Non-Binary Codes for Correcting a Burst of at Most \( t \) Deletions

Shuche Wang\( ^\text{\textregistered} \), Graduate Student Member, IEEE, Yuanyuan Tang\( ^\text{\textregistered} \), Member, IEEE, Jin Sima\( ^\text{\textregistered} \), Ryan Gabrys\( ^\text{\textregistered} \), Member, IEEE, and Farzad Farnoud\( ^\text{\textregistered} \), Member, IEEE

Abstract—The problem of correcting deletions has received significant attention, partly because of the prevalence of these errors in DNA data storage. In this paper, we study the problem of correcting a consecutive burst of at most \( t \) deletions in non-binary sequences. When the alphabet size \( q \) is even, we first propose a non-binary code correcting a burst of at most 2 deletions for \( q \)-ary alphabets. Afterwards, we extend this result to the case where the length of the burst can be at most \( t \) where \( t \) is a constant. Finally, we consider the setup where the sequences that are transmitted are permutations. The proposed codes are the largest known for their respective parameter regimes.

Index Terms—Burst of deletions, DNA data storage, error-correcting codes, permutation codes.

I. INTRODUCTION

Codes correcting insertions/deletions have garnered significant recent interest due to their relevance in many applications such as storage [1], [2], communication systems [3] and file synchronization [4]. Constructing codes in the insertion/deletion metric is a notoriously difficult problem whose origins date back to at least the 1960s [5]. One of the challenges under this setup is that deletions seem to be more destructive in nature than substitutions as only a relatively small number of insertions/deletions can cause the transmitted and received sequences to be vastly different under the Hamming metric.

Table I

| Size of burst | Redundancy |
|--------------|------------|
| \( t \leq 2 \) | \( \log n + 1 \) |
| \( t = 3 \) | \( \log n + (t - 1) \log \log n + O(1) \) |
| \( t \leq 4 \) | \( (t - 1) \log n + (t(t - 1)/2 - 1) \log \log n + O(1) \) |
| \( t \leq 5 \) | \( t \log \log n + (t(t + 1))/2 \log \log \log n + O(1) \) |
| \( t \leq 6 \) | \( t \log n + (t(t + 1)/2) \log \log n + O(1) \) |
| \( t \leq 7 \) | \( 4 \log n + o(\log n) \) |

One of the motivations for the current work is the recent emergence of DNA-based storage systems [2]. Unlike traditional information systems whose dominant source of errors stems from substitutions, data stored in DNA is often corrupted by bursts of insertions and deletions [6]. Motivated by this connection, the current work focuses on the development of non-binary codes capable of correcting a consecutive burst of deletions.

Previous works have studied the problem of constructing codes over binary alphabets, and optimal codes exist for many setups of interest. In what was perhaps the earliest work on the subject, Levenshtein constructed a code capable of correcting a burst of length at most two that had redundancy \( \log n + 1 \) [7]. In [8], Schoeny et al. proposed burst deletion correcting codes for the setup where the length of the burst is exactly \( t \) and the deletions are consecutive. In [9], Lenz and Polyanskii presented codes that correct consecutive bursts of deletions of length at most \( t \) that required only \( \log n + O(\log \log n) \) bits of redundancy. The best-known systematic codes can be found in [10]. A summary of these results is included in Table I.

Unlike the previously mentioned works, the goal in this paper is to construct low redundancy non-binary codes capable of correcting a burst of length at most \( t \) where \( t \) is a constant. In this work, we also consider the setup where the non-binary sequences that comprise our code are permutations. Our main results, which are highlighted in Table II, are the following:

1) We present a simpler proof for Levenshtein’s binary code [7], based on the well-known Varshamov-Tenengolts constraint. We then show this proof can be used for constructing a code with redundancy at most \( \log n + 2 \log q + 1 \) for correcting an induced burst of 2 deletions in alternating sequences.

\( ^1 \)Induced deletions occur in sequences where every two adjacent symbols are different. This setup is motivated by the recently proposed terminator-free synthesis of DNA sequences [6].

0018-9448 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
2) We construct non-binary codes for correcting a burst of at most \( t \) deletions for \( q \)-ary alphabets that has redundancy 
\[
\log n + 8\log q \log \log n + O(\log q),
\]
where \( q \geq 2 \) is an even integer.

3) Using ideas developed in the context of non-binary codes, we present a permutation code for correcting a burst of at most \( t \) deletions that has redundancy 
\[
\log n + O(\log n).
\]
To the best of the authors’ knowledge, the codes presented here are the largest known codes for each of the parameter regimes under consideration. We note that result (3), which appears in Section VI, was simultaneously and independently derived in [11]. Since our approach uses a different technique than the one from [11], it may be of independent interest.

The remainder of this article is structured as follows. Section II presents the notations and two well-known deletion correcting codes used throughout this paper as well as some preliminary results. Section III gives an alternative proof of the Levenshtein code and a code for correcting deletions in alternating sequences is proposed based on this proof. In Section IV, when \( q \) is even, we construct a non-binary code for correcting a burst of at most 2 deletions for \( q \)-ary alphabets with redundancy 
\[
\log n + 8\log q \log \log n + O(\log q).
\]
In Section V, when \( q \) is even, we construct a non-binary code for correcting a burst of at most \( t \) deletions for \( q \)-ary alphabets with redundancy 
\[
\log n + O(\log q \log \log n).
\]
Section VI proposes a permutation code for correcting a burst of at most \( t \) deletions with redundancy 
\[
\log n + O(\log \log n).
\]
Finally, Section VII concludes the paper.

## II. Notation and Preliminaries

We now describe the notations used throughout this paper. Let \( \Sigma_n \) denote a finite alphabet of size \( q \) and \( \Sigma_n^u \) represent the set of all sequences of length \( n \) over \( \Sigma_q \). Without loss of generality, we assume \( \Sigma_q = \{0, 1, \ldots, q-1\} \). For ease of notation, we will denote the set \( \{0, 1, \ldots, m-1\} \) as \( [m] \) and the set \( \{1, 2, \ldots, m\} \) as \( [m] \). For two integers \( i < j \), let \( [i, j] \) denote the set \( \{i, i+1, i+2, \ldots, j\} \).

We write sequences with bold letters, such as \( u \) and their elements with plain letters, e.g., \( u = u_1 \cdots u_n \) for \( u \in \Sigma_q^n \). For functions, if the output is a sequence, we also write them with bold letters, such as \( \phi(u) \). The \( i \)th position in \( \phi(u) \) is denoted \( \phi(u)_i \). We typically use \( u \) for non-binary and \( x \) for binary sequences. The length of the sequence \( x \) is denoted \( |x| \). \( x_{[i, j]} \) denotes the substring beginning at index \( i \) and ending at index \( j \), inclusive. A run is a maximal substring consisting of identical symbols. The weight \( w(x(u)) \) of a sequence \( u \) represents the number of non-zero symbols in it. A sequence \( u \) is said to have period 2 if \( u_i = u_{i+2} \) for \( i \in [u| - 2]. \)

Define \( L(u, 2) \) as the length of the longest substring of \( u \) with period 2.

A burst of \( t \) deletions deletes \( t \) consecutive symbols from \( u = u_1 \cdots u_n \) starting in position \( i + 1 \) resulting in \( u' = u_1 \cdots u_i u_{i+1+t} \cdots u_n \) for \( u \in \Sigma_q^n \) (we usually use \( u' \) to denote the sequence resulting from deleting symbols from \( u \)). For shorthand, let \( D_i(u) \subseteq \Sigma_n^{n-i} \) denote the set of all sequences possible given that a burst of \( t \) deletions occur to \( u \) and similarly let \( D_{<i}(u) = D_1(u) \cup D_2(u) \cup \cdots \cup D_i(u) \). The size of a code \( C \subseteq \Sigma_q^n \) is denoted \(|C|\) and its redundancy is defined as \( \log(\frac{q^n - |C|}{|C|}) \), where all logarithms in this paper are to the base 2. We say that a code \( C \subseteq \Sigma_q^n \) is a \( t \)-burst-error-correcting code if for two distinct \( u, v \in C \), \( D_{<i}(u) \cap D_{<i}(v) = \emptyset \).

Let \( n \) be a positive integer and \( S_n \) be the set of all permutations on the set \([n]\). Denote \( \pi = (\pi_1, \pi_2, \ldots, \pi_n) \) \( \in \) \( S_n \) as a permutation with length \( n \). A burst of at most \( t \) deletions deletes at most \( t \) consecutive symbols from the permutation \( \pi \) starting in position \( i + 1 \), leading to \( \pi' = (\pi_1, \pi_2, \ldots, \pi_{i-1}, \pi_{i+t}, \pi_{i+t+1}, \ldots, \pi_n) \) where \( t \leq \pi \). The size of a permutation code \( P \subseteq S_n \) is denoted \(|P|\) and its redundancy is defined as \( \log(n!/|P|) \).

The following codes will be of use in the paper. First for \( u \in \Sigma_q^n \), define the VT syndrome as \( VT(u) = \sum_{i=1}^n u_i \).

Furthermore, define \( \phi : \Sigma_n^q \rightarrow \Sigma_n^q \) as
\[
\phi(u)_i =\begin{cases} 1, & \text{if } u_i > u_{i-1} \\ 0, & \text{if } u_i \leq u_{i-1} \end{cases}
\]
for \( i \geq 2 \) and \( \phi(u)_1 = 1 \), where we have slightly modified the definition to simplify the proof of Theorem 7.

**Theorem 1 (Varshamov-Tenengolts (VT) code [14]):** For integers \( n \) and \( a \in [n+1] \),
\[
VT_a(n) = \{ x \in \Sigma_q^n : VT(x) \equiv a \mod (n+1) \} \tag{1}
\]
is a 1-burst-error-correcting code.

**Theorem 2 (Tenengolts code [15]):** For integers \( n, a \in [n], \) and \( b \in [q] \), the code
\[
C_T(a, b, n) = \{ u \in \Sigma_q^n : VT(\phi(u)_{[2, n]}) \equiv a \mod n, \sum_{i=1}^n u_i \equiv b \mod q \} \tag{2}
\]
is a 1-burst-error-correcting code.

To decode a received word \( u' \), we first reconstruct \( \phi(u')_{[2, n]} \) from \( \phi(u')_{[2, n]} \) using a decoder for the VT code. The difference between \( \phi(u') \) and \( \phi(u) \) tells us in which maximal monotonically increasing (or non-increasing) substring of \( u' \) the deleted symbol need to be inserted. As the identity of this symbol is known from \( b \), its correct position can be identified.

Let \( M_t(n) \subseteq \Sigma_q^n \) be a \( t \)-burst-error-correcting code of maximum cardinality. Theorem 3 provides a non-asymptotic upper bound on the size of \( M_t(n) \) using the linear programming technique from [8]. A detailed proof is included in Appendix A.

**Theorem 3:** For \( n > t \) and \( t/n \), we have
\[
|M_t(n)| \leq \frac{q^{n-t+1} - q^t}{(q-1)(n-2t+1)} \tag{3}
\]
From Table I and II, we can see that the redundancies of our constructions are only off from the minimum redundancy by a factor of at most roughly \( \log q \log \log n \).

Motivated by applications to storage systems with larger alphabets, we will also be interested in codes over permutations. Let \( S_n \) be the set of all permutations on the set \([n]\) and \( M^F_t(n) \subseteq S_n \) be a \( t \)-burst-error-correcting code of maximum cardinality. Theorem 4 provides a upper bound on the size of \( M^F_t(n) \).

**Theorem 4 (11, Theorem 1):** Let \( n > t \) be positive integers. Then,

\[
|M^F_t(n)| \leq \frac{n!}{t!(n-t+1)!}.
\]

From Table II, notice that the redundancy of our construction for permutation codes is only off from the minimum redundancy by at most \( O(\log \log n) \).

### III. Binary Code Correcting a Burst of at Most 2 Deletions

In this section, we first describe the Levenshtein code [7] in Subsection III-A, which can correct a burst of at most 2 deletions in binary sequences. In Section III-B, we provide an alternative formulation of the Levenshtein code, and using this formulation, we prove the correctness of the construction by describing the decoding algorithm. Our proof is in some ways simpler than Levenshtein’s original proof, and it is similar to the well-known proof of the VT code [14]. It will also enable us to construct a code for correcting deletions in alternating sequences in Section III-C.

#### A. Levenshtein Binary Codes for Correcting at Most 2 Deletions

For a binary sequence \( x \), let \( \psi(x) \) be the sequence whose \( i \)th element is the run index of \( x_i \) in \( x \), where the indexing starts from 0. Then, define the syndrome \( VT^{(r)}(x) \) of the sequence \( \psi(x) \) as

\[
VT^{(r)}(x) = \sum_{i=1}^{n} \psi(x)_i.
\]

For example, if \( x = 01110100 \), then \( \psi(x) = 01112344 \), and \( VT^{(r)}(x) = 1 \times 3 + 2 \times 1 + 3 \times 1 + 4 \times 2 = 16 \).

**Theorem 5 (Levenshtein code [7]):** For integers \( n \) and \( a \in [[2n]] \),

\[
C_L(a, n) = \left\{ x \in \Sigma^2_n : VT^{(r)}(0x) \equiv a \mod 2n \right\}
\]

is a 2-burst-error-correcting code.

Using a simple averaging argument, it is straightforward to observe that \( |C_L(a, n)| \geq \frac{2^n}{2n-1} \) and so the redundancy of this code is at most \( 1 + \log n \) (for some \( a \in [[2n]] \)). For shorthand, we refer to the code \( C_L(a, n) \) as the Levenshtein code.

