Edit Distance with Block Operations

Michał Gańczorz
Institute of Computer Science, University of Wrocław, Poland
mga@cs.uni.wroc.pl

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland
gawry1@gmail.com

Artur Jeż
Institute of Computer Science, University of Wrocław, Poland
aje@cs.uni.wroc.pl

Tomasz Kociumaka
Institute of Informatics, University of Warsaw, Poland
kociumaka@mimuw.edu.pl

Abstract

We consider the problem of edit distance in which block operations are allowed, i.e. we ask for the minimal number of (block) operations that are needed to transform a string s to t. We give $\mathcal{O}(\log n)$ approximation algorithms, where n is the total length of the input strings, for the variants of the problem which allow the following sets of operations: block move; block move and block delete; block move and block copy; block move, block copy, and block uncopy. The results still hold if we additionally allow any of the following operations: character insert, character delete, block reversal, or block involution (involution is a generalisation of the reversal). Previously, algorithms only for the first and last variant were known, and they had approximation ratios $\mathcal{O}(\log n \log^* n)$ and $\mathcal{O}(\log n (\log^* n)^2)$, respectively. The edit distance with block moves is equivalent, up to a constant factor, to the common string partition problem, in which we are given two strings s, t and the goal is to partition s into minimal number of parts such that they can be permuted in order to obtain t. Thus we also obtain an $\mathcal{O}(\log n)$ approximation for this problem (compared to the previous $\mathcal{O}(\log n \log^* n)$).

The results use a simplification of the previously used technique of locally consistent parsing, which groups short substrings of a string into phrases so that similar substrings are guaranteed to be grouped in a similar way. Instead of a sophisticated parsing technique relying on a deterministic coin tossing, we use a simple one based on a partition of the alphabet into two subalphabets. In particular, this lowers the running time from $\mathcal{O}(n \log^* n)$ to $\mathcal{O}(n)$. The new algorithms (for block copy or block delete) use a similar algorithm, but the analysis is based on a specially tuned combinatorial function on sets of numbers.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis

Keywords and phrases Edit distance, Block operations, Common string partition

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.33

Funding The work of the first and third author was supported under National Science Centre, Poland, project number 2014/15/B/ST6/00615. The work of the fourth author was supported under National Science Centre, Poland, project number number 2014/13/B/ST6/00770.

1 Introduction

In the *edit distance problem*, which is one of the most iconic problems in the field of string algorithms, we are given two strings and a set of allowed operations, and we ask for the minimum number of operations needed to transform one of the strings into the other. Classically, we allow single-letter operations (usually: character insert, delete, and replace), but it seems that block operations, in which the whole substrings of the input can be edited in one operation, are as important and practical.

In the classical setting, when character operations are allowed, edit distance is computable in quadratic time, and achieving strongly subquadratic time is unlikely [2]. Allowing block deletion does not make the problem substantially harder, and a polynomial-time algorithm for this variant is known [21, 20].

The variant with other block operations was first considered by Lopresti and Tomkins [13], who showed NP-hardness of edit distance with block moves, as well as with block moves and block deletions. The former problem was approximated within an $\mathcal{O}(\log n \operatorname{poly}(\log^* n))$ factor [7], which was later improved to $\mathcal{O}(\log n \log^* n)$ [6]. A slightly worse approximation ratio of $\mathcal{O}(\log n (\log^* n)^2)$ is known when we allow block move, block copying, block uncopying and block reversal [15, 16]; while all those algorithms did not explicitly allow character edits (insert, delete, replace), it is clear from their analysis that those can also be accommodated. A variant with block move and block delete was considered and some structural properties were shown [20], but in the end no approximation algorithm was given.

The three mentioned approximation algorithms are all based on the locally sensitive parsing technique, which has roots in the deterministic coin tossing by Cole and Vishkin [5] and was used previously in the context of string algorithms in general [18] and comparing strings in particular [17, 14, 1]. In this method, we partition the string into constant-length blocks such that for each letter we can decide whether it begins or ends a block based only on the $\mathcal{O}(\log^* n)$-size neighbourhood of this letter. Then we label the blocks with new symbols and iterate the process. It turns out that to approximate the edit distance between two strings, it is enough to count the difference between the numbers of labels that appear during this (iterated) process; this is turn can be abstracted as calculating the ℓ_1 norm between embeddings into a vector space.

Surprisingly, allowing *both* block deletion and block copy makes approximation of the edit distance simpler: there are $\mathcal{O}(1)$ approximation algorithms for this problem [8, 19]. Those are based on a different approach, though: in essence they parse the target into phrases using the LZ77 algorithm, copy the phrases from the source, and then delete the source.

The edit distance with move operations problem is equivalent (up to a constant coefficient) to a *common string partition problem*, which was investigated on its own due to its connections with the computational biology, however, often in variants that are not so well motivated in terms of edit distance. For instance, it was shown to be fixed parameter tractable [3] and its restricted variant is known to be NP-hard but at the same time approximable up to a constant factor [10]; heuristics for this problem were also analysed [4].

Our contribution. We present $\mathcal{O}(\log n)$ approximation algorithms for the edit distance problem with the following set of (block) operations: block move; block move and block delete; block move and block copy; block move, block copy and block uncopy. Our algorithms work also when an arbitrary subset of the following operations is also allowed: character insert, character delete, character replace, block involution. (Involution, also known as antimorphism, is a generalisation of reverse: it reverses the string and then replaces each letter a with

$f(a)$, where f is a given bijection on the letters such that $f(f(a)) = a$, note that f can be the identity.) The first algorithm improves upon the previously known $\mathcal{O}(\log n \log^* n)$ approximation ratio [6], while the last one – the $\mathcal{O}(\log n (\log^*)^2 n)$ ratio [15, 16]. The second variant was considered to no avail [20]; to the best of our knowledge, the third variant has not been considered before.

The algorithms for the cases when only block move or block move, copy and uncopy are allowed, are similar as before [6, 15, 16], but instead of the sophisticated locally consistent parsing based on the deterministic coin tossing, we use a simpler one which is based on a partition of the alphabet into two parts. Such approaches were recently investigated [11, 9].

The presented version of the parsing is much simpler than previously used and allows for the removal of the multiplicative $\log^* n$ factors from the approximation ratios. It also enables a more general treatment of involution instead of reversal.

