Block Edit Errors with Transpositions: Deterministic Document Exchange Protocols and Almost Optimal Binary Codes

Kuan Cheng
Department of Computer Science, Johns Hopkins University, USA
kcheng17@jhu.edu

Zhengzhong Jin
Department of Computer Science, Johns Hopkins University, USA
zjin12@jhu.edu

Xin Li
Department of Computer Science, Johns Hopkins University, USA
lixints@cs.jhu.edu

Ke Wu
Department of Computer Science, Johns Hopkins University, USA
ashleymo@jhu.edu

Abstract

Document exchange and error correcting codes are two fundamental problems regarding communications. In the first problem, Alice and Bob each holds a string, and the goal is for Alice to send a short sketch to Bob, so that Bob can recover Alice's string. In the second problem, Alice sends a message with some redundant information to Bob through a channel that can add adversarial errors, and the goal is for Bob to correctly recover the message despite the errors. In both problems, an upper bound is placed on the number of errors between the two strings or that the channel can add, and a major goal is to minimize the size of the sketch or the redundant information. In this paper we focus on deterministic document exchange protocols and binary error correcting codes.

Both problems have been studied extensively. In the case of Hamming errors (i.e., bit substitutions) and bit erasures, we have explicit constructions with asymptotically optimal parameters. However, other error types are still rather poorly understood. In a recent work [7], the authors constructed explicit deterministic document exchange protocols and binary error correcting codes for edit errors with almost optimal parameters. Unfortunately, the constructions in [7] do not work for other common errors such as block transpositions.

In this paper, we generalize the constructions in [7] to handle a much larger class of errors. These include bursts of insertions and deletions, as well as block transpositions. Specifically, we consider document exchange and error correcting codes where the total number of block insertions, block deletions, and block transpositions is at most $k \leq \alpha n/\log n$ for some constant $0 < \alpha < 1$. In addition, the total number of bits inserted and deleted by the first two kinds of operations is at most $t \leq \beta n$ for some constant $0 < \beta < 1$, where n is the length of Alice's string or message. We construct explicit, deterministic document exchange protocols with sketch size $O((k \log n + t) \log^2 \frac{n}{k \log n + t})$ and explicit binary error correcting code with $O(k \log n \log\log\log n + t)$ redundant bits. As a comparison, the information-theoretic optimum for both problems is $\Theta(k \log n + t)$. As far as we know, previously there are no known explicit deterministic document exchange protocols in this case, and the best known binary code needs $\Omega(n)$ redundant bits even to correct just *one* block transposition [23].[1]

2012 ACM Subject Classification Mathematics of computing → Coding theory

[1] We note that by combining the techniques in [14] and [15], one can get an explicit binary code that corrects k block transpositions with $\widetilde{O}(\sqrt{kn})$ redundant bits. However to our knowledge this result has not appeared anywhere in the literature, and moreover it requires at least $\widetilde{\Omega}(\sqrt{n})$ redundant bits even to correct one block transposition.

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 37; pp. 37:1–37:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Deterministic document exchange, error correcting code, block edit error

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.37

Category Track A: Algorithms, Complexity and Games

Related Version A full version of the paper is available at `https://arxiv.org/abs/1809.00725`.

Acknowledgements We thank an anonymous referee for catching an error in the previous version of this paper, and Bernhard Haeupler for very useful feedbacks.

1 Introduction

In communications and more generally distributed computing environments, there often arise questions regarding the synchronization of files or messages. For example, a message sent from one party to another party through a channel may get modified by channel noise or adversarial errors, and files stored on distributed servers may become out of sync due to different edit operations by different users. In many situations, these questions can be formalized in the framework of the following two fundamental problems.

- *Document exchange.* In this problem, two parties Alice and Bob each holds a string x and y, and the two strings are within distance k in some metric space. The goal is for Alice to send a short sketch to Bob, so that Bob can recover x based on his string y and the sketch.
- *Error correcting codes.* In this problem, two parties Alice and Bob are linked by a channel, which can change any string sent into another string within distance k in some metric space. Alice's goal is to send a message to Bob. She does this by sending an encoding of the message through the channel, which contains some redundant information, so that Bob can recover the correct message despite any changes to the codeword.

These two problems are closely related. For example, in many cases a solution to the document exchange problem can also be used to construct an error correcting code, but the reverse direction is not necessarily true. In both problems, a major goal is to is to minimize the size of the sketch or the redundant information. For applications in computer science, we also require the computations of both parties to be efficient, i.e., in polynomial time of the input length. In this case we say that the solutions to these problems are *explicit*. Here we focus on deterministic document exchange protocols and error correcting codes with a binary alphabet, arguably the most important setting in computer science.

Both problems have been studied extensively, but the known solutions and our knowledge vary significantly depending on the distance metric in these problems. In the case of Hamming distance (or Hamming errors), we have a near complete understanding and explicit constructions with asymptotically optimal parameters. However, for other distance metrics/error types, our understanding is still rather limited.

An important generalization of Hamming errors is edit errors, which consist of bit insertions and deletions. These are strictly more general than Hamming errors since a bit substitution can be replaced by a deletion followed by an insertion. Edit errors can happen in many practical situations, such as reading magnetic and optical media, mutations in gene sequences, and routing packets in Internet protocols. However, these errors are considerably harder to handle, due to the fact that a single edit error can change the positions of all the bits in a string.

Non-explicitly, by using a greedy graph coloring algorithm or a sphere packing argument, one can show that the optimal size of the sketch in document exchange, or the redundant information in error correcting codes is roughly the same for both Hamming errors and edit errors. Specifically, suppose that Alice's string or message has length n and the distance bound k is relatively small (e.g., $k \leq n/4$), then for both Hamming errors and edit errors, the optimal size in both problems is $\Theta(k \log(\frac{n}{k}))$ [19]. For Hamming errors, this can be achieved by using sophisticated linear Algebraic Geometric codes [16], but for edit errors the situation is quite different. We now describe some of the previous works regarding both document exchange and error correcting codes for edit errors.