#### B. An Alternative Formulation of the Levenshtein Code

Define \( \psi(x) \in \Sigma^2_n \) to be the derivative of \( x \in \Sigma^2_n \) so that

\[
\psi(x)_i = \begin{cases} 
  x_i \oplus x_{i+1}, & i = 1, 2, \ldots, n-1 \\
  x_n, & i = n
\end{cases}
\]

where \( a \oplus b \) denotes \((a + b) \mod 2\). It is clear that \( \psi \) is bijection. Levenshtein [7] showed that \( VT(\psi(x)) \equiv -VT(r)(0x) \) \((\mod 2n)\) for \( x \in \Sigma^2_n \). This equality provides another way to prove the error-correcting capability of the code.

**Theorem 6:** The code \( \{x \in \Sigma^2_n : VT(\psi(x)) \equiv a \mod 2n \} \) can correct a burst of at most 2 deletions.

**Proof:** For a codeword \( x \), let \( x' \) be obtained from \( x \) after a burst of at most 2 deletions. Also, let \( y = \psi(x) \), \( y' = \psi(x') \), and \( \Delta = VT(y) - VT(y') \). The error in \( x \) can affect \( y \) in the following ways:

- If the first one or two symbols of \( x \) are deleted, then the first one or two symbols of \( y \) are deleted, respectively.
- If \( x_i \) is deleted (where \( i \in \{2, 3, \ldots, n\} \)), then \( y_{i-1}y_i \) is replaced by \( y_{i-1}y_i \) since \( y'_{i-1} = x_{i-1} \oplus x_{i+1} = (x_{i-1} + x_i + x_{i+1}) \mod 2 = y_{i-1} \oplus y_i \).
- If \( x_i \) is deleted (where \( i \in \{1, 2, \ldots, n\} \)), then \( y_{i-1}y_iy_{i+1} \) is replaced by \( y_{i-1}y_iy_{i+1} \).
- If \( x_i, x_{i+1} \leq i \leq n - 1 \), are deleted, then \( y_{i-1}y_iy_{i+1} \) is replaced by \( y_{i-1}y_iy_{i+1} \).
- If \( x_i, x_{i+1} \leq i \leq n - 1 \), are deleted, then \( x_{i-1}y_{i+1} \) is replaced by \( x_{i-1}y_{i+1} \).
- If \( x_i, x_{i+1} \leq i \leq n - 1 \), are deleted, then \( x_{i-1}y_iy_{i+1} \) is replaced by \( x_{i-1}y_iy_{i+1} \).

We can view the error process as being performed by an adversary, who changes the codeword according to one of the aforementioned patterns. The bits that are changed are referred to as altered bits, with \( p \) being the index of the first altered bit. Note that given \( y \) and \( y' \) multiple choices for \( p \) may be possible, for example, when \( a \) is deleted among a run of 0s. Let \( R_1 \) be the number of 1s on the right of the altered bits in \( y \) and \( L_1 \) on their left. Similarly, define \( R_0, L_0 \) and let \( w \) denote the weight of \( y' \). For example, for \( y = 01010100 \), \( y' = 010000 \), the altered bits are the substring 101 and we have \( p = 4, L_0 = 2, L_1 = 1, R_0 = 2, R_1 = 0, w = 1 \). We show how \( y \) and thus \( x \) can be recovered based on \( \Delta \) and \( w \), which are computable at the decoder. Each case can be identified based on the comparison of \( \Delta \) with \( w \).

If one bit is deleted in \( x \), the following cases may occur in \( y \) resulting in \( y' \). As can be seen below, the cases can be distinguished by comparing \( \Delta \) and \( w \) so that it is possible to recover \( y \) in each case.

1.1) If \( 00 \rightarrow 0, 10 \rightarrow 1 \) or a 0 is deleted from the beginning, then \( \Delta = R_1 \leq w \). We recover \( y \) by inserting a 0 in the rightmost position before \( \Delta \) 1s.

1.2) If \( 01 \rightarrow 1 \), then \( \Delta = R_1 + 1 \leq w \). We recover \( y \) by inserting a 0 in the rightmost position before \( \Delta \) 1s.

1.3) If the first bit is deleted from \( y \) resulting in \( y' \) and it is a 1, then \( \Delta = R_1 + 1 = w + 1 \). To correct the error, we prepend 1 to \( y' \).

1.4) If \( 11 \rightarrow 0 \), then \( \Delta = 2p + 1 + R_1 = 2(L_0 + L_1 + 1) + 1 + R_1 = 2L_0 + L_1 + w + 3 \geq w + 3 \). As \( 2p + 1 + R_1 \) is strictly increasing in \( p \), there is a unique value of \( p \) satisfying the equation and we can again recover \( y \).

If two bits are deleted in \( x \), one of the following cases occurs in \( y \). Similar to before, each case can be identified based on the comparison of \( \Delta \) with \( 2w, 2w + 1, 2w + 2, 2w + 3 \), and its parity.
2.1) If 010 \to 1 occurs, then \( \Delta = 2R_1 + 1 < 2w \). We insert two 0s on both sides of the \( R_1 \)th 1 from the right, where \( R_1 = (\Delta - 1)/2 \).

2.2) If 000 \to 0, 001 \to 1 or 100 \to 1, then \( \Delta = 2R_1 \leq 2w \). We insert two 0s on the left of the \( R_1 \)th 1 from the right, where \( R_1 = \Delta/2 \). Note that in this case, \( \Delta \) is an even number whereas in Case 2.1 it was odd.

2.3) If 00 is deleted from the beginning, then \( \Delta = 2R_1 = 2w \). It can be considered a special case of Case 2.2.

2.4) If 10 or 01 is deleted from the beginning, then \( \Delta = 2R_1 + 1 = 2w + 1 \) or \( \Delta = 2w + 2 \), respectively. Here 10 or 01 is prepended, depending on the parity of \( \Delta \).

2.5) If 110 \to 0, 011 \to 0 or 111 \to 1, then \( \Delta = p + (p + 1) + 2R_1 = (L_0 + L_1 + 1) + (L_0 + L_1 + 2) \) \( \geq 2w + 3 \). In this case, we insert 11 immediately after the \( L_0 \)th 0, where \( L_0 = (\Delta - 2w - 3)/2 \).

2.6) If 11 is deleted from the beginning, then \( \Delta = 2p + 1 + 2R_1 \). It can be considered a special case of Case 2.5.

2.7) If 101 \to 0 occurs, then \( \Delta = 2p + 2 + 2R_1 \geq 2w + 4 \). We insert two 1s on the sides of the \( (L_0 + 1) \)th 0, where \( L_0 = \Delta/2 - w - 2 \). Note that in this case, \( \Delta \) is an even number whereas in Case 2.5 it was odd. \( \square \)

C. Correcting an Induced Burst of 2 Deletions in Alternating Sequences

One of the challenges facing DNA data storage is the high cost of synthesizing DNA sequences accurately. Recently, a new enzymatic method for parallel DNA synthesis was proposed in [6], which could decrease the cost of synthesis but with a loss in accuracy. Specifically, the length of the runs cannot be easily controlled. One approach to address this challenge is to store information only in the identity of the symbols of the runs. In this approach, for example, AAACGGCCT, ACCGCTTTT, and AGCCT would be equivalent. In this case, we can consider the information being encoded in a sequence with no adjacent repeats of a symbol, such as ACGCT, which we term an alternating sequence.

The second difficulty of this enzymatic synthesis method is the high proportion of deletion [6], which makes it possible for complete runs to be deleted. Over the space of alternating sequences, the deletion of a single symbol may manifest as a burst of more than 1 deletion. For example, the deletion ACCGCTTTT \to ACCTTTTT will be interpreted as ACCT \to ACT. We call such a burst resulting from a single deletion an induced deletion. Formally, a single induced deletion in an alternating sequence is a deletion that replaces a substring of the form \( ab \) with \( a \), where \( a, b \in \Sigma_q \) and \( a \neq b \).

Induced deletions may be more difficult to handle for trace reconstruction and synchronization approaches given that they cause larger shifts in the sequence. In this subsection, we propose a non-binary code with redundancy \( \log n + 2 \log q + 1 \) to correct an induced burst of length 2, which is very close to the bound shown in (3).

For sequences \( v, w \), let \( v \circ w = v_1w_{1+1}v_2w_{2+1} \cdots \) be obtained by interleaving them. Define the odd and even subsequences of \( v \) as \( v^o = v_1v_3 \cdots \) and \( v^e = v_2v_4 \cdots \), respectively.

| \( x^n \) | \( \psi(x^n) \) | \( y^n \) |
|---|---|---|
| ACCTTTTT | \( \{201, 210\} \) | \( \{201, 210\} \) |
| ACCTTTT | \( \{201, 210\} \) | \( \{201, 210\} \) |
| ACCTTT | \( \{201, 210\} \) | \( \{201, 210\} \) |
| ACCTT | \( \{201, 210\} \) | \( \{201, 210\} \) |
| ACCT | \( \{201, 210\} \) | \( \{201, 210\} \) |

**Theorem 7:** For integers \( a \in \{\pm 2n\}, b \in \{\pm q\}, c \in \{\pm q\}, \) \( C_T(a, b, n) = \{u \in \Sigma_q^n : \text{VT}(\phi(u^o) \circ \phi(u^e)) \equiv a \mod 2n, \sum_i u_i^o \equiv b \mod q, \sum_i u_i^e \equiv c \mod q\} \), can be used to correct the deleted sequence of size 2.

**Proof:** Let \( u \in C_T(a, b, n) \) be a codeword and let \( x = \phi(u^o) \circ \phi(u^e) \). Note that \( x_i = 1 \) if \( u_i > u_{i+2} \) or if \( i \leq 2 \), and \( x_i = 0 \) otherwise. Let \( u^o \) and \( u^e \) be the corresponding sequences after the deletions. We first characterize the changes in \( x \). Suppose \( u_{i+1}u_{i+2} \) are deleted, where \( u_i = u_{i+2} \), for \( 2 \leq i \leq n - 3 \). Then \( u_{i+1}u_{i+2}+1 \) will change to \( u_{i+1} \) in \( x \). Suppose \( u_{i} \equiv u_{i+2} \), then \( u_{i+1} \equiv u_{i+2} \) again because \( u_i = u_{i+2} \), so we do not consider \( u_{i+1} \) as part of the error pattern in \( x \). Besides, since \( u_i = u_{i+2} \), we have \( x_{i+2} = 0 \). Hence, the change in \( x \) can be viewed as \( x_{i+2}+1 \to x_{i+1} \), and the possible cases are \( 101 \to 1, 100 \to 1, 001 \to 0, 001 \to 0, 001 \to 0 \).

If the error is of the form \( u_{i+1}u_{i+2} \to u_3 \), then the change in \( x \) will be of the form \( 101x_{i+1} \to 100 \). The three-bit patterns are then \( 101 \to 1, 100 \to 1 \), both of which appear among the patterns above. If the error is of the form \( u_{n-2}u_{n-1}u_n \to u_{n-2} \), then the change in \( x \) is of the form \( x_{n-2}x_{n-1} \to x_{n-2} \). The last three bits of \( x \) change as \( 000 \to 0, 010 \to 0, 010 \to 0, 110 \to 1 \).

Having determined the changes in \( x \), it can be then shown that the change in \( \psi(x) \) is one of the following cases in Table III: deletion of 11, deletion of 00, 010 \to 1, or 011 \to 1. Then, similar to the proof of Theorem 6, it can be shown that with the knowledge of \( \text{VT}(\psi(x)) \), we can fix the errors in \( \psi(x) \) and in turn the errors in \( x \). We find \( x^{o} = \phi(u^{o}) \) and \( x^{e} = \phi(u^{e}) \). Further, from \( \sum_i u_i^o \equiv b \mod q \) and \( \sum_i u_i^e \equiv c \mod q \), we can get the value of the deleted symbols in \( u^{o} \) and \( u^{e} \), respectively. At last, since \( u^{o} \), \( x^{o} \) and \( x^{e} \) are known, \( u^{e} \) can be recovered via the decoding process for Tenengolts code \( C_T(a, b, n) \). Also, \( u^{e} \) can be recovered in the same way. \( \square \)

Using an averaging argument, we arrive at the following corollary.

**Corollary 1:** There exists a code \( C_T(a, b, n) \) of length \( n \) capable of correcting an induced deletion of length 2 with redundancy at most \( \log n + 2 \log q + 1 \).

**Example 1:** Suppose \( u = (1, 0, 6, 7, 6, 2, 3, 5) \in \Sigma_8^8 \) and the 4th symbol is deleted, so the retrieved sequence is

\[ x^n = (1, 0, 6, 7, 6, 2, 3, 5) \]
\[ u' = (1, 0, 6, 2, 3, 5) \in \Sigma_5^6. \] The decoding process can be shown as the following:

1) \[ u'' = (1, 6, 3), \ u' = (0, 7, 2, 5), \text{ and } \psi(\phi(u'') \circ \phi(u')) = (0, 0, 0, 1, 0, 0, 1, 1). \] Thus, \( a = 3, b = 0 \) and \( c = 6. \)

2) \[ u'' = (1, 6, 3), \ u' = (0, 2, 5). \ \psi(\phi(u'') \circ \phi(u')) = (0, 0, 0, 1, 1, 1) \text{ and } w = 3. \] We can get that \( \Delta = 4 < 2w \) and it is even. It belongs to Case 2.2 in the Proof of Theorem 6 and we insert two \( 0s \) on the left of the second 1 from the right due to \( R_1 = 2. \) Thus, we can recover \( \psi(\phi(u'') \circ \phi(u')) = (0, 0, 0, 1, 0, 0, 1, 1). \)

3) Due to \( b = 0 \) and \( c = 6, \) the value of deleted symbol in \( u'' \) and \( u' \) are 6 and 7, respectively.

