Codes for Synchronization in Channels and Sources with Edits

Mahed Abroshan

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

St Edmund's College June 2019

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This report is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Mahed Abroshan
June 2019

Acknowledgements

I would like to sincerely thank my supervisors Albert Guillén i Fàbregas and Ramji Venkataramanan for all their help and support. I especially thank Albert for giving me freedom to choose my problem and supporting me in every possible way, and Ramji for making himself accessible throughout my studies and always giving me intriguing comments.

In addition, I would like to thank Lara Dolecek and Jossy Sayir with whom I had several insightful discussions. I was fortunate to work with members of Signal processing and communication lab. I am especially thankful to Hadi, Rida, Fergal, Kuan, Ehsan, and Parham for their friendship. I also gratefully acknowledge the generous financial support from the Cambridge Trust and Trinity college.

I would like to express my deepest gratitude and appreciation to my parents, who have supported me in all the stages of my life and to whom I dedicate this work. I am also thankful to my sisters for their care and support. Last but surely not least, I am really grateful that I found Yasaman (the greatest discovery of my PhD), who gave a new color to my life. I am deeply thankful for her love and support.

Abstract

Edit channels are a class of communication channels where the output of the channel is an edited version of the input. The edits are considered to be deletions and insertions. DNA-based data storage system is one of the motivations for this model. This thesis studies various problems related to edit channel and also edit synchronization problem. Varshamov-Tenengolts (VT) codes are first introduced. These codes can correct a single deletion or insertion and have a linear-time decoder. The problem of efficient encoding of non-binary version of VT codes is addressed, where a simple linear-time encoding method to systematically map binary message sequences onto VT codewords is proposed.

Another model that is studied is segmented edit channels, where we have the additional assumption that the channel input sequence is implicitly divided into segments such that at most one edit can occur within a segment. A code construction is proposed for this model based on subsets of VT codes chosen with pre-determined prefixes and/or suffixes. Also an upper bound is derived on the rate of any zero-error code for the segmented edit channel in terms of the segment length. This upper bound shows that the rate scaling of the proposed codes as the segment length increases is the same as that of the maximal code.

Edit synchronization is another problem studied in this thesis. In this model, there are two remote nodes (encoder and decoder), each having a binary sequence. The sequence X, available at the encoder, is the updated sequence and differs from Y (available at the decoder) by a small number of edits. The goal is to construct a message M, to be sent via a one-way error free link, such that the decoder can reconstruct X using M and Y. A coding scheme is devised for this one-way synchronization model. The scheme is based on multiple layers of VT codes combined with off-the-shelf linear error-correcting codes and uses a list decoder.

Motivated by the sequence reconstruction problem from traces in DNA-based storage, the problem of designing codes for the deletion channel when multiple observations (or traces) are available to the decoder is considered. A simple binary and non-binary code is proposed that splits the codeword into blocks and employs a VT code in each block. The availability of multiple traces helps the decoder to identify deletion-free copies of a block, and to avoid mis-synchronization while decoding. The encoding complexity of the proposed scheme is linear in the codeword length; the decoding complexity is linear in the codeword length, and quadratic in the number of deletions and the number of traces. The list decoding technique for the proposed code is also considered.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

There are two fundamental questions in information theory that are addressed by Claude E. Shannon in his seminal 1948 paper [1]. Firstly, what are the conditions under which a reliable communication is possible over a noisy channel. Secondly, how much can an information source be compressed such that a decoder can retrieve the information reliably from the compressed version. Much work has been done to achieve the fundamental limits that are obtained in Shannon's work, as he did not specify efficient schemes to achieve these limits. This thesis studies code design for a specific class of channels called edit channels. Moreover, a source coding problem is studied which is closely related to edit channels. In this document we refer to this model as edit synchronization model.

1.1 Problem definition

Generally, a channel coding problem can be represented as in Figure 1.1. The encoder maps the message m into a codeword X. The channel produces the received sequence Y according to a conditional distribution $P_{Y|X}$. The decoder estimates $\hat{m}$ using Y. The goal is to design a code with small probability of error, which is defined as $\mathsf{Pr}[\hat{m} \neq m]$. In some scenarios we may need zero error probability, in this case, the decoder should always recover m correctly.

The message m belongs to set $\mathcal{M}$, and the codeword X is a length n sequence from some alphabet, for example, the binary alphabet $X \in \{0,1\}^n$. The received sequence Y is not necessarily from the same alphabet set as X nor it is necessarily of the same length.

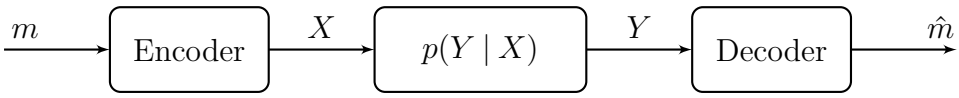

Figure 1.1: Channel Model

The rate of the code is defined as the logarithm of the number of messages over the number of channel uses (in this thesis all logarithms are in base 2):

$$R \triangleq \frac{\log|\mathcal{M}|}{n}. \tag{1.1.1}$$

For a given rate R, there are 2^{nR} messages, which correspond to nR information bits. For a binary code of length n, the redundancy of the code is defined as $n(1-R)$.

Shannon has introduced a quantity called *capacity* (usually denoted with C) for a given communication channel, and showed that reliable communication is not possible with rates greater than the capacity. Also, for any $R < C$ and any $\epsilon > 0$, there exists a code with rate R, such that the error probability of the code is less than ϵ for large enough n (code length). A channel coding scheme attempts to achieve the capacity with an efficient encoding and decoding algorithm.

The zero error capacity was also defined by Shannon [2] as the least upper bound on rates at which it is possible to send information with zero probability of error. In Chapter 2 and 3 of this thesis, we consider zero error codes, whereas in Chapter 4 and 5 a small probability of error is allowed.

1.1.1 Edit channel

The specific type of channels that will be discussed in this thesis are edit channels. In an edit channel, the output sequence is obtained from the input sequence via deletions, insertions, and substitutions. In this thesis, we will only consider edit channels with deletions and insertions. Here is an example of a transmission through an edit channel.

Example 1.1. *Assume that* $X = 1101111$ *is the transmitted codeword, and the output of the channel* $Y = 11111100$*. In this example, the third bit is deleted and two zeros are inserted at the end of* X*.*

Many of the channels that are studied in information theory are *memoryless*. This means that the ith received symbol only depends on the corresponding transmitted symbol (for example Binary Symmetric channel (BSC) is a memoryless channel). However, the edit channel is not memoryless. Each received symbol depends on the

number of deletions and insertions that have occurred up to that point. This is one of the main reasons why designing codes for correcting deletions or insertions is a challenging problem.

The edit distance is an important quantity for code design for edit channels. The edit distance between two sequences X and Y (both from the same alphabet) is defined as the minimum number of insertions and deletions required to obtain Y from X.

A special case of the edit channel is the deletion channel, where there will be only deletions in the received sequence. Let us first formally define this channel. Two main models are considered for the deletion channel.

In the first model, which is a combinatorial model, the channel chooses up to k locations (k is a parameter of the channel) in the transmitted sequence, then deletes the corresponding symbols from the sequence. In other words, $p(Y|X)$ is a distribution with support on the subsequences of X obtained by deleting at most k bits.

There is an alternative model for deletion channel, where a deletion probability p_d is associated with the channel, and each transmitted symbol will be independently deleted with the probability p_d. For instance, if $X = 111$ is transmitted and $p_d = 0.1$, then the probability of receiving $Y = 11$ is $3(0.1)(0.9)^2$.

Similarly, the insertion channel can be defined. The combinatorial insertion channel inserts at most k symbols into the transmitted sequence, $p(Y|X)$ is a distribution with support on the supersequences of X obtained by inserting at most k bits into X. The alternative definition for insertion channel is that the decoder receives each transmitted bit appended with a number of insertions (there can be also zero insertions), in the channel definition we should specify what is the probability of a given substring to be inserted after transmission of a symbol. Various models have been suggested for this. One example is duplication channel, where each symbol with probability p will be duplicated and with probability $1-p$ there will be no insertion.

Now that we defined both insertion and deletion channels, we can define the edit channel as the cascade of a deletion channel and an insertion channel. See the model in [3] for an example of an edit channel. The combinatorial channel can also be defined wherein $p(Y \mid X)$ has only non-zero values where edit distance of X and Y is less than k. In this thesis, the exact model of the channel is defined in each of the chapters separately.

1.1.2 Edit synchronization

The edit synchronization model is an instance of the source coding problem with decoder side information [4]. (Figure 1.2) The encoder represents the sequence X

using a message M from a codebook. The decoder reconstructs $\hat{X}$ using M and a side information sequence Y.

In this setup, assume that the side information Y is an edited version of X. For example, Y can be obtained by a combination of k insertions and deletions from X. The question is how to construct the message M such that the decoder can reconstruct X, using M and Y. We refer to this problem as one-way file synchronization. The motivation is that one can think of X and Y as two copies of a file which are available at two remote nodes and differ by a small number of edits. The goal is to use the communication link between the two nodes to update the file at the second node (the decoder). This problem is studied in Chapter 4 of this thesis where a new class of codes, called multilayer codes are suggested to address the problem.

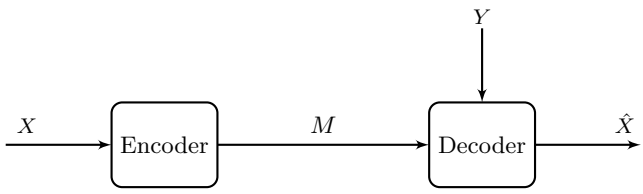

Figure 1.2: Source coding with decoder side information

We denote the rate in file synchronization problem by R_{sync} which is defined as

$$R_{\text{sync}} = \frac{\log|\mathcal{M}|}{n}, \tag{1.1.2}$$

here n is the length of sequence X, and $\mathcal{M}$ is the set of all the messages. We want the rate to be as small as possible. For the case where X and Y are binary sequences, $R_{\text{sync}} = 1$ is equivalent of sending the entire sequence X.

The connection between edit channel coding and edit synchronization:

Here it is shown how one can use a given code for file synchronization problem in the edit channel setting. Assume that an encoding scheme for file synchronization problem assign a message $M \in \mathcal{M}$ to a sequence X, i.e., $M = \mathsf{E}(X)$. And there exists a decoding function which maps Y, the edited version of X, and M to the reconstructed sequence $\hat{X}$, i.e., $\hat{X} = \mathsf{D}(Y, M)$. Now corresponding to each of the messages $M \in \mathcal{M}$ we can define a codebook as follows

$$\mathcal{C}_M = \{X \in \{0,1\}^n \mid \mathsf{E}(X) = M\}. \tag{1.1.3}$$

That is, the code for the edit channel consists of all the sequences that the file synchronization encoder maps to a given message, say M^*. To maximize the rate of the channel, we choose the M^* that has the largest number of sequences mapped to it. With this choice, the size of $\mathcal{C}_{M^*}$ is at least $\frac{2^n}{|\mathcal{M}|}$, therefore the rate (in the binary case) will be at least

$$R \geq 1 - \frac{\log|\mathcal{M}|}{n}. \tag{1.1.4}$$

For the decoding, one can use the same decoder D for communication over the edit channel. Since the decoder knows Y (the edited version) from the output of the channel, and also the message associated with X is known from the codebook.

Notice that a code for edit channel does not necessarily induce a code for file synchronization model. This is because in a channel coding scheme some structure may be assumed for codewords. For example, the codewords can be run limited, which means there is no run longer than a constant number of bits in the codewords. However, in the file synchronization, the encoder should deal with any given sequence and such assumptions are not possible. From this perspective, designing codes for file synchronization is a more difficult task. In addition, from a complexity point of view, the encoding for the edit channel model can be challenging. Finding the codebook, using (1.1.3) for large n is computationally costly, also mapping messages to codewords is challenging and for large n a systematic way for this mapping should be devised. The connection between these two problems is studied in detail in [5].

1.2 Motivation

The traditional motivation for studying edit channels was that this type of error can occur in a communication system due to the lack of synchronization between transmitter and receiver. Moreover, insertions and deletions may happen when reading information stored in magnetic or optical media. Another motivation is that deletion channel with large alphabets can model packet lost in networks like Internet [6, 7].

DNA-based data storage is a new motivation for the study of edit channels. The idea is to use DNA as the medium for storing data instead of conventional methods. The technology for synthesizing and also reading a given DNA sequence already exists. Therefore, for storing data we can first translate the information bits into the nucleotide alphabets $\{A, C, G, T\}$, and then synthesize the resulting DNA sequence. Now reading back data is possible by sequencing the DNA string. This idea recently attracted significant attention [8–11], mainly because this new method allows much greater storage densities than conventional methods. The durability of DNA sequences is

another important factor. Now in the process of synthesizing and sequencing of DNA strings, some errors may occur in the form of insertions, deletions, and substitutions. We can therefore model the storage system as an edit channel.

One of the issues of DNA storage is the cost and time of the process. Nanopore sequencing is an emerging new technology, which seems to be a promising approach to address these problems. Although it produces a significantly larger number of edit errors comparing with the existing methods. This new technology also incited a new line of research in information theory community [12, 13].

The file synchronization problem has a number of applications including file backup (e.g., Dropbox), online editing, and file sharing. If a file is slightly edited, storing or transmitting a new file is wasteful, therefore efficient synchronization techniques are required for updating the file. Most of the works in this area assuming a two-way communication link between the two parties.

The one-way File synchronization model is also interesting to study as a two-way algorithm increases latency and may incur additional overhead. Furthermore, a code for the one-way model can often be used within a two-way setup. For example, the codes that we introduce in Chapter 4, can be easily modified to be used effectively in a two-way model (we discuss this in Chapter 6). Secondly, a one-way coding scheme can be used in scenarios where there are several nodes in the network. For instance, in a broadcast network, when there are several independently edited versions of the file which we need to update simultaneously.

1.3 Previous works

1.3.1 Edit channels

For the special case of correcting one insertion or deletion, there exists an elegant class of codes called Varshamov-Tenengolts (VT) codes. Binary VT codes were first introduced by Varshamov and Tenengolts in [14] for channels with asymmetric errors. Later, Levenshtein [15] showed that they can be used for correcting a single deletion or insertion with zero error and with a simple decoding algorithm whose complexity is linear in the code length [16]. Tenengolts subsequently introduced a non-binary version of VT codes, defined over a q-ary alphabet for any $q > 2$ [17]. The q-ary VT codes retain many of the attractive properties of the binary codes. In particular, they can correct deletion or insertion of a single symbol from a q-ary VT codeword with

a linear-time decoder. VT codes are backbone of many of the schemes that will be discussed in this thesis. In Chapter 2 we study these codes in more detail.

Levenshtein in his influential paper [15] also showed that a code that can correct k deletions with zero error probability is also capable of correcting a combination of k insertions and deletions (with zero error). Moreover, he obtained asymptotic upper and lower bounds for the size of the codes that can correct multiple edits. If $C(n,k)$ denotes the size of the optimal code that can correct k deletions in a length n sequence, then we have

$$c_1 \frac{2^n}{n^{2k}} \leq C(n,k) \leq c_2 \frac{2^n}{n^k}, \tag{1.3.1}$$

where the constants c_1 and c_2 depend only on k. This shows that the optimal redundancy asymptotically is between $2k \log n$ and $k \log n$. We review the existing bounds on the codebook size in more detail in Chapter 2.

In a VT code (also known as Levenshtein code), codewords are the set of sequences $X = x_1 x_2 \cdots x_n$ that satisfy the following constraint

$$\sum_{i=1}^{n} i x_i = a \mod (n+1), \tag{1.3.2}$$

where a is a fixed integer between 0 and n. Attempts have been made to use a similar approach for multiple deletions. In [18] (see also [19]) Helberg and Ferreira suggest a code that can correct multiple deletions by changing the coefficients which are multiplied to x_i's in (1.3.2), and also adjusting the modulus based on the number of deletions. However their suggested code does not have a good rate (the rate goes to zero as n increases).

Another class of codes for edit channels are those that use markers in their construction. A marker is essentially a predetermined set of symbols that help the decoder to achieve the synchronization. The first work on such codes was introduced by Sellers [20], where he used the substring 001 at regular intervals as the marker. Levenshtein [21] and Ferreira et al. [22] have also used some form of a marker in their code. A very successful code in this class is proposed by Ratzer [3]. He uses a low-density parity-check code with some inserted markers and an iterative decoder. More recently, Wang et al. designed a new code with similar structure [23].

Recently, Brakensiek et. al [24] proposed a new code for recovering multiple deletions. The proposed code has redundancy $c_k \log n$ when there are k deletions in a length n codeword, where c_k is of order $O(k^2 \log k)$. However, their code is mostly interesting from a theoretical point of view since the construction only works for very large n. In 2018, a scheme is introduced by Gabrys and Sala [25] which optimizes

the method in [24] for the special case of two deletions. The dominant term in the redundancy of this code is $8\log n$. Concurrently, another code is suggested for the case of two deletions in [26], this code is with a redundancy of $7\log n$. The approach in [26] seems promising as it is more structured and hence is more likely to be generalizable.

Variations of the edit channels have also been studied in the literature. One example is a model called segmented edit channel [27]. Where we have the additional assumption that the channel input sequence is implicitly divided into segments such that at most one edit can occur within a segment. Segmented channels are the topic of Chapter 3 of this thesis. Another variation of an edit channels are sticky channels, where the channel will duplicate some of the symbols a random number of times [28].

