Streaming Algorithms For Computing Edit Distance Without Exploiting Suffix Trees*

Diptarka Chakraborty
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
Kanpur, India
diptarka@cse.iitk.ac.in

Elazar Goldenberg
Charles University in Prague
Computer Science Institute of Charles University
Prague, Czech Republic
elazargold@gmail.com.

Michal Koucký
Charles University in Prague
Computer Science Institute of Charles University
Prague, Czech Republic
koucky@iuuk.mff.cuni.cz

Abstract

The edit distance is a way of quantifying how similar two strings are to one another by counting the minimum number of character insertions, deletions, and substitutions required to transform one string into the other.

In this paper we study the computational problem of computing the edit distance between a pair of strings where their distance is bounded by a parameter $k \ll n$. We present two streaming algorithms for computing edit distance: One runs in time $O(n + k^2)$ and the other $n + O(k^3)$. By writing $n + O(k^3)$ we want to emphasize that the number of operations per an input symbol is a small constant. In particular, the running time does not depend on the alphabet size, and the algorithm should be easy to implement.

Previously a streaming algorithm with running time $O(n + k^4)$ was given in the paper by the current authors (STOC'16). The best off-line algorithm runs in time $O(n + k^2)$ (Landau et al., 1998) which is known to be optimal under the Strong Exponential Time Hypothesis.

1 Introduction

The *edit distance* (aka *Levenshtein distance*) [Lev66] is a widely used distance measure between pairs of strings x, y over some alphabet Σ. It finds applications in several fields like computational biol-

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement n. 616787. The first author was partially supported by ACM-India/IARCS Travel Grant and Research-I Foundation.

ogy, pattern recognition, text processing, information retrieval and many more. The edit distance between x and y, denoted by $\Delta(x, y)$, is defined as the minimum number of character insertions, deletions, and substitutions needed for converting x into y. Due to its immense applicability, the computational problem of computing the edit distance between two given strings x and $y \in \Sigma^n$ is of prime interest to researchers in various domains of computer science. Sometimes one also requires that the algorithm finds an *alignment* of x and y, i.e., a series of edit operations that transform x into y.

In this paper we study the problem of computing edit distance of strings when given an *a priori* upper bound $k \ll n$ on their distance. This is akin to *fixed parameter tractability*. Arguably, the case when the edit distance is small relative to the length of the strings is the most interesting as when comparing two strings with respect to their edit distance we are implicitly making an assumption that the strings are similar. If they are not similar the edit distance is uninformative. There are few exceptions to this rule, most notably the reduction of instances of formula satisfiability (SAT) to instances of edit distance of exponentially large strings [BI15] where the edit distance of resulting strings is close to their length. However, such instance of the edit distance problem are rather artificial. For typical applications the edit distance of the two strings is much smaller then the length of the strings. Consider for example copying DNA during cell division: Human DNA is essentially a string of about 10^9 letters from $\{A, C, G, T\}$, and due to imperfections in the copying mechanism one can expect about 50 edit operations to occur during the process. So in many applications we can be looking for a handful of edit operations in large strings.

Landau et al. [LMS98] provided an algorithm that runs in time $O(n + k^2)$ and uses space $O(n)$ when size of the alphabet Σ is constant. In general the running time of the algorithm given in [LMS98] is $O(n \cdot \min\{\log n, \log |\Sigma|\} + k^2)$. In this paper we revisit this question and study streaming algorithms for edit distance, that is, algorithms that make only one or few passes over the input x and y. We consider so called *synchronous* streaming model where x and y are processed left-to-right in parallel at about the same rate, and the internal memory of the algorithm is limited compared to the size of the whole input. We provide two algorithms for computing edit distance that run in a streaming fashion. One of them essentially matches the parameters of the algorithm given by [LMS98], improving on them slightly, while working in streaming fashion using only $O(k)$ internal memory. The other one which we consider to be the main contribution of this paper differs slightly in its parameters but we believe it is superior in practicality.

The algorithm given by [LMS98] relies on a suffix tree machinery and builds suffix trees for the entire input. While from theoretical perspective the task of building a suffix tree requires only linear time for a constant-size alphabet, practically they are quite expensive to build because of hidden constants. Moreover, for arbitrary-size alphabets suffix trees incur super-linear cost. More specifically, the known algorithms used to build a suffix tree of a string of length n over alphabet Σ run in time $O(n \cdot \min\{\log n, \log |\Sigma|\})$ [Wei73, McC76, Ukk95].

Hence, for practical purposes people prefer the algorithm by [Ukk85] to compute edit distance, despite its running time being $O(nk)$ cf. [PP08]. The algorithm by [Ukk85] does not build suffix trees. We propose a new approach for computing edit distance, which does not involve computing suffix trees either, yet, it improves over the running time of [Ukk85] algorithm. We obtain an algorithm that makes one-pass over its input x and y, uses space $O(k)$ to compute the edit distance of x and y (space $O(k^2)$ to compute an optimal alignment of x and y) and runs in time $n + O(k^3)$. By writing $n + O(k^3)$ we want to emphasize that the number of operations per an input symbol is a small constant. (The constant in the big-O is also reasonable.) Moreover, we emphasize that running time of our algorithm is independent of the alphabet size and thus to the best of our knowledge this is the first algorithm to compute edit distance that runs in "truly" linear time for small values of k. In that regard it is an improvement over the algorithm given in [LMS98] for large alphabets. We believe that due to its simplicity it should be relevant for practice. Formally our result is as follows:

Theorem 1.1. *There is a deterministic algorithm that on input $x, y \in \Sigma^n$ and an integer k, such that $\Delta(x, y) \leq k$ outputs $\Delta(x, y)$. The algorithm accesses x and y in one-way manner, runs in $c(n + k^3)$ time while using $O(k)$ space, where c is a small constant. Moreover, one can output an optimal alignment between x and y while using extra $O(k^2)$ space. The algorithm never runs in time more than $O(kn)$.*

The algorithm from the above theorem is efficient with respect to the memory access pattern when x and y are stored in the main memory. In the cache oblivious model with memory block size B the algorithm performs only $O(\frac{n+k^3}{B})$ IO operations.

Our second result confirms that from theoretical point of view, the running time of streaming algorithms is as good as that of the [LMS98] algorithm. We even improve slightly the dependency on the alphabet size. In particular, instead of $O(n \cdot \min\{\log n, \log |\Sigma|\} + k^2)$ running time of [LMS98], we achieve $O(n \cdot \min\{\log k, \log |\Sigma|\} + k^2)$ running time. The algorithm uses only $O(k)$ space to compute the edit distance between the input strings and further $O(k^2)$ space for finding an optimal alignment.

Theorem 1.2. *There is a deterministic algorithm that on input $x, y \in \Sigma^n$ and an integer k, such that $\Delta(x, y) \leq k$ outputs $\Delta(x, y)$. The algorithm accesses x and y in one-way manner, runs in $O(n \cdot \min\{\log k, \log |\Sigma|\} + k^2)$ time while using $O(k)$ space. Moreover, one can output an optimal alignment between x and y while using extra $O(k^2)$ space.*

Previously, the best known (and only) streaming algorithm for edit distance was given in [CGK16]. That algorithm has running time $O(n + k^{1/4})$, uses space $O(k^4)$, and is substantially more complex.

All our algorithms output ∞ when run on strings which have edit distance larger than the parameter k. Hence, similarly to [CGK16], if we allow the algorithms $O(\log \log k)$ passes over the input, we do not have to provide them the parameter k. One can search for the smallest k of the form $2^{(1+\epsilon)^i}$ for which the algorithm from Theorem 1.1 returns a finite edit distance to obtain an algorithm that handles all pairs of strings with running time $O(n \log \log n + k'^{3+3\epsilon})$, where $k' = \Delta(x, y)$.

1.1 Previous work

One can easily solve the problem of computing exact edit distance (the *decision* problem) in $O(n^2)$ time using a basic dynamic programming approach [WF74]. This bound was later slightly improved by Masek and Paterson [MP80] and they achieved an $O(n^2/\log n)$ time algorithm for finite alphabets. So far this is the best known upper bound for this problem. Recently, it was shown that this bound cannot be improved significantly unless the Strong Exponential Time Hypothesis is false [BI15, BK15]. They establish this fact by providing a reduction which (implicitly) maps instances of SAT into instances of edit distance with the edit distance close to n. However, their result does not exclude the possibility of getting faster algorithms in the case of small edit distance.