4) From the second step, we can get \( \psi(\phi(u'') \circ \phi(u')) = (1, 1, 0, 0). \) Since \( u'' = (1, 6, 3) \) can be recovered via the decoding process for Tenengolts code \( C_T(a, b, n). \) Also, \( u'' = (1, 0, 6, 2, 3, 5) \) can be recovered through the same way.

Therefore, we can recover \( u = (1, 0, 6, 7, 6, 2, 3, 5). \)

### IV. Non-Binary 2-Burst-Error-Correcting Codes

In this section, we propose a non-binary code correcting a burst of at most 2 deletions with redundancy \( \log n + O(\log q \log \log n), \) for even \( q. \) As discussed previously, we begin by describing a simple mapping that converts the problem of correcting deletions of \( q \)-ary symbols from sequences of length \( n \) to the problem of correcting deletions of symbols contained within a binary matrix comprised of \( \lceil \log q \rceil \) rows and \( n \) columns. Our approach will be to encode the first row of this matrix using a pattern length limited (PLL) Levenshtein code, described in Section IV-B, which provides a range for the position of the error. The remaining rows of the matrix are encoded using a \( P \)-bounded Levenshtein code, presented in Subsection IV-C, which can correct a burst of at most 2 deletions given that the deletion is known to occur within a specific range of positions. The overall construction for correcting a burst of at most two deletions and its redundancy is presented in Subsection IV-D.

#### A. Mapping From Non-Binary to Binary

For a non-binary sequence \( u \in \Sigma^n_q, \) for even \( q, \) define the binary representation matrix \( A(u) \in \Sigma_2^{\lceil \log q \rceil \times n} \) as

\[
A(u) = \begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_{\lceil \log q \rceil}
\end{bmatrix} = \begin{bmatrix}
x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\
x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
x_{\lceil \log q \rceil,1} & x_{\lceil \log q \rceil,2} & \cdots & x_{\lceil \log q \rceil,n}
\end{bmatrix}
\]

Let \( A(u)_i \) denote the \( i \)-th row of \( A(u) \), where \( A(u)_i = x_i. \) The \( j \)-th column of \( A(u) \) is the binary representation of \( u_j \), i.e., \( u_j = \left[ x_{1,j}, x_{2,j}, \ldots, x_{\lceil \log q \rceil,j} \right]^T \), where \( x_{i,j} \) is the least significant bit (LSB). Therefore, the non-binary sequence \( u \) is converted to a binary matrix with \( \lceil \log q \rceil \) rows and \( n \) columns. It is straightforward to observe that a burst of at most 2 deletions in \( u \) corresponds to the deletion of at most 2 adjacent columns in \( A(u) \).

#### B. Pattern Length Limited (PLL) Levenshtein Code

Let \( u \in \Sigma^n_q \) be the input of the channel and \( v \in \Sigma^n_q \) the output, obtained by deleting at most two adjacent symbols from \( u \) starting in position \( n. \) As stated above, \( A(v) \) is obtained from \( A(u) \) by deleting at most two adjacent columns starting with column \( n. \) Let \( A(x) = A(u)_1, y = A(u)_2. \) The first step in constructing a non-binary code is to identify the (approximate) position \( i \) of the deletion by finding \( x \) from \( y. \) Suppose \( x \) is a codeword of a code that can correct a burst of at most \( 2 \) deletions. Hence, given \( y, \) we can find \( x. \) The following claim shows that we can then find the maximal substring with period \( 2 \) that contains the position \( i. \) For example, suppose the underlined bits in \( x = 11010111 \) are deleted and we receive \( y = 110111. \) We will then know that the deletion occurs in the underlined substring in \( x = 11010111, \) which has period \( 2. \)

**Claim 1:** For binary strings \( x \in \Sigma^n_q, \ y \in \Sigma^n_q, \) suppose \( y \) is obtained from \( x \) by deleting either \( x_i \) or \( x_{i+1} \) \( x \in \Sigma^n_q. \) Let \( x = vuvz \) such that \( w = x[j,k] \) is the maximal substring of period \( 2 \) containing \( x_i, \) i.e., \( j \leq i \leq k. \) Then, \( j, k \) can be uniquely identified given \( x \) and \( y. \)

**Proof:** Let \( D_h(x) \) be a string obtained from \( x \) by deleting \( x_h \) and \( D_{h+1}(x) \) be the string obtained from \( x \) by deleting \( x_h, x_{h+1}. \)

Note that we can consider the cases in which \( x_i \) is deleted or \( x_{i+1} \) is deleted separately, as the number of deleted symbols is determined from the length of \( y. \) First, suppose that \( x_i \) is deleted. Let \( H = \{ h : D_h(x) = y \}, \) i.e., the set of indices \( h \) such that deleting \( x_h \) from \( x \) results in \( y. \) Clearly, \( H \) can be computed from \( x \) and \( y. \) For all \( h \in H, \) we have \( x_{[1,h-1]} = y_{[1,h-1]}, x_{[h+1,n]} = y_{[h+1,n-1]} \). Let \( h' = \min H, h'' = \max H. \) Then \( x_{[1,h''-1]} = y_{[1,h''-1]}, x_{[h'+1,n]} = y_{[h'+1,n-1]} \). It follows that \( x_{[h'',h'+1]} = y_{[h'',h'+1]} \) and \( x_{[h'+1,n]} = y_{[h'+1,n-1]} \). Hence, \( x_{[h'',h'+1]} = x_{[h'+1,n-1]} \). So \( x_{[h',h'']} \) is a burst and thus a substring of period \( 2 \) that contains \( x_i. \) Extending this substring with period \( 2 \) to make it maximal produces \( w. \)

Similarly, when two symbols are deleted, by setting \( H = \{ h : D_{h+1}(x) = y \} \) and defining \( h' \) and \( h'' \) as above, we find \( x_{[h', h''-1]} = x_{[h'+2, h''-1]} \), again implying that \( x_{[h', h''+1]} \) has period \( 2. \) Extending this substring of period \( 2 \) maximal produces \( w. \)

Therefore, in order to narrow the range of possible deletion positions when a burst of length at most \( 2 \) occurs, we restrict the length of the patterns with period \( 2. \) It was proven in [8] that the redundancy of the code \( \{ x \in \Sigma^n_q : L(x, 1) \leq \lceil \log n \rceil + 1 \} \) (\( L(x, 1) \) denotes the length of the longest run in \( x \)) asymptotically approaches \( \log(e)/4 \approx 0.36. \) This idea is followed in [16] where it was shown that the redundancy of the code \( \{ x \in \Sigma^n_q : L(x, 2) \leq \lceil \log n \rceil + 2 \} \) is at most 0.36 bits. Define the set of binary sequences of length \( n \) where any substring with period \( 2 \) has length at most \( r \) as \( \text{PLL}(r, n). \) Here, we provide an explicit encoding algorithm to construct the code \( \{ x : L(x, 2) \leq \lceil \log n \rceil + 5 \} \) with redundancy of
Algorithm 1 Pattern Length Limited Encoding

Input: Sequence \( x \in \Sigma_2^n \)

Output: Encoded sequence \( y \in \text{PLL}(\lfloor \log n \rfloor + 5, n + 2) \)

\[
y \leftarrow (x_1, x_2, \ldots, x_n, 1, 0), \quad i \leftarrow 1, \quad n' \leftarrow n.
\]

while \( i \leq n' - \lfloor \log n \rfloor - 3 \) do

if \( L(y[i, i + \lfloor \log n \rfloor + 5], 2) = \lfloor \log n \rfloor + 6 \) then

Delete \( y[i, i + \lfloor \log n \rfloor + 4] \) from \( y \) and append

\[
(0, y[i + 1], b(i), 11) \text{ to the end of } y.
\]

\( n' \leftarrow n' - \lfloor \log n \rfloor - 5, \quad i \leftarrow 1 \)

else

\( i \leftarrow i + 1 \)

end

We can then construct pattern length limited Levenshtein codes in which the ambiguity of the position of the deletion is restricted to an interval of length \( \lfloor \log n \rfloor + 5 \).

Lemma 2: For integer \( n \), there exists integer \( a \in [2n] \) such that the pattern length limited Levenshtein code

\[
\text{PLL}-C_L(a, [\lfloor \log n \rfloor + 5, n]) = C_L(a, n) \cap \text{PLL}([\lfloor \log n \rfloor + 5, n]).
\]

has redundancy at most \( \log n + 3 \).

Proof: By Lemma 1, \( \lfloor \text{PLL}([\lfloor \log n \rfloor + 5, n]) \rfloor \geq \lfloor \text{PLL}([\lfloor \log (n - 2) \rfloor + 5, n]) \rfloor \geq 2^{n-2} \). Hence, there exists some value of \( a \) such that \( \lfloor \text{PLL}-C_L(a, [\lfloor \log n \rfloor + 5, n]) \rfloor \geq 2^{n-2} / (2n) \).

Note that at this point, we can use the code \( \text{PLL}-C_L \) to correct the burst of length at most 2, which occurs in the first row of matrix \( A(u) \). In addition, due to the fact that we have limited the length of substrings of period two in our codewords, it follows that we can determine the range of where the burst of deletions has occurred in the remaining rows to within \( \log n + 5 \) positions. The code described in the next subsection leverages this information to correct the burst of deletions of length at most 2 in the remaining rows. We note that a similar construction appears in [8] for a code that can correct a single deletion given its approximate location. The key difference between their result and ours is that the code described in the next subsection can correct a burst of length at most two deletions given its approximate location.

C. P-Bounded Codes for Correcting a Burst of at Most 2 Deletions

Next we show that the \( P \)-bounded Levenshtein code defined below can make use of the information obtained from the first row of \( A(u) \) to correct the deletions in the remaining rows.

Theorem 8: For integers \( c, d \in [2P + 2] \) and \( d \in [3] \), the \( P \)-bounded Levenshtein code

\[
C_L(c, d, P, n) = \{ x \in \Sigma_2^n : \text{VT}(\psi(x)) \equiv c \text{ mod } 2P + 2, \quad \text{wt}(\psi(x)) = d \text{ mod } 3 \}
\]

can correct a burst of at most 2 deletions, if the position \( i \) of the first deleted symbol in \( x \) is known to lie within an interval of length \( P \). More precisely, \( i \) can be uniquely determined provided \( m \) is known where \( i \in [m, m + P - 1] \). Furthermore, there exist choices for \( c \) and \( d \) such that the redundancy of the code is at most \( \log(P + 1) + \log 6 \).

Proof: For a codeword \( x \), let \( x' \) be obtained from \( x \) after a deletion of a single symbol or two adjacent symbols. Also, let \( y = \psi(x) \) and \( y' = \psi(x') \). Further, let \( \Delta \equiv \text{VT}(y) - \text{VT}(y') \) (mod 2P + 2) and \( \Delta_w = \text{wt}(y) - \text{wt}(y') \). We can determine the number of deleted bits based on the length of \( x' \). We show that from \( y' \) we can recover \( y \), and then recover \( x \), since \( \psi \) is invertible.

Similar to the proof of Theorem 6, the error in \( x \) can affect \( y \) in the following ways:

- If the first one or two symbols of \( x \) are deleted, then the first one or two symbols of \( y \) are deleted, respectively.
- If \( x_i, i \in \{2, 3, \ldots, n\} \) is deleted, then \( y_{i-1} \) and \( y_i \) is replaced by \( y_{i-1} + y_i \) and \( y_i \). The possible error patterns are: 00 -> 0, 01 -> 1, 10 -> 1, 11 -> 0. That is, a 0 is deleted from
position $i-1$ or $i$, or a 11 in position $i-1$ is replaced with 0.

- If $x_i x_{i+1}, i \in \{2,3,\ldots,n-1\}$ are deleted, then $y_{i-1} y_i y_{i+1}$ is replaced by $y_{i-1} \oplus y_i \oplus y_{i+1}$. The error patterns are: 000 → 001 → 100 → 101 → 011 → 100 → 1,010 → 1,011 → 0,100 → 1,010 → 1,011 → 1. That is a 00, a 11, two 1s around a 0, or two 0s around a 1 are deleted.

From this point on, we will consider the errors in $y$. The definition of altered bits is the same as the proof of Theorem 6. Let $p$ denote the position of the first altered bit in $y$. Furthermore, let $R_1, L_1$ be the number of 1s on the right of the altered bits and the left of the altered bits, respectively. Similarly, define $R_0, L_0$. Let $w$ denote the weight of $y'$. From the previous discussion, we have $p \in [\max(m-1,1), m+P-1]$. The theorem is proved by showing that a contradiction arises if we assume there exist two different indices $p, p' \in [\max(m-1,1), m+P-1]$, both of which satisfy $\Delta \equiv \text{VT}(y) - \text{VT}(y') \mod (2P+2)$. Thus, we say that there is a unique value of the index $p$ satisfying the equation with the knowledge of $p \in [\max(m-1,1), m+P-1]$ and we can recover $y$ in each case.