From the capacity point of view, Dobrushin showed that the i.i.d. deletion channels are information stable [29]. In general, the capacity of the edit channel is not known. The simplest upper bound for a deletion channel that deletes each bit independently with probability p_d is $1-p_d$. This is because if the decoder knew the position of the deletions, we had an erasure channel for which the capacity is known to be $1-p_d$. Also in [30–40] various lower and upper bounds are provided. Some of these bounds are tight in special regimes, for example in a deletion channel when the probability of deletion p_d goes to zero.

Other notable works in coding for synchronization errors include [41–48]. Also comprehensive surveys are available in [7, 49].

In Chapter 5 of this thesis, the problem of coded trace reconstruction is studied. This problem is motivated by the DNA storage model. In DNA sequencing it is possible to get several reads of a single DNA string. Each of these reads (call them traces) are erroneous. The goal is to use these traces to reconstruct the target sequence. In DNA storage setup we can design the target sequence, therefore we can assume that there is an underlying codebook that the target sequence belongs to.

The trace reconstruction problem has been previously studied, with the goal being either exact recovery of the sequence [50–54], or an estimate [55]. In these papers, the sequence can be an arbitrary one from the underlying alphabet, i.e., it need not originate from a codebook. A few recent works study the reconstruction of a *coded* sequence under different trace models. The paper [56] analyzes the minimum number of deletion channel traces required to recover a sequence drawn from a single-deletion correcting code, and [57] considers a similar problem for the insertion channel. The problem of reconstructing a coded sequence from the multiset of its substrings is studied in [58]. From an information-theoretic perspective, [59] characterizes the capacity of

the multiple trace i.i.d. deletion channel as the deletion probability $p \to 0$. The related works to this problem will be discussed in more details in Chapter 5.

1.3.2 Edit synchronization

Various forms of the edit synchronization mode have been studied in previous works; the majority of them allow for *two-way* interaction between the encoder and decoder [60–72]. Interaction between the two nodes is also used in some practical file synchronization tools such as rsync [73]. Some other models assume that the encoder knows Y, the file at the decoder. Two examples of such works are [63, 74, 75]. An overview of different models for file synchronization problem can be found in [75]. However, as stated earlier in Chapter 4 we consider the one-way version of the problem. To the best of our knowledge, the only work which directly addresses the one-way file synchronization is the Guess and Check code, recently proposed by Hanna and Rouayheb [76]. Although one should notice that some of the codes that are suggested for edit channels principally can be used for one-way synchronization as well. An example of these codes is maker codes which we discussed earlier [3]. Instead of having predetermined markers the encoder in file synchronization setup can send the substring in X explicitly to the decoder and the decoder can use it exactly as it would use a marker. We will compare the work in [76] with our proposed multilayer code in Chapter 4.

1.4 Overview of the thesis

- In Chapter 2, the VT codes are first reviewed, along with the decoding algorithm. It is shown that these codes are zero error single edit correcting codes. Also, the graph interpretation of these codes is introduced and it is shown that the VT code is the only integer code that can offer a valid vertex coloring for the edit graph. Moreover, the existing lower and upper bounds on the rate of the zero error deletion codes are discussed in this chapter. Systematic encoding of VT code is also studied. In particular, a new systematic encoding for non-binary VT codes is given and a new lower bound for the size of non-binary VT classes is introduced. The result of this work is published in [77].

- In Chapter 3, the segmented model is studied for the deletion channel, insertion channel, and insertion or deletion channel. A subset of VT codes is used to devise a zero error coding scheme for these models. Also, an upper bound is found

which shows that the achieved rate is asymptotically optimal. The above results are published in [78, 79].

- The edit synchronization model is studied in Chapter 4, where we introduce multilayer codes, a new class of code that can correct multiple deletions in a one-way setup. These codes use a list decoder and therefore can be easily modified to be used in a two-way setup as well. First, it is shown how to decode multiple deletions. Then necessary modifications in the decoding algorithm are explained in order to recover a combination of insertions and deletions. The modified decoding algorithm will have a higher complexity and a longer list is expected to be produced. A variation of these codes can be used in the edit channel problem. Some of the results of this work are reported in [80].

- In Chapter 5, motivated from the DNA storage model, the problem of coding for the deletion channel with multiple traces is studied. Here, we show that even by using a single deletion code like VT codes (and suitable choice of parameters) multiple deletions can be recovered with a small error probability when several traces are available at the decoder. The results of this work are yet to be published and can be found in [81].

- In Chapter 6, we summarize the contributions of the thesis and discuss directions for future work.

1.5 Notation

We denote scalars by lower-case letters and sequences by capital letters. We denote the subsequence of X, from index i to index j, with $i < j$ by $X(i:j) = x_i x_{i+1} \cdots x_j$. Matrices are denoted by bold capitals. We use brackets for merging sequences, so $X = [X_1, \cdots, X_u]$ is a supersequence defined by concatenating the sequences $X_1, \cdots, X_u$. Random variables are denoted using bold lower case letters. The set $\mathbb{Z}_q = \{0, 1, \cdots, q-1\}$ is the finite integer ring of size q. We consider the natural order for the elements of $\mathbb{Z}_q$, i.e., $0 < 1 \ldots < (q-1)$. The term dyadic index will be used to refer to an index that is a power of 2. Finally, sets are in calligraphic, in particular, $\mathcal{D}_i(X)$ is the set of subsequences of X that can be obtained with i deletions. $\mathcal{I}_i(X)$ denotes the set of supersequences of X, obtained by inserting i bits into X.

Chapter 2

VT codes, structure and encoding

2.1 Introduction

In this chapter, VT codes and some of their properties are discussed. These codes were first introduced by Varshomov and Tenengolts in 1956 for channels with asymmetric error [14]. In the same year, Levenshtein showed that these codes can correct a single edit [15]. The rate of the code is asymptotically optimal, and it is conjectured that the optimal code also belongs to the family of VT codes [16]. Another attractive feature of these codes is that the decoding algorithm is very simple and its complexity is linear in time. Given the simplicity of VT decoding, a natural question is: can one construct a linear-time encoder to efficiently map binary message sequences onto VT codewords? For binary VT codes, such an encoder was proposed by Abdel-Ghaffar and Ferriera [82]. (A similar encoder was also described in [83].) However, the issue of efficient encoding for non-binary VT codes has not been addressed previously. In this chapter, an efficient systematic encoder for non-binary VT codes is proposed. The encoder has the complexity that is linear in the code length and is systematic in the sense that the message bits are assigned to pre-specified positions in the codeword. The encoder also yields a new lower bound on the size of q-ary VT codes, for $q > 2$.

VT codes are a key ingredient of the code construction for channels with segmented deletions and insertions that we will propose in next chapter, and also in the codes that are suggested for multiple traces problem in Chapter 5. The proposed VT encoder can be applied to these constructions. VT codes have also recently been used in algorithms for synchronization from deletions and insertions, e.g., [65, 66, 84] and are the key component of multilayer codes which will be discussed in Chapter 4.

In [17, Sec. 5], Tenengolts introduced a systematic non-binary code that can correct a single deletion or insertion. However, this code is not strictly a VT code as its

codewords do not necessarily share the same VT parameters. (Formal definitions of VT codes and their parameters are given in the next section.) In this chapter, we propose an encoder for VT codes defined in the standard way, noting that using standard VT codes is a key requirement in some of the code constructions mentioned above.

The rest of the chapter is organized as follows. In the next section, we formally define binary and non-binary VT codes and present some results on the size of the classes. Then in Section 2.3, the graph model for zero error codes (like VT codes) is introduced and it is shown that VT codes are the only integer codes that can color the graph. Moreover, existing bounds for the size of zero error codes are presented. Then in Section 2.4 the systematic encoder from [82] is briefly reviewed. In Section 2.5, a new systematic encoding method for q-ary VT codes ($q > 2$) and the resulting lower bound on the size of the codes is described.

2.2 The VT code construction

In this section we introduce the code construction and also the decoding algorithm for both binary and non-binary VT codes. Also, the size of these codes is studied.

2.2.1 Binary codes

The VT syndrome of a binary sequence $X = x_1x_2\cdots x_n \in \mathbb{Z}_2^n$ is defined as

$$\mathsf{syn}(X) \triangleq \sum_{i=1}^{n} i\,x_i \mod (n+1). \tag{2.2.1}$$

For positive integers n and $0 \le a \le n$, the VT code of length n and syndrome a, is defined as

$$\mathcal{VT}_a(n) = \{X \in \mathbb{Z}_2^n : \mathsf{syn}(X) = a\}, \tag{2.2.2}$$

i.e., the set of binary sequences X of length n that satisfy $\mathsf{syn}(X) = a$. For example, the VT code of length 3 and syndrome 2 is

$$\begin{aligned}\mathcal{VT}_2(3) &= \Big\{x_1x_2x_3 \in \mathbb{Z}_2^3 : \sum_{j=1}^{3} j\,x_j = 2 \bmod 4\Big\}\\ &= \{010, 111\}.\end{aligned} \tag{2.2.3}$$

Now we show that each of the sets $\mathcal{VT}_a(n)$, $0 \le a \le n$, is a code that can correct a single deletion or insertion by introducing the decoder. Note that this will prove that,

if two sequences $S, S' \in \mathcal{VT}_a(n)$, then

$$\mathcal{D}_1(S) \cap \mathcal{D}_1(S') = \emptyset, \quad \text{and} \quad \mathcal{I}_1(S) \cap \mathcal{I}_1(S') = \emptyset, \tag{2.2.4}$$

where $\mathcal{D}_1(S)$ denotes the set of subsequences obtained by deleting one bit from S, and $\mathcal{I}_1(S)$ is the set of supersequences obtained by inserting one bit in S.

Decoding a single deletion: Assume that $X \in \mathcal{VT}_a(n)$ and $X' = x'_1 x'_2 \cdots x'_{n-1}$ is a sequence obtained by deleting one bit from X. Compute the following checksum for X':

$$a' = \sum_{i=1}^{n-1} i\, x'_i \mod (n+1). \tag{2.2.5}$$

Define the deficiency as $\delta = a - a' \mod (n+1)$. Now assume the deleted symbol from X is x_p, if $x_p = 0$, then the deficiency is exactly equal to the number of ones in $X(p+1:n)$. Therefore, if we denote by w the weight of X' (number of ones in X'), then we have $\delta \leq w$. Now consider the case where $x_p = 1$, in this case δ will be equal to the number of ones in $X(p+1:n)$ added to p. Thus, δ will be strictly greater than w. Therefore, comparing δ and w determines the value of the deleted bit. Also, given these two values we can determine the run that the deleted bit belongs to and hence uniquely decode the codeword.

1. If $\delta \leq w$: Then the deleted bit is 0 and it belongs to the run that is after exactly δ ones from the right side of X'.

2. If $\delta > w$: Then the deleted bit is 1 and it belongs to the run that is after exactly $\delta - w$ ones from the right side of X'.

Clearly, the decoding complexity is linear in n.

Decoding a single insertion: Assume that $X' = x'_1 x'_2 \cdots x'_{n+1}$ is a sequence obtained by inserting one bit into $X \in \mathcal{VT}_a(n)$. Also assume that the inserted bit is x'_p. Similarly compute the following checksum for X':

$$a' = \sum_{i=1}^{n+1} i\, x'_i \mod (n+1). \tag{2.2.6}$$

This time define deficiency to be $\delta = a' - a \mod (n+1)$. Again define w to be the weight of X'. For decoding:

1. If $\delta \leq w$: Then the inserted bit is 0 and it belongs to the run that is after exactly δ ones from the right side of X'.

2. If $\delta > w$: Then the inserted bit is 1 and it belongs to the run that is after exactly δ ones from the right side of X'.

2.2.2 Size of VT classes

The $(n+1)$ sets $\mathcal{VT}_a(n)$, $0 \le a \le n$, partition the set of all binary sequences of length n, i.e., each sequence $X \in \mathbb{Z}_2^n$ belongs to exactly one of the sets. Therefore, the smallest of the codes $\mathcal{VT}_a(n)$ will have at most $\frac{2^n}{n+1}$ sequences. Hence, we have the following bounds on the rate of different classes:

$$\min_{0\le a\le n} \frac{1}{n}\log_2|\mathcal{VT}_a(n)| \le 1-\frac{1}{n}\log_2(n+1) \le \max_{0\le a\le n} \frac{1}{n}\log_2|\mathcal{VT}_a(n)|. \tag{2.2.7}$$

The systematic encoding that is introduced in Section 2.4 can map $2^{n-\lceil\log_2(n+1)\rceil}$ messages to any of the VT classes. This gives the following lower bound on the rate of VT classes for all n and $0 \le a \le n$.

$$1-\frac{1}{n}\lceil\log_2(n+1)\rceil \le \frac{1}{n}\log_2|\mathcal{VT}_a(n)|. \tag{2.2.8}$$

A byproduct of this inequality is that when $n+1$ is a power of 2, the size of all VT classes is $\frac{2^n}{n+1}$. This is because $|\mathcal{VT}_a(n)| \ge 2^{n-\lceil\log_2(n+1)\rceil}$, hence

$$2^n = \sum_{a=0}^{n} |\mathcal{VT}_a(n)| \tag{2.2.9}$$

$$\ge (n+1)2^{n-\lceil\log_2(n+1)\rceil} \tag{2.2.10}$$

$$= 2^n, \tag{2.2.11}$$

where (2.2.11) holds since when $n+1$ is a power of two we have $2^{\lceil\log_2(n+1)\rceil} = n+1$. Thus the inequalities used in (2.2.10) are indeed satisfied with equality. This means that we have $|\mathcal{VT}_a(n)| = 2^{n-\lceil\log_2(n+1)\rceil}$ for $0 \le a \le n$, and this completes the proof. This result motivates the misguided intuition that when $n+1$ is a power of 2, the $\mathcal{VT}_0(n)$ is a linear code, and other classes are essentially different cosets of a linear code. However, as it is shown in [16], this is not the case and for $n \ge 5$, $\mathcal{VT}_0(n)$ is not a linear code.

The exact size of each of the classes is studied in [85–87]. In [16, Theorem 2.2], an expression for the exact size of VT classes is given,

$$|\mathcal{VT}_a(n)| = \frac{1}{2(n+1)} \sum_{\substack{h|n+1 \\ h \text{ odd}}} \phi(h) \frac{\mu(\frac{h}{(h,a)})}{\phi(\frac{h}{(h,a)})} 2^{(n+1)/h}. \tag{2.2.12}$$

Here ϕ is the Euler totient function, and μ is the Möbius function [88]. Also (h,a) denotes the greatest common divisor of the integers h and a. In general, this expression is hard to compute, but in some special cases it is computable. When $(n+1)$ is a power of two, then 1 is the only odd divisor of $(n+1)$. Also $\phi(1)=\mu(1)=1$, thus we have

$$|\mathcal{VT}_a(n)| = \frac{1}{2(n+1)}2^{(n+1)} \tag{2.2.13}$$

$$= \frac{2^n}{n+1} \tag{2.2.14}$$

which confirms the earlier stated result. Also, using this formula it can be shown that for any $0 \le a \le n$

$$\mathcal{VT}_1(n) \le \mathcal{VT}_a(n) \le \mathcal{VT}_0(n), \tag{2.2.15}$$

(see [16] for the proof). It is conjectured that $\mathcal{VT}_0(n)$ is the largest codebook capable of correcting one edit (for $n \le 8$ this has been checked) [16].

Moreover, for general n, the formula can be used to deduce that the sizes of the codes $\mathcal{VT}_a(n)$ are all approximately $2^n/(n+1)$. In particular,

$$\frac{2^n}{(n+1)} - 2^{(n+1)/3} \le |\mathcal{VT}_a(n)| \le \frac{2^n}{(n+1)} + 2^{(n+1)/3}, \qquad \text{for } a \in \{0,\dots,n\}. \tag{2.2.16}$$

To prove this, first notice that for a positive integer h, the Euler totient function $\phi(h)$ is the number of positive integers up to h that are relatively prime to h, and $\phi(1)$ is defined to be 1. Thus, $1 \le \phi(h) \le h$. We will also use the fact that the Möbius function $\mu(\cdot)$ takes values in the set $\{-1,0,1\}$. Then, from (2.2.12) we have

$$|\mathcal{VT}_a(n)| = \frac{1}{2(n+1)} \sum_{\substack{h|n+1 \\ h \text{ odd}}} \phi(h) \frac{\mu(\frac{h}{(h,a)})}{\phi(\frac{h}{(h,a)})} 2^{(n+1)/h} \tag{2.2.17}$$

$$\le \frac{1}{2(n+1)} \left(2^{n+1} + \sum_{\substack{h|n+1 \\ h>1,\ \text{odd}}} \phi(h) \frac{1}{\phi(\frac{h}{(h,a)})} 2^{(n+1)/h} \right) \tag{2.2.18}$$

$$\le \frac{1}{2(n+1)} \left(2^{n+1} + (n+1) \sum_{\substack{h|n+1 \\ h>1,\ \text{odd}}} 2^{(n+1)/h} \right). \tag{2.2.19}$$

Here, (2.2.18) is obtained from $\mu(x) \le 1$, and (2.2.19) holds since $\phi(h) \le h \le n+1$. The number of terms in the summation in (2.2.19) is not more than $(1+\frac{n+1}{3})$, and

the largest term in the summation in (2.2.19) occurs when $h=3$. Moreover, we have $2^{(n+1)/h} \leq 2^{\frac{n+1}{3}-h}$. Hence we have

$$|\mathcal{VT}_a(n)| \leq \frac{1}{2(n+1)}\left(2^{n+1}+(n+1)\sum_{h=0}^{\frac{n+1}{3}} 2^{\frac{n+1}{3}-h}\right) \tag{2.2.20}$$

$$= \frac{1}{2(n+1)}\left(2^{n+1}+(n+1)\left(2^{\frac{n+1}{3}+1}-1\right)\right) \tag{2.2.21}$$

$$\leq \frac{2^n}{(n+1)}+2^{(n+1)/3}. \tag{2.2.22}$$

The lower bound in (2.2.16) is obtained by using $\mu(x) \leq -1$ in (2.2.17), and following similar steps.

