcal{A}_L}^w(X_L, Y_L) + \operatorname{ed}_{\mathcal{A}_R}^w(X_R, Y_R) = \operatorname{ed}_{\mathcal{A}}^w(X, Y)$. Thus, at any level of the recursion, we have subproblems involving strings of total length n and the total cost of (input) alignments k. As argued earlier, time spent on a subproblem involving strings of length n_i and an alignment of cost k_i is $\mathcal{O}(\min\{k_i^2 \cdot W \log^4 n_i, k_i^{2.5} \log^4 n_i\})$. Therefore, we can upper bound the total time spent at any level of the recursion by

$$\mathcal{O}(\sum_{i} \min\{k_{i}^{2} \cdot W \log^{4} n_{i}, k_{i}^{2.5} \log^{4} n_{i}\}) \leq \mathcal{O}(\min\{\sum_{i} k_{i}^{2} \cdot W \log^{4} n_{i}, \sum_{i} k_{i}^{2.5} \log^{4} n_{i}\})$$

$$\leq \mathcal{O}(\min\{\sum_{i} k \cdot k_{i} \cdot W \log^{4} n_{i}, \sum_{i} k^{1.5} \cdot k_{i} \log^{4} n_{i}\}) = \mathcal{O}(\min\{k^{2} \cdot W \log^{4} n, k^{2.5} \log^{4} n\}).$$

Similarly, the total number of PILLAR operations at each level is $\mathcal{O}(k^2 \log n)$. Finally, as we split X into two halves to go to the next level of recursion, there are $\mathcal{O}(\log n)$ recursion levels, and the overall cost is $\mathcal{O}(\min\{k^2 \cdot W \log^5 n, k^{2.5} \log^5 n\})$ time plus $\mathcal{O}(k^2 \log^2 n)$ PILLAR operations.

We now use Lemma 7.5 to design our main static algorithm. It starts from an optimal unweighted alignment and gradually improves it until we arrive at a w-optimal alignment.

Theorem 7.6. There is a PILLAR algorithm that, given two strings $X,Y \in \Sigma^{\leq n}$ as well as oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, finds $\operatorname{ed}^w(X,Y)$ as well as the breakpoint representation of a w-optimal alignment $A: X \leadsto Y$. The running time of the algorithm is $\mathcal{O}(\min\{k^2 \cdot W \log^5 n, k^{2.5} \log^6 n\})$ time plus $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations, where $k = \operatorname{ed}^w(X,Y)$.

Proof. We find the breakpoint representation of a w-optimal alignment $\mathcal{A}: X \leadsto Y$. From it, in time $\mathcal{O}(k)$, we can compute $\operatorname{ed}^w(X,Y)$.

Define $w': \overline{\Sigma}^2 \to [0..W']$ as $w'(a,b) := \min\{w(a,b),n\}$ for $W' = \min\{W,n\}$. Note that $k' := \operatorname{ed}^{w'}(X,Y) \le \operatorname{ed}^{w}(X,Y) = k$. We run the remaining part of the algorithm for the weight

function w' instead of w. At the end, we get some w'-optimal alignment $A: X \rightsquigarrow Y$ of cost k'. If $k' \geq n$, we additionally run the algorithm from Fact 3.3 for X, Y, and w and return the alignment it returns. Its running time is $\mathcal{O}(k \cdot n) \leq \mathcal{O}(k^2)$. Otherwise, if k' < n, we return \mathcal{A} . Note that in this case all edges of the alignment \mathcal{A} have weights at most k' < n, and thus $\operatorname{\sf ed}_A^w(X,Y) = \operatorname{\sf ed}_A^{w'}(X,Y) = k' \leq k = \operatorname{\sf ed}^w(X,Y)$. Hence, $\mathcal A$ is a w-optimal alignment.

Therefore, it remains to find a w'-optimal alignment A.

First, we find an optimal unweighted alignment $A^*: X \rightsquigarrow Y$ of weight $k^* \leq k'$ in time $\mathcal{O}(k'^2)$ and $\mathcal{O}(k'^2)$ PILLAR operations using Fact 3.10. Let $W^* \leq W'$ be the maximum weight w' of the edges in \mathcal{A}^* .

We now iterate from $t = t_1 := \lceil \log_2 W^* \rceil$ down to t = 0, and for each such t calculate a w'_t optimal alignment $\mathcal{A}_t: X \rightsquigarrow Y$, where $w'_t(a,b) := \lceil w'(a,b)/2^t \rceil$ for any $a,b \in \overline{\Sigma}$. Let $W'_t := \lceil W'/2^t \rceil$ be an upper bound on the values of w'_t . Note that $w'_0 = w'$, so at the end we get a w'-optimal alignment $\mathcal{A} = \mathcal{A}_0$. Furthermore, $w'_t(a,b) \leq w'(a,b)$ for any $t \in [0,t_1]$ and $a,b \in \overline{\Sigma}$, and thus $k'_t := \operatorname{\sf ed}^{w'_t}(X,Y) \le \operatorname{\sf ed}^{w'}(X,Y) = k' \le k \text{ for any } t.$

In the first iteration, we set $A_{t_1} := A^*$. Note that all edges in A^* have w'-weight at most W^* by the definition of W^* . Hence, they have w'_{t_1} -weight at most one as $2^{t_1} \geq W^*$. Therefore,

 $\operatorname{ed}_{\mathcal{A}^*}^{w'_{t_1}}(X,Y) = \operatorname{ed}_{\mathcal{A}^*}(X,Y) = \operatorname{ed}(X,Y) \leq \operatorname{ed}^{w'_{t_1}}(X,Y)$. Thus, $\mathcal{A}_{t_1} = \mathcal{A}^*$ is a w'_{t_1} -optimal alignment. Now, for every other t, given a w'_{t+1} -optimal alignment $\mathcal{A}_{t+1}: X \leadsto Y$, we apply Lemma 7.5 to find a w'_{t} -optimal alignment $\mathcal{A}_{t}: X \leadsto Y$. It takes $\mathcal{O}(\min\{\ell^2 \cdot W'_{t} \log^5 n, \ell^{2.5} \log^5 n\})$ time plus $\mathcal{O}(\ell^2 \log^2 n)$ PILLAR operations, where $\ell \coloneqq \mathsf{ed}_{\mathcal{A}_{t+1}}^{w_t'}(X,Y)$. Note that $w_t'(a,b) \leq 2 \cdot w_{t+1}'(a,b)$ for any $a, b \in \overline{\Sigma}$, so $\ell \leq 2\operatorname{ed}_{\mathcal{A}_{t+1}}^{w'_{t+1}}(X,Y) = 2k'_{t+1} \leq 2k'$. Hence, the call to Lemma 7.5 takes time $\mathcal{O}(\min\{k'^2 \cdot W'_t \log^5 n, k'^{2.5} \log^5 n\})$ and $\mathcal{O}(k'^2 \log^2 n)$ PILLAR operations.

There are $t_1 + 1 = \mathcal{O}(\log n)$ iteration of the algorithm, so in total we spend $\mathcal{O}(k'^2 \log^3 n) \le$ $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations. Furthermore, the overall time complexity is

$$\begin{split} \mathcal{O}(k'^2 + \sum_{t=0}^{t_1} \min\{k'^2 \cdot W_t' \log^5 n, k'^{2.5} \log^5 n\}) &\leq \mathcal{O}(\min\{\sum_{t=0}^{t_1} k'^2 \cdot W_t' \log^5 n, \sum_{t=0}^{t_1} k'^{2.5} \log^5 n\}) \\ &= \mathcal{O}(\min\{k'^2 \log^5 n \sum_{t=0}^{t_1} W_t', (t_1+1) \cdot k'^{2.5} \log^5 n\}) \\ &\leq \mathcal{O}(\min\{k'^2 W' \log^5 n, k'^{2.5} \log^6 n\}) \\ &\leq \mathcal{O}(\min\{k^2 W \log^5 n, k^{2.5} \log^6 n\}), \end{split}$$

where
$$\sum_{t=0}^{t_1} W_t' = \mathcal{O}(W' + t_1) = \mathcal{O}(W') \le \mathcal{O}(W)$$
 as $W_t' = \lceil W'/2^t \rceil$ for all $t \in [0..t_1]$.

Dynamic Algorithm

In this section, we maintain the weighted edit distance of two strings X and Y under edits. That is, in one update we insert, delete, or substitute a single character in X or Y. Suppose that the i-th update was applied to X. We interpret this update as a pair $(\mathcal{B}_i, \mathcal{C}_i)$ of two alignments, where $\mathcal{B}_i: X \leadsto X'$ is an alignment of the old version of X onto the new one induced by the edit, and $C_i: Y \leadsto Y'$ where Y'=Y is an identity alignment. If the *i*-th update is applied to Y, we define $(\mathcal{B}_i, \mathcal{C}_i)$ analogously. In both situations we have $\mathsf{ed}_{\mathcal{B}_i}(X, X') + \mathsf{ed}_{\mathcal{C}_i}(Y, Y') = 1$. Furthermore, given an update, we can trivially calculate the breakpoint representations of \mathcal{B}_i and \mathcal{C}_i in constant time.

We use the hierarchical alignment data structure of Proposition 5.11 to maintain information necessary to compute a w-optimal alignment under edits. As an initial step, we describe algorithms that work in time $\widetilde{\mathcal{O}}((|X|+|Y|)\cdot W)$ per update. First, we show how to update the hierarchical alignment data structure when one edit is made to one of the strings.

Lemma 8.1. Consider a weight-balanced SLP \mathcal{G} and a closed set $Q \subseteq \mathcal{S}_{\mathcal{G}}^2$ stored in a hierarchical alignment data structure D for a weight function $w : \overline{\Sigma}^2 \to [0..W]$ accessible through an $\mathcal{O}(1)$ -time oracle. There is an algorithm that, given $(A, B) \in Q$ and a single edit to $\exp(A)$ or $\exp(B)$ that does not make the corresponding string empty, extends \mathcal{G} and inserts to Q a pair (A', B') such that $\exp(A')$ and $\exp(B')$ are the edited versions of $\exp(A)$ and $\exp(B)$, respectively; the algorithm takes $\mathcal{O}(W \cdot (\operatorname{len}(A) + \operatorname{len}(B)) \log^4 n)$ time, where $n = |Q| + \operatorname{len}(A) + \operatorname{len}(B)$.

Proof. Suppose that the edit is an insertion of a character a at position i of $\exp(A)$; the remaining cases are processed analogously. Assuming that $i \in (0..\operatorname{len}(A))$, we apply the $\operatorname{Substring}(A,0,i)$ and $\operatorname{Substring}(A,i,\operatorname{len}(A))$ operations of Theorem 3.9 to obtain symbols A_L and A_R such that $\exp(A_L) = \exp(A)[0..i)$ and $\exp(A_R) = \exp(A)[i..\operatorname{len}(A))$. Then, we construct a symbol A' using two applications of the Merge operation of Theorem 3.9, that is, $A' = \operatorname{Merge}(\operatorname{Merge}(A_L,a),A_R)$. In the special cases of i = 0 and $i = \operatorname{len}(A)$, we simply set $A' = \operatorname{Merge}(a,A)$ and $A' = \operatorname{Merge}(A,a)$, respectively. In all cases, we insert (A',B) to Q using the interface of D.

