Multilayer Codes for Synchronization from Deletions

Mahed Abroshan
University of Cambridge
ma675@cam.ac.uk

Ramji Venkataramanan
University of Cambridge
ramji.v@eng.cam.ac.uk

Albert Guillén i Fàbregas
ICREA & Universitat Pompeu Fabra
University of Cambridge
guillen@ieee.org

***Abstract*—A coding scheme is proposed for synchronization from a small number of deletions via a one-way error-free link. The scheme is based on multiple layers of Varshamov-Tenengolts codes combined with off-the-shelf linear error-correcting codes.**

I. Introduction

Consider two remote nodes having binary sequences X and Y, respectively, where Y is an *edited* version of X. In this paper, we consider the edits to be deletions. Let the length of X be n bits, and the number of deletions be k. Thus, Y is a sequence of length $m = (n - k)$, obtained by deleting k bits from X. In the synchronization model shown in Fig. 1, the node with X (the "encoder") sends a message M via an error-free link to the other node (the "decoder"), which attempts to reconstruct X using M and Y. The goal is to design a scheme so that the decoder can reconstruct X with minimal communication, i.e., we want to minimize the number of bits used to represent the message M.

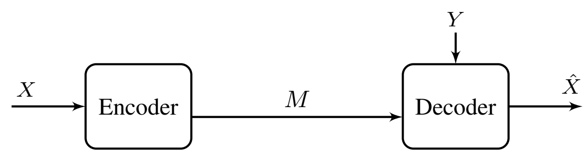

Fig. 1: Synchronization Model

The deletion model considered here is a simplified version of the general file synchronization problem where the edits can be a combination of deletions, insertions, and substitutions. The general synchronization problem has a number of applications including file backup (e.g., Dropbox) and file sharing. Various forms of the synchronization model have been studied in previous works; see, e.g., [1]–[4]. A number of these works allow *two-way* interaction between the encoder and decoder.

In contrast, we seek codes for *one-way* synchronization: the message M is produced by the encoder using only X, with no knowledge of Y except its length m. We assume that the decoder knows n, so it can infer the number of deletions $k = (n - m)$. The message M belongs to a finite set $\mathcal{M}$ with cardinality $|\mathcal{M}|$. The synchronization rate is defined as $R = \frac{\log_2 |\mathcal{M}|}{n}$. We would like to design a code for reliable synchronization with R as small as possible; $R = 1$ is equivalent to the encoder sending the entire string X.

In this paper, we construct a code for synchronization from deletions when the number of deletions k is small compared to n. The output of the decoder is a small list of sequences that is guaranteed to contain the correct sequence X. Though we do not provide theoretical bounds on the list size, we observe from simulations that with a careful choice of code parameters, the list size rarely exceeds 2 or 3; for reasonably large n, the list size can be made 1, i.e., X is exactly reconstructed. For example, we construct a code of length $n = 378$ that can synchronize from $k = 7$ deletions with $R = 0.365$, and a length $n = 2800$ code which can synchronize from $k = 10$ deletions with $R = 0.135$. (Details in Section IV.)

Overview of code construction: The starting point for our code construction is the family of Varshamov-Tenengolts (VT) codes [5], [6]. Each VT code is a single deletion correcting code. As observed in [7], the VT family gives an elegant way to exactly synchronize from a single deletion: the encoder simply sends the VT syndrome of the sequence X. The VT syndrome, defined in the next section, indicates which VT code X belongs to. The decoder then uses the single deletion correcting property of the VT code to recover the deleted bit.

In our model, the code needs to synchronize from $k > 1$ deletions. The encoder sends the VT syndromes of various substrings of X to the decoder. The length n sequence X is divided into smaller chunks of n_c bits each. The encoder then computes VT syndromes for two kinds of substrings: *blocks* which are composed of adjacent chunks, and *chunk-strings* which are composed of well-separated chunks. Fig. 2 shows an example where X of length 12 is divided into 4 length-3 chunks. The blocks B_1 and B_2 are each formed by combining two adjacent chunks, while the chunk-strings C_1 and C_2 are each formed by combining two alternate chunks. In this case, the encoder sends the VT syndromes of B_1, B_2, C_1, and C_2.