Document exchange

Orlitsky [21] first studied the document exchange problem for generally correlated strings x, y. Using the greedy graph coloring algorithm mentioned before, he obtained a deterministic protocol with sketch size $O(k \log n)$ for edit errors, but the running time is exponential in k. Subsequent improvements appeared in [9], [17], and [18], achieving sketch size $O(k \log(\frac{n}{k}) \log n)$ [17] and $O(k \log^2 n \log^* n)$ [18] with running time $\tilde{O}(n)$. A recent work by Chakraborty et al. [5] further obtained sketch size $O(k^2 \log n)$ and running time $\tilde{O}(n)$, by using a clever randomized embedding from the edit distance metric to the Hamming distance metric. Based on this work, Belazzougui and Zhang [2] gave an improved protocol with sketch size $O(k(\log^2 k + \log n))$, which is asymptotically optimal for $k = 2^{O(\sqrt{\log n})}$. The running time in [2] is $\tilde{O}(n + \mathsf{poly}(k))$.

Unfortunately, all of the above protocols, except the one in [21] which runs in exponential time, are randomized. Although randomized protocols are still useful in practice, having deterministic ones would certainly bring much more benefits. Furthermore, randomized protocols are also not suitable for the applications in constructing error correcting codes. However, designing an efficient deterministic protocol appears quite tricky, and it was not until 2015 when Belazzougui [1] gave the first deterministic protocol even for $k > 1$. The protocol in [1] has sketch size $O(k^2 + k \log^2 n)$ and running time $\tilde{O}(n)$.

Error correcting codes

As fundamental objects in both theory and practice, error correcting codes have been studied extensively from the pioneering work of Shannon and Hamming. While great success has been achieved in constructing codes for Hamming errors, the progress on codes for edit errors has been quite slow despite much research. A work by Levenshtein [19] in 1966 showed that the Varshamov-Tenengolts code [22] corrects one deletion with an optimal redundancy of roughly $\log n$ bits, but even correcting two deletions requires $\Omega(n)$ redundant bits. In 1999, Schulman and Zuckerman [23] gave an explicit asymptotically good code, that can correct up to $\Omega(n)$ edit errors with $O(n)$ redundant bits. However the same amount of redundancy is needed even for smaller number of errors. For more earlier works on this subject, we refer the reader to the survey by Mercier et al. [20].

In recent years there have been several works trying to improve the situation. Specifically, a line of work by Guruswami et. al [11], [10], [4] constructed explicit codes that can correct $1 - \varepsilon$ fraction of edit errors with rate $\Omega(\varepsilon^5)$ and alphabet size $\mathsf{poly}(1/\varepsilon)$; and codes that can correct $1 - \frac{2}{t+1} - \varepsilon$ fraction of errors with rate $(\varepsilon/t)^{\mathsf{poly}(1/\varepsilon)}$ for a fixed alphabet size $t \geq 2$. Another line of work by Haeupler et al. [13], [14], [6] introduced and constructed a combinatorial object called *synchronization string*, which can be used to transform standard error correcting codes into codes for edit errors by increasing the alphabet size. Via this

transformation, [13] achieved explicit codes that can correct δ fraction of edit errors with rate $1-\delta-\varepsilon$ and alphabet size exponential in $\frac{1}{\varepsilon}$, which approaches the singleton bound. All of these works however require a relatively large alphabet size.

In the case of binary alphabets, for any fixed constant k, a recent work by Brakensiek et. al [3] constructed an explicit code that can correct k edit errors with $O(k^2 \log k \log n)$ redundant bits. This is asymptotically optimal when k is a fixed constant, but the construction in [3] only works for constant k, and breaks down for larger k (e.g., $k = \log n$). Based on his deterministic document exchange protocol, Belazzougui [1] also gave an explicit code that can correct up to k edit errors with $O(k^2 + k \log^2 n)$ redundant bits. Finally, the work by Haeupler et. al [15] constructed explicit codes that can correct δ fraction of edit errors with rate $1 - \Theta(\sqrt{\delta \log(1/\delta)})$, whereas the (non-explicit) optimal rate is $1 - \Theta(\delta \log(1/\delta))$.

In a very recent work by the authors [7], we significantly improved the situation. Specifically, we constructed an explicit document exchange protocol with sketch size $O(k \log^2 \frac{n}{k})$, which is optimal except for an additional $\log \frac{n}{k}$ factor. This also implies an explicit binary code that can correct δ fraction of edit errors with rate $1 - \Theta(\delta \log^2(1/\delta))$, which is optimal up to an additional $\log(1/\delta)$ factor. These two results are also independently obtained by Haeupler [12]. We also constructed explicit codes for k edit errors with $O(k \log n)$ redundant bits, which is optimal for $k \leq n^{1-\alpha}$, any constant $0 < \alpha < 1$. These results bring our understanding of document exchange and error correcting codes for edit errors much closer to that of standard Hamming errors.

However, the constructions in [7] and [12] do not work for other common types of errors, such as *block transpositions*. Given any string x, a block transposition takes an arbitrary substring z of x, cuts it to make x become $\tilde{x}$, and then finds a different position in $\tilde{x}$ and insert z as a block into $\tilde{x}$. These errors happen frequently in distributed file systems and Internet protocols. For example, it is quite common that a user, when editing a file, moves a whole paragraph in the file to somewhere else; and in Internet routing protocols, packets can often get rearranged during the process. Block transpositions also arise naturally in biological processes, where a subsequence of genes can be moved in one step during mutation. In the setting of document exchange or error correcting codes, it is easy to see that even a single transposition of a block with length t can result in $2t$ edit errors, thus a naive application of document exchange protocols or codes for edit errors will result in very bad parameters.

Model of the adversary: block insertions, deletions, and transpositions

We consider edit errors that happen in *bursts*. This kind of errors is also pretty common, as most errors that happen in practice, such as in wireless or mobile communications and magnetic disk readings, tend to be concentrated. We model such errors as *block* insertions and deletions, where in one operation the adversary can insert or delete a whole block of bits. It is again easy to see that this is indeed a generalization of standard edit errors.

For some parameters k and t and an alphabet Σ, a (k, t) block edit adversary is allowed to perform three kinds of operations: block insertion, block deletion and block transposition. The adversary is allowed to perform at most k such operations, while the total number of symbols inserted/deleted by the first two operations is at most t. We also use (k, t) block edit errors to denote errors introduced by such an adversary. All our results focus on the case of binary alphabet, but in our protocols and analysis we will be using larger alphabets.