Let us analyze the running time. There are constantly many applications of Substring and Merge, each of which takes $\mathcal{O}(1+\log \operatorname{len}(A))$ time and creates $\mathcal{O}(1+\log \operatorname{len}(A))$ new symbols that were not present in $\mathcal{T}(A)$. Consequently, the construction of A' takes $\mathcal{O}(\log n)$ time and $\mathcal{T}(A')$ contains $\mathcal{O}(\log n)$ symbols that were not present in $\mathcal{T}(A)$. By Proposition 5.11, inserting (A',B) to Q takes at most $\mathcal{O}((L+1)\cdot W\log^3 n)$ time, where $L=\sum_{(C,D)\in\operatorname{cl}(A',B)\setminus Q}(\operatorname{len}(C)+\operatorname{len}(D))\leq \sum_{(C,D)\in\operatorname{cl}(A',B)\setminus\operatorname{cl}(A,B)}(\operatorname{len}(C)+\operatorname{len}(D))$. By Corollary 5.13, we have $L=\mathcal{O}(\log n\cdot (\operatorname{len}(A')+\operatorname{len}(B))+\operatorname{len}(B)\log(1+\operatorname{len}(B)/\operatorname{len}(A')))=\mathcal{O}(n\log n)$, Hence, the total running time is $\mathcal{O}(W\cdot n\log^4 n)$ as claimed.

We now use this procedure to maintain a w-optimal alignment during |X| + |Y| updates.

Lemma 8.2. There exists a dynamic algorithm that, given a positive integer n, oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, and strings $X,Y \in \Sigma^*$ that (initially) satisfy $|X| + |Y| \le n$, maintains the breakpoint representation of a w-optimal alignment $A: X \leadsto Y$ subject to n edits in X and Y such that X and Y never become empty in the algorithm lifetime. The algorithm takes $\mathcal{O}(W \cdot |X| \cdot |Y| \log^3 n + W \cdot n \log^4 n)$ time for initialization and $\mathcal{O}(W \cdot n \log^4 n)$ time per update.

Proof. The algorithm maintains a weight-balanced SLP \mathcal{G} , a pair of symbols $(A, B) \in \mathcal{S}_{\mathcal{G}}^2$ such that $\exp(A) = X$ and $\exp(B) = Y$, and a hierarchical alignment data structure D of Proposition 5.11 that stores a closed set $Q \subseteq \mathcal{S}_{\mathcal{G}}^2$ containing (A, B).

In the initialization phase, we construct A and B by building perfectly balanced parse trees on top of X and Y and using fresh symbols for all internal nodes. This step takes $\mathcal{O}(n)$ time and results in a grammar of $\mathcal{O}(n)$ different symbols. Furthermore, $\mathcal{T}(A)$ contains $\mathcal{O}(|X|)$ nodes, and thus $\mathcal{O}(|X|)$ different symbols. Analogously, $\mathcal{T}(B)$ contains $\mathcal{O}(|Y|)$ different symbols. Inserting (A,B) to Q costs $\mathcal{O}(L \cdot W \log^3 n)$ time for $L := |\sum_{(C,D) \in \mathsf{cl}(A,B)} (\mathsf{len}(C) + \mathsf{len}(D))|$ by Proposition 5.11. For analysis, consider a terminal \$ that does not occur in X so that $\mathsf{cl}(A,B)$ is disjoint with $\mathsf{cl}(\$,B)$. Now, Corollary 5.13 yields

$$L = \mathcal{O}\left(|X| \cdot (\operatorname{len}(A) + \operatorname{len}(B)) + \operatorname{len}(B)\log\left(1 + \frac{\operatorname{len}(B)}{\operatorname{len}(A)}\right)\right) = \mathcal{O}(|X| \cdot (|X| + |Y|) + n\log n).$$

Analogous symmetric arguments yield $L = \mathcal{O}(|Y| \cdot (|X| + |Y|) + n \log n)$. Thus, $L = \mathcal{O}(\min\{|X|, |Y|\})$ $(|X|+|Y|)+n\log n=\mathcal{O}(|X|\cdot|Y|+n\log n)$, so the total initialization cost is $\mathcal{O}(W\cdot|X|\cdot|Y|\log^3 n+1)$ $W \cdot n \log^4 n$.

For each edit, we apply Lemma 8.1 to construct a pair of symbols (A', B') representing the updated strings X, Y and insert this pair to Q; this costs $\mathcal{O}(W \cdot n \log^4 n)$ time.

At any time, we can retrieve the breakpoint representation of a w-optimal alignment $\mathcal{A}: X \rightsquigarrow Y$ since such an alignment corresponds to a shortest path in $\overline{AG}^w(A,B)$ from the input vertex (0,0) to the output vertex (len(A), len(B)). The alignment retrieval operation of D provides this functionality in $\mathcal{O}((d+1)\log^2 n) = \mathcal{O}(W \cdot n \log^2 n)$ time, where $d = \operatorname{ed}^w(X, Y) = \mathcal{O}(W \cdot n)$.

Next, we use a standard de-amortization trick – the idea of breaking the lifetime of the algorithm into epochs – to transform the algorithm of Lemma 8.2 into one that does not have a lifetime limitation on the number of updates it supports.

■ Corollary 8.3. There exists a dynamic algorithm that, given oracle access to a weight function $w:\overline{\Sigma}^2\to [0..W]$ and strings $X,Y\in \Sigma^*$, maintains the breakpoint representation of a w-optimal alignment $A: X \leadsto Y$ subject to edits in X and Y. The algorithm takes $\mathcal{O}(W \cdot n^2 \log^3 n)$ time for initialization and $\mathcal{O}(W \cdot n \log^4 n)$ time per update, where n = |X| + |Y| + 2.

Proof. We maintain two instances of the dynamic algorithm of Lemma 8.2, denoted A and B, and partition the lifetime of the algorithm into epochs. Each epoch lasts for $\lfloor \frac{m}{3} \rfloor$ edits, where m is the value of min $\{|X|, |Y|\}$ at the beginning of the epoch. Let us first assume that $m \geq 3$ holds at the beginning of each epoch. Note that in such a case X and Y will never become empty during the current epoch. Our procedure guarantees the following invariant at the beginning of the epoch: the algorithm **A** has been initialized with an integer threshold $\hat{n} \leq \frac{3n}{2}$ for strings \hat{X} and \hat{Y} , and it can still support at least $\left|\frac{m}{3}\right|$ edits.

The updates during the epoch are applied immediately to the algorithm \mathbf{A} , which we use to maintain a w-optimal alignment $A: X \rightsquigarrow Y$, and also buffered in a queue. In the background, we initialize **B** with a threshold n and replay the updates using the buffer so that **B** is up-to-date by the end of the epoch. In the next epoch, the algorithms $\bf A$ and $\bf B$ switch roles.

After handling c edits since the beginning of the epoch, we have $\min\{|X|, |Y|\} \in [m-c..m+c]$. In particular, the value m' of $\min\{|X|,|Y|\}$ at the end of the epoch satisfies $\frac{2}{3}m \leq m' \leq \frac{4}{3}m$, i.e., $\frac{3}{4}m' \leq m \leq \frac{3}{2}m'$. Consequently, the lifetime of the algorithm **B** allows processing at least $n - \lfloor \frac{m}{3} \rfloor \ge m - \lfloor \frac{m}{3} \rfloor \ge \lfloor \frac{2m}{3} \rfloor \ge \lfloor \frac{3 \cdot 2m'}{4 \cdot 3} \rfloor \ge \lfloor \frac{m'}{3} \rfloor$ updates from the next epoch. Overall, this means that the invariant is satisfied.

Let us analyze the update time. Each update to \mathbf{A} costs $\mathcal{O}(W \cdot \hat{n} \log^4 \hat{n})$ time. The total cost of initializing **B** and applying the updates to **B** is $\mathcal{O}(W \cdot |X| \cdot |Y| \log^3 n + W \cdot n \log^4 n + m \cdot W \cdot n \log^4 n)$. If this allowance is distributed equally among the $\lfloor \frac{m}{3} \rfloor$ updates constituting the epoch, the extra cost is $\mathcal{O}(W \cdot n \log^4 n)$ time per update. Since $|X| + |Y| \ge \frac{2}{3}n \ge \frac{4}{9}\hat{n}$ holds throughout the epoch, the worst-case update time is $\mathcal{O}(W \cdot \hat{n} \log^4 \hat{n} + W \cdot n \log^4 n) = \mathcal{O}(W \cdot (|X| + |Y|) \log^4 |X| + |Y|)$.

As far as the initialization phase is concerned, we initialize **A** using threshold n = |X| + |Y| so that it is ready to handle the first epoch (this choice clearly satisfies the invariant). This process costs $\mathcal{O}(W \cdot n^2 \log^3 n + W \cdot n \log^4 n) = \mathcal{O}(W \cdot n^2 \log^3 n)$ time.

It remains to drop the assumption that m > 3 holds at the beginning of each epoch. Whenever this does not happen, we discard the algorithms **A** and **B** and, as long as $\min\{|X|, |Y|\} < 3$, we simply compute a w-optimal alignment from scratch using Fact 3.3. Since $ed^w(X,Y) \leq W \cdot (|X| + |Y|)$, using Fact 3.3 for such a threshold costs $\mathcal{O}(W \cdot (|X| + |Y| + 1))$ time as $\min\{|X|, |Y|\} < 3$. As soon

as $\min\{|X|, |Y|\} \ge 3$, we start the usual dynamic algorithm from scratch, which also costs $\mathcal{O}(W)$ time.

Next, we use the already described algorithms that work in time $\widetilde{\mathcal{O}}((|X|+|Y|)\cdot W)$ per update to obtain algorithms that work in time $\widetilde{\mathcal{O}}(\mathsf{ed}^w(X,Y)\cdot W^2)$. We follow a similar path as for the static algorithm of Section 7. First, we describe such an algorithm for strings X and Y of small self edit distance that works for one epoch.

Lemma 8.4. There exists a dynamic algorithm that, given a positive integer k, oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, and strings $X,Y \in \Sigma^{\leq n}$ that initially satisfy self-ed(X) ≤ k and ed^w(X,Y) ≤ k, maintains the breakpoint representation of a w-optimal alignment $A: X \leadsto Y$ subject to $\lfloor k/W \rfloor$ edits in X and Y in the algorithm lifetime. The algorithm takes $\mathcal{O}(W \cdot k^2 \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations for initialization and $\mathcal{O}(W \cdot k \log^4 n)$ time per update.