The intersecting VT constraints of the blocks and the chunk-strings help the decoder to iteratively determine the approximate locations of the edits. The VT syndromes serve a dual purpose: i) they can be used to recover deleted bits in blocks or chunk-strings inferred to have a single deletion; this recovery may result in new blocks and chunk-strings with a single deletion; ii) the VT syndromes also act as checks that eliminate a large number of deletion patterns, allowing the decoder to localize the deletions to a relatively small set of chunks. The final ingredient of the message is a parity check syndrome of X using a linear code. This is used to recover the deletions in chunks that still remain uncertain at the decoder after processing the intersecting VT constraints.

This work has been funded in part by the European Research Council under ERC grant agreement 259663 and by the Spanish Ministry of Economy and Competitiveness under grant TEC2016-78434-C3-1-R.

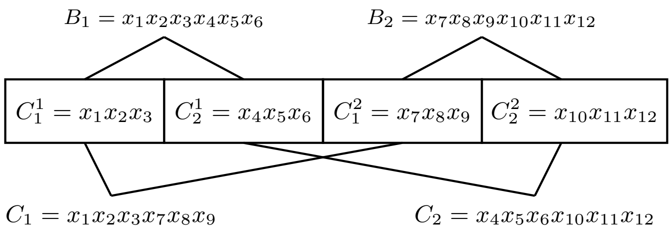

Fig. 2: Blocks and chunk-strings structure for the example where $l_1 = l_2 = 2$

We refer to this code construction as a two-layer code as the chunks are combined to form two kinds of intersecting substrings. The construction can be generalized to combine chunks in multiple ways to form many layers of intersecting substrings. Increasing the number of constraints in the code improves its synchronization capability at the cost of increasing the rate and decoding complexity.

The problem of one-way synchronization from k deletions is closely related to the problem of communicating over a deletion channel that deletes k bits from a length n codeword [8]. The channel coding version of the proposed code construction will be discussed in an extended version of this paper.

Notation: We denote scalars using lower-case letters and sequences using capital letters. The subsequence of X from index i to index j is denoted by $X(i:j) = x_i x_{i+1} \cdots x_j$. Matrices are denoted by bold capitals. We use brackets for merging sequences, so $X = [X_1, \cdots, X_u]$ is a super-sequence defined by concatenating the sequences $X_1, \cdots, X_u$.

II. Code Construction and Encoding

We begin with a review of VT codes. The VT syndrome of a binary sequence $W = (w_1, \ldots, w_n)$ is defined as

$$\mathsf{syn}(W) = \sum_{j=1}^{n} j\, w_j \pmod{(n+1)}. \tag{1}$$

For positive integers n and $0 \le s \le n$, we define the VT code of length n and syndrome s, denoted by

$$\mathcal{VT}_s(n) = \{W \in \{0,1\}^n : \mathsf{syn}(W) = s\}, \tag{2}$$

as the set of sequences W of length n for which $\mathsf{syn}(W) = s$.

The $n+1$ sets $\mathcal{VT}_s(n) \subset \{0,1\}^n$, for $0 \le s \le n$, partition the set of all sequences of length n. Each of these sets $\mathcal{VT}_s(n)$ is a single-deletion correcting code. The complexity of the VT decoding algorithm is linear in the code length n [9].

Constructing the message: The message M generated by the encoder consists of three parts, denoted by M_1, M_2, and M_3. The first part comprises the VT syndromes of the blocks, the second part comprises the VT syndromes of the chunk-strings, and the third part is the parity check syndrome of X with respect to a linear code.

