2018 IEEE 59th Annual Symposium on Foundations of Computer Science

Deterministic Document Exchange Protocols, and Almost Optimal Binary Codes for Edit Errors

Kuan Cheng*, Zhengzhong Jin†, Xin Li‡ and Ke Wu§
Department of Computer Science
Johns Hopkins University, Baltimore, USA
* *Email: kcheng17@jhu.edu*
† *Email: zjin12@jhu.edu*
‡ *Email: lixints@cs.jhu.edu*
§ *Email: AshleyMo@jhu.edu*

***Abstract*—We study two basic problems regarding edit errors (insertions and deletions). The first one is document exchange, where two parties Alice and Bob hold two strings x and y with a bounded edit distance k. The goal is to have Alice send a short sketch to Bob, so that Bob can recover x based on y and the sketch. The second one is the fundamental problem of designing error correcting codes for edit errors, where the goal is to construct an explicit code to transmit a message x through a channel that can add at most k worst case insertions and deletions, so that the original message x can be successfully recovered at the other end of the channel. Both problems have been extensively studied for decades, and in this paper we focus on deterministic document exchange protocols and binary codes for insertions and deletions (insdel codes). If the length of x is n, then it is known that for small k (e.g., $k \leq n/4$), in both problems the optimal sketch size or the optimal number of redundant bits is $\Theta(k \log \frac{n}{k})$. In particular, this implies the existence of binary codes that can correct ε fraction of insertions and deletions with rate $1 - \Theta(\varepsilon \log(\frac{1}{\varepsilon}))$. However, known constructions are far from achieving these bounds.**

In this paper we significantly improve previous results on both problems. For document exchange, we give an efficient deterministic protocol with sketch size $O(k \log^2 \frac{n}{k})$. This significantly improves the previous best known deterministic protocol, which has sketch size $O(k^2 + k \log^2 n)$ [2]. For binary insdel codes, we obtain the following results:

1) **An explicit binary insdel code which encodes an n-bit message x against k errors with redundancy $O(k \log^2 \frac{n}{k})$. In particular this implies an explicit family of binary insdel codes that can correct ε fraction of insertions and deletions with rate $1 - O(\varepsilon \log^2(\frac{1}{\varepsilon})) = 1 - \widetilde{O}(\varepsilon)$. This significantly improves the previous best known result which only achieves rate $1 - \widetilde{O}(\sqrt{\varepsilon})$ [11], [10], and is optimal up to a $\log(\frac{1}{\varepsilon})$ factor.**
2) **An explicit binary insdel code which encodes an n-bit message x against k errors with redundancy $O(k \log n)$. This significantly improves the previous best known result of [4], which only works for constant k and has redundancy $O(k^2 \log k \log n)$; and that of [2], which has redundancy $O(k^2 + k \log^2 n)$. Our code has optimal redundancy for $k \leq n^{1-\alpha}$, any constant $0 < \alpha < 1$. This is the first explicit construction of binary insdel codes that has optimal redundancy for a wide range of error parameters k, and this brings our understanding of binary insdel codes much closer to that of standard binary error correcting codes.**

In obtaining our results we introduce several new techniques. Most notably, we introduce the notion of *ε-self matching hash functions* and *ε-synchronization hash functions*. We believe our techniques can have further applications in the literature.

***Keywords*-Edit errors, Document exchange, Error correcting codes**

I. Introduction

Given two strings x, y over some finite alphabet Σ, the edit distance between them $\mathsf{ED}(x, y)$ is defined as the minimum number of edit operations (insertions, deletions and substitutions) to change x into y. Being one of the simplest metrics, edit distance has been extensively studied due to its wide applications in different areas. For example, in natural language processing, edit distance is used in automatic spelling correction to determine possible corrections of a misspelled word; and in bioinformatics it can be used to measure the similarity between DNA sequences. In this paper, we study the general question of recovering from errors caused by edit operations. Note that without loss of generality we can only consider insertions and deletions, since a substitution can be replace by a deletion followed by an insertion, and this at most doubles the number of operations. Thus from now on we will only be interested in insertions and deletions, and we define $\mathsf{ED}(x, y)$ to be the minimum number of such operations required to change x into y.

Insertion and deletion errors happen frequently in practice. For example, they occur in the process of reading magnetic and optical media, in genetic mutation where DNA sequences may change, and in internet protocols where some packets may get lost during routing. Another typical situation where these errors can occur is in distributed file systems, e.g., when a file is stored in different machines and being edited by different people working on the same project. These files then may have different versions that need to be synchronized to remove the edit errors. In

Supported by NSF Grant CCF-1617713.

this context, we study the following two basic problems regarding insertion and deletion errors.

- *Document exchange.* In this setting, two parties Alice and Bob each holds a string x and y, and we assume that their edit distance is bounded by some parameter k. The goal is to have Alice send a sketch to Bob based on her string x and the edit distance bound k, such that Bob can recover Alice's string x based on his string y and the sketch. Naturally, we would like to require both the message length and the computation time of Alice and Bob to be as small as possible.
- *Error correcting codes.* In this setting, two parties Alice and Bob are linked by a channel where the number of worst case insertion and deletions is bounded by some parameter k. Given any message, the goal is to have Alice send an encoding of the message to Bob through the channel, so that despite any possible insertion and deletion errors that may happen, Bob can recover the correct message after receiving the (possibly modified) codeword. Again, we would like to minimize both the codeword length (or equivalently, the number of redundant bits) and the encoding/decoding time. This is a generalization of the classical error correcting codes for Hamming errors.

It can be seen that these two problems are closely related. In particular, a solution to the document exchange problem can often be used to construct an error correcting code for insertion and deletion errors. In this paper, we focus on the setting where the strings have a binary alphabet, arguably the most popular and important setting in computer science.[1] In this case, assume that Alice's string (or the message she wants to send) has length n, then it is known that for small k (e.g., $k \leq n/4$) both the optimal sketch size in document exchange and the optimal number of redundant bits in an error correcting code is $\Theta(k \log(\frac{n}{k}))$, and this is true even for Hamming errors. In addition, both optimum can be achieved using exponential time, with the first one using a greedy coloring algorithm and the second one using a greedy sphere packing algorithm (which is essentially what gives the Gilbert-Varshamov bound).

It turns out that in the case of Hamming errors, both optimum (up to constants) can also be achieved efficiently in polynomial time. This is done by using sophisticated linear Algebraic Geometric codes [15]. As a special case, one can use Reed-Solomon codes to achieve $O(k \log n)$ in both problems. However, the situation becomes much harder once we switch to edit errors, and our understanding of these two basic problems lags far behind the case of Hamming errors. We now survey related previous work below.

Document exchange.: Historically, Orlitsky [20] was the first one to study the document exchange problem. His work gave protocols for generally correlated strings x, y using a greedy graph coloring algorithm, and in particular he obtained a deterministic protocol with sketch size $O(k \log n)$ for edit errors. However, the running time of the protocol is exponential in k. The main question left there is whether one can design a document exchange protocol that is both communication efficient and time efficient.

There has been considerable progress afterwards [8], [16], [17]. Specifically, Irmak et al. [16] gave a randomized protocol that achieves sketch size $O(k \log(\frac{n}{k}) \log n)$ and running time $\tilde{O}(n)$. Independently, Jowhari [17] also obtained a randomized protocol with sketch size $O(k \log^2 n \log^* n)$ and running time $\tilde{O}(n)$. A recent work by Chakraborty et al. [6] introduced a clever randomized embedding from the edit distance metric to the Hamming distance metric, and thus obtained a protocol with sketch size $O(k^2 \log n)$ and running time $\tilde{O}(n)$. Using the embedding in [6], Belazzougui and Zhang [3] gave an improved randomized protocol with sketch size $O(k(\log^2 k + \log n))$ and running time $\tilde{O}(n + \mathsf{poly}(k))$, where the sketch size is asymptotically optimal for $k = 2^{O(\sqrt{\log n})}$.

All of the above protocols, except the exponential time protocol of Orlitsky [20], are however randomized. In practice, a deterministic protocol is certainly more useful than a randomized one. Thus one natural and important question is whether one can construct a deterministic protocol for document exchange with small sketch size (e.g., polynomial in $k \log n$) and efficient computation. This question is also important for applications in error correcting codes, since a randomized document exchange protocol is not very useful in designing such codes. It turns out that this question is quite tricky, and no such deterministic protocols are known even for $k > 1$ until the work of Belazzougui [2] in 2015, where he gave a deterministic protocol with sketch size $O(k^2 + k \log^2 n)$ and running time $\tilde{O}(n)$.