Suppose we are guaranteed that the edit distance between the two input strings is bounded by $k \ll n$. Then there are algorithms that are much more efficient in terms of both time and space. Ukkonen [Ukk85] gave an algorithm to solve the decision problem in time $O(kn)$ and space $O(k)$. The same algorithm uses $O(n)$ space to find the optimal alignment (the *search* problem). Later, Landau et al. [LMS98] solved the decision problem within time $O(n \cdot \min\{\log n, \log|\Sigma|\} + k^2)$ and $O(n)$ space. By slightly modifying their algorithm the search problem can be solved as well using only $O(k^2)$ extra space. Interested readers may consult a survey by Navarro [Nav01] for a comprehensive treatment on this topic. In a very recent development, Chakraborty, Goldenberg and Koucký [CGK16] considered the search problem under the promise that the edit distance is small. They gave a single-pass algorithm that runs in time $O(n + k^4 \cdot \min\{\log k, \log|\Sigma|\})$ while using space of size $O(k^4)$. The authors also mentioned that they can remove the promise by paying a penalty in the number of passes over the input and slightly worse time and space complexity. Table 1 summarizes the above results.

Table 1: Taxonomy of Algorithms Computing Edit Distance

Authors	Time	Space
[WF74]	$O(n^2)$	$O(n)$
[MP80]	$O(n^2/\log n)$ (for finite alphabets)	$O(n)$
[LMS98]	$O(n \cdot \min\{\log n, \log|\Sigma|\} + k^2)$	$O(n)$
[Ukk85]	$O(nk)$	$O(k)$
[CGK16]	$O(n + k^4 \cdot \min\{\log k, \log|\Sigma|\})$ (randomized streaming and single pass)	$O(k^4)$
This paper	$O(n \cdot \min\{\log k, \log|\Sigma|\} + k^2)$ (streaming and single pass)	$O(k)$
This paper	$O(n + k^3)$ (streaming and single pass)	$O(k)$

The problem of computing edit distance in the streaming model has been studied first time in [CGK16]. Independently of the current paper, Belazzougui and Zhang [BZ16] give an algorithm similar to our $O(n + k^2)$ time algoritm. A related problem, namely *edit distance to monotonicity*, which is equivalent to the problem of finding *longest increasing subsequence*, has been studied extensively in streaming model [LVZ05, SW07, GJKK07, GG07, EJ08, CLL$^+$11, SS13]. However, the main focus of all of these results was to determine the upper and lower bound on the space requirement instead of time for exact as well as approximate solutions. Another important point to note is that the problem of edit distance to monotonicity is in some sense much easier because it can be solved in $O(n \log n)$ time [Sch61, Fre75] whereas general edit distance cannot be computed in strictly sub-quadratic time unless SETH is false [BI15, BK15].

Finding approximate solutions while computing general edit distance has also been studied extensively. The exact algorithm given in [LMS98] immediately gives a linear-time $\sqrt{n}$-approximation algorithm. A series of subsequent works improved this approximation factor first to $n^{3/7}$ [BYJKK04], then to $n^{1/3+o(1)}$ [BES06] and later to $2^{\widetilde{O}(\sqrt{\log n})}$ [AO09] while keeping he running time of the algorithm almost linear. Batu et al. [BEK$^+$03] gave an $O(n^{1-\alpha})$-approximation algorithm that runs in time $O(n^{\alpha/2})$. The approximation factor was further improved to $(\log n)^{O(1/\epsilon)}$, for every $\epsilon > 0$ by providing a $n^{1+\epsilon}$ time algorithm [AKO10].

1.2 Our technique

To exhibit the techniques behind our results, let us first briefly introduce the main idea behind [LMS98] and [Ukk85] algorithms. Both algorithms are based on computing the edit distance matrix for strings x and y. Ukkonen [Ukk85] shows that in order to compute the edit distance between pairs of strings of edit distance at most k, only nk values in the matrix need to be computed, and he identifies $O(k^2)$ important entries in the matrix. Developing on that, Landau et el. [LMS98] showed that using the suffix tree machinery each of these $O(k^2)$ entries can be found in $O(1)$ operations. That machinery is used in order to evaluate queries of the form: "find the largest common substring starting at positions i in x and $i+d$ in y", where $i \in [n], d \in \{-k, \dots, k\}$. We refer to such queries as slide(d, i).

Our $O(n+k^2)$ algorithm (Section 4) uses essentially the paradigm of [LMS98]. It uses suffix trees computed for blocks of size $O(k)$ of the input to compose long slides from smaller slides of size k. This saves space, and in the case of large alphabets also time. In our main algorithm (Section 5) that runs in time $n+O(k^3)$ we do not compute suffix trees at all. Instead, we implement the slide queries in the most naïve way using character by character comparison. This in general would lead to running time $O(nk)$ as in Ukkonen's algorithm. To obtain the $n+O(k^3)$-time bound we refrain from performing *long simultaneous* slides. By long simultaneous slides we mean slides slide(d, i) and slide(d', i') for which the common substrings have a large overlap. Motivated by our work in [CGK16] we show that such slides imply periodicity of the underlying substrings. We leverage this periodicity to perform only one of the two slides. This generalizes to multiple simultaneous slides while having to pay only for one of the slides. The resulting algorithm turns surprisingly simple.

2 Preliminaries

In this section we some of the main tools we use later. For the sake of presentation, through out this paper we consider both the input strings to be of the same length. However one can easily generalize all the algorithms stated in this paper for two strings of different lengths.

Notations: For an interval $[i, j]$ and a string $x \in \{0,1\}^n$ we denote by $x_{i,\dots,j}$ the substring $x_i, \dots, x_j$ and for convenience if $i \le 0$ then $x_{i,\dots,j} = x_{1,\dots,j}$, if $j \ge n$ then $x_{i,\dots,j} = x_{i,\dots,n}$, and if $i > j$ then $x_{i,\dots,j}$ is the empty string. We say that a string $x \in \Sigma^n$ is *periodic* with period size p if there exits a pattern $w \in \Sigma^p$, $p \le n/2$, and an integer $\ell \ge 2$ such that $x = w^\ell z$, where z is a prefix of w. In such a case, we refer w as *period* of x.

2.1 Dynamic programming algorithm for computing edit distance

A well known dynamic programming algorithm by [WF74] solves the problem in time $O(n^2)$. The algorithm proceeds by computing an $(n+1) \times (n+1)$-sized edit distance matrix D indexed by $\{0, \dots, n\} \times \{0, \dots, n\}$, where the $D_{i,j}$-entry stores the value $\Delta(x_{1,\dots,i}, y_{1,\dots,j})$ if $i > 0$ and $j > 0$, and $\max\{i, j\}$ otherwise. The algorithm fills in the matrix values in a lexicographic order according to the following recurrence formula:

$$D_{i,j} = \min \left\{ \begin{array}{ll} D_{i-1,j} + 1, & \text{if } i > 0 \\ D_{i,j-1} + 1, & \text{if } j > 0 \\ D_{i-1,j-1} + \delta_{x_i, y_j}, & \text{if } i, j > 0. \end{array} \right\}$$

Where $\delta_{x_i,y_j} = 0$ if $x_i = y_j$ and 1 otherwise. The recurrence formula stems from the fact that an optimal alignment for $x_{1,\ldots,i}$ and $y_{1,\ldots,j}$ can either (i) optimally align $x_{1,\ldots,i-1}$ and $y_{1,\ldots,j}$ and delete x_i, or (ii) optimally align $x_{1,\ldots,i}$ and $y_{1,\ldots,j-1}$ and delete y_j or (iii) optimally align $x_{1,\ldots,i-1}$ and $y_{1,\ldots,j-1}$ and pay additional cost of $\delta(x_i, y_j)$.

When computing the matrix in a lexicographic order the values $D_{i-1,j}, D_{i,j-1}$ and $D_{i-1,j-1}$ are already known while computing $D_{i,j}$. Hence each entry is evaluated in $O(1)$ operations, which implies the $O(n^2)$ bound on the total running time of the algorithm.

The above description can be viewed pictorially as a graph, known as *edit graph*. For any two strings $x, y \in \Sigma^n$ we define the edit graph G as follows: the set of vertices contains all pairs (i, j) where $0 \leq i, j \leq n$ and the set of edges contains an edge of cost 1 from a point (i, j) to $(i + 1, j)$ where $0 \leq i < n$ and $0 \leq j \leq n$ and from a point (i, j) to $(i, j + 1)$ where $0 \leq i \leq n$ and $0 \leq j < n$. The edge set also contains an edge of cost $\delta(x_i, y_j)$ from a point $(i-1, j-1)$ to (i, j) where $0 < i \leq n$ and $0 < j \leq n$ (See Figure 1). Note that in the above description, the cell $D_{i,j}$ corresponds to the point (i, j) in the edit graph. The problem of computing edit distance for strings x, y translates into finding the cost of a shortest path starting at $(0, 0)$ and ending at (n, n).