Proof. Let \hat{X} and \hat{Y} be strings given to the algorithm at the initialization phase. If $|\hat{X}| \leq 3k$ or $|\hat{Y}| \leq 3k$, then $|\hat{X}| + |\hat{Y}| \leq 7k$ since $\operatorname{ed}^w(X,Y) \leq k$. In this case, we simply use Corollary 8.3, which takes $\mathcal{O}(W \cdot k^2 \log^3 n)$ time for initialization and $\mathcal{O}(W \cdot k \log^4 n)$ time per update as at any point in the lifetime of the algorithm we have $|X| + |Y| \leq 8k$ as at most $\lfloor k/W \rfloor$ edits were applied.

Thus, we henceforth assume $|\hat{X}| \geq 3k$ and $|\hat{Y}| \geq 3k$. Note that in such a case at every point in the lifetime of the algorithm, X and Y are non-empty as at most $\lfloor k/W \rfloor$ edits were applied. We proceed as in the proof of Lemma 7.3 but with the threshold tripled. We use the algorithm of Lemma 7.1 for string \hat{X} and threshold $\hat{k} = 3k$, arriving at a decomposition $\hat{X} = \bigoplus_{i=0}^{m-1} \hat{X}_i$ into fragments $\hat{X}_i = \hat{X}[\hat{x}_i..\hat{x}_{i+1})$ of length $|\hat{X}_i| \in [3k..6k)$ and a sequence of fragments $\hat{Y}_i = \hat{Y}[\hat{y}_i..\hat{y}_{i+1}']$, where $\hat{y}_i \coloneqq \max\{0, \hat{x}_i - 3k\}$ and $\hat{y}'_{i+1} \coloneqq \min\{|\hat{Y}|, \hat{x}_{i+1} + 9k\}$ for $i \in [0..m)$. Lemma 7.1 also yields a set $F \subseteq [0..m)$ of size $\mathcal{O}(k)$ such that $\hat{X}_i = \hat{X}_{i-1}$ and $\hat{Y}_i = \hat{Y}_{i-1}$ holds for all $i \in [0..m) \setminus F$. To easily handle corner cases, we assume that $[0..m) \setminus F \subseteq [2..m-5]$; if the original set F does not satisfy this condition, we add the missing $\mathcal{O}(1)$ elements. Finally, an application of Lemma 7.2 lets us build a weight-balanced SLP \mathcal{G} , a hierarchical alignment data structure D that stores a closed set $Q \subseteq \mathcal{S}_G^2$ and, for each $i \in F$, a pair of $(A_i, B_i) \in Q$ such that $\exp(A_i) = \hat{X}_i$ and $\exp(B_i) = \hat{Y}_i$.

We maintain the breakpoint representations of alignments $\mathcal{B}: \hat{X} \rightsquigarrow X$ and $\mathcal{C}: \hat{Y} \rightsquigarrow Y$ representing the changes made since the initialization of the algorithm. For each $i \in [0..m)$, we define fragments $X_i = X[x_i..x_{i+1}) := \mathcal{B}(\hat{X}_i)$ and $Y_i = Y[y_i..y'_{i+1}) := \mathcal{C}(\hat{Y}_i)$ and consider the subgraph G_i of $\overline{AG}^w(X,Y)$ induced by $[x_i..x_{i+1}] \times [y_i..y'_{i+1}]$. Denote by G the union of all subgraphs G_i .

 \square Claim 8.5. The distance from (0,0) to (|X|,|Y|) in G equals $\operatorname{ed}^w(X,Y)$.

Proof. Since G is a subgraph of $\overline{\mathrm{AG}}^w(X,Y)$, it suffices to prove that G contains a path corresponding to a w-optimal alignment $\mathcal{A}:X\leadsto Y$. For this, we need to show that, for every $i\in[0..m)$, the fragment $Y[\bar{y}_i.\bar{y}'_{i+1})\coloneqq\mathcal{A}(X_i)$ is contained within Y_i .

Denote $d_X = \operatorname{ed}_{\mathcal{B}}(\hat{X}, X), d_Y = \operatorname{ed}_{\mathcal{C}}(\hat{Y}, Y),$ and $\mathcal{D} = \mathcal{C} \circ (\hat{\mathcal{A}} \circ \mathcal{B}^{-1})$ for a w-optimal alignment $\hat{\mathcal{A}}: \hat{X} \leadsto \hat{Y}$. Since $\operatorname{ed}_{\hat{\mathcal{A}}}^w(\hat{X}, \hat{Y}) = \operatorname{ed}^w(\hat{X}, \hat{Y}) \leq k$ and $d_X + d_Y \leq \lfloor k/W \rfloor$, Corollary 3.7 yields $\operatorname{ed}_{\mathcal{A}}^w(X, Y) = \operatorname{ed}^w(X, Y) \leq \operatorname{ed}_{\mathcal{D}}^w(X, Y) \leq \operatorname{ed}_{\hat{\mathcal{A}}}^w(\hat{X}, \hat{Y}) + W \cdot (d_X + d_Y) \leq k + W \cdot \lfloor k/W \rfloor \leq 2k.$ Consequently, $x_i \leq \bar{y}_i + 2k$ and $x_{i+1} \geq \bar{y}'_{i+1} - 2k$ due to Lemma 5.3. Moreover, $\hat{x}_i \leq x_i + d_X \leq \bar{y}_i + 2k + d_X$ and $\hat{x}_{i+1} \geq x_{i+1} - d_X \geq \bar{y}'_{i+1} - 2k - d_X$. The definitions of \hat{y}_i and \hat{y}'_{i+1} yield $\hat{y}_i = \max\{\hat{x}_i - 3k, 0\} \leq \max\{\bar{y}_i - k + d_X, 0\}$ and $\hat{y}'_{i+1} \geq \min\{\hat{x}_{i+1} + 3k, |\hat{Y}|\} \geq \min\{\bar{y}'_{i+1} + k - d_X, |\hat{Y}|\}$. If $\hat{y}_i = 0$, then $y_i = 0 \leq \bar{y}_i$; otherwise, $y_i \leq \hat{y}_i + d_Y \leq \bar{y}_i - k + d_X + d_Y \leq \bar{y}_i$. If $\hat{y}'_{i+1} = |\hat{Y}|$, then $y'_{i+1} = |Y| \geq \bar{y}'_{i+1}$; otherwise, $y'_{i+1} \geq \hat{y}_{i+1} - d_Y \geq \bar{y}'_{i+1} + k - d_X - d_Y \geq \bar{y}'_{i+1}$. In all cases, $y_i \leq \bar{y}_i$ and $y'_{i+1} \geq \bar{y}'_{i+1}$, so $Y[\bar{y}_i \cdot \bar{y}'_{i+1}]$ is indeed contained within $Y_i = Y[y_i \cdot y'_{i+1})$.

For each $i \in [0..m]$, denote $V_i := \{(x,y) \in V(G) \mid x = x_i, y \in [y_i..y'_i]\}$, where $y_m = |Y|$ and $y'_0 = 0$. Moreover, for $i, j \in [0..m]$ with $i \leq j$, let $D_{i,j}$ denote the matrix of pairwise distances from V_i to V_j in G, where rows of $D_{i,j}$ represent vertices of V_i in the decreasing order of the second coordinate, and columns of $D_{i,j}$ represent vertices of V_j in the decreasing order of the second coordinate. Since $V_0 = \{(0,0)\}$ and $V_m = \{(|X|,|Y|)\}$, the only entry in $D_{0,m}$ stores $\operatorname{ed}^w(X,Y)$.

Note that, for every $a, i, b \in [0..m]$ with $a \leq i \leq b$, the set V_i is a separator in G between all vertices of V_a and V_b . Thus, every path from V_a to V_b in G must cross V_i , and hence $D_{a,b} = D_{a,i} \otimes D_{i,b}$. In particular, $D_{0,m} = \bigotimes_{i=0}^{m-1} D_{i,i+1}$.

We maintain a perfectly balanced binary tree T with leaves indexed by [0..m) from left to right; we make sure that, for every node ν , the interval $[\ell_{\nu}..r_{\nu})$ of leaves in the subtree of ν is dyadic, that is, there exists an integer $j \geq 0$, called the level of the node ν , such that ℓ_{ν} is an integer multiple of 2^{j} and $r_{\nu} = \min\{\ell_{\nu} + 2^{j}, m\}$. For every node ν , we maintain $\mathsf{CMO}(D_{\ell_{\nu}, r_{\nu}})$, where $[\ell_{\nu}..r_{\nu})$ is the interval of leaves in the subtree of ν , as well as the values $x_{r_{\nu}} - x_{\ell_{\nu}}, y_{r_{\nu}} - y_{\ell_{\nu}}$, and $y'_{r_{\nu}} - y'_{\ell_{\nu}}$. Additionally, each leaf with $[\ell_{\nu}..r_{\nu}) = \{i\}$ stores a pair of symbols $(A_{i}, B_{i}) \in Q$ such that $X_{i} = \exp(A_{i})$ and $Y_{i} = \exp(B_{i})$. We emphasize that identical subtrees may share the same memory location.

Initialization. Upon initialization, we use Lemma 7.1 to compute the decomposition $\hat{X} = \bigoplus_{i=0}^{m-1} \hat{X}_i$ and a set $F \subseteq [0..m)$ of size $\mathcal{O}(k)$ such $\hat{X}_i = \hat{X}_{i-1}$ and $\hat{Y}_i = \hat{Y}_{i-1}$ for $i \in [0..m) \setminus F$. This set is returned along with the endpoints of \hat{X}_i for each $i \in F$; the endpoints of \hat{Y}_i can be retrieved by their definition from the endpoints of \hat{X}_i . We also apply Lemma 7.2 to compute (A_i, B_i) and make sure that $(A_i, B_i) \in Q$ for all $i \in F$. Furthermore, we initialize $\mathcal{B}: \hat{X} \rightsquigarrow \hat{X}$ and $\mathcal{C}: \hat{Y} \rightsquigarrow \hat{Y}$ as identity alignments.