2.2.3 Non-binary codes

For any code length n, the VT codes over $\mathbb{Z}_q$, for $q>2$ are defined as follows [17]. For each q-ary sequence $S = s_0 s_1 \cdots s_{n-1} \in \mathbb{Z}_q^n$, define a corresponding length $(n-1)$ auxiliary binary sequence $A_S = \alpha_1\alpha_2\ldots\alpha_{n-1}$ as follows[1]. For $1 \leq i \leq n-1$,

$$\alpha_i = \begin{cases} 1 & \text{if } s_i \geq s_{i-1} \\ 0 & \text{if } s_i < s_{i-1}. \end{cases} \tag{2.2.23}$$

We also define the modular sum of S as

$$\mathsf{sum}(S) = \sum_{i=0}^{n-1} s_i \pmod q. \tag{2.2.24}$$

For $0 \leq a \leq n-1$ and $b \in \mathbb{Z}_q$, the q-ary VT code with length n and parameters (a,b) is defined as

$$\mathcal{VT}_{a,b}(n) = \{S \in \mathbb{Z}_q^n : \mathsf{syn}(A_S) = a, \mathsf{sum}(S) = b\}. \tag{2.2.25}$$

Each of the sets $\mathcal{VT}_{a,b}(n)$ is a code that can correct deletion or insertion of a single symbol with a decoder whose complexity is linear in the code length n. The decoder first constructs the auxiliary sequence of the received sequence and then uses the binary VT syndrome of the auxiliary sequence to find the run which the deleted (inserted) symbol belongs to. Finally the summation of symbols will determine the value of the deleted bit and from the order of the symbols (known from auxiliary sequence)

[1]For non-binary sequences, we start the indexing from 0 as this makes it convenient to describe the encoding procedure in Section 2.5.

the decoder finds the exact position of the symbol within the run. The details of the decoding algorithm can be found in [17, Sec. II].

Similar to the binary case, the codes $\mathcal{VT}_{a,b}(n)$, for $0 \leq a \leq n-1$ and $b \in \mathbb{Z}_q$, partition the space $\mathbb{Z}_q^n$ of all q-ary sequences of length n. For a given n, there are nq of these codes, and hence the largest (smallest) of them will have at least (at most) $\frac{q^n}{nq}$ sequences. Let $R_{\min}$ be the rate of the smallest of these codes, i.e.,

$$R_{\min} \triangleq \min_{a,b} \frac{\log_2 |\mathcal{VT}_{a,b}(n)|}{n}, \tag{2.2.26}$$

where the minimum is over $0 \leq a \leq n-1$ and $b \in \mathbb{Z}_q$. We then have the bound

$$R_{\min} \leq \log_2 q - \frac{1}{n}\log_2 n - \frac{1}{n}\log_2 q \quad \text{bits/symbol}. \tag{2.2.27}$$

We are interested in $R_{\min}$, since it is an upper bound for the best rate that a systematic encoder (which can map messages to any of the classes) can achieve.

Unlike the binary case, there exists no lower bound on the size of non-binary VT classes in the previous literature. Later in this chapter Proposition 2.2 provides the first lower bound on the size of these classes.

2.3 Edit graph model

Let us define the confusability graph for a channel that deletes up to k symbols from the input sequence. In the zero error model we need to consider the worst case scenario, hence we assume that the channel deletes exactly k symbols. The confusability graph $\mathcal{G}_{n,k}$ has 2^n vertices corresponding to each of the length n binary sequences. There is an edge between two vertices when they have an edit distance smaller than or equal to $2k$. Equivalently, two vertices corresponding to sequences X and Y are connected to each other if and only if $|\mathcal{D}_k(X) \cap \mathcal{D}_k(Y)| > 0$. Now we know that a valid codebook for zero error communication via a deletion channel with k deletions is an independent set of this graph.

Remark 2.1. *Recall Levenshtein's result in [15], which shows that a code that can correct k deletions is also capable of correcting a combination of k insertions or deletions. This means that the confusability graph when k insertions and deletions are allowed is the same as $\mathcal{G}_{n,k}$.*

For a graph $\mathcal{G}$ we denote the chromatic number by $\mathcal{X}(\mathcal{G})$, this is the minimum number of required colors to color vertices of $\mathcal{G}$ such that any two connected vertices

have different colors. Also, the independence number denoted by $\alpha(\mathcal{G})$ is the maximum number of vertices for which there are no edges among them. From the definition, when we have a valid coloring, the set of vertices of any of the colors will form an independent set.

VT syndromes color $\mathcal{G}_{n,1}$ with $n+1$ colors, where each of the syndromes is a color. This is a valid coloring because each of the sets $\mathcal{VT}_a(n)$ (for $0 \leq a \leq n$) are a single edit zero error correcting code, which means the vertices corresponding to codewords are not connected to each other. An integer code, defined by Vinik and Morita [89] (see also [90]) is a code where the codewords are all length n sequences that satisfy the following constraint:

$$\sum_{i=1}^{n} w_i x_i = d \mod m, \tag{2.3.1}$$

where w_i's and d are in $\mathbb{Z}_m$. VT codes are an instance of these codes where m is set to be $n+1$ and $w_i = i$. In the following we investigate all the integer codes that can offer a vertex coloring (with $n+1$ colors) for $\mathcal{G}_{n,1}$.

Proposition 2.1. *Let*

$$d = \sum_{i=1}^{n} w_i x_i \quad mod\ n+1.$$

Then, assigning color d to node $X = x_1 \cdots x_n$ is a valid coloring if and only if $w_i = iw_1$, where $(w_1, n+1) = 1$ (i.e. w_1 and $n+1$ are co-prime).

It is clear that choosing $w_1 = 1$ gives VT syndromes. The proof of this proposition gives intuition about why VT codes are capable of coloring the graph. Moreover, we were interested in finding whether there exists another code with the same structure capable of correcting one edit. Note that this proposition shows that VT codes are in a sense unique, since other codes (when we choose $w_1 \neq 1$) are essentially relabeling of the colors and the set of sequences with the same color remains invariant.

Proof. The first observation is that w_i's are a permutation of 1 to n. This is because if $w_i = w_j$ for some $i \neq j$, then two sequences X_1, which has $n-1$ zeros and a single one at position i, and X_2, which has $n-1$ zeros and a single one at position j, are connected to each other and they have the same color. Also, we have $w_i \neq 0$, since if $w_i = 0$, then the sequence that has only a single 1 at position i and has zeros elsewhere is connected to all zero sequence and they both have the same color (d is zero for both of them) this is a contradiction. This shows that w_i's are a permutation of 1 to n.

Consider Y to be a length $n-1$ sequence. To have a valid coloring we need this property: two sequences that are obtained by inserting two symbols into Y should

have different colors. This is because the vertices corresponding to these two sequences are connected in the graph. Assume that Y_1 is the sequence obtained by inserting symbol x_{p_1} into position p_1 of Y. And Y_2 is the sequence obtained by inserting x_{p_2} into position p_2 of Y. Y_1 and Y_2 are different if either the values of symbols x_{p_1} and x_{p_2} are different or positions p_1 and p_2 belong to different runs in Y. Denote by $d(Y, x_{p_1})$ the deficiency after inserting x_{p_1} into Y. Then, we have

$$d(Y, x_{p_1}) = x_{p_1} w_{p_1} + \sum_{i=p_1+1}^{n} (w_i - w_{i-1}) y_{i-1} \mod n+1, \tag{2.3.2}$$

where y_i is the ith bit of Y. With this notation the above property can be stated as follows:

$$\forall Y \in \{0,1\}^{n-1}, \quad d(Y, x_{p_1}) \neq d(Y, x_{p_2}), \tag{2.3.3}$$

when x_{p_1} and x_{p_2} are either not equal or not in a same run. Now consider $p_2 > p_1$, when x_{p_1} and x_{p_2} are in different runs, for satisfying the property in (2.3.3) we need

$$x_{p_1} w_{p_1} + \sum_{i=p_1+1}^{n} (w_i - w_{i-1}) y_{i-1} \neq x_{p_2} w_{p_2} + \sum_{i=p_2+1}^{n} (w_i - w_{i-1}) y_{i-1} \mod n+1, \tag{2.3.4}$$

$$x_{p_1} w_{p_1} + \sum_{i=p_1+1}^{p_2} (w_i - w_{i-1}) y_{i-1} \neq x_{p_2} w_{p_2} \mod n+1. \tag{2.3.5}$$

Now using (2.3.5) we have the following inequalities:

$$\forall l > 2: \quad w_2 - w_1 \neq w_l \mod n+1. \tag{2.3.6}$$

For instance to see $w_2 - w_1 \neq w_3$, choose $p_2 = 3$ and $p_1 = 1$. Also $x_{p_1} = 0$ and $x_{p_2} = 1$. Now consider $y_1 = 1$ and $y_i = 0$ for $i > 1$. Using these parameters in (2.3.5) gives the result. Similarly using $p_2 = l$ gives the other inequalities in (2.3.6).

Notice that w_i's are a permutation, thus there exist a unique i such that $w_i = w_2 - w_1$ (all calculations are in $\mathbb{Z}_{n+1}$). Therefore, we have either $w_1 = w_2 - w_1$ or $w_2 = w_2 - w_1$. The second possibility is not acceptable since $w_1 \neq 0$. Hence, $w_1 = w_2 - w_1$ or equivalently $w_2 = 2w_1$.

Similarly for $j \geq 3$, we have following inequalities:

$$\forall l > j: \quad w_j - w_1 \neq w_l \mod n+1. \tag{2.3.7}$$

Choose $p_2 = l$, $p_1 = 1$, $x_{p_1} = 0$, and $x_{p_2} = 1$. Now for Y, choose $y_1 = y_2 = \cdots = y_{j-1} = 1$ and $y_i = 0$ for $i \geq j$. Using these parameters in (2.3.5) gives (2.3.7). Now induction on j (with the hypothesis that $w_i = iw_1$ for $i < j$) shows that $w_j - w_1 = w_{j-1}$ is the only available choice in order to have a permutation and hence we have $w_j = jw_1$.

Now for w_i's forming a permutation, we need $(w_1, n+1) = 1$. This shows that the condition of the proposition is necessary. The other direction is easy, since we know that VT syndrome colors the graph. For a constraint where $w_i = iw_1$ and $w_1 \neq 1$, since $(w_1, n+1) = 1$, there exist $w_1^{-1} \in \mathbb{Z}_{n+1}$ such that $w_1^{-1} w_1 = 1$ (in $\mathbb{Z}_{n+1}$). Therefore, if this constraint colors two sequences with the same color d, by using the VT syndrome for coloring, these two sequences will also have the same color $w_1^{-1} d$, and thus are not connected to each other. Therefore this new constraint is also a valid coloring. This completes the proof. □

2.3.1 Lower bounds for zero error codes

In this subsection we want to find some lower bounds for the size of the maximum zero error codebook ($\alpha(\mathcal{G}_{n,k})$), using graph theory concepts that we reviewed in this section. As we discussed, when there is a valid vertex coloring, the set of vertices of any of the colors will form an independent set. Hence, these sets are a zero error codebook, thus we have the following lower bound for $\alpha(\mathcal{G}_{n,k})$:

$$\frac{2^n}{\mathcal{X}(\mathcal{G}_{n,k})} \leq \alpha(\mathcal{G}_{n,k}). \tag{2.3.8}$$

Now if we upper bound the chromatic number $\mathcal{X}(\mathcal{G}_{n,k})$, we can find a lower bound for $\alpha(\mathcal{G}_{n,k})$. The simplest upper bound for $\mathcal{X}(\mathcal{G}_{n,k})$ is $\mathcal{X}(\mathcal{G}_{n,k}) \leq \Delta(\mathcal{G}_{n,k}) + 1$, where $\Delta(\mathcal{G}_{n,k})$ is the greatest degree in the graph (the proof by induction is straightforward). Thus,

$$\frac{2^n}{\Delta(\mathcal{G}_{n,k}) + 1} \leq \alpha(\mathcal{G}_{n,k}). \tag{2.3.9}$$

A simple upper bound for $\Delta(\mathcal{G}_{n,k})$ is the following:

$$\Delta(\mathcal{G}_{n,k}) \leq \binom{n}{k} \left(\sum_{i=0}^{k} \binom{n}{i} \right) \tag{2.3.10}$$

This is because a sequence like X is connected to all the sequences which can obtained by deleting k bits from X and inserting k bits into the $n-k$ subsequece. $\binom{n}{k}$ is an upper bound for the number of ways that k bits can get deleted from a length n

sequence (i.e., the number of subsequences of length $(n-k)$) and the second term in RHS is the number of ways to insert k bits into a length $(n-k)$ sequence. In general the number of distinct sequences obtained by inserting k bits into X a q-ary sequence of length n is the following (see [91] for the proof).

$$|\mathcal{I}_k(X)| = \sum_{i=0}^{k} \binom{n+k}{i} (q-1)^i \tag{2.3.11}$$

Using (2.3.10) we have

$$\Delta(\mathcal{G}_{n,k}) + 1 \leq (k+1)\binom{n}{k}^2 \tag{2.3.12}$$

$$\leq (k+1)\left(\frac{en}{k}\right)^{2k}, \tag{2.3.13}$$

where the last inequality holds because of Lemma 4.2 in Chapter 4 (see page 101). Therefore, we obtain a lower bound for $\alpha(\mathcal{G}_{n,k})$ as

$$\frac{2^n}{(k+1)\left(\frac{en}{k}\right)^{2k}} \leq \alpha(\mathcal{G}_{n,k}). \tag{2.3.14}$$

This is in accordance with Levenshtein's asymptotic lower bound (1.3.1) which gives redundancy of order $2k\log n$.

For the non-binary alphabet, denote the confusability graph for the q-ary alphabet as $\mathcal{G}_{n,k,q}$. Similarly an upper bound for $\Delta(\mathcal{G}_{n,k,q})$ can be found:

$$\Delta(\mathcal{G}_{n,k,q}) + 1 \leq (k+1)q^k\binom{n}{k}^2. \tag{2.3.15}$$

This will result to the following lower bound on the size of the zero error code

$$\frac{q^{n-k}}{(k+1)\binom{n}{k}^2} \leq \alpha(\mathcal{G}_{n,k,q}). \tag{2.3.16}$$

One can refine the above lower bound by bounding Δ more accurately. This can be done by using a better bound for the number of subsequences. Unlike the number of supersequences which is given explicitly in (2.3.11), for the number of subsequences we can only give bounds based on the number of runs (see [92, 93] for details). However, there exist a more powerful method which gives the best known lower bound for zero

error deletion codes. It has been proved in [94] (another proof is given in [95]), that one can used the average degree instead of $\Delta(\mathcal{G}_{n,k})+1$ in the denominator of (2.3.9). Using this method Levenshtein gave the following lower bound in [96] for a zero error code that can correct k deletions in a q-ary sequence:

$$\alpha(\mathcal{G}_{n,k,q}) \geq \frac{q^{n+k}}{\left(\sum_{i=0}^{k}\binom{n}{i}(q-1)^i\right)^2}. \tag{2.3.17}$$

For small values of n and k in [97, 98] better bounds for $\alpha(\mathcal{G}_{n,k})$ are found by using computer search.

Remark 2.2. *The bound $\mathcal{X}(\mathcal{G}_{n,k}) \leq \Delta(\mathcal{G}_{n,k})+1$, is the simplest upper bound on the chromatic number. One idea is to use better existing upper bounds for chromatic number to get a better lower bound on the independence number. One example of a tighter upper bound is the following:*

$$\mathcal{X}(\mathcal{G}) \leq \max_{i\in\{1,\cdots,n\}} \min(d_i+1, i), \tag{2.3.18}$$

here $d_1 \geq d_2 \geq \cdots \geq d_n$ are the degrees of the vertices in $\mathcal{G}$ (with this notation $\Delta(\mathcal{G}) = d_1$). One can check that using this bound, if there exist s such that $d_s+1 \leq s$ then we have $\mathcal{X}(\mathcal{G}) \leq s$. Using this technique we have not been able to find a bound which strictly improves on (2.3.17) *for the edit channel. However, it is interesting to investigate whether there exists a graph (which is important from information theory perspective) for which this type of bound can strictly improve the bound given in [94], which is based on the average degree of the graph.*

2.3.2 Upper bounds on the size of deletion codes

Levenshtein [96] also introduced a family of upper bounds on the the size of $\alpha(\mathcal{G}_{n,k,q})$. The following inequality holds for all r and s, such that $1 \leq s \leq r+1 \leq n$.

$$\alpha(\mathcal{G}_{n,k,q}) \leq \frac{q^{n-k}}{\sum_{i=0}^{k}\binom{r+i-s}{i}} + q\sum_{i=0}^{r-1}\binom{n-1}{i}(q-1)^i \tag{2.3.19}$$

A slightly improved upper bound is given in [99].