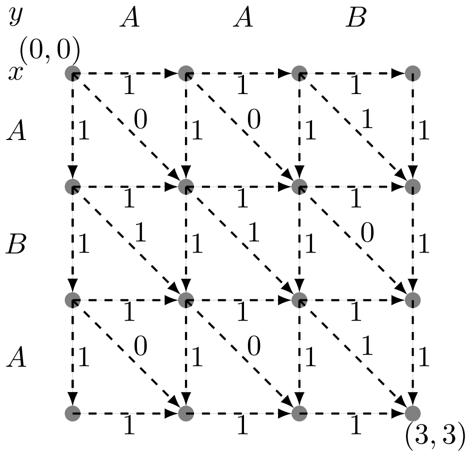

Figure 1: An example of edit graph for string $x, y \in \{A, B\}^3$

2.2 An $O(n + k^2)$-time algorithm for computing edit distance

Suppose now that we are guaranteed that $\Delta(x, y) \leq k$, can we derive an algorithm with a better running time? Ukkonen [Ukk85] provided an $O(kn)$ time algorithm by realizing that in fact the algorithm does not have to compute all the matrix D values but only the ones that reside within the diagonals $\{-k, \ldots, k\}$, where the *d-diagonal* is the set of pairs (i, j) such that $j = i + d$. Sometimes we refer to the 0-diagonal as the *main diagonal.* Notice, the values along each diagonal form a non-decreasing sequence of integers from range $\{0, \ldots, n\}$. For $h \in \{0, \ldots, n\}, d \in \{-h, \ldots, h\}$ we define $F^h(d)$ as the furthest point on the diagonal d that can be reached within h edit operations, formally:

$$F^h(d) = \max\{i : D_{i,i+d} = h\}.$$

The values $F^h(d)$ fully determine the matrix, and provided that $\Delta(x, y) \le k$, the values $F^h(d)$ for $h \le k$ determine the relevant content of the diagonals $\{-k, \dots, k\}$. In [LMS98] authors were able to compute each of these $F^h(d)$ values within $O(1)$ operations by preprocessing the input (by building a corresponding generalized suffix tree) resulting in an $O(n + k^2)$ algorithm for computing the edit distance. In the sequel, we briefly sketch their algorithm.

For a diagonal $d \in \{-h, \dots, h\}$, and for a row $i \in [n] \cup \{0\}$ we denote by $\mathrm{slide}_{x,y}(d, i)$ the furthest row $q \ge i$ that can be reached from row i on diagonal d while incurring no edit cost, formally:

$$\mathrm{slide}_{x,y}(d, i) = \max\{q : i \le q \le \min\{n, n - d\} \ \&\ x_{i+1,\dots,q} = y_{i+d+1,\dots,q+d}\}.$$

Notice that the slide *does not* compare x_i against y_i. To illustrate the definitions, consider the value $F^0(0)$, this corresponds to the size of the largest shared prefix of x and y, which equals $\mathrm{slide}_{x,y}(0, 0)$. Furthermore, observe that whenever $x_{i+1} \ne y_{i+1}$, the value $\mathrm{slide}_{x,y}(d, i)$ is i (as $x_{i+1,\dots,q} = x_{i+1,i}$ which is the empty string). Generally, the following recurrence formula holds (see [LMS98] Lemma 2.8) for $h > 0$ and $d \in \{-h, \dots, h\}$:

$$F^h(d) = \mathrm{slide}_{x,y}\left(d, \max S_{d,h}\right) \tag{1}$$

$$\text{where } S_{d,h} \text{ is the set containing } \left\{ \begin{array}{lr} F^{h-1}(d) + 1, & \text{if } |d| \le h - 1 \\ F^{h-1}(d + 1) + 1, & \text{if } d + 1 \le h - 1 \\ F^{h-1}(d - 1), & \text{if } d - 1 \ge -(h - 1), \end{array} \right\}$$

For convenience $S^0(0) = \{0\}$. The intuition behind the recurrence is that an alignment of $x_{1,\dots,F^h(d)}$ and $y_{1,\dots,F^h(d)+d}$ of cost h can be obtained by one of the three possibilities:

- Optimally align $x_{1,\dots,F^{h-1}(d)}$ and $y_{1,\dots,F^{h-1}(d)+d}$, align $x_{F^{h-1}(d)+1}$ and $y_{F^{h-1}(d)+d+1}$ (this would cost an additional edit cost as these values must be different) and slide on diagonal d, starting at $F^{h-1}(d) + 1$, or
- optimally align $x_{1,\dots,F^{h-1}(d+1)}$ and $y_{1,\dots,F^{h-1}(d+1)+d+1}$, insert the $(1+F^{h-1}(d+1))$-th character of x and slide on diagonal d starting at $F^{h-1}(d + 1) + 1$, or
- optimally align $x_{1,\dots,F^{h-1}(d-1)}$ and $y_{1,\dots,F^{h-1}(d-1)+d-1}$, insert the $(F^{h-1}(d-1)+d)$-th character of y and slide on diagonal d starting at $F^{h-1}(d - 1)$.

We define values $c_{d,h}$ to be the maximum size of $S_{d,h}$, i.e., the number of values contributing to $S_{d,h}$ counting multiplicity. Clearly, $c_{d,h} = 3$ when $|d| \le h - 2$, $c_{d,h} = 2$ when $|d| = h - 1 \ge 1$, and $c_{d,h} = 1$ when $|d| = h$ or $h = 1$. (We define $c_{0,0} = 1$ for convenience.)

We define the h-wave as the set of points: $F^h(-h), \dots F^h(0), \dots, F^h(h)$. The algorithm proposed by [Ukk85] proceeds by computing first the 0-wave, then the 1-wave and so on. The algorithm terminates whenever it encounters a wave e such that $F^e(0) = n$. The final output of the algorithm is $\Delta(x, y) = e$.

To obtain the upper bound on the running time of the algorithm the authors in [LMS98] show that the computation of $\mathrm{slide}_{x,y}(d, i)$ can be done in $O(1)$ operations. This is done by first preprocessing the input and building a generalized suffix tree for the string $x\$y\#$ where x, y are the input strings, and $\$, \#$ are characters that do not belong to the alphabet Σ and a data structure that answer a query for lowest common ancestor for this suffix tree in $O(1)$ time. Using that data structure they are able to evaluate a query $\mathrm{slide}_{x,y}(d, i)$ in $O(1)$-operations, see Section 2.3 in [LMS98].

The above implementation of [LMS98] is done in a non-streaming fashion, since the suffix tree computation requires $O(n)$ space. A natural approach to bypass this obstacle is by dividing the input strings into blocks, compute a suffix tree for each block separately so that we can compute the slide function on each block efficiently. However, the aforementioned implementation of [LMS98] computes the $F^h(d)$ values in waves. Therefore, if for some value of h the values $\{F^h(d)\}_{d\in\{-k,\dots,k\}}$ are far apart we might need to go back and forth between different blocks. Thus, we first present an algorithm for computing the values $F^d(h)$ in a different order. In our implementation slides with smaller starting row will be computed earlier. This algorithm is given in Section 3. From this algorithm we derive a streaming algorithm in Section 4. Finally, we present our main algorithm that does not computes suffix trees at all in Section 5.

We present our algorithms as computing only the edit distance. All our algorithms compute the values of $F^h(d)$ for all $|d| \leq h \leq k$. From these values, one can easily reconstruct an optimal alignment of x and y in time $O(k)$. Storing these values requires $O(k^2)$ space. If we are interested only in the edit distance (the number) then Algorithms 2 and 3 need only space $O(k)$ otherwise they need space $O(k^2)$.

3 Towards a Streaming Algorithm: a Row Modification of [LMS98]

Our goal is to design a modification of the [LMS98] algorithm that will perform all slide operations in the order of increasing starting row. Our algorithm will determine $F^h(d)$ for all values of d and h such that $|d| \leq h \leq k$, where k is a provided parameter. To determine $F^h(d)$ using (1), we need to take the maximum row of up-to three possible candidate rows obtained from values of $F^{h-1}(d-1), F^{h-1}(d)$ and $F^{h-1}(d+1)$, and perform a slide on diagonal d from that row. Our algorithm will maintain $n+1$ lists, $L_0, \dots, L_n$, each list containing entries of the form (d,h). The meaning of an entry (d,h) on a list L_i is that the slide to compute $F^d(h)$ should possibly start at row i, i.e., i is one of the three values $F^{h-1}(d-1), F^{h-1}(d)+1$ or $F^{h-1}(d+1)+1$. At a given time, each (d,h) is contained in lists $L_0, \dots, L_n$ at most once, in particular, it appears on the list L_i that corresponds to the maximum starting row i for the slide of $F^h(d)$ computed thus far. An array D of size $(2k+1)\times(k+1)$ is used to point to this unique occurrence of entry (d,h) on these lists. An array C of the same dimension is used to count the number of candidate rows for the slide of $F^h(d)$ computed thus far, i.e., the number of times entry (d,h) was put onto these lists.