It remains to initialize the tree T. We say that node ν is primary if $F \cap (2\ell_{\nu} - r_{\nu}...r_{\nu}) \neq \emptyset$; this condition is equivalent to $F \cap (\ell_{\nu} - 2^{j}..\ell_{\nu} + 2^{j}) \neq \emptyset$, where j is the level of the node. Observe that if ν is not primary, then the subtree rooted at ν is identical to the subtree rooted at the preceding node of the same level, i.e., the node μ with $[\ell_{\mu}...r_{\mu}) = [\ell_{\nu} - 2^{j}..\ell_{\nu})$. It follows from the fact that for two consecutive elements $i, i' \in F \cup \{m\}$, we have $D_{i,i+1} = D_{i+1,i+2} = \cdots = D_{i'-1,i'}$ analogously to Claim 7.4. We traverse primary nodes of T in the post-order, maintaining a stack consisting of the most recently visited node at each level. Upon entering any node ν , we first check if ν is primary and, if not, simply return a pointer to the most recently visited node at the same level. If ν is a primary leaf with label i, we must have $i \in F$. Thus, we create a new node storing (A_i, B_i) and $\mathsf{CMO}(D_{\ell_i,i+1})$, retrieved using D and Lemma 4.7 as $D_{i,i+1}$ is a contiguous submatrix of $\mathsf{BM}^w(A_i, B_i)$. If ν is a primary internal node, we recurse on the two children λ and ρ . Since $D_{\ell_{\nu},r_{\nu}} = D_{\ell_{\lambda},r_{\lambda}} \otimes D_{\ell_{\rho},r_{\rho}}$, we can construct $\mathsf{CMO}(D_{\ell_{\nu},r_{\nu}})$ from $\mathsf{CMO}(D_{\ell_{\lambda},r_{\lambda}})$ and $\mathsf{CMO}(D_{\ell_{\rho},r_{\rho}})$, retrieved from the children of ν , using Lemma 4.8.

Overall, first phase, which uses Lemma 7.1 to build F along with the fragments \hat{X}_i and \hat{Y}_i for each $i \in F$, takes $\mathcal{O}(k^2)$ time in the PILLAR model. An application of Lemma 7.2 to build (A_i, B_i) for all $i \in F$ costs $\mathcal{O}(W \cdot k^2 \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations. Finally, while building T we create $\mathcal{O}(k \log n)$ primary nodes $(\mathcal{O}(k))$ per level). Initializing each of them costs $\mathcal{O}(W \cdot k \log^3 n)$ time, dominated by the application of Lemma 4.8 for matrices whose cores are of size $\mathcal{O}(kW)$ by Lemma 5.7. The total initialization time is therefore $\mathcal{O}(W \cdot k^2 \log^4 k)$ plus $\mathcal{O}(k^2)$ PILLAR operations.

Updates. Given some edit $(\mathcal{B}_i, \mathcal{C}_i)$, we use it to update \mathcal{B} and \mathcal{C} using Corollary 3.5 in time $\mathcal{O}(k/W)$. Define a set $\Delta \subseteq [0..m)$ consisting of all integers i such that X_i or Y_i is affected by this update. That is, either \mathcal{B}_i does not match X_i perfectly or \mathcal{C}_i does not match Y_i perfectly. The fragments \hat{X}_i are disjoint, so each edit in X affects at most one fragment $X_i = \mathcal{B}(\hat{X}_i)$. The fragments \hat{Y}_i, \hat{Y}_{i+1} for $i \geq 3$ start at least 3k positions apart and overlap by at most 12k characters, so each position is contained

in at most eight fragments $Y_i = \mathcal{C}(\hat{Y}_i)$. Consequently, each edit in Y affects at most eight fragments $Y_i = \mathcal{C}(\hat{Y}_i)$. We reconstruct all affected nodes ν in T such that $[\ell_{\nu}...r_{\nu}) \cap \Delta \neq \emptyset$. For this, we traverse the tree T in a post-order fashion skipping subtrees rooted at unaffected nodes. Keeping track of the relative location of the edit compared to $x_{\ell_{\nu}}$, $y_{\ell_{\nu}}$, and $y'_{\ell_{\nu}}$, we can check this condition based on the stored values $x_{r_{\nu}} - x_{\ell_{\nu}}$, $y_{r_{\nu}} - y_{\ell_{\nu}}$, and $y'_{r_{\nu}} - y'_{\ell_{\nu}}$. For each affected leaf i, we create an updated leaf using Lemma 8.1 to obtain a new pair of symbols $(A_i, B_i) \in Q$ and Proposition 5.11 and Lemma 4.7 to retrieve $\mathsf{CMO}(D_{i,i+1})$; this costs $\mathcal{O}(W \cdot k \log^4 n)$ time per affected leaf due to Lemma 5.7. For each affected internal node ν , after processing the two children λ and ρ , we can construct $\mathsf{CMO}(D_{\ell_{\nu},r_{\nu}})$ from $\mathsf{CMO}(D_{\ell_{\lambda},r_{\lambda}})$ and $\mathsf{CMO}(D_{\ell_{\rho},r_{\rho}})$, retrieved from the children of ν , using Lemma 4.8. Processing each internal node costs $\mathcal{O}(W \cdot k \log^3 n)$ time since $|X_i| + |Y_i| \leq 25k$ and $|V_i| \leq 13k$ hold throughout the lifetime of the algorithm, each processed matrix is of size $\mathcal{O}(k) \times \mathcal{O}(k)$ and, by Lemma 5.7, has core of size $\mathcal{O}(kW)$. Overall, we update $\mathcal{O}(1)$ leaves and $\mathcal{O}(\log n)$ internal nodes, so the total update time is $\mathcal{O}(W \cdot k \log^4 n)$.

Alignment retrieval. It remains to provide a subroutine that constructs the breakpoint representation of a w-optimal alignment $\mathcal{A}: X \rightsquigarrow Y$. For this, we develop a recursive procedure that, given a node ν in T, a vertex $p \in V_{\ell_{\nu}}$, and a vertex $q \in V_{r_{\nu}}$, construct the breakpoint representation of a shortest path from p to q in G.

The distance $d = \operatorname{dist}_G(p,q)$ is stored within $D_{\ell_{\nu},r_{\nu}}$, so we can retrieve it in $\mathcal{O}(\log n)$ time from $\operatorname{CMO}(D_{\ell_{\nu},r_{\nu}})$. If d=0, there is a trivial shortest path from p to q always taking diagonal edges of weight zero, and we can return the breakpoint representation of such a path in constant time. We henceforth assume d>0 and consider two further cases.

If ν is a leaf with $[\ell_{\nu}..r_{\nu}) = \{i\}$, then p can be interpreted as in input vertex of G_i and q can be interpreted as an output vertex of G_i . Moreover, by Lemma 5.2, every shortest path from p to q is completely contained in G_i . Since G_i is isomorphic to $\overline{AG}^w(A_i, B_i)$, it suffices to use the alignment retrieval operation of D.

If ν is an internal node with children λ and ρ , then $V_{r_{\lambda}} = V_{\ell_{\rho}}$ separates $V_{\ell_{\nu}} = V_{\ell_{\lambda}}$ from $V_{r_{\nu}} = V_{r_{\rho}}$. Hence, every path from p to q contains a vertex $v \in V_{r_{\lambda}}$ and $d = \min_{v \in V_{r_{\lambda}}} (\operatorname{dist}_{G}(p, v) + \operatorname{dist}_{G}(v, q))$. By Lemma 5.3, we can identify $\mathcal{O}(d)$ vertices $v \in V_{r_{\lambda}}$ for which $\operatorname{dist}_{G}(p, v) \leq \operatorname{dist}_{\overline{\operatorname{AG}}^{w}(X,Y)}(p,q) \leq d$ may hold. For each such vertex, we can compute $\operatorname{dist}_{G}(p,v)$ and $\operatorname{dist}_{G}(v,q)$ in $\mathcal{O}(\log n)$ time using the random access provided by $\operatorname{CMO}(D_{\ell_{\lambda},r_{\lambda}})$ and $\operatorname{CMO}(D_{\ell_{\rho},r_{\rho}})$. We pick any vertex v such that $\operatorname{dist}_{G}(p,v) + \operatorname{dist}_{G}(v,q) = d$, recurse on (λ,p,v) and (ρ,v,q) , and concatenate the obtained breakpoint representations.

Let us analyze the running time of this algorithm. For this, for every node ν , denote with d_{ν} the cost that the returned path incurs on the way from $V_{\ell_{\nu}}$ to $V_{r_{\nu}}$. If $d_{\nu} = 0$, then the retrieval algorithm spends $\mathcal{O}(\log n)$ time (dominated by the random access to $D_{\ell_{\nu},r_{\nu}}$). Unless ν is the root, this can be charged to the parent μ of ν that satisfies $d_{\mu} > 0$. If $d_{\nu} > 0$ and ν is a leaf, then the running time is $\mathcal{O}(d_{\nu}\log^2 n)$, dominated by the alignment retrieval operation of D; see Proposition 5.11. If $d_{\nu} > 0$ and ν is an internal node, then the running time (excluding recursive calls) is $\mathcal{O}(d_{\nu}\log n)$, dominated by $\mathcal{O}(d_{\nu})$ random access queries to $D_{\ell_{\lambda},r_{\lambda}}$ and $D_{\ell_{\rho},r_{\rho}}$. For each level of the tree T, the values d_{ν} sum up to d. Hence, the total cost at the leaf level is at most $\mathcal{O}(d\log^2 n)$ and the total cost at every other single level is $\mathcal{O}(d\log n)$. There are $\mathcal{O}(\log n)$ levels overall, so the global cost is $\mathcal{O}(d\log^2 n)$ if d > 0 and $\mathcal{O}(\log n)$ if d = 0. Since $d = \operatorname{ed}^w(X,Y) \leq 2k$, this is clearly dominated by the update cost of $\mathcal{O}(W \cdot k \log^4 n)$.

We now describe a procedure that uses Lemma 8.4 to maintain a data structure able to combine w-optimal alignments for two halves of the strings into a w-optimal alignment for the whole strings.

■ Lemma 8.6. There exists a dynamic algorithm that, given a positive integer k, oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, and strings $X_L, X_R, Y_L, Y_R \in \Sigma^{\leq n}$ that initially satisfy $\operatorname{ed}^w(X_L, Y_L) + \operatorname{ed}^w(X_R, Y_R) \leq k$, maintains, subject to $\lfloor k/W \rfloor$ edits in X_L, X_R, Y_L, Y_R in the algorithm lifetime, a data structure supporting the following queries: given w-optimal alignments $\mathcal{O}_L: X_L \leadsto Y_L$ and $\mathcal{O}_R: X_R \leadsto Y_R$, construct a w-optimal alignment $\mathcal{O}: X \leadsto Y$, where $X = X_L X_R$ and $Y = Y_L Y_R$. The algorithm takes $\mathcal{O}(W \cdot k^2 \log^3 n)$ time plus $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations for initialization, $\mathcal{O}(W \cdot k \log^4 n)$ time per update, and $\mathcal{O}(k)$ time per query.

Proof. Let $\hat{X}_L, \hat{X}_R, \hat{Y}_L, \hat{Y}_R$ be strings given to the algorithm at the initialization phase. Consider w-optimal alignments $\mathcal{A}_L: \hat{X}_L \leadsto \hat{Y}_L$ and $\mathcal{A}_R: \hat{X}_R \leadsto \hat{Y}_R$. Moreover, let \hat{X}_L^* be the longest suffix of \hat{X}_L satisfying self-ed $(\hat{X}_L^*) \leq 11k$ and \hat{X}_R^* be the longest prefix of \hat{X}_R satisfying self-ed $(\hat{X}_R^*) \leq 11k$. Furthermore, let $\hat{Y}_L^* = \mathcal{A}_L(\hat{X}_L^*)$ and $\hat{Y}_R^* = \mathcal{A}_R(\hat{X}_R^*)$.