The algorithm for block moves and block delete is almost the same as in the case when only block moves are allowed. However, the analysis employs complex combinatorial functions defined on sets of lengths of letter repetitions. Unlike the previously used embedding to ℓ_1 spaces, this function depends on *both* strings and cannot be computed separately for each of them. The algorithm for the variant with block move and block copy uses the same function in the analysis, but in contrast to other presented algorithms (as well as the previously known ones), it is no longer a simple greedy algorithm. It constructs the sequence of operations in two steps: in the second one, the earlier copy operations may be revoked and move operations may be forced.

Our algorithms can be generalised to the case when the input is given in a grammar-compressed form: then its running time becomes $\mathcal{O}(n \log N)$, when n is the compressed size of the input and N the sum of lengths of the decompressed strings.

To streamline the presentation, in the extended abstract we give the algorithms in the variant when the involution is not allowed. The generalisation to the case with involution is natural, though tedious.

2 Definitions and basic reductions

A *string* is a sequence of elements, called *letters*, from a finite set, called *alphabet* and usually denoted by Σ, and it is denoted as $w = w_1 w_2 \cdots w_k$, where each w_i is a letter; the *length* $|w|$ of such a string w is k. For any two strings $w = w_1 \cdots w_k$ and $w' = w_{k+1} \cdots w_{k+\ell}$, their *concatenation* is $ww' = w_1 \cdots w_{k+\ell}$. A string v is a *substring* of w if there exist strings w', w'' such that $w = w'vw''$, it is a *prefix* if $w = vw''$ and a *suffix* if $w = w'v$. The empty string, i.e. the one of length 0, is denoted by ϵ. For a letter a and a string w, the number of occurrences of a in w is denoted $|w|_a$.

Given two strings s, t their *edit distance* is the minimum number of operations needed to transform s to t. The usual operations are insert (ins) and delete (del): the former turns a string $s = s_1 s_2$ to $s_1 a s_2$ and the latter $s' = s_1 a s_2$ to $s_1 s_2$ for arbitrary letter a and strings s_1, s_2. Replace, which replaces a single letter with another, is usually considered as well, but it can be simulated by insert and delete, so we ignore it later on. Other operations include block copy (called copy for short, cp), block move (called move for short, mv) and block delete (b-del), which can transform $s = s_1 s_2 s_3 s_4$ to, respectively, $s_1 s_2 s_3 s_2 s_4$ or $s_1 s_3 s_2 s_3 s_4$, $s_1 s_3 s_2 s_4$ and $s_1 s_3 s_4$, for arbitrary strings s_1, s_2, s_3, s_4. The block uncopy (called uncopy for short, uncp) is the inverse operation to copy, i.e. it can transform any $s_1 s_2 s_3 s_2 s_4$ to $s_1 s_2 s_3 s_4$ or $s_1 s_3 s_2 s_4$ for arbitrary strings $s_1, \ldots, s_4$. By $\mathrm{ED}_{\mathsf{Op}}(s, t)$ we denote the minimal number of operations from the set Op that transform s to t, where $\mathsf{Op} \subseteq \{\mathsf{ins}, \mathsf{del}, \mathsf{cp}, \mathsf{mv}, \mathsf{uncp}, \mathsf{b\text{-}del}\}$,

and the edit distance with operations Op problem asks, for given strings s and t, to find the sequence of $\mathrm{ED}_{\mathsf{Op}}(s,t)$ operations that transforms s to t. Note that the "edit distance" is a distance only when block deletion is not allowed and for each operation its inverse is also allowed. Nevertheless, in each case ED does satisfy the *(directed) triangle inequality*: $\mathrm{ED}_{\mathsf{Op}}(s,t) + \mathrm{ED}_{\mathsf{Op}}(t,\ell) \geq \mathrm{ED}_{\mathsf{Op}}(s,\ell)$. Still, we use the name *distance* for historic reasons.

For two strings s and t, their *common partition with operations* is a representation $s = s_1 s_2 \cdots s_{d_s}$ and $t = t_1 t_2 \cdots t_{d_t}$ with two sets of indices $I_s \subseteq [1\,..\,d_s]$ and $I_t \subseteq [1\,..\,d_t]$ (equal to $[1\,..\,d_s]$ and $[1\,..\,d_t]$, respectively, unless otherwise stated), and a bijection $f : I_s \to I_t$ such that $s_i = t_{f(i)}$ for each $i \in I_s$; we say that parts s_i and $t_{f(i)}$ are *matched*. The *size* of such a partition is $d_s + d_t$. Depending on the allowed operations, we may relax some of those requirements and give new ones:

delete If deletion of single letters is allowed (del), then we allow $I_s \neq [1\,..\,d_s]$ but require that $|s_i| = 1$ for $i \notin I_s$. We say that such letters are *deleted*.

insert If insertion of single letters is allowed (ins), then we allow $I_t \neq [1\,..\,d_t]$ but require that $|t_i| = 1$ for $i \notin I_t$. We say that such letters are *inserted*.

block-delete If block-deletion is allowed (b-del), then we allow $I_s \neq [1\,..\,d_s]$. This operation supersedes deletion. We say that such blocks are *deleted*.

By $\mathrm{CP}_{\mathsf{Op}}(s,t)$ for $\mathsf{Op} \subseteq \{\mathsf{del}, \mathsf{ins}, \mathsf{b\text{-}del}\}$, we denote the minimal size of the common partition with operations Op for s and t. In the minimum common string partition with operations Op problem, we want to compute, for the given strings s and t, their partition of minimal size and the corresponding function f.

Note that the different names for deletion and insertion of letters are chosen for consistency between the common partition and the edit distance problems. In the later sections, we will consider a common string partition (without operations) problem generalised to two sets of strings, which is defined in the obvious way.

It is folklore knowledge that edit distance with move operations corresponds to a common partition; more precisely, it is within constant factor of the minimal common partition. Moreover, the same holds when block deletion and/or character operations are allowed.