Additionally, the algorithm stores a $(2k+1)\times(k+1)$ array $L^h(d)$ of values $F^h(d)$, and a generalized suffix tree for the concatenation of x and y together with a data structure to answer the lowest common ancestor query of this suffix tree in $O(1)$ time.

The algorithm proceeds as follows. It starts by adding the value $(0,0)$ to L_0, and performs a slide on diagonal 0, starting at row 0. Now suppose that this slide ended at row q, then the algorithm first updates $L^0(0) = q$ and then adds the entries $(1,1)$ into L_q and $(0,1)$ $(-1,1)$ into L_{q+1}. Now for every row $i = 0, \dots, n$: The algorithm scans the list L_i, for each entry (d,h) in the list it checks whether all the required values for slide $F^h(d)$ have been computed yet. If they already have been computed, then it performs a slide on diagonal d starting at row i. Assuming the slide ends at row q, the algorithm sets $L^h(d) = q$ and inserts $(d+1,h+1)$ into L_q and $(d,h+1)$, $(d-1,h+1)$ into L_{q+1}.

Pseudo-code for the algorithm is below:

Algorithm 1 A row modification of the [LMS98] algorithm

```
Input  : $x, y \in \{0,1\}^n$, and a parameter $k \in [n]$ such that $\Delta_e(x,y) \le k$.
Output: $\Delta_e(x,y)$
// Initialization:
Build a generalized suffix tree for the concatenation of $x$ and $y$ in order to evaluate queries $slide_d(i)$
using $O(1)$ operations, as in [LMS98].
For $i = 0, \dots, n$, initialize each list $L_i$ to be empty;
For all integers $d, h$ such that $|d| \le h \le k$, set $D(d,h) = \text{null}$ and $C(d,h) = 0$;
Invoke $Update(0,0,0)$;
// Main Loop:
for $i = 0, \dots, n$ do
    while $L_i$ is not empty do
        Pick $(d,h)$ from $L_i$ and remove it from the list;
        if $C(d,h) = c_{d,h}$ then
            $q = \text{slide}_{x,y}(d,i)$;
            $L^h(d) = q$;
            if $h < k$ then
                $Update(d, q+1, h+1)$;
                if $d < k$ then $Update(d+1, q, h+1)$;
                if $d > -k$ then $Update(d-1, q+1, h+1)$;
            end
        end
    end
end
Output the smallest $h \le k$ such that $L^h(0) = n$.
```

```
Procedure: $Update(d,i,h)$

Increment $C(d,h)$ by one.
if $D(d,h) = null$ or $D(d,h)$ points to an entry in $L_{i'}$ where $i' < i$ then
    if $D(d,h) \neq null$ then
        Delete the current node pointed to by $D(d,h)$ from $L_{i'}$;
    end
    Add $(d,h)$ into the back of $L_i$;
    Set $D(d,h)$ to point to the new entry $(d,h)$;
end
```

The algorithm satisfies two key properties captured in the next lemma.

Lemma 3.1. *Let $i \in \{0, \dots, n\}$ and let d, h be integers such that $|d| \le h \le k$.*

1. *$i \in S_{d,h}$ iff (d,h) appears on the list L_i during the run of the algorithm iff $Update(d,i,h)$ is invoked during the run of the algorithm.*

2. *If $i = \max S_{d,h}$ then while processing list L_i, the value of $L^h(d)$ is set to $F^h(d)$.*

Proof. We provide a brief sketch of an argument that proceeds by induction on h. Notice that $Update(d,i,h)$ is only invoked for values satisfying $|d| \le h \le k$. Also, the only way for (d,h)

to get on some list L_i is by invocation of *Update*(d, i, h). This proves the second 'iff' of the first part. For values $i = d = h = 0$, the first property is true because *Update*$(0, 0, 0)$ is invoked during the initialization, and the second property is true because $(0, 0)$ is on list L_0 after that, $C(0, 0) = c_{0,0} = 1$, so $\text{slide}_{x,y}(0, 0)$ will be eventually computed and $L^0(0)$ will receive the value of that slide which corresponds to $F^0(0)$.

The second property claims that the value of $L^h(d)$ is set to $F^h(d)$. In order to compute $F^h(d)$ we need to know up-to three values $F^{h-1}(d - 1), F^{h-1}(d) + 1$ and $F^{h-1}(d + 1) + 1$, and perform a slide along diagonal d from their maximum. Assuming inductively that for $L^{h-1}(d - 1), L^{h-1}(d)$ and $L^{h-1}(d + 1)$ the second property holds, when the value of $L^{h-1}(d - 1)$ is set to $F^{h-1}(d - 1)$, *Update*$(d, F^{h-1}(d - 1), h)$ is invoked and indeed, $F^{h-1}(d - 1) \in S_{d,h}$. Similarly, when $L^{h-1}(d)$ is set to $F^{h-1}(d)$, *Update*$(d, F^{h-1}(d) + 1, h)$ is invoked and $F^{h-1}(d) + 1 \in S_{d,h}$. Last, when $L^{h-1}(d+1)$ is set to $F^{h-1}(d+1)$, *Update*$(d, F^{h-1}(d+1)+1, h)$ is invoked and again, $F^{h-1}(d+1)+1 \in S_{d,h}$. (These updates happen provided $|d| \leq h \leq k$.) This means that the only way for (d, h) to get on some list L_i is if $i \in S_{d,h}$. Moreover, after the last value of the three $L^{h-1}(d - 1), L^{h-1}(d)$ and $L^{h-1}(d + 1)$ is computed during processing of some list L_i, $C(d, h) = c_{d,h}$. Furthermore, $i \leq \max S_{d,h}$ as the last *Update*(d, j, h) happens with $i \leq j \leq \max S_{d,h}$. Hence, (d, h) will appear on the list L_i that is currently processed or on some list that will be processed later. Once we reach (d, h) on list $L_{\max S_{d,h}}$, $C(d, h) = c_{d,h}$, so the $\text{slide}_{x,y}(d, \max S_{d,h})$ is computed and $L^h(d)$ is set to $F^h(d)$. This finishes the argument. □

The correctness of the output of the algorithm follows from the second part of the lemma.

Complexity analysis: Let us now discuss the time complexity of Algorithm 1. Time complexity of Algorithm 1 is determined by two main tasks. First, constructing the generalized suffix tree and lowest common ancestor data structure as in [LMS98] requires $O(n{\cdot}\min\{\log n, \log |\Sigma|\})$ time [Gus97]. Second, we need to process points present in the lists L_i for $0 \leq i \leq n$. As explained above, the processing of each point of the form (d, h) takes $O(1)$ time. By Lemma 3.1 the total number of points added in the lists throughout the run of the algorithm is bounded by $O(k^2)$ (each point can be inserted at most 3 times). Hence the overall time complexity of Algorithm 1 is $O(n \cdot \min\{\log n, \log |\Sigma|\} + k^2)$.

Algorithm 1 requires space for three purposes. First, to construct and store the generalized suffix tree we need $O(n)$ space as in [LMS98]. Second, to maintain the lists $L_0, \ldots, L_n$ and the array D we use space of size $O(n + k^2)$. Third, we need to maintain all the values of $L^h(d), C(h, d)$ for $h \in \{0, \ldots, k\}, d \in \{-k, \ldots, k\}$ and that requires total $O(k^2)$ space. Hence total space requirement is $O(n + k^2)$.

4 An $O(n + k^2)$-time streaming algorithm for computing edit distance

In this section we prove Theorem 1.2. The algorithm we present is based on the algorithm described in Section 3. The bottleneck of that algorithm was the space needed to store the suffix tree data structure for efficient implementation of slides. To eliminate the bottleneck, we divide the input strings x and y into (overlapping) blocks of length $O(k)$ and process the input block by block.