We maintain the breakpoint representations of alignments $\mathcal{B}_L: \hat{X}_L \rightsquigarrow X_L$, $\mathcal{B}_R: \hat{X}_R \rightsquigarrow X_R$, $\mathcal{C}_L: \hat{Y}_L \rightsquigarrow Y_L$, and $\mathcal{C}_R: \hat{Y}_R \rightsquigarrow Y_R$ representing the changes made throughout the lifetime of the algorithm, and let $X_L^* = \mathcal{B}_L(\hat{X}_L^*)$, $X_R^* = \mathcal{B}_R(\hat{X}_R^*)$, $Y_L^* = \mathcal{C}_L(\hat{Y}_L^*)$, and $Y_R^* = \mathcal{C}_R(\hat{Y}_R^*)$.

At the initialization phase, we apply Fact 6.3 combined with binary search to determine \hat{X}_L^* and \hat{X}_R^* . We also use Theorem 7.6 to compute (breakpoint representations of) \mathcal{A}_L and \mathcal{A}_R ; this lets us derive \hat{Y}_L^* and \hat{Y}_R^* , respectively. Finally, we initialize an instance of the dynamic algorithm of Lemma 8.4 for $(X_L^*X_R^*, Y_L^*Y_R^*, 22k)$. We prove that such a threshold is sufficient. Firstly, self-ed $(X_L^*X_R^*) \leq \text{self-ed}(X_L^*) + \text{self-ed}(X_R^*) \leq 11k + 11k = 22k$ follows from Lemma 6.2. Secondly, denote $d_X = \text{ed}_{\mathcal{B}_L}(\hat{X}_L, X_L)$, $d_Y = \text{ed}_{\mathcal{C}_L}(\hat{Y}_L, Y_L)$, and $\mathcal{D}_L = \mathcal{C}_L \circ (\mathcal{A}_L \circ \mathcal{B}_L^{-1})$. Since $\text{ed}_{\hat{\mathcal{A}}_L}^w(\hat{X}_L, \hat{Y}_L) = \text{ed}_w^w(\hat{X}_L, \hat{Y}_L) \leq k$ and $d_X + d_Y \leq |k/W|$, Corollary 3.7 yields

$$\begin{split} \operatorname{ed}^w(X_L^*,Y_L^*) & \leq \operatorname{ed}^w_{\mathcal{D}_L}(X_L^*,Y_L^*) \\ & \leq \operatorname{ed}^w_{\mathcal{A}_L}(\hat{X}_L^*,\hat{Y}_L^*) + W \cdot (\operatorname{ed}_{\mathcal{B}_L}(\hat{X}_L^*,X_L^*) + \operatorname{ed}_{\mathcal{C}_L}(\hat{Y}_L^*,Y_L^*)) \\ & \leq \operatorname{ed}^w_{\mathcal{A}_L}(\hat{X}_L,\hat{Y}_L) + W \cdot (\operatorname{ed}_{\mathcal{B}_L}(\hat{X}_L,X_L) + \operatorname{ed}_{\mathcal{C}_L}(\hat{Y}_L,Y_L)) \\ & \leq k + W \cdot \lfloor k/W \rfloor \\ & \leq 2k. \end{split}$$

Analogously, we have $\operatorname{\sf ed}^w(X_R^*,Y_R^*) \leq 2k$. Hence, $\operatorname{\sf ed}^w(X_L^*X_R^*,Y_L^*Y_R^*) \leq \operatorname{\sf ed}^w(X_L^*,Y_L^*) + \operatorname{\sf ed}^w(X_R^*,Y_R^*) \leq 2k + 2k < 22k$ follows from sub-additivity of edit distance. Therefore, the threshold of 22k is indeed sufficient.

Furthermore, we initialize $\mathcal{B}: \hat{X} \leadsto \hat{X}$ and $\mathcal{C}: \hat{Y} \leadsto \hat{Y}$ as identity alignments.

The three steps of the initialization algorithm cost $\mathcal{O}(k^2 \log n)$ PILLAR operations, $\mathcal{O}(W \cdot k^2 \log^5 n)$ time plus $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations, and $\mathcal{O}(W \cdot k^2 \log^4 n)$ time plus $\mathcal{O}(k^2)$ PILLAR operations, respectively, for a total of $\mathcal{O}(W \cdot k^2 \log^5 n)$ time plus $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations.

Given some edit $(\mathcal{B}_i, \mathcal{C}_i)$, we use it to update \mathcal{B} and \mathcal{C} using Corollary 3.5 in time $\mathcal{O}(k/W)$. Furthermore, we reflect such an edit to X_L, X_R, Y_L, Y_R in the instance $(X_L^*X_R^*, Y_L^*Y_R^*, 22k)$ of Lemma 8.4. It can support at least $\lfloor 22k/W \rfloor \geq \lfloor k/W \rfloor$ updates and its update time is $\mathcal{O}(W \cdot k \log^4 n)$. After each update, we store the breakpoint representation of a w-optimal alignment $\mathcal{O}_M: X_L^*X_R^* \rightsquigarrow Y_L^*Y_R^*$.

The query algorithm constructs a w-optimal alignment $\mathcal{O}: X \rightsquigarrow Y$, where $X = X_L X_R$ and $Y = Y_L Y_R$, based on the following claim:

□ Claim 8.7. Let $\mathcal{O}_L: X_L \rightsquigarrow Y_L$, $\mathcal{O}_R: X_R \rightsquigarrow Y_R$, and $\mathcal{O}_M: X_L^* X_R^* \rightsquigarrow Y_L^* Y_R^*$ be w-optimal alignments, interpreted as alignments mapping fragments of X to fragments of Y. Then, $\mathcal{O}_L \cap \mathcal{O}_M \neq$

 $\emptyset \neq \mathcal{O}_R \cap \mathcal{O}_M$, and all points $(x_L^*, y_L^*) \in \mathcal{O}_L \cap \mathcal{O}_M$ and $(x_R^*, y_R^*) \in \mathcal{O}_R \cap \mathcal{O}_M$ satisfy

$$\operatorname{ed}^w(X,Y) = \operatorname{ed}^w_{\mathcal{O}_L}(X[0..x_L^*),Y[0..y_L^*)) + \operatorname{ed}^w_{\mathcal{O}_M}(X[x_L^*..x_R^*),Y[y_L^*..y_R^*)) + \operatorname{ed}^w_{\mathcal{O}_R}(X[x_R^*..|X|),Y[y_R^*..|Y|)).$$

Proof. Consider an alignment $\mathcal{A}: \hat{X}_L\hat{X}_R \rightsquigarrow \hat{Y}_L\hat{Y}_R$ obtained by concatenating \mathcal{A}_L and \mathcal{A}_R , an alignment $\mathcal{B}: \hat{X}_L\hat{X}_R \rightsquigarrow X_LX_R$ obtained by concatenating \mathcal{B}_L and \mathcal{B}_R , and an alignment $\mathcal{C}: \hat{Y}_L\hat{Y}_R \rightsquigarrow Y_LY_R$ obtained by concatenating \mathcal{C}_L and \mathcal{C}_R . Finally, let $\mathcal{D}: X \rightsquigarrow Y$ be an alignment obtained as a composition $\mathcal{D} = \mathcal{C} \circ (\mathcal{A} \circ \mathcal{B}^{-1})$. Since $\operatorname{ed}_{\mathcal{A}}^w(\hat{X}_L\hat{X}_R,\hat{Y}_L\hat{Y}_R) = \operatorname{ed}_{\mathcal{A}_L}^w(\hat{X}_L,\hat{Y}_L) + \operatorname{ed}_{\mathcal{A}_R}^w(\hat{X}_R,\hat{Y}_R) = \operatorname{ed}_{\mathcal{A}_L}^w(\hat{X}_L,\hat{Y}_L) + \operatorname{ed}_{\mathcal{A}_R}^w(\hat{X}_R,\hat{Y}_R) = \operatorname{ed}_{\mathcal{A}_L}^w(\hat{X}_L,\hat{Y}_L) + \operatorname{ed}_{\mathcal{A}_R}^w(\hat{X}_R,\hat{Y}_R) = \operatorname{ed}_{\mathcal{A}_L}^w(\hat{X}_L,\hat{Y}_L) + \operatorname{ed}_{\mathcal{A}_L}^w(\hat{X}_L$

If we denote $\mathcal{D}=(x_i,y_i)_{i=0}^t$ then, by the definitions above, there exist integers $0 \leq \ell \leq m \leq r \leq t$ such that $X[0..x_m)=X_L$, $X[x_\ell..x_m)=X_L^*$, and $X[x_m..x_r)=X_R^*$. Thus, we can reinterpret alignments $\mathcal{O}_L, \mathcal{O}_M$, and \mathcal{O}_R in the following way: $\mathcal{O}_L: X[0..x_m) \rightsquigarrow Y[0..y_m)$, $\mathcal{O}_M: X[x_\ell..x_r) \rightsquigarrow Y[y_\ell..y_r)$, and $\mathcal{O}_R: X[x_m..|X|) \rightsquigarrow Y[y_m..|Y|)$. The claim statement is identical to that of Proposition 6.7, so it suffices to check that the assumptions of Proposition 6.7 are satisfied.

The maximality of \hat{X}_L^* and continuity of self-ed from Lemma 6.2 show that $\hat{X}_L^* = \hat{X}_L$ or self-ed $(\hat{X}_L^*) = 11k$. In the former case, we also have $X_L^* = \mathcal{B}(\hat{X}_L^*) = \mathcal{B}(\hat{X}_L) = X_L$ and $Y_L^* = \mathcal{C}(\mathcal{A}(\hat{X}_L^*)) = \mathcal{C}(\mathcal{A}(\hat{X}_L)) = \mathcal{C}(\hat{Y}_L) = Y_L$, so $\ell = 0$. In the latter case, Lemma 6.2 implies self-ed $(X_L^*) \geq \text{self-ed}(\hat{X}_L^*) - 2\text{ed}(\hat{X}_L^*, X_L^*) \geq 11k - 2\text{ed}_{\mathcal{B}_L}(\hat{X}_L^*, X_L^*) \geq 11k - 2\lfloor k/W \rfloor \geq 9k > 4 \cdot 2k \geq 4\text{ed}_{\mathcal{D}}^w(X, Y)$. A symmetric argument shows that r = t or ed $(X_R^*) > 4\text{ed}_{\mathcal{D}}^w(X, Y)$, so the claim indeed follows from Proposition 6.7.