Kulkarni and Kiyavash [100] proposed a new method for achieving nonasymptotic upper bounds on the size of edit correcting codes. They formulate the problem as an integer linear program, and propose upper bounds by finding feasible solutions for the

dual of the linear programming relaxation of the integer linear program. In particular, they showed that the size of any single deletion correcting q-ary code of length n is bounded by

$$\alpha(\mathcal{G}_{n,1,q}) \leq \frac{q^n - q}{(q-1)(n-1)}. \tag{2.3.20}$$

In [101] a generalization of their method is given.

2.3.3 Properties of the confusability graph

Another relevant parameter in the edit channel confusability graph is the clique number. A clique is a subset of vertices where each pair in the subset are connected to each other. Clique number is the size of the largest clique of the graph, and is denoted by $\omega(\mathcal{G}_{n,k})$. A simple observation is that $\omega(\mathcal{G}_{n,k}) \leq \mathcal{X}(\mathcal{G}_{n,k})$, since all the vertices of a clique in a valid coloring should have different colors. Moreover, the following lower bound is known for the clique number of $\mathcal{G}_{n,k}$:

$$\sum_{i=1}^{k} \binom{n}{i} \leq \omega(\mathcal{G}_{n,k}). \tag{2.3.21}$$

This is because all the sequences that have at most k ones are connected to each other in $\mathcal{G}_{n,k}$. Therefore, we have the following inequalities for $\mathcal{G}_{n,k}$:

$$\sum_{i=1}^{k} \binom{n}{i} \leq \omega(\mathcal{G}_{n,k}) \leq \mathcal{X}(\mathcal{G}_{n,k}). \tag{2.3.22}$$

For the special case of $k = 1$, VT codes show that both of these inequalities are tight (all three quantities are $n+1$). In other trivial cases (e.g. when $k = n-1$) these inequalities are also tight. Hence, one can question whether these inequalities are tight in general. Using computer search it can be shown that for $k = 2$ the first inequality is not tight. For example for $n = 4$ the lower bound is 11, while $\omega(\mathcal{G}_{4,2}) = 12$. However, it is curious to see whether $\omega(\mathcal{G}_{n,k}) = \mathcal{X}(\mathcal{G}_{n,k})$ holds, the graphs for which we have this equality are called weakly perfect graphs.

2.4 Systematic encoding of binary VT codes

Abdel-Ghaffar and Ferriera [82] proposed a systematic encoder to map k-bit message sequences onto codewords in $|\mathcal{VT}_a(n)|$, where $k = n - \lceil \log_2(n+1) \rceil$. We briefly review the encoding procedure which is also an ingredient of the systematic q-ary VT encoder.

Consider a k-bit message $M = m_1 \cdots m_k$ to be encoded into a codeword $C = c_1 \cdots c_n \in \mathcal{VT}_a(n)$, for some $a \in \{0, 1, \cdots, n\}$. The number of "parity" bits is denoted by $t = n - k = \lceil \log_2(n+1) \rceil$. The idea is to use the code bits in dyadic positions, i.e., c_{2^i}, for $0 \le i \le (t-1)$, to ensure that $\mathsf{syn}(C) = a$. The encoding steps are:

1. Denote the first k non-dyadic indices by $\{j_1, \cdots, j_k\}$, where the indices are in ascending order, i.e., $j_1 = 3, j_2 = 5, \ldots$ We set c_{j_i} equal to the message bit m_i, for $1 \le i \le k$.

2. First set the bits in all the dyadic positions to be zero and denote the resulting sequence by $C' = c'_1 \cdots c'_n$ (so that we have $c'_{2^i} = 0$ for $0 \le i \le t-1$ and $c'_{j_l} = m_l$ for $1 \le l \le k = (n-t)$). Define the deficiency d as the difference between the desired syndrome a and the syndrome of C'. That is,

$$d = a - \mathsf{syn}(C') \mod (n+1). \tag{2.4.1}$$

3. Let the binary representation of d be $d_{t-1} \ldots d_1 d_0$, i.e., $d = \sum_{i=0}^{t-1} 2^i d_i$. Set $c_{2^i} = d_i$, for $0 \le i \le (t-1)$, to obtain C.

The rate of this systematic encoder is $R = 1 - \frac{1}{n}\lceil \log_2(n+1) \rceil$, regardless of the syndrome $a \in \{0, \ldots, n\}$. Comparing with (2.2.7), we observe that the rate loss for the smallest VT code of length n is less than $\frac{1}{n}$. On the other hand, if $(n+1)$ is not a power of two, the rate loss for the larger VT codes may be higher due to codewords that are unused by the encoder. However, this rate loss is unavoidable with any systematic encoder [82].

Remark 2.3. *The dyadic positions are not the only set of positions that can be used for syndrome bits. For instance, the following set of indices also produce all syndromes:*

$$\{c_{i_0}, \cdots, c_{i_{t-1}}\} \quad \textit{where} \quad i_j = -2^j \quad (\textit{mod } n+1) \quad \textit{for } 0 \le j \le t-1.$$

This can be helpful in some applications (see e.g. [78]) where some code bits are already reserved for prefixes or suffixes, and thus cannot be used as syndrome bits. In general, a set of positions $\{p_1, p_2, \cdots, p_r\}$ can be used for syndrome bits if for each syndrome $a \in \{0, \cdots, n\}$, there exists a subset $\mathcal{P} \subseteq \{p_1, p_2, \cdots, p_r\}$ such that

$$\sum_{j \in \mathcal{P}} p_j = a \quad (\textit{mod } n+1). \tag{2.4.2}$$

In other words, for each $a \in \{0, \cdots, n\}$, there should exist binary coefficients $b_1, \cdots, b_r$ such that

$$\sum_{j=1}^{r} b_j p_j = a \quad (mod\ n+1). \tag{2.4.3}$$

2.5 Efficient encoding for non-binary VT codes

The encoding procedure described below yields a lower bound on the size of $|\mathcal{VT}_{a,b}(n)|$ (see Proposition 2.2), which shows that for $q \geq 4$,

$$R_{\min} \geq \log_2 q - \frac{1}{n}\lceil \log_2 n \rceil (3\log_2 q - 2\log_2(q-1)) - \frac{1}{n}(5\log_2(q-1) - 3\log_2 q) \text{ bits/symbol.} \tag{2.5.1}$$

Kulkarni and Kiyavash [100] have shown that the size of any single deletion correcting q-ary code of length n is bounded by $\frac{q^n - q}{(q-1)(n-1)}$. This yields a rate upper bound $R_{\max}$ for any single deletion correcting code as

$$R_{\max} \leq \log_2 q - \frac{\log_2(n-1)}{n} - \frac{\log_2(q-1)}{n}. \tag{2.5.2}$$

We now describe the encoder to map a sequence of message bits to a codeword of the q-ary VT code $\mathcal{VT}_{a,b}(n)$. For simplicity, we first assume that q is a power of two, and address the case of general q at the end of this section. We also assume that $n \neq 2^j + 1$ for any integer j, for such code lengths the rate is slightly less. We will map a k-bit message $M = m_1 m_2 \cdots m_k$ to a codeword in $\mathcal{VT}_{a,b}(n)$, where

$$k = (n - 3t + 3)\log_2 q + (t-3)(2\log_2 q - 1) + (\log_2 q - 1) \tag{2.5.3}$$

$$= n\log_2 q - t(\log_2 q + 1) - 2(\log_2 q - 1), \tag{2.5.4}$$

with $t = \lceil \log_2 n \rceil$. Therefore, the rate of our encoding scheme is

$$R = \log_2 q - \frac{\lceil \log_2 n \rceil (\log_2 q + 1)}{n} - \frac{2\log_2 q - 2}{n} \text{ bits/symbol.} \tag{2.5.5}$$

In Figure 2.1 we plot the upper bound in (2.5.2) and the rate of the proposed method (2.5.5) for the case of $q = 4$ for $n = 20$ up to $n = 2000$.

Our encoding method gives a lower bound on the size of any non-binary VT code of length n. An immediate lower bound on the size is 2^k, with k given by (2.5.4). The proposition below gives a slightly better bound, which is obtained by modifying the

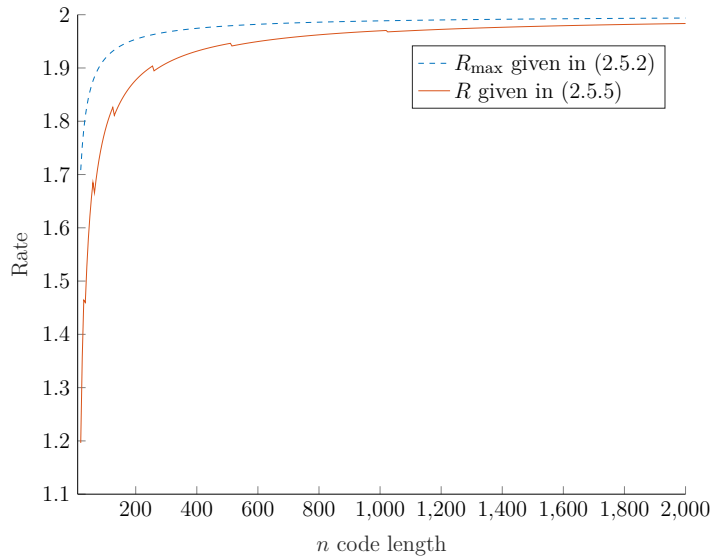

Figure 2.1: Comparison of the rate of the encoder with the upper bound on the rate for $q = 4$, and $20 \leq n \leq 2000$.

encoding method to map q-ary message sequences to q-ary VT codewords, rather than a binary message sequence to a q-ary VT codeword.

Proposition 2.2. *For $n \geq 6$, $q \geq 4$, and any $0 \leq a \leq n$, and $b \in \mathbb{Z}_q$, we have*

$$\begin{aligned} |\mathcal{VT}_{a,b}(n)| &\geq (q-1)^{2t-5} q^{n-3t+3}, \\ &= q^{n\left(1-\frac{t}{n}[3\log_2 q - 2\log_2(q-1)] - \frac{1}{n}[5\log_2(q-1) - 3\log_2 q]\right)} \end{aligned}$$

where $t = \lceil \log_2 n \rceil$.

The proof of the proposition is given at the end of this section, after describing the encoding procedure. We emphasize that we use $t = \lceil \log_2 n \rceil$ throughout this section (as opposed to $\lceil \log_2(n+1) \rceil$ used for binary VT encoding) because the binary auxiliary sequence has length $(n-1)$.

2.5.1 Encoding procedure

The high level idea for mapping a k-bit message to a codeword $C \in \mathcal{VT}_{a,b}(n)$ is the following. Similar to the binary case, we reserve the t dyadic positions in the binary auxiliary sequence A_C to ensure that $\mathsf{syn}(A_C) = a$. Recall from (2.2.23) that each bit of A_C is determined by comparing two adjacent symbols of the q-ary sequence C. Therefore, to ensure that $\mathsf{syn}(A_C) = a$, in addition to reserving the symbols in the dyadic positions of C, we also place some restrictions on the symbols adjacent to the

dyadic positions. Finally, we use the first three symbols of C to ensure that $\mathsf{sum}(C) = b$. We explain the method in six steps with the help of the following running example.

Example 2.1. *Let $q = 8, n = 16$, and suppose that we wish to encode a binary message M to a codeword C in $\mathcal{VT}_{0,1}(16)$. We have $t = \lceil \log_2 n \rceil = 4$ and $\log_2 q = 3$. Therefore, from* (2.5.3) *the length of M is $k = 3(n-3t+3)+5(t-3)+2 = 28$ bits. Let*

$$M = 110\ 001\ 000\ 111\ 010\ 101\ 000\ 11100\ 11, \tag{2.5.6}$$

where the spacing indicates the bits corresponding to the three terms in (2.5.3).

Step 1. Let $\mathcal{S}$ be the set of pairs of symbols adjacent to a dyadic symbol, i.e.,

$$\mathcal{S} = \{(c_{2^j-1}, c_{2^j+1}), \quad \text{for } 2 \le j \le (t-1)\}. \tag{2.5.7}$$

There are $|\mathcal{S}| = (t-2)$ pairs of symbols in $\mathcal{S}$. Excluding c_0, the number of symbols in C that are neither in dyadic positions nor in $\mathcal{S}$ is

$$(n-1) - 2|\mathcal{S}| - t = (n-3t+3). \tag{2.5.8}$$

Assign the first $(n-3t+3)\log_2 q$ bits of the message M to these symbols, by converting each set of $\log_2 q$ bits to a q-ary symbol. This corresponds to the first term in (2.5.3).

In Example 2.1, $(n-3t+3)\log_2 q = 21$, and the representation of first 21 bits of M in $\mathbb{Z}_8$ is 6 1 0 7 2 5 0. Therefore the sequence C is

$$C = c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ 6\ c_7\ c_8\ c_9\ 1\ 0\ 7\ 2\ 5\ 0. \tag{2.5.9}$$

Step 2. In this step, we assign the remaining bits of the message to the symbols in $\mathcal{S}$. For a given dyadic position c_{2^j}, $j = 2, 3, \cdots, (t-1)$, we constrain the pair of adjacent symbols (c_{2^j-1}, c_{2^j+1}) to belong to the following set

$$\mathcal{T} = \{(r,l) \in \mathbb{Z}_q \times \mathbb{Z}_q : r \neq 0,\ l \neq (r-1)\}. \tag{2.5.10}$$

Via (2.5.10), we enforce $c_{2^j-1} \neq 0$ because if c_{2^j-1} were 0, then we necessarily have $c_{2^j} \ge c_{2^j-1}$ which constrains the value of α_{2^j} to 1. Recall from (2.2.23) that $\alpha_1 \ldots \alpha_{n-1}$ is the auxiliary sequence. However, α_{2^j} needs to be unconstrained in order to guarantee that any desired syndrome can be generated. Furthermore, we will see in Step 5 that if $c_{2^j+1} = c_{2^j-1} - 1$, then we may be unable to find a suitable symbol c_{2^j} compliant with

the restrictions induced by the auxiliary sequence. We therefore enforce the constraint $c_{2^j+1} \neq c_{2^j-1} - 1$ using (2.5.10). It is easy to see that $|\mathcal{T}| = (q-1)^2$.

Excluding the pair (c_3, c_5), there are $(t-3)$ pairs in $\mathcal{S}$. If we were encoding q-ary message symbols, each of these $(t-3)$ pairs could take any pair of symbols in $\mathcal{T}$. Since we are encoding message bits, we use a look up table to map $\lfloor \log_2 |\mathcal{T}| \rfloor = 2\log_2 q - 1$ bits to each of the pairs in $\mathcal{S}$ excluding (c_3, c_5). We thus map $(t-3)(2\log_2 q - 1)$ bits to the pairs in $\mathcal{S}$ excluding (c_3, c_5). This corresponds to the second term in (2.5.3).

Next, set $c_3 = q-1$. This choice is important as it will facilitate step 6. Since $(c_3, c_5) \in \mathcal{T}$, when $c_3 = q-1$, then c_5 has to be such that $c_5 \neq q-2$. Hence, there are $q-1$ possible values for c_5. As we are encoding a binary message, we map $\lfloor \log_2(q-1) \rfloor = \log_2 q - 1$ bits to c_5 using a look-up table. This corresponds to the third term in (2.5.3). We note that the two look-up tables used in this step have sizes at most $(q-1)^2$ and q, respectively. Thus, in steps one and two in total we have mapped the claimed k message bits to the symbols of C.

In Example 2.1, as seen from (2.5.9), (c_7, c_9) is the only pair in $\mathcal{S}$ other than (c_3, c_5). We can assign $2\log_2 q - 1 = 5$ bits to (c_7, c_9). After the first 18 message bits mapped in Step 1, the next five bits in M are 11100. Suppose that in our look-up table these bits correspond to the pair $(3,5)$. We then have $(c_7, c_9) = (3,5)$. Also, we fix $c_3 = q-1 = 7$, and the last two message bits determine c_5. The last two message bits are 11. Suppose that 3 is the corresponding symbol in the look-up table. We therefore set $c_5 = 3$. Therefore, we have

$$C = c_0 \; c_1 \; c_2 \; 7 \; c_4 \; 3 \; 6 \; 3 \; c_8 \; 5 \; 1 \; 0 \; 7 \; 2 \; 5 \; 0. \tag{2.5.11}$$

Up to this point, we have mapped our k message bits to a partially filled q-ary sequence. In the following steps we ensure that the resulting sequence lies in the correct VT code by carefully choosing remaining $(t+1)$ symbols to obtain the auxiliary sequence syndrome a and the modular sum b.

Step 3. In this step, we specify the bits in the non-dyadic locations of the auxiliary sequence A_C. Notice that according to (2.2.23), in order to define α_{2^j+1}, the value of c_{2^j} should be known. This is not the case here as the dyadic positions in C have been reserved to generate the required syndrome. To circumvent this issue, we determine α_{2^j+1} (for $1 < j < t$) by comparing c_{2^j+1} with c_{2^j-1} as follows:

$$\alpha_{2^j+1} = \begin{cases} 1 & \text{if } c_{2^j+1} \geq c_{2^j-1}, \\ 0 & \text{if } c_{2^j+1} < c_{2^j-1}. \end{cases} \tag{2.5.12}$$

As we shall show in step 5, we will be able to make these choices for the auxiliary sequence compatible with the definition of a valid auxiliary sequence in (2.2.23).

Next, since we have chosen $c_3 = q-1$, from the rule in (2.2.23) we have $\alpha_3 = 1$, regardless of what c_2 is. The other non-dyadic positions of the auxiliary sequence A_C can be filled in using (2.2.23), i.e., $\alpha_i = 1$ if $c_i \geq c_{i-1}$, and 0 otherwise.