1.1) If a 0 is deleted, we have $\Delta_w = 0$. Then $\Delta \equiv R_1 \mod (2P+2)$ and $p \in [i-1, i]$ where $i \in [m, m+P-1]$. Let $r_1$ be the number of 1s on the right of the position $m + P - 1$ in $y$, exclusive. Then, $r_1 \leq R_1 \leq r_1 + P$. Note that $r_1$ equals the number of 1s on the right of the position $m + P - 2$ in $y'$ and thus it is known. Let another true value of $R'_1$ with $R'_1 < R_1$ satisfy the equation $\Delta \equiv R'_1 \mod (2P+2)$. Hence, $R'_1 \equiv R_1 \mod (2P+2)$. But this is impossible as $0 \leq R_1 - R'_1 \leq R_1 - r_1 + P$. Hence, there is a unique value of $R_1$ satisfying the equation and we can recover $y$.

1.2) If a 1 is deleted, we have $\Delta_w = 1$. This can only be the first element of $y$. We prepend 1 to $y$.

1.3) If an 11 is replaced with a 0, we have $\Delta_w = 2$. Then $\Delta \equiv 2p + 1 + R_1 \mod (2P+2)$ and $p \in [m-1, m+P-2]$. Suppose on the contrary that $p' > p$ denotes the location of the deletion where $p'$ satisfies the equation $\Delta \equiv 2p' + 1 + R'_1 \mod (2P+2)$. Hence, $2(p' - p) \equiv R_1 - R'_1 \mod (2P+2)$. But this is impossible as $0 \leq R_1 - R'_1 \leq p - 1$. Hence, there is a unique value of $p$ satisfying the equation and we can again recover $y$.

Next, we consider $|y'| = |y| - 2$: The following cases can be distinguished based on the change in $\Delta_w$ and the parity of $\Delta$.

2.1) If 010 → 1 occurs, we have $\Delta_w = 0$. Then $\Delta \equiv 2R_1 + 1 \mod (2P+2)$ and $p \in [m-1, m+P-2]$. Let another true value of $R'_1$ with $R'_1 < R_1$ satisfy the equation $\Delta \equiv 2R'_1 + 1 \mod (2P+2)$. Hence, $R_1 \equiv R'_1 \mod (P+1)$. But this is impossible as $0 \leq R_1 - R'_1 \leq P - 1$. Hence, there is a unique value of $R_1$ satisfying the equation and we can recover $y$.

2.2) If 00 is deleted, we have $\Delta_w = 0$. Then $\Delta \equiv 2R_1 \mod (2P+2)$ and $p = i - 1 \in [m-1, m+P-2]$ or $p = i \in [m, m+P-1]$. Note that in this case, $\Delta$ is even whereas in Case 2.1, $\Delta$ was odd. It helps us to distinguish Case 2.1 and Case 2.2 since mod2P + 2 does not change the parity. The proof of this item is the same as in Case 2.1 to show there is a unique value of $R_1$ satisfying the equation.

2.3) If 10 or 01 is deleted from the beginning, we have $\Delta_w = 1$. Then $\Delta \equiv 2w + 1 \mod (2P+2)$ or $\Delta \equiv 2w + 2 \mod (2P+2)$, respectively. When $\Delta_w = 1$, prepend 10 or 01 depending on the parity of $\Delta$.

2.4) If 11 is deleted, we have $\Delta_w = 2$. Then $\Delta \equiv 2p + 1 + 2R_1 \mod (2P+2)$ and $p = i - 1 \in [m-1, m+P-2]$ or $p = i \in [m, m+P-1]$. Suppose, on the contrary there exists a $p' > p$ that satisfies the equation $\Delta \equiv 2p' + 1 + 2R'_1 \mod (2P+2)$. Hence, $p' - p \equiv R_1 - R'_1 \mod (2P+1)$. But since $0 \leq R_1 - R'_1 \leq p' - p \leq P - 1$, the only possible ambiguous pattern is that the symbols between $p$ and $p'$ are all 1. Therefore, the two sequences that result from inserting 11 in position $p$ or in position $p'$ are equivalent. Hence, there is a unique value of $p$ satisfying the equation and we can recover $y$.

2.5) If 101 → 0 occurs, we have $\Delta_w = 2$. Then $\Delta \equiv 2p + 2 + 2R_1 \mod (2P+2)$ and $p = i - 1 \in [m-1, m+P-2]$. Note that in this case $\Delta$ is even whereas in the previous case it was odd. The same as Case 2.4, the only possible ambiguous pattern is that the symbols between $p$ and $p'$ are all 1. Further, since the error pattern in this case is 101 → 0, hence this ambiguous pattern is impossible.

The redundancy follows from the fact that there are 6(P+1) options for choosing $c,d$.

As mentioned before, a similar construction shows in [8] for a code that can correct a single deletion given its approximate location. The difference of construction between our $P$-bounded code with their result is that the parity constraint is now mod3 rather than mod2.

The proof of our construction follows from Theorem 6 via the case-by-case discussion. The codes in [8] identify the change in the syndrome in a substring of bounded length. We also present the proof for our $P$-bounded code by checking the change in the syndrome based on runs in a substring of bounded length in [17].

D. Construction for Correcting a Burst of at Most 2 Deletions

Theorem 9: Let $q$ be an even integer. For all $a \in [2[\lceil \log n \rceil + 6]]$ and $d_i \in [\lceil 3 \rceil]$, $c_{2B}(a, c_i, d_i, P, n)$

$$
\begin{align*}
&= \left\{ u \in \Sigma_q^n : A(u)_{1} \in P\mathcal{L}C_L(a, \lceil \log n \rceil + 5, n), \\
&\quad A(u)_{1} \subseteq \mathcal{C}_L(c_i, d_i, \lceil \log n \rceil + 5, n), \forall i \in [2, \lceil \log q \rceil] \right\}
\end{align*}
$$
is a 2-burst-error-correcting code. Furthermore, there exists choices for \( a, c_1, d_1 \) such that the redundancy is at most \( \log n + \log q (\log (\log n) + 6) + \log 6 \) + 3.

**Proof:** If a burst of at most 2 deletions occurs in \( u \in C_{2B}(a, c_1, d_1, P, n) \), at most 2 adjacent columns are deleted in the binary matrix \( A(u) \). The decoder for \( PLL_{C_L}(a, \log n + 5, n) \) can insert the deleted bits into \( A(u)_1 \). Since any substring with period 2 in \( A(u_1) \) has length at most \( \log n + 5 \), we can find an interval of length at most \( \log n + 5 \) in which the deletion has occurred. Then, \( C_L(c_2, d_2, \log n + 5, n) \) can correct the deletions in the remaining \( \log q - 1 \) rows of \( A(u) \).

We now bound the redundancy. Since \( q \) is even, the LSBs in the matrix representation, i.e., \( A(u)_1 \), are free to take any values, regardless of other bits. (If \( q \) was not even, for some values of bits in higher positions, the LSB could not be 1.) Hence, we can use Lemma 2, which implies that there exists a value of \( a \) such that the number of possibilities for \( A(u)_1 \) in the code \( C_{2B} \) is at least \( 2^n/8(n) \). The number of unrestricted possibilities for \( (A(u)_i)_{i=2}^{\log q} \) is \( \frac{q^a}{2^n} \). The total number of possibilities for \( c_1 \) and \( d_1 \) is \( (6(\log n + 6))^{\log q - 1} \). Hence, the size of the code \( C_{2B} \) is at least

\[
\frac{2^n}{8n} \times \frac{(q/2)^n}{(6(\log n + 6))^{\log q - 1}} \geq \frac{q^n}{8n(6(\log n + 6))^{\log q}}.
\]

Taking the log of the denominator of the right side completes the proof.

The code has redundancy \( \log n + O(\log q \log \log n) \) for even \( q \). We note that a lower bound on the redundancy of the \( q \)-ary code correcting a burst of 2 deletions, which is also a lower bound for the redundancy of correcting a burst of at most 2 deletions, is \( \log n + O(\log q) \) in Theorem 3.

**Remark:** To get the systematic construction of the non-binary code for correcting a burst of at most 2 deletions, the encoding and decoding process can be outlined as the following:

- **Encoding:** Suppose the syndrome of Levenshtein code \( C_L(a, n) \) is \( H_{cor, 2}(x) \). The \( i \)-th row of the binary representation matrix \( A(u) \) can be encoded into \( (A(u)_1, 0, 0, 1, H_{cor, 2}(A(u)_i)) \) of length \( N \).
- **Decoding:** Denote the length of received sequence \( u' \) as \( N' \) and let \( m = n + 3 - (N - N') \). First, we need to determine the value of \( A(u')_{1,m} \), where \( A(u')_{i,m} \) denote the element in the 1st row and \( m \)-th column of the matrix \( A(u') \):
  1. If \( A(u')_{1,m} = 1 \), this means the deletion occurs in information bits because the 1 in the protecting bits has shifted \( N - N' \) positions to the left. Then, \( A(u)_1 \) can be recovered by their corresponding syndromes.
  2. If \( A(u')_{1,m} = 0 \), this means the deletion occurs in the syndrome part because the 1 in the protecting bits does not shift. Then, we can directly get \( u \).

The total redundancy of this systematic construction is \( \log q \log n + O(\log q) \), which is much larger than our construction proposed in this section. As will be discussed in Section VII, the design of a systematic encoding scheme for a low-redundancy non-binary 2-burst-error-correcting code remains a direction for future work.

V. NON-BINARY \( t \)-BURST-ERROR-CORRECTING CODES

In this section, we construct non-binary \( t \)-burst-error-correcting codes. We will employ the same mapping from non-binary to binary symbols that was used in the previous section whereby we will interpret our length \( n \) codewords as \( [\log q] \times n \) binary matrices. In this setting, rather than setting the first row of this matrix to be a binary code capable of correcting a burst of at most 2 deletions, we will instead set the first row of our codeword matrix to be a code capable of correcting a burst of at most \( t \) deletions. To this end, we will leverage the construction from [9], as described in detail in Section V-A.

Suppose that a burst of length at most \( t \) occurs to one of our codewords under the assumption that the first row of each of our codewords belongs to a code capable of correcting a burst of at most \( t \) deletions. If we also require that the first row of our codeword matrix is a so-called \((w, \delta)\)-dense string, then it can be shown that we can approximately determine the location of the burst deletions (to within roughly \( O(\log n) \) positions). It then remains to correct the deletions in each of the remaining rows using this knowledge. A straightforward approach is to introduce a set of roughly \( \log n \) parity constraints, which implies the redundancy of this naive approach is much larger than \( \log n \). A better approach, which requires only \( \log n + O(\log q \log \log n) \) bits of redundancy, is to first partition each row of our codeword matrix into blocks of size \( O(\log n) \) and then for each row to enforce two parity constraints on the syndrome of each of these blocks. This approach requires only at most \( O(\log \log n) \) additional bits of redundancy for each row of the codeword matrix, which is significantly less than the naive approach. This portion of the construction is explained in more detail in Section V-B.

A. Locating the Burst of Deletions

In this subsection, we utilize the code in Construction 1 from [9] to approximately determine the location of a burst of at most \( t \) deletions (to within roughly \( O(\log n) \) positions). Furthermore, we provide an explicit encoding method which is not given in [9], for the key step of this construction: encoding the binary sequence into a so-called \((w, \delta)\)-dense string.

Denote the indicator vector of the pattern \( w \) as \( I_w \).

\[
I_w(x)_i = \begin{cases} 1, & \text{if } x_{i,i+|w|-1} = w \\ 0, & \text{otherwise} \end{cases} \quad (7)
\]

Let \( n_w \) be the number of ones in \( I_w \) and define \( \alpha_w(x) \) to be a vector of length \( n_w + 1 \) whose \( i \)-th entry is the distance between positions of the \( i \)-th and \((i + 1)\)-th 1 in the string (1, 1, \ldots, 1).

Fix the pattern \( w = 0^t1^t \) and \( \delta = t2^t+1 \log n \). We now introduce the set of \((w, \delta)\)-dense strings:

\[
D_{w, \delta}(n) = \{ x \in \Sigma^2_n : \alpha_w(x)_i \leq \delta, \forall i \in [\alpha_w(x)] \}.
\]

Since the length of \( w \) is \( 2t \) and its occurrences are non-overlapping, every component in \( \alpha_w(x) \) has value at least \( 2t \), except \( \alpha_w(x)_1 \). Furthermore, the number \( n_w \) of patterns in \( x \) is at most \( n_w \leq \frac{|w|}{2t} \).
Lemma 3 [9, Lemma 1]: For any \( n \geq 5 \), the number of \((w, \delta)\)-dense strings of length \( n \) is at least
\[
|D_{w, \delta}(n)| \geq 2^{n^2(1-n^{-1-\log \varepsilon})} \geq 2^{n^2-1}
\]

Lemma 4 [9, Lemma 2]: For any integers \( c_0 \in [4]\) and \( c_1 \in [2n] \), define the code
\[
C_{loc}(c_0, c_1, n) = \{ x \in D_{w, \delta}(n) : n_w(x) \equiv c_0 \pmod{4}, \forall \alpha \in \alpha_w(\varepsilon) \equiv c_1 \pmod{2n} \}.
\]

Let \( x \in C_{loc}(c_0, c_1, n) \) and \( x' \in D_{\leq t}(x) \). Given \( x' \), \( c_0 \) and \( c_1 \), one can find an interval \( L \subseteq [n] \) of length at most \( \delta = 2^{2t+1}[\log n] \) such that \( x' = (x_{[t2^{-t}t+1]}, x_{[t2^{-t}t+n]}(m+1)) \) for some \( m \in L \), where \( t = \lfloor |x| - |x'| \rfloor \). Furthermore, there exist choices for \( c_0 \) and \( c_1 \) such that the redundancy of the code is at most \( \log n + 4 \).