Error correcting codes.: Error correcting codes are fundamental objects in both theory and practice. Starting from the pioneering work of Shannon, Hamming and many others, error correcting codes have been intensively studied in the literature. This is true for both standard Hamming errors such as symbol corruptions and erasures, and edit errors such as insertions and deletions. While the study of codes against standard Hamming errors has been a great success, leading to a near complete knowledge and a powerful toolbox of techniques together with explicit constructions that match various bounds, our understanding of codes for insertion and deletion errors (insdel codes for short) is still rather poor. Indeed, insertion and deletion errors are strictly more general than Hamming errors, and the study of codes against such errors has resisted progress for quite some time, as demonstrated by previous work which we discuss below.

Since insertion and deletion errors are strictly more general than Hamming errors, all the upper bounds on the rate of standard codes also apply to insdel codes. Moreover,

[1]Although, our document exchange protocols can be easily extended to larger alphabets, we omit the details here.

by using a similar sphere packing argument, similar lower bounds on the rate (such as the Gilbert-Varshamov bound) can also be shown. In particular, one can show (e.g., [18]) that for binary codes, to encode a message of length n against k insertion and deletion errors with $k \leq n/2$, the optimal number of redundant bits is $\Theta(k \log(\frac{n}{k}))$; and to protect against ε fraction of insertion and deletion errors, the optimal rate of the code is $1 - \Theta(\varepsilon \log(\frac{1}{\varepsilon}))$. On the other hand, if the alphabet of the code is large enough, then one can potentially recover from an error fraction approaching 1 or achieve the singleton bound: a rate $1 - \varepsilon$ code that can correct ε fraction of insertion and deletion errors.

However, achieving these goals have been quite challenging. In 1966, Levenshtein [18] first showed that the Varshamov-Tenengolts code [21] can correct one deletion with roughly $\log n$ redundant bits, which is optimal. Since then many constructions of insdel codes have been given, but all constructions are far from achieving the optimal bounds. In fact, even correcting two deletions requires $\Omega(n)$ redundant bits, and even the first explicit asymptotically good insdel code (a code that has constant rate and can also correct a constant fraction of insertion and deletion errors) over a constant alphabet did not appear until the work of Schulman and Zuckerman in 1999 [22], who gave such a code over the binary alphabet. We refer the reader to the survey by Mercier et al. [19] for more details about the extensive research on this topic.

In the past few years, there has been a series of work trying to improve the situation for both the binary alphabet and larger alphabets. Specifically, for larger alphabets, a line of work by Guruswami et. al [11], [10], [5] constructed explicit insdel codes that can correct $1 - \varepsilon$ fraction of errors with rate $\Omega(\varepsilon^5)$ and alphabet size $\mathsf{poly}(1/\varepsilon)$; and for a fixed alphabet size $t \geq 2$ explicit insdel codes that can correct $1 - \frac{2}{t+1} - \varepsilon$ fraction of errors with rate $(\varepsilon/t)^{\mathsf{poly}(1/\varepsilon)}$. These works aim to tolerate an error fraction approaching 1 by using a sufficiently large alphabet size. Another line of work by Haeupler et al [13], [14], [7] introduced and constructed a combinatorial object called *synchronization string*, which can be used to transform standard error correcting codes into insdel codes, at the price of increasing the alphabet size. Using explicit constructions of synchronization strings, [13] achieved explicit insdel codes that can correct δ fraction of errors with rate $1 - \delta - \varepsilon$ (hence approaching the singleton bound), although the alphabet size is exponential in $\frac{1}{\varepsilon}$.

For the binary alphabet, which is the focus of this paper, it is well known that no code can tolerate an error fraction approaching 1 or achieve the singleton bound. Instead, the major goal here is to construct explicit insdel codes for some small fraction (ε) or some small number (k) of errors that can achieve the optimal rate of $1 - \Theta(\varepsilon \log(\frac{1}{\varepsilon}))$ or the optimal redundancy of $\Theta(k \log(\frac{n}{k}))$, which is analogous to achieving the Gilbert-Varshamov bound for standard error correcting codes. Slightly less ambitiously, one can ask to achieve redundancy $O(k \log n)$, which is optimal when $k \leq n^{1-\alpha}$ for any constant $\alpha > 0$, and easy to achieve in the case of Hamming errors by using Reed-Solomon codes. In this context, Guruswami et. al [11], [10] constructed explicit insdel codes that can correct ε fraction of errors with rate $1 - \tilde{O}(\sqrt{\varepsilon})$, which is the best possible by using code concatenation. For any fixed constant k, another work by Brakensiek et. al [4] constructed an explicit insdel code that can encode an n-bit message against k insertions/deletions with $O(k^2 \log k \log n)$ redundant bits, which is asymptotically optimal when k is a fixed constant. We remark that the construction in [4] only works for constant k, and does not give anything when k becomes larger (e.g., $k = \log n$). Finally, using his deterministic document exchange protocol, Belazzougui [2] constructed an explicit insdel code that can encode an n-bit message against k insertions/deletions with $O(k^2 + k \log^2 n)$ redundant bits. In summary, there remains a huge gap between the known constructions and the optimal bounds in the case of a binary alphabet. In particular, even achieving $O(k \log n)$ redundancy has been far out of reach.

A. Our results

In this paper we significantly improve the situation for both document exchange and binary insdel codes. Our new constructions of explicit insdel codes are actually almost optimal for a wide range of error parameters k. First, we have the following theorem which gives an improved deterministic document exchange protocol.

Theorem I.1. *There exists a single round deterministic protocol for document exchange with communication complexity (sketch length) $O(k \log^2 \frac{n}{k})$, time complexity $\mathsf{poly}(n)$, where n is the length of the string and k is the edit distance upper bound.*

Note that this theorem significantly improves the sketch size of the deterministic protocol in [2], which is $O(k^2 + k \log^2 n)$. In particular, our protocol is interesting for k up to $\Omega(n)$ while the protocol in [2] is interesting only for $k < \sqrt{n}$.

We can use this theorem to get improved binary insdel codes that can correct ε fraction of errors.

Theorem I.2. *There exists a constant $0 < \alpha < 1$ such that for any $0 < \varepsilon \leq \alpha$ there exists an explicit family of binary error correcting codes with codeword length n and message length m, that can correct up to $k = \varepsilon n$ edit errors with rate $m/n = 1 - O(\varepsilon \log^2 \frac{1}{\varepsilon})$.*

Note that the rate of the code is $1 - O(\varepsilon \log^2 \frac{1}{\varepsilon}) = 1 - \widetilde{O}(\varepsilon)$, which is optimal up to an extra $\log(\frac{1}{\varepsilon})$ factor. This significantly improves the rate of $1 - \widetilde{O}(\sqrt{\varepsilon})$ in [11], [10].

For the general case of k errors, the document exchange protocol also gives an insdel code.

Theorem I.3. *For any $n, k \in \mathsf{N}$ with $k \leq n/4$, there exists an explicit binary error correcting code with message length n, codeword length $n + O(k \log^2 \frac{n}{k})$ that can correct up to k edit errors.*

When k is small, e.g., $k = n^{\alpha}$ for some constant $\alpha < 1$, the above theorem gives $O(k \log^2 n)$ redundant bits. Our next theorem shows that we can do better, and in fact we can achieve redundancy $O(k \log n)$, which is asymptotically optimal for small k.

Theorem I.4. *For any $n, k \in \mathsf{N}$, there exists an explicit binary error correcting code with message length n, codeword length $n + O(k \log n)$ that can correct up to k edit errors.*

Note that in this theorem, the number of redundant bits needed is $O(k \log n)$, which is asymptotically optimal for $k \leq n^{1-\alpha}$, any constant $0 < \alpha < 1$. This significantly improves the construction in [4], which only works for constant k and has redundancy $O(k^2 \log k \log n)$, and the construction in [2], which has redundancy $O(k^2 + k \log^2 n)$. In fact, this is the first explicit construction of binary insdel codes that have optimal redundancy for a wide range of error parameters k, and this brings our understanding of binary insdel codes much closer to that of standard binary error correcting codes.