For our example with C shown in (2.5.11), at the end of this step we have

$$A_C = \alpha_1\ \alpha_2\ 1\ \alpha_4\ 0\ 1\ 0\ \alpha_8\ 1\ 0\ 0\ 1\ 0\ 1\ 0. \tag{2.5.13}$$

Step 4. In this step, we use the binary encoding method described in Section 2.4 to find the bits in the dyadic positions $\alpha_{2^0}, \cdots, \alpha_{2^{t-1}}$ such that $\mathsf{syn}(A_C) = a$. With this, the auxiliary sequence A_C is fully determined.

In the example, we need to find $\alpha_1, \alpha_2, \alpha_4$ and α_8 such that $\mathsf{syn}(A_C) = 0$. First, we set the syndrome bits $\alpha_1 = \alpha_2 = \alpha_4 = \alpha_8 = 0$, and denote the resulting sequence by

$$A'_C = 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0. \tag{2.5.14}$$

Now, $\mathsf{syn}(A'_C) = 12$, and the deficiency $d = 0 - 12 \pmod{16} = 4$. The binary representation of d is $d_3 d_2 d_1 d_0 = 0100$. Hence, $\alpha_1 = \alpha_2 = \alpha_8 = 0$ and $\alpha_4 = 1$ will produce the desired syndrome. Summarizing, we have the following auxiliary sequence

$$A_C = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0. \tag{2.5.15}$$

Step 5. In this step, we specify the symbols of C in the dyadic positions (except c_1 and c_2). This will be done by ensuring that A_C is a valid auxiliary sequence consistent with the definition in (2.2.23). In particular, the choice of c_{2^j} for $j = 2, \cdots, t-1$, should be consistent with α_{2^j+1} and α_{2^j}. We ensure this by choosing c_{2^j} (for $1 < j < t$) as follows:

$$c_{2^j} = \begin{cases} c_{2^j-1} - 1 & \text{if } \alpha_{2^j} = 0, \\ c_{2^j-1} & \text{if } \alpha_{2^j} = 1. \end{cases} \tag{2.5.16}$$

From the definition in (2.2.23), this choice is consistent with α_{2^j}. Now we show that it is also consistent with α_{2^j+1}. If $\alpha_{2^j+1} = 1$, then according to (2.5.12), $c_{2^j+1} \geq c_{2^j-1}$; then the choice of c_{2^j} in (2.5.16) always guarantees that $c_{2^j-1} \geq c_{2^j}$, and thus $c_{2^j+1} \geq c_{2^j}$. Next suppose that $\alpha_{2^j+1} = 0$. Then according to (2.5.12), $c_{2^j+1} < c_{2^j-1}$. We need to verify that $c_{2^j+1} < c_{2^j}$ Now, if $\alpha_{2^j} = 1$, then $c_{2^j} = c_{2^j-1}$ and $c_{2^j+1} < c_{2^j}$. Also, if $\alpha_{2^j} = 0$, from (2.5.16) we have $c_{2^j} = c_{2^j-1} - 1$. Since symbols adjacent to dyadic positions (c_{2^j-1}, c_{2^j+1}) are chosen from $\mathcal{T}$ (see step 2), then $c_{2^j+1} \neq c_{2^j-1} - 1$. Thus,

we have that $c_{2^j+1} < c_{2^j-1} - 1 = c_{2^j}$. Therefore, in either case the choice is consistent with (2.2.23).

For the example, using (2.5.16) and (2.5.15) we obtain

$$C = c_0\ c_1\ c_2\ 7\ 7\ 3\ 6\ 3\ 2\ 5\ 1\ 0\ 7\ 2\ 5\ 0. \tag{2.5.17}$$

Step 6. Finally, we need to find c_0, c_1 and c_2 that are compatible with $\alpha_1, \alpha_2, \alpha_3$ (the first three bits of the auxiliary sequence), and such that $\mathsf{sum}(C) = b$. Let

$$w \triangleq b - \sum_{i=3}^{n-1} c_i \pmod q. \tag{2.5.18}$$

Hence we need $c_0 + c_1 + c_2 = w \pmod q$. We will show that when $q \geq 4$, we can find three distinct integers (x, y, z) such that $0 \leq x < y < z < q$ and $x + y + z = w \pmod q$. We will assign these numbers to c_0, c_1 and c_2. Also recall that we set $c_3 = q - 1$; hence we always have $x, y, z \leq c_3$, which is consistent with $\alpha_3 = 1$.

The triplet with smallest numbers that we can choose is $x = 0, y = 1, z = 2$. For this choice, $w = 0 + 1 + 2 = 3 \pmod q$. By increasing z from 2 to $q - 1$ with $x = 0$ and $y = 1$, we can produce any value of w from 3 to $q - 1$ as well as $w = 0$. Finally, the only remaining values are $w = 1, 2$. To obtain these values, we choose x, y, z as follows.

1. $\underline{w = 1}$: Choose $x = 0, y = 2, z = q - 1$.

2. $\underline{w = 2}$: Choose $x = 1, y = 2, z = q - 1$.

Hence, we have shown that for $q \geq 4$, any $w \in \mathbb{Z}_q$ can be expressed as the $\pmod q$ sum of three distinct elements of $\mathbb{Z}_q$. Assigning these elements to c_0, c_1, c_2 in the order required by the auxiliary sequence completes the encoding procedure. We now have $\mathsf{sum}(C) = b$ and $\mathsf{syn}(A_C) = a$, and thus $C \in \mathcal{VT}_{a,b}(n)$ as required.

In our example, from (2.5.17) we have

$$\sum_{i=3}^{15} c_i = 48 = 0 \pmod 8, \tag{2.5.19}$$

and $b = 1$. Therefore we need $c_0 + c_1 + c_2 = 1 \pmod 8$. We have $\alpha_1 = \alpha_2 = 0$ so $c_0 > c_1 > c_2$ is the correct order. We therefore assign $c_0 = 7$, $c_1 = 2$, and $c_2 = 0$ to obtain the codeword.

$$C = 7\ 2\ 0\ 7\ 7\ 3\ 6\ 3\ 2\ 5\ 1\ 0\ 7\ 2\ 5\ 0 \in \mathcal{VT}_{0,1}(16). \tag{2.5.20}$$

It can be verified that $\mathsf{sum}(C) = 1$, and the auxiliary sequence syndrome $\mathsf{syn}(A_C) = 0$.

2.5.2 The case where q is not a power of two

When $\log_2 q$ is not an integer, the main difference is that we map longer sequences of bits to sequences of q-ary symbols. Recall that in step 1, we determine $(n-3t+3)$ symbols of the q-ary codeword. One can map $\lfloor (n-3t+3)\log_2 q \rfloor$ bits to these $(n-3t+3)$ symbols using standard methods to convert an integer expressed in base 2 into base q. In the second step, as described earlier we can map $\lfloor \log_2 (q-1)^2 \rfloor$ bits to $(t-3)$ pairs in $\mathcal{S}$ (excluding (c_3, c_5)). Moreover $\lfloor \log_2 (q-1) \rfloor$ bits can be mapped to c_5. Therefore, in total we can map k bits to a q-ary VT codeword of length n, where

$$k = \lfloor (n-3t+3)\log_2 q \rfloor + (t-3)\lfloor \log_2 (q-1)^2 \rfloor + \lfloor \log_2 (q-1) \rfloor \tag{2.5.21}$$

$$\geq n\log_2 q - t(\log_2 q + 2) - (2\log_2 q - 4). \tag{2.5.22}$$

For $q \geq 4$, the remaining steps are identical to the case where q is a power of two. The case of $q = 3$ is slightly different and it is discussed below.

2.5.3 Encoding for $q = 3$

For $q = 3$, we need to slightly modify the proposed algorithm. The first step is as described in Section 2.5.2. The difference in the second step is that we do not embed data in c_5 and simply choose $c_5 = c_3 = 2$. Steps three to five remain the same. In the sixth step, we compute w as in (2.5.18), and choose c_0, c_1, c_2 as follows depending on the values of α_0 and α_1:

1. $\underline{\alpha_1 = \alpha_2 = 1}$: Choose $c_2 = c_1 = 2$ and $c_0 = w - 4 \pmod 3$.
2. $\underline{\alpha_1 = 1, \alpha_2 = 0}$: Choose $c_2 = 1$ and $c_1 = 2$ and $c_0 = w - 3 \pmod 3$.
3. $\underline{\alpha_1 = 0, \alpha_2 = 1}$: Choose $c_2 = 2$. If $w = 1$, then $c_1 = 0, c_0 = 2$. If $w = 0$, then $c_1 = 0, c_0 = 1$. If $w = 2$, then $c_1 = 1, c_0 = 2$.

The only remaining case is when $\alpha_1 = \alpha_2 = 0$. For this case, we need to change c_3 and c_4, and also the first three bits of A_C. Since c_3 has been set to 2, the first three bits of A_C in this case are 001. If we change these three bits to 110, $\mathsf{syn}(A_C)$ will remain unchanged. We therefore set $\alpha_1 = \alpha_2 = 1$ and $\alpha_3 = 0$. Now we update c_0, c_1, c_2, c_3 to be compatible with the new auxiliary sequence. Set $c_3 = 1$, recall that $c_5 = 2$ so we still have $c_5 \geq c_3$ and hence this change will not affect α_5. Update c_4 according to (2.5.16).

Set $c_2 = c_1 = 2$, and $c_0 = w - 2 \pmod 3$. Now we have $c_3 < c_2$ which is consistent with $\alpha_3 = 0$. Also $c_2 \geq c_1 \geq c_0$ is consistent with $\alpha_1 = \alpha_2 = 1$.

Hence, for $q = 3$, we have mapped $k = \lfloor \log_2 3(n - 3t + 3) \rfloor + 2(t - 3)$ bits to a q-ary codeword C. This induces following rate:

$$R = \frac{\lfloor \log_2 3\ (n - 3\lceil \log_2 n \rceil + 3) \rfloor}{n} + \frac{2(\lceil \log_2 n \rceil - 3)}{n} \tag{2.5.23}$$

$$\geq \log_2 3 - \frac{2.76\ \lceil \log_2 n \rceil}{n} - \frac{2.25}{n} \tag{2.5.24}$$

Similarly in Proposition 2, we can show that for $q = 3$ there are at least $2^{2(t-3)} 3^{n-3t+3}$ codewords in each of the VT codes.

2.5.4 Proof of Proposition 2.2

The result can be directly derived from steps one and two of our encoding method by mapping sequences of q-ary message symbols (rather than sequences of message bits) to distinct codewords in $|\mathcal{VT}_{a,b}(n)|$. In step 1, we can assign $(n - 3t + 3)$ arbitrary symbols to positions that are neither dyadic nor in $\mathcal{S}$. There are q^{n-3t+3} ways to choose these symbols. Then in step two, we can choose $(q-1)^2$ pairs for each of the $(t-3)$ specified pairs of positions; furthermore, there are $(q-1)$ choices for c_5. According to steps 3 to 6, we can always choose the remaining symbols such that resulting codeword lies in $\mathcal{VT}_{a,b}(n)$. Therefore, we can map $q^{n-3t+3}(q-1)^{2t-5}$ different sequences of message symbols to distinct codewords in $\mathcal{VT}_{a,b}(n)$. This yields the lower bound on $|\mathcal{VT}_{a,b}(n)|$.

Chapter 3

Segmented model

3.1 Introduction

In this chapter the problem of constructing codes for *segmented* edit channels is considered, where the channel input sequence is implicitly divided into disjoint segments. Each segment can undergo at most one edit, which can be either an insertion or a deletion. There are no segment markers in the received sequence.

This model, introduced by Liu and Mitzenmacher [27], is a simplified version of the general edit channel, where the insertions and deletions can be arbitrarily located in the input sequence. The assumption of segmented edits not only simplifies the coding problem, but is also likely to hold in many edit channels that arise in practice, e.g., in data storage and in sequenced genomic data, where the number of edits is small compared to the length of the input sequence. As explained in [27], when edits (deletions or insertions of symbols) occur due to timing mismatch between the data layout and the data-reading mechanism, there is often a minimum gap between successive edits. The segmented edit model includes such cases, though it also allows for nearby edits that cross a segment boundary. Furthermore, a complete understanding of the segmented edit model may provide insights into the open problem of constructing efficient, high-rate codes for general edit channels. As it is shown in this chapter, the segmented edit assumption allows for the construction of low-complexity, zero error codes with the optimal rate scaling for any finite alphabet. The segmented deletion channel is also studied in [102], where the authors found capacity bounds for the channel in terms of p_d, the probability of having a deletion in a given segment. They also suggested a practical encoding scheme. We highlight that here we are assuming the zero error model where in [102] small probability of error is allowed.

Here three examples are considered to illustrate the model. For simplicity, we consider a binary alphabet and assume that the segment length, denoted by b, is 3 in each case.

1) *Segmented Deletion Channel*: Each segment can undergo at most one deletion; no insertions occur. Consider the following pair of input and output sequences:

$$X = 01\underline{1}10\underline{0}010 \longrightarrow Y = 0110010, \tag{3.1.1}$$

with the underlined bits in X being deleted by the channel to produce the output sequence Y. It is easily verified that many other input sequences could have produced the same output sequence, e.g., $01\underline{0}100\underline{0}10$, $01\underline{0}10\underline{1}010$, $011\,00\underline{0}\,1\underline{0}0$ etc. The receiver has no way of distinguishing between these candidate input sequences. In particular, despite knowing the segment length and that deletions occurred, it does not know in which two segments the deletions occurred.

2) *Segmented Insertion Channel*: Each segment can undergo at most one insertion; no deletions occur. The inserted bit can be placed anywhere within the segment, including before the first bit or after the last bit of the segment. For example, consider

$$X = 011\,100\,010 \longrightarrow Y = 011\underline{10}1000\underline{1}0, \tag{3.1.2}$$

with the underlined bits in Y indicating the insertions. Two inserted bits can appear between two segments whenever there is an insertion after the last bit of first segment and before the first bit of the next segment.

3) *Segmented Insertion-Deletion Channel*: This is the most general case, where a segment could undergo either an insertion or a deletion, or remain unaffected. For example, consider

$$X = 01\underline{1}\,100\,010 \longrightarrow Y = 01\underline{0}1000\underline{1}0, \tag{3.1.3}$$

with the underlined bits on the left indicating deletions, and the underlined bits on the right indicating insertions. Unlike the previous two cases, the receiver cannot even infer the exact number of edits that have occurred. In the example above, an input sequence 9 bits (three segments) long could result in a 10-bit output sequence in two different ways: either via one segment with an insertion, or via two segments with insertions and the other with a deletion.

The above examples demonstrate that one cannot reduce the problem to one of correcting one edit in a b-bit input sequence. To see this, consider the example in (3.1.1), and suppose that we used a single-deletion correcting code for each segment.

Such a code would declare the first three bits of Y to be the first segment of X, which would result in incorrect decoding of the following segments.

In this chapter, zero error codes for each of the three segmented edit models above is constructed, for any finite alphabet of size $q \geq 2$. The codes can easily be constructed even for relatively large segment sizes (several tens), and can be decoded segment-by-segment in linear time. Moreover, the proposed codes have rate R of at least

$$R \geq \log_2 q - \frac{1}{b}\log_2(b+1) - \frac{\kappa}{b}\log_2 q, \tag{3.1.4}$$

where the constant κ is at most 2.5 for the segmented deletion channel, 4 for the segmented insertion channel, and 8 for the segmented insertion-deletion channel. (Slightly better bounds on κ are obtained for the binary case $q = 2$.)

An upper bound is also derived in terms of the segment length b on the maximum rate of any code for the segmented edit channel. This upper bound (Theorem 1) shows that the rate R of any zero error code with code length n satisfies

$$R \leq \log_2 q - \frac{1}{b}\log_2 b - \frac{1}{b}\log_2(q-1) + \frac{1}{b} + \frac{\log_2(2q)}{n} + O\left(\frac{\ln b}{b^{4/3}}\right). \tag{3.1.5}$$

Comparing (3.1.4) and (3.1.5), we see that the rate scaling for the proposed codes is the same as that of the maximal code with the rate penalty being $O(1/b)$. The bound we find in Theorem 1 is non-asymptotic, however it will not be close to the expression in (3.1.4) even for large values of b.

The starting point for the code constructions is the VT codes studied in the previous chapter. In our constructions, the codewords in each segment are drawn from subsets of VT codes satisfying certain prefix/suffix conditions, which are carefully chosen to enable fast segment-by-segment VT decoding.

3.1.1 Previous work on segmented channels

The segmented edit assumption places a restriction on the kinds of edit patterns that can be introduced in the input sequence. Other models with restrictions on edit patterns include the forbidden symbol model considered in [103].

We now highlight some similarities and differences with the codes proposed by Liu and Mitzenmacher in [27] for the binary segmented deletion and segmented insertion channels.

Code construction: The code in [27] is a binary segment-by-segment code specified via sufficient conditions [27, Theorems 2.1, 2.2] that ensure that as decoding proceeds,

there are at most two choices for the starting position of the next undecoded segment. Finding the maximal code that satisfies these conditions corresponds to an independent set problem, which is challenging for large b. The maximal code satisfying these conditions was reported in [27] for $b = 8, 9$. For larger b, a greedy algorithm was used to find a set of codewords satisfying the conditions. It was also suggested that one could restrict the code to a subset of VT codes that satisfy the sufficient conditions.

