AMB ALGORITHMS FOR
MOLECULAR BIOLOGY

RESEARCH

Efficient edit distance with duplications and contractions

Tamar Pinhas[1†], Shay Zakov[2†], Dekel Tsur[1] and Michal Ziv-Ukelson[1*]

Abstract

We propose three algorithms for string edit distance with duplications and contractions. These include an efficient general algorithm and two improvements which apply under certain constraints on the cost function. The new algorithms solve a more general problem variant and obtain better time complexities with respect to previous algorithms. Our general algorithm is based on min-plus multiplication of square matrices and has time and space complexities of $O\left(|\Sigma|MP(n)\right)$ and $O\left(|\Sigma|n^2\right)$, respectively, where $|\Sigma|$ is the alphabet size, n is the length of the strings, and $MP(n)$ is the time bound for the computation of min-plus matrix multiplication of two $n \times n$ matrices (currently, $MP(n) = O\left(\frac{n^3 \log^3 \log n}{\log^2 n}\right)$ due to an algorithm by Chan).

For integer cost functions, the running time is further improved to $O\left(\frac{|\Sigma|n^3}{\log^2 n}\right)$. In addition, this variant of the algorithm is online, in the sense that the input strings may be given letter by letter, and its time complexity bounds the processing time of the first n given letters. This acceleration is based on our efficient matrix-vector min-plus multiplication algorithm, intended for matrices and vectors for which differences between adjacent entries are from a finite integer interval D. Choosing a constant $\frac{1}{\log_{|D|} n} < \lambda < 1$, the algorithm preprocesses an $n \times n$ matrix in $O\left(\frac{n^{2+\lambda}}{|D|}\right)$ time and $O\left(\frac{n^{2+\lambda}}{|D|\lambda^2 \log_{|D|}^2 n}\right)$ space. Then, it may multiply the matrix with any given n-length vector in $O\left(\frac{n^2}{\lambda^2 \log_{|D|} n}\right)$ time. Under some discreteness assumptions, this matrix-vector min-plus multiplication algorithm applies to several problems from the domains of context-free grammar parsing and RNA folding and, in particular, implies the asymptotically fastest $O\left(\frac{n^3}{\log^2 n}\right)$ time algorithm for single-strand RNA folding with discrete cost functions. Finally, assuming a different constraint on the cost function, we present another version of the algorithm that exploits the run-length encoding of the strings and runs in $O\left(\frac{|\Sigma|nMP(\tilde{n})}{\tilde{n}}\right)$ time and $O(|\Sigma|n\tilde{n})$ space, where $\tilde{n}$ is the length of the run-length encoding of the strings.

Keywords: Edit distance, Minisatellites, Min-plus matrix multiplication, Four Russians

Background

Comparing strings is a well-studied problem in computer science as well as in bioinformatics. Traditionally, string similarity is measured in terms of *edit distance*, which reflects the minimum-cost edit of one string to the other, based on the edit operations of substitutions (including matches) and deletions/insertions (indels). In this paper, we address the problem of string edit distance with the

additional operations of duplication and contraction. The motivation for this problem originated from the study of minisatellites and their comparisons in the context of population genetics [1].

Motivation: minisatellite comparison

A minisatellite is a section of DNA that consists of tandem repetitions of short (6–100 nucleotides) sequence motifs spanning 500 nucleotides to several thousand nucleotides. The repeated motifs also vary in sequence through base substitutions and indels. For one minisatellite locus, both the type and the number of motifs vary between

---

*Correspondence: michaluz@cs.bgu.ac.il
†Equal contributors
[1] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel
Full list of author information is available at the end of the article

individuals in a population. Therefore, pairwise comparisons of minisatellites are typically applied in studying the evolution of populations.

A *minisatellite map* represents a minisatellite region, where each motif is encoded by a letter and handled as one entity (denoted *unit*). When comparing minisatellite maps, one has to consider that regions of the map have arisen as a result of duplication events from the neighboring units. The single copy duplication model, where only one unit can duplicate at a time, is the most popular and its biological validation was asserted for the MSY1 minisatellites [1,2]. According to this model, one unit can mutate to another unit via an indel or a mutation of a single nucleotide within it. Also, a unit can be duplicated, that is, an additional copy of the unit may appear next to the original one in the map (tandem repeat). Thus, when comparing minisatellite maps, two additional operations are considered: unit duplication and unit contraction.

The EDDC problem definition

The single copy duplication model of minisatellite maps gave rise to a new variant of the string edit distance problem, *Edit Distance with Duplications and Contractions* (EDDC), which allows five edit operations: insertion, deletion, mutation, duplication and contraction.

We start with some string notations. Let s be a string. Denote by s_i the i-th letter in s, starting at index 0, and by $s_{i,j}$ the substring $s_i s_{i+1} \ldots s_{j-1}$ of s. A substring of the form $s_{i,i}$ is an empty string, which will be denoted by ε. We use superscripts to denote substrings without an explicit indication of their start and end positions, and write e.g. $s = s^a s^b$ to indicate that s is a concatenation of the two substrings s^a and s^b.

In the edit distance problem, one is given a source string s and a target string t over a finite alphabet Σ. An *edit script* from s to t is a series of strings $\mathcal{ES} = \langle s = u^0, u^1, u^2, \ldots, u^r = t \rangle$, where each intermediate string u^i is obtained by applying a single edit operation to the preceding string u^{i-1}. In the standard problem definition [3-5], the allowed edit operations are *insertion* of a letter at some position in an intermediate string u^i, *deletion* of a letter in u^i, and *mutation* of a letter in u^i to another letter. The single-copy EDDC problem variant adds two operations: *duplication* - inserting into u^i a letter in a position adjacent to a position that already contains the same letter, and *contraction* - deleting from u^i one copy of a letter where there are two consecutive copies of this letter. Denote by $ins(\alpha)$, $dup(\alpha)$ and $del(\alpha)$ costs associated with the insertion, duplication and deletion operations applied to a letter α in the alphabet, respectively, by $cont(\alpha)$ the cost of contracting two consecutive occurrences of α into a single occurrence, and by $mut(\alpha, \beta)$ the cost of mutating

α to a letter β. Define the *cost* of $\mathcal{ES}$ to be the summation of its implied operation costs, and the *length* $|\mathcal{ES}| = r$ of $\mathcal{ES}$ to be the number of operations performed in $\mathcal{ES}$. Clearly, for every pair of strings s and t, there is some script transforming s to t, e.g. the script that first deletes all letters in s and then inserts all letters in t. An optimal edit script from s to t is one which has a minimum cost. The *edit distance* from s to t, denoted by $ed(s, t)$, is the cost of an optimal edit script from s to t. The goal of the EDDC problem is, given strings s and t, to compute $ed(s, t)$.

Previous algorithms assume various constraints on operation costs (see Section "A comparison with previous works"). In this paper, the only limiting assumption made is that all operation costs are nonnegative. In addition, we can make the following assumption without loss of generality, which will be required by the algorithms presented in this paper:

Property 1. *It may be assumed without loss of generality that for every $\alpha, \beta \in \Sigma$,*

- $ins(\alpha) = ed(\varepsilon, \alpha), del(\alpha) = ed(\alpha, \varepsilon),$
- $dup(\alpha) = ed(\alpha, \alpha\alpha), cont(\alpha) = ed(\alpha\alpha, \alpha),$
- $mut(\alpha, \beta) = ed(\alpha, \beta).$

This assumption can be made, since in case one of the operation costs violates the assumption, then such an operation can always be replaced by a series of operations that would induce the same modification at lower cost. For example, it cannot be that $mut(\alpha, \beta) < ed(\alpha, \beta)$, since $ed(\alpha, \beta)$ is smaller then or equal to the cost of any script from α to β, among which is the script containing the single mutation operation of α into β. Moreover, if $mut(\alpha, \beta) > ed(\alpha, \beta)$, then we can always replace any mutation of α into β by a series of operations that transform α into β at cost $ed(\alpha, \beta)$. In this case, we may simply assume that $mut(\alpha, \beta) = ed(\alpha, \beta)$, and interpret any such a mutation appearing in a script as being implem ented by the corresponding series of operations. In particular, Property 1 implies that $mut(\alpha, \alpha) = 0$ (since all operation costs are nonnegative, $ed(w, w) = 0$ for every string w), $dup(\alpha) \leq ins(\alpha)$ (since the cost of the script from α to $\alpha\alpha$ that applies a single insertion of α is $ins(\alpha) \geq ed(\alpha, \alpha\alpha) = dup(\alpha)$), $cont(\alpha) \leq del(\alpha)$, and $mut(\alpha, \beta) \leq mut(\alpha, \gamma) + mut(\gamma, \beta)$ for every $\gamma \in \Sigma$.

Insertions and duplications are considered to be *generating* operations, increasing by one letter the length of the string. Similarly, deletions and contractions are considered to be *reducing* operations, decreasing by one letter the length of the string. An edit script containing no reducing operation is called a *non-reducing* script, and an edit script contain-

ing no generating operation is called a *non-generating* script.

Previous work

The EDDC problem was first defined by Bérard and Rivals [2], who suggested an $O(n^4)$ time and $O(n^3)$ space algorithm for the problem, where n is the length of the two input strings (for the sake of simplicity, we assume that both strings are of the same length). This was followed by the work of Behzadi and Steyaert [6], who gave an $O(|\Sigma|n^3)$ time and $O(|\Sigma|n^2)$ space algorithm for the problem, where $|\Sigma|$ is the alphabet size (typically a few tens of unique units). Behzadi and Steyaert [7] improved their algorithms' complexity, based on run-length encoding, to $O(n^2 + n\tilde{n}^2 + |\Sigma|\tilde{n}^3 + |\Sigma|^2\tilde{n}^2)$ time and $O\left(|\Sigma|(n + \tilde{n}^2) + n^2\right)$ space, where $\tilde{n}$ is the length of the run-length encoding of the input strings. Run-length encoding was also used by Bérard et al. [8], who proposed an $O(n^3 + |\Sigma|\tilde{n}^3)$ time and $O(n^2 + |\Sigma|\tilde{n}^2)$ space algorithm. Abouelhoda et al. [9] gave an algorithm with an alphabet size independent time and space complexities of $O(n^2 + n\tilde{n}^2)$ and $O(n^2)$, respectively. A detailed comparison between the different problem models appears in Section "A comparison with previous works".

Our contribution

This paper presents several algorithms for EDDC which are currently the most general and efficient for the problem.

1. We give an algorithm for EDDC for general non-negative cost functions that is based on min-plus square matrix multiplication. This algorithm is an adaptation of the framework of [10] (see also [11]). For two input strings over an alphabet Σ and of length n each, the time and space complexities of this algorithm are $O(|\Sigma|MP(n))$ and $O(|\Sigma|n^2)$, respectively, where $MP(n)$ is the time complexity of a min-plus multiplication of two $n \times n$ matrices. Using the matrix multiplication algorithm of Chan [12], this algorithm runs in $O\left(\frac{|\Sigma|n^3 \log^3 \log n}{\log^2 n}\right)$ time (Section "A matrix multiplication based algorithm for EDDC"). Moreover, our algorithm applies less restrictions on the cost function with respect to previous algorithms and is currently the only algorithm that works for the most general problem settings (Section "A comparison with previous works").

2. We describe a more efficient algorithm for EDDC when all operation costs are integers. This algorithm can also be applied in an online setting, where in each step a letter is added to one of the input strings. The time complexity of processing n letters in the input is $O\left(\frac{|\Sigma|n^3}{\log^2 n}\right)$, where the base of the log function

is determined by the range of cost values (Section "An online algorithm for EDDC using min-plus matrix-vector multiplication for discrete cost functions"). In order to achieve this, we obtained the following stepping-stone results, which are of interest on their own.

(a) Let A be an $n \times m$ matrix for which differences between adjacent entries are within some finite integer interval D. Choosing a time/space complexity tradeoff parameter λ, where $\frac{1}{\log_{|D|}(n+m)} < \lambda < 1$, we describe a preprocessing algorithm for A that runs in $O\left(\frac{nm(n+m)^\lambda}{|D|}\right)$ time and requires $O\left(\frac{nm(n+m)^\lambda}{|D|\lambda^2 \log_{|D|}^2(n+m)}\right)$ space. This preprocessing allows later to compute min-plus multiplications between A and m-length vectors sustaining the same discreteness requirement in $O\left(\frac{nm}{\lambda^2 \log_{|D|}^2(n+m)}\right)$ time (Section "An efficient D-discrete min-plus matrix-vector multiplication algorithm"). The algorithm is an adaptation of Williams' matrix-vector multiplication algorithm over a finite semiring [13], with some notions similar to those in Frid and Gusfield's RNA folding algorithm [14].

(b) The manner in which the new matrix-vector multiplication algorithm is integrated into the EDDC algorithm can be generalized to algorithms for a family of related problems, denoted VMT problems [11], under certain discreteness assumptions. This family includes many important problems from the domains of RNA folding and CFG parsing. An example of such a problem is the *single strand RNA folding problem* [15] under discrete scoring schemes. Our new matrix-vector multiplication algorithm can be integrated into an algorithm for the latter problem to yield an $O\left(\frac{n^3}{\log^2 n}\right)$ time algorithm, improving the best previously known asymptotic time bound for the problem (see Section "Online VMT algorithms").

3. We extend our approach to exploit run-length encodings of the input strings, assuming some restrictions on the cost functions. This reduces the time and space complexities of the algorithm to $O\left(|\Sigma|n^2 + \frac{|\Sigma|nMP(\tilde{n})}{\tilde{n}}\right)$ and $O\left(|\Sigma|n\tilde{n}\right)$, respectively, where $\tilde{n}$ is the length of the run-length encoding

of the input (Section "Additional acceleration using run-length encoding").

The rest of the paper is organized as follows. In Section "A baseline algorithm for the EDDC problem", a recursive computation for EDDC and its implementation using dynamic programming (DP) is presented. Section "A matrix multiplication based algorithm for EDDC" shows how to accelerate the algorithm by incorporating efficient min-plus matrix multiplication subroutines. In Section "An online algorithm for EDDC using min-plus matrix-vector multiplication for discrete cost functions", an efficient min-plus matrix-vector multiplication algorithm is described for matrices and vectors which differences between adjacent entries are taken from a finite integer interval. This algorithm can be used for obtaining an accelerated online version of the EDDC algorithm, as well as for improving time complexities of several related problems. Section "Additional acceleration using run-length encoding" describes a variant of the EDDC algorithm that exploits run-length encoding. Comparison between this and previous works is given in Section "A comparison with previous works", and Section "Conclusions and discussion" gives a concluding discussion. Additional proofs omitted from the main manuscript are given in the Appendix.

A baseline algorithm for the EDDC problem

In this section, we give a simple algorithm for the EDDC problem. We start by showing some recursive properties of the problem, and then formulate a straightforward dynamic programming implementation for the recursive computation.

The recurrence formula

Our recursive formulas resemble previous formulations [6,9], yet solve a slightly more general variant of the problem (see discussion in Section "A comparison with previous works"). Since the proof of correctness of these recursive formulas is similar to previous ones, we defer it to Appendix "Correctness of the recursive computation".

A (strict) *partition* of a string w of length at least 2 is a pair of strings (w^a, w^b), such that $w = w^a w^b$ and $w^a, w^b \neq \varepsilon$. Denote by $P(w)$ the set of all partitions of w. For example, for $w = abac$, $P(w) = \{(a, bac), (ab, ac), (aba, c)\}$.

For a source string which is either empty or contains a single letter and a target string t, Equations 1 to 3 (Figure 1) describe a recursive EDDC computation. This computation interleaves, in a mutually recursive manner, the computation of an additional special value $ed'(\alpha, t)$, where $ed'(\alpha, t)$ is defined to be the minimum cost of a

non-reducing edit script from α to t that does not start with a mutation (t is required to contain at least two letters).

$$ed(\varepsilon, t) = \begin{cases} 0, & t = \varepsilon, \\ \min\{ins(\alpha) + ed(\alpha, t) \mid \alpha \in \Sigma\}, & \text{otherwise.} \end{cases} \tag{1}$$

$$ed(\alpha, t) = \begin{cases} del(\alpha), & t = \varepsilon, \\ mut(\alpha, \beta), & t = \beta, \\ \min\{mut(\alpha, \beta) + ed'(\beta, t) \mid \beta \in \Sigma\}, & \text{otherwise.} \end{cases} \tag{2}$$
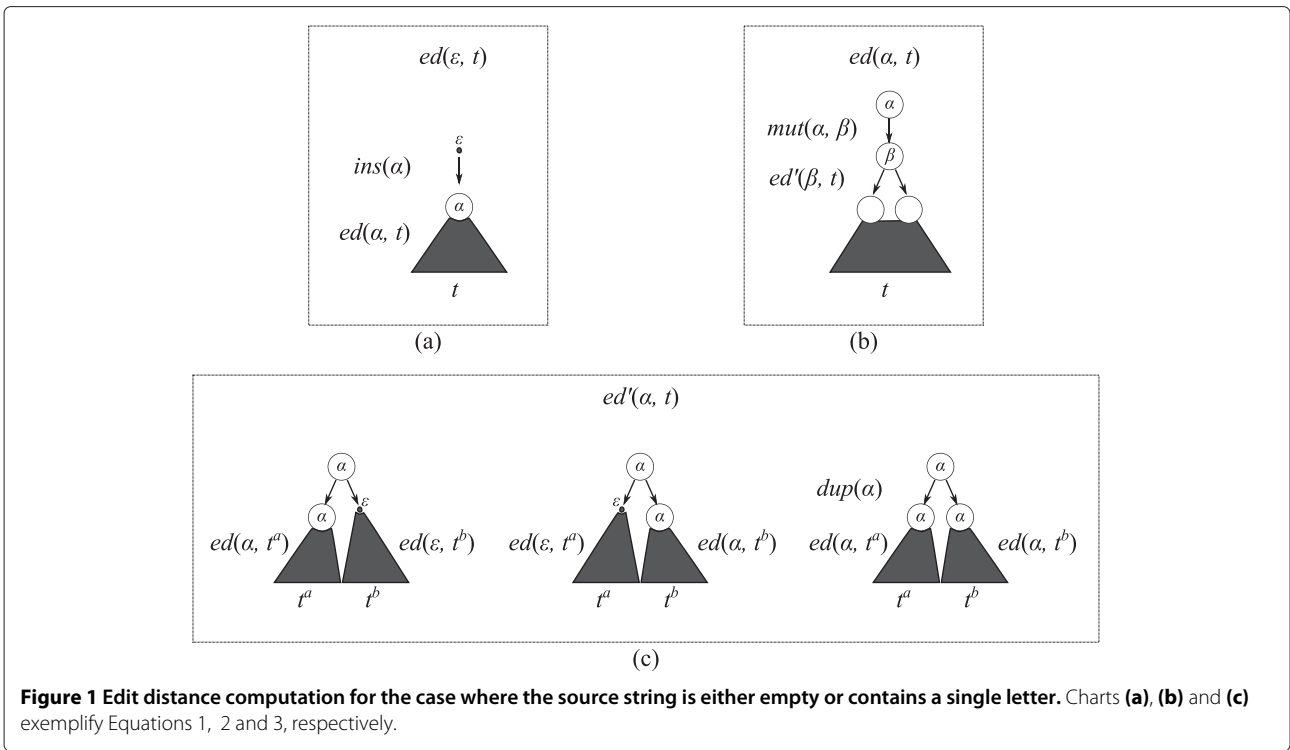
$ed'(\alpha, t)$

$$= \min \begin{cases} ed(\alpha, t^a) + ed(\varepsilon, t^b), \\ ed(\varepsilon, t^a) + ed(\alpha, t^b), \\ dup(\alpha) + ed(\alpha, t^a) + ed(\alpha, t^b) \end{cases} \left| (t^a, t^b) \in P(t) \right\}$$

(t is of length ≥ 2)

$$\tag{3}$$

Symmetrically, Equations 4 to 6 give the recursive computation for a source string s and a target string which is either empty or contains a single letter. Here, $ed'(s, \alpha)$ is defined as the minimum cost of a non-generating edit script from s to α which does not end with a mutation (s is required to contain at least two letters).

$$ed(s, \varepsilon) = \begin{cases} 0, & s = \varepsilon, \\ \min\{ed(s, \alpha) + del(\alpha) \mid \alpha \in \Sigma\}, & \text{otherwise.} \end{cases} \tag{4}$$

$$ed(s, \alpha) = \begin{cases} ins(\alpha), & s = \varepsilon, \\ mut(\beta, \alpha), & s = \beta, \\ \min\{ed'(s, \beta) + mut(\beta, \alpha) \mid \beta \in \Sigma\}, & \text{otherwise.} \end{cases} \tag{5}$$

$ed'(s, \alpha)$

$$= \min \begin{cases} ed(s^a, \alpha) + ed(s^b, \varepsilon), \\ ed(s^a, \varepsilon) + ed(s^b, \alpha), \\ ed(s^a, \alpha) + ed(s^b, \alpha) + cont(\alpha) \end{cases} \left| (s^a, s^b) \in P(s) \right\}$$

(s is of length ≥ 2)

$$\tag{6}$$

Figure 1 Edit distance computation for the case where the source string is either empty or contains a single letter. Charts **(a)**, **(b)** and **(c)** exemplify Equations 1, 2 and 3, respectively.