Next, we will provide an explicit construction for encoding binary sequences into \((w, \delta)\)-dense strings. The key idea of the encoding is to identify a substring of length \( \delta - 2t \) in the input string \( x \) that does not include a pattern \( w \), and each such substring will be removed from \( x \). Then, a compressed version of the removed substring is appended to the end of the input string along with a pattern. We first demonstrate that all substrings of length \( \delta - 2t \) without a pattern can be compressed.

Proposition 1: Let \( S \) be the set of strings of length \( \delta - 2t \) that do not contain a pattern \( w \). Then, every string \( s \in S \) can be compressed into a string \( g(s) \in \Sigma_{2}[<\log n]-6t-2 \) if \( n \) is large enough, where function \( g \) is an invertible map such that \( g \) and \( g^{-1} \) can be computed in \( O(\delta) \) time.

The compression works as follows. Split the string \( s \) into \( (2t)[\log n] - 1 \) substrings with each length of \( 2t \). Each substring can be represented by a symbol from the alphabet of size \( 2^{2t-1} \) since no substring can be equal to \( 0^t1^t \). In other words, the string \( s \) can be represented by a string \( v \) consisting of \( (2t)[\log n] - 1 \) symbols from the alphabet of size \( 2^{2t-1} \). The number of bits \( n_v \) required to represent \( v \) is
\[
n_v = \left\lceil \log (2^{2t-1} - 2^{2t[\log n] - 1}) \right\rceil
\]
\[
= \left\lceil \log (2^{2t-1} - 2^{2\log n} - \log (2^{2t} - 1)) \right\rceil
\]
\[
\leq \left\lceil \log (1 - 2^{2t-2} - 2^{2\log n}) + (2t)(2^{2t}[\log n]) - \log (2^{2t} - 1) \right\rceil + 2
\]
\[
\leq \log n \log (1/e) + 2 + \delta - 2t - 1
\]
\[
\leq \delta - 1.4[\log n] - 2t + 3
\]
\[
\leq \delta - 6\log n - 6t - 2
\]
where (a) follows from the fact that for the function \( (1/1-x)^x \) is increasing in \( x \) for \( x > 1 \) and \( \lim_{x \to 1} (1/1-x)^x = 1/e \), and (b) follows from \( 0.4[\log n] + 4t \geq 5t \) for large value \( n \).

We first convert the binary string \( s \in S \) to the string \( u \), which is a string of alphabet size \( 2^{2t-1} \) with length \( (2^{2t}[\log n] - 1) \). Then, we transform \( u \) to a binary string with length \( \delta - \log n - 6t - 2 \). This can be achieved by a lookup table. Hence, the overall complexity is \( O(\delta) \) when \( t \) is a constant. Also, this process is reversible and \( g^{-1} \) can also be computed in \( O(\delta) \) time.

Algorithm 2 Pattern-Dense Strings Encoding

**Input:** Sequence \( x \in \Sigma_2^n \)

**Output:** Encoded sequence \( E_{w, \delta}(x) \in D_{w, \delta}(n + 2t) \)

**Initialization:** Let \( z = x0^t1^t \) and \( n = n \).

**Define:** For an integer \( i, b(i) \) is its binary representation with length \( \lfloor \log n \rfloor \).

**Step 1:** If there exists \( i \in [n'] \) such that \( z_{[i, i+\delta-2t-1]} \) does not contain \( 0^t1^t \), go to Step 2 or 3. Else go to Step 4.

1. **Step 2:** If \( i \leq n' - \delta + 2t + 1 \), then delete \( z_{[i, i+\delta-2t-1]} \) from \( z \) and append \( (b(i), g(z_{[i, i+\delta-2t-1]}), 1, 0^t1^t(2^{2t}, 0)) \).

2. **Step 3:** If \( i > n' - \delta + 2t + 1 \), then delete \( z_{[i, n']} \) from \( z \) and append \( (b(i), g(z_{[i, n']}0^t1^t(\delta-n'-2t-1), 1, 0^t1^t(4t-(\delta-n'-1), 0)) \).

**Step 4:** Let \( E_{w, \delta}(x) = \sim z \) and output \( E_{w, \delta}(x) \).

Lemma 5: Given a sequence \( x \in \Sigma_2^n \), Algorithm 2 outputs a pattern-dense string \( E_{w, \delta}(x) \in D_{w, \delta}(n + 2t) \).

**Proof:** Notice that the length of \( E_{w, \delta}(x) \) stays stable during encoding. Each time we delete \( (\delta-2t) \) or \( n'-1 \) bits, we will append a new string with the same length as that of the deleted substring at the end of \( E_{w, \delta}(x) \). Next, we show that the encoding sequence \( E_{w, \delta}(x) \) satisfies the condition that each interval of length \( \delta \) in \( E_{w, \delta}(x) \) contains at least one pattern \( w = 0^t1^t \). First, we know \( n' \) is always the splitting index between the original string and the new appended string. Since \( E_{w, \delta}(x)_{[n'+1, n'+2t]} = 0^t1^t \), the integer \( i \) in Step 3 satisfies \( 2t < i + \delta - n' - 1 < 4t \). Otherwise \( E_{w, \delta}(x)_{[i, i+\delta-2t-1]} \) should contain \( E_{w, \delta}(x)_{[n'+1, n'+2t]} = 0^t1^t \), which contradicts with the if condition in Step 1. Then, the \( 0^t1^t+(\delta-n'-1) \) appended in Step 3 has length \( 4t-i+\delta-n'-1 \) in the range of \([1, 2t]\). Therefore, the start index between the two patterns \( 0^t1^t \) in the appended strings is at most \( \delta - 2t \). Furthermore, for any \( i \in [1, n'] \), if there exists \( j \in [i, i + \delta - 4t] \) such that \( E_{w, \delta}(x)_{[j, j+2t-1]} \neq 0^t1^t \), the corresponding substrings beginning with \( E_{w, \delta}(x)_i \) will be deleted in Step 2 or 3. Hence, each interval of length \( \delta \) in \( E_{w, \delta}(x) \) contains at least one pattern \( 0^t1^t \).

The decoding process of recovering \( x \) from \( E_{w, \delta}(x) \) is given in Algorithm 3.

**B. P-Bounded Code for Correcting a Burst of at Most t Deletions**

Next, we discuss how to correct a burst of at most \( t \) deletions provided we know approximately where the deletions occur. We will make use of a code that was designed in prior work.

**Lemma 6 (c.f. [10]):** There exists a labeling function \( f_t : \Sigma_2^k \rightarrow \Sigma_2 R(t,k) \), where \( R(t,k) \leq O(t^2 \log(k + t)) \). Specifically, for any \( x \in \Sigma_2^k \), there exists a positive integer \( a \leq 2^{\log k + \theta \log k} \) such that \( f_t(x) \neq f_t(x') \mod a \) for all \( x' \in D_{\leq t}(x) \). Therefore, the systematic code
\[
C_{SB}(n,t) = \{ (x, 1, 0^t1^t, 0, a, f_t(x)) \mod a : x \in \Sigma_2^n \}.
\]
Algorithm 3 Pattern-Dense Strings Decoding

\begin{enumerate}
\item \textbf{Input:} Sequence \(E_{w,d}(x) \in D_{w,d}(n + 2t)\)
\item \textbf{Output:} Decoded sequence \(x \in \Sigma_2^n\)
\item \textbf{Initialization:} Let \(x = E_{w,d}(x)\).
\item if \(x_{n+2t} = 0\) then
\begin{itemize}
\item Find the beginning index \(k\) of the first pattern
\item \(w = 0^t1^t\) from the right to the left and let \(l = n + 2t - k\). Then, let \(i\) be the integer representation of \(x_{[n+8t-l-t+1,n+8t-l+t+\log n]}\).
\item Let \(y\) be the sequence obtained by \(g^{-1}(x_{[n+8t-l-t+1,n+8t-l+t+\log n]}+1,n+2t-l-2)\). Next, delete \(x_{[n+8t-l-t+1,n+2t]}\) from \(x\) and insert \(y_{[1,\delta-6t+1]}\) at the location \(i\) of \(x\). Repeat.
\end{itemize}
\item if \(x_{n+2t} = 1\) then
\begin{itemize}
\item Delete \(x_{[n+1,n+2t]}\) and output \(x\).
\end{itemize}
\end{enumerate}

is capable of correcting a burst of at most \(t\) consecutive deletions, where \(a\) and \((f_i(x) \mod a)\) are represented as binary vectors.

Let \(E_{SB}(x) = (a, f_i(x) \mod a)\) denote the non-systematic encoder of \(C_{SB}\) without representing \(a\) and \(f_i(x) \mod a\) as binary vectors. Given an input \(x \in \Sigma_2^t\), it outputs \((a, f_i(x) \mod a)\). \(E_{SB}(x)\) contains two integers, where the first is the integer \(a\) and the second is \(f_i(x) \mod a\).

Our next step is to introduce an additional constraint, which leverages the encoder from \(C_{SB}(n, t)\), that allows us to recover the burst of at most \(t\) deletions provided we know approximately where the burst of deletions occurs. The idea behind the approach is to first compute the non-systematic portion of the code \(C_{SB}(n, t)\) for each block (defined below) of our codewords, and then to protect this information by enforcing two parity constraints (one on the even blocks, the other on the odd blocks). More precisely, we split the sequence \(x\) into two sets \(x_e = \{x_{e,1}, x_{e,2}, \ldots, x_{e,s}\}\) and \(x_o = \{x_{o,1}, x_{o,2}, \ldots, x_{o,s+1}\}\), where \(s = \lceil n/2P \rceil\), \(P = \lceil 2t/2t^2 + 1 \rceil\) and we can append 0s at the last block to make it full length:

- **Even Blocks:** \(x_{e,i} = x_{[2(t-2)P+1,2tP]}\), \(i \in [s]\)
- **Odd Blocks:** \(x_{o,i} = x_{[(2t-1)P+1,2(t+1)P]}\), \(i \in [s-1]\).

For \(i \in [s]\), let \(a_{e,i}\) be the first integer of \(E_{SB}(x_{e,i})\) and similarly let \(a_{o,i}\) be the first integer of \(E_{SB}(x_{o,i})\) for \(i \in [s-1]\). Note that \(x_e\) and \(x_o\) each cover the sequence \(x\) and that any interval of length \(P\) is fully contained in at least one block in \(x_e\) or in \(x_o\). We can use the \(E_{SB}\) to protect each block of length \(2P\), as in the following lemma.

**Lemma 7:** There exists an integer \(a\) where \(a = 2^2 \log P + o(\log P)\) such that for \(d_1, e_1 \in [a]\) and \(d_2, e_2 \in [a]\), the code \(C_{PB}(n, t, P) = \{x \in \Sigma_2^n : \sum_{i=1}^s a_{e,i} \equiv d_1 \mod a, \sum_{i=1}^s (f_i(x_{e,i}) \mod a_{e,i}) = e_1 \mod a, \sum_{i=1}^{s-1} a_{o,i} \equiv d_2 \mod a, \sum_{i=1}^{s-1} (f_i(x_{o,i}) \mod a_{o,i}) = e_2 \mod a\}\) can correct a burst of at most \(t\) deletions with the knowledge of the location of a substring of length \(P\) from which the symbols are deleted. Furthermore, there exist choices for \(d_1, d_2, e_1\) and \(e_2\) such that the redundancy of the code is at most \(8 \log P + o(\log P)\).

**Proof:** The interval of length \(P\) in which the edit has occurred is fully contained in a block of \(x_e\) or in a block of \(x_o\). Without loss of generality, let us assume the former and also assume that the index of this block is \(l\). We can recover all blocks of \(x_e\) except \(x_{e,l}\). The value of \(a_{e,l}\) and \(f_i(x_{e,l}) \mod a_{e,l}\) can be determined by solving the equation \(\sum_{i=1}^s a_{e,i} \equiv d_1 \mod a\) and \(\sum_{i=1}^s (f_i(x_{e,i}) \mod a_{e,i}) = e_1 \mod a\), respectively. Then, by Lemma 6, the block \(x_{e,l}\) can be recovered.

**C. Overall Construction of Non-Binary Code for Correcting at Most \(\text{t Deletions}\)**

In this subsection, we will provide the overall construction of the non-binary code for correcting at most \(t\) deletions.

**Theorem 10:** Let \(q\) be an even integer. Then, there exists an integer \(a = 2^2 \log n + o(\log n)\) such that for all \(c_1 \in [2n]\), \(c_0 \in [4]\), \(d_{i,1}, e_{i,1} \in [a]\) and \(d_{i,2}, e_{i,2} \in [a]\), where \(i \in [\log q]\). The code \(C_{LB}(n)\)

\[ C_{LB}(n) = \{u \in \Sigma_q^n : A(u) \in C_{loc}(n, c_0, c_1)\} \]

\[ A(u) \in C_{PB}(n, t, O(\log n)), \forall i \in [\log q]\]  

\[ C_{LB}(n)\] can correct a burst of at most \(t\) deletions in \(q\)-ary sequences. Furthermore, there exists choices for \(c_0, c_1, d_{i,1}, d_{i,2}, e_{i,1}, e_{i,2}\) such that the redundancy is at most \(\log n + 8 \log q \log \log n\).

**Proof:** If a burst of at most \(t\) deletions occurs in \(u \in C_{LB}(n)\), at most \(t\) adjacent columns are deleted in the binary matrix \(A(u)\). Since the LSB of the binary representation of \(u \in \Sigma_q^n\) can take any value regardless of the other bits, the pattern dense string can be applied to \(A(u)\). The decoder for \(C_{loc}(n, c_0, c_1)\) can insert the deleted bits into \(A(u)\) and we can find an interval of length at most \(O(\log n)\) in which the deletion has occurred. Then, \(C_{PB}(n, t, O(\log n))\) can correct the deletions in all \(\log q\) rows of \(A(u)\) with the positional knowledge.