Following Claim 8.7, it suffices to reinterpret \mathcal{O}_L , \mathcal{O}_R , and \mathcal{O}_M as alignments mapping fragments of X to fragments of Y (for which, we traverse the breakpoint representations and shift the coordinates accordingly), find intersection points $(x_L^*, y_L^*) \in \mathcal{O}_L \cap \mathcal{O}_M$ and $(x_R^*, y_R^*) \in \mathcal{O}_R \cap \mathcal{O}_M$, and construct the resulting alignment \mathcal{O} by following \mathcal{O}_L from (0,0) to (x_L^*, y_L^*) , following \mathcal{O}_M from (x_L^*, y_L^*) to (x_R^*, y_R^*) , and following \mathcal{O}_R from (x_R^*, y_R^*) to (|X|, |Y|). Each of these steps takes time proportional to $\mathcal{O}(1 + \operatorname{ed}^w(X_L, Y_L) + \operatorname{ed}^w(X_R, Y_R) + \operatorname{ed}^w(X_L^*, X_R^*, Y_L^*, Y_R^*)) = \mathcal{O}(k)$ due to Corollary 3.7.

We now use the de-amortization trick (the same one as in Corollary 8.3) on top of Lemma 8.6 so that the resulting algorithm does not have a lifetime limitation on the number of updates it supports.

Corollary 8.8. There exists a dynamic algorithm that, given oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$ and strings $X_L, X_R, Y_L, Y_R \in \Sigma^{\leq n}$, maintains, subject to edits in X_L, X_R, Y_L, Y_R , a data structure supporting the following queries: given w-optimal alignments $\mathcal{O}_L: X_L \leadsto Y_L$ and $\mathcal{O}_R: X_R \leadsto Y_R$, construct a w-optimal alignment $\mathcal{O}: X \leadsto Y$, where $X = X_L X_R$ and $Y = Y_L Y_R$. The algorithm takes $\mathcal{O}(W \cdot d^2 \log^5 n)$ time plus $\mathcal{O}(d^2 \log^3 n)$ PILLAR operations for initialization, $\mathcal{O}(W^2 \cdot d \log^5 n)$ time plus $\mathcal{O}(W \cdot d \log^3 n)$ PILLAR operations for updates, and $\mathcal{O}(d)$ time for queries, where $d = 1 + \operatorname{ed}^w(X_L, Y_L) + \operatorname{ed}^w(X_R, Y_R)$.

Proof. We maintain two instances of the dynamic algorithm of Lemma 8.6, denoted **A** and **B**, and partition the lifetime of the algorithm into epochs. Each epoch lasts for $\lfloor \frac{k}{3W} \rfloor$ edits, where k is the value of $\operatorname{ed}^w(X_L,Y_L) + \operatorname{ed}^w(X_R,Y_R)$ at the beginning of the epoch. Let us first assume that the $k \geq 3W$ holds at the beginning of each epoch. Our procedure guarantees the following invariant at the beginning of the epoch: the algorithm **A** has been initialized with an integer threshold \hat{k} such that $\hat{k} \leq \frac{3}{2}k$, and it can still support at least $\lfloor \frac{k}{3W} \rfloor$ updates.

The updates are applied immediately to the algorithm **A**, which we use to answer queries, and additionally buffered in a queue. In the background, we compute $k = \operatorname{\sf ed}^w(X_L, Y_L) + \operatorname{\sf ed}^w(X_R, Y_R)$

using Theorem 7.6, initialize \mathbf{B} with a threshold k, and replay the updates using the buffer so that \mathbf{B} is up-to-date by the end of the epoch. In the next epoch, the algorithms \mathbf{A} and \mathbf{B} switch roles.

By Corollary 3.7 applied to $\mathcal{B}_L, \mathcal{B}_R, \mathcal{C}_L$, and \mathcal{C}_R of Lemma 8.6, after handing c edits since the beginning of the epoch, we have $\operatorname{ed}^w(X_L, Y_L) + \operatorname{ed}^w(X_R, Y_R) \in [k - cW..k + cW]$. In particular, the value k' of $\operatorname{ed}^w(X_L, Y_L) + \operatorname{ed}^w(X_R, Y_R)$ at the end of the epoch satisfies $\frac{2}{3}k \leq k' \leq \frac{4}{3}k$, i.e., $\frac{3}{4}k' \leq k \leq \frac{3}{2}k'$. Consequently, the lifetime of the algorithm \mathbf{B} allows processing at least $\lfloor \frac{k}{W} \rfloor - \lfloor \frac{k}{3W} \rfloor \geq \lfloor \frac{2k}{3W} \rfloor \geq \lfloor \frac{3\cdot 2k'}{3W} \rfloor \geq \lfloor \frac{k'}{3W} \rfloor$ updates from the next epoch. Overall, this means that the invariant is satisfied.

Let us analyze the update time. Each update to \mathbf{A} costs $\mathcal{O}(W \cdot \hat{k} \log^4 n)$ time. The total cost of computing k, initializing \mathbf{B} and applying the updates to \mathbf{B} is $\mathcal{O}(W \cdot k^2 \log^5 n + \frac{k}{W} \cdot W \cdot k \log^4 n) = \mathcal{O}(W \cdot k^2 \log^5 n)$ time plus $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations. If this allowance is distributed equally among the $\lfloor \frac{k}{3W} \rfloor$ updates constituting the epoch, the extra cost is $\mathcal{O}(W^2 \cdot k \log^5 n)$ time plus $\mathcal{O}(W \cdot k \log^3 n)$ PILLAR operations per update. Since $d \geq \frac{2}{3}k \geq \frac{4}{9}\hat{k}$ holds throughout the epoch, the worst-case update cost is $\mathcal{O}(W \cdot \hat{k} \log^4 n + W^2 \cdot k \log^5 n) = \mathcal{O}(W^2 \cdot d \log^5 n)$ time plus $\mathcal{O}(W \cdot k \log^3 n)$ PILLAR operations.

As far as the initialization phase is concerned, we compute $k = \operatorname{ed}^w(X_L, Y_L) + \operatorname{ed}^w(X_R, Y_R)$ using Theorem 7.6 and then initialize **A** using threshold k so that it is ready to handle the first epoch (this choice clearly satisfies the invariant). This process costs $\mathcal{O}(W \cdot k^2 \log^5 n)$ time plus $\mathcal{O}(k^2 \log^3 n)$ PILLAR operations.

It remains to drop the assumption that $k \geq 3W$ holds at the beginning of each epoch. Whenever this does not happen, we discard the algorithms \mathbf{A} and \mathbf{B} and, as long as $\operatorname{ed}^w(X_L,Y_L) + \operatorname{ed}^w(X_R,Y_R) < 3W$, we simply use Theorem 7.6 to compute $\operatorname{ed}^w(X_L,Y_L)$, $\operatorname{ed}^w(X_R,Y_R)$, and $\operatorname{ed}^w(X_LX_R,Y_LY_R)$ from scratch after every update. Since a single update may change each of these values by at most $\pm W$ due to Lemma 3.6, we have $\operatorname{ed}^w(X_LX_R,Y_LY_R) \leq \operatorname{ed}^w(X_L,Y_L) + \operatorname{ed}^w(X_R,Y_R) < 4W$ and using Theorem 7.6 costs $\mathcal{O}(W \cdot d^2 \log^5 n) = \mathcal{O}(W^2 \cdot d \log^5 n)$ time plus $\mathcal{O}(d^2 \log^3 n) = \mathcal{O}(W \cdot d \log^3 n)$ PILLAR operations. As soon as $\operatorname{ed}^w(X_L,Y_L) + \operatorname{ed}^w(X_R,Y_R) \geq 3W$, we start the usual dynamic algorithm from scratch, which also costs $\mathcal{O}(W \cdot d^2 \log^5 n) = \mathcal{O}(W^2 \cdot d \log^5 n)$ time plus $\mathcal{O}(d^2 \log^3 n) = \mathcal{O}(W \cdot d \log^3 n)$ PILLAR operations.

We now use algorithms from Corollary 8.8 in a divide-and-conquer manner to maintain the w-optimal alignment of the whole string during an epoch.

■ Lemma 8.9. There exists a dynamic algorithm that, given a positive integer k, oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$, and strings $X,Y \in \Sigma^{\leq n}$ that initially satisfy $\operatorname{ed}^w(X,Y) \leq k$, maintains the breakpoint representation of a w-optimal alignment $\mathcal{O}: X \leadsto Y$ subject to $\lfloor k/W \rfloor$ edits in X and Y in the algorithm lifetime. The algorithm takes $\mathcal{O}(W \cdot k^2 \log^6 n)$ time plus $\mathcal{O}(k^2 \log^4 n)$ PILLAR operations for initialization and $\mathcal{O}(W^2 \cdot k \log^6 n)$ time plus $\mathcal{O}(W \cdot k \log^4 n)$ PILLAR operations per update.

Proof. Let \hat{X}, \hat{Y} be strings given to the algorithm at the initialization phase. Consider a w-optimal alignment $\mathcal{A}: \hat{X} \leadsto \hat{Y}$. Furthermore, we maintain the breakpoint representations of alignments $\mathcal{B}: \hat{X} \leadsto X$ and $\mathcal{C}: \hat{Y} \leadsto Y$ representing the changes made throughout the lifetime of the algorithm. In the initialization phase \mathcal{B} and \mathcal{C} are set to identity alignments, and upon an edit, they are updated in $\mathcal{O}(k/W)$ time using Corollary 3.5.

Let us also consider a perfectly balanced binary tree T built on top of \hat{X} . For each node ν of the tree, let \hat{X}_{ν} be the corresponding fragment of \hat{X} , and let $\hat{Y}_{\nu} = \mathcal{A}(\hat{Y}_{\nu})$, $X_{\nu} = \mathcal{B}(\hat{X}_{\nu})$, and $Y_{\nu} = \mathcal{C}(\hat{Y}_{\nu})$. Moreover, denote $c_{\nu} = \operatorname{ed}_{\mathcal{A}}^{w}(\hat{X}_{\nu}, \hat{Y}_{\nu}) + W\operatorname{ed}_{\mathcal{B}}(\hat{X}_{\nu}, X_{\nu}) + W\operatorname{ed}_{\mathcal{C}}(\hat{Y}_{\nu}, Y_{\nu})$. By Corollary 3.7, we have

 $\operatorname{\sf ed}^w(X_{\nu},Y_{\nu}) \leq c_{\nu}$ at any time. Consequently, $\operatorname{\sf ed}^w(X_{\lambda},Y_{\lambda}) + \operatorname{\sf ed}^w(X_{\rho},Y_{\rho}) \leq c_{\lambda} + c_{\rho} = c_{\nu}$ holds for any internal node ν with children λ and ρ .

Our algorithm explicitly stores all nodes ν with $c_{\nu} > 0$ and, for all such nodes, maintains the breakpoint representation of a w-optimal alignment $\mathcal{O}_{\nu}: X_{\nu} \rightsquigarrow Y_{\nu}$; for the remaining nodes, the w-optimal cost-0 alignment $\mathcal{O}_{\nu}: X_{\nu} \rightsquigarrow Y_{\nu}$ is maintained implicitly.