We note that by the result of Schulman and Zuckerman [23], to correct $\Omega(n/\log n)$ block transpositions one needs at least $\Omega(n)$ redundant bits. Thus we only consider $k \leq \alpha n/\log n$ for some constant $0 < \alpha < 1$. Similarly, we only consider $t \leq \beta n$ for some constant $0 < \beta < 1$ since otherwise the adversary can simply delete the whole string. We also note the following

subtle difference between the three block edit operations. While we need a bound t on the total number of bits that the adversary can insert or delete, for block transposition an adversary can choose to move an *arbitrarily long* substring. Therefore, we need to consider the three operations separately, and cannot simply replace a block transposition by a block deletion followed by a block insertion.

Edit errors with block transpositions have been studied before in several different contexts. For example, Shapira and Storer [24] showed that finding the distance between two given strings under this metric is NP-hard, and they gave an efficient algorithm that achieves $O(\log n)$ approximation. Interestingly, a work by Cormode and Muthukrishnan [8] showed that this metric can be embedded into the L_1 metric with distortion $O(\log n \log^* n)$; and they used it to give a near linear time algorithm that achieves $O(\log n \log^* n)$ approximation for this distance, something currently unknown for the standard edit distance. Coming back to document exchange and error correcting codes, in our model, we show in the appendix that non-explicitly, the information optimum for both the sketch size of document exchange, and the redundancy of error correcting codes, is $\Theta(k \log n + t)$.

Related previous work on block transpositions

When it comes to more general errors such as block transpositions, as far as we know, there are no known explicit deterministic document exchange protocols. The only known randomized protocols which can handle edit errors as well as block transpositions are the protocol of [17], which has sketch size $O(k \log(\frac{n}{k}) \log n)$; and the protocol of [18], which has sketch size $\tilde{O}(k \log^2 n)$. The protocol of [17] uses a recursive tree structure and random hash functions, while the protocol of [18] is based on the embedding of Cormode and Muthukrishnan [8]. We stress that both of these protocols are randomized, and there are very good reasons why it is not easy to modify them into deterministic ones. Specifically, unlike in our previous work [7] and the work of Haeupler [12], a direct derandomization of the hash functions used in [17] (for example by using almost k-wise independent sample space) does *not* give a deterministic protocol, because block transpositions will make the computation of a matching problematic. We shall discuss this in more details when we give an overview of our techniques. On the other hand, the embedding of Cormode and Muthukrishnan [8] results in an exponentially large dimension, thus directly sending a sketch deterministically will result in a prohibitively large size. This is why the protocol of [18] has to perform a dimension reduction first, which is necessarily randomized.

Similarly, the only previous explicit codes that can handle edit errors as well as block transpositions are the work of Schulman and Zuckerman [23], and the work of Haeupler et al. [14]. Both can recover from $\Omega(n/\log n)$ block transpositions with $\Omega(n)$ redundant bits ([14] can also recover from block replications), but [23] has a binary alphabet while [14] has a constant size alphabet. However the work of Schulman and Zuckerman [23] also needs $\Omega(n)$ redundant bits even to correct one block transposition. We further note that by combining the techniques in [14] and [15], one can get an explicit binary code that corrects k block transpositions with $\widetilde{O}(\sqrt{kn})$ redundant bits. However to our knowledge this result has not appeared anywhere in the literature, and moreover it requires at least $\widetilde{\Omega}(\sqrt{n})$ redundant bits even to correct one block transposition. We note that however none of the previous works mentioned studied edit errors that can allow block insertions/deletions.

1.1 Our results

In this paper we construct explicit deterministic document exchange protocols, and error correcting codes for adversaries discussed above. We have the following theorems.

▶ **Theorem 1.** *There exist constants $\alpha, \beta \in (0,1)$ such that for every $n, k, t \in \mathbb{N}$ with $k \leq \alpha n/\log n, t \leq \beta n$, there exists an explicit binary document exchange protocol with sketch size $O((k\log n + t)\log^2 \frac{n}{k\log n+t})$, against a (k,t) block edit adversary.*

This is the first explicit, deterministic document exchange protocol for block edit errors. The sketch size matches the randomized protocols of [17] and [18] up to an additional $\log \frac{n}{k\log n+t}$ factor, and is optimal up to an additional $\log^2 \frac{n}{k\log n+t}$ factor. Using this protocol, we can construct the following error correcting code.

▶ **Theorem 2.** *There exist constants $\alpha, \beta \in (0,1)$ such that for every $n, k, t \in \mathbb{N}$ with $k \leq \alpha n/\log n, t \leq \beta n$, there exists an explicit binary error correcting code with message length n and codeword length $n + O((k\log n + t)\log^2 \frac{n}{k\log n+t})$, against a (k,t) block edit adversary.*

For small k, t we can actually achieve the following result, which gives better parameters.

▶ **Theorem 3.** *There exist constants $\alpha, \beta \in (0,1)$ such that for every $n, k, t \in \mathbb{N}$ with $k \leq \alpha n/\log n, t \leq \beta n$, there exists an explicit binary code with message length n and codeword length $n + O(k\log n \log\log\log n + t)$, against a (k,t) block edit adversary.*

In the case of small k, t, these results significantly improve the result of Schulman and Zuckerman [23], which needs $\Omega(n)$ redundant bits even to correct one block transposition, and the result obtained by combining the techniques in [14] and [15], which needs $\widetilde{\Omega}(\sqrt{n})$ redundant bits even to correct one block transposition. The redundancy here is also optimal up to an extra $\log\log\log n$ factor or $\log^2 \frac{n}{k\log n+t}$ factor.

As a special case, we obtain the following corollaries for standard edit errors with block transpositions.