Remark I.5. *In all our insdel codes, both the encoding function and the decoding function run in time* $\mathsf{poly}(n)$.

Independent work.: In a recent independent work [12], Haeupler also studied the document exchange problem and binary error correcting codes for edit errors. Specifically, he also obtained a deterministic document exchange protocol with sketch size $O(k \log^2 \frac{n}{k})$, which leads to an error correcting code with the same redundancy, thus matching our theorems I.1, I.2 and I.3. He further gave a randomized document exchange protocol that has optimal sketch size $O(k \log \frac{n}{k})$. However, for small k, our Theorem I.4 gives a much better error correcting code. Our code is optimal for $k \leq n^{1-\alpha}$, any constant $0 < \alpha < 1$, and thus better than the code given in [12].

B. Overview of the techniques

In this section we provide a high level overview of the ideas and techniques used in our constructions. We start with the document exchange protocol.

Document exchange.: Our starting point is the randomized protocol by Irmak et al. [16], which we refer to as the IMS protocol. The protocol is one round, but Alice's algorithm to generate the message proceeds in $L = O(\log(\frac{n}{k}))$ levels. In each level Alice computes some sketch about her string x, and her final message to Bob is the concatenation of the sketches. After receiving the message, Bob's algorithm also proceeds in h levels, where in each level he uses the corresponding sketch to recover part of x.

More specifically, in the first level Alice divides her string into $\Theta(k)$ blocks where each block has size $O(\frac{n}{k})$, and in each subsequent level every block is divided evenly into two blocks, until the final block size becomes $O(\log n)$. This takes $O(\log(\frac{n}{k}))$ levels. Using shared randomness, in each level Alice picks a set of random hash functions, one for each block which outputs $O(\log n)$ bits, and computes the hash values. In the first level, Alice's sketch is just the concatenation of the $O(k)$ hash values. In all subsequent levels, Alice obtains the sketch in this level by computing the redundancy of a systematic error correcting code (e.g., the Reed-Solomon code) that can correct $O(k)$ erasures and symbol corruptions, where each symbol has $O(\log n)$ bits (the hash value). Note that this sketch has size $O(k \log n)$ and thus the total sketch size is $O(k \log n \log(\frac{n}{k}))$.

On Bob's side, he always maintains a string $\tilde{x}$ which is the partially corrected version of x. Initially $\tilde{x}$ is the empty string, and in each level Bob tries to use his string y to fill $\tilde{x}$. This is done as follows. In each level Bob first tries to recover all the hash values of Alice in this level (notice that the hash values of the first level are directly sent to Bob). Suppose Bob has successfully recovered all the hash values, Bob then tries to match every block of Alice's string in this level with one substring of the same length in his string y, by finding such a substring with the same hash value. We say such a match is bad if the substring Bob finds is not the same as Alice's block (i.e., a hash collision). The key idea here is that if the hash functions output $O(\log n)$ bits, and they are chosen independently randomly, then with high probability a bad match only happens if the substring Bob finds contains at least one edit error. Moreover, Bob can find at least $l_i - k$ matches, where l_i is the number of blocks in the current level. Bob then uses the matched substrings to fill the corresponding blocks in $\tilde{x}$, and leaves the unmatched blocks blank. From the above discussion, one can see that there are at most k unmatched blocks and at most k mismatched blocks. Therefore in the next level when both parties divide every block evenly into two blocks, x and $\tilde{x}$ have at most $4k$ different blocks. This implies that there are also at most $4k$ different hash values in the next level, and hence Bob can correctly recover all the hash values of Alice using the redundancy of the error correcting code.

Our deterministic protocol for document exchange is a derandomized version of the IMS protocol, with several modifications. First, we observe that the IMS protocol as we presented above, can already be derandomzied. This is because that to ensure a bad match only happens if the substring Bob finds contains at least one edit error, we in fact just need to ensure that under any hash function, no *block* of x can have a collision with a *substring* of the same length in x itself. We emphasize one subtle point here: Alice's hash function is applied to a block of her string x, while when trying to fill $\tilde{x}$, Bob actually checks every substring of the string y. Therefore we need to consider hash collisions between blocks of x and substrings of x. If the hash functions are chosen independently uniformly, then such a collision happens with probability $1/\mathsf{poly}(n)$, and thus by a union bound with high probability no collision happens. However, notice that if we write out the outputs of all hash functions on all inputs, then any collision is only concerned with two outputs which consists of $O(\log n)$ bits. Thus it's enough to use $O(\log n)$-wise independence to generate these outputs. To further save the random bits used, we can instead use an almost κ-wise independent sample space with $\kappa = O(\log n)$ and error $\varepsilon = 1/\mathsf{poly}(n)$. Using for example the construction by Alon et. al. [1], this results in a total of $O(\log n)$ random bits (the seed), and thus Alice can exhaustively search for a fixed set of hash functions in polynomial time. Now in each level, Alice's sketch will also include the specific seed that is used to generate the hash functions, which has $O(\log n)$ bits. Note this only adds $O(\log n \log \frac{n}{k})$ to the final sketch size. Bob's algorithm is essentially the same, except now in each level he needs to use the seed to compute the hash functions.

The above construction gives a deterministic document exchange protocol with sketch size $O(k \log n \log \frac{n}{k})$, but our goal is to further improve this to $O(k \log^2 \frac{n}{k})$. The key idea here is to use a relaxed version of hash functions with nice "self matching" properties. To motivate our construction, first observe that in each level, when Bob tries to match every block of Alice's string with one substring of the same length in his string y, it is not only true that Bob can find a matching of size at least $l_i - k$ (where l_i is the number of blocks in this level), but also true that Bob can find a *monotone* matching of at least this size. A monotone matching here means a matching that does not have edges crossing each other. In this monotone matching, there are at most k bad matches caused by edit errors, and thus there exists a *self matching* between x and itself with size at least $l_i - 2k$. In the previous construction, we in fact ensure that all these $l_i - 2k$ matches are correct. To achieve better parameters, we instead relax this condition and only require that at most k of these self matches are bad. Note if this is true then the total number of different blocks between x and $\tilde{x}$ is still $O(k)$ and we can again use an error correcting code to send the redundancy of hash values in the next level.

This relaxation motivates us to introduce *ε-self matching hash functions*, which is similar in spirit to ε-self matching strings introduced in [13]. Formally, we have the following definitions.

Definition I.6. *(monotone matching) For every $n, n', t, p, q \in \mathbb{N}, q \le p$, any hash functions $h_1, h_2, \dots, h_{n'}$ where $\forall i \in [n'], h_i : \{0,1\}^p \to \{0,1\}^q$, given two strings $x \in (\{0,1\}^p)^{n'}$ and $y \in \{0,1\}^n$, a monotone matching of size t between x, y under hash functions $h_1, \dots, h_{n'}$ is a sequence of pairs of indices $w = ((i_1, j_1), (i_2, j_2), \dots, (i_t, j_t)) \in ([n'] \times [n])^t$ s.t. $i_1 < i_2 < \dots < i_t$, $j_1 + p - 1 < j_2, \dots, j_{t-1} + p - 1 < j_t, j_t + p - 1 \le n$ and $\forall l \in [t], h_{i_l}(x[i_l]) = h_{i_l}(y[j_l, j_l + p - 1])$. When $h_1, \dots, h_{n'}$ are clear from the context, we simply say that w is a monotone matching between x, y.*

For $l \in [t]$, if $x[i_l] = y[j_l, j_l + p - 1]$, we say (i_l, j_l) is a good match, otherwise we say it is a bad match. We say w is a correct matching if all matches in w are good. We say w is a completely wrong matching is all matches in w are bad.

If x and y are the same in terms of their binary expression, then w is called a self-matching.

For simplicity, in the rest of the paper, when we say a matching w we always mean a monotone matching.