If ν is an internal node with $c_{\nu} > 0$ and children λ, ρ , we use a dynamic algorithm of Corollary 8.8 for $(X_{\lambda}, X_{\rho}, Y_{\lambda}, Y_{\rho})$. This algorithm lets us construct \mathcal{O}_{ν} from \mathcal{O}_{λ} and \mathcal{O}_{ρ} . If ν is a leaf with $c_{\nu} > 0$, we use a dynamic algorithm of Corollary 8.3 that maintains \mathcal{O}_{ν} directly; for such leaves, we have $|X_{\nu}| + |Y_{\nu}| \le 2 + c_{\nu} \le 3c_{\nu}$. It holds as $|\hat{X}_{\nu}| = 1$, $|X_{\nu}| \le |\hat{X}_{\nu}| + \operatorname{ed}(\hat{X}_{\nu}, X_{\nu}) \le 1 + \operatorname{ed}_{\mathcal{B}}(\hat{X}_{\nu}, X_{\nu})$, and $|Y_{\nu}| \le |\hat{X}_{\nu}| + \operatorname{ed}(\hat{X}_{\nu}, Y_{\nu}) \le 1 + \operatorname{ed}(\hat{X}_{\nu}, Y_{\nu}) + \operatorname{ed}_{\mathcal{C}}(\hat{Y}_{\nu}, Y_{\nu})$ due to Fact 3.4.

Nodes from every level **L** of the balanced binary tree represent disjoint fragments of X, \hat{X} , Y, and \hat{Y} . Hence, $\sum_{\nu \in \mathbf{L}} c_{\nu} = \sum_{\nu \in \mathbf{L}} \operatorname{ed}_{\mathcal{A}}^{w}(\hat{X}_{\nu}, \hat{Y}_{\nu}) + W \operatorname{ed}_{\mathcal{B}}(\hat{X}_{\nu}, X_{\nu}) + W \operatorname{ed}_{\mathcal{C}}(\hat{Y}_{\nu}, Y_{\nu}) \leq \operatorname{ed}_{\mathcal{A}}^{w}(\hat{X}, \hat{Y}) + W \cdot \lfloor k/W \rfloor \leq 2k$. Therefore, at any time, we have $\sum_{\nu \in T} c_{\nu} \leq 2k \cdot (1 + \log |\hat{X}|) = \mathcal{O}(k \log n)$. In particular, there are $\mathcal{O}(k \log n)$ nodes with $c_{\nu} > 0$.

The initialization phase costs $\mathcal{O}(W \cdot c_{\nu}^2 \log^5 n) = \mathcal{O}(W \cdot c_{\nu} \cdot k \log^5 n)$ time and $\mathcal{O}(c_{\nu}^2 \log^3 n) = \mathcal{O}(c_{\nu}k \log^3 n)$ PILLAR operations per node, for a total of $\mathcal{O}(W \cdot k^2 \log^6 n)$ time and $\mathcal{O}(k^2 \log^4 n)$ PILLAR operations. Each update costs $\mathcal{O}(W^2 \cdot c_{\nu} \log^5 n)$ time plus $\mathcal{O}(W \cdot c_{\nu} \log^3 n)$ PILLAR operations per node, for a total of $\mathcal{O}(W^2 k \log^6 n)$ time plus $\mathcal{O}(W \cdot k \log^4 n)$ PILLAR operations. This includes the initialization cost for all nodes ν for which c_{ν} becomes positive; each such node satisfies $c_{\nu} \leq W$ after the update, so the initialization cost of $\mathcal{O}(W \cdot c_{\nu}^2 \log^5 n)$ time plus $\mathcal{O}(c_{\nu}^2 \log^3 n)$ PILLAR operations matches the update cost of $\mathcal{O}(W^2 \cdot c_{\nu} \log^5 n)$ time plus $\mathcal{O}(W \cdot c_{\nu} \log^3 n)$ PILLAR operations.

We once more use the de-amortization trick to transform the algorithm of Lemma 8.9 into one that does not have a lifetime limitation on the number of updates it supports.

Theorem 8.10. There exists a dynamic algorithm that, given oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$ and strings $X, Y \in \Sigma^{\leq n}$, maintains the breakpoint representation of a w-optimal alignment $\mathcal{O}: X \leadsto Y$ subject to edits in X and Y. The algorithm takes $\mathcal{O}(W \cdot d^2 \log^6 n)$ time plus $\mathcal{O}(d^2 \log^4 n)$ PILLAR operations for initialization and $\mathcal{O}(W^2 \cdot d \log^6 n)$ time plus $\mathcal{O}(W \cdot d \log^4 n)$ PILLAR operations for updates, where $d = 1 + \operatorname{ed}^w(X, Y)$ (here distance is taken after the edit).

Proof. We maintain two instances of the dynamic algorithm of Lemma 8.9, denoted **A** and **B**, and partition the lifetime of the algorithm into epochs. Each epoch lasts for $\lfloor \frac{k}{3W} \rfloor$ edits, where k is the value of $\operatorname{ed}^w(X,Y)$ at the beginning of the epoch. Let us first assume that $k \geq 3W$ holds at the beginning of each epoch. Our procedure guarantees the following invariant at the beginning of the epoch: the algorithm **A** has been initialized with an integer threshold \hat{k} such that $\hat{k} \leq \frac{3}{2}k$, and it can still support at least $\lfloor \frac{k}{3W} \rfloor$.

The updates during the epoch are applied immediately to the algorithm \mathbf{A} , which we use to maintain $\operatorname{ed}^w(X,Y)$, and also buffered in a queue. In the background, we initialize \mathbf{B} with a threshold k and replay the updates using the buffer so that \mathbf{B} is up-to-date by the end of the epoch. In the next epoch, the algorithms \mathbf{A} and \mathbf{B} switch roles.

By Corollary 3.7, after handing c edits since the beginning of the epoch, we have $\operatorname{\sf ed}^w(X,Y) \in [k-cW.k+cW]$. In particular, the value k' of $\operatorname{\sf ed}^w(X,Y)$ at the end of the epoch satisfies $\frac{2}{3}k \leq k' \leq \frac{4}{3}k$, i.e., $\frac{3}{4}k' \leq k \leq \frac{3}{2}k'$. Consequently, the lifetime of the algorithm $\mathbf B$ allows processing at least $\lfloor \frac{k}{W} \rfloor - \lfloor \frac{k}{3W} \rfloor \geq \lfloor \frac{2k}{3W} \rfloor \geq \lfloor \frac{3 \cdot 2k'}{4 \cdot 3W} \rfloor \geq \lfloor \frac{k'}{3W} \rfloor$ updates from the next epoch. Overall, this means that the invariant is satisfied.

Let us analyze the update time. Each update to \mathbf{A} costs $\mathcal{O}(W^2 \cdot \hat{k} \log^6 n)$ time plus $\mathcal{O}(W \cdot \hat{k} \log^4 n)$ PILLAR operations. The total cost of initializing \mathbf{B} and applying to updates to \mathbf{B} is $\mathcal{O}(W \cdot k^2 \log^6 n) + \frac{k}{W} \cdot W^2 \cdot k \log^6 n) = \mathcal{O}(W \cdot k^2 \log^6 n)$ time plus $\mathcal{O}(k^2 \log^4 n + \frac{k}{W} \cdot W \cdot k \log^4 n) = \mathcal{O}(k^2 \log^4 n)$ PILLAR operations. If this allowance is distributed equally among the $\lfloor \frac{k}{3W} \rfloor$ updates constituting the epoch, the extra cost is $\mathcal{O}(W^2 \cdot k \log^6 n)$ time plus $\mathcal{O}(W \cdot k \log^4 n)$ PILLAR operations per update. Since $d \geq \frac{2}{3}k \geq \frac{4}{9}\hat{k}$ holds throughout the epoch, the worst-case update cost is $\mathcal{O}(W^2 \cdot \hat{k} \log^6 n + W^2 \cdot k \log^6 n) = \mathcal{O}(W^2 \cdot d \log^6 n)$ time plus $\mathcal{O}(W \cdot \hat{k} \log^4 n + W \cdot k \log^4 n) = \mathcal{O}(W \cdot d \log^4 n)$ PILLAR operations.

As far as the initialization phase is concerned, we compute $k = \operatorname{ed}^w(X, Y)$ using Theorem 7.6 and then initialize **A** using threshold k so that it is ready to handle the first epoch (this choice clearly satisfies the invariant). This process costs $\mathcal{O}(W \cdot k^2 \log^5 n)$ time plus $\mathcal{O}(k^2 \log^4 n)$ PILLAR operations.

It remains to drop the assumption that $k \geq 3W$ holds at the beginning of each epoch. Whenever this does not happen, we discard the algorithms \mathbf{A} and \mathbf{B} and, as long as $\operatorname{ed}^w(X,Y) < 3W$, we simply use Theorem 7.6 to compute $\operatorname{ed}^w(X,Y)$ from scratch after every update. Since a single update may change $\operatorname{ed}^w(X,Y)$ by at most $\pm W$ due to Lemma 3.6, we have $\operatorname{ed}^w(X,Y) < 4W$, and using Theorem 7.6 costs $\mathcal{O}(W \cdot d^2 \log^5 n) = \mathcal{O}(W^2 \cdot d \log^5 n)$ time plus $\mathcal{O}(d^2 \log^4 n) = \mathcal{O}(W \cdot d \log^4 n)$ PILLAR operations. As soon as $\operatorname{ed}^w(X,Y) \geq 3W$, we start the usual dynamic algorithm from scratch, which also costs $\mathcal{O}(W \cdot d^2 \log^6 n) = \mathcal{O}(W^2 \cdot d \log^6 n)$ time plus $\mathcal{O}(d^2 \log^4 n) = \mathcal{O}(W \cdot d \log^4 n)$ PILLAR operations.

Finally, we show how to implement the PILLAR operations of Theorem 8.10 with little overhead.

Corollary 8.11. There exists a dynamic algorithm that, given oracle access to a weight function $w: \overline{\Sigma}^2 \to [0..W]$ and strings $X, Y \in \Sigma^{\leq n}$, maintains the breakpoint representation of a w-optimal alignment $\mathcal{O}: X \leadsto Y$ subject to edits in X and Y. The algorithm takes $\mathcal{O}((|X| + |Y|) \log^{\mathcal{O}(1)} \log n + W \cdot d^2 \log^6 n)$ time for initialization and $\mathcal{O}(W^2 \cdot d \log^6 n)$ time for updates, where $d = 1 + \operatorname{ed}^w(X, Y)$.