▶ **Lemma 1.** *For any set of operations* $\mathsf{Op} \subseteq \{\mathsf{del}, \mathsf{ins}, \mathsf{b\text{-}del}\}$*, there is a constant* c_{Op} *such that for any strings* s, t*:*

$$\mathrm{ED}_{\{\mathsf{mv}\}\cup\mathsf{Op}}(s,t) \leq \mathrm{CP}_{\mathsf{Op}}(s,t) \leq c_{\mathsf{Op}}\,\mathrm{ED}_{\{\mathsf{mv}\}\cup\mathsf{Op}}(s,t)\ .$$

Moreover, this correspondence is effective: given a sequence of d *operations from* Op *that transform* s *to* t*, we can compute the common partition with* Op *of* s *and* t *of size at most* $c_{\mathsf{Op}}d$*, and given a common partition with* Op *of size* d*, we can compute a sequence of* d *operations from* Op *that transform* s *to* t*.*

As approximation algorithms given in this work have approximation factors $\mathcal{O}(\log|st|)$, due to Lemma 1 we will content ourselves with considering one or the other problem of edit distance or common partition, depending on whichever is easier to argue about.

3 Locally consistent parsing

A *parsing* of a string s is a sequence $s_1, \ldots, s_k$ such that $s = s_1 s_2 \cdots s_k$; the strings $s_1, \ldots, s_k$ are called *phrases*, the integer k is the *size* of this parsing, and we say that s is *parsed* into $s_1, \ldots, s_k$. Given a substring t of s, we say that it is *parsed into* $s_i, \ldots, s_j$ when $s_i \cdots s_j$ contain this occurrence of t while $s_{i+1} \cdots s_j$ and $s_i \cdots s_{j-1}$ do not. A *parsing scheme* is a way of producing parsings for strings. We consider parsing schemes given by a pair of disjoint alphabets $\Sigma_0, \Sigma_1 \subseteq \Sigma$. This defines a parsing in the following way:

repetitions We group into a phrase each maximal *repetition* a^ℓ with $a \in \Sigma$ and $\ell > 1$.
pairs We group each $ab \in \Sigma_0\Sigma_1$ into a phrase.
All the remaining letters form length-1 phrases.

Next, we construct a new alphabet which has a letter for each constructed phrase.

Using the new alphabet, a parsing of w gives raise to a new string w', which is obtained by replacing phrases of length greater than 1 by their new symbols. This is called a *signature* of the string w and denoted by $\mathrm{sig}(w)$. Note that the signature depends on the parsing scheme (i.e. Σ_0 and Σ_1) as well as on the chosen symbols; the former is always clear from the context and the latter is ignored as the exact choice is irrelevant as long as it is consistent.

Given a letter a, its expansion $\exp(a)$ is the phrase that it replaced and its full expansion $\mathrm{Exp}(a)$ is the substring of the original text that it represents, which is obtained by iterative application of exp. This is generalised to strings in the obvious way.

Given two strings, we can find in linear time a parsing scheme which replaces those string with signatures that are shorter by a constant fraction.

▶ **Lemma 2.** *Given two strings s, t over an alphabet Σ we can find in time $\mathcal{O}(|s| + |t| + |\Sigma|)$ a parsing scheme of size at most $\frac{11}{12}|st| + \frac{1}{3}$ and produce the corresponding signatures.*

The proof is a variant of a known construction [11, 9]. The idea is that when Σ is randomly partitioned into Σ_0 and Σ_1, then among every two consecutive letters, with constant probability at least one is going to be parsed into a phrase of length two or more. Case inspection shows that the claim holds in expectation, and we can derandomise the procedure using the conditional expectations.

We call the parsing from Lemma 2 the *parsing for s, t*; given that there could be many such parsings, we choose one arbitrarily. We iterate the parsing process for two strings until they are reduced to single letters: an *iterated parsing scheme* is a sequence of parsing schemes $(\Sigma_{0,1}, \Sigma_{1,1}), (\Sigma_{0,2}, \Sigma_{1,2}), \ldots, (\Sigma_{0,\ell}, \Sigma_{1,\ell})$; its *height* is ℓ. Given a string s, an iterated parsing scheme defines a sequence of signatures $s = \mathrm{sig}_0(s), \mathrm{sig}_1(s), \mathrm{sig}_2(s), \ldots, \mathrm{sig}_\ell(s)$, in which $\mathrm{sig}_i(s)$ is the signature of $\mathrm{sig}_{i-1}(s)$ according to parsing scheme $(\Sigma_{0,i-1}, \Sigma_{1,i-1})$, where $\mathrm{sig}_{i-1}(s)$ is a string over $\Sigma_{i-1} = \Sigma_{0,i-1} \cup \Sigma_{1,i-1}$. Note that Σ_i and Σ_j for $i \neq j$ are not necessarily disjoint and in constructions they are usually not: not all letters from a string are replaced with their signatures, and so we want to replace them later on. We say that a letter a is from the ith level, or simply an i-letter, if $a \in \Sigma_i \setminus \bigcup_{j<i} \Sigma_j$.

▶ **Lemma 3.** *Given two strings s, t, there is an iterated parsing scheme of height $\mathcal{O}(\log |st|)$ such that $\mathrm{sig}_\ell(s)$ and $\mathrm{sig}_\ell(t)$ are letters.*

We call this parsing scheme the *parsing scheme for s, t*.

A parsing scheme is *locally consistent* if different occurrences of v (in the same or different strings) are parsed into the same phrases, possibly except $\mathcal{O}(1)$ beginning and ending phrases. Formally, if different occurrences of v are parsed into $s_1, \ldots, s_i$ and $s'_1, \ldots, s'_{i'}$, then there are $b, b', e, e' \in \mathcal{O}(1)$ such that the sequences of phrases $s_{1+b}, \ldots, s_{j-e}$ and $s'_{1+b'}, \ldots, s'_{j'-e'}$ are equal (in particular they have the same length).

▶ **Lemma 4.** *A parsing scheme defined by a partition of the alphabet are locally consistent.*

4 Approximation via embedding into normed vector spaces

Idea. While different occurrences of the same substring in s, t may be parsed differently by an iterated parsing scheme, the same symbol always fully expands to the same substring of the original strings s, t. This leads to a natural meta-algorithm for the common partition

(and the edit distance with block moves) for s, t, which was first proposed by Cormode and Muthukrishnan [6] (earlier work [7] used a similar though more involved approach): given s, t calculate their iterated parsing scheme and set of signatures $s_0, \ldots, s_k$ and $t_0, \ldots, t_k$, where $k = \mathcal{O}(\log(st))$, and then iteratively look at i-symbols for $i = k, k-1, \ldots$. If there are common symbols in s_i and t_i, then make corresponding full expansions in s, t as parts and match them to each other. For the remaining symbols, expand them to phrases in s_{i-1} and t_{i-1}. The algorithm depends only on the number of occurrences of each i-symbol in s_i an t_i; thus, we can represent s, t as vectors of counts of occurrences of letters in appropriate signatures. Amortised analysis shows that the size of the resulting partition (the number of edit moves) is within a constant factor of the ℓ_1 norm of the difference of the vectors for s and t. As a last step, one argues that this difference is at most $\mathcal{O}(k)$ times the size of the minimal common partition (the edit distance), which is shown by induction on the edit distance value. Adding insertion and deletion as allowed operations keeps the whole scheme more or less the same; in particular, we still use the ℓ_1 norm.