Definition I.7. *(ε-self matching hash function) Let $p, q, n, n' \in \mathbb{N}$ be such that $n = n'p$. For any $0 < \varepsilon < 1$ and $x \in (\{0,1\}^p)^{n'}$, we say that a sequence of hash functions $h_1, h_2, \dots, h_{n'}$ where $\forall i \in [n'], h_i : \{0,1\}^p \to \{0,1\}^q$ is a sequence of ε-self matching hash functions with respect to x, if any matching between x and $y \in \{0,1\}^n$ under $h_1, h_2, \dots, h_n$, where y is the binary expression of x, has at most εn bad matches.*

The advantage of using ε-self matching hash functions is that the output range of the hash functions can be reduced. Specifically, we can show that a sequence of ε-self matching hash functions exists with output range $\mathsf{poly}(1/\varepsilon)$ (i.e., $O(\log(1/\varepsilon))$ bits) when the block size is at least $c \log(1/\varepsilon)$ bits for some constant $c > 1$. Furthermore, we can generate such a sequence of ε-self matching hash functions with high probability by again using an almost κ-wise independent sample space with $\kappa = O(kb_i)$, where b_i is the current block length, and error $\varepsilon = 1/\mathsf{poly}(n)$. The idea is that in a monotone matching with εn bad matches, we can divide the matching gradually into small intervals such that at least one small interval will have the same fraction of bad matches. Thus in order to ensure the ε-self matching property we just need to make sure every small interval does not have more than ε fraction of bad matches, and this is enough by using the almost κ-wise independent sample space.

As discussed above, we need to ensure that there are at most k bad matches in a self matching, thus we set $\varepsilon = \frac{k}{n}$. Consequently now the output of the hash functions only has $O(\log(n/k))$ bits instead of $O(\log n)$ bits. Now in each level, in order to get optimal sketch size, instead of using the Reed-Solomon code we will be using an Algebraic Geometric code [15] which has redundancy $O(k \log \frac{n}{k})$. The almost κ-wise independent sample space in this case again uses only $O(\log n)$ random bits, so in each level Alice can exhaustively search the correct hash functions in polynomial time and include the $O(\log n)$ bits of description in the sketch. This gives Alice's algorithm with total sketch size $O(k \log^2 \frac{n}{k})$. On Bob's side, we need another modification:

in each level after Bob recovers all the hash values, instead of simply searching for a match for every block, Bob runs a dynamic programming to find the longest monotone matching between his string y and the sequence of hash values. He then fills the blocks of $\tilde{x}$ using the corresponding substrings of matched blocks.

Error correcting codes.: Our deterministic document exchange protocol can be used to directly give an insdel code for k edit errors. The idea is that to encode an n-bit message x, we can first compute a sketch of x with size r, and then encode the small sketch using an insdel code against $4k$ edit errors. Since the sketch size is larger than k, we can use an asymptotically good code such as the one by Schulman and Zuckerman [22], which results in an encoding size of $n_0 = O(r)$. The actual encoding of the message is then the original message concatenated with the encoding of the sketch.

To decode, we can first obtain the sketch by looking at the last $n_0 - k$ bits of the received string. The edit distance between these bits and the encoding of the sketch is at most $4k$, and thus we can get the correct sketch from these bits. Now we look at the bits of the received string from the beginning to index $n + k$. The edit distance between these bits and x is at most $3k$, thus if r is a sketch for $3k$ edit errors then we will be able to recover x by using r. This gives our first insdel code with redundancy $O(k \log^2 \frac{n}{k})$.

We now describe our second insdel code, which has redundancy $O(k \log n)$ and uses many more interesting ideas. The basic idea here is again to compute a sketch of the message and encode the sketch, as we described above. However, we are not able to improve the sketch size of $O(k \log^2 \frac{n}{k})$ in general. Instead, our first observation here is that if the n-bit message is a *uniform random* string, then we can actually do better. Thus, we will first describe how to come up with a sketch of size $O(k \log n)$ for a uniform random string, and then use this to get an encoding for any given n-bit string.

To warm up, we first explain a simple algorithm to compute a sketch of size $O(k \log^2 n)$ in this case. A uniform random string has many nice properties. In particular, for some $B = O(\log n)$, one can show that with high probability, every length B substring in a uniform random string is *distinct*. If this property holds (which we refer to as the *B-distinct property*), then Alice can compute a sketch as follows. First create a vector of length $2^B = \mathsf{poly}(n)$, where in each entry indexed by the string $s \in \{0,1\}^B$, Alice looks at the substring s in x and record the bit left to it and the bit right to it. We need three special symbols, one to indicate the case of no left bit, one to indicate the case of no right bit, and one to indicate the case that there is no such string s in x. Thus it is enough to use an alphabet of size 8 for each entry. Similarly Bob can create a vector V' from his string y. Notice that the entries in V have no collisions since we assume that Alice's string is B-distinct, while some entries in V' may have collisions due to edit errors, in which case Bob just treats it as there is no corresponding substring in y. One can then show that V and V' differ in at most $O(kB) = O(k \log n)$ entries. Now Alice can use the Reed-Solomon code to send a redundancy of size $O(k \log^2 n)$, and Bob can recover the correct V. Bob can then recover the string x by picking an entry in V and growing the string on both ends gradually until obtaining the full string x.

We now show how to reduce the sketch size. In the above approach, V and V' can differ in $O(k \log n)$ entries since Alice is looking at *every* substring of length B in x. To improve this, instead we will have Alice first partition her string into several blocks, and then just look at the substrings corresponding to each block. Ideally, we want to make sure that each block has length at least B so that again all blocks are distinct. Alice then creates the vector V of length 2^B by using the B-prefix of each block as the index in V, and for each entry in V Alice will record some information. Bob will do the same thing using his string y to create another vector V', and we will argue that V and V' do not differ much so Alice can send some redundancy information to Bob, and Bob can recover the correct V based on V' and the redundancy information.

However, the partitions need to be done carefully. For example, we cannot just partition both strings sequentially into blocks of size some $T \geq B$, since if so then a single insertion/deletion at the beginning of x could result in the case where all blocks of x and y are distinct. Instead, we will choose a specific string p with length s, for some parameter s that we will choose appropriately. We call this string p a *pattern*, and we will use this pattern to divide the blocks in x and y. More specifically, in our construction we will simply choose $p = 1 \circ 0^{s-1}$, the string with a 1 followed by $s - 1$ 0's. We use p to divide the string x as follows. Whenever p appears as a substring in x, we call the index corresponding to the bit of p in x a *p-split point*. The set of split points then naturally gives a partition of the string x (and also y into blocks). We note that the idea of using patterns and split points is also used in [4]. However, there the construction uses $2k + 1$ patterns and takes a majority vote, which is why the sketch has a k^2 factor. Here instead we use a single pattern, and thus we can avoid the k^2 factor.

We now describe how to choose the pattern length s. For the blocks of x and y obtained by the split points, we certainly do not want the block size to be too large. This is because if there are large blocks then an adversary can create errors in these blocks, and large blocks need longer sketches to recover from error. At the same time, we do not want the block size to be too small either. This is because if the block size is too small, then some of the blocks may actually be the same, while we would like to keep all blocks of x to be distinct. In particular, we would like to keep the size of every block in x to be at least B. If x is a uniform random string, then the pattern p appears with probability

2^{-s} and thus the expected distance between two consecutive appearances of p is 2^s. Moreover, one can show that with high probability any interval of length some $O(s2^s \log n)$ contains a p-split point. We will call this *property* 1. If this property holds then we can argue that every block of x has length at most $O(s2^s \log n)$.

To ensure that each block is not too short, we simply look at a p-split point and the immediate next p-split point after it. If the distance between these two split points is less than $2^s/2$ then we just ignore the first p-split point. In other words, we change the process of dividing x into blocks so that we will only use a p-split point if the next p-split point is at least $2^s/2$ away from it, and we call such a p-split point a good p-split point. We again show that if x is uniform random, then with high probability every block of length some $O(2^s \log n)$ contains a good p-split point. We call this *property* 2. Combined with the previous paragraph, we can now argue that with high probability every interval of length some $O(s2^s \log n) + O(2^s \log n) = O(s2^s \log n)$ will contain a p-split point that we will choose. By setting $s = \log\log n + O(1)$, we can ensure that every block of x has length at least $B = O(\log n)$ and at most $O(s2^s \log n) = \mathsf{poly}\log(n)$.