**Remark:** It is worth noticing that the redundancy \(8 \log P + o(\log P)\) of \(P\)-bounded code for correcting at most \(t\) deletions is higher than that of code correcting at most \(2t\) deletions with \(P = P + O(1)\). Therefore, when \(t = 2\), our construction in Section III is better than the construction for arbitrary \(t\).

**VI. CORRECTING A BURST OF AT MOST \(t\) DELETIONS FOR THE PERMUTATION CODE**

In this section, we construct a family of permutation codes that are capable of correcting a burst of at most \(t\) deletions with redundancy \(\log n + O(\log \log n)\). Our approach is similar in spirit to previous settings considered in this paper whereby we first attempt to approximately locate the burst of deletions, and then we seek to correct the burst. In Section VI-A, we give a construction for the first code, which requires roughly \(\log n\) bits of redundancy, that is based upon using a simple mapping between non-binary and binary symbols. The more difficult task is to design the second code, which corrects the burst of
deletions provided we roughly know its location, given that we want the second code to have redundancy of less than \( \log n \) bits.

As an illustration of this difficulty, suppose we are provided with the sequence \( \pi^t \), which is the result of \( t \) symbols being deleted from the permutation \( \pi = (\pi_1, \ldots, \pi_n) \in S_n \), and suppose that we know roughly where the burst of deletions has occurred, namely in an interval of length \( O(\log n) \). Recall that the constraint in [1] only works for exactly \( t \) deletions and a set of \( t \) related constraints are needed to handle at most \( t \) deletions. One naive approach is to directly split the permutation \( \pi \) into several blocks each with length \( O(\log n) \) and introduce constraints into each block. However, the alphabet size for each block is still \( n \), which would result in the construction of the second code that cannot have less than \( \log n \) bits of redundancy.

In order to avoid this issue, we introduce what is referred to as an “overlapping ranking sequence” for the permutation \( \pi \), which allows us to avoid using codes defined over large alphabets. To make use of the connection between the overlapping ranking sequence and its associated permutation, we design codes that are capable of correcting substring edits of length at most \( 2t \) in Section VI-C. The resulting code, which requires \( O(\log \log n) \) bits of redundancy, is then shown to be capable of correcting a burst of deletions provided that we know its approximate location. The overall construction for correcting a burst of at most \( t \) deletions for the permutation code and its total redundancy are presented in Section VI-D.

### A. Locating the Deletion

In this subsection, we want to identify the location of the burst of deletions to be within an interval of size at most \( O(\log n) \). Our approach will be to first convert each of the permutations in our code to binary sequences of length \( n \) by way of a simple mapping as described in Section VI-A.1. Afterwards, we will introduce some additional constraints (similar to the ones from Section V-A) on the resulting binary sequences that will allow us to obtain the desired localizing code.

1) Mapping From Permutations to Binary Sequences: Define \( b_p: S_n \to \Sigma^n \) as:

\[
b_p(\pi)_i = \begin{cases} 
1, & \text{if } \pi_i > n/2 \\
0, & \text{if } \pi_i \leq n/2
\end{cases}
\]  

(8)

**Example 3:** Suppose \( \pi = (5, 3, 4, 1, 2, 6) \). Then, the corresponding binary sequence is \( b_p(\pi) = (1, 0, 1, 0, 0, 1) \).

The binary sequence \( b_p(\pi) \) after mapping will have an equal number of 0s and 1s when \( n \) is even, and the number of 1s is one more than 0s when \( n \) is odd. For even \( n \), let \( D_e(n) \) be the set of binary sequences of length \( n \) with an equal number of 0s and 1s, and when \( n \) is odd, let \( D_o(n) \) be the set of binary sequences that have one more 1s than 0s. We call sequences in \( D_e(n) \) balanced sequences. The size of the set \( \mathcal{D}_e(n) \) is

\[
|\mathcal{D}_e(n)| = \begin{cases} 
\binom{n}{n/2}, & \text{when } n \text{ is even}, \\
\binom{n}{(n+1)/2}, & \text{when } n \text{ is odd}.
\end{cases}
\]  

(9)

For simplicity, we will focus on the case where \( n \) is even, but similar results also hold for the case where \( n \) is odd.

2) Densifying Binary Sequences by a Fixing Pattern \( w \): Next, we make use of \((w, \delta)\)-dense strings from the set \( \mathcal{D}_{w, \delta}(n) \) [9], which was introduced in Subsection V-A. The only difference in this section is that we increase the value of \( \delta \) to \( \delta = t2^{2t+2} [\log n] \).

The next lemma provides the bound on the size of set \( |\mathcal{D}_e(n)| \). The proof is given in Appendix B.

**Lemma 8:** From Stirling approximation, for all even \( n \geq 2 \), we have

\[
\frac{2^n \sqrt{6}}{\sqrt{\pi(3n + 2)^2}} \leq |\mathcal{D}_e(n)| = \binom{n}{n/2} \leq \frac{2^{n+1}}{\sqrt{\pi(2n + 1)}}.
\]  

**Lemma 9:** For all even \( n \geq 2 \), the number of \((w, \delta)\)-dense strings of length \( n \) among balanced sequences is

\[
|\mathcal{D}_e(n) \cap \mathcal{D}_{w, \delta}(n)| \geq \frac{n}{n/2} - \frac{2^n}{n^{2\log \epsilon - 1}} \geq \frac{2^n \sqrt{6}}{\sqrt{\pi(3n + 2)^2}} - \frac{2^n}{n^{2\log \epsilon - 1}}.
\]  

**Proof:** Similar to the Proof of Lemma 1 in [9], let \( z \in \Sigma^n \) and \( E_i \) be the event that \( z_{[i+i, i+\delta]} \) does not contain the pattern \( w \). The probability of \( E_i \) is

\[
\Pr(E_i) \leq (1 - \frac{1}{2^{2t}}) \frac{n}{n/2} \leq \frac{1}{n^{2\log \epsilon}}
\]  

(10)

where \((a)\) follows from the fact that the function \((1 - 1/x)^x\) is increasing in \( x \) for \( x > 1 \) and \( \lim_{x \to \infty} (1 - 1/x)^x = 1/e \). To bound the probability of the event that \( z \in \Sigma^2 \) is not in \( \mathcal{D}_{w, \delta} \), the union bound yields

\[
\Pr(z \notin \mathcal{D}_{w, \delta}) \leq (n - \delta + 1) \Pr(E_i) \leq \frac{1}{n^{2\log \epsilon - 1}}.
\]  

(11)

Thus,

\[
|\mathcal{D}_e(n) \cap \mathcal{D}_{w, \delta}(n)| \geq |\mathcal{D}_e(n)| - 2^n \cdot \Pr(z \notin \mathcal{D}_{w, \delta}) = \frac{2^n \sqrt{6}}{\sqrt{\pi(3n + 2)^2}} - \frac{2^n}{n^{2\log \epsilon - 1}}.
\]

Since \( \frac{1}{n^{2\log \epsilon - 1}} = o\left(\frac{\sqrt{\pi}}{\sqrt{\pi(n+2)^2}}\right) \), the value of \( |\mathcal{D}_e(n) \cap \mathcal{D}_{w, \delta}(n)| \) is dominated by the first term.

3) Approximately Locating the Deletions:

**Lemma 10:** For integers \( c_0 \in [4] \) and \( c_1 \in [2n] \), the code \( C^p_{loc,n}(n, c_0, c_1) = \{ x \in \Sigma^n : x \in \mathcal{D}_e(n) \cap \mathcal{D}_{w, \delta}(n) \} \),

\[
n_w(x) \equiv c_0 \mod 4, \ \VT(\alpha_w(x)) \equiv c_1 \mod 2n
\]

is capable of locating the burst of deletions to an interval of length at most \( \delta = t2^{2t+2} [\log n] \).

**Proof:** It can be derived from Lemma 4. The only difference is that the binary sequence \( x \) should be in \( \mathcal{D}_e(n) \cap \mathcal{D}_{w, \delta}(n) \).

Note that if a burst of at most \( t \) deletions occurs in the permutation \( \pi \), then the corresponding binary sequence \( b_p(\pi) \) suffers a burst of at most \( t \) deletions at the same location. Thus, the localizing code \( C^p_{loc,n}(n, c_0, c_1) \) can be used to help us determine the location of the deletions in \( \pi \).
Lemma 11: There exist integers $c_0$ and $c_1$ such that the size of the permutation code whose codewords $\pi$ satisfy $b_p(\pi) \in C_{\text{loc}}(n, c_0, c_1)$ is at least $n!/16n$.

Proof: By Lemma 8, Lemma 9 and the pigeonhole principle, there exist values of $c_0$ and $c_1$ such that the size of the resulting permutation code with its corresponding binary mapping sequence $b_p(\pi) \in C_{\text{loc}}(n, c_0, c_1)$ is at least:

$$\frac{|D_e(n) \cap D_{w,s}(n)| \cdot (n/2)!}{4 \cdot 2n} \geq \frac{n!}{8n} \cdot \frac{(n/2)!(\frac{2^m n}{\sqrt{n}} + 1)}{(n/2)!(\frac{n^{1.5-2.5}}{\sqrt{n}})}$$

where the first inequality follows from Lemma 9 and the second inequality follows from the lower bound in Lemma 8 for $n \geq 2$. The final inequality can also be shown to hold for $n \geq 3$. \qed

B. Mapping Permutation Code to the Overlapping Ranking Sequence

In the following, we define a mapping that bears a resemblance to one originally introduced in [1] for the purpose of correcting a burst of deletions when the length of the burst was known. The key difference between their mapping and the one introduced here is that the ranking sequence in [1] was known. The key difference between their mapping and the following arises if we assume that there exist $\pi''$ and $\pi$ such that their corresponding overlapping ranking sequence $p_{t+1}(\pi'')$ and $p_{t+1}(\pi)$ are not identical.

Suppose the deleted symbols from $\pi$ are $\pi_{[t,i+t'-1]}$ and $\pi''$ is the result of inserting these symbols (consecutively) beginning at position $j$. Without loss generality, we assume that $j < i$. Thus, $\pi$ and $\pi''$ can be shown as:

$$\pi = (\pi_{[1]} \pi_{[2]} \pi_{[3]} \ldots \pi_{[i-1]} \pi_{[i]} \pi_{[i+1]} \ldots)$$

$$\pi'' = (\pi_{[1]} \pi_{[2]} \pi_{[3]} \ldots \pi_{[i-1]} \pi_{[i]} \pi_{[i+1]} \ldots)$$

From the definition of $\pi''$, we can have $\pi''_{[i]} = \pi_{k-i'}$ when $i + t' - 1 \geq k \geq j + t'$. To illustrate the changed and unchanged part in $\pi$ and $\pi''$, we denote the unchanged part in both $\pi$ and $\pi''$ as $\pi_{[1]} \pi_{[2]} \pi_{[3]} \ldots \pi_{[i-1]} = (\pi_{[1]} \pi_{[2]} \pi_{[3]} \ldots \pi_{[i-1]}) = (a_1, a_2, a_3, \ldots)$, where $m = i - j$. Further, we use $(x_1, x_2, \ldots, x_t)$ to denote the deleted symbols from $\pi$ and $(x''_1, x''_2, \ldots, x''_t)$ to denote the inserted symbols in $\pi''$, where $(x''_1, x''_2, \ldots, x''_t)$ is a permutation of the deleted symbols $(x_1, x_2, \ldots, x_t)$. Then, $\pi_{[j,i+t'-1]}$ and $\pi''_{[j,i+t'-1]}$ can be rewritten as the following, taking $m > t'$ case as example:

$$\pi_{[j,i+t'-1]} = (a_1, a_2, a_3, \ldots, a_{t'}, a_{t+1}, a_{t+2}, \ldots, a_m, x_1, \ldots, x_t)$$

$$\pi''_{[j,i+t'-1]} = (x''_1, x''_2, \ldots, x''_t, a_1, a_{m-t'}, a_{m-t'+1}, \ldots, a_m)$$

For uniformity of notation, we sometimes denote $x_j$ by $a_{m-j}$ and $x''_j$ by $a_{t-j}$. Let $y_{[1]}, \ldots, y_{[k]} \in \{y_1, y_2, \ldots, y_{i+t'}\}$. For a set $\{y_1, y_2, \ldots, y_k\}$ with distinct elements, we say $a \prec \{y_1, y_2, \ldots, y_k\}$ if $a \preceq y_i$ for all $i \in \{1, \ldots, k\}$, with equality holding for at most one value of $i$.

- Consider the case where $i - j > t$. To guarantee each element in $p_{t+1}(\pi')$ and $p_{t+1}(\pi'')$ are the same, we can have the following two relationships:

  If $a_i \prec [a_i, a_{i+t'}]$ for $1 \leq i \leq m - t'$, then $a_i \prec [a_i, a_{i+t'}]$ (12)

  and if $a_i \prec [a_i, a_{m}]$, for $m - t' < i < m$, then $a_i \prec [a_i, a_{m}]$ (13)

  Let $a^* = \min\{a_1, a_2, a_{m}, a_{m-t'}, \ldots, a_{m-t'+1}, \ldots, a_{m}\}$. Recall that all elements are distinct.