In case both source string s and target string t are of length at least 2, the following equation can be used for computing $ed(s,t)$ (Figure 2(a)):

$$ed(s,t)$$
$$= \min \left\{ \begin{array}{l} ed(s,\alpha) + ed(\alpha,t), \\ ed\left(s^a, t^a\right) + ed\left(s^b,\alpha\right) + ed\left(\alpha,t^b\right) \end{array} \;\middle|\; \begin{array}{c} \left(s^a, s^b\right) \in P(s), \\ \left(t^a, t^b\right) \in P(t), \\ \alpha \in \Sigma \end{array} \right\}$$
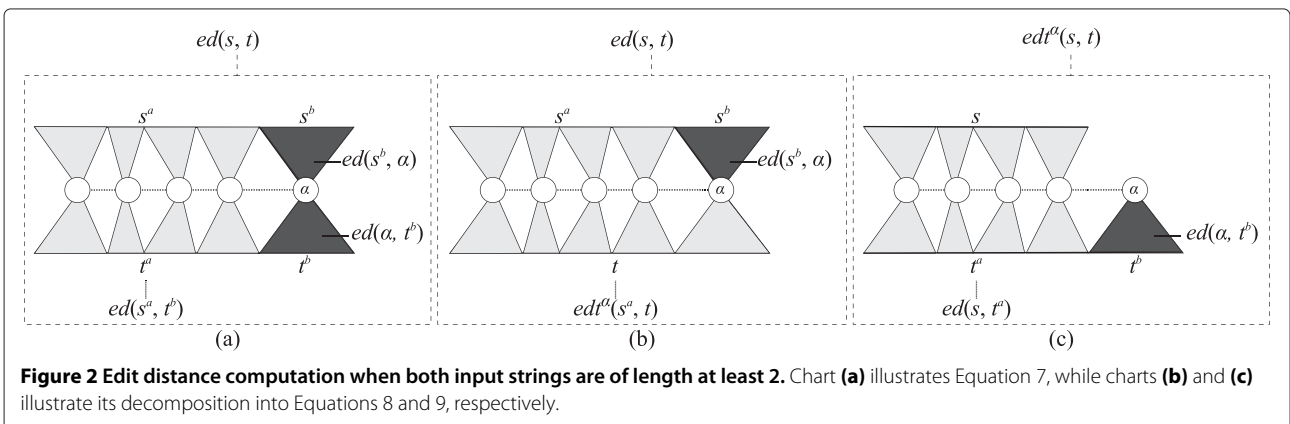
(s and t are of lengths ≥ 2)

(7)

For allowing efficient computation, Equation 7 can be replaced by Equations 8 and 9, which are computed in a mutually recursive manner to yield an equivalent computation (Figure 2(b) and Figure 2(c), respectively).

$$ed(s,t) = \min \left\{ \begin{array}{l} ed(s,\alpha) + ed(\alpha,t), \\ edt^\alpha\left(s^a, t\right) + ed\left(s^b,\alpha\right) \end{array} \;\middle|\; \begin{array}{c} \left(s^a, s^b\right) \in P(s) \\ \alpha \in \Sigma \end{array} \right\}$$

(s and t are of lengths ≥ 2)

(8)

$$edt^\alpha(s,t) = \min \left\{ ed\left(s, t^a\right) + ed\left(\alpha, t^b\right) \;\middle|\; \left(t^a, t^b\right) \in P(t) \right\}$$

(t is of lengths ≥ 2)

(9)

Figure 2 Edit distance computation when both input strings are of length at least 2. Chart **(a)** illustrates Equation 7, while charts **(b)** and **(c)** illustrate its decomposition into Equations 8 and 9, respectively.

All base-cases of the above recursive equations are implied from Property 1 in a straightforward manner.

Theorem 1. *EDDC is correctly solved by Equations 1-9.*

The proof of Theorem 1 appears in Appendix "Correctness of the recursive computation".

A baseline dynamic-programming algorithm for EDDC

In this section, we describe a DP algorithm implementing the recursive EDDC computation given by Equations 1 to 9, which is the basis for improvements introduced later in this paper.

Let s and t be the input source and target strings, respectively, and for simplicity assume both strings are of length n. The algorithm maintains the following matrices for storing solutions to sub-instances of the input which occur along the recursive computation. All matrices are of size $(n+1) \times (n+1)$, with row and column indices in the range $0, 1, 2, \ldots, n$.

- For every $\alpha \in \Sigma$, the algorithm maintains matrices S'^α, S^α, T'^α, and T^α. Entries $S'^\alpha[k, i]$, $S^\alpha[k, i]$, $T'^\alpha[l, j]$ and $T^\alpha[l, j]$ are used for storing the values $ed'(s_{k,i}, \alpha)$,

  $ed(s_{k,i}, \alpha)$, $ed'(\alpha, t_{l,j})$, and $ed(\alpha, t_{l,j})$, respectively.
- Two matrices S^ε and T^ε, whose entries $S^\varepsilon[k, i]$ and $T^\varepsilon[l, j]$ are used for storing values of the forms $ed(s_{k,i}, \varepsilon)$ and $ed(\varepsilon, t_{l,j})$, respectively.
- For every $\alpha \in \Sigma$, a matrix EDT^α whose entries $EDT^\alpha[i, j]$ are used for storing the values $edt^\alpha(s_{0,i}, t_{0,j})$.
- A matrix ED, whose entries $ED[i, j]$ are used for storing the values $ed(s_{0,i}, t_{0,j})$.

The algorithm consists of two stages: Stage 1 computes solutions to all sub-instances in which one of the substrings is either empty or single-lettered, applying Equations 1 to 6. Stage 2 uses the values computed in Stage 1 in order to compute all prefix-to-prefix solutions $ed(s_{0,i}, t_{0,j})$ and $edt^\alpha(s_{0,i}, t_{0,j})$ according to Equations 8 and 9. In particular, Stage 2 computes the edit distance $ed(s_{0,n}, t_{0,n}) = ed(s, t)$ between the two complete strings. The entries are traversed in an order which guarantees that upon computing each entry, all solutions to subinstances appearing on the right-hand side of the relevant equation are already computed and contained in the corresponding entries. Algorithm 1 gives the pseudo-code for this computation.

Algorithm 1: BASELINE-EDDC(s, t)

---

1 Let n be the length of s and t. Allocate $(n+1) \times (n+1)$ matrices S'^α, S^α, T'^α, T^α, and EDT^α for every $\alpha \in \Sigma$, and three $(n+1) \times (n+1)$ matrices S^ε, T^ε, and ED.

   // Stage 1

2 For every $0 \leq i < n$, set $T^\varepsilon[i, i+1]$ to $ins(t_i)$, and for every $\alpha \in \Sigma$ set $T^\alpha[i, i+1]$ to $mut(\alpha, t_i)$.

3 **for** $j = 2, 3, \ldots, n$ **do**

4   **for** $i = j-2, j-3, \ldots, 0$ **do**

5   For every $\alpha \in \Sigma$, set $T'^\alpha[i, j] \overset{\text{Eq.3}}{\leftarrow} \min \left\{ \begin{array}{l} T^\alpha[i, h] + T^\varepsilon[h, j], \\ T^\varepsilon[i, h] + T^\alpha[h, j], \\ dup(\alpha) + T^\alpha[i, h] + T^\alpha[h, j] \end{array} \middle| \; i < h < j \right\}$.

6   For every $\alpha \in \Sigma$, set $T^\alpha[i, j] \overset{\text{Eq.2}}{\leftarrow} \min \left\{ mut(\alpha, \beta) + T'^\beta[i, j] \mid \beta \in \Sigma \right\}$.

7   Set $T^\varepsilon[i, j] \overset{\text{Eq.1}}{\leftarrow} \min \left\{ ins(\alpha) + T^\alpha[i, j] \mid \alpha \in \Sigma \right\}$.

8 Fill similarly all matrices S'^α, S^α, and S^ε using Equations 4 to 6.

   // Stage 2

9 Set $ED[0, 0]$ to 0. For every $0 < i \leq n$, set $ED[0, i]$ to $T^\varepsilon[0, i]$, set $ED[1, i]$ to $T^\alpha[0, i]$ for $\alpha = s_0$, set $ED[i, 0]$ to $S^\varepsilon[0, i]$, and set $ED[i, 1]$ to $S^\alpha[0, i]$ for $\alpha = t_0$.

10 **for** $j = 2, 3, \ldots, n$ **do**

11   **for** $i = 2, 3, \ldots, n$ **do**

12   For every $\alpha \in \Sigma$, set $EDT^\alpha[i, j] \overset{\text{Eq.9}}{\leftarrow} \min \left\{ ED[i, h] + T^\alpha[h, j] \mid 0 < h < j \right\}$.

13   Set $ED[i, j] \overset{\text{Eq.8}}{\leftarrow} \min \left\{ \begin{array}{l} S^\alpha[0, i] + T^\alpha[0, j], \\ EDT^\alpha[h, j] + S^\alpha[h, i] \end{array} \middle| \begin{array}{l} {\scriptstyle 0 < h < i} \\ {\scriptstyle \alpha \in \Sigma} \end{array} \right\}$.

14 **return** $ED[n, n]$.

Complexity analysis of Algorithm 1

The running time of Algorithm 1 is dictated by the total time required to compute all entries in the DP matrices. Each entry is computed according to one of the recursive equations, where the number of operations in such a computation depends on the number of expressions examined on the right-hand side of the corresponding recursive equation. Note that the value of each examined expression is obtained in a constant time, by querying previously computed values stored in the matrices.

The computation of each entry in matrices T^ε and S^ε and in matrices of the form T^α and S^α takes $O(|\Sigma|)$ time, due to Equations 1, 4, 2, and 5, respectively. As there are $O(|\Sigma|)$ such matrices and each matrix contains $O(n^2)$ entries, their overall computation time is $O(|\Sigma|^2 n^2)$. The computation of entries in T'^α and S'^α take $O(n)$ time, due to Equations 3 and 6, respectively. There are $O(|\Sigma|)$ such matrices, each of size $O(n^2)$, and so the total time for computing all entries in these matrices is $O(|\Sigma|n^3)$. According to Equation 9, computing each entry of the form $EDT^\alpha[i,j]$ takes $O(n)$ time, and as there are $O(|\Sigma|n^2)$ such entries the total time for computing all these entries is $O(|\Sigma|n^3)$. According to Equation 8, computing each entry of the form $ED[i,j]$ takes $O(|\Sigma|n)$ time, and since there are $O(n^2)$ such entries, the total time for computing all these entries is again $O(|\Sigma|n^3)$. Thus, the total running time of the algorithm is $O(|\Sigma|n^3 + |\Sigma|^2 n^2)$. Under the assumption that $|\Sigma| = O(n)$, the time is $O(|\Sigma|n^3)$. The algorithm requires $O(|\Sigma|n^2)$ space for maintaining the DP matrices.

A matrix multiplication based algorithm for EDDC

In previous work by the authors [11], Vector Multiplication Templates (VMTs) were identified as templates for computational problems sustaining certain properties, such that algorithms for solving these problems can be accelerated using efficient matrix multiplication subroutines (similarly to Valiant's algorithm for CFG recognition [10]). Intuitively, standard algorithms for VMT problems perform computations that can be expressed in terms of vector multiplications, and these computations can be computed and combined more efficiently using efficient matrix multiplications. In this section, we show that EDDC exhibits such VMT properties, and formulate a new algorithm that incorporates matrix-matrix min-plus multiplications. This algorithm yields a better running time than that of the baseline algorithm in the previous section.

Notations for matrices

For two integers p, q such that $p \leq q$, $I_{p,q}$ denotes the interval of integers $I_{p,q} = [p, p+1, \ldots, q-1]$. We use the notation $A_{n \times m}$ to imply that the matrix A has n rows and m columns, and say that A has the dimensions $n \times m$ (rows and column indices start at 0). For a subset of row indices I and a subset of column indices J, denote by $I \times J$ the *region* which contains all pairs of indices (i,j), such that $i \in I$ and $j \in J$. Define $A[I, J]$ to be the submatrix of A, which is induced by all entries in the region $I \times J$. When I contains a single row i or J contains a single column j, we simplify the notation and write $A[i, J]$ or $A[I, j]$, respectively.

Define the following operations on matrices. Let $tr(\cdot)$ denote the *transpose* operation for matrices. For a set of matrices $\mathcal{A} = \{A^1, A^2, \ldots, A^r\}$ all of the same dimensions $n \times m$, denote by $\min\{\mathcal{A}\}$ the *entry-wise min* operation over $\mathcal{A}$, whose result is a matrix $C_{n \times m}$, such that $C[i, j] = \min\{A[i, j] \mid A \in \mathcal{A}\}$. $\min\{\mathcal{A}\}$ can be computed in $O(|\mathcal{A}|nm)$ time in a straightforward manner. For matrices $A_{n \times k}$ and $B_{k \times m}$, the *min-plus multiplication* of A and B, denoted $A \otimes B$, results in a matrix $C_{n \times m}$, where the entries of C are defined by $C[i, j] = \min\{A[i, h] + B[h, j] \mid 0 \leq h < k\}$. Naively, $A \otimes B$ can be computed in $O(nkm)$ operations. Denote the time complexity of a min-plus multiplication of two $n \times n$ matrices by $MP(n)$. At present, the asymptotically fastest algorithm for min-plus square matrix multiplication is that of Chan [12], taking $O\left(\frac{n^3 \log^3 \log n}{\log^2 n}\right)$ time.

In the following observation, we point out how matrix multiplication can be computed as a composition of two parts, where each of the items (1-3) in the observation addresses a partitioning in one of the three dimensions. This will be later used by our recursive computation which is based on such partitioning.

Observation 1. *Let $A_{n \times k}, B_{k \times m}$ and $C_{n \times m}$ be matrices, such that $C = A \otimes B$ (see Figure 3).*

1. *For every $0 \leq h < m$, $C[I_{0,n}, I_{0,h}] = A \otimes B[I_{0,k}, I_{0,h}]$ and $C[I_{0,n}, I_{h,m}] = A \otimes B[I_{0,k}, I_{h,m}]$.*
2. *For every $0 \leq h < n$, $C[I_{0,h}, I_{0,m}] = A[I_{0,h}, I_{0,k}] \otimes B$ and $C[I_{h,n}, I_{0,m}] = A[I_{h,n}, I_{0,k}] \otimes B$.*
3. *For every $0 \leq h < k$,*
   $C = \min\{A[I_{0,n}, I_{0,h}] \otimes B[I_{0,h}, I_{0,m}], A[I_{0,n}, I_{h,k}] \otimes B[I_{h,k}, I_{0,m}]\}$.

EDDC expressed via min-plus vector multiplications

The key observation that enables a further improvement of the worst-case bounds of EDDC is that Equations 3, 6, 8, and 9 can be expressed in terms of *min-plus vector multiplications*. Under the assumption that all solutions to sub-instances appearing on the right-hand side of the equations are computed and stored in the corresponding entries, these equations can be written as follows:
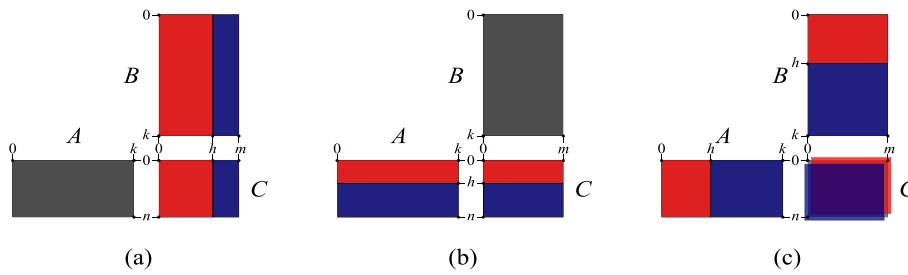
Figure 3 Decomposition of a matrix-multiplication. In all three charts, the matrix C is the result of the multiplication $A \otimes B$. Charts **(a)**, **(b)**, and **(c)** illustrate items 1, 2, and 3 of Observation 1, respectively.

$$ed'\left(\alpha, t_{l,j}\right) \overset{\text{Eq.3}}{=} \min \left\{ \begin{array}{l} T^\alpha[l,h] + T^\varepsilon[h,j], \\ T^\varepsilon[l,h] + T^\alpha[h,j], \\ dup(\alpha) + T^\alpha[l,h] + T^\alpha[h,j] \end{array} \middle| \; l < h < j, \right\}$$

$$= \min \left\{ \begin{array}{l} T^\alpha[l, I_{l+1,j}] \otimes T^\varepsilon[I_{l+1,j}, j], \\ T^\varepsilon[l, I_{l+1,j}] \otimes T^\alpha[I_{l+1,j}, j], \\ dup(\alpha) + T^\alpha[l, I_{l+1,j}] \otimes T^\alpha[I_{l+1,j}, j] \end{array} \right\},$$

(10)

$$ed'\left(s_{k,i}, \alpha\right) \overset{\text{Eq.6}}{=} \min \left\{ \begin{array}{l} S^\alpha[k,h] + S^\varepsilon[h,i], \\ S^\varepsilon[k,h] + S^\alpha[h,i], \\ S^\alpha[k,h] + S^\alpha[h,i] + cont(\alpha) \end{array} \middle| \; k < h < i \right\}$$

$$= \min \left\{ \begin{array}{l} S^\alpha[k, I_{k+1,i}] \otimes S^\varepsilon[I_{k+1,i}, i], \\ S^\varepsilon[k, I_{k+1,i}] \otimes S^\alpha[I_{k+1,i}, i], \\ S^\alpha[k, I_{k+1,i}] \otimes S^\alpha[I_{k+1,i}, i] + cont(\alpha) \end{array} \right\},$$

(11)

$$ed\left(s_{0,i}, t_{0,j}\right) \overset{\text{Eq.8}}{=} \min \left\{ \begin{array}{l} S^\alpha[0,i] + T^\alpha[0,j], \\ EDT^\alpha[k,j] + S^\alpha[k,i] \end{array} \middle| \begin{array}{l} 0 < k < i \\ \alpha \in \Sigma \end{array} \right\}$$

$$= \min \left\{ \begin{array}{l} S^\alpha[0,i] + T^\alpha[0,j], \\ tr(S^\alpha)[i, I_{1,i}] \otimes EDT^\alpha[I_{1,i}, j] \end{array} \middle| \; \alpha \in \Sigma \right\},$$

(12)

$$edt^\alpha\left(s_{0,i}, t_{0,j}\right) \overset{\text{Eq.9}}{=} \min \left\{ ED[i,l] + T^\alpha[l,j] \mid 0 < l < j \right\}$$

(13)

$$= ED[i, I_{1,j}] \otimes T^\alpha[I_{1,j}, j].$$

The algorithm

The new algorithm has the same two stages as the baseline algorithm. It can be observed that the computation of all matrices of the forms S'^α, S^α, S^ε, T'^α, T^α, and T^ε
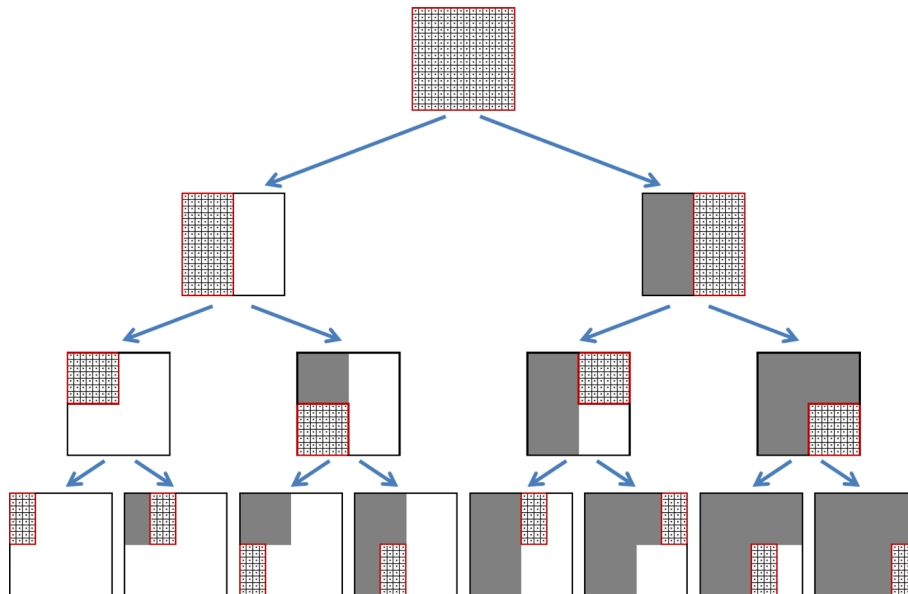
Figure 4 The tree of recursive calls to COMPUTE-MATRIX. Each call over a region containing more than one entry partitions the region either vertically or horizontally, and performs two recursive calls over the two sub-regions. After the first call was performed (the left or top sub-region for vertical or horizontal partition, respectively), a matrix multiplication involving the computed region is computed in order to meet the precondition required for the computation of the sibling region.

performed in Stage 1 of the baseline algorithm adhere to the *Inside-VMT requirements* as given in Definition 1 in [11]. The application of the generic *Inside-VMT algorithm* [11] to this computation is immediate, and therefore we focus only on adapting the method to the computation of matrices of the form EDT^α and ED conducted in Stage 2 of the baseline algorithm.

After allocating all dynamic programming matrices and performing Stage 1 of the algorithm, the COMPUTE-MATRIX procedure is used for implementing Stage 2

(see Algorithm 2 and Figure 4). This is a divide-and-conquer recursive procedure that accepts a region $I \times J$ and computes the values in all entries of ED and EDT^α within the region. The procedure partitions the given region into two parts and performs recursive calls on each part. In order to maintain a required precondition, the procedure applies min-plus matrix multiplication subroutines between recursive calls. The correctness proof of Algorithm 2 appears in Appendix "Correctness of Algorithm 2".