The first step is to divide $X = x_1 x_2 \cdots x_n$ into l_1 equal-sized blocks (assume that n is divisible by l_1). The length of each block is denoted by $n_b = \frac{n}{l_1}$. For $1 \le i \le l_1$, the ith block is denoted by $B_i = X((i-1)n_b + 1 : i n_b)$, and its VT syndrome is $s_{B_i} = \mathsf{syn}(B_i)$. The first part of the message is the collection of VT syndromes for the l_1 blocks, i.e., $M_1 = \{s_{B_1}, s_{B_2}, \cdots, s_{B_{l_1}}\}$. Since each s_{B_i} is an integer between 0 and n_b, the number of bits required to represent the VT syndromes of the l_1 blocks is $l_1 \lceil \log(n_b + 1) \rceil$.

For the second part of the message, we divide each block into l_2 chunks, each of size n_c bits. We assume that $\frac{n}{l_1}$ is divisible by l_2; the length of X is $n = n_c l_1 l_2$. For $1 \le j \le l_2$, the jth chunk within the ith block is denoted by

$$C_j^i = X((i-1)n_b + (j-1)n_c + 1 : (i-1)n_b + j n_c). \tag{3}$$

The jth *chunk-string* is then formed by concatenating the jth chunk from each of the l_1 blocks. That is, the jth chunk string $C_j = [C_j^1, C_j^2, \cdots, C_j^{l_1}]$, for $1 \le j \le l_2$. Fig. 2 shows the blocks and the chunk-strings in an example where X of length $n = 12$ is divided into $l_1 = 2$ blocks, each of which is divided into $l_2 = 2$ chunks of $n_c = 3$ bits.

The second part of the message is the collection of VT syndromes for the l_2 chunk-strings, i.e., $M_2 = \{s_{C_1}, s_{C_2}, \cdots, s_{C_{l_2}}\}$, where s_{C_j} denotes the VT syndrome of the jth chunk string. Since the length of each chunk-string is $n_c l_1$, each s_{C_j} is an integer between 0 and $n_c l_1$. Therefore the number of bits required to represent the VT syndromes of the l_2 chunk-strings is is $l_2 \lceil \log(n_c l_1 + 1) \rceil$.

The final part of the message is the parity check syndrome of X with respect to a linear code. Consider a linear code of length n with parity check matrix $\boldsymbol{H} \in \{0,1\}^{z \times n}$. Then $M_3 = \boldsymbol{H} X$ is the third component of M. The coset of the linear code containing X will be used as an erasure correcting code. In our experiments in Sec. IV, the linear code is chosen to be either a Reed-Solomon code, or a random linear code defined by a random binary parity check matrix. The number of bits in M_3 is equal to the number of rows of $\boldsymbol{H}$, i.e., number of binary parity checks in the code, z. The overall number of bits required to represent the message $M = [M_1, M_2, M_3]$ is

$$l_1 \lceil \log_2(n_b + 1) \rceil + l_2 \lceil \log_2(n_c l_1 + 1) \rceil + z. \tag{4}$$

Since $n_b = n_c l_2$, normalizing by $n = n_c l_1 l_2$ gives the *synchronization rate* R of our scheme

$$R = \frac{z}{n} + \frac{\lceil \log_2(n_c l_2 + 1) \rceil}{n_c l_2} + \frac{\lceil \log_2(n_c l_1 + 1) \rceil}{n_c l_1}. \tag{5}$$

Example 1. *Suppose that we want to design a code for synchronizing a sequence of length $n = 60$ from $k = 4$ deletions. Choose the chunk length $n_c = 4$, so that there are 15 chunks in the string. Divide the string into $l_1 = 5$ blocks, each comprising $l_2 = 3$ chunks. Thus there are 5 blocks each consisting of 3 adjacent chunks, and 3 chunk-strings each consisting of 5 separated chunks.*

We use a Reed-Solomon code defined over $GF(2^4)$ with length $2^4 - 1 = 15$. We also choose the parity check matrix to have 4 parity check equations in $GF(2^4)$, so we can recover 4 erased chunks using this Reed-Solomon code.