Unfortunately, allowing more operations (and in particular their combinations) distorts this approach. The needed modifications are explained at appropriate places.

Embedding to normed vector spaces. Given a string s and an iterated parsing scheme $(\Sigma_{0,i}, \Sigma_{1,i})_{i=1}^{k}$, let $s = s_0, \ldots, s_k$ be the sequence of its signatures and let $\Sigma_i = \Sigma_{0,i} \cup \Sigma_{1,i}$. We embed s into a vector space whose coordinates are indexed with elements of $\bigcup_{i=0}^{k} \Sigma_i$: for an i-letter a, we set $V(s)[a] = |s_i|_a$, i.e. the number of occurrences of the letter a in the appropriate signature of s (the first one to use letter a). Define a symmetric difference $V(s) \triangle V(t)$ of such vectors as

$$(V(s) \triangle V(t))[a] = \lceil |V(s)[a] - V(t)[a]| \rceil \ .$$

Note that taking the ceiling is not needed, as coordinates are natural numbers, but it is used for vectors defined later on. We also define the *support* $\sup(v)$ of a vector, in which every non-zero component of v is replaced with 1, i.e. $\sup(v)[a] \in \{0, 1\}$ and $\sup(v)[a] = 0 \iff v[a] = 0$. Lastly, the standard ℓ_1 norm is the sum of its coordinates (which are all non-negative): $\|V(s)\|_1 = \sum_a V(s)[a]$. Define also $V_i(s)$ that restricts $V(s)$ to coordinates in $\bigcup_{j \le i} \Sigma_j$

Algorithms. We now give the algorithms for several variants of the edit distance with operations and bound their sizes in terms of vectors related to input strings. The basic case is when the move operation is allowed; it serves as a model for other algorithms.

Common partitions for repetitions. Our algorithms try to match identical symbols in two signatures, yet it is more beneficial to match long repetitions instead of single letters, i.e. partition repetitions into subrepetitions such that those of larger lengths can be matched using less parts. It turns out that this is a variant of the original common partition problem; we state the simple result for later reference.

▶ **Lemma 5.** *For two sets S, T of a repetitions with, respectively, n_S and n_T repetitions and having the same sum of lengths of repetitions, there exists a common partition between S and T of size at most $2n_S + 2n_T$.*

It is enough to match any repetition from one set to a prefix in the other.

Move. AlgMove works as follows: We compute the iterated parsing scheme for s, t, the corresponding signatures $s_0, \ldots, s_k$ and $t_0, \ldots, t_k$, and the vectors $V(s)$ and $V(t)$. In the same time bounds, we can also create the list of occurrences of a in s_i and t_i for each i-letter a. Also, for each letter in the signature, we compute the beginning and the end of the corresponding full expansions in the original string.

During the algorithm, we consider the strings s_i, t_i for $i = k, \ldots, 1, 0$. We colour some letters of s_i, t_i black, such that the multisets of black coloured letters in s_i, t_i are the same.

Initially, there are no coloured letters in s_k, t_k. For each i we proceed as follows: first, we consider s_i, t_i with the coloured letters removed, which yields two sets of strings, called S and T, respectively. For each i-letter a, take the sets of all maximal a-repetitions in S and T (which includes those of length 1, i.e. single letters a). Let their total sum of lengths be ℓ_s, ℓ_t, respectively, and let $\ell = \min(\ell_s, \ell_t)$. Choose among those two sets (sub)repetitions with total length ℓ (we take all repetitions from one of the sets, while in the other we may need to split one repetition). Let the numbers of the chosen repetitions be n_s, n_t, respectively. Using Lemma 5, we find a common partition for them of size at most $2(n_s + n_t)$. We then colour those letters black, remove them from S, T and declare their full expansions in s and t as parts and map the ones in s to t. If the removal happens in the middle of some string in $S \cup T$, then this string is split into two strings and both are added back to the appropriate set. After that, we expand each i-letter to the corresponding phrase in s_{i-1} or t_{i-1}; the expansion is black coloured if and only if this letter is black coloured.

After processing 0-letters, the final actions depend on the allowed operations: if there are any uncoloured letters in $s_0 = s$, then we delete them or reject if deletion is not allowed; similarly, if there are any uncoloured letters in $t_0 = t$, then we insert them or reject if insertion is not allowed.

▶ **Lemma 6** (cf. [6]). *Given an iterated parsing scheme for strings s and t,* AlgMove *constructs in linear time a common partition of size $\mathcal{O}(\|V(s) \triangle V(t)\|_1)$.*

Proof.

▶ **Claim.** *When the algorithm processes s_i, for each a the number of black coloured letters a in s_i and t_i is the same.*

This is true when there are no coloured letters; we show that this number changes in the same way for s and t. When we expand the letters, by the inductive assumption the multiset of black-coloured letters is the same in s_i and t_i. Each such letter is replaced with the same expansion, so the claim holds also after the expansion. When we colour letters, we do it on the same (multi)sets of letters in s_i, t_i.

Claim 4 implies that after processing an i-letter a, but before the expansion, the number of uncoloured a's in s_i, t_i is exactly $(V(s) \triangle V(t))[a]$: those uncoloured letters are exactly in one of s_i, t_i and the coloured letters have the same number of occurrences in s_i and t_i.

Concerning the cost, we assume that the creation of one part in the common partition consumes one unit of credit. We keep the invariant that right before processing i-letters, each repetition in S and T (including length-1 repetitions) has 2 units of credit. The credit is spent when the partition is formed: The common partition of repetitions costs on average 2 per paired repetition, which is paid by the credit on this repetition. After the processing, the unused credit on the repetitions of i-letters is discarded and 4 fresh units of credit are issued to each i-level symbol that has not been removed (i.e., coloured black). Recall that a fixed i-letter a has exactly $(V(s) \triangle V(t))[a]$ such occurrences, so in in total $4\,\|V(s) \triangle V(t)\|_1$ units of credit are issued.