▶ **Corollary 4.** *There exist a constant $\alpha \in (0,1)$ such that for every $n, k \in \mathbb{N}$ with $k \leq \alpha n/\log n$, there exists an explicit binary document exchange protocol with sketch size $O(k\log n \log^2 \frac{n}{k\log n})$, against an adversary who can perform k edit operations or block transpositions.*

▶ **Corollary 5.** *There exist a constant $\alpha \in (0,1)$ such that for every $n, k \in \mathbb{N}$ with $k \leq \alpha n/\log n$, there exists an explicit binary error correcting code with message length n and codeword length $\min\{n + O(k\log n \log^2 \frac{n}{k\log n}), n + O(k\log n \log\log\log n)\}$, against an adversary who can perform k edit operations or block transpositions.*

▶ Remark 6. As illustrated by our theorems and corollaries, the sketch size in our document exchange protocol or the number of redundant bits in our error correcting codes do *not* depend on the size of a block in block transpositions, they only depend on the number of such operations performed. In contrast, the sketch size or the number of redundant bits do depend on the size of a block in block insertions or deletions. This again shows that we cannot simply treat a block transposition as a block deletion followed by a block insertion, because that will lead to a sketch size dependent on the block size.

2 Document Exchange

This section describes the construction of our document exchange protocol and its proof in sketch. Details are deferred to the appendix.

▶ **Definition 7** (Collision free hash functions). *Given $n, p, q \in \mathbb{N}, p \leq n$ and a string $x \in \{0,1\}^n$, we say a hash function $h : \{0,1\}^p \to \{0,1\}^q$ is collision free (for x), if for every $i, j \in [n-p+1]$, $h(x[i, i+p)) = h(x[j, j+p))$ if and only if $x[i, i+p) = x[j, j+p)$.*

▶ **Theorem 8.** *There exists an algorithm which, on input $n, p, q \in \mathbb{N}, p \leq n, q = c_0 \log n$ for large enough constant c_0, $x \in \{0,1\}^n$, outputs a description of a hash function $h : \{0,1\}^p \to \{0,1\}^q$ that is collision free for x, in time $\mathsf{poly}(n)$, where the description length is $O(\log n)$.*

Also there is an algorithm which, given the description of h and any $u \in \{0,1\}^p$, can output $h(u)$ in time $\mathsf{poly}(n)$.

Proof Sketch. The construction uses almost κ-wise independence generator to get the hash functions, assuring no pair of blocks of x collides. The number of pairs is $O(n^2)$. The seed length is $O(\log n)$. So we can do an exhaustive search to find such sequence of collision free hash functions. For details, see the full version. ◀

▶ **Definition 9** (Matching). *Given $n, n', p, q \in \mathbb{N}, p \leq n, p \leq n'$, a function $h : \{0,1\}^p \to \{0,1\}^q$ and two strings $x \in \{0,1\}^n, y \in \{0,1\}^{n'}$, a matching (may not be monotone) between x and y under h is a sequence of matches (pairs of indices) $w = ((i_1, j_1), \ldots (i_{|w|}, j_{|w|}))$ s.t.*

- *for every $k \in [|w|]$,*
  - *$i_k = 1 + pl_k \in [n]$ for some l_k,*
  - *$j_k \in [n']$,*
  - *$h(x[i_k, i_k + p)) = h(y[j_k, j_k + p))$,*
- *$i_1, \ldots, i_{|w|}$ are distinct.*

A non-overlapping matching is a matching with one more restriction.

- *Intervals $[j_k, j_k + p), k \in [|w|]$, are disjoint.*

When considering overlaps, the matching has overlapping degree d, if each bit of y appears in at most d matched pairs for some small number d.

For a match (i, j), it matches two intervals, one from x, the other from y. When we say the y's interval (of the match (i, j)), we mean $[j, j + p)$, and similarly the x's interval is $[i, i + p)$. A match (i, j) in a matching is called a wrong match (or wrong pair) if $x[i, i+p) \neq y[j, j+p)$. Otherwise it is called a correct match (or correct pair). A pair of indices (i, j) is called a potential match between x and y if $h(x[i, i + p)) = h(y[j, j + p))$. It may be wrong because $x[i, i + p)$ may not be $y[j, j + p)$. When x, y are clear from the context we simply say (i, j) is a potential match.

To compute a monotone non-overlapping matching we can use the dynamic programming method in [7]. But our matching is not necessarily monotone. So this raises the question of how hard this problem is.

It seems difficult to find a polynomial algorithm which can exactly compute it. So instead we use constant approximation techniques. There're two difficulties at the first thought. One is that if we compute the non-overlapping matching over the entire strings, then a constant approximation is too bad since there will be $O(n)$ unmatched blocks. So for each level, we restrict our attention to blocks that are uncovered and wrongly recovered (but discovered by us). The other problem is that we need the approximation rate to be a large enough constant. To achieve this goal, we actually computing matchings with constant degree.

We start from a 1/3-approximation algorithm, which is greedy.

▶ **Construction 10.** *Given $n, n', p, q \in \mathbb{N}, p \leq n, p \leq n'$, a polynomial time computable function $h : \{0,1\}^p \to \{0,1\}^q$ and two strings $x \in \{0,1\}^n, y \in \{0,1\}^{n'}$, we have the following 1/3-approximation algorithm for computing the non-overlapping matching.*

1. *Let the sequence of matches w be empty;*
2. *Find $i = 1 + pl \in [n]$ and $j \in [n']$, where $l \in \mathbb{N}$, s.t.*
   - *$h(x[i, i+p)) = h([j, j+p))$,*
   - *i is not in any match (as the first entry) of the current w,*
   - *$[j, j+p)$ does not overlap with any $[j', j'+p)$ for any j' as the second entry in any matches of the current w;*
3. *If there is such a pair of indices i, j, then add the match (i, j) to w and go to step 2; Otherwise, output w and stop.*

▶ **Lemma 11.** *Construction 10 gives a 1/3-approximation algorithm for computing the non-overlapping matching.*

Proof deferred to the full version.

Next we give an explicit algorithm which computes a even larger matching (better approximation), but it allows overlaps.

▶ **Construction 12.** *Given $n, n', p, q \in \mathbb{N}, p \leq n, p \leq n'$, a (polynomial time computable) function $h : \{0,1\}^p \to \{0,1\}^q$ and two strings $x \in \{0,1\}^n, y \in \{0,1\}^{n'}$, we have the following algorithm.*

1. *Let the matching w be empty, set $S = \{i = 1 + pl \mid l \in \mathbb{N}, i \in [n]\}$, integer $c = 0$;*
2. *Conduct Construction 10 to compute a matching w' between x_S and y under h. Here x_S is the projection of x on intervals in set S;*
3. *Let $w = w \cup w'$;*
4. *Let $S = S \setminus \{u \mid \exists (u, v) \in w\}$;*
5. *$c = c + 1$;*
6. *If $c \geq 3$, output w; Otherwise go to step 2.*

Note that Construction 12 is in polynomial time since it simply conducts Construction 10 for 3 times and after each conduction it removes matched blocks of x and only considers the remaining blocks in the next iteration. So we only need to show its correctness.