Assume that the sequence X in $GF(2^4)$ is

$$X = [4\ 10\ 5 \quad 0\ 3\ 14 \quad 7\ 7\ 1 \quad 0\ 2\ 4 \quad 4\ 6\ 8]^T. \tag{6}$$

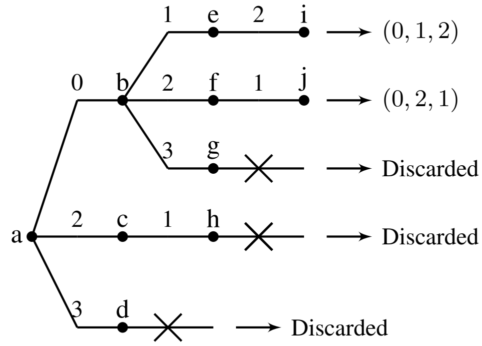

Fig. 3: Tree representing the valid block vectors for Example 2.

Each symbol above represents a chunk of $n_c = 4$ *bits. The first block* $[4\ 10\ 5]$ *in binary is* $B_1 = 0100\ 1010\ 0101$. *The VT syndrome of this sequence is* $s_{B_1} = \mathsf{syn}(B_1) = 10$. *The VT syndromes of the other four blocks are* $6, 3, 4,$ *and* 11*, respectively. We therefore have* $M_1 = \{10, 6, 3, 4, 11\}$.

We similarly compute M_2. *The first chunk-string* $[4\ 0\ 7\ 0\ 4]$ *in binary is* $C_1 = 0100\ 0000\ 0111\ 0000\ 0100$, *with VT syndrome* $s_{C_1} = 11$. *Computing the VT syndromes of the other chunk-strings in a similar manner, we get* $M_2 = \{11, 20, 4\}$.

The final part of the message is the syndrome of X *with respect to the Reed-Solomon parity check matrix. We use the following parity check matrix* $\boldsymbol{H}$ *in* $GF(2^4)$*:*

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 4 & 8 & \cdots & 2^{14} \\ 1 & 4 & 3 & 12 & \cdots & 2^{2(14)} \\ 1 & 8 & 12 & 10 & \cdots & 2^{3(14)} \end{bmatrix} \tag{7}$$

to compute $M_3 = \boldsymbol{H}X = [11, 6, 13, 2]^T$. *As* $z = 16$ *bits are needed to represent the parity check syndrome, the total number of bits to convey the message is* $5\lceil\log(13)\rceil + 3\lceil\log(21)\rceil + 16 = 51$ *bits.*

III. Decoding algorithm

The goal of the decoder is to recover X given Y, n and the message $M = [M_1, M_2, M_3]$. From M_1, M_2, the decoder knows the VT syndrome of each block and each chunk-string. Using this, the decoder first finds all possible configurations of deletions across blocks, and then for each of these configurations, it finds all possible chunk deletion patterns. Since each chunk is the intersection of a block and a chunk-string, each chunk plays a role in determining exactly two VT syndromes. The intersecting construction of blocks and chunk-strings enables the decoder to iteratively recover the deletions in a large number of cases. The decoder is then able to localize the positions of the remaining deletions to within a few chunks. These chunks are considered erased, and are finally recovered by the erasure-correcting code. The decoding algorithm consists of six steps, as described below.

Step 1: Block boundaries

In the first step, the decoder produces a list of candidate *block-deletion patterns* $V = (a_1, \cdots, a_{l_1})$ compatible with Y, where a_i is the number of deletions in the ith block. Each pattern in the list should satisfy $\sum_{i=1}^{l_1} a_i = k$ with $0 \le a_i \le k$. The list of candidates always includes the true block-deletion pattern. It is convenient to represent the candidate block-deletion patterns as branches on a tree of depth l_1, as shown in Fig. 3. At every level (block) $i = 1, \dots, l_1$, branches are added and labeled with all possible values of a_i. Specifically, the tree is constructed as follows.

Depth 1 *of the tree*: Consider the first n_b received bits $Y(1:n_b)$, compute its VT syndrome $u = \mathsf{syn}(Y(1:n_b))$ and compare it with s_{B_1}, the correct syndrome of the first block. There are two alternatives for the k branches of the first level.