Algorithm 2: MATRIX-EDDC(s, t)

---

1 Let n be the length of s and t. Allocate $(n + 1) \times (n + 1)$ matrices S'^α, S^α, T'^α, T^α, and EDT^α for every $\alpha \in \Sigma$, and three $(n + 1) \times (n + 1)$ matrices S^ε, T^ε, and ED.
    // Stage 1
2 Fill all matrices T^ε, T^α, T'^α, S^ε, S^α, and S'^α applying the generic Inside-VMT algorithm of [11], using Equations 1, 2, 10, 4, 5, and 11, correspondingly.
    // Stage 2
3 Initialize entries in all matrices EDT^α and ED which correspond to base-case instances as describe in Algorithm 1. This includes all entries in the first two rows and first two columns in these matrices.
4 Set $EDT^\alpha[I_{2,n+1}, I_{2,n+1}] \leftarrow ED[I_{2,n+1}, 1] \otimes T^\alpha[1, I_{2,n+1}]$ for every $\alpha \in \Sigma$, and set
$ED[I_{2,n+1}, I_{2,n+1}] \leftarrow \min \{ tr(S^\alpha)[I_{2,n+1}, 1] \otimes EDT^\alpha[1, I_{2,n+1}] \mid \alpha \in \Sigma \}$.
5 Run COMPUTE-MATRIX($I_{2,n+1}, I_{2,n+1}$).
6 **return** $ED[n, n]$.

Procedure: COMPUTE-MATRIX($I_{i,k}, I_{j,l}$)

Precondition: $2 \le i < k$, $2 \le j < l$, and all entries in matrices S^α and T^α, as well as all entries in submatrices $EDT^\alpha[I_{0,i}, I_{j,l}]$ and $ED[I_{i,k}, I_{0,j}]$, contain the solutions for the corresponding sub-instances. In addition, $EDT^\alpha[I_{i,k}, I_{j,l}] = ED[I_{i,k}, I_{1,j}] \otimes T^\alpha[I_{1,j}, I_{j,l}]$, and
$ED[I_{i,k}, I_{j,l}] = \min \{ tr(S^\alpha)[I_{i,k}, I_{1,i}] \otimes EDT^\alpha[I_{1,i}, I_{j,l}] \mid \alpha \in \Sigma \}$.

Postcondition: All entries in the region $I_{i,k} \times I_{j,l}$ in matrices EDT^α and ED contain the solutions for the corresponding sub-instances.

1 **if** $k = i + 1$ *and* $l = j + 1$ **then**
2   Set $ED[i, j] \overset{\substack{\text{precondition,} \\ \text{Eq.12}}}{\leftarrow} \min \{ ED[i, j], S^\alpha[0, i] + T^\alpha[0, j] \mid \alpha \in \Sigma \}$
3 **else**
4   **if** $l - j \ge k - i$ **then**
         // vertical partitioning
5   Let $h = \lceil \frac{j+l}{2} \rceil$
6   Run COMPUTE-MATRIX($I_{i,k}, I_{j,h}$)
7   Update $EDT^\alpha[I_{i,k}, I_{h,l}] \leftarrow \min \{ EDT^\alpha[I_{i,k}, I_{h,l}], ED[I_{i,k}, I_{j,h}] \otimes T^\alpha[I_{j,h}, I_{h,l}] \}$ for every $\alpha \in \Sigma$
8   Run COMPUTE-MATRIX($I_{i,k}, I_{h,l}$)
9   **else**
         // horizontal partitioning
10   Let $h = \lceil \frac{i+k}{2} \rceil$
11   Run COMPUTE-MATRIX($I_{i,h}, I_{j,l}$)
12   Update $ED[I_{h,k}, I_{j,l}] \leftarrow \min \{ ED[I_{h,k}, I_{j,l}], \min \{ tr(S^\alpha)[I_{h,k}, I_{i,h}] \otimes EDT^\alpha[I_{i,h}, I_{j,l}] \mid \alpha \in \Sigma \} \}$
13   Run COMPUTE-MATRIX($I_{h,k}, I_{j,l}$)

Time complexity analysis

The time complexity of Algorithm 2 can be established by an identical analysis to that of the Inside-VMT algorithm of [11] (see Section 3.3.1 of [11]). For completeness, we repeat this analysis here for Stage 2 of the computation, where the time complexity of Stage 1 can be inferred similarly. The complexity is expressed as a function of the bound $MP(n)$ over the running time of a min-plus multiplication of two $n \times n$ matrices. Note that $MP(n) = \Omega(n^2)$, as the input and output matrices of the computation contain $O(n^2)$ entries. We assume here that $MP(n) = \Omega(n^{2+\delta})$ for some constant $0 < \delta \leq 1$, which is true for the current best bound over $MP(n)$ [12]. In some of the expressions developed below, we avoid the "big O" notation and give explicit bounds over the number of operations, as constant factors that may be hidden due to this notation cannot be ignored in the analysis.

The initialization time of Stage 2 is dominated by the matrix multiplications and entry-wise min operations performed in line 4 of Algorithm 2. This initialization performs $2|\Sigma|$ multiplications of matrices of dimensions $(n-1) \times 1$ with matrices of dimensions $1 \times (n-1)$, which can naively be implemented in $O(|\Sigma|n^2)$ time, and an entry-wise min operation over a set containing $|\Sigma|$ matrices of dimensions $(n-1) \times (n-1)$, which is also implemented in $O(|\Sigma|n^2)$ time.

The computation of the remaining entries in ED and EDT^α matrices is done within recursive calls to the COMPUTE-MATRIX procedure. Observe that when COMPUTE-MATRIX is called over a region of dimensions $r \times r$ for some even integer $r \geq 2$, the procedure applies a vertical partitioning and performs two recursive calls over regions of dimensions $r \times \frac{r}{2}$ (lines 6 and 8). For a call over a region of dimensions $r \times \frac{r}{2}$, the procedure applies a horizontal partitioning and performs two recursive calls over regions of dimensions $\frac{r}{2} \times \frac{r}{2}$ (lines 11 and 13). For simplicity, assume that $n - 1 = 2^p$ for some integer $p \geq 0$, and thus it follows that the dimensions of all regions occurring as inputs in recursive calls are either 1×1, or of the form $r \times r$ or $r \times \frac{r}{2}$ for some even integer r. Denote by $T(x \times y)$ an upper bound over the number of operations conducted when applying COMPUTE-MATRIX over a region of dimensions $x \times y$.

From line 2 of the procedure, $T(1 \times 1) = O(|\Sigma|)$. Consider a region of dimensions $r \times r$ for an even integer $r \geq 2$. For such a region, the code in lines 5-8 of COMPUTE-MATRIX is executed. In order to implement line 7 for some $\alpha \in \Sigma$, it is necessary to compute first a min-plus matrix multiplication $C = A \otimes B$, where the matrix $A = ED[I_{i,k}, I_{j,h}]$ is of dimensions $r \times \frac{r}{2}$, the matrix $B = T^\alpha[I_{j,h}, I_{h,l}]$ is of dimensions $\frac{r}{2} \times \frac{r}{2}$, and the resulting matrix C is of dimensions $r \times \frac{r}{2}$. Due to Observation 1, it is possible to compute independently

the upper and lower halves of C, where $C[I_{0,\frac{r}{2}}, I_{0,\frac{r}{2}}] = A[I_{0,\frac{r}{2}}, I_{0,\frac{r}{2}}] \otimes B$ and $C[I_{\frac{r}{2},r}, I_{0,\frac{r}{2}}] = A[I_{\frac{r}{2},r}, I_{0,\frac{r}{2}}] \otimes B$. The time required to conduct this computation is $2MP(\frac{r}{2})$. Then, it is required to compute $min\{EDT^\alpha[I_{i,k}, I_{h,l}], C\}$ and to update $EDT^\alpha[I_{i,k}, I_{h,l}]$ to be the result of this operation, a computation which requires at most cr^2 operations for some constant c. Since line 7 is computed for every $\alpha \in \Sigma$, the total number of applied operations due to this line is at most $|\Sigma|\left(2MP(\frac{r}{2}) + cr^2\right)$. Besides line 7, two recursive calls are made in lines 6 and 8 over regions of dimensions $r \times \frac{r}{2}$, and therefore we get

$$T(r \times r) \leq 2T\left(r \times \frac{r}{2}\right) + |\Sigma|\left(2MP\left(\frac{r}{2}\right) + cr^2\right).$$

When the procedure is called over a region of dimensions $r \times \frac{r}{2}$, the code in lines 10-13 is executed. Similarly as above, it can be shown that the computation in line 12 requires at most $|\Sigma|\left(MP(\frac{r}{2}) + \frac{cr^2}{4}\right)$ operations, and due to the two recursive calls in lines 11 and 13 over regions of dimensions $\frac{r}{2} \times \frac{r}{2}$, we get

$$T\left(r \times \frac{r}{2}\right) \leq 2T\left(\frac{r}{2} \times \frac{r}{2}\right) + |\Sigma|\left(MP\left(\frac{r}{2}\right) + \frac{cr^2}{4}\right).$$

Therefore,

$$T(r \times r) \leq 4T\left(\frac{r}{2} \times \frac{r}{2}\right) + |\Sigma|\left(4MP\left(\frac{r}{2}\right) + \frac{3cr^2}{2}\right).$$

The explicit form of the above recursive equation can be established by the Master Theorem (under the assumption that $MP(n) = \Omega(n^{2+\delta})$, see Chapter 4 in [16]), yielding the expression $T(r \times r) = O(|\Sigma|MP(r))$. Thus, the time complexity of Stage 2 of the algorithm is $O(|\Sigma|MP(n))$. The time analysis of the Inside-VMT algorithm of [11], applied to implement Stage 1 of the algorithm yields the same bound of $O(|\Sigma|MP(n))$, and thus $O(|\Sigma|MP(n))$ is the time complexity of the entire algorithm. Using the currently asymptotically fastest algorithm for min-plus matrix multiplication [12] $MP(n) = \Theta\left(\frac{n^3 \log^3 \log n}{\log^2 n}\right)$, we get the currently best explicit time bound for EDDC of $O\left(\frac{|\Sigma|n^3 \log^3 \log n}{\log^2 n}\right)$.

An online algorithm for EDDC using min-plus matrix-vector multiplication for discrete cost functions

In this section, we present an EDDC algorithm which is based on the general algorithm (given in Section "A matrix multiplication based algorithm for EDDC") and improves its time complexity by a factor of $O(\log^3 \log n)$.

This EDDC algorithm is intended for integer cost functions, but can also be applied to rational cost functions after they are scaled. It is an online algorithm; it can process the input strings letter by letter with a guaranteed low time bound for any prefix of the input. The EDDC algorithm presented in this section is based on a *D*-discrete matrix-vector min-plus multiplication algorithm we developed, which is generic and may be applied to other problems as well.

D-discrete matrices and the EDDC problem with integer costs

Given a matrix of integers $A_{n \times m}$ and indices $1 \leq i < n$ and $0 \leq j < n$, call the pair of entries $A[i-1, j]$ and $A[i, j]$ *adjacent*. Let $D = I_{a,b} = [a, a+1, \ldots, b-1]$ be an integer interval for some integers $a < b$. Say that matrix A is *D*-discrete if for every pair of adjacent entries $A[i-1, j]$ and $A[i, j]$, their difference $A[i-1, j] - A[i, j]$ is in D.

Consider the EDDC problem in the case of integer costs for all edit operations. In Lemma 1, we show that in this case, all matrix multiplications applied by Algorithm 2 are between *D*-discrete metrices, with respect to a certain integer interval D. This proof is similar to that of Masek and Paterson for simple edit distance [17]. This would allow conducting such matrix multiplications using a faster algorithm, described in Section "An efficient *D*-discrete min-plus matrix-vector multiplication algorithm".

Lemma 1. *Given strings s and t and an integer cost function for EDDC, all matrix multiplications applied by Algorithm 2 are over D-discrete matrices, where $D = I_{a,b}$ is determined according to the cost function by $a = -\max\{del(\alpha) \,|\, \alpha \in \Sigma\}$ and $b = \max\{ins(\alpha) \,|\, \alpha \in \Sigma\} + 1$.*

The proof of Lemma 1 appears in Appendix "Proofs to lemmas corresponding to the EDDC algorithm for discrete cost functions ".

D-discrete matrices and vectors

Here, we present some properties of *D*-discrete matrices and vectors that are similar to those previously observed in [14,17]. The following lemmas show that the set of *D*-discrete matrices is closed under the min-plus multiplication and entry-wise min operations. In what follows, let $D = I_{a,b}$ be some integer interval. The proofs of the following lemmas appear in Appendix "Proofs to lemmas corresponding to the EDDC algorithm for discrete cost functions ".

Lemma 2. *Let X, Y and Z be matrices, such that X and Y contain only integer elements and $Z = X \otimes Y$. If X is D-discrete, then Z is D-discrete.*

Lemma 3. *Let X, Y and Z be matrices, such that X and Y contain only integer elements and $Z = \min\{X, Y\}$. If X and Y are D-discrete, then Z is D-discrete.*

The following lemma implies that when the absolute difference between the first elements of two *q*-length *D*-discrete vectors x and y is sufficiently large, one of the vectors can be immediately taken as the result of the $\min(x, y)$ operation.

Lemma 4. *Let $x = (x_0, \ldots, x_{q-1})$ and $y = (y_0, \ldots, y_{q-1})$ be two q-length D-discrete vectors for some $q > 0$. If $y_0 - x_0 \geq q|D|$, then $\min(x, y) = x$.*

In what follows, fix an integer $q > 1$. Let $x = (x_0, x_1, \ldots, x_{q-1})$ be a *q*-length *D*-discrete vector. By definition, for every $0 < i < q$, $x_{i-1} - x_i$ is an integer within D, and so $x_{i-1} - x_i - a$ is an integer within the interval $I_{0,b-a} = I_{0,|D|}$. Therefore, the series $x_0 - x_1 - a, x_1 - x_2 - a, \ldots, x_{q-2} - x_{q-1} - a$ can be thought of as a series of $q-1$ digits in a $|D|$-base representation of an integer $\Delta x = \sum_{0 \leq i \leq q-1} |D|^i (x_i - x_{i+1} - a)$, where $0 \leq \Delta x < |D|^{q-1}$. The Δ-*encoding* of x is defined to be the pair of integers $(x_0, \Delta x)$. We write $x = (x_0, \Delta x)$ to indicate that $(x_0, \Delta x)$ is the Δ-encoding of x, where x_0 is called the *offset* of x and Δx is called the *canonical index* of x. Note that for two *q*-length *D*-discrete vectors $x = (x_0, \Delta x)$ and $y = (y_0, \Delta y)$, $\Delta x = \Delta y$ if and only if for every $0 \leq i < q$, $x_i - y_i = c$ for some constant c. In particular, x and y share the same Δ-encoding if and only if they are identical. Call a *D*-discrete vector of the form $x = (0, \Delta x)$ (with an offset $x_0 = 0$) a *canonical* vector.

The next observations show that both operations of entry-wise min and min-plus multiplication, with respect to *D*-discrete matrices and vectors, can be expressed via canonical vectors.

Observation 2. *Let $x = (x_0, \Delta x)$, $y = (y_0, \Delta y)$, and $z = (z_0, \Delta z)$ be q-length D-discrete vectors such that $z = \min(x, y)$. Then, for every number c it holds that $\min((x_0 - c, \Delta x), (y_0 - c, \Delta y)) = (z_0 - c, \Delta z)$. In particular, $\min((0, \Delta x), (y_0 - x_0, \Delta y)) = (z_0 - x_0, \Delta z)$.*

Observation 3. *Let $B_{q \times q}$ be a D-discrete matrix, $x = (0, \Delta x)$ a q-length canonical D-discrete vector, and $y = (y_0, \Delta y)$ a q-length D-discrete vector, such that $B \otimes x = y$. Then, for any number c it holds that $B \otimes (c, \Delta x) = (y_0 + c, \Delta y)$.*

An efficient *D*-discrete min-plus matrix-vector multiplication algorithm

Let $A_{n \times m}$ be a *D*-discrete matrix, and fix a constant $\frac{1}{\log_{|D|}(n+m)} < \lambda < 1$. We give an algorithm for min-plus

D-discrete matrix-vector multiplication that, after preprocessing A in $O\left(\frac{nm(n+m)^\lambda}{|D|}\right)$ time and $O\left(\frac{nm(n+m)^\lambda}{|D|\lambda^2 \log_{|D|}^2(n+m)}\right)$ space, computes $A \otimes x$ for any m-length D-discrete vector x in $O\left(\frac{nm}{\lambda^2 \log_{|D|}^2(n+m)}\right)$ time under the RAM computational model. Our algorithm is an adaptation of Williams' algorithm [13] for finite semiring matrix-vector multiplications, with some notions similar to Frid and Gusfield's acceleration technique for RNA folding [14]. It follows the concept of the *Four-Russians Algorithm* [18] (see also [14,17,19]), i.e. preprocessing reoccurring computations, tabulating their results in lookup tables, and retrieving such results in order to accelerate the general computation.

Specifically, the algorithm stores preprocessed computations of two kinds: matrix-vector min-plus multiplications, and vector entry-wise minima, where vectors and matrices are of q-length and of $q \times q$ dimensions, respectively, for $q = \left\lfloor \lambda \log_{|D|}(n+m) \right\rfloor$. For conducting this preprocessing, we will assume that $|D| \le n + m$, otherwise $q = 0$ and the multiplication cannot be accelerated using the suggested method. In addition, for simplicity of the analysis we assume that $q^3 \le \min(n,m)$. If this does not hold, a multiplication of the form $A \otimes x$ can be naively computed in the relatively efficient time complexity of $O\left(\max(n,m) \log_{|D|}^3(n+m)\right)$. The space complexity of the preprocessing phase is higher than the $O(nm)$ space complexity of the standard multiplication algorithm and depends on the constant λ, ranging between $O(nm|D|)$ and $O\left(\frac{nm(n+m)}{\log_{|D|}^2(n+m)}\right)$ for λ values between $\frac{1}{\log_{|D|}(n+m)}$ and 1, correspondingly. The lower bound $\frac{1}{\log_{|D|}(n+m)}$ for λ is chosen so that the time complexity $O\left(\frac{nm}{\lambda^2 \log_{|D|}^2(n+m)}\right)$ of matrix-vector multiplications involving the preprocessed matrix would be better than the naive time complexity $O(nm)$.

Preprocessing of matrix-vector $\otimes$ computations

Let $n' = q\left\lfloor \frac{n}{q} \right\rfloor$ and $m' = q\left\lfloor \frac{m}{q} \right\rfloor$, and note that $0 \le n - n' < q$ and $0 \le m - m' < q$. Let Q_k denote the q-length integer interval $Q_k = [kq, kq + 1, \ldots, (k+1)q - 1]$. The sub-matrix $A' = A\left[I_{0,n'}, I_{0,m'}\right]$ is decomposed into $\frac{n'm'}{q^2}$ blocks $B_{i,j} = A\left[Q_i, Q_j\right]$ where $i = 0, 1, \ldots, \frac{n'}{q} - 1$ and $j = 0, 1, \ldots, \frac{m'}{q} - 1$. For each block B, a corresponding lookup table MUL_B is created, which tabulates min-plus multiplications between B and all canonical q-length D-discrete vectors. For the canonical vector $x = (0, \Delta x)$, the result $y = B \otimes x$ is stored in the entry $MUL_B[\Delta x]$ by its encoding $(y_0, \Delta y)$ (by

Lemma 2, y is also D-discrete and thus can be encoded accordingly).

The multiplication of a $q \times q$ block with a q-length vector can be done in $O(q^2)$ time in a straightforward manner and the encoding of the resulting q-length vector requires additional $O(q)$ time. There are $\frac{n'm'}{q^2}$ blocks in the decomposition of A', each is multiplied by $|D|^{q-1}$ canonical vectors, and so the total time required for computing these multiplications is $O\left(q^2|D|^{q-1}\frac{nm}{q^2}\right) = O\left(|D|^{q-1}nm\right) = O\left(\frac{nm(n+m)^\lambda}{|D|}\right)$.

Let $(y_0, \Delta y)$ be the Δ-encoding of some result $y = B \otimes x$ computed in the preprocessing of A' as described above. Note that $y_0 = \min_{0 \le i < q}\{B[0,i] + x[i]\} \le \min_{0 \le i < q}\{2\max(B[0,i], x[i])\}$. Therefore, the number of bits in the binary representation of y_0 is at most one plus the maximum number of bits required for the representations of $B[0,i]$ and $x[i]$ for some $0 \le i < q$. Also, note that $0 \le \Delta y < |D|^{q-1} = \frac{(n+m)^\lambda}{|D|}$, and Δy can be represented in $O(\log(n+m))$ bits. Thus, under the RAM computational model assumptions, each such encoding $(y_0, \Delta y)$ requires $O(1)$ space units and can be written and read in a constant time, and therefore the overall space complexity for maintaining all MUL_B tables is $O\left(\frac{|D|^{q-1}nm}{q^2}\right) = O\left(\frac{nm(n+m)^\lambda}{|D|\lambda^2 \log_{|D|}^2(n+m)}\right)$. In addition, given a canonical index Δx, it is possible to retrieve the encoding $(y_0, \Delta y) = B \otimes (0, \Delta x)$ stored in the entry $MUL_B[\Delta x]$ in a constant time.

Let $x = (x_0, \Delta x)$ be some (not necessarily canonical) q-length D-discrete vector, for which we wish to compute $B \otimes x$. Due to Observation 3, the multiplication result can be obtained in constant time by retrieving $(y_0, \Delta y) = MUL_B[\Delta x]$, and returning the encoding $(y_0 + x_0, \Delta y)$.

Preprocessing of vector entry-wise min computations

The algorithm constructs a lookup table MIN, storing entry-wise min calculations between every canonical q-length D-discrete vector $x = (0, \Delta x)$ and every q-length D-discrete vector $y = (y_0, \Delta y)$ such that $abs(y_0) < q|D|$ (here $abs(y_0)$ denotes the absolute value of y_0). For every such x and y, the table entry $MIN[\Delta x, y_0, \Delta y]$ stores the Δ-encoding $(z_0, \Delta z)$ of the vector $z = \min(x, y)$ (due to Lemma 3, z is D-discrete and can be encoded accordingly). There are $O\left(q|D||D|^{2(q-1)}\right) = O\left(\frac{(n+m)^{2\lambda}\lambda \log_{|D|}(n+m)}{|D|}\right)$ entries in the table MIN, and each entry can be computed in $O(q) = O\left(\lambda \log_{|D|}(n+m)\right)$ time. Thus, the computation of all entries in MIN requires $O\left(\frac{(n+m)^{2\lambda}\lambda^2 \log_{|D|}^2(n+m)}{|D|}\right)$ time, and the table occupies $O\left(\frac{(n+m)^{2\lambda}\lambda \log_{|D|}(n+m)}{|D|}\right)$ space.