If $i > 0$, this credit freshly assigned to an i-letter a is then reassigned to letters in the expansion of a: if $\exp(a)$ is a repetition, 4 units are reassigned to this repetition, if it is a pair, 2 units of credit is given to each of those letters.

In case of $i = 0$, we observe that each remaining symbol is a 0-letter and has 4 units of fresh credit, which can be used to pay for the final operations of delete and insert. ◀

The second step of the analysis is to show that $\|V(s) \triangle V(t)\|_1$ indeed upper bounds the edit distance (multiplied by $\mathcal{O}(\log n)$).

▶ **Lemma 7** (cf. [6]). *Let s, t be two strings and let $\{\mathsf{mv}\} \subseteq \mathsf{Op} \subseteq \{\mathsf{ins}, \mathsf{del}, \mathsf{mv}\}$. Fix an iterated parsing scheme of height k. Then $\|V(t) \triangle V(s)\|_1 = \mathcal{O}(d(k+1))$, where $d = \mathrm{ED}_{\mathsf{Op}}(s, t)$.*

Proof. As $\|\cdot \triangle \cdot\|_1$ satisfies the triangle inequality, it is enough to give the proof for $d = 1$.

Let $s = s_0, \ldots, s_k$ and $t = t_0, \ldots, t_k$ be the consecutive signatures for s, t according to the parsing scheme and $V_0(s), \ldots, V_k(s)$ and $V_0(t), \ldots, V_k(t)$ be the corresponding vectors. We first show by induction on k a stronger claim for move and then adapt it to other operations:

▶ **Claim.** *There are at most 3 substrings in s_k and at most 3 substrings in t_k, called* difference strings*, of total length ℓ_k, such that the multisets of substrings of $\mathrm{sig}_k(s)$ and $\mathrm{sig}_k(t)$ obtained after the removal of the difference strings are equal and $4\ell_k + \|V_k(s) \triangle V_k(t)\|_1 = \mathcal{O}(k+1)$.*

For the base of the induction, if $s = w_1w_2w_3w_4$ is turned to $t = w_1w_3w_2w_4$, let the difference substrings be length-2 substrings on the boundary between each w_i and subsequent w_j. We merge the chosen substrings if they overlap or are adjacent, which results in at most 3 such substrings in s_0 and t_0; their total length is at most $\ell_0 = 12$. Clearly, $(V_0(s) \triangle V_0(t))[a] = 0$ as no letters are removed nor added.

For the induction step, consider how s_k and t_k are parsed. Define the difference strings in s_{k+1} and t_{k+1} as those whose expansions are contained in the difference strings in s_k, t_k or form the $\mathcal{O}(1)$ phrases around the difference substrings that may be parsed differently; see Lemma 4. So the increase ℓ_{k+1} from ℓ_k is upper bounded by $\mathcal{O}(1)$, but it can also decrease if there are phrases in the difference strings that are longer than 1.

Consider the multisets of strings obtained from s_{k+1}, t_{k+1} after the removal of the difference strings. By the choice of the difference strings, their expansions were parsed in the same way (see Lemma 4), and thus those multisets are identical. Consider now $V_{k+1}(s) \triangle V_{k+1}(t)$ and new letters (compared to $V_k(s) \triangle V_k(t)$). Those are $(k+1)$-letters and they are in the difference strings of s_{k+1}, t_{k+1}. Either they are one of the $\mathcal{O}(1)$ letters that replaced phrases that were parsed differently or letters whose phrases consists of the letters in the difference strings for s_k, t_k. But each of the latter letters decreases ℓ_{k+1} when compared to ℓ_k by at least 1: difference strings for s_k, t_k did not include any $(k+1)$-letters and each such a letter corresponds to a phrase of length at least 2.

For del and ins, the difference strings on the 0-th level include the deleted (or inserted) letter and otherwise the proof is only simpler (as there are fewer substrings after the removal of the difference strings and they are in the same order). ◀

▶ **Theorem 8.** AlgMove *gives an $\mathcal{O}(\log n)$ approximation of common partition problem with a set of operations which is any subset of insert, delete. Its running time is linear assuming integer sorting runs in linear time. The same applies to the edit distance with set of operations that include block move and any subset of operations of insert, delete.*

Move and block delete. We now investigate the case in which we allow move as well as block delete operation. This makes the situation asymmetric with respect to s and t. The algorithm is almost the same as AlgMove, though the analysis becomes more involved.

The differences between AlgBdel and AlgMove are as follows: the first is the treatment of the remaining uncoloured letters in s_0 after processing level-0 letters: we delete each maximal string of such letters using block delete. The second is that we make the common partition for repetitions in a more clever way (though it is still a valid one for AlgMove): for a fixed letter a, consider the a-repetitions in S, T (recall that those are the sets of uncoloured a-repetitions in s_i, t_i, respectively); let them have lengths $M = \{m_i\}_{i\in I}$ and $N = \{n_i\}_{i\in J}$. We make the common partition for a-(sub)repetitions in a two-step process. First, we match the a-repetitions of length 1: Consider the 1's that are common in M and N; we colour the corresponding a's in S, T black, remove them from S, T and make their full expansions parts in common partition, and update N, M by removing the common 1's. Note, that now $M \cap N = \emptyset$: if there is a^ℓ in both of them, then this a^ℓ was expanded from the same letter in s_{i+1} and t_{i+1}. But this is not possible, as we colour all such letters black.

For the remaining a-repetitions in S, T, let ℓ_s be the total length of a-repetitions in S (i.e. $\ell_s = \sum_{p\in M} p$), let ℓ_t be the corresponding total length of a-repetitions in T, and let $\ell = \min(\ell_s, \ell_t)$. Choose a-repetitions in S with a total length ℓ, preferring the longer repetitions. We make the common partition between the chosen repetitions in S and the ones in T of length ℓ.

Concerning the analysis, it is clear that $\|V(s) \triangle V(t)\|_1$ cannot be used, as for $t = \epsilon$ it is useless; $\|V(t) \setminus V(s)\|_1$ is a natural candidate, but it is not subtle enough: consider $s = (ab)^\ell$ and $t = a^\ell$. It is clear that at least ℓ operations are needed to transform s to t, yet $\|V(t) \setminus V(s)\|_1 = \mathcal{O}(1)$. The problem is that several short a-repetitions from s are needed to form one long a-repetition in t. On the other hand, identical a-repetitions should be "for free": when $s = t$, then we should not impose any cost.