Proof. We use Theorem 8.10, along with a deterministic implementation of the PILLAR model in the dynamic setting [KK22, CKW22]. The extra overhead of the latter is $\mathcal{O}((|X|+|Y|)\log^{\mathcal{O}(1)}\log n)$ for the initialization and $\mathcal{O}(\log n\log^{\mathcal{O}(1)}\log n)$ for each update and PILLAR operation. As there is an $\Theta(W \cdot \log^2 n) = \omega(\log n\log^{\mathcal{O}(1)}\log n)$ gap in Theorem 8.10 between the time complexity and the number of PILLAR operations, the time complexity only changes by an additive factor of $\mathcal{O}((|X|+|Y|)\log^{\mathcal{O}(1)}\log n)$ in the initialization phase.

Bibliography

- AB18 Amihood Amir and Itai Boneh. Locally maximal common factors as a tool for efficient dynamic string algorithms. In *CPM 2018*, volume 105 of *LIPIcs*, pages 11:1–11:13. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CPM.2018.11.
- AB20 Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In *ISAAC* 2020, volume 181 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ISAAC.2020.63.
- ABCK19 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In ESA 2019, volume 144 of LIPIcs, pages 5:1–5:18. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.5.
- ABR00 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.

- ABW15 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.
- AHWW16 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.
- AJ21 Shyan Akmal and Ce Jin. Faster algorithms for bounded tree edit distance. In *ICALP 2021*, volume 198 of *LIPIcs*, pages 12:1–12:15. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.12.
- AKM⁺87 Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987. doi:10.1007/BF01840359.
- Karl Bringmann, Alejandro Cassis, Nick Fischer, and Tomasz Kociumaka. Faster sublinear-time edit distance. In SODA 2024, volume abs/2312.01759, pages 3274–3301. SIAM, 2024. doi:10.1137/1.9781611977912.117.
- BCFN22a Karl Bringmann, Alejandro Cassis, Nick Fischer, and Vasileios Nakos. Almost-optimal sublinear-time edit distance in the low distance regime. In STOC 2022, pages 1102–1115. ACM, 2022. doi:10.1145/3519935.3519990.
- BCFN22b Karl Bringmann, Alejandro Cassis, Nick Fischer, and Vasileios Nakos. Improved sublinear-time edit distance for preprocessed strings. In *ICALP 2022*, volume 229 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.32.
- 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.
- 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/F0CS.2015.15.
- BK23 Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching. In STOC 2023, pages 219–232. ACM, 2023. doi: 10.1145/3564246.3585239.
- Arturs Backurs and Krzysztof Onak. Fast algorithms for parsing sequences of parentheses with few errors. In *PODS 2016*, pages 477–488. ACM, 2016. doi:10.1145/2902251.2902304.
- Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In FOCS 2016, pages 51–60. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.15.
- CGK⁺22 Raphaël Clifford, Paweł Gawrychowski, Tomasz Kociumaka, Daniel P. Martin, and Przemysław Uznański. The dynamic k-mismatch problem. In *CPM 2022*, volume 223 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CPM.2022.18.
- Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *ICALP 2020*, volume 168 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.27.
- CKM20 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.
- CKW20 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/F0CS46700.2020.00095.
- CKW22 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster pattern matching under edit distance: A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In FOCS 2022, pages 698–707. IEEE, 2022. doi:10.1109/F0CS54457.2022.00072.
- CKW23 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/F0CS57990.2023.00135.

- DGH⁺22 Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, Barna Saha, and Hamed Saleh. $\tilde{O}(n + \text{poly}(k))$ -time algorithm for bounded tree edit distance. In FOCS 2022, pages 686–697. IEEE, 2022. doi:10.1109/F0CS54457.2022.00071.
- DGH⁺23 Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, and Barna Saha. Weighted edit distance computation: Strings, trees, and Dyck. In *STOC 2023*, STOC 2023, page 377–390, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3564246. 3585178.
- DSO78 Margaret Dayhoff, R Schwartz, and B Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5:345–352, 1978.
- Dür23 Anita Dürr. Improved bounds for rectangular monotone min-plus product and applications. *Inf. Process. Lett.*, 181:106358, 2023. doi:10.1016/j.ipl.2023.106358.
- FGK⁺23 Dvir Fried, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Tatiana Starikovskaya. An improved algorithm for the k-Dyck edit distance problem. ACM Trans. Algorithms, 2023. doi:10.1145/3627539.
- FR06 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.
- Gaw11 Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In ESA 2011, volume 6942 of LNCS, pages 421–432. Springer, 2011. doi:10.1007/978-3-642-23719-5_36.
- GJ21 Paweł Gawrychowski and Wojciech Janczewski. Fully dynamic approximation of LIS in polylogarithmic time. In STOC 2021, pages 654–667. ACM, 2021. doi:10.1145/3406325.3451137.
- GJKT24 Daniel Gibney, Ce Jin, Tomasz Kociumaka, and Sharma V. Thankachan. Near-optimal quantum algorithms for bounded edit distance and Lempel-Ziv factorization. In SODA 2024, pages 3302–3332. SIAM, 2024. doi:10.1137/1.9781611977912.118.
- GKK⁺15 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings, 2015. arXiv:1511.02612.
- GKK⁺18 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In *SODA 2018*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.
- GKKS22 Elazar Goldenberg, Tomasz Kociumaka, Robert Krauthgamer, and Barna Saha. Gap edit distance via non-adaptive queries: Simple and optimal. In FOCS 2022, pages 674–685. IEEE, 2022. doi: 10.1109/F0CS54457.2022.00070.
- GKKS23 Elazar Goldenberg, Tomasz Kociumaka, Robert Krauthgamer, and Barna Saha. An algorithmic bridge between Hamming and Levenshtein distances. In *ITCS 2023*, volume 251 of *LIPIcs*, pages 58:1–58:23. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ITCS.2023.58.
- GKLS22 Arun Ganesh, Tomasz Kociumaka, Andrea Lincoln, and Barna Saha. How compression and approximation affect efficiency in string distance measures. In SODA 2022, pages 2867–2919. SIAM, 2022. doi:10.1137/1.9781611977073.112.
- GKS19 Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance.
 In FOCS 2019, pages 1101–1120. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00070.
- Gra16 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Appl. Math.*, 212:96–103, 2016. doi:10.1016/J.DAM.2015.10.040.
- GRS20 Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. Does preprocessing help in fast sequence comparisons? In STOC 2020, pages 657–670. ACM, 2020. doi:10.1145/3357713.3384300.
- Gus97 Dan Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997. doi:10.1017/CB09780511574931.
- HH92 Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci.*, 89(22):10915–10919, November 1992. doi:10.1073/pnas.89.22.10915.

- HHS22 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms A quick reference guide. ACM J. Exp. Algorithmics, 27:1.11:1–1.11:45, 2022. doi:10.1145/3555806.
- Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Fully incremental LCS computation. In FCT 2005, volume 3623 of LNCS, pages 563–574. Springer, 2005. doi: 10.1007/11537311_49.
- IP01 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.
- IPZ01 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.
- Dan Jurafsky and James H. Martin. Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009. URL: https://www.worldcat.org/oclc/315913020.
- JNW21 Ce Jin, Jelani Nelson, and Kewen Wu. An improved sketching algorithm for edit distance. In STACS 2021, volume 187 of LIPIcs, pages 45:1–45:16. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.STACS.2021.45.
- KK22 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.
- KMS23 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/F0CS57990.2023.00098.
- Koc22 Tomasz Kociumaka. Recent advances in dynamic string algorithms. A survey talk at the STOC 2022 satellite workshop "Dynamic Algorithms: Recent Advances and Applications", 2022. URL: https://youtu.be/cvFTTPlZX5U?si=USI5N41Y5BYA1CrY.
- KP04 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. J. Discrete Algorithms, 2(2):303–312, 2004. Combinational Pattern Matching. doi:https://doi.org/10.1016/S1570-8667(03)00082-0.
- KPS21 Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya. Small-space and streaming pattern matching with k edits. In FOCS 2021, pages 885–896. IEEE, 2021. doi:10.1109/F0CS52979.2021.00090.
- KS20 Tomasz Kociumaka and Barna Saha. Sublinear-time algorithms for computing & embedding gap edit distance. In FOCS 2020, pages 1168–1179. IEEE, 2020. doi:10.1109/F0CS46700.2020.00112.
- KS21 Tomasz Kociumaka and Saeed Seddighin. Improved dynamic algorithms for longest increasing subsequence. In STOC 2021, pages 640–653. ACM, 2021. doi:10.1145/3406325.3451026.
- KS24 Michal Koucký and Michael E. Saks. Almost linear size edit distance sketch. In STOC 2024, 2024.
- 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.
- LMS98 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. SIAM J. Comput., 27(2):557–582, 1998. doi:10.1137/S0097539794264810.
- LV88 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. J. Comput. System Sci., 37(1):63-78, 1988. doi:https://doi.org/10.1016/0022-0000(88)90045-1.
- WBCT15 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge University Press, 2015. doi:10.1017/CB09781139940023.
- William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. J. Comput. System Sci., 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.
- MS20 Michael Mitzenmacher and Saeed Seddighin. Dynamic algorithms for LIS and distance to monotonicity. In STOC 2020, pages 671–684. ACM, 2020. doi:10.1145/3357713.3384240.
- MSU97 Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17:183–198, 1997. doi:10.1007/bf02522825.
- Mye86 Eugene W. Myers. An O(ND) difference algorithm and its variations. Algorithmica, 1(2):251–266, 1986. doi:10.1007/BF01840446.

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

- NII⁺20 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discrete Appl. Math.*, 274:116–129, 2020. Stringology Algorithms. doi:https://doi.org/10.1016/j.dam.2019.01.014.
- NW70 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.
- Rus12 Luís M. S. Russo. Monge properties of sequence alignment. Theor. Comput. Sci., 423:30–49, 2012. doi:10.1016/J.TCS.2011.12.068.
- Ryt03 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.
- Sel74 Peter H. Sellers. On the theory and computation of evolutionary distances. SIAM J. Appl. Math., 26(4):787–793, 1974. doi:10.1137/0126070.
- Sel80 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.
- Tis08 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.
- Tis15 Alexander Tiskin. Fast distance multiplication of unit-Monge matrices. *Algorithmica*, 71(4):859–888, 2015. doi:10.1007/S00453-013-9830-Z.
- Tou07 Hélène Touzet. Comparing similar ordered trees in linear-time. J. Discrete Algorithms, 5(4):696–705, 2007. doi:10.1016/J.JDA.2006.07.002.
- Ukk85 Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985. doi: 10.1016/0196-6774(85)90023-9.
- Vin68 Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968. doi:10.1007/BF01074755.
- Wat95 Michael S. Waterman. Introduction to Computational Biology: Maps, Sequences and Genomes. Chapman & Hall/CRC Interdisciplinary Statistics. Taylor & Francis, 1995. doi:10.1201/9780203750131.
- WF74 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.
- Wil85 Dan E. Willard. New data structures for orthogonal range queries. SIAM J. Comput., 14(1):232–253, 1985. doi:10.1137/0214019.
- ZL77 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.