Now for each block of x, Alice creates an entry in V indexed by the B-prefix of this block. The entry contains the length of this block and the B-prefix of the next block, which has total size $O(\log n)$. Similarly Bob also creates a vector V'. We show that our approach of choosing p-split points can ensure that V and V' differ in at most $O(k)$ entries, thus Alice can send a string with $O(k \log n)$ redundant bits to Bob (using the Reed-Solomon code) and Bob can recover the correct V. The structure of V guarantees that Bob can learn the correct order of the B-prefix of all Alice's blocks, and their lengths. At this point Bob will again try to fill a string $\tilde{x}$, where for each of Alice's block Bob searches for a substring with the same B-prefix and the same length. We show that after this step, at most $O(k)$ blocks in $\tilde{x}$ are incorrectly filled or missing. This completes state 1 of the sketch.

We now move to stage 2, where Bob correctly recovers the at most $O(k)$ blocks in $\tilde{x}$ that are incorrectly filled or missing. One way to do this is by noticing that every block has size at most $\mathsf{poly}\log(n)$, thus we can use a deterministic IMS protocol as we described before, which will last $O(\log\log n)$ levels and thus have sketch size $O(k \log \frac{n}{k} \log\log n)$ (since we start with block size $\mathsf{poly}\log(n)$). However, our goal is to do better and achieve sketch size $O(k \log n)$.

To achieve this, we modify the IMS protocol so that in stage 2, in each level we are not dividing every block into 2 smaller blocks. Instead, we divide every block evenly into $O(\log^{0.4} n)$ smaller blocks. We continue this process until the final block size is $O(\log n)$, and thus this only takes $O(1)$ levels. In each level, we will do something similar to our deterministic document exchange protocol: Alice sends a description of a sequence of hash functions to Bob, together with some redundancy of the hash values. Bob recovers the correct hash values and uses a dynamic programming to find the longest monotone matching between substrings of y and the hash values. Bob then tries to fill the blocks of $\tilde{x}$ by using the matched substrings.

In order for the above approach to work, we need to ensure three things. First, Alice's description of the hash functions should be short, ideally only $O(\log n)$ bits. Second, the redundancy of the hash values only uses $O(k \log n)$ bits. Finally, after Bob recovers the hash values, the matching he finds contains at most $O(k)$ unmatched blocks and mismatched blocks. For the second issue, we design the hash functions so that the outputs only have $O(\log^{0.5} n)$ bits. One issue here is that if in some level there are at most $O(k)$ unmatched blocks and mismatched blocks, then in the next level after dividing there may be $O(k \log^{0.4} n)$ unmatched blocks and mismatched blocks. However we observe that these blocks are actually concentrated (i.e., they are smaller blocks in a larger block of the previous level). Thus we can pack all the hash values of the $O(\log^{0.4} n)$ smaller blocks together into a package, and the total size of the hash values is $O(\log^{0.4} n) \cdot O(\log^{0.5} n) = O(\log n)$. Now we can show that again there are at most $O(k)$ different packages between Alice's version and Bob's version, thus it is still enough to use $O(k \log n)$ bits of redundancy.

The third issue and the first issue are actually related, and require new ideas. Specifically, we cannot use the ε-self matching hash functions as we discussed earlier, since that would require the outputs of the hash functions to have $O(\log(1/\varepsilon)) = O(\log(n/k))$ bits, while we can only afford $O(\log^{0.5} n)$ bits. To solve this problem, we strengthen the notion of ε-self matching hash functions and introduce *ε-synchronization hash functions*, similar in spirit to the notion of ε-synchronization strings introduced in [13]. Specifically, we have the following definition.

Definition I.8. *Let $T, n', B, R \in \mathbf{N}$ be such that $T \geq B$, and $0 < \varepsilon < 1$. Let x be a string of length $n = n'T$, and $x_T = (x[1,T], x[T+1,2T], \ldots, x[(n'-1)T+1,n])$ be a partition of x into blocks of size T. Let $\Phi = (\Phi[1], \Phi[2], \ldots, \Phi[n'])$ be a sequence of functions, where for any $k \in [n']$, $\Phi[k]$ is a function from $\{0,1\}^B$ to $\{0,1\}^R$.*

For a string y of length m, and some indices $0 \leq l_1 < r_1 \leq n'$, $0 \leq l_2 < r_2 \leq m$, let $\mathsf{MATCH}_\Phi(x_T(l_1,r_1], y(l_2,r_2])$ denote the size of the maximum matching between $x_T(l_1,r_1]$ and $y(l_2, \min(r_2+T, m+1))$ under $\Phi(l_1,r_1]$. We say Φ is a sequence of ε-synchronization hash functions with respect to x, if it satisfies the following properties:

- *For any three integers i, j, k where $i < Tj$ and $j < k$, denote $l_1 = k - j$ and $l_2 = Tj - i$.*

$$\mathsf{MATCH}_\Phi(x_T(j,k], x(i,Tj]) < \varepsilon \left(l_1 + \frac{l_2}{T} \right) \quad (1)$$

- *For any three integers i, j, k where $i < j$ and $k > T(j-1)+1$, denote $l_1 = j-i$ and $l_2 = k-T(j-1)-1$.*

$$\mathsf{MATCH}_\Phi(x_T(i,j], x(T(j-1)+1,k]) < \varepsilon \left(l_1 + \frac{l_2}{T}\right) \tag{2}$$

It can be seen that ε-synchronization hash functions are indeed stronger than ε-self matching hash functions, and we show that even a sequence of ε-synchronization hash functions with $\varepsilon = \Omega(1)$ is enough to guarantee that there are at most $O(k)$ unmatched blocks and mismatched blocks (in fact, the number of unmatched blocks and mismatched blocks is bounded by $\frac{1+2\varepsilon}{1-2\varepsilon}k$). A similar property for ε-synchronization strings is also shown in [13].

Our construction of the ε-synchronization hash functions actually consists of two parts. In the first part, Alice generates a sequence of hash functions that ensures the ε-synchronization property holds for relatively large intervals (i.e., when $l_1 + \frac{l_2}{T}$ is large). We show that this sequence of hash functions can again be generated by using an almost κ-wise independent sample space which uses $O(\log n)$ random bits (the argument is similar to that of ε-self matching hash functions). Thus Alice can exhaustively search for a fixed set of hash functions in polynomial time, and send Bob the description using $O(\log n)$ bits. The output of these hash functions has $O(\log^{0.5} n)$ bits. To protect the small intervals, we compute a hash function for each block such that when the inputs are restricted to length B substrings in a small interval, this function is *injective*. Since all substrings of length B are distinct in x, this ensures that the outputs of the same hash function within a small interval are also distinct, and thus the ε-synchronization property also holds for small intervals. The output of these hash functions has $O(\log\log n)$ bits. We show that these functions can be constructed deterministically using an almost κ-wise independent sample space which uses $O(\log\log n)$ random bits, and hence also has description size $O(\log\log n)$. Moreover the set of functions computed by Alice and the set of functions computed by Bob have at most $O(k)$ differences (after packing each successive $\log^{0.6} n$ such functions together, which only needs $O(\log^{0.6} n \log\log n)$ bits), and thus Alice can again send $O(k \log n)$ redundant bits to Bob and Bob can recover the correct set of hash functions. The final sequence of ε-synchronization hash functions is then the combination of these two sets of functions, where the output has $O(\log^{0.5} n + O(\log\log n)) = O(\log^{0.5} n)$ bits. This concludes our algorithm to generate a sketch of size $O(k \log n)$ for a uniform random string.

We now turn to solve the issue of assuming a uniform random string x. Put simply, our idea is that given any arbitrary string x, we first compute the XOR of x and a *pseudorandom* string z (a mask), to turn x into a pseudorandom string as well. We will use $O(\log n)$ random bits to generate z and show that with high probability over the random bits used, the XOR of x and z satisfies the B-distinct property, as well as property 1 and property 2. For this purpose, we construct three pseudorandom generators (PRGs), one for each property. The B-distinct property can be ensured by again using an almost κ-wise independent sample space which uses $O(\log n)$ random bits. For property 1, we divide the string of length n into blocks of length $T = O(s2^s \log n)$ and generate the same mask for every block. Within each block, we can view it as a sequence of $t = O(\log n)$ sub-blocks each of length $s2^s$. For each sub-block, the test of whether it contains the pattern p can be realized as a DNF with size $\mathsf{poly}\log n$, so we can use a PRG for DNF to fool this test with constant error, and this has length $\mathsf{poly}\log\log(n)$. We then use a random walk on a constant-degree expander graph of length t to generate the full mask, which guarantees that the pattern occurs with probability $1 - 1/\mathsf{poly}(n)$. A union bound now shows that property 1 holds with high probability, and the PRG has seed length $O(\log n)$. For property 2, we can essentially do the same thing, i.e., divide the string of length n into blocks of length $T = O(2^s \log n)$ and generate the same mask for every block. For each block, again we use a PRG for DNF together with a random walk on a constant-degree expander graph to get a PRG with seed length $O(\log n)$. Finally we take the XOR of the outputs of the three PRGs, and we show that with high probability all three properties hold for $x \oplus z$.