1) $\underline{u = s_{B_1}}$: First, the decoder adds a branch with $a_1 = 0$, corresponding to the case that the first n_b bits are deletion-free. The first block cannot have just one deletion, because in this case the single-deletion correcting property of the VT code would imply that $u \neq s_{B_1}$. However, it is possible that two or more than two deletions happened in block one, and by considering additional bits from the next block, the VT-syndrome of first n_b bits accidentally matches with s_{B_1}. For example, consider blocks of length $n_b = 4$, and let the first two blocks of X be $\underline{01}00\ 1111\ \dots$, with the underlined bits deleted we get $Y = 001111\dots$. In this case $u = s_{B_1} = 2$. The decoder thus adds a branch for $a_1 = 0, 2, \dots, k$.
2) $\underline{u \neq s_{B_1}}$: Block one contains one or more deletions and the decoder adds a branch for $a_1 = 1, 2, \dots, k$.

Depth $i+1$, $1 \le i < l_1$: Assume that we have constructed the tree up to depth i. Consider a branch of the tree at depth i with the number of deletions in blocks 1 through i given by $a_1, a_2, \cdots, a_i$, respectively. This gives us the starting position of block $(i+1)$ in Y. Denote this starting position by

$$p_{i+1} = n_b i - d_i + 1. \tag{8}$$

where $d_i = \sum_{j=1}^{i} a_j$ is the number of deletions on the branch up to block i. Compute the VT syndrome of next n_b bits $u = \mathsf{syn}\left(Y\left(p_{i+1} : p_{i+1} + n_b - 1\right)\right)$. There are two alternatives:

1) $\underline{u = s_{B_{i+1}}}$: If $(k - d_i) < 2$ then the only possibility is that $a_{i+1} = 0$. Instead, if $(k - d_i) \ge 2$, $k - d_i - 1$ branches are added for $a_{i+1} = 0, 2, \dots, k - d_i$.
2) $\underline{u \neq s_{B_{i+1}}}$: If $(k - d_i) > 0$ then there are $(k - d_i)$ possibilities at this branch: the ith block can have $1, 2, \cdots, (k - d_i)$ deletions. If $(k - d_i) = 0$, it is assumed this is an invalid branch, and the path is discarded.

Example 2. *Assume* $k = 3$ *deletions,* $l_1 = 3$ *blocks, and that the true deletion pattern is (0,2,1), i.e., there are zero deletions in the first block, two deletions in second block, and one deletion in third block. The tree constructed by the decoder depends on the underlying sequences* X *and* Y*. In Fig. 3, we illustrate one possible tree constructed for this scenario without explicitly specifying* X *and* Y*.*

Assume that in the first step, the syndrome matches with s_{B_1}*, so we have* $a_1 = 0, 2,$ *or* 3 *. At node b (corresponding to* $a_1 = 0$*), suppose that the syndrome does not match with* s_{B_2}*, so we have* $a_2 = 1, 2,$ *or 3. Now suppose that at nodes* *c and d, the syndrome does not match with s_{B_2}. At node d, $a_1 = 3$, so there are no more deletions available for the second block; so this branch is discarded. At node c, $a_1 = 2$, so the only possibility is one deletion in the second block. Then if the syndrome at node h does not match s_{B_3}, the branch is discarded. At nodes e and f, we assign the remaining deletions to the last block. At node g, the syndrome does not match with a_3, and the branch is discarded.*