For each block we will build a suffix tree data structure so that each slide operation within the block will be evaluated in $O(1)$-time. Slide operations that span several blocks will be split into pieces of size at most k. When processing the j-th block we will process all continuing slide operations and all slide operations that start on rows between jk and $(j+1)k-1$ of the original edit distance matrix for x and y. Instead of maintaining lists $L_0, \ldots, L_n$ we will only maintain lists $L_0, \ldots, L_k$ that will contain the starting positions of slides within the current block. The list L_0 will hold the slides continuing from the previous block, L_k will maintain the slides that should continue into the next block. Whenever a slide operation continues past the $(j+1)k$ row, we will put it on the list L_k. After finishing the current block we will move list L_k to L_0 and we will process the next block.

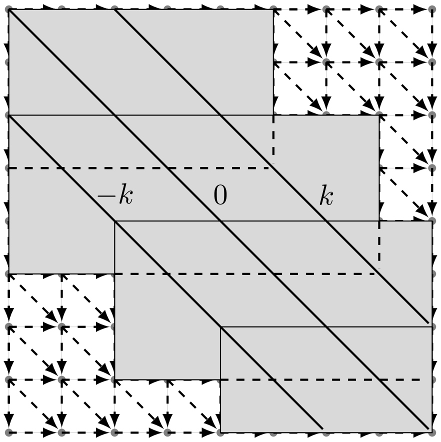

Figure 2: Sketch of a block-wise division of edit graph.

The j-th block of x will consist of $x_{jk+1,\ldots,(j+3)k+1}$ and of y will be $y_{(j-1)k+1,\ldots,(j+2)k+1}$. This provides enough context so that slides on diagonals $\{-k, \ldots, k\}$ on rows between jk and $(j+1)k-1$ of the original matrix for x and y can be computed from slides on these blocks of x and y (see Fig. 2). Diagonal $d+k$ of the edit distance matrix of x' and y' corresponds to diagonal d of the edit distance matrix of x and y.

The pseudo-code of our algorithm is below. The procedure *Update*() remains the same as in the previous section.

Algorithm 2 A streaming algorithm for computing edit distance

```
Input  : $x, y \in \{0,1\}^n$, and a parameter $k \in [n]$ such that $\Delta_e(x,y) \le k$.
Output: $\Delta_e(x,y)$
// Initialization:
For $i = 0, \dots, k$, initialize each list $L_i$ to be empty;
For all integers $d, h$ such that $|d| \le h \le k$, set $D(d,h) = \text{null}$ and $C(d,h) = 0$;
Invoke $\mathit{Update}(0,0,0)$;
// Main loop over blocks of size $O(k)$:
for $j = 0, \dots, \lceil n/k \rceil - 1$ do
    Let $x' = x_{jk+1,\dots,(j+3)k+1}$ and $y' = y_{(j-1)k+1,\dots,(j+2)k+1}$;
    if $j = 0$ then $y' = 0^k \cdot y'$;
    Build a generalized suffix tree for the concatenation of $x', y'$ in order to evaluate queries
    $\mathit{slide}_{x',y'}(d,i)$ using $O(1)$ operations, as in [LMS98].
    for $i = 0, \dots, k-1$ do
        // Process each list $L_i$ within the current block
        while $L_i$ is not empty do
            Pick the first entry $(d,h)$ stored in $L_i$ and remove it from the list;
            if $C(d,h) = c_{d,h}$ then
                $q = \text{slide}_{x',y'}(k+d, i)$;
                if $q \ge k$ then
                    // Partial slide $\to$ the slide will continue during the next block
                    Insert $(d,h)$ into the list $L_k$;
                    Set $D(d,h)$ pointing to the new entry $(d,h)$;
                end
                else
                    $L^h(d) = q + jk$;
                    if $h < k$ then
                        $\mathit{Update}(d, q+1, h+1)$;
                        if $d < k$ then $\mathit{Update}(d+1, q, h+1)$;
                        if $d > -k$ then $\mathit{Update}(d-1, q+1, h+1)$;
                    end
                end
            end
        end
    end
    Move the list $L_k$ to be $L_0$; // All lists except for $L_0$ are empty.
end
Output the smallest $h \le k$ such that $L^h(0) = n$.
```

The correctness of our new algorithm follows from the correctness of the algorithm in Section 3. The difference between the two algorithms lies only in dividing longer slides into smaller pieces.

Complexity Analysis: Time complexity of Algorithm 2 follows from the following claim.

Lemma 4.1. *In every iteration $j \in \{0, \dots, \lceil n/k \rceil - 1\}$, the total number of steps performed is bounded by $O(k \cdot \min\{\log k, \log|\Sigma|\} + k_j)$, where $\sum_j k_j = O(k^2)$.*

Now clearly the overall time complexity of Algorithm 2 is $O(n+k^2)$. It remains to prove the above claim.

Proof of Lemma 4.1. At each iteration, first we need to construct a generalized suffix tree for blocks of size $O(k)$ and a data structure for finding lowest common ancestor for that generalized suffix tree as in [LMS98] and thus we require $O(k \cdot \min\{\log k, \log |\Sigma|\})$ time [Gus97]. To set list L_0 to L_k, we need only $O(1)$ of pointer updates. Finally we need to process items stored in the lists $L_0, \ldots, L_{k-1}$. As in the case of Algorithm 1, processing each point takes only $O(1)$ time, and thus to conclude the proof it suffices to bound the number of items on these lists by $O(k+k_j)$. Let k_j be the number of items that are added to lists $L_{jk+1}, \ldots, L_{(j+1)k-1}$ during execution of Algorithm 1 on x and y. Clearly, those are precisely the items that will be added to lists $L_1, \ldots, L_{k-1}$ by Algorithm 2 during the j-th iteration. Since L_0 may contain at most $2k+1$ items and the same item can be added to any list at most three times, in total we process $O(k+k_j)$ items from lists $L_0, \ldots, L_{k-1}$ during the j-th iteration. The total number of points added (with multiplicity) to lists $L_0, \ldots, L_n$ by Algorithm 1 is $O(k^2)$, hence $\sum_j k_j = O(k^2)$. This completes the proof. □

Let us now discuss the space complexity of Algorithm 2. Algorithm 2 requires space for three purposes. First, to construct generalized suffix tree for blocks of size $O(k)$ and a data structure for finding lowest common ancestor for that generalized suffix tree as in [LMS98]. This requires $O(k)$ space [Gus97]. Second, to maintain the lists $L_0, \ldots, L_k$ and the array D we use space of size $O(k^2)$. Third, we need to maintain all the values of $L^h(d)$ and $L^h(-d)$ for $h, d \in \{0, \ldots, k\}$ and that requires total $O(k^2)$ space. Hence total space requirement is $O(k^2)$.

Reducing the space requirements for computing edit distance: If we are interested in computing only the edit distance of x and y instead of their optimal alignment, we can reduce the space used by the algorithm to $O(k)$. At any instant of time, lists $L_0, \ldots, L_n$ of Algorithm 1 contain $O(k)$ items so, the same is true for lists $L_0, \ldots, L_k$ of Algorithm 2. Indeed, at any time, for a given diagonal d, if h is maximal such that $L^h(d)$ was already set then lists $L_0, \ldots, L_n$ can contain only entries (d, h') for $h' \in \{h+1, h+2\}$. So in total the lists contain $O(k)$ items at any given time. This also means, that at any given time, arrays C and D have only $O(1)$ relevant entries for each d so they can be replaced by a $O(k)$-space data structure that maintains only the relevant entries and provides look-up and udate in $O(1)$ time. If we are interested only in the edit distance of x and y, we do not need to store $L^h(d)$ for all possible d and $h \leq k$ but we can only look for the relevant entry for $d = 0$. As the suffix tree data structure for efficient slides requires only $O(k)$ space in Algorithm 2 the total space used by the algorithm can be reduced to $O(k)$.

5 Computing Edit Distance without using Suffix Trees

While from theoretic perspective the task of building a suffix tree requires only linear time, practically they are quite expensive to build. Hence, for practical purposes people are using a straightforward implementation of [Ukk85] algorithm to compute edit distance, i.e. computing the slide function by comparing character by character, cf. [PP08]. In this section we propose a new approach for implementing [Ukk85] algorithm, which does not involve computing suffix trees but still avoids long parallel slides.

We obtain an algorithm that makes one-pass over its input x and y, uses space $O(k)$ to compute the edit distance of x and y ($O(k^2)$ space to compute an optimal alignment of x and y) and runs in time $n + O(k^3)$. By writing $n + O(k^3)$ we want to emphasize that the number of operations per an input symbol is a small constant. Indeed, to process most of the symbols of x and y we perform a single comparison for each character within a simple loop. Hence, an ideal implementation of our algorithm would just zip through most of the two strings x and y.