Since the PRG has seed length $O(\log n)$, again we can exhaustively search for a fixed z that works for the string x, and z can be described by $O(\log n)$ bits. The encoding of x is then $x \oplus z$, together with an encoding of the concatenation of the description of z and the sketch. For decoding, one can first recover $x \oplus z$ and then recover x by using the description of z and the PRG.

II. Preliminaries

A. Notation

Let Σ be an alphabet (which can also be a set of strings). For a string $x \in \Sigma^*$,

1) $|x|$ denotes the length of the string.
2) $x[i,j]$ denotes the substring of x from position i to position j (Both ends included).
3) $x[i]$ denotes the i-th symbol of x.
4) $x \circ x'$ denotes the concatenation of x and some other string $x' \in \Sigma^*$.
5) B-prefix denotes the first B symbols of x. (Usually used when $\Sigma = \{0,1\}$.)
6) x^N the concatenation of N number of string x.

We use U_n to denote the uniform distribution on $\{0,1\}^n$.

B. Edit distance and longest common subsequence

Definition II.1 (Edit distance)**.** *For any two strings $x, x' \in \Sigma^n$, the edit distance $ED(x, x')$ is the minimum number* *of edit operations (insertions and deletions) required to transform x into x'.*

Definition II.2 (Longest Common Subsequence)**.** *For any strings x, x' over Σ, the longest common subsequence of x and x' is the longest pair of subsequences of x and x' that are equal as strings. $LCS(x, x')$ denotes the length of the longest common subsequence between x and x'.*

Note that $ED(x, x') = |x| + |x'| - 2LCS(x, x')$.

C. Almost k-wise independence

Definition II.3 (ε-almost κ-wise independence in max norm [1])**.** *Random variables $X_1, X_2, \ldots, X_n \in \{0,1\}^n$ are ε-almost κ-wise independent in max norm if $\forall i_1, i_2, \ldots, i_\kappa \in [n]$, $\forall x \in \{0,1\}^\kappa$, $|\Pr[X_{i_1} \circ X_{i_2} \circ \cdots \circ X_{i_\kappa} = x] - 2^{-\kappa}| \le \varepsilon$.*

A function $g : \{0,1\}^d \to \{0,1\}^n$ is an ε-almost κ-wise independence generator in max norm if $g(U) = Y = Y_\kappa \circ \cdots Y_n$ are ε-almost κ-wise independent in max norm.

In the following passage, unless specified, when we say ε-almost κ-wise independence, we mean in max norm.

Theorem II.4 (ε-almost κ-wise independence generator [1])**.** *There exists an explicit construction s.t. for every $n, \kappa \in \mathbb{N}$, $\varepsilon > 0$, it computes an ε-almost κ-wise independence generator $g : \{0,1\}^d \to \{0,1\}^n$, where $d = O(\log \frac{\kappa \log n}{\varepsilon})$.*

The construction is highly explicit in the sense that, $\forall i \in [n]$, the i-th output bit can be computed in time $\mathsf{poly}(\kappa, \log n, \frac{1}{\varepsilon})$ given the seed and i.

Theorem II.5 (PRG for CNF/DNFs [9])**.** *There exists an explicit PRG g s.t. for every $n, m \in \mathbb{N}, \varepsilon > 0$, every CNF/DNF f with n variables, m terms,*

$$|\Pr[f(U_n) = 1] - \Pr[f(g(n, m, \varepsilon, U_r)) = 1]| \le \varepsilon$$

where $|g(n, m, \varepsilon, U_r)| = n$, $r = O(\log n + \log^2(m/\varepsilon) \cdot \log\log(m/\varepsilon))$.

D. Error correcting codes (ECC)

An (n, m, d)-code C is an ECC (for hamming errors) with codeword length n, message length m. The hamming distance between every pair of codewords in C is at least d.

Next we recall the definition of ECC for edit errors.

Definition II.6. *An ECC $C \subseteq \{0,1\}^n$ for edit errors with message length m and codeword length n consists of an encoding mapping $Enc : \{0,1\}^m \to \{0,1\}^n$ and a decoding mapping $Dec : \{0,1\}^* \to \{0,1\}^m \cup \{Fail\}$. The code can correct k edit errors if for every y, s. t. $ED(y, Enc(x)) \le k$, we have $Dec(y) = x$. The rate of the code is defined as $\frac{m}{n}$.*

An ECC family is explicit (or has an explicit construction) if both encoding and decoding can be done in polynomial time.

We will utilize linear algebraic geometry codes to compute the redundancy of the hamming error case.

Theorem II.7 ([15])**.** *There exists an explicit algebraic geometry ECC family $\{(n, m, d)_q\text{-code } C \mid n, m \in \mathbb{N}, m \le n, d = n - m - O(1), q = \mathsf{poly}(\frac{n}{d})\}$ with polynomial-time decoding when the number of errors is less than half of the distance.*

Moreover, $\forall n, m \in \mathbb{N}$, for every message $x \in \mathbb{F}_q^m$, the codeword is $x \circ z$ for some redundancy $z \in \mathbb{F}_q^{n-m}$.

To construct ECC from document exchange protocol, we need to use a previous result about asymptotically good binary ECC for edit errors given by Schulman and Zuckerman [22].

Theorem II.8 ([22])**.** *There exists an explicit binary ECC family in which a code with codeword length n, message length $m = \Omega(n)$, can correct up to $k = \Omega(n)$ edit errors.*

III. Deterministic protocol for document exchange

We derandomize the IMS protocol given by Irmak et. al. [16], by first constructing ε-self-matching hash functions and then use them to give a deterministic protocol.

Theorem III.1. *There exists an algorithm which, on input $n, p, q \in \mathbb{N}$, $\varepsilon \in (0,1)$, $n \ge p \ge q$, $q = \Theta(\log \frac{1}{\varepsilon})$, $x \in \{0,1\}^n$, outputs a description of ε-self-matching functions $h_1, \ldots, h_{n'} : \{0,1\}^p \to \{0,1\}^q$, in time $\mathsf{poly}(n)$, where the description length is $O(\log n)$ and $n' = \frac{n}{p}$.*

Also there is an algorithm which, given the same $n, p, q \in \mathbb{N}$, the description of $h_1, \ldots, h_{n'}$, $i \in [n']$ and any $a \in \{0,1\}^p$, can output $h_i(a)$ in time $\mathsf{poly}(n)$.

Our deterministic protocol for document exchange is as follows.

Construction III.2. *The protocol is for every input length $n \in \mathbb{N}$, every $k \le \alpha n$ number of edit errors where α is a constant. For the case $k > \alpha n$, we simply let Alice send her input string.*

Both Alice's and Bob's algorithms have $L = O(\log \frac{n}{k})$ levels.