Given two encoded q-length D-discrete vectors $x = (x_0, \Delta x)$ and $y = (y_0, \Delta y)$, the encoding $(z_0, \Delta z)$ of the vector $z = \min(x, y)$ can now be obtained in a constant time as follows: $(z_0, \Delta z) = (x_0, \Delta x)$ if $y_0 - x_0 \geq q|D|$ or $(z_0, \Delta z) = (y_0, \Delta y)$ if $x_0 - y_0 \geq q|D|$, due to Lemma 4. Otherwise, $|y_0 - x_0| < q|D|$, and for the vectors $x' = (0, \Delta x)$, $y' = (y_0 - x_0, \Delta y)$, and $z' = (z'_0, \Delta z') = \min(x', y')$, we have that $(z', \Delta z') = MIN[\Delta x, y_0 - x_0, \Delta y]$. From Observation 2, $(z_0, \Delta z) = (z'_0 + x_0, \Delta z')$.

Computing matrix-vector multiplications

Given an m-length D-discrete vector x and assuming the preprocessing of matrix $A_{n \times m}$ was preformed as described above, we next explain how to efficiently compute the vector $y = A \otimes x$. Note that y is an n-length D-discrete vector, due to Lemma 2.

Our algorithm computes first the multiplication $y' = A' \otimes x[I_{0,m'}]$ in parts of length q. First, for every $0 \leq j < \frac{m'}{q}$, the algorithm computes the encoding $\left(x^j_0, \Delta x^j\right)$ of the sub-vector $x^j = x[Q_j]$ of x. These encodings can be obtained in a total time of $O(m)$. Then, for every $0 \leq i < \frac{n'}{q}$, the encoding $\left(y^i_0, \Delta y^i\right)$ of the sub-vector $y^i = A'[Q_i, I_{0,m'}] \otimes x[I_{0,m'}]$ of y' is computed independently of the other sub-vectors of y'. By definition (see Figure 5),
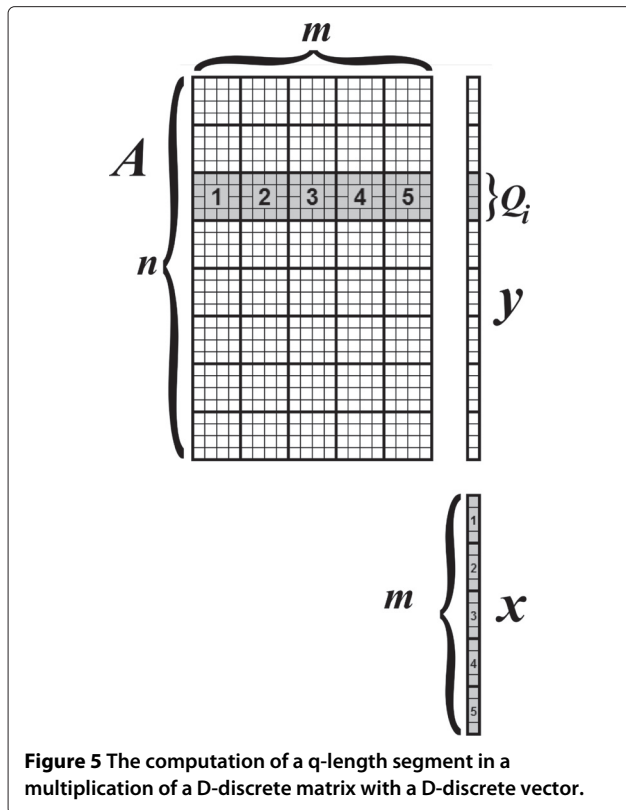

Figure 5 The computation of a q-length segment in a multiplication of a D-discrete matrix with a D-discrete vector.

$$y^i = \min\left\{ A'[Q_i, Q_j] \otimes x[Q_j] \mid 0 \leq j < \frac{m'}{q} \right\}$$

$$= \min\left\{ B_{i,j} \otimes x^j \mid 0 \leq j < \frac{m'}{q} \right\}.$$

The encoded result $\left(z^{i,j}_0, \Delta z^{i,j}\right)$ of each multiplication $z^{i,j} = B_{i,j} \otimes x^j$ can be obtained in a constant time as explained in Section "Preprocessing of matrix-vector $\otimes$ computations". As there are $\frac{m'}{q}$ such terms to compute with respect to y^i, their total computation time is $O\left(\frac{m}{q}\right)$. In addition, the entry-wise min over all these terms can be computed by initializing $\left(y^i_0, \Delta y^i\right) \leftarrow \left(z^{i,0}_0, \Delta z^{i,0}\right)$, and iteratively updating $\left(y^i_0, \Delta y^i\right) \leftarrow \min\left(\left(y^i_0, \Delta y^i\right), \left(z^{i,j}_0, \Delta z^{i,j}\right)\right)$ for all $0 < j < \frac{m'}{q}$. Each such update is computed in a constant time as described in Section "Preprocessing of vector entry-wise min computations", and so the encoding of a single segment y^i in y' is computed in a total time of $O\left(\frac{m}{q}\right)$, and the encodings of all $O\left(\frac{n}{q}\right)$ such segments are computed in $O\left(\frac{nm}{q^2}\right)$ time. Decoding all encoded vectors y^i can be done in additional $O(n)$ operations, obtaining an explicit form of y' in a total time of $O\left(\frac{nm}{q^2}\right)$.

Let $y'' = A[I_{0,n'}, I_{m',m}] \otimes x[I_{m',m}]$, where from Observation 1, $y[I_{0,n'}] = \min(y', y'')$. The computation of y'' can be conducted in $O(nq)$ time in a straightforward manner, and the computation of $\min(y', y'')$ requires additional $O(n)$ time. In addition, $y[I_{n',n}] = A[I_{n',n}, I_{0,m}] \otimes x$, where this computation can be done naively in $O(mq)$ time, and so the overall running time for computing y is

$$O\left(\frac{nm}{q^2} + nq + mq\right) = O\left(\frac{nm}{q^2}\right) = O\left(\frac{nm}{\lambda^2 \log^2_{|D|}(n+m)}\right).$$

The above matrix-vector min-plus multiplication algorithm can be used as a fast square matrix-matrix multiplication algorithm in a straightforward manner. For two D-discrete matrices $A_{n \times n}$ and $B_{n \times n}$, the computation of $C = A \otimes B$ can be conducted by first preprocessing A as described in Sections "Preprocessing of matrix-vector $\otimes$ computations" and "Preprocessing of vector entry-wise min computations" in $O\left(\frac{n^{2+\lambda}}{|D|}\right)$ time and $O\left(\frac{n^{2+\lambda}}{|D|\lambda^2 \log^2_{|D|} n}\right)$ space, and then computing each column j of C independently by multiplying A with the j-th column of B, in $O\left(\frac{n^2}{\lambda^2 \log^2_{|D|} n}\right)$ time as explained above. The total computation time of all n columns of C is therefore $O\left(\frac{n^3}{\lambda^2 \log^2_{|D|} n}\right)$.

Online preprocessing of *D*-discrete matrices

In the previous section, we assumed the settings in which a *D*-discrete matrix is given, and that it is preprocessed once prior to any multiplication operation. Next, we describe how to maintain the required lookup tables for the case the input matrix is dynamic, acquiring additional rows and columns. Consider a streaming computational model, which begins with an initial empty matrix $A_{0 \times 0}^0$. In each step r, the current matrix $A_{n_r \times m_r}^r$ is obtained from the previous matrix $A_{n_{r-1} \times m_{r-1}}^{r-1}$ by either adding an m_{r-1}-length vector as the last row or adding an n_{r-1}-length vector as the last column in the matrix. Note that $n_r + m_r = r$, and therefore the preprocessing block length corresponding to A^r is $q = \left\lfloor \lambda \log_{|D|}(n_r + m_r) \right\rfloor = \left\lfloor \lambda \log_{|D|}(r) \right\rfloor$. For the purpose of this analysis, we assume that $\lambda \leq 0.5$ (note that this does not limit the asymptotic upper bounds of the running time). This assumption implies the following inequality

$$|D|^{\frac{1}{\lambda}} \geq 2^2 = 4. \tag{14}$$

Lookup tables corresponding to intermediate matrices along the series can be maintained as follows. Let r_0 and r_1 be the smallest integers such that the block sizes corresponding to A^{r_0} and A^{r_1} are q and $q + 1$, respectively. Assume that upon reaching A^{r_0} in the matrix sequence, all required lookup tables with respect to A^{r_0} are already computed. Along the series of steps $r_0, r_0 + 1, \ldots, r_1$, we distribute two kinds of computations: (1) new MUL_B tables for accumulated $q \times q$ blocks in matrices A^r for $r_0 \leq r < r_1$, and (2) a new MIN table, as well as new MUL_B tables, with respect to block length $q + 1$.

(1) Computing MUL_B tables for accumulated $q \times q$ blocks. Assume that for some $r_0 \leq r < r_1$, a column was added to the matrix at step r so that the number of columns m_r in the intermediate matrix A^r is divisible by q. Thus, at most $\frac{n_r}{q} \leq \frac{r}{q}$ new $q \times q$ complete blocks are now available for preprocessing. The computation of lookup tables of the form MUL_B corresponding to these new blocks will be equally distributed along the series of q consecutive steps $r, r + 1, \ldots, r + q - 1$, during which it is guaranteed that no column addition would introduce new complete $q \times q$ blocks in the matrix. As shown in Section "Preprocessing of matrix-vector $\otimes$ computations", the time required for processing a single $q \times q$ block is $O\left(q^2 |D|^{q-1}\right)$, and so the total time for processing all $O\left(\frac{r}{q}\right)$ blocks is $O\left(qr |D|^{q-1}\right)$. Thus, in each step among the q steps, there is a need to perform $O\left(r |D|^{q-1}\right) = O\left(\frac{r^{1+\lambda}}{|D|}\right)$ operations due to these computations. Symmetrically, computing lookup tables corresponding to new blocks added due to the accumulation

of rows can be performed by conducting $O\left(\frac{r^{1+\lambda}}{|D|}\right)$ operations per step r.

(2) Computing a new MIN lookup table and new MUL_B tables with respect to block length $q + 1$. By the selection of r_0 and r_1, $q - 1 = \left\lfloor \lambda \log_{|D|}(r_0 - 1) \right\rfloor > \lambda \log_{|D|}(r_0 - 1) - 1$, and $q + 1 = \left\lfloor \lambda \log_{|D|}(r_1) \right\rfloor \leq \lambda \log_{|D|}(r_1)$. Therefore, $\log_{|D|}(r_1) > \log_{|D|}(r_0 - 1) + \frac{1}{\lambda}$, and so $r_1 > (r_0 - 1)|D|^{\frac{1}{\lambda}} \geq r_0 \frac{|D|^{\frac{1}{\lambda}}}{2} \overset{\text{Eq. 14}}{\geq} 2r_0$. In particular, $r_2 = \frac{r_1}{2}$ satisfies $r_0 < r_2 < r_1$, and for every $r_2 \leq r < r_1$ we have that $O(r) = O(r_1)$. The computation of the table MIN and tables of the form MUL_B with respect to block length $q + 1$ is distributed along the series of $\frac{r_1}{2}$ steps $r_2, r_2 + 1, \ldots, r_1$.

The new MIN table is computed independently from the specific input instance, and its overall computation time is $O\left(\frac{r_1^{2\lambda} \lambda^2 \log_{|D|}^2(r_1)}{|D|}\right)$ (see Section "Preprocessing of vector entry-wise min computations"). By distributing this computation evenly along all $O(r_1)$ steps, the computation time required for each step $r_2 \leq r < r_1$ is $O\left(\frac{r_1^{2\lambda - 1} \lambda^2 \log_{|D|}^2(r_1)}{|D|}\right) = O\left(\frac{r^{2\lambda - 1} \lambda^2 \log_{|D|}^2(r)}{|D|}\right)$.

The MUL_B tables are computed similarly as done in (1), starting with $(q + 1) \times (q + 1)$ blocks already present in A^{r_2}, and continuing with blocks accumulated as the sequence progress. The overall preprocessing time of all these blocks is $O\left(\frac{r_1^{2+\lambda}}{|D|}\right)$ (see Section "Preprocessing of matrix-vector $\otimes$ computations"), and so the computation time required for each step $r_2 \leq r < r_1$ is $O\left(\frac{r_1^{1+\lambda}}{|D|}\right) = O\left(\frac{r^{1+\lambda}}{|D|}\right)$.

All in all, the time complexity due to computations of (1) and (2) for each step $r_0 \leq r < r_1$ is $O\left(\frac{r^{1+\lambda}}{|D|}\right)$. In particular, the overall time complexity of preprocessing the n-size prefix $A^0, A^1, \ldots, A^n$ of the streamed matrices is $O\left(\frac{n^{2+\lambda}}{|D|}\right)$.

The EDDC algorithm based on efficient *D*-discrete min-plus matrix-vector multiplication

Consider the EDDC problem in cases where all edit operation costs are integers. As explained in Section "*D*-discrete matrices and the EDDC problem with integer costs", the EDDC DP tables can be considered *D*-discrete. This property allows for efficient min-plus square *D*-discrete matrix-vector multiplications, using the algorithm described in Section "An efficient *D*-discrete min-plus matrix-vector multiplication algorithm" to yield an $O\left(\frac{|\Sigma| n^3}{\log_{|D|}^2 n}\right)$ running time algorithm for EDDC. We next describe an online version of the algorithm, in which the letters of the input strings s and t are received in a streaming model.

Assume that some pair of prefixes $s_{0,i}$ and $t_{0,j-1}$ was already processed, and all entries in the DP matrices corresponding to these prefixes are computed. We explain how to update the tables in case where the next letter to arrive is the letter t_{j-1} in t, where the case in which the arriving letter is from s is symmetric. The DP matrices are D-discrete, and assume that lookup tables for efficient min-plus multiplications of these matrices are maintained as explained in the previous section. The addition of t_{j-1} requires updating all matrices of the forms T^ε, T^α, and T'^α, for which the j-th row and column should be added. In addition, it is required to add the j-th column to matrices of the form ED and EDT^α.

In the first stage, the algorithm computes rows and columns j in all matrices of the form T'^α, T^α, and T^ε. The process is similar to the computation of these entries by the loop in lines 5 to 8 of Algorithm 1, with the following modification. Let $q_j = \left\lfloor \lambda \log_{|D|}(2j) \right\rfloor$, and let $j' = q_j \left\lfloor \frac{j}{q_j} \right\rfloor$. The algorithm first initializes the entries $[j-1, j]$ in all these matrices with the corresponding base-case values. The column is partitioned to intervals of length q_j, where as before Q_k denotes the interval $I_{kq_j,(k+1)q_j}$. Once an interval Q_k is computed (i.e. the loop was executed with respect to index $l = kq_j$), the Δ-encoding of the subvector $T^\alpha \left[Q_k, j\right]$ is computed and kept for its later usage as lookup index. In addition, upon starting to compute the entries within an interval Q_k (i.e. when $l = (k+1)q_j - 1$), the following multiplications are computed for every $\alpha \in \Sigma$:

$$y^{\alpha,k} = T^\alpha \left[Q_k, I_{q_j(k+1),j'}\right] \otimes T^\alpha \left[I_{q_j(k+1),j'},j\right]$$

$$\overset{\text{Obs.1}}{=} \min \left\{ T^\alpha \left[Q_k, Q_p\right] \otimes T^\alpha \left[Q_p,j\right] \mid k < p < \frac{j'}{q_j} \right\}$$

Observe that all required entries for the computation of $y^{\alpha,k}$ are already computed and stored in T^α, and that similarly as done in Section "Computing matrix-vector multiplications", $y^{\alpha,k}$ can be computed by performing $O\left(\frac{j}{q_j}\right)$ constant time lookup table queries. After $y^{\alpha,k}$ is computed, $y^{\alpha,k}[x]$ contains the value $\min \left\{ T^\alpha[kq_j + x, h] + T^\alpha[h,j] \mid (k+1)q_j \le h < j' \right\}$. Given $y^{\alpha,k}[x]$, the number of expressions that need to be examined in line 5 of the loop with respect to $l = kq_j + x$ reduces to $O(q_j)$ per entry (considering values of the index h between l and $(k+1)q_j$, and between j' and j). Entries in matrices of the form T^α and T^ε are computed exactly as done in lines 6 and 7 of Algorithm 1, respectively.

In the second stage, column j is computed in matrices EDT^α and ED. This is achieved by extending Equations 13 and 12 to have an entire column on the left-hand side, as follows:

$$EDT^\alpha \left[I_{2,i+1},j\right] \overset{\text{Eq.13}}{\longleftarrow} ED\left[I_{2,i+1},I_{1,j}\right] \otimes T^\alpha \left[I_{1,j},j\right] \qquad (15)$$

$$ED\left[I_{2,i+1},j\right] \overset{\text{Eq.12}}{\longleftarrow}$$

$$\min \left\{ \begin{array}{l} S^\alpha \left[0, I_{2,i+1}\right] + T^\alpha[0,j], \\[2ex] tr(S^\alpha) \left[I_{2,i+1}, I_{1,i+1}\right] \otimes EDT^\alpha \left[I_{1,i+1},j\right] \end{array} \middle| \; \alpha \in \Sigma \right\}$$
$$(16)$$

This completes the update of the DP tables due to the addition of the letter t_{j-1}.

Complexity analysis

After receiving n letters, the prefixes $s_{0,i}$ and $t_{0,j}$ of the input strings were preprocessed for some i,j such that $i + j = n$. The maintenance of lookup tables for efficient D-discrete multiplications requires at most $O\left(\frac{|\Sigma|n^{1+\lambda}}{|D|}\right)$ operations per step among the first n steps, and $O\left(\frac{|\Sigma|n^{2+\lambda}}{|D|}\right)$ operations for all first n steps, as shown in Section "Online preprocessing of D-discrete matrices".

Adding a letter t_{j-1} to the instance, the time required for processing the entries in column j of the T matrices is as follows. $O\left(\frac{|\Sigma|j}{q_j}\right)$ vectors $y^{\alpha,k}$ need to be computed, each vector is computed in $O\left(\frac{j}{q_j}\right)$ time, and their total computation time is therefore $O\left(\frac{|\Sigma|j^2}{q_j^2}\right)$. In addition, $O(|\Sigma|j)$ entries in tables T'^α are computed in $O(q_j)$ time each, and $O(|\Sigma|j)$ entries in tables T^α and T^ε are computed in $O(|\Sigma|)$ time each. Therefore, the total time for computing column j in all these matrices is $O\left(\frac{|\Sigma|j^2}{q_j^2} + |\Sigma|^2 j\right) = O\left(\frac{|\Sigma|n^2}{\lambda^2 \log_{|D|}^2(n)} + |\Sigma|^2 n\right)$.

Computing column j in matrices EDT^α and ED, the algorithm performs $O(|\Sigma|)$ matrix-vector min-plus multiplications (Equations 15 and 16), each taking $O\left(\frac{n^2}{\lambda^2 \log_{|D|}^2(n)}\right)$ time using the algorithm in Section "The EDDC algorithm based on efficient D-discrete min–plus matrix-vector multiplication", and computes the

entry-wise minimum of $|\Sigma|$ i-length vectors (Equation 16) in $O(|\Sigma|i)$ time. Hence, the total time complexity of computing column j is $O\left(\frac{|\Sigma|n^2}{\lambda^2 \log^2_{|D|}(n)} + |\Sigma|^2 n\right)$. Symmetrically, this bounds the running time when the n-th letter comes from the source string s, and so the total running time over all first n steps is $O\left(\frac{|\Sigma|n^3}{\lambda^2 \log^2_{|D|}(n)} + |\Sigma|^2 n^2\right)$. The algorithm requires $O\left(\frac{|\Sigma|n^{2+\lambda}}{\lambda^2 \log_{|D|}(n)}\right)$ space for the computed tables.

Online VMT algorithms

The online algorithm for EDDC presented in the previous section can be generalized for other problems with similar properties. Specifically, VMT problems [11], which utilize min-plus multiplications and for which it can be guaranteed that computed DP matrices are D-discrete, can have their algorithms implemented using the same framework as we have presented above. Thus, in contrast to the general case for VMT problems in which it is required that the complete input be available at the beginning of the algorithm's run, in the D-discrete case the input can be obtained in a streaming model. In addition, the asymptotic time complexity in such cases is slightly reduced with respect to the time complexity of the case of min-plus multiplication of general matrices. A concrete example to such a problem is the RNA base-pairing maximization problem [11,15], in which the difference between adjacent entries (in the single DP matrix the algorithm uses) is either 0 or 1. This property was previously exploited by Frid and Gusfield [14] to obtain an $O\left(\frac{n^3}{\log n}\right)$ algorithm for the problem. Using the D-discrete min-plus multiplication algorithm presented here, this immediately implies an algorithm having the improved time bound of $O\left(\frac{n^3}{\log^2 n}\right)$. Additional related problems from the domains of RNA folding and Context Free Grammars (CFGs) parsing fall under the VMT framework, and it is likely that D-discreteness can be exploited for accelerating the computation of more problems within this family.