▶ **Lemma 13.** *Construction 12 computes a degree 3 overlapping matching w between x and y under h, such that $|w| \geq 2/3|w^*|$, where w^* is the maximum non-overlapping matching between x and y under h.*

Proof. Let $w_i, i = 1, 2, 3$ be the matching the algorithm computes after round i. Also let $S_i, i = 1, 2, 3$ be the set S after the ith round.

By Lemma 11, $|w_1| \geq 1/3|w^*|$. The number of unmatched blocks is $\bar{n} - |w_1| \leq \bar{n} - 1/3|w^*|$, where $\bar{n} = \lfloor n/p \rfloor$ is the total number of blocks of x.

The maximum matching between x_{S_1} and y is at least $|w^*| - |w_1|$. This is because that, each of the matched blocks of x by w_1, should be among the x's blocks in the matches of w^*. There are at most $|w_1|$ of them. So there are still $|w^*| - |w_1|$ remaining matches in w^* which corresponds to blocks in x_{S_1}.

Again by Lemma 11, for $i \geq 2$, at least $1/3(|w^*| - |w_{i-1}|)$ blocks of $x_{S_{i-1}}$ will be matched in the ith round.

Thus

$$\begin{aligned}
|w_i| &\geq |w_{i-1}| + 1/3(|w^*| - |w_{i-1}|) && (1)\\
&= 1/3|w^*| + 2/3|w_{i-1}| && (2)\\
&\geq (1-(2/3)^{i-1})|w^*| + (2/3)^{i-1}|w_1| && (3)\\
&\geq (1-(2/3)^{i-1})|w^*| + (1/3)(2/3)^{i-1}|w^*| && (4)\\
&= (1-(2/3)^{i})|w^*|. && (5)
\end{aligned}$$

Inequality 1 is due to Lemma 11 as explained above. Equality 2 is due to a direct computation. 3 is by recursively applying 1 and 2 from $i-1$ to 2. 4 is because $|w_1| \geq 1/3|w^*|$.

As a result, $|w_3| \geq 19/27|w^*| \geq 2/3|w^*|$.

Note that we apply Construction 10 for 3 times, where in each time, it gives a non-overlapping matching. So each entry of y is in at most one of the matches in that round. So finally we get a degree 3 overlapping matching. ◀

We now give the following document exchange protocol.

▶ **Construction 14.** *The protocol works for every input length $n \in \mathbb{N}$, every (k_1, t) block-insertions/deletions k_2 block-transpositions, $k_1, k_2 \leq \alpha n/\log n, t \leq \beta n$, for some constant α, β. (If k_1 or $k_2 > \alpha n/\log n$, or $t > \beta n$, we simply let Alice send her input string.) Let $k = k_1 + k_2$.*

Both Alice's and Bob's algorithms have $L = O(\log \frac{n}{k\log n + t})$ levels.

For every $i \in [L]$, in the i-th level,

- *Let the block size be $b_i = \frac{n}{18\cdot 2^i(k+\frac{t}{\log n})}$, i.e., in each level, divide every block of x in the previous level evenly into two blocks. We choose L properly s.t. $b_L = O(\log n)$;*
- *The number of blocks $l_i = n/b_i$;*

Alice: On input $x \in \{0,1\}^n$,

1. *For the i-th level,*
   a. *Construct a hash function $h_i : \{0,1\}^{b_i} \to \{0,1\}^{b^* = \Theta(\log n)}$ for x by Theorem 8.*
   b. *Compute the sequence of hash values i.e. $v[i] = (h_i(x[1, 1+b_i)), h_i(x[1+b_i, 1+2b_i)), \ldots, h_i(x[1+(l_i-1)b_i, l_i b_i)))$;*
   c. *Compute the redundancy $z[i] \in (\{0,1\}^{b^*})^{\Theta((k+\frac{t}{\log n})i)}$ for $v[i]$ by using an algebraic geometry code[2], where the code has distance at least $180(k+\frac{t}{\log n})i$;*
2. *Compute the redundancy $z_{\text{final}} \in (\{0,1\}^{b_L})^{\Theta((k+\frac{t}{\log n})\log L)}$ for the blocks of the L-th level by using an algebraic geometry code[2]., where the code has distance at least $90(k+\frac{t}{\log n})L$;*
3. *Send $h = (h_1, \ldots, h_L)$, $z = (z[1], z[2], \ldots, z[L])$, $v[1]$, z_{final} to Bob.*

Bob: On input $y \in \{0,1\}^{O(n)}$ and received h, z, $v[1]$, z_{final},

1. *Create $\tilde{x} \in \{0,1,*\}^n$ (i.e. Bob's current version of Alice's x), initiating it to be $(*, *, \ldots, *)$;*
2. *For the i-th level where $1 \leq i \leq L-1$,*
   a. *Apply the decoding of the algebraic geometry code on $h_i(\tilde{x}'[1, 1+b_i)), h_i(\tilde{x}'[1+b_i, 1+2b_i)), \ldots, h_i(\tilde{x}'[1+(l_i-1)b_i, l_i b_i)), z[i]$ to get the sequence of hash values $v[i]$. Note that $v[1]$ is received directly, thus Bob does not need to compute it;*
   b. *Let $S = \{j \in [n] \mid h_i(\tilde{x}[1+(j-1)b_i, 1+jb_i)) \neq v[i][j]$ or $x[1+(j-1)b_i, 1+jb_i) = (*, \ldots, *)\}$;*

[2] See the full version `https://arxiv.org/abs/1809.00725`.