Motivated by those examples, we define a new cost function for s, t. It is somehow related to Wassersteiner ("earth mover") distance, but it is directed and applies to sets with different sums as well. Let us first define it on multisets of natural numbers: given two such multisets $\{x_i\}_{i\in I}$ and $\{y_i\}_{i\in J}$, we first exclude from those sets their common part and look for the smallest number of elements in $\{x_i\}_{i\in I}$ whose sum is at least the sum of $\{y_i\}_{i\in J}$; if $\{x_i\}_{i\in I}$ is not enough, we pad it with an arbitrary number of 1's. Formally, let $I' \subseteq I$ and $J' \subseteq J$ be such that $\{x_i\}_{i\in I'} = \{x_i\}_{i\in I} \setminus (\{x_i\}_{i\in I} \cap \{y_i\}_{i\in J})$ and $\{y_i\}_{i\in J'} = \{y_i\}_{i\in J} \setminus (\{x_i\}_{i\in I} \cap \{y_i\}_{i\in J})$, define $x = \sum_{i\in I'} x_i$, $y = \sum_{i\in J'} y_i$. Then the $\mathrm{SD}(\{x_i\}_{i\in I'}, \{y_i\}_{i\in J'})$ is defined as follows: if $x < y$, then it is $(y - x) + |I'|$; otherwise, it is the smallest m such that the sum of the largest m elements in $\{x_i\}_{i\in I'}$ is at least y. Lastly, we set $\mathrm{SD}(\{x_i\}_{i\in I}, \{y_i\}_{i\in J})$ as $\mathrm{SD}(\{x_i\}_{i\in I'}, \{y_i\}_{i\in J'})$.

▶ **Lemma 9.** SD *satisfies the directed triangle inequality.*

Then for the input strings s, t and a j-letter a, we define $\mathrm{SD}(s, t)[a]$ as $\mathrm{SD}(\{x_i\}_{i\in I}, \{y_i\}_{i\in J})$, where $\{x_i\}_{i\in I}, \{y_i\}_{i\in J}$ are the multisets of lengths of a-repetitions in s_j and t_j, respectively. Note that, unlike embedding to vectors, $\mathrm{SD}(s, t)$ cannot be computed for s and t separately; it is defined for a pair s, t.

The following two lemmata are the counterparts of Lemma 6 and Lemma 7 in case when block delete is allowed; their proofs are similar.

▶ **Lemma 10.** *Let* $\{\mathsf{b\text{-}del}, \mathsf{mv}\} \subseteq \mathsf{Op} \subseteq \{\mathsf{b\text{-}del}, \mathsf{mv}, \mathsf{del}, \mathsf{ins}, \mathsf{uncp}\}$. *Given an iterated parsing string for strings s and t,* AlgBdel *construct a partition of size $\mathcal{O}(\|V(t) \setminus V(s)\|_1 + \|\mathrm{SD}(s, t)\|_1)$. Moreover, it runs in linear time.*

▶ **Lemma 11.** *Let s, t be two strings and let $\{\mathsf{b\text{-}del}, \mathsf{mv}\} \subseteq \mathsf{Op} \subseteq \{\mathsf{mv}, \mathsf{b\text{-}del}, \mathsf{ins}, \mathsf{del}, \mathsf{uncp}\}$. Fix an iterated parsing scheme of height k. Then $\|V(t) \setminus V(s)\|_1$, $\|\mathrm{SD}(s,t)\|_1 \in \mathcal{O}(d(k+1))$, where $d = \mathrm{ED}_{\mathsf{Op}}(s,t)$.*

▶ **Theorem 12.** AlgBdel *is an $\mathcal{O}(\log n)$ approximation of edit distance for a set of operations that include block move, block delete, and any subset of block uncopy, insert, delete. Its running time is linear assuming integer sorting running time is linear.*

Move and copy. We now give an algorithm that deals with the scenario in which both block move and block copy are allowed. As a simple example, consider $s = a^n$ and $t = a^m$, where $m \geq n$; the easiest way to obtain t is to repeatedly "square" the string $\lceil \log(m/n) \rceil$ times.

Thus, if copy is allowed, we need also to take into the account the lengths of maximal repetitions. To model this in the analysis,[1] given an iterated parsing scheme, we define a vector $\mathrm{LMax}(s)$, indexed by letters of $\bigcup_i \Sigma_i$, so that $\mathrm{LMax}(s)[a]$ for an i-letter a is the logarithm of the longest a-repetition in s_i; we set $\mathrm{LMax}(s)[a] = 0$ if there is no such repetition.

As a second part of the intuition, we note that having copied a symbol, after some expansions we may realise that it would be better to perform moves instead. Imagine that an i-letter a occurs twice in t_i and we declare one occurrence to be a copy of the other. Later on, a is expanded to bc, and it turns out that in s_{i-1} there are two uncoloured copies of b and c. In this case, it is better to cancel the copying and move two b's and two c's into t_{i-1}.

AlgBcp proceeds similarly as AlgMove: for $i = k, k-1, \ldots, 0$, we consider s_i, t_i. We construct move and copy operations: the move operations are performed in the order in which AlgBcp constructs them, their sources are always in s_i and targets in t_i; we copy only within t_i and those operations are performed in the reverse order (compared to how the algorithm constructs them) after all the other operations. This should be intuitively clear: when in t_i we declare one occurrence of a substring t' to be a copy of another occurrence, then it may be that we still do not know how t' is constructed from the substrings of s.

The target of a copy operation is coloured grey and this colour is preserved by expansions. However, we may always change our mind and uncolour any grey substring. To simulate this, we split the target into (at most 3) shorter blocks, replacing the original copy operation by more such operations, and we cancel one of them. In fact, we uncolour only to make room for the target of a move operation, so the uncoloured symbols are immediately coloured black.

Let the multisets of uncoloured letters in s_i, t_i be S, T, respectively. For each i-letter a, we list all a-repetitions in S, T: let ℓ_s and ℓ_t be the total length of a-repetitions in S and T.