In comparison, our codes are directly defined as subsets of VT codes that satisfy certain simple prefix/suffix conditions; these conditions are different from those in [27]. Our conditions ensure that upon decoding each segment, there is no ambiguity in the starting position of the next segment. These subsets of VT codes are relatively simple to enumerate, so it is possible to find the largest code satisfying our conditions for b of the order of several tens. Table 3.1 lists the number of codewords per segment for the three segmented edit channels for $q = 2$ and lengths up to $b = 24$. For the segmented deletion and segmented insertion-deletion channels, another difference from the code in [27] is that our codebook for each segment is chosen based on the final bit of the previous segment.

Rate: The VT subsets and sufficient conditions we define allow us to obtain a lower bound of the form (3.1.4) on the rate of our code for any segment length b. Though the *maximal* codes satisfying the Liu-Mitzenmacher conditions have rate very close to the largest possible rate with segment-by-segment decoding, finding the maximal code satisfying these conditions is computationally hard, so one has to resort to greedy algorithms to construct codes for larger b. This is reflected in the rate comparison: for $b = 8, 9$, the optimal Liu-Mitzenmacher code for segmented deletions is larger than our code (12, 20 vs. 8, 13 codewords). However for $b = 16$, the code obtained in [27] using a greedy algorithm has 652 codewords, whereas our code has 964 codewords, as shown in Table 3.1. For large b, our codes are nearly optimal since the rate penalty decays as κ/b.

For the segmented insertion channel, it is shown in Section 3.5.3 that our code construction satisfies the sufficient conditions specified [27]. The lower bound on the rate of our code affirmatively answers the conjecture in [27] that the rates of the maximal codes satisfying the sufficient conditions increases with b.

Encoding and decoding complexity: As discussed in the previous chapter, an efficient encoding for VT codes (both binary and non-binary) is available, and hence our codes can also be efficiently encoded even for large segment sizes b, without the need for look-up tables. As segment-by-segment decoding is enforced by design, the decoding complexity grows linearly with the number of segments for both our codes and those in

Table 3.1: Number of codewords per segment of the proposed codes. Lower bounds computed from (3.4.3), (3.5.4), and (3.6.5) are given in brackets.

b	Deletion	Insertion	Insertion-Deletion
8	8 (8)	6 (6)	1 (1)
9	13 (13)	10 (10)	2 (1)
10	24 (24)	18 (18)	2 (1)
11	44 (43)	33 (32)	2 (2)
12	79 (79)	60 (59)	4 (3)
13	147 (147)	111 (110)	6 (5)
14	276 (274)	208 (205)	12 (9)
15	512 (512)	384 (384)	16 (16)
16	964 (964)	724 (723)	34 (31)
17	1,824 (1,821)	1,368 (1,366)	59 (57)
18	3,450 (3,450)	2,588 (2,587)	114 (108)
19	6,554 (6,554)	4,916 (4,916)	206 (205)
20	12,490 (12,484)	9,369 (9,363)	399 (391)
21	23,832 (23,832)	17,847 (17,874)	746 (745)
22	45,591 (45,591)	34,194 (34,193)	1,435 (1,425)
23	87,392 (87,382)	65,544 (65,536)	2,736 (2,731)
24	167,773 (167,773)	125,831 (125,830)	5,257 (5,243)

[27]. Within each segment, the decoding complexity of our code is also linear in b, since VT codes can be decoded with linear complexity [16]. In general, for each segment, the maximal Liu-Mitzenmacher codes have to be decoded via look-up tables, in which case the complexity is exponential in b. Using subsets of VT codes was suggested in [27] as a way to reduce the decoding complexity.

Finally, we remark that codes proposed in this chapter are the first for the binary segmented insertion-deletion model, and for all the non-binary segmented edit models.

3.1.2 Organization of the chapter

The remainder of the chapter is organized as follows. In Section 3.2, we formally define the channel model. In Section 3.3, we derive an upper bound on the rate of any code for a segmented edit channel, in terms of the segment length. In Sections 3.4, 3.5, and 3.6, we present our code constructions for the segmented deletion channel, segmented insertion channel, and the segmented insertion-deletion channel, respectively. For each model, we first treat the binary case to highlight the key ideas, and then extend the construction to general non-binary alphabets.

3.2 Channel model

The channel input sequence is denoted by $X = x_1 x_2 \cdots x_n$, with $x_i \in \mathcal{X}$ for $i = 1, \cdots, n$, where $\mathcal{X} = \{0, \cdots, q-1\}$ is the input alphabet, with $q \geq 2$. The channel input sequence is divided into k segments of b symbols each. We denote the subsequence of X, from index i to index j, with $i < j$ by $X(i:j) = x_i x_{i+1} \cdots x_j$. The i-th segment of X is denoted by $S_i = s_{i,1} \cdots s_{i,b} = X(b(i-1)+1 : bi)$ for $i = 1, \cdots, k$.

In the segmented deletion channel, the channel output $Y = Y(1:m) = y_1 \cdots y_m$, with $m \leq n$ is obtained by deleting at most one symbol in each segment, i.e., at most one symbol in S_i, $i = 1, \ldots, k$, is deleted. Similarly, in the segmented insertion channel, the channel output $Y = y_1 \ldots y_m$, with $m \geq n$ is obtained by inserting at most one symbol per segment. In the segmented insertion-deletion channel, the channel output is such that each segment S_i, $i = 1, \ldots, k$ undergoes at most one edit. In all cases, we assume that the decoder knows k and b, but not the segment boundaries.

We consider coded communication using a code $\mathcal{C} = \{X^{(1)}, \ldots, X^{(M)}\} \subseteq \mathcal{X}^n$ of length n, M codewords and rate $R = \frac{1}{n} \log_2 M$. We consider segment-by-segment coding, where M_s is the number of codewords per segment. The overall code of length $n = kb$ has $(M_s)^k$ codewords, and rate

$$R = \frac{1}{n} \log_2 (M_s)^k \tag{3.2.1}$$

$$= \frac{1}{b} \log_2 M_s. \tag{3.2.2}$$

The decoder produces an estimate $\hat{X}$ of the transmitted sequence. We denote the corresponding segment estimates by $\hat{S}_i = \hat{s}_{i,1} \cdots \hat{s}_{i,b}$, for $i = 1, \ldots, k$. Thus $\hat{X} = (\hat{S}_1, \ldots, \hat{S}_k)$. We consider zero error codes that always ensure the recoverability of the transmitted sequence, i.e., codes for which $\hat{X} = X$.

3.3 Upper bound on rate

In this section, we derive an upper bound on the rate of any code for q-ary segmented edit channels, for $q \geq 2$. The upper bound is valid for all zero error codes, including those that cannot be decoded segment-by-segment.

Theorem 1. *For each of the three segmented edit models, with segment length b, the rate R of any zero error code with code length $n = kb$ satisfies*

$$R \leq \log_2 q - \frac{1}{b}\log_2 b - \frac{1}{b}\log_2(q-1) + \frac{1}{b} + \frac{\log_2(2q)}{kb} + O\left(\frac{\ln b}{b^{4/3}}\right). \tag{3.3.1}$$

Remarks:

1. In the theorem, the alphabet size q is held fixed as the segment size b grows. The number of segments per codeword, k, is arbitrary, and need not grow with b.

2. The theorem is obtained via non-asymptotic bounds on the size and the rate of any zero error code. These bounds, given in (3.3.30)–(3.3.35), may be of independent interest.

3. The dominant terms in the upper bound may be interpreted as follows for the case of the segmented deletion channel. For a noiseless q-ary input channel the rate is $\log_2 q$ bits/transmission. The $\log_2 b/b$ term corresponds to a penalty required to convey the run in which the deletion occurred in each segment. The $\log_2(q-1)/b$ term is a penalty required to convey the value of the deleted symbol.

Proof of Theorem 1. We give the proof for the segmented deletion model with segment length b. The argument for the segmented insertion model is similar.

The proof technique is similar to that used by Tenengolts in [17, Theorem 2]. The high-level idea is the following. The codewords are split into two groups: the first group contains the codewords in which a large majority of segments have at least $b\frac{(q-1)}{q} - O(b^{2/3})$ runs. The other group contains the remaining codewords. As b grows larger, the fraction of length b sequences with close to $b\frac{(q-1)}{q}$ runs (the 'typical' value) approaches 1. So we carefully bound the number of codewords in the first group, while the number of codewords in the second group can be bounded by a direct counting argument.

Consider a code $\mathcal{C}$ of length $n = kb$, i.e., each codeword has k segments of length b. Let $M = |C| = 2^{nR}$ denote the size of the code. For integers $r \geq 0$ and $0 \leq l \leq k$, define $\mathcal{M}(r,l) \subset \mathcal{C}$ as the set of the codewords that have exactly l segments with more than r runs. Let $M(r,l) = |\mathcal{M}(r,l)|$. Note that for any $r \geq 0$, we have

$$\sum_{l=0}^{k} M(r,l) = M. \tag{3.3.2}$$

For any $l \leq k$ and a codeword $x \in \mathcal{M}(r,l)$, let $\rho_l(x)$ denote the number of distinct sequences of length $(n-l)$ by deleting exactly l symbols from x, following the segmented assumption. We then have

$$(r-1)^l \leq \rho_l(x). \tag{3.3.3}$$

To show (3.3.3), we only need to consider $r \geq 3$ as the inequality is trivial for $r \leq 2$. Considering the l segments that each have at least $(r+1)$ runs. There are at least $(r-1)^l$ ways of choosing one run from each segment so that the l chosen runs are non-adjacent. For each such choice of l non-adjacent runs, we get a distinct subsequence of length $(n-l)$ by deleting one symbol from each run. This proves (3.3.3).

Since $\mathcal{C}$ is a zero error code, for two distinct codewords $x_1, x_2 \in \mathcal{M}(r,l)$, the set of length $(n-l)$ sequences obtained via l deletions (in a segmented manner) from x_1 must be distinct from the corresponding set for codeword x_2. We therefore have

$$q^{n-l} \geq \sum_{x \in \mathcal{M}(r,l)} \rho_l(x) \tag{3.3.4}$$

$$\overset{(a)}{\geq} \sum_{x \in \mathcal{M}(r,l)} (r-1)^l \tag{3.3.5}$$

$$= M(r,l)(r-1)^l, \tag{3.3.6}$$

where (a) is obtained from (3.3.3). We therefore obtain

$$M(r,l) \leq \frac{q^{n-l}}{(r-1)^l}. \tag{3.3.7}$$

Fix $\alpha \in (0,1)$. Summing (3.3.7) over $\alpha k \leq l \leq k$, we obtain

$$\sum_{l \geq \alpha k} M(r,l) \leq \sum_{l \geq \alpha k} \frac{q^{n-l}}{(r-1)^l} \tag{3.3.8}$$

$$\leq \frac{2q^{n-\alpha k}}{(r-1)^{\alpha k}}. \tag{3.3.9}$$

Now define r' as follows

$$r' = \frac{(q-1)}{q} b - \sqrt{\frac{2\kappa(q-1)b \ln b}{q}}, \tag{3.3.10}$$

where $\kappa > \frac{\log(2q)}{\log b}$ will be specified later. We choose $r = \lceil r' \rceil$. Using this r in (3.3.7), and noting that $n = kb$, we have

$$\sum_{l \geq \alpha k} M(r,l) \leq \frac{2q^{kb-\alpha k}}{(r-1)^{\alpha k}} \tag{3.3.11}$$

$$\leq \frac{2q^{kb-\alpha k}}{(r'-1)^{\alpha k}} \tag{3.3.12}$$

$$= \frac{2q^{kb}}{(b(q-1))^{\alpha k}\left(1-\sqrt{\frac{2\kappa q \ln b}{(q-1)b}}-\frac{q}{(q-1)b}\right)^{\alpha k}}. \tag{3.3.13}$$

For $l < \alpha k$, we use the looser bound

$$M(r,l) \leq \binom{k}{k-1}\left[q\sum_{t=0}^{r-1}(q-1)^t\binom{b-1}{t}\right]^{k-l} q^{bl}, \tag{3.3.14}$$

which is obtained as follows. We first choose the $(k-l)$ segments with at most r runs. Then, a segment with t runs is determined by the choice of the first symbol, and the starting positions and values of the next $(t-1)$ runs. There are q choices for the first symbol, $\binom{b-1}{t-1}$ choices for the starting position of the next $(t-1)$ runs, and $(q-1)^{t-1}$ choices for the values of these runs. Therefore, the number of possible length b sequences with at most r runs is $q\sum_{t=1}^{r}\binom{b-1}{t-1}(q-1)^{t-1} = q\sum_{t=0}^{r-1}\binom{b-1}{t}(q-1)^t$. We then obtain (3.3.14) by noting that: i) there are $(k-l)$ segments with at most r runs, and ii) there are at most q^{bl} choices for the remaining l segments. We write the right hand side of (3.3.14) as

$$\binom{k}{k-1}\left[q\sum_{t=0}^{r-1}(q-1)^t\binom{b-1}{t}\right]^{k-l} q^{bl}$$

$$= \binom{k}{k-1}\left[q^{b+1}\sum_{t=0}^{r-1}\left(1-\frac{1}{q}\right)^t\left(\frac{1}{q}\right)^{b-t}\binom{b-1}{t}\right]^{k-l} q^{bl} \tag{3.3.15}$$

$$\leq 2^k q^{bk+k-l}\left[\sum_{t=0}^{r-1}\left(1-\frac{1}{q}\right)^t\left(\frac{1}{q}\right)^{b-t}\binom{b-1}{t}\right]^{k-l}. \tag{3.3.16}$$

Now we prove that

$$\sum_{t=0}^{r-1}\left(1-\frac{1}{q}\right)^t\left(\frac{1}{q}\right)^{b-t}\binom{b-1}{t} \leq \frac{1}{b^\kappa}. \tag{3.3.17}$$

Let U be a Binomial$\left(b, \frac{q-1}{q}\right)$ random variable, with mean $\mu = \frac{b(q-1)}{q}$. Then, using a standard Chernoff bound for a binomial random variable (see, for example [104, Theorem 4.5]), we have for any $\epsilon > 0$:

$$\mathbb{P}(U \leq \mu(1-\epsilon)) \leq \exp\left(\frac{-\mu\epsilon^2}{2}\right). \tag{3.3.18}$$

Choosing $\epsilon = \sqrt{\frac{2\kappa q \ln b}{(q-1)b}}$, we have

$$\mu(1-\epsilon) = \frac{b(q-1)}{q} - \sqrt{\frac{2\kappa(q-1)b\ln b}{q}} \tag{3.3.19}$$

$$= r', \tag{3.3.20}$$

where r' is defined in (3.3.10). Using this in (3.3.18), we obtain

$$\mathbb{P}(U \leq r') = \mathbb{P}(U \leq \mu(1-\epsilon)) \tag{3.3.21}$$

$$\leq \exp\left(\frac{-\mu\epsilon^2}{2}\right) \tag{3.3.22}$$

$$= b^{-\kappa}, \tag{3.3.23}$$

where the last equality is obtained by substituting the values of μ and ϵ. Finally, note that

$$\begin{aligned}\mathbb{P}(U \leq r') &\geq \mathbb{P}(U \leq (r-1)) \\ &= \sum_{t=0}^{r-1} \left(1 - \frac{1}{q}\right)^t \left(\frac{1}{q}\right)^{b-t} \binom{b}{t} \\ &\geq \sum_{t=0}^{r-1} \left(1 - \frac{1}{q}\right)^t \left(\frac{1}{q}\right)^{b-t} \binom{b-1}{t}.\end{aligned} \tag{3.3.24}$$

Combining (3.3.24) and (3.3.23) yields the desired inequality.