Step 2: Primary fixing of blocks

Denote by r_1 the size of the list after the first step and denote the corresponding block-deletion patterns by $V_1, \cdots, V_{r_1}$. In this second step, for each of the block-deletion patterns, we restore the deleted bit in blocks containing a single deletion by using the VT decoder. Specifically, for every block-deletion pattern $V = (a_1, \cdots, a_{l_1})$, let the ith block of Y with respect to V be $S = Y(p_i : p_i + n_b - a_i - 1)$ where p_i is the starting position of the ith block in Y, defined analogously to (8). If $a_i = 1$, feed the sequence S to the VT decoder and in Y, replace S with the decoded sequence. After this, the ith block in Y is deletion free, so update the block-deletion pattern V by setting $a_i = 0$. We carry out this procedure for all blocks with one deletion in V. This results in a sequence $\hat{Y}$, which is obtained from Y by recovering the single-deletion blocks corresponding to block-deletion pattern V. Denote the updated version of block-deletion pattern V by $\hat{V}$. Thus at the end of this step, we have r_1 updated candidate sequences $\hat{Y}_1, \cdots, \hat{Y}_{r_1}$ with corresponding block-deletion patterns $\hat{V}_1, \cdots, \hat{V}_{r_1}$.

Example 3. *Consider the code of Example 1 with $l_1 = 5$ blocks, and $k = 4$ deleted bits. If the list of block-deletion patterns at the end of the first step is $V_1 = (1, 1, 1, 1, 0)$, $V_2 = (1, 1, 2, 0, 0)$, $V_3 = (1, 2, 1, 0, 0)$, $V_4 = (2, 0, 2, 0, 0)$, then the updated list of block-deletion patterns is $\hat{V}_1 = (0, 0, 0, 0, 0)$, $\hat{V}_2 = (0, 0, 2, 0, 0)$, $\hat{V}_3 = (2, 0, 0, 0, 0)$, $\hat{V}_4 = (2, 0, 2, 0, 0)$.*

Step 3: Chunk Boundaries

In this step, for each updated block-deletion pattern $\hat{V}$ and the corresponding $\hat{Y}$, we list all possible allocations of deletions across chunks. More precisely, for each pair $(\hat{Y}, \hat{V})$ we list all possible $l_1 \times l_2$ matrices $\boldsymbol{A} = (a_{ij})$, where a_{ij} is the number of deletions in the jth chunk of the ith block, such that $\sum_{j=1}^{l_2} a_{ij} = a_i$, the ith entry of $\hat{V}$. The jth column of matrix $\boldsymbol{A}$, specifies the number of deletions in the l_1 chunks of the jth chunk-string. For example, some of the possible matrices for $\hat{V}_4 = (2, 0, 2, 0, 0)$ in Example 3 are

$$\boldsymbol{A}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \boldsymbol{A}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \boldsymbol{A}_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{9}$$

The algorithm that lists all chunk-deletion matrices $\boldsymbol{A}$ compatible with a given block-deletion pattern $\hat{V} = (a_1, \ldots, a_{l_1})$ is completely analogous to the tree construction described in Step 1. In this case, for each block-deletion pattern $\hat{V}$, another tree will be constructed, with each path in the tree representing a valid chunk-deletion matrix $\boldsymbol{A}$. A detailed description of this tree construction is given in [10, p.4]

Step 4: Iterative correction of blocks and chunk-strings

At the end of step 3, the decoder provides a list of pairs $(\hat{Y}, \boldsymbol{A})$, where $\hat{Y}$ is a candidate sequence to be decoded using the chunk-deletion pattern matrix $\boldsymbol{A}$, with a_{ij} being the number of deletions in the jth chunk of the ith block. Denote the number of such pairs in the list by r_3.

Similarly to step 2, in step 4 we use the VT syndromes (known from M_1 and M_2) to recover deletions in blocks and chunk-strings for which the matrix $\boldsymbol{A}$ indicates a single deletion. Whenever a deletion recovered using a VT decoder lies in a chunk different from the one indicated by $\boldsymbol{A}$, the candidate is discarded. Simulations indicate that this is an effective way of discarding several invalid candidates. The iterative procedure is described in detail in [10, p.5]. Denote the updated candidate pairs at the end of this procedure by $(\tilde{Y}, \tilde{\boldsymbol{A}})$, and assume there are r_4 of them.