- If $\ell_s \geq \ell_t$, then we use Lemma 5 to make a common partition of repetitions from S of total length ℓ_t and all repetition in T, colour those letters black, remove them from S, T, and move their full expansions from s_0 to t_0. If there are repetitions of a left in s_i, then we look whether there are any grey a-repetitions in t_i and we proceed as in Lemma 10, but in the other direction: we first match single a-repetitions in S and single a-repetitions coloured grey (so in T). After this operation, it cannot be that a repetition a^k has an uncoloured occurrence in s_i and a grey one in t_i, as this would mean that the $(i+1)$-letter representing a^k had such occurrences in s_{i+1} and t_{i+1}, which is not possible. Then we take the longest grey a-repetitions, enough to make the common partition with the repetitions in S, or all grey a-repetitions, if there are not enough of them. We make a common partition for those repetitions, recolouring the matched grey letters black and move the corresponding full expansions from s to t.

[1] This can be also solved by ensuring that the symbol that replaced a^k is not grouped in the next $\log k$ phases of the iterated parsing scheme [15]; this moves the burden from the analysis to the algorithm.

- If $\ell_s < \ell_t$, then we choose repetitions in T of total length ℓ_s, including the longest repetition of T or, if it is longer than ℓ_s, including its prefix of length ℓ_s. We colour the corresponding letters black and then move their full expansions from s to t. Next, we make sure that the longest repetition in t_i is fully coloured (except the first letter if $\ell_s = 0$). For this, we iteratively copy its longest coloured prefix to its following part, colouring the latter grey, which doubles the length of the coloured prefix of this repetition; if the longest repetition is fully uncoloured (i.e., if $\ell_s = 0$), then we begin with copying its first letter to the second. Finally, if there is any other uncoloured (sub)repetition left in T, then we colour it grey and mark it as a copy of the prefix of the longest repetition.

After processing all 0-letters, we perform the final operations as in AlgMove: when insertion is allowed, we insert all remaining letters in t (or reject, when insertion is not allowed) and delete all remaining letters in s, (or reject, when deletion is not allowed).

As before, the analysis has two steps: on one hand we estimate the cost in terms of various functions based on $V(s), V(t)$ (see Lemma 13) and on the other we show that those functions are bounded by $\mathcal{O}(d(k+1))$, where k is the height of the parsing scheme for s, t. When the appropriate functions are known, the proofs follow similarly as in Lemma 6 and 7 (recall that for a vector v the $\sup(v)$ changes each v's non-zero component to 1).

▶ **Lemma 13.** *Let* $\{\mathsf{cp}, \mathsf{mv}\} \subseteq \mathsf{Op} \subseteq \{\mathsf{cp}, \mathsf{mv}, \mathsf{ins}, \mathsf{del}\}$. *Given an iterated parsing scheme for strings* s *and* t, AlgBcp *returns a sequence of* $\mathcal{O}(\|V(s) \setminus V(t)\|_1 + \|\sup(V(t)) \setminus \sup(V(s))\|_1 + \|\mathrm{LMax}(t) \setminus \mathrm{LMax}(s)\|_1 + \|\mathrm{SD}(t,s)\|_1)$ *operations from* Op *that transform* s *to* t. *Moreover, it runs in linear time.*

▶ **Lemma 14.** *Let* s, t *be two strings and let* $\{\mathsf{cp}, \mathsf{mv}\} \subseteq \mathsf{Op} \subseteq \{\mathsf{cp}, \mathsf{mv}, \mathsf{ins}, \mathsf{del}\}$ *with a fixed iterated parsing scheme of height* k. *Then* $\|V(s) \setminus V(t)\|_1$, $\|\sup(V(t)) \setminus \sup(V(s))\|_1$, $\|\mathrm{LMax}(s) \triangle \mathrm{LMax}(t)\|_1$ *and* $\|\mathrm{SD}(t,s)\|_1$ *are in* $\mathcal{O}(d(k+1))$, *where* $d = \mathrm{ED}_{\mathsf{Op}}(s,t)$.

▶ **Theorem 15.** AlgBcp *gives an* $\mathcal{O}(\log n)$ *approximation of the edit distance for operations that include block copy and move and any operations from: insert, delete. Its running time is linear assuming integer sorting runs in linear time.*

Copy and uncopy. We now investigate the case in which both copy and uncopy operations are allowed. Although the move can be simulated by them, we still use the move operation as it makes the description of the algorithm and the analysis more similar to those of previous algorithms. As previously, AlgBcpuncp can deal also with letter insertions and deletions.

AlgBcpuncp, as AlgBcp, colours the letters grey or black to represent that they are already dealt with. Initially all letters are uncoloured. When we expand a letter, its expansion gets coloured if and only if the letter was coloured. While we construct the sequences of all operations in parallel, we in fact perform first all uncopy operations, then all moves and lastly all copy operations. Uncopying is always done witin s_i, those operations are performed in order of their construction. In such a case, we colour the uncopied grey letters. We move elements from s_i to t_i and those operations are performed in the order in which the algorithm constructs them; we colour both the source and target letters of this operation black. We copy only within t_i and those operations are performed in the reverse order (compared to how the algorithms constructs them). Targets of the copy (uncopy) operation are coloured grey. Concerning other operations, insertion and deletions are done all at once, after all uncopy and move operations but before copy operations.

We compute the iterated parsing scheme and process the strings s_i, t_i in phases for $i = k, k-1, \ldots, 0$. In the ith phase, we consider each i-letter a and introduce some move,

copy, and uncopy operations to make sure that if a occurs in both s_i and t_i, then all the occurrences are coloured; otherwise, exactly one occurrence shall be uncoloured. Let the lengths of the longest a-repetition in s_i and t_i be ℓ_s and ℓ_t, respectively (these values can be equal to 0 if a does not occur in s_i or t_i). Fix some occurrences of those longest a-repetitions, preferring black, then uncoloured, and then grey. We uncopy each uncoloured a-repetition in s_i (except the chosen one) from the chosen one and, symmetrically, copy each uncoloured a-repetition in t_i (except the chosen one) from the chosen one; the targets of those operations are coloured grey. Now the actions depends on whether those chosen repetitions are coloured. To streamline the argument for $\min(\ell_s, \ell_t) = 0$, we assume that an empty repetition is black.