Using (3.3.17) to bound (3.3.16), and then substituting in (3.3.14), we obtain

$$M(r,l) \leq \frac{2^k q^{bk+k-l}}{b^{\kappa(k-l)}}. \tag{3.3.25}$$

Summing over $0 \le l < \alpha k$ and considering $\kappa > \frac{\log(2q)}{\log b}$, we obtain

$$\sum_{l<\alpha k} M(r,l) \le \frac{2^k q^{(b+1)k}}{b^{\kappa k}} \sum_{l<\alpha k} \left(\frac{b^\kappa}{q}\right)^l \tag{3.3.26}$$

$$\le \frac{2^k q^{(b+1-\alpha)k+1}}{b^{\kappa(1-\alpha)k}}. \tag{3.3.27}$$

Combining the bounds in (3.3.13) and (3.3.27), we have

$$M = \sum_{l=0}^{k} M(r,l) \tag{3.3.28}$$

$$\le \frac{2q^{kb}}{(b(q-1))^{\alpha k}\left(1-\sqrt{\frac{2\kappa q \ln b}{(q-1)b}}-\frac{q}{(q-1)b}\right)^{\alpha k}} + \frac{2^k q^{(b+1-\alpha)k+1}}{b^{\kappa(1-\alpha)k}} \tag{3.3.29}$$

$$\le 2\max\{T_1, T_2\} \tag{3.3.30}$$

where

$$T_1 = \frac{2q^{kb}}{(b(q-1))^{\alpha k}\left(1-\sqrt{\frac{2\kappa q \ln b}{(q-1)b}}-\frac{q}{(q-1)b}\right)^{\alpha k}}, \tag{3.3.31}$$

$$T_2 = \frac{2^k q^{(b+1-\alpha)k+1}}{b^{\kappa(1-\alpha)k}}. \tag{3.3.32}$$

Therefore the rate can be bounded as

$$R = \frac{\log M}{kb} \le \frac{1}{kb} + \max\left\{\frac{\log T_1}{kb}, \frac{\log T_2}{kb}\right\}. \tag{3.3.33}$$

From (3.3.31) and (3.3.32), we have

$$\frac{\log T_1}{kb} \le \log_2 q - \frac{\alpha \log_2(b(q-1))}{b} - \frac{\alpha}{b}\log_2\left(1-\sqrt{\frac{2\kappa q \ln b}{(q-1)b}}-\frac{q}{(q-1)b}\right) + \frac{1}{kb}, \tag{3.3.34}$$

$$\frac{\log T_2}{kb} \le \log_2 q - \frac{\kappa(1-\alpha)\log_2 b}{b} + \frac{(1-\alpha)\log_2 q}{b} + \frac{1}{b} + \frac{\log_2 q}{kb}. \tag{3.3.35}$$

Now choose α and κ as follows:

$$\alpha = 1 - \frac{1}{\sqrt[3]{b}}, \tag{3.3.36}$$

$$\kappa = \frac{\alpha}{1-\alpha}\frac{\log_2(b(q-1))}{\log_2 b} \tag{3.3.37}$$

$$= \left(\sqrt[3]{b}-1\right)\frac{\log_2(b(q-1))}{\log_2 b}. \tag{3.3.38}$$

Note that we have $\alpha \to 1$ and $\frac{2\kappa q \ln b}{(q-1)b} \to 0$ as $b \to \infty$. Using the fact that $\ln(1/(1-x)) \leq 2x$ for $x \in (0,1/2]$ in (3.3.34), we have the following bound on T_1 for sufficiently large b:

$$\begin{aligned}\frac{\log T_1}{kb} &\leq \log_2 q - \frac{\alpha \log_2(b(q-1))}{b} + \frac{1}{kb} \\ &\quad + \frac{2\alpha}{b\ln 2}\left(\sqrt{\frac{2\kappa q \ln b}{(q-1)b}} + \frac{q}{(q-1)b}\right) \\ &= \log_2 q - \frac{\log_2(b(q-1))}{b} + \frac{\log_2(b(q-1))}{b^{4/3}} \\ &\quad + \frac{1}{kb} + \frac{2\alpha}{b\ln 2}\left(\sqrt{\frac{2\kappa q \ln b}{(q-1)b}} + \frac{q}{(q-1)b}\right).\end{aligned} \tag{3.3.39}$$

Also substituting the values of α, κ from (3.3.36) and (3.3.38) in (3.3.35), we have

$$\begin{aligned}\frac{\log T_2}{kb} &\leq \log_2 q - \frac{\log_2(b(q-1))}{b} + \frac{1}{b} + \frac{\log_2(b(q-1))}{b^{4/3}} \\ &\quad + \frac{\log_2 q}{b^{4/3}} + \frac{\log_2 q}{kb}.\end{aligned} \tag{3.3.40}$$

Finally, substituting the values of α, κ into the last term in (3.3.39), it can be seen that this term is $O(\sqrt{\ln b}/b^{4/3})$, which yields the desired result. □

3.4 Segmented deletion correcting codes

In this section, we show how to construct a segment-by-segment zero error code for the segmented deletion channel. For simplicity, we first introduce binary codes and explain the binary decoder. We then highlight the differences in the non-binary case.

If the decoder knew the segment boundaries, then simply using a VT code for each segment would suffice. Since the segment boundaries are not known, recall from the example in (3.1.1) that this approach is inadequate if segment-by-segment decoding is

to be used. Our construction chooses a subset of a VT code for each segment, with prefixes determined by the last symbol of the previous segment.

3.4.1 Binary code construction

For $0 \leq a \leq b$, define the following sets.

$$\begin{aligned} \mathcal{A}_a^0 &\triangleq \{S \in \{0,1\}^b : \mathsf{syn}(S) = a,\ s_1 s_2 = 00\}, \\ \mathcal{A}_a^1 &\triangleq \{S \in \{0,1\}^b : \mathsf{syn}(S) = a,\ s_1 s_2 = 11\}. \end{aligned} \tag{3.4.1}$$

For $c \in \{0,1\}$, the set $\mathcal{A}_a^c \subseteq \mathcal{VT}_a(b)$ is the set of VT codewords that start with prefix cc. We now choose the sets with the largest number of codewords, i.e., we choose $\mathcal{A}_{a_0}^0$ and $\mathcal{A}_{a_1}^1$ where we define

$$a_0 = \arg\max_{0 \leq a \leq b} |\mathcal{A}_a^0|, \quad a_1 = \arg\max_{0 \leq a \leq b} |\mathcal{A}_a^1|. \tag{3.4.2}$$

By defining $M_s = \min\{|\mathcal{A}_{a_0}^0|, |\mathcal{A}_{a_1}^1|\}$, we can now construct $\mathcal{A}^0 \subseteq \mathcal{A}_{a_0}^0$ by choosing any M_s sequences from $\mathcal{A}_{a_0}^0$; similarly construct $\mathcal{A}^1 \subseteq \mathcal{A}_{a_1}^1$ by choosing any M_s sequences from $\mathcal{A}_{a_1}^1$. The sets $\mathcal{A}^0$ and $\mathcal{A}^1$ are subsets of the VT codes $\mathcal{VT}_{a_0}(b)$ and $\mathcal{VT}_{a_1}(b)$, containing sequences starting with 00 and 11, respectively.

Finally, the overall code of length $n = kb$ is constructed by choosing a codeword for each segment from either $\mathcal{A}^0$ or $\mathcal{A}^1$. The codeword for the first segment is chosen from $\mathcal{A}^0$. The codeword for segment $i = 2, \ldots, k$ is chosen as follows: if the last code bit in segment $(i-1)$ equals 0, then the codeword for segment i is chosen from $\mathcal{A}^1$; otherwise it is chosen from $\mathcal{A}^0$.

3.4.2 Rate

The rate of the above codes can be bounded from below as

$$R \geq 1 - \frac{1}{b}\log_2(b+1) - \frac{2}{b}. \tag{3.4.3}$$

Indeed, there are 2^{b-2} binary sequences of length b whose first two bits equal 0. Each of these sequences belongs to exactly one of the sets $\mathcal{A}_0^0, \ldots, \mathcal{A}_b^0$. Therefore, the largest among these $(b+1)$ sets will contain at least $2^{b-2}/(b+1)$ sequences and thus,

$$|\mathcal{A}_{a_0}^0| \geq \frac{2^{b-2}}{b+1}. \tag{3.4.4}$$

A similar argument gives the same lower bound for $|\mathcal{A}^1_{a_1}|$, hence

$$M_s \geq \frac{2^{b-2}}{b+1}. \tag{3.4.5}$$

Taking logarithms gives (3.4.3).

From (3.4.3), we see that the rate penalty with respect to VT codes is at most $\frac{2}{b}$ due to the prefix of length 2. As an example, for $b = 16$ our code has 964 codewords, while the greedy algorithm described in [27], gives 740; this is reduced to 652 when the search is restricted to VT codes. More examples are reported in Table 3.1.

3.4.3 Decoding

Thanks to the segment-by-segment code construction, decoding will also proceed segment by segment. Decoding proceeds in the following simple steps.

In order to decode segment i, for $i = 1, \ldots, k$, assume that the first $i-1$ segments have been decoded correctly. Thus the decoder knows the correct starting position of segment i in Y; we denote it by $p_i + 1$.

By examining the last bit of segment $(i-1)$, the decoder learns the correct syndrome for the codeword in segment i, i.e., either a_0 or a_1; recall that segment 1 was drawn from $\mathcal{A}^0$. Without loss of generality, assume it is a_0; the decoding for a_1 is identical.

1. The decoder computes the VT syndrome

$$\hat{a} = \mathsf{syn}(Y(p_i + 1 : p_i + b)) \tag{3.4.6}$$

   and compares it to the correct syndrome (assumed to be a_0). There are two possibilities:

   (a) $\underline{\hat{a} = a_0}$: The decoder concludes that there is no deletion in segment i. This is because if there was a deletion in segment i, then $Y(p_i + 1 : p_i + b)$ cannot have VT syndrome a_0 unless $Y(p_i + 1 : p_i + b) = S_i$ — indeed, if $Y(p_i + 1 : p_i + b) \neq S_i$, then both these length b sequences would have syndrome a_0 and $Y(p_i + 1 : p_i + b - 1)$ as a subsequence, contradicting the property of VT codes in (2.2.4).
   In this case, the decoder outputs $\hat{S}_i = Y(p_i + 1 : p_i + b)$. The starting position of the next segment in Y is $p_i + b + 1$.

   (b) $\underline{\hat{a} \neq a_0}$: The decoder knows that there is a deletion in segment i and feeds $Y(p_i + 1 : p_i + b - 1)$ to the VT decoder to recover the codeword. The output

of the VT decoder is the decoded segment $\hat{S}_i$. The starting position of the next segment in Y is $p_i + b$.

2. The decoder now checks the last bit of the decoded segment $\hat{s}_{i,b}$. If $\hat{s}_{i,b} = 0$, the decoder knows that segment $(i+1)$ has been drawn from $\mathcal{A}^1$; otherwise it has been drawn from $\mathcal{A}^0$. Thus the decoder is now ready to decode segment $(i+1)$.

3.4.4 Non-binary code construction

We now construct segmented deletion correcting codes for alphabet size $q > 2$. For $a = 0, \ldots, b-1$, and $c = 0, \ldots, q-1$, define following sets:

$$\mathcal{A}^j_{a,c} \triangleq \{S \in \mathcal{X}^b : \mathsf{syn}(A_S) = a, \mathsf{sum}(S) = c, s_1, s_2 \in \mathcal{X} \setminus \{j\}\}, \tag{3.4.7}$$

for $j = 0, \ldots, q-1$. Now for each $j = 0, \ldots, q-1$ define

$$\{a_j, c_j\} = \underset{\substack{0 \le a \le b-1 \\ 0 \le c \le q-1}}{\arg\max} |\mathcal{A}^j_{a,c}|. \tag{3.4.8}$$

Similarly to the binary case, the sets $\mathcal{A}^j_{a_j,c_j}$ for $0 \le j \le q-1$ are used to construct the codebook. Choose the first segment from $A^0_{a_0,c_0}$. For encoding ith segment $(i > 1)$ we choose a word from $\mathcal{A}^j_{a_j,c_j}$ if j is the last symbol of segment $i-1$. The size each set $\mathcal{A}^j_{a_j,c_j}$, for $0 \le j \le q-1$, can be bounded from below as

$$M_s \ge \frac{q^{b-2}(q-1)^2}{qb}. \tag{3.4.9}$$

Indeed, for any $j \in \{0, (q-1)\}$, there are $q^{b-2}(q-1)^2$ sequences of length b with the first two symbols are not equal to j. Each of these symbols belong to one of the sets $\mathcal{A}^j_{a,c}$, where $0 \le a \le b-1$, and $0 \le c \le q-1$. Therefore the largest set has size at least $\frac{q^{b-2}(q-1)^2}{qb}$. This gives a lower bound on the rate

$$R \ge \log_2 q - \frac{1}{b}\log_2 b - \frac{1}{b}\log_2 q - \frac{2}{b}\log_2\left(\frac{q}{q-1}\right). \tag{3.4.10}$$

Decoding proceeds in a similar way to the binary case. The main difference is that instead of computing (3.4.6), the decoder computes

$$\hat{a} = \mathsf{syn}(A_Z), \quad \hat{c} = \mathsf{sum}(Z) \tag{3.4.11}$$

where

$$Z = Y(p_i + 1 : p_i + b). \tag{3.4.12}$$

Then, the conditions in cases 1) a) and 1) b) are replaced by $\{\hat{a} = a_0 \text{ and } \hat{c} = c_0\}$ and by $\{\hat{a} \neq a_0 \text{ or } \hat{c} \neq c_0\}$, respectively.

3.5 Segmented insertion correcting codes

3.5.1 Binary code construction

As in the deletion case, we define a subset of VT codewords such that upon decoding a segment, there is no ambiguity in the starting position of the next segment. We define the following set of sequences

$$\mathcal{A}_a \triangleq \{S \in \{0,1\}^b : \mathsf{syn}(S) = a,\, s_1 s_2 = 01,\, s_3 s_4 \neq 01,\, S \neq 011\cdots 1\} \tag{3.5.1}$$

and

$$a_0 = \arg\max_{0 \leq a \leq b} |\mathcal{A}_a|. \tag{3.5.2}$$

Similarly to the previous section, the sets $\mathcal{A}_a \subseteq \mathcal{VT}_a(b)$ are sets of VT codewords with a prefix of a certain form. Our code is thus the maximal code in this family, i.e., $\mathcal{C} = \mathcal{A}_{a_0}^k$. In contrast to the deletion case, the codeword for each segment is drawn from the same set $\mathcal{A}_{a_0}$.

In order to find the size of the code, we use similar arguments to those in the previous section. There are 2^{b-2} sequences with prefix 01, out of which 2^{b-4} are removed because they have prefix 0101; $01\cdots 1$ is excluded from $\mathcal{A}_a$ by construction. Each of the $2^{b-2} - 2^{b-4} - 1$ sequences belong to exactly one of the sets $\mathcal{A}_0, \ldots, \mathcal{A}_b$. Therefore, the largest of these $b+1$ sets will have size at least

$$|\mathcal{A}_{a_0}| \geq \frac{2^{b-2} - 2^{b-4} - 1}{b+1}. \tag{3.5.3}$$

This yields the following lower bound for the rate for $b \geq 6$:

$$R \geq 1 - \frac{1}{b}\log_2(b+1) - \frac{2.5}{b}. \tag{3.5.4}$$

Hence the rate penalty is at most $\frac{2.5}{b}$ due to the added constraints on the prefix.

3.5.2 Decoding

Decoding proceeds on a segment-by-segment basis, and as in the case of deletions, the code structure ensures that before decoding segment i, the previous $(i-1)$ segments have been correctly decoded. Thus the decoder knows the correct starting position of segment i in Y; as before, denote it by p_i+1.

1. The decoder computes the VT syndrome

$$\hat{a} = \mathsf{syn}(Y(p_i+1:p_i+b)) \tag{3.5.5}$$

   and compares it to the correct syndrome a_0. There are two possibilities:

   (a) $\underline{\hat{a} \neq a_0}$: The decoder knows that there has been an insertion in this segment and feeds $Y(p_i+1:p_i+b+1)$ to the VT decoder to recover the codeword. The output of the VT decoder is the decoded segment $\hat{S}_i$. The decoder proceeds decoding segment $i+1$, skipping step 2. The starting position in Y for decoding segment $i+1$ is p_i+b+2.

   (b) $\underline{\hat{a} = a_0}$: The decoder concludes that there is no insertion in $Y(p_i+1:p_i+b)$. This is because if there was an insertion in segment i, then $Y(p_i+1:p_i+b)$ cannot have VT syndrome a_0 unless $Y(p_i+1:p_i+b) = S_i$ — indeed, if $Y(p_i+1:p_i+b) \neq S_i$, then both these length b sequences would have syndrome a_0 and $Y(p_i+1:p_i+b+1)$ as a supersequence, which contradicts the property of VT codes in (2.2.4).

   In this case, the decoder outputs $\hat{S}_i = Y(p_i+1:p_i+b)$.

2. If case 1.b) holds, the decoder has to check whether y_{p_i+b+1} could be an inserted bit at the very end of the segment. To this end, the $Y(p_i+b+1:p_i+b+4)$ is checked against the prefix conditions for segment $i+1$ set in $\mathcal{A}_{a_0}$.

   (a) If $\underline{y_{p_i+b+1}y_{p_i+b+2} \neq 01}$: the decoder understands that there is an irregularity caused by either an insertion in y_{p_i+b+1}, or in y_{p_i+b+2} or both. Therefore it deletes y_{p_i+b+1} and proceeds to decode segment $i+1$ starting from y_{p_i+b+2}.

   (b) If $\underline{y_{p_i+b+1}y_{p_i+b+2} = 01,\ y_{p_i+b+3}y_{p_i+b+4} \neq 01}$, then y_{p_i+b+1} is the correct start of segment $i+1$.

   (c) If $\underline{y_{p_i+b+1}y_{p_i+b+2} = 01,\ y_{p_i+b+3}y_{p_i+b+4} = 01}$: In this case, the decoder needs to decide among three alternatives by decoding segment $i+1$:

i. $y_{p_i+b+3} = 0$ is an inserted bit in segment $i+1$ and no inserted bit in segment i; let $\tilde{Y}_1 = y_{p_i+b+1}y_{p_i+b+2}y_{p_i+b+4}\cdots y_{p_i+2b+1}$ denote the length b sequence resulting from deleting y_{p_i+b+3} from the received sequence. If $\mathsf{syn}(\tilde{Y}_1) = a_0$ then $\hat{S}_{i+1} = \tilde{Y}_1$.

ii. $y_{p_i+b+4} = 1$ is an inserted bit in segment $i+1$ and no inserted bit in segment i; let $\tilde{Y}_2 = y_{p_i+b+1}y_{p_i+b+2}y_{p_i+b+3}y_{p_i+b+5}\cdots y_{p_i+2b+1}$ denote the length b sequence resulting from deleting y_{p_i+b+4} from the received sequence. If $\mathsf{syn}(\tilde{Y}_2) = a_0$ then $\hat{S}_{i+1} = \tilde{Y}_2$.

iii. $y_{p_i+b+1} = 0$, $y_{p_i+b+2} = 1$ are inserted bits in segments i and $i+1$, respectively; let $\tilde{Y}_3 = y_{p_i+b+3}y_{p_i+b+4}\cdots y_{p_i+2b+2}$ denote the length b sequence resulting from deleting y_{p_i+b+1}, y_{p_i+b+2} from the received sequence. If $\mathsf{syn}(\tilde{Y}_3) = a_0$ then $\hat{S}_{i+1} = \tilde{Y}_3$.