   c. *Compute the matching* $w_i = ((p_1, p'_1), \ldots, (p_{|w|}, p'_{|w|})) \in ([l_i] \times [|y|])^{|w_i|}$ *between* x_S *and* y *under* h_i*, using* $v[i]$*, by Lemma 12;*
   d. *Evaluate* $\tilde{x}$ *according to the matching, i.e. let* $\tilde{x}[p_j, p_j + b_i) = y[p'_j, p'_j + b_i)$*, where* $p_j, p'_j \in w_i, j \in [|w_i|]$*;*
3. *In the* L*'th level, apply the decoding of the algebraic geometry code on the blocks of* $\tilde{x}$ *and* z_{final} *to get* x*;*
4. *Return* x*.*

▶ **Lemma 15.** *For every* i*, the maximum non-overlapping matching between* x_S *and* y *under* h_i *has size at least* $|S| - (2k_1 + 3k_2 + t/\log n)$*.*

▶ **Lemma 16.** *For every* i*,* $|w_i| \geq 2/3(|S| - (2k_1 + 3k_2 + t/\log n))$*.*

▶ **Lemma 17.** *For every* i*, if* $v[1], \ldots, v[i]$ *are correctly recovered, then in the* i*-th level the number of wrongly recovered blocks of* x *is at most* $3i(2k_1 + 3k_2 + \frac{t}{\log n})$*.*

Proof. Consider the matching w^* corresponding to the current recovering of x after i levels, i.e., this matching is generated at level 1 and adjusted level by level. In level j, we first use hash values to test every block to see if it is correctly recovered. For wrongly recovered blocks we delete their corresponding matches. Then for remaining wrongly recovered blocks and unrecovered blocks, we compute a matching w_j for them, and add all matches in w_j to w^*.

For $w_j, j \leq i$, after level i, the number of wrongly recovered blocks in level i caused by (the remaining part of) w_j is at most $3(2k_1 + 3k_2 + \frac{t}{\log n})$.

This is because in w_j is constructed by Construction 12, which is a union of 3 matchings. Each matching of them is non-overlapping. We only need to show that w_j, after eliminating detected wrong pairs in these i levels, contains at most $2k_1 + 3k_2 + \frac{t}{\log n}$ wrong matches between x's and y's blocks in the i-th level. To see this, first note that these matches' y intervals are only from blocks which are modified from x's blocks or newly inserted. For each block-insertion of t_j bits, it can contribute at most $\lceil t_j/b_i \rceil + 1$ wrong matches. Each block-deletion can contribute at most 2 wrong matches. So totally block insertions/deletions can cause $\sum_{j=1}^{k_1}(\lceil t_j/b_i \rceil + 1) \leq 2k_1 + t/b_i$ wrong matches. On the other hand, k_2 block-transpositions can contribute at most $3k_2$ wrong matches, because 1 block-transposition can only cause 1 wrong match when deleting the block and inserting the block to its destination may contribute 2 wrong matches. Hence the total number wrong matches is at most $2k_1 + 3k_2 + t/b_i$.

Since there are i matchings $w_1, \ldots, w_i$, each containing 3 non-overlapping matchings, the number of wrongly recovered blocks remaining in w^* is at most $3i(2k_1 + 3k_2 + \frac{t}{\log n})$. ◀

▶ **Lemma 18.** *For every* i*, if* $v[1], \ldots, v[i]$ *are correctly recovered, then in level* i*, the number of unrecovered blocks is at most* $36i(k + \frac{t}{\log n})$*.*

Proof is in the full version.

▶ **Lemma 19.** *Bob can recover* x *correctly.*

Proof. We use induction to show that for every $i \in [L]$, $v[i]$ can be computed correctly by Bob.

For the first level, $v[1]$ is directly received from Alice.

Assume $v[1], \ldots, v[i-1]$ can be computed correctly. By Lemma 18, the number of unrecovered blocks after level $i-1$ is at most $36(i-1)(k + t/\log n)$. By Lemma 17, the number of wrongly recovered blocks is at most $9(i-1)(k + t/\log n)$. So the total number of wrongly recovered and unrecovered blocks is at most

$$2 \times (36(i-1)(k + t/\log n) + 9(i-1)(k + t/\log n)) \leq 90(i-1)(k + t/\log n) < 90i(k + t/\log n).$$

Note that with the redundancy $z[i]$, its corresponding code has distance at least $180(k+t/b_i)i$. So Bob can recover $v[i]$ correctly by the property of the algebraic geometry code.

As a result, at level L. By Lemma 17, the number of wrongly recovered blocks is at most $3L(2k_1 + 3k_2 + \frac{t}{b_L})$. By Lemma 18 the number of unrecovered blocks, is at most $36L(k + t/\log n)$. So the total number of wrongly recovered and unrecovered blocks is at most $45L(k + t/\log n)$. Note that the code distance corresponding to the redundancy z_{final} is at least $90(k + t/b_L)L$. So all blocks of x can be recovered correctly by using the decoding of the algebraic geometry code. ◀

Communication Complexity and running time computation are in the full version.

To this end, we showed Theorem 1.

3 Error Correcting Codes

We now briefly describe how to construct an error correcting code from a document exchange protocol for block edit errors. Similar to the construction in [7], our starting point is to first encode the sketch of the document exchange protocol using the code by Schulman and Zuckerman [23], which can resist edit errors and block transpositions. Then we concatenate the message with the encoding of the sketch. When decoding, we first decode the sketch, then apply the document exchange protocol on Bob's side to recover the message.

However, here we have an additional issue with this approach: a block transposition may move some parts of the encoding of the sketch to somewhere in the middle of the message, or vice versa. In this case, we won't be able to tell which part of the received string is the encoding of the sketch, and which part is the original message.

To solve this issue, we use a fixed string $\mathsf{buf} = 0^{\ell_{\mathsf{buf}}} \circ 1$ as a buffer to mark the encoding of the sketch, for some $\ell_{\mathsf{buf}} = O(\log n)$. More specifically, we evenly divide the encoding of the sketch into small blocks of length ℓ_{buf}, and insert buf before every block. Note that this only increases the length of the encoding of the sketch by a constant factor. The reason we use such a small block length is that, even if the adversary can forge or destroy some buffers, the total number of bits inserted or deleted caused by this is still small. In fact, we can bound this by $O(k)$ block insertions/deletions with at most $O(k \log n)$ bits inserted/deleted, for which both the sketch and the encoding of the sketch can handle. When decoding, we first recognize all the buf's. Then we take the ℓ_{buf} bits after each buf to form the decoding of the sketch, and take the remaining bits as the message.