Additional acceleration using run-length encoding

Let w be a string. A maximal substring of w containing multiple repeats of the same letter is called a *run* in w. The *Run Length Encoding* (RLE) of w is a representation of the string in which each run is encoded by the corresponding repeating letter α and its repeat count p (denoted α^p). For example, the string $w = aabbbaccc$ is a concatenation of the four runs *aa*, *bbb*, *a*, and *ccc*, and its RLE is $a^2b^3a^1c^3$. Denote by $\tilde{w}$ the *compressed form*

of w, which replaces each run in w by a single occurrence of the corresponding letter. When n denotes the length of w, $\tilde{n}$ will denote the length of the compressed form of w. The *run index $\tilde{i}$* of a letter w_i in w is the index of the run in which w_i participates. It can be asserted that the compressed form of the substring $w_{i,j}$ of w is the substring $\tilde{w}_{\tilde{i},(\widetilde{j-1})+1}$ of $\tilde{w}$. In the above example, $\tilde{w} = abac$, and therefore $\tilde{n} = 4$ (while $n = 9$). The run indices of all letters in w are given by the sequence $[0, 0, 1, 1, 1, 2, 3, 3, 3]$, and the compressed form of $w_{3,8} = bbacc$ is $\tilde{w}_{3,\tilde{7}+1} = \tilde{w}_{1,4} = bac$.

Previous works [7-9] showed how RLE can be exploited for improving the efficiency of EDDC algorithms. In these works it was required that the costs of duplications and contractions be less than the costs of all other operations (the requirement was implicit in [9], see discussion in Section "A comparison with previous works"). This requirement is somewhat unnatural for the application of minisatellite map comparison, since it assumes that mutations, which are typically common events, should cost more than the less common events of duplications and contractions. In this section, we adapt a similar RLE-based acceleration to our EDDC algorithm. The application of this acceleration requires the following constraint over cost functions:

Constraint 1. *For every $\alpha, \beta \in \Sigma$, $dup(\alpha) \leq dup(\beta) + mut(\beta, \alpha) \leq ins(\alpha)$, and $cont(\alpha) \leq cont(\beta) + mut(\alpha, \beta) \leq del(\alpha)$.*

The constraint $dup(\beta) + mut(\beta, \alpha) \leq ins(\alpha)$ implies that it never costs more to replace an insertion of some letter α into some nonempty string by the duplication of a letter β adjacent to the insertion position, and its consecutive mutation to α. Thus, we may assume w.l.o.g that optimal edit scripts do not contain insertions (unless applied to empty strings), or in other words, generation of new letters can only be obtained via duplications. Such an assumption is relatively reasonable in the context of minisatellite map comparison, considering the biological mechanisms that describe generative modifications.

The constraint $dup(\alpha) \leq dup(\beta) + mut(\beta, \alpha)$ can be intuitively understood by the example of generating a string of the form $\alpha\alpha\beta$ from a string of the form $\alpha\beta$. Due to the constraint, it would cost the same or less if the string $\alpha\alpha\beta$ is obtained by duplicating the α letter in $\alpha\beta$, rather than by duplicating the β letter and mutating its left copy into α. Again, such an assumption is relatively reasonable for the minisatellite map application. Symmetric arguments hold with respect to the constraint over contraction and deletion costs.

Algorithm 3: RL-LETTER-TO-STRING(t)

1 Run Stage 1 of Algorithm 2 over the input string $\tilde{t}$. Denote computed matrices T^α by $\tilde{T}^\alpha$.
2 For n the length of t, allocate a vector DC of length $(n+1)$, and set its two first entries $DC[0]$ and $DC[1]$ to zeros.
3 **For** $j = 2, 3, \ldots, n$ **do**
4 $\quad$ Set $DC[j] \stackrel{\text{Eq.17}}{\longleftarrow} \begin{cases} DC[j-1] + dup(t_{j-1}), & t_{j-1} = t_{j-2}, \\ DC[j-1], & \text{otherwise.} \end{cases}$

Observation 4. *Let s and w be strings. Then, $ed(s, w\beta\beta) \leq ed(s, w\beta) + dup(\beta)$, and $ed(s, \beta\beta w) \leq ed(s, \beta w) + dup(\beta)$ for every $\beta \in \Sigma$.*

The correctness of Observation 4 follows from the existence of a script from s to $w\beta\beta$ whose cost is $ed(s, w\beta) + dup(\beta)$: this script first applies an optimal script to transform s into $w\beta$ at cost $ed(s, w\beta)$, and then duplicates the last β in $w\beta$ at cost $dup(\beta)$.

Lemma 5. *Let α, β be letters and $w \neq \varepsilon$ a string. When Constraint 1 holds, $ed(\alpha, \beta w), ed(\alpha, w\beta) \geq ed(\alpha, w) + dup(\beta)$, and $ed(\beta w, \alpha), ed(w\beta, \alpha) \geq ed(w, \alpha) + cont(\beta)$.*

The proof of Lemma 5 appears in Appendix "Proofs to lemmas corresponding to the run-length encoding based EDDC algorithm".

Next, we show how to reduce the number of expressions that need to be considered in the EDDC recursive equations, in case Constraint 1 applies. For a string w of length at least 2, denote by $R(w) \subseteq P(w)$ the set of all partitions (w^a, w^b) of w such the last letter in w^a is different from the first letter in w^b. For example, for $w = aabbbcdddd, R(w) = \{(aa, bbbcdddd), (aabbb, cdddd), (aabbbc, dddd)\}$. Observe that $|R(w)| = \tilde{n} - 1$.

We start by describing how to improve the computation efficiency of EDDC for cases in which one of the input strings contains a single letter. Denote by $dupcost(w)$ the cost of the edit script from $\tilde{w}$ to w which generates each run α^p in w by applying $p - 1$ duplication operations over the corresponding letter α in $\tilde{w}$. Similarly, denote by $contcost(w)$ the cost of the edit script from w to $\tilde{w}$ which reduces each run α^p in w by applying $p - 1$ contraction operations over α. For example, for $w = aabbbbbaaccc, dupcost(w) = 2dup(a) + 3dup(b) + 2dup(c)$

and $contcost(w) = 2cont(a) + 3cont(b) + 2cont(c)$. Note that $dupcost(w) \geq ed(\tilde{w}, w)$, and $contcost(w) \geq ed(w, \tilde{w})$. It is simple to assert the following recursive relations:

The following lemma shows that when one of the input strings contains a single letter, the edit distance can be inferred from the edit distance between this letter and the compressed form of the second string.

Lemma 6. *Let α be a letter and w a string. When Constraint 1 holds, $ed(\alpha, w) = ed(\alpha, \tilde{w}) + dupcost(w)$, and $ed(w, \alpha) = contcost(w) + ed(\tilde{w}, \alpha)$.*

The following lemma shows that given a certain edit script from string u, its cost is greater than or equal to the cost of its application on a superstring of u.

For a string s of the form $s = s^a u s^b$ and an edit script $\mathcal{ES} = \langle u = u^0, u^1, \ldots, u^r = w \rangle$ from u to w, denote by $\mathcal{ES}(s)$ the edit script $\mathcal{ES}(s) = \langle s = s^a u s^b = s^a u^0 s^b, s^a u^1 s^b, \ldots, s^a u^r s^b = s^a w s^b \rangle$ from $s = s^a u s^b$ to $t = s^a w s^b$.

Lemma 7. *For $s = s^a u s^b$ and $\mathcal{ES} = \langle u = u^0, u^1, \ldots, u^r = w \rangle$, $cost(\mathcal{ES}(s)) \leq cost(\mathcal{ES})$.*

The proofs of Lemma 6 and Lemma 7 appear in the Appendix.

Equations 17 and 18 and Lemma 6 support the following preprocessing algorithm, Algorithm 3. Given a target string t, Algorithm 3 generates data structures that enable retrieving in constant time values of the form $ed(\alpha, t_{i,j})$ for every $\alpha \in \Sigma$ and every substring $t_{i,j}$ of t. The algorithm generates tables of the form $\tilde{T}^\alpha$ for every $\alpha \in \Sigma$, such that entries $\tilde{T}^\alpha[i, j]$ contain the corresponding values $ed(\alpha, \tilde{t}_{i,j})$. In addition, the algorithm generates a vector DC, such that entries $DC[j]$ contain the corresponding values $dupcost(t_{0,j})$. Then, queries of the form $ed(\alpha, t_{i,j})$ can be answered in a constant time according to Equation 19 below.

$$dupcost(w\beta) = \begin{cases} dupcost(w) + dup(\beta), & w \text{ ends with } \beta, \\ dupcost(w), & \text{otherwise.} \end{cases} \tag{17}$$

$$dupcost(w) = \begin{cases} dupcost(w^a) + dupcost(w^b), & (w^a, w^b) \in R(w), \\ dupcost(w^a \beta) + dupcost(\beta w^b) + dup(\beta), & (w^a \beta, \beta w^b) \in P(w). \end{cases} \tag{18}$$

$$ed\left(\alpha, t_{i,j}\right) \overset{\substack{\text{Lem.6}\\ \text{Eq.18}}}{=} \tilde{T}^\alpha[i, (j \overset{\frown}{-} 1) + 1]$$
$$+ \begin{cases} DC[j] - DC[i], & t_{i-1} \neq t_i, \\ DC[j] - DC[i] - dup(t_i), & \text{otherwise}. \end{cases}$$
$$(19)$$

An algorithm which is symmetric to Algorithm 3 can be described in order to preprocess a string s for queries of the form $ed\left(s_{i,j}, \alpha\right)$.

We continue to describe the improved computation in the case where both input strings s and t are of length at least 2. To do so, we first add some auxiliary notation. For an interval $I_{x,y}$ of positions within a string w, denote by $\tilde{I}_{x,y}$ the subsequence of indices in $I_{x,y}$ which are start positions of runs in w. For example, for $w = aabbbaacccc$, the interval $I_{1,7} = [1, 2, \ldots, 6]$ contains all positions of letters within the substring $w_{1,7} = abbbaa$, and $\tilde{I}_{1,7} = [2, 5]$ contains the start positions in w of the runs bbb and aa that are included in $I_{1,7}$ (the first letter a in $w_{1,7}$ belongs to a run in w that starts in position 0, and therefore position 1 is not included in $\tilde{I}_{1,7}$). This notation will be used for defining subsequences of rows and columns in DP matrices maintained by the algorithm, where some of these intervals are derived from the source string s, and some from the target string t. We will assume that the string from which $\tilde{I}_{x,y}$ was derived is clear from the context, and will not specify it explicitly. For example, when $\tilde{I}_{x,y}$ defines rows in matrices ED or EDT^α, or either rows or columns in matrices S^α, then the indices in $\tilde{I}_{x,y}$ are derived from the source string s. When $\tilde{I}_{x,y}$ defines columns in matrices ED or EDT^α, or either rows or columns in matrices T^α, then the indices in $\tilde{I}_{x,y}$ are derived from the target string t. Subsequences $\tilde{I}_{x,y}$ will be used for defining *sparse regions* in matrices, i.e. regions containing sets of rows or columns which are not necessarily adjacent.

Consider the computation of $ed(s, t)$ as expressed in Equation 8. Assume first the special case where s ends with a run of length at least 2. In this case, s is of the form $s = w\beta\beta$ for some string w and a letter β. For every partition (t^a, t^b) of t, it is possible to combine an optimal script $\mathcal{ES}^1$ from the prefix $w\beta$ of s to t^a and an optimal script $\mathcal{ES}^2$ from the suffix β of s to t^b, and to obtain a script $\mathcal{ES} = \langle \mathcal{ES}^1(w\beta\beta), \mathcal{ES}^2(t^a\beta) \rangle$ from s to t. Therefore, $ed(w\beta\beta, t) \leq cost(\mathcal{ES}) = cost(\mathcal{ES}^1(w\beta\beta)) + cost(\mathcal{ES}^2(t^a\beta)) \overset{\text{Lem.7}}{\leq} cost(\mathcal{ES}^1) + cost(\mathcal{ES}^2) = ed(w\beta, t^a) + ed(\beta, t^b)$. In particular, $ed(w\beta\beta, t) \leq \min\{ed(w\beta, t^a) + ed(\beta, t^b) \mid (t^a, t^b) \in P(t)\} \overset{\text{Eq.9}}{=} edt^\beta(w\beta, t)$. In addition, it is possible to compose an edit script from $w\beta\beta$ to t by first contracting the last two letters to obtain the string $w\beta$, and then applying an optimal script from $w\beta$ to t. The cost of such a script is $ed(w\beta, t) + cont(\beta)$, and therefore

we get that $ed(w\beta\beta, t) \leq \min\{edt^\beta(w\beta, t), ed(w\beta, t) + cont(\beta)\}$.

Next, we show that $ed(w\beta\beta, t) \geq \min\{edt^\beta(w\beta, t), ed(w\beta, t) + cont(\beta)\}$. From Equation 8, either $ed(w\beta\beta, t) = ed(w\beta\beta, \alpha) + ed(\alpha, t)$ or $ed(w\beta\beta, t) = edt^\alpha(s^a, t) + ed(s^b, \alpha)$ for some $\alpha \in \Sigma$ and $(s^a, s^b) \in P(w\beta\beta)$. Consider first the latter case. If $(s^a, s^b) = (w\beta, \beta)$, then

$$ed(w\beta\beta, t) = edt^\alpha(w\beta, t) + ed(\beta, \alpha)$$
$$\overset{\text{Eq.9}}{=} \min\left\{ed\left(w\beta, t^a\right) + ed\left(\alpha, t^b\right) \mid \left(t^a, t^b\right) \in P(t)\right\}$$
$$+ ed(\beta, \alpha)$$
$$\overset{\text{Obs.5}}{\geq} \min\left\{ed\left(w\beta, t^a\right) + ed\left(\beta, t^b\right) \mid \left(t^a, t^b\right) \in P(t)\right\}$$
$$\overset{\text{Eq.9}}{=} edt^\beta(w\beta, t).$$

Else, s^b is of length at least 2, and there is some string u such that $s^b = u\beta\beta$ and $w = s^au$. In this case, $ed(w\beta\beta, t) = edt^\alpha(s^a, t) + ed(u\beta\beta, \alpha) \overset{\text{Lem.5}}{\geq} edt^\alpha(s^a, t) + ed(u\beta, \alpha) + cont(\beta) \overset{\text{Eq.8}}{\geq} ed(w\beta, t) + cont(\beta)$. Similarly, it can be shown that when $ed(w\beta\beta, t) = ed(w\beta\beta, \alpha) + ed(\alpha, t)$ for some $\alpha \in \Sigma$, $ed(w\beta\beta, t) \geq ed(w\beta, t) + cont(\beta)$, and so $ed(w\beta\beta, t) \geq \min\{edt^\beta(w\beta, t), ed(w\beta, t) + cont(\beta)\}$. Thus,

$$ed(w\beta\beta, t) = \min\left\{edt^\beta(w\beta, t), ed(w\beta, t) + cont(\beta)\right\}$$
$$(20)$$

Formulating Equation 20 with respect to the data structures defined in Section "A baseline dynamic-programming algorithm for EDDC" (under the assumption that all values appearing at the right-hand side of the equation are computed and stored in the corresponding entries), we get the following equation:

$$ed\left(s_{0,i}, t_{0,j}\right) = \min\left\{EDT^{s_{i-1}}[i-1, j], ED[i-1, j]\right.$$
$$\left. + cont(s_{i-1})\right\} \text{ (when } s_{i-1} = s_{i-2})$$
$$(21)$$

Now, consider the case where the last run in s is of length 1 (i.e. s is not of the form $w\beta\beta$). Assume first that the term that yields the minimum value of the right-hand side of Equation 8 is of the form $edt^\alpha(s^a, t) + ed(s^b, \alpha)$ for some partition $(s^a, s^b) \in P(s)$ and a letter $\alpha \in \Sigma$. If $(s^a, s^b) \notin R(s)$, then there is some letter $\beta \in \Sigma$ which is both the last letter of s^a and the first letter of s^b. In this case, we can write $s^a = w\beta$ and $s^b = \beta u$. Note that $u \neq \varepsilon$ (since $s \neq w\beta\beta$ by definition), and so $ed(s, t) = edt^\alpha(w\beta, t) + ed(\beta u, \alpha) \overset{\text{Eq.9}}{=} \min\{ed(w\beta, t^a) + ed(\alpha, t^b) \mid (t^a, t^b) \in P(t)\} + ed(\beta u, \alpha)$

Lem.5,

Obs.4

$$\geq \min \left\{ (ed\,(w\beta\beta, t^a) - dup(\beta)) + ed\,(\alpha, t^b) \mid (t^a, t^b) \in P(t) \right\} + (ed\,(u, \alpha) + dup(\beta)) = \min \left\{ ed\,(w\beta\beta, t^a) + ed(\alpha, t^b) \mid (t^a, t^b) \in P(t) \right\} + ed(u, \alpha) \stackrel{\text{Eq.9}}{=} edt^\alpha(w\beta\beta, t) + ed(u, \alpha).$$ From the optimality of the partition $(s^a, s^b) = (w\beta, \beta u)$, it follows that $ed(s, t) = edt^\alpha(w\beta\beta, t) + ed(u, \alpha)$. If u starts with β this step can be repeated, and inductively we can apply such partition refinements until obtaining a partition (s^a, s^b) of s such that $ed(s, t) = edt^\alpha\,(s^a, t) + ed\,(s^b, \alpha)$ and $(s^a, s^b) \in R(s)$. Now, Equation 8 can be revised as follows:

$$ed(s, t) = \min \left\{ \begin{array}{l} ed(s, \alpha) + ed(\alpha, t), \\ edt^\alpha\,(s^a, t) + ed\,(s^b, \alpha) \end{array} \left| \begin{array}{l} (s^a, s^b) \in R(s), \\ \alpha \in \Sigma \end{array} \right. \right\}$$

(when s is not of the form $w\beta\beta$)

(22)

Using the DP formulation, we get

$$ed\,(s_{0,i}, t_{0,j}) = \min \left\{ \begin{array}{l} S^\alpha[0, i] + T^\alpha[0, j], \\ EDT^\alpha[h, j] + S^\alpha[h, i] \end{array} \left| \begin{array}{l} h \in \tilde{I}_{0,i}, \\ \alpha \in \Sigma \end{array} \right. \right\}$$

$$= \min \left\{ \begin{array}{l} S^\alpha[0, i] + T^\alpha[0, j], \\ tr\,(S^\alpha)\,[i, \tilde{I}_{0,i}] \otimes EDT^\alpha\,[\tilde{I}_{0,i}, j] \end{array} \left| \alpha \in \Sigma \right. \right\}$$

(when $s_{i-1} \neq s_{i-2}$)

(23)

Similarly as shown for the computation of $ed(s, t)$, it is possible to revise the computation of $edt^\alpha(s, t)$ under Constraint 1 and obtain the following equations:

$$edt^\alpha(s, w\beta\beta) = \min \left\{ \begin{array}{l} edt^\alpha(s, w\beta) + dup(\beta), \\ ed(s, w\beta) + mut(\alpha, \beta) \end{array} \right\}$$

(24)

$$edt^\alpha\,(s_{0,i}, t_{0,j}) = \min \left\{ \begin{array}{l} EDT^\alpha[i, j-1] + dup\,(t_{j-1}), \\ ED[i, j-1] + mut\,(\alpha, t_{j-1}) \end{array} \right\}$$

(when $t_{j-1} = t_{j-2}$)

(25)

$$edt^\alpha(s, t) = \min \left\{ ed\,(s, t^a) + ed\,(\alpha, t^b) \mid (t^a, t^b) \in R(t) \right\}$$

(when t is not of the form $w\beta\beta$)

(26)

$$edt^\alpha\,(s_{0,i}, t_{0,j}) = \min \left\{ ED[i, h] + T^\alpha[h, j] \mid h \in \tilde{I}_{0,j} \right\}$$
$$= ED\,[i, \tilde{I}_{0,j}] \otimes T^\alpha\,[\tilde{I}_{0,j}, j]$$

(when $t_{j-1} \neq t_{j-2}$)

(27)

Finally, we present Algorithm 4, which is an efficient version of Algorithm 2. Stage 1 of the new algorithm is accelerated by using Algorithm 3 to compute for every $\alpha \in \Sigma$ distances from α all substrings of s and distances from all substrings of t to α. In Stage 2, we use the equations developed above in order to accelerate the computation. The correctness of the algorithm follows from the correctness of the recursive equations, and can be asserted similarly as done for Algorithm 2.

Complexity analysis

Assume for simplicity that compressed forms of both input strings s and t have the same length $\tilde{n}$.

Algorithm 3. The running time of line 1 of the algorithm is $O(n)$ for computing the compressed form of the input string, and $O(|\Sigma|MP(\tilde{n}))$ for running Stage 1 of Algorithm 2 over this compressed string. Lines 2 and 3 require $O(n)$ time, and so the overall time complexity of Algorithm 3 is $O(n + |\Sigma|MP(\tilde{n}))$. The space complexity for computing and maintaining all matrices is $O(|\Sigma|\tilde{n}^2)$, and an additional $O(n)$ space is required for the vector DC. Hence, the overall space complexity of the algorithm is $O(n + |\Sigma|\tilde{n}^2)$ (see Section "Time complexity analysis" for complexity analysis of Stage 1 of Algorithm 2).

Algorithm 4. Time and space complexities of Stage 1 of the algorithm are identical to those of Algorithm 3. As in Section "The algorithm", the computations governing the running time of Stage 2 are those of matrix multiplications performed within recursive calls to RL-COMPUTE-MATRIX.