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

1) *We set up the following parameters;*
   - *For every $i \in [L]$, in the i-th level,*
     - *The block size is $b_i = \frac{n}{3 \cdot 2^i k}$, i.e., in each level we divide a block in the previous level evenly into two blocks. We choose L properly s.t. $b_L = O(\log \frac{n}{k})$;*
     - *The number of blocks $l_i = n/b_i$;*
2) *For the i-th level,*
   a) *Divide x into consecutive blocks to get $x' \in (\{0,1\}^{b_i})^{l_i}$;*
   b) *Construct a sequence of $\varepsilon = \frac{k}{n}$-self-matching hash functions $h_1, \ldots, h_{l_i} : \{0,1\}^{b_i} \to \{0,1\}^{b^*}$ for x by Theorem III.1, with $b^* = O(\log \frac{n}{k})$.*

Let the description of the hash functions be $u[i] \in \{0,1\}^{O(\log n)}$ by Theorem III.1;

   c) *Compute $v[i] = (h_1(x'[1]), h_2(x'[2]), \dots, h_{l_i}(x'[l_i]))$;*
   d) *Compute the redundancy $z[i] \in (\{0,1\}^{b^*})^{\Theta(k)}$ for $v[i]$ by Theorem II.7, where the code has distance $14k$;*

3) *Compute the redundancy $z_{\mathsf{final}} \in (\{0,1\}^{b_L})^{\Theta(k)}$ for the blocks of the L-th level by Theorem II.7, where the code has distance $8k$;*
4) *Send $u = (u[1], u[2], \dots, u[L])$, $z = (z[1], z[2], \dots, z[L])$, $v[1]$, z_{final}.*

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

1) *Create $\tilde{x} \in \{0,1,*\}^n$ (i.e. his current version of Alice's x), initiating it to be $\{*, *, \dots, *\}$;*
2) *For the i-th level where $1 \leq i \leq L-1$,*
   a) *Divide $\tilde{x}$ into consecutive blocks to get $\tilde{x}' \in (\{0,1\}^{b_i})^{l_i}$;*
   b) *Apply the decoding of Theorem II.7 on $h_1(\tilde{x}'[1]) \circ h_2(\tilde{x}'[2]) \circ \dots \circ h_{l_i}(\tilde{x}'[l_i]) \circ z_i$ to get the sequence of hash values $v[i] = (h_1(x[1]), h_2(x[2]), \dots, h_{l_i}(x[l_i]))$. Note that $v[1]$ is received directly, thus Bob does not need to compute it;*
   c) *Compute $w = ((\rho'_1, \rho_1), \dots, (\rho'_{|w|}, \rho_{|w|})) \in ([l_i] \times [|y|])^{|w|}$ which is the maximum matching between x' and y under $h_1, \dots, h_{l_i}$, using $v[i]$;*
   d) *Evaluate $\tilde{x}$ according to the matching, i.e. let $\tilde{x}'[\rho'_j] = y[\rho_j, \rho_j + b_i - 1]$;*
3) *In the L'th level, apply the decoding of Theorem II.7 on the blocks of $\tilde{x}$ and z_{final} to get x;*
4) *Return x.*

IV. Document exchange for a uniform random string

In this section we prove the following theorem.

Theorem IV.1. *There exists a deterministic document exchange protocol for a uniformly random string with success probability $1 - 1/\mathsf{poly}(n)$ and redundancy size $O(k \log n)$.*

A. String properties

Definition IV.2 (p-split point[4])**.** *For a string $p \in \{0,1\}^s$ and $x \in \{0,1\}^n$, a p-split point of x is an index $1 \leq i \leq n - s + 1$ such that $x[i, i+s) = p$.*

Definition IV.3 (next p-split point)**.** *Let p and x be two strings, and i be a p-split point of x. Define the* next p-split point *of i to be the smallest j such that j is a p-split point and $j > i$. If such j does not exist, we define the* next p-split point *of i to be $(n+1)$.*

We will use the following properties of a uniform random string.

Theorem IV.4. *For a uniform random string $x \in \{0,1\}^n$, let $p = 1 \circ 0^{s-1}$ be a string of length s. There exist three integers $B_1 = O(s2^s \log n)$, $B_2 = O(2^s \log n)$ and $B = O(\log n)$ such that the following properties hold with probability $1 - 1/\mathsf{poly}(n)$.*

- *(Property 1) Any interval of x with length B_1 contains a p-split point.*
- *(Property 2) Any interval of x with length B_2 starting at a p-spit point contains a p-split point i such that its next p-split point j satisfies $j - i > 2^s/2$.*
- *(B-distinct) Every two substrings of length B at different positions of x are distinct.*

B. Construction

In the first stage, the two parties use a fixed small string $p = 1 \circ 0^{s-1}$ of length s, and find all p-split points in their strings. As the string x is uniform random, with high probability, the distance between any two adjacent p-split points is $O(s2^s \log n)$. But some p-split points may be too close to each other. So we only choose the p-split points i such that the next p-split point of i is at least $2^s/2$ away, and use these chosen p-split points to partition the string into blocks.

Construction IV.5 (Stage I). *Let n denote the input length. Parameters $s = \log\log n + 3$, $B = 3\log n$, $T_0 = s2^s \log n$, and $p = 1 \circ 0^{s-1}$ is a fixed string of length s. To make the representation simple, we assume n is a multiple of T_0.*

Alice: On input uniform random string $x \in \{0,1\}^n$.

1) *Choose all p-split points i of x such that its next p-split point j satisfies $j - i > 2^s/2$. Denote the chosen p-split points as $i_1, i_2, \dots, i_{n'}$. Partition the string x into blocks $x[1, i_2)$, $x[i_2, i_3)$, $x[i_3, i_4)$ $\dots, x[i_{n'}, n]$, and index these blocks as $1, 2, 3, \dots, n'$.*
2) *Create a set $V = \{(\mathsf{len}_b, \mathsf{B\text{-}prefix}_b, \mathsf{B\text{-}prefix}_{b+1}) \mid 1 \leq b \leq n' - 1\}$, where len_b is the length of the b-th block, and $\mathsf{B\text{-}prefix}_b$ and $\mathsf{B\text{-}prefix}_{b+1}$ are the B-prefix of the b-th block and the $(b+1)$-th block respectively.*
3) *Represent the set V as its indicator vector, which has size $\mathsf{poly}(n)$, and send the redundancy z_V being able to correct $4k$ Hamming errors, using Theorem II.7 (or simply using a Reed-Solomon code).*
4) *Partition the string x evenly into n/T_0 blocks, each of size T_0.*

Bob: On input string $y \in \{0,1\}^m$ satisfying $ED(x,y) \leq k$, and the redundancy z_V sent by Alice.

1) *Choose the p-split points i of y such that its next p-split point j satisfies $j - i > 2^s/2$. Denote the chosen p-split points as $i'_1, i'_2, \dots, i'_{m'}$. Partition the string y into blocks $y[1, i'_2)$, $y[i'_2, i'_3)$, $y[i'_3, i'_4)$ $\dots, y[i'_{m'}, n]$, and index these blocks as $1, 2, 3, \dots, m'$.*
2) *Create a set $V' = \{(\mathsf{len}_b, \mathsf{B\text{-}prefix}_b, \mathsf{B\text{-}prefix}_{b+1}) \mid 1 \leq b \leq m' - 1\}$*

using the partition of y.

3) *Use the indicator vector of V' and the redundancy z_V to recover Alice's set V.*
4) *Create an empty string $\tilde{x}$ of length n, and partition $\tilde{x}$ according to the set V in the following way: first find the element $(\textsf{len}^{(1)}, \textsf{B-prefix}^{(1)}, \textsf{B-prefix}'^{(1)})$ in V such that for all elements $(\textsf{len}, \textsf{B-prefix}, \textsf{B-prefix}')$ in V, $\textsf{B-prefix}^{(1)} \neq \textsf{B-prefix}'$. Then partition $\tilde{x}[1, \textsf{len}^{(1)}]$ as the first block, and fill $\tilde{x}[1, B]$ with $\textsf{B-prefix}^{(1)}$. Then find the element $(\textsf{len}^{(2)}, \textsf{B-prefix}^{(2)}, \textsf{B-prefix}'^{(2)})$ such that $\textsf{B-prefix}^{(2)} = \textsf{B-prefix}'^{(1)}$, and partition $\tilde{x}[\textsf{len}^{(1)} + 1, \textsf{len}^{(1)} + \textsf{len}^{(2)}]$ as the second block, and fill $\tilde{x}[\textsf{len}^{(1)} + 1, \textsf{len}^{(1)} + B]$ with $\textsf{B-prefix}^{(2)}$. Continue doing this until all elements in V are used to recover the partition of x.*
5) *For each block b in $\tilde{x}$, if Bob finds a unique block b' in y such that the B-prefix of b' matches the B-prefix of b and the lengths of b and b' are equal, Bob fills the block b using b'. If such b' doesn't exist or Bob has multiple choices of b', then Bob just leaves the block b as blank.*
6) *Partition the string $\tilde{x}$ evenly into n/T_0 blocks, each of size T_0.*