Unfortunately, this approach introduces two additional problems here. The first problem is that the original message may contain buf as a substring. If this happens then in the decoding procedure again we will be taking part of the message to be in the encoding of the sketch. The second problem is that the small blocks of the encoding of the sketch may also contain buf. In this case we will be deleting information from the encoding of the sketch, which causes too many edit errors.

To address the first problem, we turn the original message into a *pseudorandom* string by computing the XOR of the message with the output of an appropriate pseudorandom generator that has seed length $O(\log n)$. We show that with high probability buf does not appear as a substring in the XOR. We can then exhaustively search for a seed that satisfies this requirement, and append the seed to the sketch of the document exchange protocol.

To address the second problem, we choose the length of the buffer to be longer than the length of each block in the encoding of the sketch, so that buf doesn't appear as a substring in any block. This is exactly why we choose the length of the buffer to be $\ell_{\mathsf{buf}} + 1$ while we choose the length of each block to be ℓ_{buf}.

If we directly apply our document exchange protocol to the construction above, we obtain an error correcting code with $O((k \log n + t) \log^2 \frac{n}{k \log n + t})$ redundant bits. Next we discuss how to achieve better redundancy for small k and t.

We first briefly describe the construction of the explicit binary code for k edit errors with redundancy $O(k \log n)$ in [7]. The construction in [7] starts by transforming the message into a string with the *B-distinct* property: any two substrings of length some $B = O(\log n)$ are distinct. This is obtained by computing the XOR of the message with the output of an appropriately designed pseudorandom generator. The construction then designs a document exchange protocol for such a string, and encodes the sketch of the document exchange protocol to give an error correcting code.

The document exchange protocol for a B-distinct string in [7] actually consists of two stages: in stage I, Alice uses a fixed pattern p to divide her string into blocks of size $\mathsf{poly}(\log n)$. Next, Alice sends a sketch of size $O(k \log n)$ to help Bob recover the partition of her string. To achieve this, Bob also divides his string into blocks in the same way that Alice does. Alice creates a vector V where each entry of V is indexed by a binary string of length B. Specifically, Alice looks at each block in her partition, and stores the B-prefix (the prefix of length B) of its next block and the length of the current block in the entry of V indexed by the B-prefix of the current block. This ensures each entry of the vector V has only $O(\log n)$ bits. Bob creates a vector V' in the same way. [7] shows that V and V' differ in at most $O(k)$ entries, thus Alice can send a sketch of size $O(k \log n)$ using the Reed-Solomon Code to help Bob recover V from V'. Once this is done, Bob can use V to obtain a guess of Alice's string.

Stage II consists of several levels. In each level, both parties divide each of their blocks evenly into $O(\log^{0.4} n)$ smaller blocks, and Alice generates a sequence of special hash functions called *ϵ-synchronization hash functions*. The nice properties of these hash functions guarantee that in each level Alice can send $O(k \log n)$ bits to Bob, so that Bob can recover all but $O(k)$ blocks of Alice's string. This stage ends in $O(\log_{\log^{0.4} n}(\mathsf{poly}(\log n))) = O(1)$ levels, where in the last level Alice can simply send a sketch of size $O(k \log n)$ for Bob to recover her string x.

Checking these two stages, it turns out that stage I can be modified to work for block edit errors as well. Intuitively, this is because it is still true that such errors won't cause too many different blocks between V and V'. On the other hand, stage II becomes problematic, since the use of ϵ-synchronization hash functions crucially relies on the monotone property of standard edit errors. Allowing block transpositions ruins this property, and it is not clear how to give suitable ϵ-synchronization hash functions to work in this case.

To solve the issue, in stage II, we can apply the deterministic document exchange protocol we developed earlier. This implies an error correcting code of redundancy $O(k \log n \log\log n + t)$. However, we show that we can further reduce the redundancy to $O(k \log n \log\log\log n + t)$ by using the string parsing idea in [8] to improve the partition in Stage I.

Given an input string, string parsing builds a tree where each leaf corresponds to a symbol of the input string, and each non-leaf node corresponds to a substring of the input string. Each node of the tree is associated with a label, which is the hash value of its corresponding substring under some hash function. The structure of the tree only depends locally on the input string, e.g., an edit error on the input string only affects $O(\log n \log^* n)$ nodes.

More specifically, string parsing builds the tree bottom-up from one level to another. The labels in the bottom level are obtained by directly applying the hash function to the symbols. Then, the algorithm builds one level of the tree as follows. The labels of the nodes in the previous level form a string of alphabet size $\mathsf{poly}(n)$. The algorithm first finds all repetitive substrings in this string (we say a substring is repetitive, if it's of the form a^l, for some $l \geq 2$). The remaining substrings satisfy the property that any two adjacent symbols are different, and we say such substrings are *non-repetitive*. [8] then applies an alphabet reduction algorithm to the non-repetitive substrings, and obtains a new non-repetitive string

for each substring over the alphabet $\{0,1,2\}$. The alphabet reduction works in $\log^* n$ steps, where in each step the alphabet size is reduced from the current size a to $\log a$. Thus in $\log^* n$ steps the alphabet size becomes a constant. Now for all the new strings obtained, the algorithm finds local maximums and local minimums that are not adjacent to any local maximum as *landmarks*, and partition the strings into small blocks of length 2 or 3 by using the landmarks. Finally, for each block, the algorithm builds a new node in this level, whose children are the nodes in the block and whose label is the hash value of the subtree.

Here, in our construction of error correcting codes, we use the idea of string parsing in stage I to partition Alice's string x into small blocks. Our goal is to partition the string into blocks of length roughly $\Theta(\log n \cdot \mathsf{poly}(\log\log n))$, while an edit error on the string can only affect a small number of contiguous blocks. In this way, stage II only takes $O(\log\log\log n)$ levels and the sketch size in stage II is $O(k\log n\log\log\log n + t)$. Note that each node in the parsing tree depends only locally on the input string. We use this property to bound the number of errors among the small blocks obtained in stage I.