The recursive computation of RL-COMPUTE-MATRIX can be visualized as a tree (see Figure 4). Each node in the tree corresponds to a call to RL-COMPUTE-MATRIX over some regions $I_{i,k} \times I_{j,l}$, which is either a leaf in case that $k = i + 1$ and $l = j + 1$, or otherwise an internal node. In the latter case, the node has exactly two children, corresponding to the two recursive calls obtained from either a vertical (lines 11 and 13) or a horizontal (lines 16 and 18) partition of the region. For simplicity, assume that the interval length $\tilde{n} = |\tilde{I}_{2,n+1}| = 2^x$ for some integer x. It can be observed that the algorithm alternates between vertical and horizontal partitions along paths from the root of the tree, where regions of two different nodes in the same depth y are disjoint, and the union of all regions of nodes in depth y covers the entire initial region $I_{2,n+1} \times I_{2,n+1}$ of the root node. For every $0 \leq y \leq \log(n)$, there are two series of intervals $I^{y,0}, I^{y,1}, \ldots, I^{y,2^y-1}$ and $J^{y,0}, J^{y,1}, \ldots, J^{y,2^y-1}$, such that the set of regions corresponding to all nodes in depth $2y$ is $\left\{ I^{y,f} \times J^{y,g} \mid 0 \leq f < 2^y, 0 \leq g < 2^y \right\}$, and the set of regions corresponding to all nodes in depth $2y + 1$ is $\left\{ I^{y,f} \times J^{y+1,g} \mid 0 \leq f < 2^y, 0 \leq g < 2^{y+1} \right\}$. In

addition, the corresponding subsequences $\tilde{I}^{y,0}, \ldots, \tilde{I}^{y,2^y-1}$ and $\tilde{J}^{y,0}, \ldots, \tilde{J}^{y,2^y-1}$ have all the same size 2^{x-y}.

Consider a node of depth $2y$ whose corresponding region is $I^{y,f} \times J^{y,g}$, and the two regions corresponding to its children $I^{y,f} \times J^{y+1,2g}$ and $I^{y,f} \times J^{y+1,2g+1}$. The computation time spent on the node is dominated by

the matrix multiplications performed in line 12 of RL-COMPUTE-MATRIX. This includes $|\Sigma|$ matrix multiplications between pairs of matrices such the dimensions of the first matrix in each pair is $|I^{y,f}| \times |\tilde{J}^{y+1,2g}| = |I^{y,f}| \times 2^{x-y-1}$, and the dimensions of the second matrix in a pair is $|\tilde{J}^{y+1,2g}| \times |\tilde{J}^{y+1,2g+1}| = 2^{x-y-1} \times 2^{x-y-1}$. Observe that

---

Algorithm 4: RL-MATRIX-EDDC(s, t)

// Stage 1

1   Let n denote the lengths of s and t. Run Algorithm 3 to preprocess s and t, and generate $(n+1) \times (n+1)$ matrices S^α and T^α for every $\alpha \in \Sigma$ as defined in Section "The algorithm" (applying Equation 19).

// Stage 2

2   Allocate $(n+1) \times (n+1)$ matrices EDT^α for every $\alpha \in \Sigma$, and an $(n+1) \times (n+1)$ matrix ED. Initialize base-case corresponding entries in all matrices EDT^α and ED as describe in Algorithm 1. This includes all entries in the first two rows and the first two columns in these matrices.

3   Set $EDT^\alpha[I_{2,n+1}, I_{2,n+1}] \leftarrow ED[I_{2,n+1}, I_{0,2}] \otimes T^\alpha[I_{0,2}, I_{2,n+1}]$ for every $\alpha \in \Sigma$, and set $ED[I_{2,n+1}, I_{2,n+1}] \leftarrow \min\{tr(S^\alpha)[I_{2,n+1}, I_{0,2}] \otimes EDT^\alpha[I_{0,2}, I_{2,n+1}] \mid \alpha \in \Sigma\}$.

4   Run RL-COMPUTE-MATRIX($I_{2,n+1}, I_{2,n+1}$).

5   **return** *ED[n, n]*.

Procedure: RL-COMPUTE-MATRIX($I_{i,k}, I_{j,l}$)

Precondition: $2 \leq i < k$, $2 \leq j < l$, and all entries in matrices S^α and T^α, as well as all entries in submatrices $EDT^\alpha[I_{0,k}, I_{j,l}]$ and $ED[I_{i,k}, I_{0,j}]$, contain the solutions for the corresponding sub-instances. In addition,
$EDT^\alpha[I_{i,k}, \tilde{I}_{j,l}] = ED[I_{i,k}, \tilde{I}_{0,j}] \otimes T^\alpha[\tilde{I}_{0,j}, \tilde{I}_{j,l}]$, and $ED[I_{i,k}, I_{j,l}] = \min\{tr(S^\alpha)[\tilde{I}_{i,k}, \tilde{I}_{0,i}] \otimes EDT^\alpha[\tilde{I}_{0,i}, I_{j,l}] \mid \alpha \in \Sigma\}$.

Postcondition: All entries in the region $I_{i,k} \times I_{j,l}$ in matrices EDT^α and ED contain the solutions for the corresponding sub-instances.

1   **If** $k = i + 1$ *and* $l = j + 1$ **then**

2   **If** $t_{j-1} = t_{j-2}$ **then**

3   Set $EDT^\alpha[i,j] \overset{Eq.25}{\leftarrow} \min\{EDT^\alpha[i, j-1] + dup(t_{j-1}), ED[i, j-1] + mut(\alpha, t_{j-1})\}$ for every $\alpha \in \Sigma$.

4   **If** $s_{i-1} = s_{i-2}$ **then**

5   Set $ED[i,j] \overset{Eq.21}{\leftarrow} \min\{EDT^{s_{i-1}}[i-1, j], ED[i-1, j] + cont(s_{i-1})\}$.

6   **Else**

7   Set $ED[i,j] \overset{\text{precondition,} \ Eq.23}{\leftarrow} \min\{ED[i,j], S^\alpha[0,i] + T^\alpha[0,j] \mid \alpha \in \Sigma\}$.

8   **Else**

9   **If** $|\tilde{I}_{j,l}| \geq |\tilde{I}_{i,k}|$ **then**

      // vertical partitioning

10   Let $j < h < l$ be the smallest index such that $\tilde{h} = \lceil \frac{\tilde{j}+\tilde{l}}{2} \rceil$.

11   Run RL-COMPUTE-MATRIX($I_{i,k}, I_{j,h}$).

12   Update $EDT^\alpha[I_{i,k}, \tilde{I}_{h,l}] \leftarrow \min\{EDT^\alpha[I_{i,k}, \tilde{I}_{h,l}], ED[I_{i,k}, \tilde{I}_{j,h}] \otimes T^\alpha[\tilde{I}_{j,h}, \tilde{I}_{h,l}]\}$ for every $\alpha \in \Sigma$.

13   Run RL-COMPUTE-MATRIX($I_{i,k}, I_{h,l}$).

14   **Else**

      // horizontal partitioning

15   Let $i < h < k$ be the smallest index such that $\tilde{h} = \lceil \frac{\tilde{i}+\tilde{k}}{2} \rceil$.

16   Run RL-COMPUTE-MATRIX($I_{i,h}, I_{j,l}$).

17   Update $ED[\tilde{I}_{h,k}, I_{j,l}] \leftarrow \min\{ED[\tilde{I}_{h,k}, I_{j,l}], \min\{tr(S^\alpha)[\tilde{I}_{h,k}, \tilde{I}_{i,h}] \otimes EDT^\alpha[\tilde{I}_{i,h}, I_{j,l}] \mid \alpha \in \Sigma\}\}$.

18   Run RL-COMPUTE-MATRIX($I_{h,k}, I_{j,l}$).

$|I^{yf}| \geq 2^{x-y-1}$, and such multiplications can be implemented by dividing the interval I^{yf} into $\frac{|I^{yf}|}{2^{x-y-1}}$ intervals of length 2^{x-y-1} each, and performing $\frac{|I^{yf}|}{2^{x-y-1}}$ multiplications between square matrices of dimensions $2^{x-y-1} \times 2^{x-y-1}$ in a total time of $\frac{|I^{yf}|}{2^{x-y-1}}MP(2^{x-y-1})$. Therefore, the time required for all matrix multiplications performed within nodes in depth $2y$ is

$$\sum_{0 \leq f < 2^y} \sum_{0 \leq g < 2^y} \frac{|\Sigma| |I^{yf}| MP(2^{x-y-1})}{2^{x-y-1}}$$
$$= \sum_{0 \leq f < 2^y} \frac{2^y |\Sigma| |I^{yf}| MP(2^{x-y-1})}{2^{x-y-1}}$$
$$= \frac{2|\Sigma| n MP(2^{x-y-1})}{\tilde{n}}.$$

Similarly, it is possible to show that the total time required for all matrix multiplications performed within nodes in depth $2y + 1$ is also $\frac{2n|\Sigma|MP(2^{x-y-1})}{\tilde{n}}$, and so the total computation time of matrix multiplications throughout the entire algorithm run is $O\left(\frac{|\Sigma|n}{\tilde{n}}\sum_{0 \leq y < x} MP(2^y)\right)$. As in Section "The algorithm", using the Master Theorem [16], this summation evaluates to $O\left(\frac{|\Sigma|nMP(\tilde{n})}{\tilde{n}}\right)$. In addition to matrix multiplications, RL-COMPUTE-MATRIX performs $O(|\Sigma|n^2)$ operations in base computations (lines 2-7), and so the total time complexity of the complete algorithm is $O\left(|\Sigma|n^2 + \frac{|\Sigma|nMP(\tilde{n})}{\tilde{n}}\right)$.

A simple implementation of Algorithm 4 can be done using the same space complexity of $O\left(|\Sigma|n^2\right)$, as the space complexity of Algorithm 2. A more involved implementation can be applied by observing that in fact the algorithm only examines and updates entries in matrices of dimensions at most $n \times \tilde{n}$ or $\tilde{n} \times n$ when performing matrix multiplications, and in addition it examines adjacent entries "to the left" or "above" an entry in a base-case region. This observation can be used in order to reduce the space complexity of the algorithm to $O\left(|\Sigma|n\tilde{n}\right)$, where the complete details of such an implementation are omitted from this text.

A comparison with previous works

In this section, we review the previous main algorithms for EDDC by Behzadi and Steyaert [7], Bérard et al. [8] and Abouelhoda et al. [9], and point out similarities and improvements made in our current work.

The main contribution of our work is in obtaining sub-cubic algorithms for EDDC, whereas all previous algorithms have cubic time complexities (for $|\Sigma|$ the alphabet size, n the length of the input strings and $\tilde{n}$

the length of their RLE compressed forms, the algorithms of [7], [8], and [9] obtain the time complexities $O\left(n^2 + n\tilde{n}^2 + |\Sigma|\tilde{n}^3 + |\Sigma|^2\tilde{n}^2\right)$, $O\left(n^3 + |\Sigma|\tilde{n}^3\right)$, and $O(n^2 + n\tilde{n}^2)$, respectively).

Notably, the algorithm of [9] eliminates a $|\Sigma|$ factor that appears in the time complexities of the algorithms given in [7,8] and here. However, this improvement is confined to a constrained model of duplication histories. As we do not assume this model here, we could not use the representation of [9] that allows the elimination of the $|\Sigma|$ time complexity factor.

In general, the frameworks of all algorithms in [7-9] as well as the algorithms presented here are similar. All these algorithms apply two phases, where the first phase computes costs corresponding to all substrings of each one of the input strings separately, and the second phase uses these precomputed costs in order to compute the edit distance between each pair of prefixes of the input strings (our online variant described in Section "An online algorithm for EDDC using min-plus matrix-vector multiplication for discrete cost functions" interleaves these two phases, yet each operation it conducts can be conceptually attributed to one of the phases). The recursive formulas are similar as well, where those for the first phase can be viewed as special kinds of Weighted Context Free Grammar derivation rules.

Next, we address the cost function constraints. All algorithms assume that operation costs are nonnegative and apply additional assumptions similarly to those listed in our Property 1, which can be made without loss of generality.

In [8], operation costs were limited so that all duplications and contractions have the same constant cost (regardless of the letter over which they are applied), all deletions and insertions have the same constant cost, and all mutation costs are symmetric (i.e. $mut(\alpha, \beta) = mut(\beta, \alpha)$ for every $\alpha, \beta \in \Sigma$). While it was argued that these restrictions allow edit distance to be a metric, they limit the generality of the algorithm of [8], where the rest of the previous algorithms we mentioned can handle scoring schemes that not necessarily abide by these restrictions.

Both in [7] and in [8], it was required that all duplication and contraction costs are lower than the costs of any of the insertion, deletion, or mutation costs. This restriction is not explicitly stated in [9], yet seems to be required there as well. For the application of mini-satellite map comparison, this requirement is somewhat unnatural since it assumes that mutations, which are typically common events, should cost more than the less common events of duplications and contractions. Our algorithms can be applied even when this restriction does not hold. However, one of our algorithms, the RLE variant (Section "Additional acceleration using run-length

encoding") adds a new requirement that was absent from those previous algorithms: it requires that for every $\alpha, \beta \in \Sigma$, $dup(\alpha) \leq dup(\beta) + mut(\beta, \alpha) \leq ins(\alpha)$, and $cont(\alpha) \leq cont(\beta) + mut(\alpha, \beta) \leq del(\alpha)$ (our Constraint 1). On one hand, our Constraint 1 is more strict than the constraint of [7] and [8], in the sense that it implies nonnegative lower bounds over differences of the form $ins(\alpha) - dup(\alpha)$ and $del(\alpha) - cont(\alpha)$, while in [7] and [8] it was only required that these differences be nonnegative. On the other hand, our Constraint 1 does not require that the cost of mutations be higher than the cost of duplications and contractions.

We showed that our algorithms are more general with respect to the assumed constraints. We also claim that our algorithms are more precise with respect to the formal problem specification. All previous algorithms (excluding the first algorithm by Bérard and Rivals [2], which had an $O(n^4)$ running time and assumed a constant cost for all mutations in addition to the restrictions in [8]) might output non-optimal solutions in certain cases, as demonstrated in the following example. Consider the input $s = ab$, $t = ef$, and the cost function in which all duplications and contractions cost 1, all deletions and insertions cost 20, and the symmetric mutation costs are as given in Table 1. It can be shown that all three algorithms in [7], [8], and [9] would output the value 18 as the edit distance between the input strings, reflecting one of the edit scripts $\langle ab, eb, ef \rangle$ or $\langle ab, af, ef \rangle$. Nevertheless, the correct value is 17, due to the script $\langle ab, cb, cc, c, d, dd, ed, ef \rangle$. Perhaps it could be possible to specify additional restrictions over the cost functions in order to guarantee that the algorithms in [7], [8], and [9] return optimal solutions for all instances.

Conclusions and discussion

This work presents computational techniques for improving the time complexity of algorithms for the EDDC problem. We adapt the problem to the *VMT* framework defined in [11], which incorporates efficient matrix multiplication subroutines in order to accelerate standard dynamic programming algorithms. We describe an efficient algorithm, as well as two variants which are even more efficient, given some restrictions on the cost functions.

An additional result we give is the currently most efficient algorithm for the min-plus multiplication of D-discrete matrices (matrices for which differences between adjacent entries are integers within an interval of length D).

We note that the running times of our algorithms depend on the alphabet size $|\Sigma|$. For the general algorithm, the running time is $O(|\Sigma| \cdot MP(n))$, where $MP(n)$ is the time complexity of the min-plus multiplication of two $n \times n$ matrices, which is currently upper-bounded by $O\left(\frac{n^3 \log^3 \log n}{\log^2 n}\right)$ [12]. Some of the previous algorithms obtain alphabet independent time complexities, for example the algorithms in [9] and [2]. As we discussed in Section "A comparison with previous works", such algorithms do not solve the most general variant of the problem and require some assumptions on the cost function. Nevertheless, we believe that the matrix multiplication-based techniques for improving the time complexity presented in this paper can also be incorporated to the algorithm of [9], however the details of this enhancement are beyond the scope of this paper.

In contrast to the work of [9], our model assumes that intermediate strings along edit scripts may contain characters which are absent from both source and target strings. This implies that the size of the alphabet $|\Sigma|$ is not bounded by the length of the input sequences. In the context of minisatellite comparison, identifying a feasible alphabet and cost function for this task is an interesting problem beyond the scope of this paper.

Appendix
Correctness of the recursive computation

This section proves Theorem 1, thus asserting the correctness of the recursive computation for the EDDC problem given in Section "The recurrence formula". We start by adding some required notation and showing how long edit scripts can be decomposed to shorter partial scripts. Then, we use the observed recursive properties in order to prove the correctness of the recurrence.

Let s, w, t be strings, $\mathcal{ES}^1 = \langle s = u^{1,0}, u^{1,1}, \ldots, u^{1,r_1} = w \rangle$ an edit script from s to w, and $\mathcal{ES}^2 = \langle w = u^{2,0}, u^{2,1}, \ldots, u^{2,r_2} = t \rangle$ an edit script from w to t. Denote by $\mathcal{ES} = \langle \mathcal{ES}^0, \mathcal{ES}^1 \rangle$ the concatenated edit script $\mathcal{ES} = \langle s = u^{1,0}, u^{1,1}, \ldots, u^{1,r_1} = w = u^{2,0}, u^{2,1}, \ldots, u^{2,r_2} = t \rangle$ from s to t. Note that $cost(\mathcal{ES}) = cost(\mathcal{ES}^1) + cost(\mathcal{ES}^2)$, and $|\mathcal{ES}| = |\mathcal{ES}^1| + |\mathcal{ES}^2|$. This notation extends naturally to concatenations of more than two scripts. For example, $\mathcal{ES} = \langle \mathcal{ES}^1, \mathcal{ES}^2, \ldots, \mathcal{ES}^q \rangle$ denotes an edit script from a string s to a string t obtained by a concatenation of q

Table 1 Mutation costs for the instance *s = ab, t = ef*

	a	b	c	d	e	f
a	0	6	3	6	9	9
b	6	0	3	6	9	9
c	3	3	0	3	6	6
d	6	6	3	0	3	3
e	9	9	6	3	0	6
f	9	9	6	3	6	0

scripts, each script $\mathcal{ES}^i$ transforms some intermediate string w^{i-1} into a string w^i, and $s = w^0$ and $t = w^q$.

Observation 5. *For every three strings s, w, t, $ed(s,t) \leq ed(s,w) + ed(w,t)$.*

The correctness of the above observation follows from the fact that for a pair of optimal edit scripts $\mathcal{ES}^1$ from s to w and $\mathcal{ES}^2$ from w to t, the script $\mathcal{ES} = \langle \mathcal{ES}^1, \mathcal{ES}^2 \rangle$ from s to t satisfies $ed(s,t) \leq cost(\mathcal{ES}) = cost(\mathcal{ES}^1) + cost(\mathcal{ES}^2) = ed(s,w) + ed(w,t)$.

Lemma 7.
For $s = s^a u s^b$ and $\mathcal{ES} = \langle u = u^0, u^1, \ldots, u^r = w \rangle$, $cost(\mathcal{ES}(s)) \leq cost(\mathcal{ES})$.

Proof. Each edit operation transforming $s^a u^i s^b$ to $s^a u^{i+1} s^b$ in $\mathcal{ES}(s)$ corresponds to an operation transforming u^i to u^{i+1} in $\mathcal{ES}$. The only cases where corresponding operations may have different costs are those of insertions or deletions in $\mathcal{ES}$ at the beginning or ending of u^i, which become duplications or contractions in $\mathcal{ES}(s)$, respectively. For example, in case the applied operation over u^i in $\mathcal{ES}$ is the deletion of its first letter α, and α is also the last letter of s^a, then the cost of the operation in $\mathcal{ES}$ is $del(\alpha)$ while the cost of the corresponding operation in $\mathcal{ES}(s)$ is $cont(\alpha) \leq del(\alpha)$. Similar scenarios may occur in case of an insertion of a letter at the beginning of u^i which is identical to the last letter of s^a, as well as in cases of insertions and deletions at the end of u^i of letters identical to the first letter of s^b. In any other case, each pair of corresponding operations have the same cost. Therefore the cost of each operation in $\mathcal{ES}(s)$ is smaller than or equals to the cost of its corresponding operation in $\mathcal{ES}$, and $cost(\mathcal{ES}(s)) \leq cost(\mathcal{ES})$. □

Lemma 8. *Let s and t be two strings, and $(s^a, s^b) \in P(s)$, $(t^a, t^b) \in P(t)$ partitions of s and t, respectively. Then, $ed(s,t) \leq ed(s^a, t^a) + ed(s^b, t^b)$.*

Proof. Let $\mathcal{ES}^a$ be an optimal script from s^a to t^a and $\mathcal{ES}^b$ an optimal script from s^b to t^b. The script $\mathcal{ES}^a(s)$ is a script from $s = s^a s^b$ to $t^a s^b$. Similarly, $\mathcal{ES}^b(t^a s^b)$ is a script from $t^a s^b$ to $t^a t^b = t$. For the script $\mathcal{ES} = \left\langle \mathcal{ES}^a(s), \mathcal{ES}^b(t^a s^b) \right\rangle$ from s to t, we have that

$$ed(s,t) \leq cost(\mathcal{ES}) = cost(\mathcal{ES}^a(s)) + cost(\mathcal{ES}^b(t^a s^b)) \overset{\text{Lem.7}}{\leq} cost(\mathcal{ES}^a) + cost(\mathcal{ES}^b) = ed(s^a, t^a) + ed(s^b, t^b). \quad \square$$

Let s and t be strings. Call a pair of partitions $(s^a, s^b) \in P(s)$ and $(t^a, t^b) \in P(t)$ an *optimal pairwise partition* of s and t if $ed(s,t) = ed(s^a, t^a) + ed(s^b, t^b)$. Say that

an edit script $\mathcal{ES}$ from s to t is a *shortest optimal* script from s to t if $\mathcal{ES}$ is optimal, and for every other optimal script $\mathcal{ES}'$ from s to t, $|\mathcal{ES}| \leq |\mathcal{ES}'|$. For a script $\mathcal{ES} = \langle s = u^0, u^1, \ldots, u^r = t \rangle$ from s to t and $0 \leq i \leq j \leq r$, denote by $\mathcal{ES}^{i,j} = \langle u^i, u^{i+1}, \ldots, u^j \rangle$ the partial script of $\mathcal{ES}$ from u^i to u^j.