Our new algorithm extends Algorithms 1 and 2 but implements all the slides in the most naïve way using character by character comparison. This in general would lead to running time $O(nk)$. To bring down the cost to $n+O(k^3)$ we use ideas from [CGK16]. In particular, if we slide along two diagonals $d \le d'$ for more than $2(d' - d)$ common rows then the corresponding two parts of x and y are periodic with period $d' - d$. Hence, we do not need to slide on both of the diagonals, we may slide only on one of them and keep verifying the periodicity. Once the periodicity stops we know that at least one the two diagonals must end its slide and pay an edit operation. For two diagonals sliding in parallel, this does not give much of savings but this naturally generalizes to sliding along multiple diagonals $d_1 < d_2 < \cdots < d_\ell$ in parallel. If they slide in parallel for more than $2(d_\ell - d_1)$ rows then the corresponding parts of x and y are periodic with period $gcd(d_\ell - d_1, d_\ell - d_2, \ldots, d_\ell - d_{\ell-1})$. Again, it suffices to slide along only one of them (most conveniently along the rightmost one) and keep verifying its periodicity (see Figure 3). Once the periodicity stops, either the rightmost diagonal has to pay an edit operation, all of them have to pay, or all but the rightmost one. As the average length of a slide is $n/k \gg 4k$, for $k \ll \sqrt{n}$, this gives a noticeable advantage.

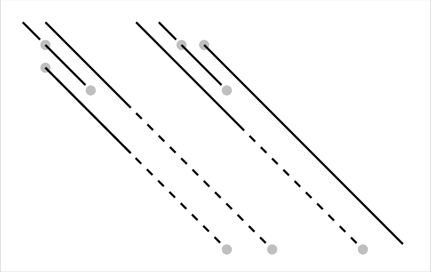

Figure 3: Illustration of the slides performed by Algorithm 3: Diagonals on which the algorithm compares character by character are marked by continuous lines; Mature diagonals which are not the right most are marked by dashed lines; Edit operations are marked by a circle.

We extend our previous algorithms using this idea. We will say that a diagonal d is *mature* at row i if $x_{i-4k+1,\ldots,i} = y_{i-4k+1+d,\ldots,i+d}$. If we have two or more diagonals that are mature at row i then we know that the corresponding parts of x and y are periodic. Our algorithm mimics Algorithm 2 in the way that it breaks each slide operation into atomic pieces of just one character slides. The algorithm maintains lists $L_0, L_1, \ldots, L_n$ that keep track of the sliding diagonals as in Algorithm 2. (At any given time, only two lists L_i and L_{i+1} are non-empty. We process diagonals on list L_i in turn which puts some diagonals on the next list L_{i+1}.) For each diagonal on list L_i we also keep track of when it got on a list for the first time, so we extend our entries (d, h) to $(d, h, \mathit{startSlide})$. If a diagonal slides for more than $4k$ steps, it is put on a special list of mature diagonals *matureList* where it will stay up until a mismatch on this diagonal occurs. (The mature diagonals will only require little attention most of the time and they are handled in *processMatureDiagonals*(i).) Procedure *processMatureDiagonals*$()$ maintains the length of the current period of x and y in *maturePeriod*, and *rightMature* stores the index of the rightmost diagonal on *matureList*. In addition to that, our algorithm maintains arrays C and D that have the same meaning as in Algorithm 2.

The algorithm processes the input strings x and y row by row of the edit distance matrix of x and y. At row i it performs two main tasks: First, it deals with the mature diagonals encountered so far. In this part, the algorithm checks whether there is a mismatch on the right most mature diagonal and whether the input strings respect the periodic pattern implied by mature diagonals. If both the conditions are met, then no edit operation is taken at row i for any of the mature diagonals. Otherwise, the algorithm identifies which are the mature diagonals that pay an edit operation and migrates them to the corresponding list L_i.

Second, the algorithm processes the list L_i. The algorithm first checks whether the current entry $(d, h, \mathit{startSlide})$ is *definite*: if it is not, then it is discarded. (The entry is *definite* if $\mathit{startSlide} = \max S_{d,h}$ which happens iff $C(d, h) = c_{d,h}$.) Otherwise, the algorithm checks for a mismatch in row $i + 1$ on diagonal d. In the case that there is a mismatch it mimics the behavior of Algorithm 1. Otherwise, it checks whether the current slide on diagonal d is long enough (by checking whether $i - \mathit{startSlide} \geq 4k$) and if it is then the algorithm migrates this entry into the mature diagonals list. Otherwise it is migrated to the list L_{i+1}.

Before we give a pseudo-code of the algorithm let us explain how we determine the periodicity implied by the mature diagonals. Our key lemma, see Corollary 5.2 below, asserts that for every row i, if diagonals $d < d'$ are mature with respect to row i, then $x_{i-4k+1,\dots,i}$ and $y_{i-4k+1+d,\dots,i+d}$ are periodic with period $d' - d$. Since this is true for every pair of mature diagonals, using properties of periodicity, this implies that the corresponding substrings of x and y are periodic with period size $p = gcd\{d_\ell - d : d \in M\}$, where M is the set of mature diagonals and d_ℓ is the right most one. A pseudo-code of the algorithm follows.

Algorithm 3 An algorithm for computing edit distance that does not use suffix trees

```
Input  : $x, y \in \{0,1\}^n$, and a parameter $k \in [n]$ such that $\Delta_e(x,y) \le k$.
Output: $\Delta_e(x,y)$
// Initialization:
Initialize all lists $L_i$ and $matureList$ to be empty;
For all integers $d, h$ such that $|d| \le h \le k$, set $D(d,h) = \text{null}$ and $C(d,h) = 0$;
Set $rightMature = \infty$ and $maturePeriod = 0$ ;
Invoke $Update(0,0,0)$;
// Main Loop:
for $i = 0, \dots, n-1$ do
    $processMatureDiagonals(i)$;
    while $L_i$ is not empty do
        Pick the first entry $(d, h, startSlide)$ stored in $L_i$ and remove it from the list;
        if $C(d,h) = c_{d,h}$ then
            if $x_{i+1} \ne y_{i+d+1}$ or $i + d + 1 > n$ then
                // Diagonal $d$ just finished sliding
                $L^h(d) = i$;
                if $h < k$ then
                    $Update(d, i+1, h+1)$;
                    if $d < k$ then $Update(d+1, i, h+1)$;
                    if $d > -k$ then $Update(d-1, i+1, h+1)$;
                end
            end
            else
                if $i - startSlide > 4k$ then $moveToMatureDiagonals(d, h, startSlide)$;
                else Insert $(d, h, startSlide)$ into $L_{i+1}$;
            end
        end
    end
end
Output the smallest $h \le k$ such that $L^h(0) = n$.
```

Procedure: *resetRightMature*(d)

```
Set $rightMature = d$ and $maturePeriod = 0$;
// Recompute the period
foreach $(d', h', startSlide') \in matureList$ do
    if $d \ne d'$ then $maturePeriod = gcd(maturePeriod, d - d')$;
end
```

```
Procedure: $moveToMatureDiagonals(d, h, startSlide)$
Add $(d, h, startSlide)$ into $matureList$, and let $D(d, h)$ point to that entry;
if $matureList$ contains only one entry then Set $rightMature = d$ and $maturePeriod = 0$;
else
    if $d > rightMature$ then $resetRightMature(d)$ ;
    else $maturePeriod = gcd(maturePeriod, rightMature - d)$ ;
end
```

```
Procedure: $processMatureDiagonals(i)$
if $matureList$ is non-empty then
    if $matureList$ contains only one entry then
        if $x_{i+1} \neq y_{i+rightMature+1}$ then
            // The unique mature diagonal has a mismatch
            Move the entry of diagonal $rightMature$ from $matureList$ to $L_i$;
        end
        return;
    end
    if $x_{i+1} = x_{i+1-maturePeriod}$ then
        if $x_{i+1} \neq y_{i+rightMature+1}$ then
            // The rightmost mature diagonal has a mismatch but none else
            Move the entry of diagonal $rightMature$ from $matureList$ to $L_i$;
            Find $d$, the current largest diagonal in $matureList$, and invoke $resetRightMature(d)$;
        end
        return;
    end
    // Mismatch on all but possibly the rightmost mature diagonals
    foreach $(d, h, startSlide) \in matureList$ do
        if $d \neq rightMature$ then
            Move the entry $(d, h, startSlide)$ from $matureList$ to $L_i$;
        end
    end
    if $x_{i+1} \neq y_{i+rightMature+1}$ then
        // Mismatch also on the rightmost diagonal
        Move the entry of diagonal $rightMature$ from $matureList$ to $L_i$;
    end
    Set $maturePeriod = 0$;
end
```

The procedure $Update(d, i, h)$ is as in Algorithm 1 except that instead of inserting (d, h) to the list L_i, it inserts (d, h, i) into that list. For convenient we define $gcd(0, a) = a$.