As an illustrative example, consider the three chunk-matrices given in (9). In $\boldsymbol{A}_1$, we can successfully recover all the deletions. In $\boldsymbol{A}_2$, we can only fix two deletions in the third block. However, for $\boldsymbol{A}_3$, we cannot recover any of the deletions. Thus, the updated $\tilde{\boldsymbol{A}}$ matrices are

$$\tilde{\boldsymbol{A}}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \tilde{\boldsymbol{A}}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \tilde{\boldsymbol{A}}_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{10}$$

Step 5: Replacing deletions with erasures

In this step, for each of the r_4 surviving pairs $(\tilde{Y}, \tilde{\boldsymbol{A}})$, we replace each chunk of $\tilde{Y}$ that still contains deletions with n_c erasures. Hence, if there are ν chunks with deletions (where $0 \leq \nu \leq k$), the resulting sequence will have length n, with $n_c \nu$ erasures and no deletions. Notice that this operation of replacing with erasures can be performed without ambiguity since $\tilde{\boldsymbol{A}}$ precisely indicates the starting position of each chunk and also the number of deletions within that chunk.

The purpose of the linear code is to recover from the remaining erasures. The minimum distance of the linear code should be large enough to guarantee that we can resolve all the νn_c erased bits. In Example 1 with four deletions, we will have at most $\nu = 4$ erased chunks, so we choose a Reed-Solomon code with 4 parity check equations in $GF(2^4)$. Some invalid candidates may be discarded in the process of correcting the erasures as we may find that the parity check equations cannot be solved.

Step 6: Discarding invalid candidates

The reconstructed sequences at the end of Step 5, denoted by $\hat{X}$, all have length n and are deletion free. For each of the r_5 sequences $\hat{X}$, we check the VT and parity-check constraints for each of the block and chunk-strings and discard those not meeting any of the constraints. The surviving r_6 distinct sequences comprise the final list produced by the decoder.

The final list of reconstructed sequences comprises all length-n sequences that can be obtained by adding k bits to Y and also satisfy all the VT and parity check constraints. The correct sequence is always among the r_6 candidates. The synchronization algorithm is said to be zero-error if and only if $r_6 = 1$ for all sequences and deletion patterns. When $r_6 > 1$, the list size can be further reduced if additional hash functions or cyclic redundancy checks are available from the encoder.

IV. Numerical Examples

To understand the effect of the various system parameters, we considered the setups shown in Table I. For each setup, the performance was recorded over 10^6 simulation trials. In each trial, the sequence X and the k deletion locations were chosen independently and uniformly at random. For the first five setups, we used parity check constraints from a Reed-Solomon code over $GF(2^{n_c})$ with code length $(2^{n_c} - 1)$. For example, setup 5 uses 7 parity check constraints from a Reed-Solomon code over $GF(2^6)$, corresponding to $z = 42$ parity bits. In the last two setups, denoted with asterisks, we used a random linear code, i.e., z binary parity check constraints drawn in an equiprobable manner.

TABLE I: Number of deletions and code parameters for each setup.

	k	n	l_1	l_2	n_c	z	R
Setup 1	3	60	5	3	4	4	0.650
Setup 2	3	60	5	3	4	8	0.717
Setup 3	3	60	5	3	4	12	0.783
Setup 4	4	60	5	3	4	16	0.850
Setup 5	7	378	9	7	6	42	0.365
Setup 6	7	486	9	9	6	50*	0.325
Setup 7	10	2800	20	20	7	60*	0.135

TABLE II: List size after each step.

	$\bar{r}_1$	$\bar{r}_3$	$\bar{r}_4$	$\bar{r}_6$	$\max r_6$	$r_6 > 1$
Setup 1	1.87	1.92	1.42	1.003	3	3256
Setup 2	1.87	1.92	1.42	1.000	2	25
Setup 3	1.87	1.92	1.42	1	1	0
Setup 4	3.39	6.18	2.53	1	1	0
Setup 5	11.51	74.43	3.42	1	1	0
Setup 6	11.20	28.64	2.55	1	1	0
Setup 7	12.76	26.16	1.57	1	1	0