- If they are both coloured, then we do nothing: all a-repetitions in both s_i, t_i are coloured.
- If they are both uncoloured, then we move $\min(\ell_s, \ell_t)$ letters a from s_i to t_i. If $\ell_s > \ell_t$, then using $\lceil \log(\ell_s/\ell_t) \rceil$ uncopy operations we colour the rest of the chosen repetition in s_i; if $\ell_t > \ell_s$, then, symmetrically, the rest of this longest a-repetition in t_i is coloured grey using $\lceil \log(\ell_t/\ell_s) \rceil$ copy operations.
- If the one in s_i is coloured and the one in t_i is not, then the one in s_i must be black by the choice of the longest repetition (grey last): it is impossible that all repetitions a^{ℓ_s} in s_i are grey. Furthermore, it can be shown that there is a black repetition a^{ℓ_s} in t_i. Since we chose uncoloured repetition in t_i, by the choice strategy (black first) it holds that $\ell_t > \ell_s$. If $\ell_s > 0$, we copy the chosen uncoloured repetition a^{ℓ_t} from the black repetition a^{ℓ_s} in t, using $1 + \lceil \log(\ell_t/\ell_s) \rceil$ copy operations. Otherwise, we leave the first character of a^{ℓ_t} uncoloured and colour the remaining letters grey using $\lceil \log \ell_t \rceil$ copy operations.
- If the one in t_i is coloured and the one in s_i is not, the algorithm is symmetric to the previous case.

▶ **Lemma 16** (cf. [15, 16]). *Let* $\{\mathsf{cp}, \mathsf{uncp}\} \subseteq \mathsf{Op} \subseteq \{\mathsf{cp}, \mathsf{uncp}, \mathsf{mv}, \mathsf{ins}, \mathsf{del}\}$. *Given an iterated parsing scheme for strings* s *and* t*, one can construct in linear time a sequence of* $\mathcal{O}(\|\mathrm{LMax}(s) \triangle \mathrm{LMax}(t)\|_1 + \|\sup(V(s)) \triangle \sup(V(t))\|_1)$ *operations from* Op *that transform* s *to* t*.*

The bound of $\mathcal{O}(d(k+1))$ on $\|\sup(V(t)) \triangle \sup(V(s))\|_1$ and $\|\mathrm{LMax}(t) \triangle \mathrm{LMax}(s)\|_1$ follows already from Lemmata 7, 11, and 14.

▶ **Theorem 17.** AlgBcpuncp *is an* $\mathcal{O}(\log n)$ *approximation of the edit distance with set of operations that include block copy and uncopy and any subset of insert, delete, block move. Its running time is linear assuming integer sorting runs in linear time.*

5 Compressed Input

A *Straight-Line Programme* (SLP) is a context-free grammar that produces exactly one string and is treated as a compressed representation of this string. Its size is the sum of lengths of the right-hand sides of the productions.

The presented algorithms can be also implemented, when the input (i.e. strings s, t) are given as SLPs. In such a case, the running time increases to $\mathcal{O}(n \log N)$ and the approximation ratio is $\mathcal{O}(\log N)$, where n is the size of the SLPs representing s, t in the input and $N = \max(|s|, |t|)$ is the maximum of the lengths of strings defined by those SLPs.

The algorithms require only an implementation of the iterated parsing scheme for strings given as SLPs, which is known; see for instance [9, 12]. This is no surprise, as such techniques were introduced and are developed mostly in the context of grammar-compressed data.

References

1 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In David B. Shmoys, editor, *11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000*, pages 819–828. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338645.

2 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 51–58. ACM, 2015. doi:10.1145/2746539.2746612.

3 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In Chandra Chekuri, editor, *25th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, pages 102–121. SIAM, 2014. doi:10.1137/1.9781611973402.8.

4 Marek Chrobak, Petr Kolman, and Jirí Sgall. The greedy algorithm for the minimum common string partition problem. *ACM Transactions on Algorithms*, 1(2):350–366, 2005. doi:10.1145/1103963.1103971.

5 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

6 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1):2:1–2:19, 2007. doi:10.1145/1219944.1219947.

7 Graham Cormode, Mike Paterson, Süleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In David B. Shmoys, editor, *11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000*, pages 197–206. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338252.

8 Funda Ergün, S. Muthukrishnan, and Süleyman Cenk Sahinalp. Comparing sequences with segment rearrangements. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2003*, volume 2914 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 2003. doi:10.1007/978-3-540-24597-1_16.

9 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In Artur Czumaj, editor, *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.

10 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electronic Journal of Combinatorics*, 12, 2005. URL: http://www.combinatorics.org/Volume_12/Abstracts/v12i1r50.html.

11 Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027.

12 Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

13 Daniel P. Lopresti and Andrew Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997. doi:10.1016/S0304-3975(96)00268-X.

14 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

15 S. Muthukrishnan and Süleyman Cenk Sahinalp. Approximate nearest neighbors and sequence comparison with block operations. In F. Frances Yao and Eugene M. Luks, editors,

32nd Annual ACM Symposium on Theory of Computing, STOC 2000, pages 416–424. ACM, 2000. doi:10.1145/335305.335353.

16 S. Muthukrishnan and Süleyman Cenk Sahinalp. Simple and practical sequence nearest neighbors with block operations. In Alberto Apostolico and Masayuki Takeda, editors, *Combinatorial Pattern Matching, CPM 2002*, volume 2373 of *LNCS*, pages 262–278. Springer, 2002. doi:10.1007/3-540-45452-7_22.

17 Süleyman Cenk Sahinalp and Uzi Vishkin. On a parallel-algorithms method for string matching problems. In Maurizio A. Bonuccelli, Pierluigi Crescenzi, and Rossella Petreschi, editors, *Algorithms and Complexity, CIAC 1994*, volume 778 of *LNCS*, pages 22–32. Springer, 1994. doi:10.1007/3-540-57811-0_3.

18 Süleyman Cenk Sahinalp and Uzi Vishkin. Symmetry breaking for suffix tree construction. In Frank Thomson Leighton and Michael T. Goodrich, editors, *26th Annual ACM Symposium on Theory of Computing, STOC 1994*, pages 300–309. ACM, 1994. doi:10.1145/195058.195164.

19 Dana Shapira and James A. Storer. Edit distance with move operations. *Journal of Discrete Algorithms*, 5(2):380–392, 2007. doi:10.1016/j.jda.2005.01.010.

20 Dana Shapira and James A. Storer. Edit distance with block deletions. *Algorithms*, 4(1):40–60, 2011. doi:10.3390/a4010040.

21 Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985. doi:10.1016/S0019-9958(85)80046-2.