Correctness of Algorithm 3. To prove the correctness of our algorithm we will need the following standard facts (cf. [CR94]).

Proposition 5.1 (Cf. [CR94]). *1. Let $x \in \{0, 1\}^*$ be a string and assume x is periodic with period size p and q. Then x is periodic with period size $gcd(p, q)$.*

2. *Let $w, u, v \in \{0,1\}^*$ be such that $vw = wu$ and $|v| = |u| \leq |w|$. Then vw is periodic with a period of size $|v|$.*

3. *Let $w, u, v \in \{0,1\}^*$ be such that vw and wu are periodic with a period of the same size $\leq |v|, |u| \leq |w|$. Then vwu is also periodic with a period of the same size.*

We will make use of the following simple corollary:

Corollary 5.2. *Let $x, y \in \Sigma^n$. Let $d' > d \in \{-k, \ldots, k\}$ be diagonals, let $2(d' - d) \leq m \leq i \leq n$. If*

$$x_{i-m+1,\ldots,i} = y_{i-m+1+d,\ldots,i+d} \text{ and } x_{i-m+1,\ldots,i} = y_{i-m+1+d',\ldots,i+d'}$$

Then both $x_{i-m+1,\ldots,i}$ and $y_{i-m+1+d,\ldots,i+d}$ are periodic with a period of size $d - d'$.

Proof. Set $v = y_{i-m+1+d,\ldots,i-m+1+d'-1}$, $w = y_{i-m+1+d',\ldots,i+d}$ and $u = y_{i+d+1,\ldots,i+d'}$. Apply the second part of the previous proposition. □

We will need the following main technical lemma about mature diagonals.

Lemma 5.3. *Let $x, y \in \Sigma^n$. Let i be an integer so that $4k \leq i < n$. Let $M = \{d_1 < d_2 < \cdots < d_\ell\} \subseteq \{-k, \ldots, k\}$ be such that for each $d \in M$, $x_{i-4k+1,\ldots,i} = y_{i-4k+1+d,\ldots,i+d}$. Let $p = gcd\{d_\ell - d : d \in M\}$. Then:*

1. *$x_{i-4k+1,\ldots,i}$ and $y_{i-4k+1+d_1,\ldots,i+d_\ell}$ are periodic with period size p.*

2. *If $x_{i+1} = x_{i+1-p}$ then for all $j \in \{1, \ldots, \ell-1\}$, $x_{i+1} = y_{i+1+d_j}$.*

3. *If $x_{i+1} \neq x_{i+1-p}$ then for all $j \in \{1, \ldots, \ell-1\}$, $x_{i+1} \neq y_{i+1+d_j}$.*

Proof. For the first part. By applying the previous corollary for $d' = d_\ell$ and $d = d_j$, $j = 1, \ldots, \ell-1$, we get that $x_{i-4k+1,\ldots,i}$ is periodic with each period size $d_\ell - d_j$. By the first part of Proposition 5.1, $x_{i-4k+1,\ldots,i}$ is periodic with a period of size p, so each $y_{i-4k+1+d,\ldots,i+d}$, $d \in M$, is periodic with a period of size p. By repeated application of the third part of Proposition 5.1, $y_{i-4k+1+d_1,\ldots,i+d_\ell}$ is periodic with a period of size p.

For the second and third parts. By the first part, $y_{i-4k+1+d_1,\ldots,i+d_\ell}$ is periodic with a period of size p. Since $d_j < d_\ell$, we get $y_{i+1+d_j} = y_{i+1+d_j-p}$. By the assumption, $x_{i+1-p} = y_{i+1-p+d_j}$, so $y_{i+1+d_j} = x_{i+1-p}$. Hence, $x_{i+1} = x_{i+1-p}$ iff $x_{i+1} = y_{i+1+d_j}$. □

When running Algorithm 3 on strings x and y we can assert several properties.

Claim 5.4. *Let $i \in [n]$, at beginning of the i-th iteration, if $(d, h, startSlide)$ is on list L_i, $m = i - startSlide \leq 4k$ and $C(d, h) = c_{d,h}$ then $x_{i-m+1,\ldots,i} = y_{i-m+1+d,\ldots,i+d}$.*

The claim follows from an easy inspection of the main loop of the algorithm. The next combinatorial property justifies our handling of mature diagonals.

Claim 5.5. *Let $i \in [n]$, during the i-th iteration, after invoking the procedure processMatureDiagonals(i), the following holds:*

1. *The list matureList consists of diagonals d that were on this list at the end of iteration $i-1$ and for which $x_{i+1} = y_{i+1+d}$. For such diagonals it holds that $x_{i-4k+1,\ldots,i+1} = y_{i-4k+1+d,\ldots,i+1+d}$. Diagonals d that were stored in matureList on previous iteration for which $x_{i+1} \neq y_{i+d+1}$ were migrated to L_i.*

2. *rightMature stores the value of the largest diagonal stored in matureList.*

3. *If matureList contains at least two entries then* $maturePeriod = gcd\{rightMature - d : d \in matureList\}$*.*

Moreover, after invoking the procedure moveToMatureDiagonals(d, i, h) *items 2-3 still hold, and the list matureList contains only mature diagonals.*

Proof. For the second and third property. *rightMature* is updated whenever the rightmost diagonal leaves *matureList* or a new rightmost diagonal enters the list. Similarly, *maturePeriod* is updated to the claimed value when either the rightmost diagonal changes, a new diagonal enters *matureList*, or all the other diagonals leave the list because of a mismatch.

For the first part. By Claim 5.4 diagonal d satisfies $x_{i-4k+1,\dots,i+1} = y_{i-4k+1+d,\dots,i+1+d}$ when it is moved to the *matureList* in the main loop. Then it maintains this property inductively: If d is the rightmost diagonal then it remains on the *matureList* at iteration i if $x_{i+1} = y_{i+1+rightMature}$, and it is removed from the list otherwise. If d is not the rightmost diagonal then at iteration i either $x_{i+1} = x_{i+1-maturePeriod}$ or not. If $x_{i+1} = x_{i+1-maturePeriod}$ then the diagonal stays on *matureList* for the next iteration and by Lemma 5.3, $x_{i+1} = y_{i+1+d}$. On the other hand, if $x_{i+1} \neq x_{i+1-maturePeriod}$ then the diagonal is moved from *matureList* to L_i at iteration i, and by Lemma 5.3 it is the case that $x_{i+1} \neq y_{i+1+d}$. □

From the above we can conclude that diagonals are on *matureList* only when they are sliding. Once they stop sliding they are moved back to list L_i. List L_i maintains sliding diagonals for up-to $4k$ steps of each slide and then it moves them to *matureList* where they continue sliding, or they end their slide. Algorithm 3 mimics in this way the behavior of Algorithm 1.

5.1 Complexity Analysis

Lemma 5.6. *Let* $x, y \in \{0,1\}^n$*, be such that* $\Delta_e(x,y) \le k$*, then Algorithm 3 computes* $\Delta_e(x,y)$ *in time* $O(n + k^3)$ *using* $O(k)$ *space, and can compute the optimal alignment using* $O(k^2)$ *extra space. The algorithm can be modified to never run in time more than* $O(kn)$*.*

Proof. First, consider the procedure *resetRightMature*(). In total this procedure can be invoked at most $O(k^2)$ times as every diagonal can become a mature diagonal at most k times and we have $2k + 1$ different diagonals. The procedure has to compute the greatest common divisor of up-to $2k + 1$ numbers from the range $\{0, \dots, 2k\}$. If we use Euclid's algorithm then for each diagonal the algorithm either finishes in constant time or decreases the greatest common divisor by at least one. So the the total number of steps spent in the *gcd* computation is $O(k)$, and hence the procedure always finishes its computation with $O(k)$ steps. Thus the total time spent in procedure *resetRightMature*() is bounded by $O(k^3)$.

Similar argument gives that the algorithm spends in total at most $O(k^3)$ steps in procedure *moveToMatureDiagonals*().