  Suppose there exists some $1 \leq i \leq m$ such that $a^* = a_i$. Note that elements $[a_i, a_{i+t'}]$ in $\pi$ and $[a_{i-t'}, a_i]$ in $\pi''$ have the same value in the corresponding overlapping ranking sequence as the number of elements in each of these segments is $t' + 1 \leq t + 1$. On the other hand, if the minimum $a_i$ appears in two different places in a segment with the same starting and end location in $\pi$ and $\pi''$, then the overlapping ranking sequence cannot be
the same. Thus, it implies the contradiction arises when considering the minimum element $a^* \in \{a_1, \ldots, a_m\}$. Hence, there must be some $1 \leq i \leq t'$ such that $a^* = x'_i$. Noting that $r(a_1, \ldots, a_{t+1}) = r(x'_1, \ldots, x'_{t+1}, a_1, \ldots, a_1)$ and we have $a_i < [a_i, a_i + t']$. We now show that

\[
\begin{align*}
& a_i < [a_i, a_i + t'] \\
& a_i + t' < [a_i + t', a_i + 2t'] \\
& a_i + 2t' < [a_i + 2t', a_i + 3t'] \\
& \vdots \\
& a_i + kt' < [a_i + kt', a_i + kt' + t'] \\
& a_i + kt' + t' < [a_i + kt', a_i + m + t']
\end{align*}
\]

where $k$ is the largest integer such that $i + kt' \leq m$. All relations except the last one follow from (12) and the last one follows from (13). The last two relations imply that $a_i + kt' < [a_i + kt', a_i + m + t'] = (a_i + m + 1, \ldots, a_m, x_1, \ldots, x_t)$, which is a contradiction since the minimum among all elements is $a_m$ and $x_i, x_1, \ldots, x_t$ in reversed order in both.

When $i = j$, the elements in $\pi_{[i, i+t]}$ and $\pi''_{[i, i+t]}$ cannot be in the same order due to $\pi'' \neq \pi$. Thus, the overlapping ranking sequence $p_{t+1}$ of $\pi$ and $\pi''$ are not the same in this case.

After mapping the permutation code to the overlapping ranking sequence, the alphabet size of the sequence can be reduced from $n$ to $(t + 1)!$. As a result, we want to correct bursts of deletions in $\pi$ by first recovering the corresponding overlapping ranking sequence $p_{t+1}(\pi)$, and then we will use this information to uniquely determine $\pi$ according to Lemma 12. Recall that for a string $(v_1, v_2, \ldots, v_n)$, we say that $(v_{i_1}, v_{i_2}, \ldots, v_{i_{t-1}})$ is a substring of length $\ell$ that appears in $(v_1, v_2, \ldots, v_n)$ at position $i$.

Claim 2: After deleting a burst of at most $t$ symbols in a permutation $\pi$ resulting in $\pi'$, the corresponding overlapping ranking sequence $p_{t+1}(\pi')$ can be obtained from $p_{t+1}(\pi)$ by at most $t$ consecutive substitutions followed by a burst of at most $t$ deletions.

**Proof:** We can write $\pi$ and $p_{t+1}(\pi)$ as:

\[
\pi = (\pi_1, \pi_2, \ldots, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1}, \pi_{t-1})
\]

\[
p_{t+1}(\pi) = (p_1, p_2, \ldots, p_{t-1}, p_{t-1}, p_{t-1}, p_{t-1}, p_{t-1}, p_{t-1}, p_{t-1}, p_{t-1}, p_{t-1})
\]

Let $\pi' = (\pi_1', \pi_2', \ldots, \pi'_{n-t})$ be the result of deleting $t' \leq t$ consecutive symbols from the permutation $\pi$ and suppose the deleted symbols from $\pi$ are $\pi_{[i, i+t'-1]}$.

Thus, $\pi'$ and $p_{t+1}(\pi')$ can be written as:

\[
\pi' = (\pi_1', \pi_2', \ldots, \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}', \pi_{i-t}')
\]

\[
p_{t+1}(\pi') = (p_1, p_2, \ldots, p_{i-t}, p_{i-t}, p_{i-t}, p_{i-t}, p_{i-t}, p_{i-t}, p_{i-t}, p_{i-t}, p_{i-t})
\]

where we use $p_i$ to denote an unchanged value and $p'_j$ to denote a possibly changed value in $p_{t+1}(\pi')$ compared with $p_{t+1}(\pi)$.

By comparing (16) with (15), we see that there are at most $t$ consecutive substitutions (substituting $(p_{1-}, \ldots, p_{1-})$ by $(p'_{1-}, \ldots, p'_{1-})$) followed by at most $t$ consecutive deletions (deleting $(p_{1-}, \ldots, p_{1-})$).

Based on this observation, we characterize this error pattern as substring edits that replace a substring of length at most $2t$ with another substring of length at most $2t$, which is a more general type of error. Thus, in the next subsection, we will discuss how to construct codes capable of correcting substring edits of length at most $2t$ for recovering the overlapping ranking sequence $p_{t+1}(\pi)$.

C. Correcting Substring Edits of Length at Most $2t$ in the Overlapping Ranking Sequence

In this section, our goal is to construct a code for correcting substring edits of length at most $2t$ in the overlapping ranking sequence $p_{t+1}(\pi)$ based on the systematic binary code capable of correcting up to $t$ edits [18], where each edit is a deletion, insertion, or substitution error.

For $q < \log n$, the basic idea is to consider $q$-ary sequences as a set of $[\log q]$ binary sequences. Unlike the setup in Section V where we only had to correct deletions, for our current setup we want to correct deletions and substitutions. It is straightforward to see that the number of edits for substring edits of length at most $2t$ is also at most $2t$. Thus, we should set the number of edits to $2t$ in our problem.

**Lemma 13** (c.f., [18]): Let $t$ be a constant with respect to $k$. There exist an integer $a \leq 2^{k \log k + o(\log k)}$ and a labeling function $f_{2t} : \Sigma^k \rightarrow \Sigma^2 \Sigma_{=2}(k)$, where $\Sigma_{=2}(k) = O(tk \log k)$ such that $\{ (x, a, f_{2t}(x) \mod a) : x \in \Sigma^2_2 \}$ can correct substring edits of length at most $2t$.

To extend this base code to nonbinary, we can apply the code in Lemma 13 to each row of $A(u)$ for $u \in \Sigma^2_q$. Therefore, we can get the following lemma for $q$-ary sequences.

**Lemma 14:** Let $t$ be a constant with respect to $k$. There exist an integer $a_q \leq 2^{[\log q] (4t \log k + o(\log k))}$ and a labeling function $f_{2t}^q : \Sigma^k_q \rightarrow \Sigma^2_q \Sigma_{=2}(q) \Sigma_{=2}(k)$, where $f_{2t}^q(u) = \sum_{i=1}^{[\log q]} g_{\Sigma_{=2}(q) \Sigma_{=2}(k)}(a_{i-1} f_{2t}(A(u)_i))$ such that $\{ (u, a_q, f_{2t}^q(u) \mod a_q) : u \in \Sigma^2_q \}$ can correct substring edits of length at most $2t$ in $q$-ary sequences.

From Lemma 10, we can narrow the deletion to an $O(\log n)$ interval in the permutation $\pi$. Then, we will make use of Lemma 14 to construct a code for correcting substring edits of length at most $2t$ in the corresponding overlapping ranking sequence $p_{t+1}(\pi)$ with this positional knowledge (We omit the argument $t + 1$ and $\pi$ from $p_{t+1}(\pi)$ and simply write $p$ in the rest of this subsection).

We split the sequence $p$ into two sets $p_e = \{ p_{e,1}, p_{e,2}, \ldots, p_{e,s} \}$ and $p_o = \{ p_{o,1}, p_{o,2}, \ldots, p_{o,s+1} \}$, where $s = n/2P$ and $P = t2^{kt+2[\log n]}$ for even and
odd blocks, respectively, which is the same as the manner in Section V-B. Similarly, let the encoder of the code in Lemma 14 be $E_{1B}(u) = (a_q, f_{2i}^q(u) \mod a_q)$, which is used to protect each block of length $2P$, as in the following lemma.

All of notations are analogous to those from Lemma 7 except that $x$ is replaced with $p$ and $a_{e,i}/a_{o,i}$ are replaced with $a_{e,i}^q/a_{o,i}^q$, where $a_{e,i}^q$ is the first integer of $E_{1B}(p_{e,i})$ for $i \in [s]$ and similarly $a_{o,i}^q$ is the first integer of $E_{1B}(p_{o,i})$ for $i \in [s−1]$.

**Lemma 15:** There exists an integer $a = 2^\log((t+1)!)(4t \log+o(\log P))$ such that for any $d_1, e_1 \in [[a]]$, $d_2, e_2 \in [[a]],$ the code

$$C_{2t}(n, t, P) = \left\{ p \in \Sigma_n^{(t+1)!} : \sum_{i=1}^{s} a_{e,i}^q \mod a \sum_{i=1}^{s} (f_{2i}^q(p_{e,i}) \mod a_{e,i}^q) = e_1 \mod a,$$

$$\sum_{i=1}^{s-1} a_{o,i}^q = d_2 \mod a, \sum_{i=1}^{s-1} (f_{2i}^q(p_{o,i}) \mod a_{o,i}^q) = e_2 \mod a \right\}. $$

can correct one substring edit of length at most $2t$ with the knowledge that the location of the edited symbols is within $P$ consecutive positions. Furthermore, there exist choices for $d_1, d_2$ and $e_1, e_2$ such that the redundancy of the code is at most $4 \log a$.

**D. Overall Construction**

Building on the previous sections, we can present the overall construction of the permutation code for correcting a burst of at most $t$ deletions. First, we apply the code $C_{2t}(n, c_t, c_1)$ to narrow the deletion into an interval of length $t_2^{2t+2} \log n$ with redundancy $\log n + O(1)$. Then, we recover the permutation via $C_{2t}(n, (t+1)!, t_2^{2t+2} \log n)$ for correcting the corresponding overlapping ranking sequence with the positional knowledge of the deletion.

**Theorem 11:** There exists an integer $a = 2^{\log((t+1)!) (4 t \log + o(\log P))}$ such that for all $c_0 \in [[4]], c_1 \in [[2n]], d_1, d_2 \in [[a]]$ and $e_1, e_2 \in [[a]]$, the permutation code $P_{\leq t}(n)$ over $S_n$ is capable of correcting a burst of at most $t$ deletions with the redundancy at most $\log n + 16t \log((t+1)! \log n) + o(\log n)$ bits.

**Proof:** The error-correcting capability of the code has already been discussed. From Lemma 15, the redundancy of the second part in the permutation code $P_{\leq t}(n)$ for correcting the overlapping ranking sequence will be $4 \log a$. Since $P = t_2^{2t+2} \log n$ and $t$ is a constant, we have

$$4 \log a = 16t \log((t+1)! \log n) + o(\log n).$$

Combining with Lemma 11, the code size $|P_{\leq t}(n)|$ is at least

$$|P_{\leq t}(n)| \geq \frac{n!}{16n^a} \geq \frac{n!}{16n \cdot 2^{4 \log a}}.$$

Therefore, the total redundancy of the permutation code $P_{\leq t}(n)$ is at most $\log n + 16t \log((t+1)! \log n) + o(\log n)$.

**Remark:** Given a sequence $u \in \Sigma_n^{(t+1)!}$, denote $B_{2t}(u)$ as the size of the confusable set of the substring edit with length at most $2t$. We have $B_{2t}(u) = (P + 2t)^2(2t + 1)^t$. Then, we can apply the syndrome compression technique introduced in [10] to construct the codes for correcting substring edits of length at most $2t$ with redundancy $16 \log n + o(\log n)$, where $P = O(\log n)$. Thus, the total redundancy of the code $P_{\leq t}(n)$ can be reduced to $\log n + 16 \log n + o(\log n)$. However, since we assume the number of deletions $t$ is constant throughout this paper, we do not need to devote excessive efforts to improve this factor before $\log n$.

**VII. Conclusion**

Motivated by applications to DNA storage, we have constructed non-binary codes capable of correcting bursts of deletions. By considering a variation of the well-known Levenshtein code, we presented a non-binary code capable of correcting bursts of length at most $2t$. We then developed codes capable of correcting bursts of length at most $t$ before turning our attention to burst-error-correcting codes for permutations. Each of the proposed codes in this paper is nearly optimal in terms of the number of redundant bits.

Although in many cases our results improve upon the prior art, there are many avenues for future research:

- **Systematic t-burst-error-correcting codes:** Although the non-binary codes presented in this work were nearly optimal in terms of their redundancy, the proposed codes were non-systematic. As discussed in Section IV, even for the case of $t = 2$, the authors were unaware of a systematic encoding that approaches our results.

- **Codes correcting bursts of edits:** Codes that correct bursts of insertions/deletions/substitutions have applications not only in DNA storage but also in other areas such as in the document exchange problem. Currently, no optimal constructions for this setup have been reported in the open literature.

- **Codes correcting multiple bursts of deletions:** Even for the case of 2 bursts of deletions, there are many different problems of interest. One could consider the setup where the bursts are each of the same length or possibly of different lengths. Additionally, the problem of constructing codes correcting multiple bursts of insertions/deletions/substitutions is another area of future work.

**APPENDIX A**

**Proof of Theorem 3**

In order to make use of this technique, we need a few results related to the set $D_t(u)$, which appear as Claims 3 and 4. In the following, let $N(n, t, i) = \{ u \in \Sigma_n : |D_t(u)| = i \}$.