Table II shows the list sizes of the number of candidates at the end of various steps of the decoding process. Recall that r_1 is the number of candidate block-deletion patterns at the end of step 1, r_3 is the number of pairs $(\hat{Y}, \boldsymbol{A})$ at the end of step 3, r_4 is the number of pairs $(\tilde{Y}, \tilde{\boldsymbol{A}})$ at the end of step 4, and r_6 is the number of sequences $\hat{X}$ in the final list. Over 10^6 trials, the average of r_i is denoted by $\bar{r}_i$, the column $\max r_6$ shows the maximum size of the final list, and the column $r_6 > 1$ shows the number of trials for which $r_6 > 1$.

The first three setups have identical parameters, except for the number of Reed-Solomon parity checks. We observe that adding more parity check constraints increases the rate and improves decoding performance by reducing the number of trials with list size greater than one.

The fourth setup is precisely the code described in Example 1. It has the same values of (n_c, l_1, l_2) as the first three setups but with a larger number of deletions and parity check constraints. We observe that increasing the number of deletions (with n_c, l_1, l_2 unchanged) increases the average number of candidates in the different decoding steps. In general, choosing $l_1 \geq k$ ensures that the average list size after step 1 is small.

The fifth setup is a more practical code with length $n = 378$, and $k = 7$ deletions. Though the final list size is always one, there are a large number of candidates at the end of the third step; this increases the decoding complexity. Comparing this with setup six, we observe that increasing l_2 significantly reduces the number of candidates at the end of the third step. This is because of the increase in the number of chunk-string VT constraints, which allows the decoder to eliminate more candidates while determining chunk boundaries.

The last setup is a relatively long code. Although the average number of candidates in each of the decoding steps is not very high, a small fraction of trials have a very large number of candidates, resulting in considerably slower decoding for these trials. For future work, an interesting direction is to consider lower-complexity decoders that allow for an early elimination of highly unlikely candidates. This would limit the number of candidates at the end of each decoding step at the expense of introducing a probability of error, i.e., a non-zero probability that the final list does not contain the true X sequence.

Acknowledgment

We thank Jossy Sayir for kindly making his Reed-Solomon code simulator available to us.

References

[1] G. Cormode, M. Paterson, S. C. Sahinalp, and U. Vishkin, "Communication complexity of document exchange," in *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pp. 197–206, 2000.

[2] S. M. S. Tabatabaei Yazdi and L. Dolecek, "Synchronization from deletions through interactive communication," *IEEE Trans. Inf. Theory*, vol. 60, pp. 397–409, Jan. 2014.

[3] R. Venkataramanan, V. N. Swamy, and K. Ramchandran, "Low-complexity interactive algorithms for synchronization from deletions, insertions, and substitutions," *IEEE Trans Inf. Theory*, vol. 61, no. 10, pp. 5670–5689, 2015.

[4] N. Ma, K. Ramchandran, and D. Tse, "Efficient file synchronization: A distributed source coding approach," in *Proc. IEEE Int. Symp. Inf. Theory*, 2011.

[5] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors," *Automatica i Telemekhanica*, vol. 26, no. 2, pp. 288–292, 1965.

[6] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965.

[7] A. Orlitsky, "Interactive communication of balanced distributions and of correlated files," *SIAM J. Discrete Math.*, vol. 6, no. 4, pp. 548–564, 1993.

[8] A. Orlitsky and K. Viswanathan, "One-way communication and error-correcting codes," *IEEE Trans. Inf. Theory*, vol. 49, no. 7, pp. 1781–1788, 2003.

[9] N. J. A. Sloane, "On single-deletion-correcting codes," in *Codes and Designs, Ohio State University (Ray-Chaudhuri Festschrift)*, pp. 273–291, 2000. Online: https://arxiv.org/abs/math/0207197.

[10] M. Abroshan, R. Venkataramanan, and A. Guillén i Fàbregas, "Multilayer codes for synchronization from deletions," 2017. (Online) https://arxiv.org/abs/1705.06670.