Consider the procedure *processMatureDiagonals*(). The procedure runs in constant time whenever there is no need to remove any diagonal from *matureList*. Otherwise, it runs in time $O(matureList) \le O(k)$ (not counting time spent in *resetRightMature*()). In this latter case we remove at least one diagonal from *matureList* which might happen at most $O(k^2)$ times. So the total time spent in *processMatureDiagonals*() when removing some diagonal is bounded by $O(k^3)$. When not removing any diagonals we spend there $O(n)$ steps.

As for the main loop. The main loop will perform n iterations. Each iteration may involve slides of several diagonals by one step. However, each diagonal in its given slide can slide step-wise in the main loop for at most $4k$ iterations, and then it is moved to *matureList*. As each of the $2k+1$ diagonals undergoes at most k slides, the total number of slide steps performed in the main loop is bounded by $O(k^3)$.

Thus the algorithm runs in time $O(n+k^3)$, and one can verify that the amount of work needed per one of the n symbols is tiny when not contributing to the $O(k^3)$ time bound.

The algorithm can be modified so that it never runs in time more than $O(kn)$. Indeed, for each n we have to process at most $O(k)$ diagonals, and except for computing *gcd* each of them costs $O(1)$ operations. Asymptotically the most time consuming part appears to be recomputing *gcd* after every change in the rightmost mature diagonal. But this needs to be done at most once for each of the n rows. As computing the *gcd* for a given row will take at most $O(k)$ time the total running time is $O(nk)$.

The space requirements are similar to Algorithm 2, except that we do not need to build the suffix tree data structure. At each iteration $i \in [n]$ the algorithm maintains only non-empty lists $\mathit{matureList}, L_i$ and L_{i+1}, maintains arrays C and D, and accesses symbols $x_{i-2k,\dots,i+1}$ and $y_{i-k,\dots,i+k+1}$ from the input strings. The arrays C and D can be stored in $O(k)$ space as only $O(k)$ entries need to be preserved at any given time (similarly to Algorithm 2). If we are interested only in computing the edit distance but not an optimal alignment of x and y we do not have to store all the computed values of $L^h(D)$ so we need only $O(k)$ space in total. Otherwise we need $O(k^2)$ space to store all the values of $L^h(d)$. □

6 Disussion and further improvements

An optimal implementation of our algorithm can represent linked lists using fixed arrays of size $O(k)$ so that there is no need to allocate and deallocate memory for each individual item. Most of the time, more than $n - O(k^3)$ steps, the algorithm spends sliding along the rightmost mature diagonal as there are only mature diagonals sliding at those steps. An optimal implementation should take this into account, and it should be centered around sliding the rightmost mature diagonal. Concieveably, the sliding could be sped up by precomputed hashes of various substrings.

Algorithms 2 and 3 can be combined to get an algorithm running in time $\min(O(n+k^2), n+O(k^3))$: if $O(n+k^2) \leq n+O(k^3)$ run the former algorithm otherwise run the latter one. Algorithm 3 can also be modified to build a suffix tree data structure for slides for a block of the next $4k$ rows like Algorithm 2 whenever a new diagonal starts sliding. When there were only mature diagonals we would run as Algorithm 3. This would again achieve time complexity $\min(O(n+k^2), n+O(k^3))$, and perhaps even $n + O(k^2)$ assuming a certain combinatorial properties of edit distance matrices were true.

Another avenue to design an algorithm with time complexity $n + \tilde{O}(k^2)$ is by generalizing the idea of mature diagonals to diagonals of various age groups sliding for between 2^m and 2^{m+1} steps. Diagonals within a given age group would be treated similarly to mature diagonals in Algorithm 3. For each age group $2^m \leq \ell \leq 2^{m+1}$, we divide the diagonals into equal segments of size $O(\ell)$. Diagonals within a given age group belonging to the same segment would be treated as mature diagonals in Algorithm 3. In such a way we refrain from parallel sliding not only for slides of length $\Omega(k)$, but rather for any slide length, provided that the distance between the diagonals is small. We believe that this achieves running time $n + \tilde{O}(k^2)$ albeit for the cost of more complex code. We plan to include this modification in the full version of this paper.

References

[AKO10] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak, *Polylogarithmic approximation for edit distance and the asymmetric query complexity*, 51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA, 2010, pp. 377–386.

[AO09] Alexandr Andoni and Krzysztof Onak, *Approximating edit distance in near-linear time*, Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '09, ACM, 2009, pp. 199–204.

[BEK$^+$03] Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami, *A sublinear algorithm for weakly approximating edit distance*, Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '03, ACM, 2003, pp. 316–324.

[BES06] Tuğkan Batu, Funda Ergun, and Cenk Sahinalp, *Oblivious string embeddings and edit distance approximations*, Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm (Philadelphia, PA, USA), SODA '06, Society for Industrial and Applied Mathematics, 2006, pp. 792–801.

[BI15] Arturs Backurs and Piotr Indyk, *Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (New York, NY, USA), STOC '15, ACM, 2015, pp. 51–58.

[BK15] Karl Bringmann and Marvin Künnemann, *Quadratic conditional lower bounds for string problems and dynamic time warping*, IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015, 2015, pp. 79–97.

[BYJKK04] Z. Bar-Yossef, T.S. Jayram, R. Krauthgamer, and R. Kumar, *Approximating edit distance efficiently*, Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on, Oct 2004, pp. 550–559.

[BZ16] Djamal Belazzougui and Qin Zhang, *Edit distance: Sketching, streaming and document exchange*, In Proc. of IEEE Symposium on Foundations of Computer Science (FOCS 2016), 2016, p. to appear.

[CGK16] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký, *Streaming algorithms for embedding and computing edit distance in the low distance regime*, Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016, 2016, pp. 712–725.

[CLL$^+$11] Ho-Leung Chan, Tak Wah Lam, Lap-Kei Lee, Jiangwei Pan, Hing-Fung Ting, and Qin Zhang, *Edit distance to monotonicity in sliding windows*, Algorithms and Computation

- 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings, 2011, pp. 564–573.

[CR94] Maxime Crochemore and Wojciech Rytter, *Text algorithms*, Oxford University Press, 1994.

[EJ08] Funda Ergün and Hossein Jowhari, *On distance to monotonicity and longest increasing subsequence of a data stream*, Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008, 2008, pp. 730–736.

[Fre75] Michael L. Fredman, *On computing the length of longest increasing subsequences*, Discrete Mathematics **11** (1975), no. 1, 29 – 35.

[GG07] Anna Gál and Parikshit Gopalan, *Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence*, 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings, 2007, pp. 294–304.

[GJKK07] Parikshit Gopalan, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar, *Estimating the sortedness of a data stream*, Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007, 2007, pp. 318–327.

[Gus97] Dan Gusfield, *Algorithms on strings, trees, and sequences - computer science and computational biology*, Cambridge University Press, 1997.

[Lev66] VI Levenshtein, *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*, Soviet Physics Doklady **10** (1966), 707.

[LMS98] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt, *Incremental string comparison*, SIAM J. Comput. **27** (1998), no. 2, 557–582.

[LVZ05] David Liben-Nowell, Erik Vee, and An Zhu, *Finding longest increasing and common subsequences in streaming data*, Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings, 2005, pp. 263–272.

[McC76] Edward M. McCreight, *A space-economical suffix tree construction algorithm*, J. ACM **23** (1976), no. 2, 262–272.

[MP80] William J. Masek and Michael S. Paterson, *A faster algorithm computing string edit distances*, Journal of Computer and System Sciences **20** (1980), no. 1, 18 – 31.

[Nav01] Gonzalo Navarro, *A guided tour to approximate string matching*, ACM Comput. Surv. **33** (2001), no. 1, 31–88.

[PP08] Dimitrios P. Papamichail and Georgios P. Papamichail, *Improved algorithms for approximate string matching (extended abstract)*, CoRR **abs/0807.4368** (2008).

[Sch61] C. Schensted, *Longest increasing and decreasing subsequences*, Canadian Journal of Mathematics **13** (1961), 179–191.

[SS13] Michael E. Saks and C. Seshadhri, *Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance*, Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013, 2013, pp. 1698–1709.

[SW07] Xiaoming Sun and David P. Woodruff, *The communication and streaming complexity of computing the longest common and increasing subsequences*, Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007, 2007, pp. 336–345.

[Ukk85] Esko Ukkonen, *Algorithms for approximate string matching*, Inf. Control **64** (1985), no. 1-3, 100–118.

[Ukk95] Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.

[Wei73] Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973) (Washington, DC, USA), SWAT '73, IEEE Computer Society, 1973, pp. 1–11.

[WF74] Robert A. Wagner and Michael J. Fischer, *The string-to-string correction problem*, J. ACM **21** (1974), no. 1, 168–173.