Observation 6. *Let $\mathcal{ES} = \langle u^0, u^1, \ldots, u^r \rangle$ be a shortest optimal edit script from u^0 to u^r. For every $0 \leq i \leq j \leq r$, the partial script $\mathcal{ES}^{i,j}$ is a shortest optimal edit script from u^i to u^j. Moreover, for any shortest optimal script $\mathcal{ES}^{*i,j}$ from u^i to u^j within $\mathcal{ES}$, the script $\mathcal{ES}^* = \langle \mathcal{ES}^{0,i}, \mathcal{ES}^{*i,j}, \mathcal{ES}^{j,r} \rangle$ is a shortest optimal script from u^0 to u^r.*

The correctness of the above observation is obtained by noting that if $\mathcal{ES}^{i,j}$ is not a shortest optimal script from u^i to u^j, then for some shortest optimal script $\mathcal{ES}^{*i,j}$ from u^i to u^j we get that the script $\mathcal{ES}^* = \langle \mathcal{ES}^{0,i}, \mathcal{ES}^{*i,j}, \mathcal{ES}^{j,r} \rangle$ either has a lower cost than $\mathcal{ES}$, or is a shorter script of the same cost, in contradiction to $\mathcal{ES}$ being a shortest optimal script from u^0 to u^r.

Lemma 9. *Let $\mathcal{ES} = \langle u^0, u^1, \ldots, u^r \rangle$ be a shortest optimal edit script from u^0 to u^r. If there are two indices $0 \leq i < j \leq r$ such that u^i and u^j are strings of length 1, then $j = i + 1$. In addition, for every $0 < k < r$, $u^k \neq \varepsilon$.*

Proof. Assume there are two indices $0 \leq i < j \leq r$ such that u^i and u^j are strings of length 1, i.e $u^i = \alpha$ and $u^j = \beta$ for some $\alpha, \beta \in \Sigma$. From Observation 6, the partial script $\mathcal{ES}^{i,j} = \langle \alpha = u^i, u^{i+1}, \ldots, u^j = \beta \rangle$ is a shortest optimal script from α to β. Since $j > i$, it must be that $\alpha \neq \beta$ (otherwise the script $\mathcal{ES}' = \langle \mathcal{ES}^{0,i}, \mathcal{ES}^{j,r} \rangle$ is a shorter script from u^0 to u^r of no greater cost than $\mathcal{ES}$, in contradiction to $\mathcal{ES}$ being a shortest optimal script from u^0 to u^r). From Property 1, $ed(\alpha, \beta) = mut(\alpha, \beta)$, and so the edit script containing the single operation of mutating α to β is an optimal script from α to β, and it must be that $j = i + 1$.

In addition, assume by contradiction there is some index $0 < k < r$ such that $u^k = \varepsilon$. The only edit operation which may yield an empty string is a deletion from a single-letter string, and therefore $u^{k-1} = \alpha$ for some letter α. Similarly, the only edit operation which may be applied over an empty string is an insertion, therefore $u^{k+1} = \beta$ for some letter β, in contradiction to the fact that two intermediate strings of length 1 must be consecutive along a shortest optimal script, as shown above. □

Call an edit script $\mathcal{ES}$ from a string s to a string t *simple* if $\mathcal{ES}$ is a shortest optimal script from s to t, in which no generating operation precedes a reducing operation. The following lemma generalizes Lemma 2 of

[6], by considering also indels in addition to contractions and duplications.

Lemma 10. *For every pair of strings s and t, there exists a simple edit script from s to t.*

Proof. Let s and t be two strings, and r the length of a shortest optimal script from s to t. When $r \leq 1$, any shortest optimal script from s to t either contains no reducing operation or contains no generating operation, and in particular is a simple script. Otherwise, $r > 1$, and assume by induction the lemma holds for every pair of strings such that the length of a shortest optimal script from the source string to the target string is less than r. Let $\mathcal{ES} = \langle s = u^0, u^1, \ldots, u^r = t \rangle$ be a shortest optimal script from s to t.

Case 1: The first operation in $\mathcal{ES}$ is not a generating operation. From Observation 6, the partial script $\mathcal{ES}^{1,r}$ is a shortest optimal script from u^1 to u^r, whose length is $r - 1$. From the inductive assumption, there is a simple script $\mathcal{ES}^{*1,r}$ from u^1 to u^r, and from Observation 6 the script $\mathcal{ES}^* = \langle \mathcal{ES}^{0,1}, \mathcal{ES}^{*1,r} \rangle$ is a shortest optimal script from s to t. As the first operation in $\mathcal{ES}^*$ is non-generating (being the same first operation as in $\mathcal{ES}$), $\mathcal{ES}^*$ is simple, and the lemma follows.

Case 2: The first operation in $\mathcal{ES}$ is a generating operation. Similarly as above, we may assume w.l.o.g. by applying the inductive assumption that the partial script $\mathcal{ES}^{1,r}$ is simple. If this partial script is non-reducing, then $\mathcal{ES}$ is non-reducing, and in particular it is simple. Otherwise, let $1 \leq i < r$ be the smallest index such that the transformation of u^i to u^{i+1} is by a reducing operation. Since neither generating nor reducing operations may precede this operation in the partial script $\mathcal{ES}^{1,i}$, it follows that all operations in the partial script $\mathcal{ES}^{1,i}$ (if there are any) are mutations.

The generating operation transforming u^0 to u^1 is either an insertion or a duplication of some letter α in u^0. In both cases, we can write $s = u^0 = vxw$ and $u^1 = vx'w$ (v, x, x' and w are strings), where in the former case $x = \varepsilon$ and $x' = \alpha$, and in the latter case $x = \alpha$ and $x' = \alpha\alpha$. As all operations in the partial script $\mathcal{ES}^{1,i}$ are mutations, each intermediate string u^j, for $1 \leq j \leq i$, is of the form $v^j x^j w^j$, where v^j, x^j, and w^j are string obtained by applying zero or more mutations over v, x', and w, respectively. We argue that the reducing operation transforming $u^i = v^i x^i w^i$ to u^{i+1} cannot be the deletion of a letter or a contraction involving at least one letter within the substring x^i. This is true, since in such a case it would have been possible to avoid the first generating operation in $\mathcal{ES}$ (transforming x to x'), as well as all mutation operations over a reduced letter in x^i, and the reducing operation from u^i

to u^{i+1}. This would yield a script $\mathcal{ES}^{*0,i+1}$ from u^0 to u^{i+1} which is shorter and of no higher coast than $\mathcal{ES}^{0,i+1}$, in contradiction to Observation 6. Hence, the reducing operation from u^i to u^{i+1} either deletes a letter or contracts two letters within one of the substrings v^i or w^i of u^i.

Consider first the case where the reducing operation over u^i is applied within its prefix v^i. Thus, we can write $u^{i+1} = v^{i+1}x^{i+1}w^{i+1}$, where v^{i+1} is the string obtained by applying the corresponding reducing operation over v^i, $x^{i+1} = x^i$, $w^{i+1} = w^i$, and $cost(\langle u^i, u^{i+1}\rangle) = cost(\langle v^i, v^{i+1}\rangle)$. The operations in $\mathcal{ES}^{0,i+1}$ can be assigned into two independent scripts: a script $\mathcal{ES}_v = \langle v = v^0, v^1, \ldots, v^p = v^{i+1}\rangle$ from v to v^{i+1} obtained by merging each multiple occurrence of consecutive identical strings in the series $v = v^1, v^2, \ldots, v^{i+1}$ into a single occurrence, and similarly a script $\mathcal{ES}_{xw} = \langle xw = (xw)^0, (xw)^1 = x'w = x^1w^1, (xw)^2, \ldots, (xw)^q = x^{i+1}w^{i+1}\rangle$ from xw to $x^{i+1}w^{i+1}$. Each operation in $\mathcal{ES}^{0,i+1}$ corresponds to exactly one operation in either $\mathcal{ES}_v$ or $\mathcal{ES}_{xw}$, where the costs of corresponding operations are equal, and therefore $cost(\mathcal{ES}^{0,i+1}) = cost(\mathcal{ES}_v) + cost(\mathcal{ES}_{xw})$ and $|\mathcal{ES}^{0,i+1}| = |\mathcal{ES}_v| + |\mathcal{ES}_{xw}|$.

Now, the script $\mathcal{ES}_v(u^0) = \langle u^0 = vxw = v^0xw, v^1 xw, \ldots, v^p xw = v^{i+1}xw\rangle$ is a script from u^0 to $v^{i+1}xw$, and similarly the script $\mathcal{ES}_{xw}(v^{i+1}xw) = \langle v^{i+1}xw = v^{i+1}(xw)^0, v^{i+1}(xw)^1, \ldots, v^{i+1}(xw)^q = v^{i+1}x^{i+1}w^{i+1} = u^{i+1}\rangle$ is a script from $v^{i+1}xw$ to u^{i+1}. Thus, the script $\mathcal{ES}^{*0,i+1} = \langle \mathcal{ES}_v(u^0), \mathcal{ES}_{xw}(v^{i+1}xw)\rangle$ is a script from u^0 to u^{i+1}. Since $\mathcal{ES}_v$ contains at least one operation (the reducing operation from v^i to v^{i+1}) and no generating operation (since besides the reducing operation $\mathcal{ES}_v$ may contain only mutations), $\mathcal{ES}^{*0,i+1}$ starts with a non-generating operation. In addition, $cost(\mathcal{ES}^{*0,i+1}) = cost(\mathcal{ES}_v(u^0)) + cost(\mathcal{ES}_{xw}(v^{i+1}xw)) \overset{\text{Lem.7}}{\leq} cost(\mathcal{ES}_v) + cost(\mathcal{ES}_{xw}) = cost(\mathcal{ES}^{0,i+1})$ and $|\mathcal{ES}^{*0,i+1}| = |\mathcal{ES}_v(u^0)| + |\mathcal{ES}_{xw}(v^{i+1}xw)| = |\mathcal{ES}_v| + |\mathcal{ES}_{xw}| = |\mathcal{ES}^{0,i+1}|$. From Observation 6, $\mathcal{ES}^{0,i+1}$ is a shortest optimal script from u^0 to u^{i+1}, and so $\mathcal{ES}^{*0,i+1}$ is a shortest optimal script from u^0 to u^{i+1}. Applying Observation 6 again, the script $\mathcal{ES}^* = \langle \mathcal{ES}^{*0,i+1}\mathcal{ES}^{i+1,r}\rangle$ is a shortest optimal script from s to t. Now, the lemma follows from Case 1 of this proof and from the fact the first operation in $\mathcal{ES}^*$ is not a generating operation. □

Lemma 11. *For every $\alpha \in \Sigma$ and every nonempty string t, any simple script from α to t is non-reducing.*

Proof. Let $\mathcal{ES}$ be a simple script from α to t, and assume by contradiction $\mathcal{ES}$ contains a reducing operation. Since $\mathcal{ES}$ is simple, all reducing operations in $\mathcal{ES}$ occur prior to any generating operation, and in particular the first reducing operation is applied after applying zero or more

mutations over α. Such a reducing operation must be a deletion from a string of length 1, resulting with an empty intermediate string, in contradiction to Lemma 9. □

Lemma 12. *Let w and t be strings and β a letter, such that $w \neq \varepsilon$, t is of length at least 2, and there is a non-reducing simple script from $w\beta$ to t. Then, $ed(w\beta, t) = \min \left\{ ed\left(w, t^a\right) + ed\left(\beta, t^b\right) \mid \left(t^a, t^b\right) \in P(t) \right\}$.*

Proof. Let $\mathcal{ES} = \left\langle w\beta = u^0, u^1, \ldots, u^r = t \right\rangle$ be a non-reducing simple script from $w\beta$ to t. For every $0 \leq i \leq r$, construct a partition $(u^{i,a}, u^{i,b})$ of u^i which sustains that $ed\left(w\beta, u^i\right) \geq ed\left(w, u^{i,a}\right) + ed\left(\beta, u^{i,b}\right)$, as follows. For $i = 0$, set $(u^{0,a}, u^{0,b}) = (w, \beta)$, where by definition $ed\left(w\beta, u^0\right) = ed\left(w, u^{0,a}\right) + ed\left(\beta, u^{0,b}\right) = 0$. Now, assume inductively for some $0 < i \leq r$ and a partition $\left(u^{i-1,a}, u^{i-1,b}\right)$ of u^{i-1} that $ed\left(w\beta, u^{i-1}\right) \geq ed\left(w, u^{i-1,a}\right) + ed\left(\beta, u^{i-1,b}\right)$. If the non-reducing operation transforming u^{i-1} to u^i is a mutation, an insertion, or a duplication of a letter within the prefix $u^{i-1,a}$, then set $u^{i,a}$ to be the string obtained by applying this operation over $u^{i-1,a}$, and set $u^{i,b} = u^{i-1,b}$. Otherwise, the operation is a mutation, an insertion, or a duplication of a letter within the suffix $u^{i-1,b}$, and in this case set $u^{i,b}$ to be the string obtained by applying this operation over $u^{i-1,b}$, and $u^{i,a} = u^{i-1,a}$. Note that in both cases, $cost(\mathcal{ES}^{i-1,i}) = ed\left(u^{i-1,a}, u^{i,a}\right) + ed\left(u^{i-1,b}, u^{i,b}\right)$, therefore we get from the inductive assumption that $ed\left(w\beta, u^i\right) \overset{\text{Obs.6}}{=} cost(\mathcal{ES}^{0,i}) = cost(\mathcal{ES}^{0,i-1}) + cost(\mathcal{ES}^{i-1,i}) \geq \left(ed\left(w, u^{i-1,a}\right) + ed\left(\beta, u^{i-1,b}\right)\right) + \left(ed\left(u^{i-1,a}, u^{i,a}\right) + ed\left(u^{i-1,b}, u^{i,b}\right)\right) \overset{\text{Obs.5}}{\geq} ed\left(w, u^{i,a}\right) + ed\left(\beta, u^{i,b}\right)$.

The process above generates a partition $\left(t^{*a}, t^{*b}\right) = \left(u^{r,a}, u^{r,b}\right)$ of $t = u^r$, for which $ed\left(u^i, t\right) \geq ed\left(w, t^{*a}\right) + ed\left(\beta, t^{*b}\right)$. In particular, $ed(w\beta, t) \geq \min \left\{ ed\left(w, t^a\right) + ed\left(\beta, t^b\right) \mid \left(t^a, t^b\right) \in P(t) \right\}$. On the other hand, $ed(w\beta, t) \overset{\text{Lem.8}}{\leq} \min \left\{ ed\left(w, t^a\right) + ed\left(\beta, t^b\right) \mid \left(t^a, t^b\right) \in P(t) \right\}$, and the lemma follows. □

Based on the above observations and lemmas, we now turn to prove the recursive computation given in Section "The recurrence formula", starting with Equation 7. Fix henceforth a pair of input strings s and t, each containing at least two letters. Note that for every $\alpha \in \Sigma$ and every partitions $(s^a, s^b) \in P(s)$ and $(t^a, t^b) \in P(t)$, $ed(s, t) \overset{\text{Obs.5}}{\leq} ed(s, \alpha) + ed(\alpha, t)$, and $ed(s, t) \overset{\text{Lem.8}}{\leq} ed(s^a, t^a) + ed(s^b, t^b) \overset{\text{Obs.5}}{\leq} ed(s^a, t^a) + ed(s^b, \alpha) + ed(\alpha, t^b)$, therefore

$$
ed(s, t) \leq \min \left\{ \begin{array}{l} ed(s, \alpha) + ed(\alpha, t), \\[2mm] ed(s^a, t^a) + ed(s^b, \alpha) + ed(\alpha, t^b) \end{array} \middle| \begin{array}{l} (s^a, s^b) \in P(s), \\[1mm] (t^a, t^b) \in P(t), \\[1mm] \alpha \in \Sigma \end{array} \right\}.
$$

Thus, to prove the correctness of Equation 7, it remains to show that

$$
ed(s, t) \geq \min \left\{ \begin{array}{l} ed(s, \alpha) + ed(\alpha, t), \\[2mm] ed(s^a, t^a) + ed(s^b, \alpha) + ed(\alpha, t^b) \end{array} \middle| \begin{array}{l} (s^a, s^b) \in P(s), \\[1mm] (t^a, t^b) \in P(t), \\[1mm] \alpha \in \Sigma \end{array} \right\}.
$$

From Lemma 10, there is a simple script $\mathcal{ES} = \left\langle s = u^0, u^1, \ldots, u^r = t \right\rangle$ from s to t, and in particular, there is a string u^i along $\mathcal{ES}$ such that the partial script $\mathcal{ES}^{0,i}$ is non-generating, and the partial script $\mathcal{ES}^{i,r}$ is non-reducing. Recall that $ed(s, t) = cost(\mathcal{ES}) = cost(\mathcal{ES}^{0,i}) + cost(\mathcal{ES}^{i,r}) \overset{\text{Obs.6}}{=} ed(s, u^i) + ed(u^i, t)$.

If $u^i = \beta$ for some letter β, then $ed(s, t) = ed(s, \beta) + ed(\beta, t) \geq \min \{ed(s, \alpha) + ed(\alpha, t) \mid \alpha \in \Sigma\}$. Otherwise, u^i contains at least two letters. In this case, we can write $u^i = w\beta$, where β is the last letter in u^i and w is the nonempty prefix of u^i containing all letters except for the last one. From Lemma 12, $ed(u^i, t) = ed(w\beta, t) = \min \left\{ ed(w, t^a) + ed(\beta, t^b) \mid (t^a, t^b) \in P(t) \right\}$. Symmetrically, it is possible to show that $ed(s, u^i) = \min \left\{ ed(s^a, w) + ed(s^b, \beta) \mid (s^a, s^b) \in P(s) \right\}$, and so

$$
\begin{aligned}
ed(s, t) &= ed(s, u^i) + ed(u^i, t) \\
&= \min \left\{ ed(s^a, w) + ed(s^b, \beta) \mid (s^a, s^b) \in P(s) \right\} + \\
&\quad \min \left\{ ed(w, t^a) + ed(\beta, t^b) \mid (t^a, t^b) \in P(t) \right\} \\
&\overset{\text{Obs.5}}{\geq} \min \left\{ ed(s^a, t^a) + ed(s^b, \beta) + ed(\beta, t^b) \mid (s^a, s^b) \in P(s), (t^a, t^b) \in P(t) \right\} \\
&\geq \min \left\{ \begin{array}{l} ed(s, \alpha) + ed(\alpha, t), \\ ed(s^a, t^a) + ed(s^b, \alpha) + ed(\alpha, t^b) \end{array} \middle| \begin{array}{l} (s^a, s^b) \in P(s), \\ (t^a, t^b) \in P(t), \\ \alpha \in \Sigma \end{array} \right\},
\end{aligned}
$$

concluding the proof of Equation 7.

We next continue to develop the recursive computation, considering the simpler cases where one of the input strings is either empty or contains a single letter, and the other string contains at least two letters. Let $\mathcal{ES}$ be a simple script from ε to t whose length is r. $\mathcal{ES}$ must start with an insertion of some letter α, and from Observation 6, the remainder of the script $\mathcal{ES}^{1,r}$ is an optimal script from α to t, implying the correctness of Equation 1. The correctness of Equation 4 is shown symmetrically.

Now, consider the computation of $ed(\alpha, t)$, as expressed in the last term of Equation 2. From Lemma 11, a simple script from α to t is non-reducing, and so the first operation in such a script is either the mutation of α, or some generating operation. If there is such a script in which the first operation is generating, then $ed(\alpha, t) = ed'(\alpha, t) = mut(\alpha, \alpha) + ed'(\alpha, t) \geq \min \left\{ mut(\alpha, \beta) + ed'(\beta, t) \mid \beta \in \Sigma \right\}$. Else, there is a simple script from α to t in which the first operation is the mutation of α into some letter β. Due to Lemma 9, the following operation must be a generating operation, and so the reminder of the script is an optimal script from β to t

in which the first operation is generating, implying again that $ed(\alpha, t) \geq \min\left\{mut(\alpha, \beta) + ed'(\beta, t) \mid \beta \in \Sigma\right\}$. The other direction of the inequality is shown similarly as done above for Equation 7, concluding the correctness proof of Equation 2.

We now address the correctness of Equation 3. Consider the minimum cost of a script from α to t which starts with a generating operation. Let $\mathcal{ES}$ be such a script, and let r denote its length.

For the case where the first operation in $\mathcal{ES}$ is an insertion of some letter γ after α, from Observation 6 we get that $\mathcal{ES}^{1,r}$ is a non-reducing optimal script from $u^1 = \alpha\gamma$ to t, and therefore in this case

$$
\begin{aligned}
ed'(\alpha, t) &= cost(\mathcal{ES}) = ins(\gamma) + ed(\alpha\gamma, t) \\
&\overset{\text{Lem.12}}{=} \min\left\{ins(\gamma) + ed(\alpha, t^a) \right. \\
&\qquad \left. + ed(\gamma, t^b) \mid (t^a, t^b) \in P(t)\right\} \\
&\overset{\text{Obs.5}}{\geq} \min\left\{ed(\alpha, t^a) + ed(\varepsilon, t^b) \mid (t^a, t^b) \in P(t)\right\}.
\end{aligned}
$$

The cases where the first operation in $\mathcal{ES}$ is the insertion of some letter before α, or the duplication of α, are solved similarly and imply that

$$
ed'(\alpha, t) \geq \min\left\{
\begin{array}{l}
ed(\alpha, t^a) + ed(\varepsilon, t^b), \\
ed(\varepsilon, t^a) + ed(\alpha, t^b), \\
dup(\alpha) + ed(\alpha, t^a) + ed(\alpha, t^b)
\end{array}
\middle| (t^a, t^b) \in P(t)
\right\}
$$

The other direction of the inequality is shown similarly as shown for Equation 7, concluding the proof for Equation 3. The correctness of Equations 5 and 6 is shown symmetrically.