To achieve our goal, instead of building a full parsing tree, we only build a partial parsing tree. That is, in each level of the parsing tree, we check the number of leaves under each node. If a node has more than T leaves for some threshold T, we mark the node as "finish". We also mark a node as "frozen", if all its adjacent nodes are marked as "finish". For each "finish" node, we build a new node in the next level, with the only child being this "finish" node. We then use these "finish" nodes to divide the string into several substrings, and apply alphabet reduction to the substrings, choose the landmarks, and partition each substring into small blocks according to the landmarks. Then for each small block, we build a new node in the next level, and set the children of the new node to be all nodes in the same block. We keep doing this until each node is either marked as "finish" or "frozen". Finally, we merge each "frozen" node to the "finish" node on its left or right. At the end of this process, we obtain several trees, and we partition the string x into small blocks, where each block consists of all the leaves in a tree. To remove the $O(\log^* n)$ factor, we only do two levels of alphabet reduction in each level of the tree. However, this will result in an alphabet size of $O(\log\log n)$, which means the tree may have $O(\log\log n)$ children. Hence, the block size may be as large as $O(T\log\log n)$. Note that each block depends on $O(\log T)$ blocks on its left and right, since in each level of the partial parsing tree, each node depends locally on a constant number of adjacent nodes. We prove that, if y is obtained from x by (k,t) block edit errors, then the partition of y can be obtained from the partition of x by $(k, O(t/T + k\log T))$ block edit errors over a larger alphabet. If we set $T = \log n$, then in stage I Alice still needs to send a sketch of $O(k\log n\log\log n + t)$ bits. To further reduce the redundancy, we apply the partial parsing tree method again with another threshold $T' = \Theta(\log\log n)$. Now the errors are reduced to $(k, O(\frac{t}{TT'} + k\log T'))$ block edit errors over a larger alphabet, and the block size increases by a $O(T'\log\log n\log\log\log n)$ factor, and becomes $O(\log n(\log\log n)^2)$.

We show that now in stage I, Alice can send a sketch with $O(\frac{t}{TT'} + k\log T')\cdot O(\log n) = O(k\log n\log\log\log n + t)$ bits; and in stage II, Alice can send a sketch with $O(k\log n\log\log\log n + t)$ bits. So the total sketch size is still $O(k\log n\log\log\log n + t)$. By using the encoding of Schulman and Zuckerman [23] and the buffer buf, the final redundancy of the error correcting code is also $O(k\log n\log\log\log n + t)$.

References

1 Djamal Belazzougui. Efficient Deterministic Single Round Document Exchange for Edit Distance. *CoRR*, abs/1511.09229, 2015. `arXiv:1511.09229`.

2 Djamal Belazzougui and Qin Zhang. Edit Distance: Sketching, Streaming, and Document Exchange. In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science*, pages 51–60. IEEE, 2016.

3 J. Brakensiek, V. Guruswami, and S. Zbarsky. Efficient Low-Redundancy Codes for Correcting Multiple Deletions. *IEEE Transactions on Information Theory*, PP(99):1–1, 2017. doi:10.1109/TIT.2017.2746566.

4 Boris Bukh and Venkatesan Guruswami. An improved bound on the fraction of correctable deletions. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1893–1901. ACM, 2016.

5 Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Low Distortion Embedding from Edit to Hamming Distance using Coupling. In *Proceedings of the 48th IEEE Annual Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 2016.

6 K. Cheng, B. Haeupler, X. Li, A. Shahrasbi, and K. Wu. Synchronization Strings: Efficient and Fast Deterministic Constructions over Small Alphabets. *ArXiv e-prints*, March 2018. arXiv:1803.03530.

7 Kuan Cheng, Zhengzhong Jin, Xin Li, and Ke Wu. Deterministic Document Exchange Protocols, and Almost Optimal Binary Codes for Edit Errors. In *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018.

8 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1), 2007.

9 Graham Cormode, Mike Paterson, Suleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 197–206. ACM, 2000.

10 V. Guruswami and R. Li. Efficiently decodable insertion/deletion codes for high-noise and high-rate regimes. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 620–624, July 2016. doi:10.1109/ISIT.2016.7541373.

11 V. Guruswami and C. Wang. Deletion Codes in the High-Noise and High-Rate Regimes. *IEEE Transactions on Information Theory*, 63(4):1961–1970, April 2017. doi:10.1109/TIT.2017.2659765.

12 Bernhard Haeupler. Optimal Document Exchange and New Codes for Small Number of Insertions and Deletions. *arXiv preprint*, 2018. arXiv:1804.03604.

13 Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization strings: codes for insertions and deletions approaching the Singleton bound. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 33–46. ACM, 2017.

14 Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization Strings: Explicit Constructions, Local Decoding, and Applications. In *Proceedings of the 50th Annual ACM Symposium on Theory of Computing*, 2018.

15 Bernhard Haeupler, Amirbehshad Shahrasbi, and Ellen Vitercik. Synchronization Strings: Channel Simulations and Interactive Coding for Insertions and Deletions. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming*, 2018.

16 Tom Høholdt, Jacobus H Van Lint, and Ruud Pellikaan. Algebraic geometry codes. *Handbook of coding theory*, 1(Part 1):871–961, 1998.

17 Utku Irmak, Svilen Mihaylov, and Torsten Suel. Improved single-round protocols for remote file synchronization. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1665–1676. IEEE, 2005.

18 Hossein Jowhari. Efficient Communication Protocols for Deciding Edit Distance. In *ESA*, 2012.

19 V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

20 H. Mercier, V. K. Bhargava, and V. Tarokh. A survey of error-correcting codes for channels with symbol synchronization errors. *IEEE Communications Surveys Tutorials*, 12(1):87–96, First 2010. doi:10.1109/SURV.2010.020110.00079.

21 A. Orlitsky. Interactive communication: balanced distributions, correlated files, and average-case complexity. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 228–238, October 1991. doi:10.1109/SFCS.1991.185373.

22 G. M. Tenengol'ts R. R. Varshamov. Code Correcting Single Asymmetric Errors. *Avtomat. i Telemekh*, 26:288–292, 1965.

23 L. J. Schulman and D. Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory*, 45(7):2552–2557, November 1999. doi:10.1109/18.796406.

24 D Shapira and J. A. Storer. Edit distance with move operations. In *Proceedings of the 13th Symposium on Combinatorial Pattern Matching*, pages 85–98, 2002.