Construction IV.6 (Stage II). *The second stage consists of $L = \left\lceil \frac{\log(O(s2^s))}{\log(\log^{0.4} n)} \right\rceil = O(1)$ levels. Let $T' = \log^{0.6} n, T'' = \log^{0.4} n$, $T_l = T_{l-1}/T''$ for $1 \leq l \leq L-1$. In the last level, we choose $T_L = B$ and $T_L \geq T_{L-1}/\log^{0.4} n$. To make the representation simple, in this stage we assume n is a multiple of T_l, for all $l \in [L]$.*

Alice: For $l = 1, 2, \dots L$, in l-th level,

1) *Partition the string x evenly into $n'_l = n/T_l$ blocks, each of size T_l.*
2) *Let block size $T = T_l$, Alice gets a sequence of ε-synchronization hash functions $\Phi = (\Phi[1], \Phi[2], \dots, \Phi[n'_l])$ with respect to x, where each $\Phi[t], t \in [n'_l]$ consists of a pair of functions $(\phi[t], \theta[t])$.*
3) *Alice sends the description of $(\phi[t])_{t \in [n'_l]}$ to Bob. The description uses $O(\log n)$ bits.*
4) *Alice packs every successive T' elements of $(\theta[t])_{t \in [n'_l]}$ into a vector V_θ, i.e. $V_\theta = (\theta[1, T'], \theta[T'+1, 2T'], \dots)$. Each $\theta[t]$ has a description of size $O(\log\log n)$. Alice sends the redundancy z_θ being able to correct some $O(k)$ Hamming errors of V_θ, using Theorem II.7.*
5) *For any $t \in [n'_l]$, Alice evaluates $\Phi[t]$ on the t-th block of x, and obtains the hash values $I[t] = \Phi[t](x[T_l(t-1)+1, T_l(t-1)+B])$, and stores $(I[t])_{t \in [n'_l]}$ into a vector I. Then she packs every successive T'' elements of I into a vector V_I, i.e. $V_I = (I[1, T''], I[T''+1, 2T''], \dots)$, and sends the redundancy z_I being able to correct some $O(k)$ Hamming errors of V_I, using Theorem II.7.*

After L levels, Alice evenly partitions her string x into n/T_L small blocks, each of size T_L. Alice then sends a redundancy z_x being able to correct some $O(k)$ wrong blocks or unmatched blocks, using Theorem II.7.

Bob: For $l = 1, 2, \dots, L$, in the l-th level, Bob receives the description of the functions $(\phi[t])_{t \in [n'_l]}$, and the redundancies z_θ, z_I. Finally he receives z_x.

1) *Partition the string $\tilde{x}$ evenly into $n'_l = n/T_l$ blocks, each of size T_l.*
2) *For any $t \in [n'_l]$, denote $S_t = \{\tilde{x}[u, u+B) \mid |T(t-1)+1-u| < T_l \log^{0.6} n,\ 1 \leq u \leq n-B+1\}$. Bob obtains $(\theta'[t])_{t \in [n'_l]}$, using S_t for any $t \in [n'_l]$.*
3) *Bob packs every successive T' elements of $(\theta'[t])_{t \in [n'_l]}$ into a vector $V'_\theta = (\theta'[1, T'], \theta'[T'+1, 2T'], \dots)$. Then he uses the redundancy z_θ and V'_θ to recover V_θ. Bob unpacks $(\theta[t])_{t \in [n'_l]}$ from V_θ to obtain Φ.*
4) *For any $t \in [n'_l]$, Bob evaluates the hash function $\Phi[t]$ on the t-th block of $\tilde{x}$, so he obtains the hash values $I'[t] = \Phi[t](\tilde{x}[T_l(t-1)+1, T_l(t-1)+B])$. Bob packs every successive T'' elements of I' into the a vector V'_I, i.e. $V'_I = (I'[1, T''], I'[T''+1, 2T''], \dots)$.*
5) *Bob uses the redundancy z_I and V'_I to recover V_I, then obtains I from V_I by unpacking.*
6) *Bob finds the maximum matching Π between x and y under Φ using I. Note that in order to find such a matching, Bob only needs to know the hash values of $\Phi[t]$ on the t-th block of x, which can be obtained from the vector I. For each pair (a, b) in Π, Bob fills the a-th block $\tilde{x}[T_l(a-1)+1, T_l a]$ with $y[b, b+T_l)$.*

After L levels, Bob partitions $\tilde{x}$ evenly into blocks of length T_L, then uses the redundancy z_x to recover x.

V. Explicit binary ECC for edit errors

In this section we'll show how to use the document exchange protocol for uniform random strings in Section IV to construct ECC for edit errors.

Our general strategy is as follows. For any given message $x \in \{0,1\}^n$, we use a generator to generate a mask string s.t. the xor of the mask and the message, say $y(U_r) \in \{0,1\}^n$, has the three properties: IV.4, IV.4, IV.4. We ensure that the seed length for the generator is small enough s.t. we can exhaustively search the seed $u \in \{0,1\}^r$ s.t. $y(u)$ has the three properties. Then we apply the method in Section IV to create a redundancy z for $y(u)$. After that we use an asymptotically good binary ECC for edit errors to encode the redundancy and the seed. Concatenating this with y gives the final codeword.

A. The generator for the mask

We show that there exists an explicit generator of seed length $O(\log n)$ s.t. given any message $x \in \{0,1\}^n$, w.h.p.

the xor of x and the output of the generator has IV.4, IV.4 and IV.4, where the randomness is over the uniform random seed of the generator. Formally we have the following theorem.

Theorem V.1. *There exists an algorithm (generator) g s.t. for every $n \in \mathbb{N}, x \in \{0,1\}^n$, with probability $1-1/\mathsf{poly}(n)$, $g(n, U_r)+x$ satisfies IV.4, IV.4 and IV.4, where $r = O(\log n)$. (Let s in IV.4, IV.4 be $\log\log n + O(1)$.)*

g is the xor of three generators, each of which generates a string satisfying one of the three properties. We utilize random walks on expander graphs and PRG for AC^0 circuits to reduce the seed length of generators for IV.4 and IV.4. And we use almost κ-wise independence generator for IV.4.

B. Binary insdel codes with almost optimal parameters

We can get the following results.

Theorem V.2. *For any $n, k \in \mathsf{N}$ with $k \leq n/4$, there exists an explicit binary error correcting code with message length n, codeword length $n + O(k \log^2 \frac{n}{k})$ that can correct up to k edit errors.*

Corollary V.3. *There exists a constant $0 < \alpha < 1$ such that for any $0 < \varepsilon \leq \alpha$ there exists an explicit family of binary error correcting codes with codeword length n and message length m, that can correct up to $k = \varepsilon n$ edit errors with rate $m/n = 1 - O(\varepsilon \log^2 \frac{1}{\varepsilon})$.*

Theorem V.4. *For any $n, k \in \mathsf{N}$, there exists an explicit binary error correcting code with message length n, codeword length $n + O(k \log n)$ that can correct up to k edit errors.*

References

[1] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple constructions of almost k-wise independent random variables. *Random Structures & Algorithms*, 1992.

[2] Djamal Belazzougui. Efficient deterministic single round document exchange for edit distance. *CoRR*, abs/1511.09229, 2015.

[3] Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In *FOCS*. IEEE, 2016.

[4] 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.

[5] Boris Bukh and Venkatesan Guruswami. An improved bound on the fraction of correctable deletions. In *SODA*, 2016.

[6] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Low distortion embedding from edit to hamming distance using coupling. In *STOC*, 2016.

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

[8] Graham Cormode, Mike Paterson, Suleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In *SODA*, 2000.

[9] Anindya De, Omid Etesami, Luca Trevisan, and Madhur Tulsiani. Improved pseudorandom generators for depth 2 circuits. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 504–517. Springer, 2010.

[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)*, 2016.

[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.

[12] Bernhard Haeupler. Optimal document exchange and new codes for small number of insertions and deletions. *arXiv preprint arXiv:1804.03604*, 2018.

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

[16] Utku Irmak, Svilen Mihaylov, and Torsten Suel. Improved single-round protocols for remote file synchronization. In *INFOCOM*, volume 3. IEEE, 2005.

[17] Hossein Jowhari. Efficient communication protocols for deciding edit distance. In *ESA*, 2012.

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

[19] 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.

[20] A. Orlitsky. Interactive communication: balanced distributions, correlated files, and average-case complexity. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, 1991.

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

[22] L. J. Schulman and D. Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory*, 45(7):2552–2557, 1999.