**Claim 3:** (c.f., [8, Lemma 1]) Let $u \in \Sigma^n_q$ and suppose $u' \in D_t(u)$. Then $|D_t(u')| \geq |D_t(u)|$.

**Claim 4:** For $i \leq n - t + 1$ and $t \leq n$, we have that

$$N(n, t, i) = q^i(q - 1)^{n - i - 1} \binom{n - t}{i - 1}. \tag{4}$$
Proof: We can arrange sequence $u = (u_1, u_2, \ldots, u_n) \in \Sigma_q^n$ into a $t \times \frac{n}{t}$ array $A_q(u)$ as the following

$$A_q(u) = \begin{bmatrix}
    u_1 & u_{t+1} & \cdots & u_{n-t+1} \\
    u_2 & u_{t+2} & \cdots & u_{n-t+2} \\
    \vdots & \vdots & \ddots & \vdots \\
    u_t & u_{2t} & \cdots & u_n
\end{bmatrix}$$

Let $N_r(u_j)$ denote the number of runs in the $j$th row of $A_q(u)$. The size of $t$-burst-deletion ball $|D_t(u)|$ was shown in [5]

$$|D_t(u)| = \left( \sum_{j=1}^{t} N_r(u_j) \right) - t + 1. \quad (17)$$

Then, the problem of counting the number of $q$-ary sequences with length $n$ whose $t$-burst-deletion ball is $i$ is equivalent to counting the number of $q$-ary sequences with length $n$ for which

$$\left( \sum_{j=1}^{t} N_r(u_j) \right) = i + t - 1. \quad (18)$$

The number of $q$-ary sequences of length $n$ with $r$ runs is

$$M(n, r) = q^{(q-1)r-1} \binom{n-1}{r}. \quad (19)$$

Let $R_i = \sum_{j=1}^{t} r_j$. Combining (17), (18) and (19), we have

$$N(n, t, i) = \sum_{R_i = t+i-t}^{t} M \left( \binom{n}{r}, r_1 \right) M \left( \binom{n}{r_2}, r_2 \right) \cdots M \left( \binom{n}{r_t}, r_t \right)
= q^t (q-1)^{i-1} \sum_{R_i = t+i-t}^{t} \left( \binom{n}{r_1} \binom{n-r_1}{r_2} \cdots \binom{n-r_1-r_2}{r_t} \right).
= q^t (q-1)^{i-1} \sum_{R_i = t+i-t}^{t} \left( \binom{n}{r_1} \binom{n-r_1}{r_2} \cdots \binom{n-r_1-r_2}{r_t} \right).
= \left( \binom{n}{r_1} \binom{n-r_1}{r_2} \cdots \binom{n-r_1-r_2}{r_t} \right).
$$

Since $\sum_{j=1}^{t} \frac{n}{t} - r_j = n - (i + t - 1)$ and $\sum_{j=1}^{t} \frac{n}{t} - 1 = n - t$,
using a generalized Vandermonde identity we have

$$N(n, t, i) = q^t (q-1)^{i-1} \binom{n-t}{i-1}. \quad (20)$$

We now proceed to the proof of Theorem 3.

Proof of Theorem 3: We proceed similarly to the method presented in [8]. Let $\mathcal{H}_{q,t,n}$ be the following hypergraph:

$$\mathcal{H}_{q,t,n} = \left( \Sigma_q^{n-t}, \{ D_t(u) : u \in \Sigma_q^n \} \right).$$

It is known [19] that under this setup $|\mathcal{M}_t(n)| \leq \tau^*(\mathcal{H}_{q,t,n})$ where $\tau^*(\mathcal{H}_{q,t,n})$ is the solution to the following linear program:

$$\begin{align*}
    \tau^*(\mathcal{H}_{q,t,n}) = & \min \left\{ \sum_{u' \in \Sigma_q^{n-t}} w(u') \right\} \\
    \text{s.t.} & \sum_{u' \in D_t(u)} w(u') \geq 1, \forall u \in \Sigma_q^n.
\end{align*} \quad (21)$$

Let $w : \Sigma_q^{n-t} \rightarrow \mathbb{R}$ be defined such that $w(u') = \frac{1}{|D_t(u)|}$, $\forall u' \in \Sigma_q^{n-t}$. Clearly $w(u') \geq 0$ for any $u' \in \Sigma_q^{n-t}$. As a result of Claim 3, we have

$$\sum_{u' \in D_t(u)} w(u') = \sum_{u' \in D_t(u)} \frac{1}{|D_t(u)|} \geq \sum_{u' \in D_t(u)} \frac{1}{|D_t(u)|} = 1,$$

so that the function $w$ satisfies both (21) and (22).

Then, according to (20)

$$|\mathcal{M}_t(n)| \leq \sum_{u' \in D_t(u)} \frac{1}{|D_t(u')|}.$$

For $1 \leq i \leq n - t + 1$, denote $N(n, t, i)$ as the size of the set $\{ u \in \Sigma_q^n : |D_t(u)| = i \}$, where $N(n, t, i) = q^t (q-1)^{i-1} \binom{n-t}{i-1}$.

$$\sum_{u' \in D_t(u)} \frac{1}{|D_t(u')|} = \sum_{i=1}^{n-2t+1} \frac{N(n, t, i)}{i} \quad (23)$$

$$= q^t \sum_{i=1}^{n-2t+1} (q-1)^{i-1} \frac{(n-2t)!}{i!(n-2t-i+1)!}$$

$$= \left( \frac{q}{(q-1)(n-2t+1)} \sum_{i=1}^{n-2t+1} (q-1)^i \frac{(n-2t+1)!}{i!(n-2t-i+1)!} \right)$$

$$= \left( \frac{q}{(q-1)(n-2t+1)} \sum_{i=1}^{n-2t+1} (q-1)^i \frac{(n-2t+1)!}{i!(n-2t-i+1)!} \right)$$

$$= q^t \left( \frac{1}{(q-1)(n-2t+1)} \right)$$

where (a) follows from the Binomial theorem. \hfill \Box

APPENDIX B
PROOF OF LEMMA 8

Proof: First, we have

$$\binom{n+\frac{1}{2}}{n} = \frac{n^2 + n + \frac{1}{4}}{n^2 + 2n + 1} \leq \frac{n + \frac{1}{4}}{n + \frac{3}{4}}.$$

Thus,

$$\binom{2n}{n} = \binom{n+\frac{3}{4}}{n+\frac{1}{2}} \leq \left( \frac{2n+1}{n+1} \right) \left( \frac{n+\frac{1}{2}}{n+\frac{3}{4}} \right)$$

which implies $\binom{2n}{n}^{n+\frac{1}{3}}$ is decreasing.

We also have

$$\binom{n+\frac{1}{2}}{n+1} = \frac{n^2 + n + \frac{1}{4}}{n^2 + 2n + 1} \leq \frac{n + \frac{1}{4}}{n + \frac{3}{4}}.$$
Then,
\[
\frac{2n+2}{\binom{2n}{n}} = 4 \frac{n + \frac{1}{2}}{n + 1} \leq 4 \sqrt{\frac{n + \frac{1}{4}}{n + 4}},
\]
which implies \(\binom{2n}{n}\) is increasing.

From
\[
\lim_{n \to \infty} \frac{\sqrt{\pi n} \binom{2n}{n}}{4^n} = 1,
\]
we can show that
\[
\frac{4^n}{\sqrt{\pi} \left(n + \frac{1}{4}\right)} \leq \frac{2n}{n} \leq \frac{4^n}{\sqrt{\pi} \left(n + \frac{1}{4}\right)},
\]
which implies
\[
\frac{2^n \sqrt{6}}{\sqrt{\pi} \left(3n + 2\right)} \leq \frac{n}{n/2} \leq \frac{2^{n+1}}{\sqrt{\pi} \left(2n + 1\right)}.
\]

**REFERENCES**

[1] Y. M. Chee, S. Ling, T. T. Nguyen, V. K. Vu, H. Wei, and X. Zhang, “Burst-deletion-correcting codes for permutations and multi-permutations,” IEEE Trans. Inf. Theory, vol. 66, no. 2, pp. 957–969, Feb. 2020.

[2] S. M. H. T. Yazidi, H. M. Kiah, E. Garcia-Ruiz, J. Ma, H. Zhao, and O. Milenkovic, “DNA-based storage: Trends and methods,” IEEE Trans. Mol. Biol. Multi-Scale Comput., vol. 1, no. 3, pp. 23–248, Sep. 2015.

[3] L. Dolecek and V. Anantharam, “Using Reed–Muller RM (1, m) codes over channels with synchronization and substitution errors,” IEEE Trans. Inf. Theory, vol. 53, no. 4, pp. 1430–1443, Mar. 2007.

[4] R. Venkataramanan, H. Zhang, and K. Ramchandran, “Interactive low-complexity codes for synchronization from deletions and insertions,” in Proc. 46th Annu. Allerton Conf. Commun., Control, Comput. (Allerton), Sep. 2010, pp. 1412–1419.

[5] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” Sov. Phys.-Dokl., vol. 10, no. 8, pp. 707–710, 1966.

[6] H. H. Lee, R. Kallhor, N. Goela, J. Bolot, and G. M. Church, “Terminator-free template-independent enzymatic DNA synthesis for digital information storage,” Nature Commun., vol. 10, no. 1, pp. 1–12, Jun. 2019.

[7] V. Levenshtein, “Asymptotically optimum binary code with correction for losses of one or two adjacent bits,” Problemy Kibernet, vol. 19, pp. 293–298, Jan. 1967.

[8] C. Schoeny, A. Wachtet-Zeh, R. Gabrys, and E. Yaakobi, “Codes correcting a burst of deletions or insertions,” IEEE Trans. Inf. Theory, vol. 63, no. 4, pp. 1971–1985, Apr. 2017.

[9] A. Lenz and N. Polyanskiy, “Optimal codes correcting a burst of deletions of variable length,” in Proc. IEEE Int. Symp. Inf. Theory (ISIT), Jun. 2020.

[10] J. Sima, R. Gabrys, and J. Bruck, “Syndrome compression for optimal redundancy codes,” in Proc. IEEE Int. Symp. Inf. Theory (ISIT), Jun. 2020, pp. 751–756.

[11] Y. Sun, Y. Zhang, and G. Ge, “Improved constructions of permutation and multi-permutation codes correcting a burst of stable deletions,” IEEE Trans. Inf. Theory, vol. 69, no. 7, pp. 4429–4441, Mar. 2023.

[12] R. Gabrys, E. Yaakobi, and O. Milenkovic, “Codes in the Damerau distance for deletion and adjacent transposition correction,” IEEE Trans. Inf. Theory, vol. 64, no. 4, pp. 2550–2570, Apr. 2018.

[13] C. Schoeny, P. Sala, and L. Dolecek, “Novel combinatorial coding results for DNA sequencing and data storage,” in Proc. 51st Asilomar Conf. Signals, Syst., Comput., Oct. 2017, pp. 511–515.

[14] N. J. Sloane, “On single-deletion-correcting codes,” Codes Des., vol. 10, pp. 273–291, May 2002.

[15] G. Tenengolts, “Nonbinary codes, correcting single deletion or insertion (corresp.),” IEEE Trans. Inf. Theory, vol. 17, no. 5, pp. 766–769, Sep. 1984.

[16] Y. M. Chee, H. M. Kiah, A. Vardy, V. K. Vu, and E. Yaakobi, “Coding for racetrack memories,” IEEE Trans. Inf. Theory, vol. 64, no. 11, pp. 7094–7112, Nov. 2018.

Shuiche Wang (Graduate Student Member, IEEE) received the B.Eng. and M.Sc. degrees in information and communication engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2017 and 2020, respectively. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Virginia.

His research interests consist of information theory, coding theory, wireless communications, and DNA data storage.

Yuanyuan Tang (Member, IEEE) received the bachelor’s degree in engineering from the Department of Communication Engineering, Chongqing University, in 2015, and the master’s degree in engineering from the Department of Electronic Engineering, Tsinghua University, in 2018. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Virginia.

His research interests consist of information theory, coding theory, wireless communications, and DNA data storage.

Jin Sima received the B.Eng. and M.Sc. degrees in electronic engineering from Tsinghua University, China, in 2013 and 2016, respectively, and the Ph.D. degree in electrical engineering from the California Institute of Technology (Caltech) in 2022. He is currently a Post-Doctoral Researcher with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His research interests include information and coding theory and their applications, with its applications in data storage systems. He was a recipient of the 2019 IEEE Jack Keil Wolf ISIT Student Paper Award, the 2020–2021 IEEE Communication Society Data Storage Best Paper Award, and the 2022 Caltech Charles Willis Prize for Best Doctoral Thesis.

Ryan Gabrys (Member, IEEE) received the B.S. degree in mathematics and computer science from the University of Illinois at Urbana-Champaign in 2005 and the Ph.D. degree in electrical engineering from the University of California, Los Angeles, in 2014. He is currently a scientist jointly affiliated with the Naval Information Warfare Center, California Institute for Telecommunications and Information Technology (Calit2), University of California, San Diego. His research interests broadly lie in the areas of theoretical computer science and electrical engineering, including coding theory, combinatorics, and communication theory.

Farzad Farnoud (Member, IEEE) received the first M.S. degree in electrical and computer engineering from the University of Toronto in 2008 and the second M.S. degree in mathematics and the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 2012 and 2013, respectively.

He is an Associate Professor with the Department of Electrical and Computer Engineering and the Department of Computer Science, University of Virginia. Previously, he was a Post-Doctoral Scholar with the California Institute of Technology. His research interests include coding for storage, data compression, probabilistic modeling and analysis, and machine learning. He was a recipient of the 2022 Faculty Early Career Development Award (CAREER) from the National Science Foundation, the 2013 Robert T. Chien Memorial Award from the University of Illinois for demonstrating excellence in research in electrical engineering, and the 2014 IEEE Data Storage Best Student Paper Award.