Correctness of Algorithm 2

We next show that when the precondition of COMPUTE-MATRIX holds with respect to its input region $I_{i,k} \times I_{j,l}$, executing the procedure derives its postcondition, i.e. the procedure computes correctly all entries in the input region within EDT^α and ED.

The base case of COMPUTE-MATRIX occurs when $k = i + 1$ and $l = j + 1$. In this case, $I_{i,k} = i$ and $I_{j,l} = j$, and from the precondition we get that $EDT^\alpha[i, j] = ED[i, I_{1,j}] \otimes T^\alpha[I_{1,j}, j] \overset{\text{Eq.13}}{=} edt^\alpha(s_{0,i}, t_{0,j})$, and $ED[i, j] = \min\left\{tr(S^\alpha)[i, I_{1,i}] \otimes EDT^\alpha[I_{1,i}, j] \mid \alpha \in \Sigma\right\}$. After running line 2 of the procedure, we have from Equation 12 that $ED[i, j] = ed\left(s_{0,i}, t_{0,j}\right)$. Thus, all entries of the form $EDT^\alpha[i, j]$ and the entry $ED[i, j]$ are correctly computed, and the postcondition holds.

Else, either $k > i + 1$ or $l > j + 1$. In the case where $l - j \geq k - i$ (lines 5-8), the algorithm partitions vertically the region to be computed into two parts of approximately equal sizes. Let $h = \left\lceil \frac{j+l}{2} \right\rceil$ be the value computed in line 5 of the procedure. Note that from Item 1 of Observation 1, the fact that $EDT^\alpha[I_{i,k}, I_{j,l}] = ED[I_{i,k}, I_{1,j}] \otimes T^\alpha[I_{1,j}, I_{j,l}]$ implies that $EDT^\alpha[I_{i,k}, I_{j,h}] = ED[I_{i,k}, I_{1,j}] \otimes T^\alpha[I_{1,j}, I_{j,h}]$, and similarly $ED[I_{i,k}, I_{j,h}] =$

$\min\left\{tr(S^\alpha)[I_{i,k}, I_{1,i}] \otimes EDT^\alpha[I_{1,i}, I_{j,h}] \mid \alpha \in \Sigma\right\}$. Thus, all requirements of the precondition with respect to the region $I_{i,k} \times I_{j,h}$ are met, and the procedure is called recursively in line 6 over this region. From the postcondition of the recursive call, upon arriving to line 7 all entries in the region $I_{i,k} \times I_{j,h}$ in matrices EDT^α and ED contain the solutions for the corresponding sub-instances. In particular, it may be observed that at this point of the run, all requirements for the precondition to hold with respect to the region $I_{i,k} \times I_{h,l}$ are met, with the exception of the requirements regarding entries in the region $I_{i,k} \times I_{h,l}$ of matrices EDT^α. Again, from the precondition and Observation 1, at this stage $EDT^\alpha[I_{i,k}, I_{h,l}] = ED[I_{i,k}, I_{1,j}] \otimes T^\alpha[I_{1,j}, I_{h,l}]$ for every $\alpha \in \Sigma$. From Item 3 of Observation 1, $\min\left\{EDT^\alpha[I_{i,k}, I_{h,l}], ED[I_{i,k}, I_{j,h}] \otimes T^\alpha[I_{j,h}, I_{h,l}]\right\} = \min\left\{ED[I_{i,k}, I_{1,j}] \otimes T^\alpha[I_{1,j}, I_{h,l}], ED[I_{i,k}, I_{j,h}] \otimes T^\alpha[I_{j,h}, I_{h,l}]\right\} = ED[I_{i,k}, I_{1,h}] \otimes T^\alpha[I_{1,h}, I_{h,l}]$, and therefore after executing line 7, the precondition holds with respect to the region $I_{i,k} \times I_{h,l}$. After returning from the recursive call in line 8, all entries in the region $I_{i,k} \times I_{j,l}$ are computed, and the postcondition of the procedure is met. The correctness of the computation conducted lines 10-13 in the case where $l - j < k - i$ is shown similarly.

Note that the initial call to COMPUTE-MATRIX from line 5 of Algorithm 2 is applied over the complete region $I_{i,k} \times I_{j,l} = I_{2,n+1} \times I_{2,n+1}$. It may be observed that after the initialization in lines 3 and 4 of Algorithm 2, the precondition of COMPUTE-MATRIX is met with respect to this region. Therefore, it follows from the postcondition that once the computation terminates all entries in matrices EDT^α and ED contain the solutions for the corresponding sub-instances. In particular, $ED[n, n]$ holds the solution $ed(s_{0,n}, t_{0,n}) = ed(s, t)$, and the returned value in line 6 of Algorithm 2 is correct.

Proofs to lemmas corresponding to the EDDC algorithm for discrete cost functions

Proof of lemma 1: Matrix multiplications computed along the run of Algorithm 2 occur in lines 7 and 12 of Procedure COMPUTE-MATRIX, and additional implicit such multiplications occur when the Inside-VMT algorithm is used in Stage 1 of the algorithm. Note that in such computations, all entries in the multiplied sub-matrices already contain the computed solutions for the corresponding sub-instances. In addition, matrix multiplications conducted by the Inside-VMT algorithm are applied only over sub-matrices $A[I_{i_1,i_2}, I_{j_1,j_2}]$ such that $i_2 \leq j_1$ (see [11]), and thus, D-discreteness in matrices computed in Stage 1 need to be shown only with respect to adjacent entries $A[i, j], A[i - 1, j]$ such that $i \leq j$. In what follows, let $0 < i \leq n$ and $0 \leq j \leq n$ be two integers for n the length of s and t.

Consider first the pair of adjacent entries $ED[i,j]$ and $ED[i-1,j]$, which already contain the corresponding sub-instance solutions $ed(s_{0,i},t_{0,j})$ and $ed(s_{0,i-1},t_{0,j})$, respectively. An edit script transforming $s_{0,i-1}$ to $t_{0,j}$ can be composed by first inserting the letter s_{i-1} at the end of $s_{0,i-1}$ to obtain the string $s_{0,i}$ at cost $ins(s_{i-1})$, and then transforming $s_{0,i}$ to $t_{0,j}$ by applying an optimal script at cost $ed(s_{0,i},t_{0,j})$. Therefore, $ed(s_{0,i-1},t_{0,j}) \leq ins(s_{i-1}) + ed(s_{0,i},t_{0,j})$. Also, an edit script transforming $s_{0,i}$ to $t_{0,j}$ can be composed by first deleting last letter s_{i-1} from $s_{0,i}$ at cost $del(s_{i-1})$, and then transforming $s_{0,i-1}$ to $t_{0,j}$ at cost $ed(s_{0,i-1},t_{0,j})$. Therefore, $ed(s_{0,i},t_{0,j}) \leq del(s_{i-1}) + ed(s_{0,i-1},t_{0,j})$. Thus, $a \leq -del(s_{i-1}) \leq ed(s_{0,i-1},t_{0,j}) - ed(s_{0,i},t_{0,j}) \leq ins(s_{i-1}) < b$. Since all operation costs are integers, the cost of any edit script is an integer. Hence, after the adjacent entries $ED[i-1,j]$ and $ED[i,j]$ are computed, $ED[i-1,j] - ED[i,j] = ed(s_{0,i-1},t_{0,j}) - ed(s_{0,i},t_{0,j})$ is an integer within the interval $D = I_{a,b}$. The D-discreteness proofs for computed sub-matrices in all matrices of the form $T'^{\alpha}, T^{\alpha}, T^{\varepsilon}, S'^{\alpha}, S^{\alpha}, S^{\varepsilon}$ (as well as for the transformed matrix $tr(S^{\alpha})$) are obtained similarly.

For the matrix EDT^{α}, note that there exists an integer l^* such that $edt^{\alpha}(s_{0,i-1},t_{0,j}) \overset{\text{Eq.9}}{=} ed(s_{0,i-1},t_{0,l^*}) + ed(\alpha,t_{l^*,j})$. In addition, in the same manner as above, for the same l^* we get that $edt^{\alpha}(s_{0,i},t_{0,j}) \overset{\text{Eq.9}}{\leq} ed(s_{0,i},t_{0,l^*}) + ed(\alpha,t_{l^*,j}) \leq del(s_{i-1}) + ed(s_{0,i-1},t_{0,l^*}) + ed(\alpha,t_{l^*,j}) = del(s_{i-1}) + edt^{\alpha}(s_{0,i-1},t_{0,j})$. Similarly, it can be shown that $edt^{\alpha}(s_{0,i-1},t_{0,j}) \leq ins(s_{i-1}) + edt^{\alpha}(s_{0,i},t_{0,j})$, and so after the entries $EDT^{\alpha}[i-1,j]$ and $EDT^{\alpha}[i,j]$ are computed, $EDT^{\alpha}[i-1,j] - EDT^{\alpha}[i,j] = edt^{\alpha}(s_{0,i-1},t_{0,j}) - edt^{\alpha}(s_{0,i},t_{0,j})$ is an integer within D.

Proof of lemma 2: Consider a pair of adjacent entries $Z[i-1,j], Z[i,j]$ in Z. Let r_1 and r_2 be indices, such that $Z[i-1,j] = X[i-1,r_1] + Y[r_1,j]$ and $Z[i,j] = X[i,r_2] + Y[r_2,j]$. Then:

$$\begin{aligned} Z[i-1,j] - Z[i,j] &= X[i-1,r_1] + Y[r_1,j] - (X[i,r_2] + Y[r_2,j]) \\ &\leq X[i-1,r_2] + Y[r_2,j] - (X[i,r_2] + Y[r_2,j]) \\ &= X[i-1,r_2] - X[i,r_2] < b. \end{aligned}$$

Similarly, it can be shown that $Z[i-1,j] - Z[i,j] \geq a$. Since X and Y contain only integer entries, it follows that Z contains only integer entries, and thus $Z[i-1,j] - Z[i,j]$ is an integer within D.

Proof of lemma 3: Consider a pair of adjacent entries $Z[i-1,j], Z[i,j]$ in Z. Then:

$$\begin{aligned} Z[i-1,j] - Z[i,j] &= \min\{X[i-1,j], Y[i-1,j]\} \\ &\quad - \min\{X[i,j], Y[i,j]\} \\ &< \min\{X[i,j]+b, Y[i,j]+b\} \\ &\quad - \min\{X[i,j], Y[i,j]\} = b. \end{aligned}$$

Similarly, it can be shown that $Z[i-1,j] - Z[i,j] \geq a$. Since X and Y contain only integer entries, it follows that Z contains only integer entries, and thus $Z[i-1,j] - Z[i,j]$ is an integer within D.

Proof of lemma 4: Since both x and y are D-discrete, for every $0 < i < q$, $x_i < x_0 + ib$ and $y_0 + ia \leq y_i$. Hence, $x_i < x_0 + ib = x_0 + ib + (y_0 - y_0 + ia - ia) = (y_0 + ia) + i(b-a) - (y_0 - x_0) < y_i + q|D| - (y_0 - x_0)$. Therefore, when $y_0 - x_0 \geq q|D|$, $x_i < y_i$ for every $0 \leq i < q$.

Proofs to lemmas corresponding to the run-length encoding based EDDC algorithm

Proof of lemma 5: We show that $ed(\alpha,\beta w) \geq ed(\alpha,w) + dup(\beta)$, where the other inequalities are proven similarly.

Let r be the length of a simple script from α to βw. Observe that $r \geq 1$, since by definition $\beta w \neq \alpha$. When $r = 1$, the single operation applied over α must be a generating operation (since $w \neq \varepsilon$). As discussed in Section "Additional acceleration using run-length encoding", we may assume this operation is the duplication of α, and so $\beta = w = \alpha$. In this case, $ed(\alpha,\beta w) = dup(\alpha) = ed(s,w) + dup(\beta)$.

When $r > 1$, assume by induction the lemma holds for every instance such that the length of a simple script from the source to the target string is less than r. A simple script from α to βw is non-reducing (as shown in Section "Correctness of the recursive computation"). If there is such a script in which the first operation is a mutation, then this operation mutates α into some letter $\gamma \neq \alpha$, and the remainder of the script is a simple script of length $r-1$ from γ to βw. In this case, the inductive assumption implies that $ed(\alpha,\beta w) = mut(\alpha,\gamma) + ed(\gamma,\beta w) \geq mut(\alpha,\gamma) + ed(\gamma,w) + dup(\beta) \overset{\text{Obs.5}}{\geq} ed(\alpha,w) + dup(\beta)$. Otherwise, there is a simple script from α to βw which starts with a generating operation. Again, we may assume this generating operation is the duplication of α. As shown in Section "Correctness of the recursive computation", this implies that $ed(\alpha,\beta w) = dup(\alpha) + ed(\alpha,t^a) + ed(\alpha,t^b)$ for some partition $(t^a,t^b) \in P(\beta w)$, where the lengths of simple scripts from α to t^a and to t^b are strictly shorter than r. If $t^a = \beta$ and $t^b = w$, then $ed(\alpha,\beta w) = dup(\alpha) + ed(\alpha,\beta) + ed(\alpha,w) = dup(\alpha) + mut(\alpha,\beta) + ed(\alpha,w) \overset{\text{Const.1}}{\geq} ed(\alpha,w) + dup(\beta)$. Otherwise, t^a is of the form βu and $w = ut^b$ for some string $u \neq \varepsilon$. From the inductive assumption, $ed(\alpha,\beta w) = dup(\alpha) + ed(\alpha,\beta u) + ed(\alpha,t^b) \geq dup(\alpha) + ed(\alpha,u) + dup(\beta) + ed(\alpha,t^b) \overset{\text{Lem.8}}{\geq} dup(\alpha) + ed(\alpha\alpha,w) + dup(\beta) \overset{\text{Obs.5}}{\geq} ed(\alpha,w) + dup(\beta)$.

Proof of lemma 6: Note that if $w = \varepsilon$ or $w = \beta$ for some $\beta \in \Sigma$, then $\tilde{w} = w$, $dupcost(w) = contcost(w) = 0$, and the lemma holds in a straightforward manner.

Otherwise, w is of length at least 2, and we prove the lemma by induction over the length r of a simple script between α and w. Assume by induction the lemma holds for every pair of input strings such that the length of a simple script from the source to the target string is less than r. We show that $ed(\alpha, w) = ed(\alpha, \tilde{w}) + dupcost(w)$, where the proof that $ed(w, \alpha) = contcost(w) + ed(\tilde{w}, \alpha)$ is symmetric.

Observe that $ed(\alpha, w) \overset{\text{Obs.5}}{\leq} ed(\alpha, \tilde{w}) + ed(\tilde{w}, w) \leq ed(\alpha, \tilde{w}) + dupcost(w)$, and therefore it remains to show that $ed(\alpha, w) \geq ed(\alpha, \tilde{w}) + dupcost(w)$. As discussed in the proof of Lemma 5, there is a simple script from α to w which either starts with a mutation of α or its duplication. If the first operation in such a script is the mutation of α to some letter $\beta \in \Sigma$, then the remainder of the script is a simple script from β to w of length $r - 1$, and from the inductive assumption $ed(\alpha, w) = mut(\alpha, \beta) + ed(\beta, w) = mut(\alpha, \beta) + ed(\beta, \tilde{w}) + dupcost(w) \overset{\text{Obs.5}}{\geq} ed(\alpha, \tilde{w}) + dupcost(w)$. Otherwise, the first operation is the duplication of α, and there is some partition $(w^a, w^b) \in P(w)$ such that $ed(\alpha, w) = dup(\alpha) + ed(\alpha, w^a) + ed(\alpha, w^b)$ and the sum of lengths of shortest optimal scripts from α to w^a and to w^b is $r - 1$. From the inductive assumption, $ed(\alpha, w^a) = ed(\alpha, \tilde{w}^a) + dupcost(w^a)$ and $ed(\alpha, w^b) = ed(\alpha, \tilde{w}^b) + dupcost(w^b)$. If $(w^a, w^b) \in R(w)$, then $\tilde{w} = \tilde{w}^a \tilde{w}^b$ and $dupcost(w) \overset{\text{Eq.18}}{=} dupcost(w^a) + dupcost(w^b)$. In this case, $ed(\alpha, w) = dup(\alpha) + \left(ed(\alpha, \tilde{w}^a) + dupcost(w^a)\right) + \left(ed(\alpha, \tilde{w}^b) + dupcost(w^b)\right) \overset{\text{Lem.8}}{\geq} dup(\alpha) + ed(\alpha\alpha, \tilde{w}) + dupcost(w) \overset{\text{Obs.5}}{\geq} ed(\alpha, \tilde{w}) + dupcost(w)$. Else, $(w^a, w^b) \notin R(w)$, and so there is some letter $\beta \in \Sigma$ such that w^a ends with β and w^b starts with β. In this case, there are some strings u^a, u^b and integers $p, q > 0$, such that $w^a = u^a \beta^p$, $w^b = \beta^q u^b$, u^a does not end with β, and u^b does not start with β. Moreover, $\tilde{w}^a = \tilde{u}^a \beta$, $\tilde{w}^b = \beta \tilde{u}^b$, and $\tilde{w} = \tilde{u}^a \beta \tilde{u}^b = \tilde{w}^a \tilde{u}^b$. Note that $ed(\alpha, \tilde{w}^b) = ed(\alpha, \beta \tilde{u}^b) \overset{\text{Lem.5}}{\geq} ed(\alpha, \tilde{u}^b) + dup(\beta)$, and therefore $ed(\alpha, w) = dup(\alpha) + ed(\alpha, w^a) + ed(\alpha, w^b) = dup(\alpha) + \left(ed(\alpha, \tilde{w}^a) + dupcost(w^a)\right) + \left(ed(\alpha, \tilde{w}^b) + dupcost(w^b)\right) \geq dup(\alpha) + ed(\alpha, \tilde{w}^a) + ed(\alpha, \tilde{u}^b) + dup(\beta) + dupcost(w^a) + dupcost(w^b) \overset{\text{Eq.18}}{=} dup(\alpha) + ed(\alpha, \tilde{w}^a) + ed(\alpha, \tilde{u}^b) + dupcost(w) \overset{\text{Lem.8}}{\geq} dup(\alpha) + ed(\alpha\alpha, \tilde{w}) + dupcost(w) \overset{\text{Obs.5}}{\geq} ed(\alpha, \tilde{w}) + dupcost(w)$.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors developed the algorithms, drafted the manuscript, read and approved the final manuscript.

Author details
[1] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel. [2] Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA, USA.

References
1. Jobling MA, Heyer E, Dieltjes P, de Knijff P: **Y-chromosome-specific microsatellite mutation rates re-examined using a minisatellite, MSY1.** *Human Mol Genet* 1999, **8**(11):2117–2120.
2. Bérard S, Rivals E: **Comparison of minisatellites.** *J Comput Biol* 2003, **10**(3–4):357–372.
3. Levenshtein V: **Binary codes capable of correcting deletions, insertions and reversals.** *Soviet Physics Doklady* 1966, **10**(8):707–710.
4. Waterman M: *Introduction to computational biology: maps, sequences and genomes.* London: Chapman & Hall/CRC; 1995.
5. Needleman S, Wunsch C: **A general method applicable to the search for similarities in the amino acid sequence of two proteins.** *J Mol Biol* 1970, **48**(3):443–453.
6. Behzadi B, Steyaert JM: **An improved algorithm for generalized comparison of minisatellites.** *J Discrete Algorithms* 2005, **3**(2–4):375–389.
7. Behzadi B, Steyaert JM: **The minisatellite transformation problem revisited: A run length encoded approach.** *Lecture Notes in Comput-Sci* 2004, **3240:**290–301.
8. Bérard S, Nicolas F, Buard J, Gascuel O, Rivals E: **A fast and specific alignment method for minisatellite maps.** *Evol Bioinformatics Online* 2006, **2:**303.
9. Abouelhoda MI, Giegerich R, Behzadi B, Steyaert JM: **Alignment of minisatellite maps based on run-length encoding scheme.** *J Bioinformatics Comput Biol* 2009, **7**(2):287–308.
10. Valiant LG: **General context-free recognition in less than cubic time.** *J Comput Syst Sci* 1975, **10**(2):308–314.
11. Zakov S, Tsur D, Ziv-Ukelson M: **Reducing the worst case running times of a family of RNA and CFG problems, using valiant's approach.** *Algorithms Mol Biol* 2011, **6**(1):20. [http://www.almob.org/content/6/1/20]
12. Chan TM: **More algorithms for all-pairs shortest paths in weighted graphs.** *SIAM J Comput* 2010, **39**(5):2075–2089. 2007:590–598.
13. Williams R: **Matrix-vector multiplication in sub-quadratic time:(some preprocessing required).** In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA).* Philadelphia, PA, USA: SIAM; 2007:995–1001.
14. Frid Y, Gusfield D: **A simple, practical and complete** $O\left(\frac{n^3}{\log n}\right)$**-time Algorithm for RNA folding using the Four-Russians Speedup.** *Algorithms Mol Biol* 2010, **5:**13.
15. Nussinov R, Jacobson AB: **Fast algorithm for predicting the secondary structure of single-stranded RNA.** *PNAS* 1980, **77**(11):6309–6313.
16. Cormen TH, Leiserson CE, Rivest RL, Stein C: *Introduction to Algorithms.* Cambridge, MA: MIT Press; 2001.
17. Masek WJ, Paterson MS: **A faster algorithm computing string edit distances.** *J Comput Syst Sci* 1980, **20**:18–31.
18. Arlazarov VL, Dinic EA, Kronod MA, Faradzev IA: **On economical construction of the transitive closure of an oriented graph.** *Sov Math Dokl* 1970, **11:**1209–1210.
19. Gusfield D: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge: Cambridge University Press; 1997. ISBN0521585198.