When $Y(bi+1:bi+4) = 0101$, we now show that the three cases listed in step 2.c) are mutually exclusive, and hence only one of them will give a matching VT syndrome. What needs to be checked is that the syndromes of $\tilde{Y}_1, \tilde{Y}_2, \tilde{Y}_3$ will all be different. From the very properties of VT codes we know that $\mathsf{syn}(\tilde{Y}_1) \neq \mathsf{syn}(\tilde{Y}_2)$. Now find that

$$\mathsf{syn}(\tilde{Y}_1) - \mathsf{syn}(\tilde{Y}_3) \pmod{b+1} \tag{3.5.6}$$

$$= \sum_{j=1}^{b} j\,\tilde{y}_{1,j} - \sum_{j=1}^{b} j\,\tilde{y}_{3,j} \pmod{b+1} \tag{3.5.7}$$

$$= 5 + \sum_{j=p_i+b+5}^{p_i+2b+1} y_j - 2 - by_{p_i+2b+2} \pmod{b+1} \tag{3.5.8}$$

$$= 3 + w_H(Y(p_i+b+5:p_i+2b+1)) + y_{p_i+2b+2} \pmod{b+1} \tag{3.5.9}$$

$$\neq 0 \tag{3.5.10}$$

where $w_H(Z)$ denotes the Hamming weight of sequence Z. The last step of (3.5.10) holds because

$$3 + w_H(Y(p_i+b+5:p_i+2b+1)) + y_{p_i+2b+2} \pmod{b+1} \tag{3.5.11}$$

can equal to 0 only if $w_H(Y(p_i+b+5:p_i+2b+1)) = b-3$ and $y_{p_i+2b+2} = 1$, implying that both $\tilde{Y}_1 = \tilde{Y}_3 = 011\cdots 1$. Since this sequence has been explicitly excluded from the codebook, we always have strict inequality, and hence $\mathsf{syn}(\tilde{Y}_1) \neq \mathsf{syn}(\tilde{Y}_3)$. Furthermore,

Table 3.2: State of y_{p_i+b+1} when $\hat{a} = a_0$ and $\hat{c} = c_0$.

$Y(p_i+b+1:p_i+b+3)$	State of $y_{p_i+b+1} = 0$	Starting point of next segment
001	No action (y_{p_i+b+1} is not an insertion)	p_i+b+1
000	Delete the first zero (y_{p_i+b+1} is an insertion)	p_i+b+2
010	Delete the 1 (y_{p_i+b+1} may or may not be inserted)	p_i+b+1

since

$$\mathsf{syn}(\tilde{Y}_2) - \mathsf{syn}(\tilde{Y}_3) = \mathsf{syn}(\tilde{Y}_1) - \mathsf{syn}(\tilde{Y}_3) - 1 \tag{3.5.12}$$

is always non-zero, we conclude that there is no ambiguity at the decoder .

3.5.3 The Liu-Mitzenmacher conditions for binary segmented codes

In [27], Liu and Mitzenmacher specified three conditions such that any set of binary sequences satisfying these conditions is a zero error code for both the segmented insertion channel and the segmented deletion channel. We list these conditions in Section 3.6.4, and show that the segmented insertion correcting code $\mathcal{A}_{a_0}$ described in Section 3.5.1 satisfies these conditions. This shows that the segmented insertion correcting code can also be used for the segmented deletion channel, with the decoder proposed in [27]. The deletion correcting code described in Section 3.4 has a slightly higher rate than the insertion correcting code in in Section 3.5.1. Moreover, the construction for the deletion case is more direct and can be generalized to non-binary alphabets and the segmented insertion-deletion channel.

However, the binary deletion correcting code proposed in Section 3.4.1 (or more precisely, the combined set of codewords $\mathcal{A}_a^0 \cup \mathcal{A}_a^1$) cannot be guaranteed to satisfy the Liu-Mitzenmacher conditions. Therefore, the construction in Section 3.4.1 may not be a zero error code for the segmented insertion channel, even with an optimal decoder.

It was conjectured in [27] that the rate and size of the maximal code satisfying the three sufficient conditions grows with b. As our insertion correcting code $\mathcal{A}_{a_0}$ satisfies the sufficient conditions, the lower bounds on its rate and size given in (3.5.3) and (3.5.4) confirm this conjecture.

3.5.4 Non-binary code construction

For the segmented insertion channel with alphabet size $q > 2$, we use prefix VT codes similar to those for the binary case. In this case, however, we set a prefix of length

3. This incurs a small penalty in rate with respect to the binary code described in Section 3.5.1, but results in a slightly simpler decoder. Define the following sets for all $a = 0, \ldots, b-1$ and $c = 0, \ldots, q-1$:

$$\mathcal{A}_{a,c} \triangleq \{S \in \mathcal{X}^b : \mathsf{syn}(A_S) = a, \mathsf{sum}(S) = c, s_1 s_2 s_3 = 001\}. \tag{3.5.13}$$

Now choose the largest set as the codebook, i.e., $\mathcal{C} = \mathcal{A}_{a_0,c_0}$ where

$$\{a_0, c_0\} = \underset{\substack{0 \le a \le b-1 \\ 0 \le c \le q-1}}{\arg\max} |\mathcal{A}_{a,c}|. \tag{3.5.14}$$

Similar to the binary case, the number of codewords can be bounded from below as

$$M_s \ge \frac{q^{b-3}}{qb}, \tag{3.5.15}$$

which gives the following lower bound on the rate:

$$R \ge \log_2 q - \frac{1}{b}\log_2 b - \frac{4}{b}\log_2 q. \tag{3.5.16}$$

Decoding proceeds in a similar manner to the binary case. As the code is somewhat different from the binary one, we give a few more details about the decoder. Assume that the first $(i-1)$ segments have been decoded correctly, and let $p_i + 1$ is the starting point of the ith segment. Let

$$Z = Y(p_i + 1 : p_i + b), \tag{3.5.17}$$

and compute

$$\hat{a} = \mathsf{syn}(A_Z), \quad \hat{c} = \mathsf{sum}(Z). \tag{3.5.18}$$

1. $\underline{\hat{a} \ne a_0 \text{ or } \hat{c} \ne c_0}$: The decoder knows that there has been an insertion in the ith segment and feeds $Y(p_i + 1 : p_i + b + 1)$ to the non-binary VT decoder to recover the codeword. The output of the VT decoder is the decoded segment $\hat{S}_i$. The starting position of the next segment in Y is $p_i + b + 2$.

2. $\underline{\hat{a} = a_0 \text{ and } \hat{c} = c_0}$: The decoder concludes that there is no insertion in segment i and outputs $\hat{S}_i = Y(p_i + 1 : p_i + b)$. The decoder must then investigate the possibility of an insertion at the very end of the ith segment in order to find the correct starting point of the next segment. This is done as follows. First, if the symbol y_{p_i+b+1} is not equal to 0, it is an insertion. The decoder deletes the

inserted symbol, and the starting position for the next segment is (p_i+b+2). Next, if $y_{p_i+b+1}=0$ and there is any symbol different from 0 or 1 in position (p_i+b+2) or (p_i+b+3), it is an inserted symbol thanks to the binary prefix. The decoder deletes the inserted symbol and sets the starting position of the next segment to (p_i+b+1). If neither of these cases hold, the decoder follows Table 3.2.

3.6 Segmented insertion-deletion correcting codes

3.6.1 Binary code construction

Since we now have both insertion and deletions, the decoder must first identify the type of edit in a segment prior to correcting it. Define the following sets:

$$\begin{aligned}\mathcal{A}_a^0 \triangleq \{S\in\{0,1\}^b : \mathsf{syn}(S)=a,\, s_1s_2s_3s_4s_5=00111,\\ s_{b-2}=s_{b-1}=s_b\}\end{aligned} \tag{3.6.1}$$

$$\begin{aligned}\mathcal{A}_a^1 \triangleq \{S\in\{0,1\}^b : \mathsf{syn}(S)=a,\, s_1s_2s_3s_4s_5=11000,\\ s_{b-2}=s_{b-1}=s_b\}.\end{aligned} \tag{3.6.2}$$

As in previous sections, these are subsets of VT codewords with certain constraints. In this case, in order to be able to identify the edit type, both prefix and suffix constraints have been added. Based on the above sets, we further define

$$a_0=\arg\max_{0\le a\le b}|\mathcal{A}_a^0|, \quad a_1=\arg\max_{0\le a\le b}|\mathcal{A}_a^1| \tag{3.6.3}$$

and $M_s=\min\{|\mathcal{A}_{a_0}^0|,|\mathcal{A}_{a_1}^1|\}$. We construct the sets $\mathcal{A}^0,\mathcal{A}^1$ by choosing M_s sequences from $\mathcal{A}_{a_0}^0,\mathcal{A}_{a_1}^1$, respectively. Finally, the overall code of length $n=kb$ is constructed by choosing a codeword for each segment from either $\mathcal{A}^0$ or $\mathcal{A}^1$. The codeword for the first segment is chosen from $\mathcal{A}^0$. For $i\in\{2,\dots,k\}$, if the last bit of segment $(i-1)$ is 0, then the codeword for segment i is drawn from $\mathcal{A}^1$ and otherwise from $\mathcal{A}^0$.

The size and rate are lower-bounded using the same arguments as in the previous sections. For $b\ge 7$, we obtain

$$M_s \ge \frac{2^{b-7}}{b+1} \tag{3.6.4}$$

which yields a rate lower bound given by

$$R \geq 1 - \frac{1}{b}\log(b+1) - \frac{7}{b}. \tag{3.6.5}$$

Due to the prefix and suffix constraints, our segmented insertion-deletion correcting codes have a rate penalty of at most $\frac{7}{b}$.

3.6.2 Decoding

As in the previous two cases, decoding proceeds segment-by segment. We ensure that before decoding segment i, the previous $(i-1)$ segments have all been correctly decoded. Hence, the decoder knows the correct starting position in Y for segment i, which is denoted by p_i+1. The decoder also knows whether S_i belongs to $\mathcal{A}^0$ or to $\mathcal{A}^1$. We discuss the case where $S_i \in \mathcal{A}^0$, so $\mathsf{syn}(S_i) = a_0$; the case where $S_i \in \mathcal{A}^1$ is similar, with the roles of the bits reversed.

The decoder computes the syndrome $\mathsf{syn}(Y(p_i+1:p_i+b))$, and checks whether it equals a_0. There are two possibilities:

1. $\mathsf{syn}(Y(p_i+1:p_i+b)) \neq a_0$: This means that there is an edit in this segment, we should identify the type of edit and correct it. We show that can be done without ambiguity by using the fact that three last bits of each segment (suffix) are the same, and considering prefix of the next segment. The decoder's decision for each combination of the three consecutive bits $(y_{p_i+b-1}, y_{p_i+b}, y_{p_i+b+1})$ is listed in Table 3.3. Once the type of edit is known, the decoder corrects the segment using the appropriate VT decoder. We now justify the decisions listed in Table 3.3.

   (a) If $\underline{y_{p_i+b-1} = y_{p_i+b} = y_{p_i+b+1}}$: The edit is an insertion. To see this, assume by contradiction that it was a deletion. Then at least one of y_{p_i+b} and y_{p_i+b+1} are the first bit of the prefix of S_{i+1}, and y_{p_i+b-1} is a suffix bit of S_i. This is not possible because by construction, the first two prefix bits of S_{i+1} must be different from the suffix bits of S_i.

   (b) If $\underline{y_{p_i+b-1} = y_{p_i+b} \neq y_{p_i+b+1}}$: The edit is a deletion. To see this, suppose that the edit was an insertion; then the suffix condition can only be satisfied if y_{p_i+b+1} is the inserted bit. However, this implies that $\mathsf{syn}(Y(p_i+1:p_i+b)) = a_0$, which is contradiction.

   (c) If $\underline{y_{p_i+b-1} = y_{p_i+b+1} \neq y_{p_i+b}}$: The edit could be either an insertion, or a deletion, according to the rules in lines 3, 4, 5 of Table 3.3. If the the edit is

Table 3.3: Type of edit when $\mathsf{syn}(Y(p_i+1:p_i+b)) \neq a_0$

State of sequence	Type of edit
$y_{p_i+b-1} = y_{p_i+b} = y_{p_i+b+1}$	Insertion
$y_{p_i+b-1} = y_{p_i+b} \neq y_{p_i+b+1}$	Deletion
$y_{p_i+b-1} = y_{p_i+b+1} \neq y_{p_i+b}$ and $\mathsf{syn}(Z) \neq a_0$, where $Z = [Y(p_i+1:p_i+b-1), y_{p_i+b+1}]$	Deletion
$y_{p_i+b-1} = y_{p_i+b+1} \neq y_{p_i+b}$ and $\mathsf{syn}(Z) = a_0$ and $y_{p_i+b+1} = y_{p_i+b+2} = y_{p_i+b+3}$	Deletion
$y_{p_i+b-1} = y_{p_i+b+1} \neq y_{p_i+b}$ and $\mathsf{syn}(Z) = a_0$ and $(y_{p_i+b+1} \neq y_{p_i+b+2}$ or $y_{p_i+b+1} \neq y_{p_i+b+3})$	Insertion
$y_{p_i+b-1} \neq y_{p_i+b} = y_{p_i+b+1}$ and $y_{p_i+b-2} = y_{p_i+b-1}$	Deletion
$y_{p_i+b-1} \neq y_{p_i+b} = y_{p_i+b+1}$ and $y_{p_i+b-2} \neq y_{p_i+b-1}$	Insertion

an insertion, then y_{p_i+b} is the inserted bit, therefore by omitting this bit, the sequence $Z = [Y(p_i+1:p_i+b-1), y_{p_i+b+1}]$ should have VT-syndrome equal to a_0. Therefore, if $\mathsf{syn}(Z) \neq a_0$, then the edit is deletion; if $\mathsf{syn}(Z) = a_0$, we need to check the prefix of the next segment to determine the type of edit.

If $\underline{\mathsf{syn}(Z) = a_0}$: If $y_{p_i+b+1} = y_{p_i+b+2} = y_{p_i+b+3}$, then the edit in segment i is a deletion (it can be verified that the prefix condition for segment $(i+1)$ cannot otherwise be satisfied with at most one edit),. In all other cases the edit in segment i is insertion, with y_{p_i+b} being the inserted bit. We observe that when $\mathsf{syn}(Z) = a_0$, $S_i = Z$ with either type of edit, but the decoder needs to infer the type of edit in order to guarantee the correct starting position for the next segment.

(d) If $\underline{y_{p_i+b-1} \neq y_{p_i+b} = y_{p_i+b+1}}$: In this case, y_{p_i+b-2} determines the type of edit: if $y_{p_i+b-2} = y_{p_i+b-1}$ the edit is a deletion, otherwise it is an insertion. This can be seen by examining the suffix condition: if the edit is an insertion then y_{p_i+b-1} is the inserted bit therefore y_{p_i+b-2} belongs to suffix of S_i, hence $y_{p_i+b-2} = y_{p_i+b} = y_{p_i+b+1}$. On the other hand, if the edit is a deletion, then y_{p_i+b-2} and y_{p_i+b-1} belongs to suffix of S_i, so they should be equal.

Hence we have shown that whenever $\mathsf{syn}(Y(p_i+1:p_i+b)) \neq a_0$, we can uniquely decode S_i and determine the correcting starting position for the next segment.

2. $\mathsf{syn}(Y(p_i+1:p_i+b)) = a_0$: In this case, by combining the arguments in step 1.a) of the deletion decoder and step 1.b) of the insertion encoder, we conclude that $\hat{S}_i = Y(p_i+1:p_i+b)$. To determine the correct starting position for the next segment, we have to investigate the possibility of an insertion at the end of the block, i.e., determine whether y_{p_i+b+1} is an inserted bit. This can be done by examining the prefix of S_{i+1}. We consider five bits, $Y(p_i+b+1:p_i+b+5)$, and for all 32 cases determine the state of y_{p_i+b+1}. For the simplicity, assume

Table 3.4: State of y_{p_i+b+1} when $\mathsf{syn}(Y(p_i+1:p_i+b)) = a_0$.

$Y(p_i+b+1:p_i+b+5)$	State of y_{p_i+b+1}
$1uvst$	Inserted
$000uv$	Inserted
$011uv$	Not Inserted
01000	Not possible
01001	Inserted
01010	Not possible
01011	Not inserted
00100	Not possible
00101 and $\mathsf{syn}(Z_1)$ matches	Inserted
00101 and $\mathsf{syn}(Z_2)$ matches	Not Inserted
00110	Not Inserted
00111	Not Inserted

that the last bit of S_i is 1, so that the prefix for S_{i+1} is 00111; the other case is identical, with 0 and 1 interchanged.

First, if $y_{p_i+b+1} = 1$, then it is an inserted bit (this is 16 of the 32 cases). Table 3.4 lists the type of edit for each of the other cases when $y_{p_i+b+1} = 0$. These are justified below.

(a) If $\underline{Y(p_i+b+1:p_i+b+5) = 011uv}$ for some bits u, v, then y_{p_i+b+1} is not an insertion corresponding to segment i: if it was inserted, then decoding for segment $(i+1)$ would start with the bits $11\ldots$, which cannot be matched with the prefix 00111 with only one edit. Hence the correct starting position for decoding segment $(i+1)$ is p_i+b+1.

(b) If $\underline{Y(p_i+b+1:p_i+b+5) = 000uv}$, then y_{p_i+b+1} (or another 0 from the run) is an insertion for segment i, as $000u$ does not match 0011 unless we remove a zero form the run.

(c) The cases $\underline{Y(p_i+b+1:p_i+b+5) = 01000{,}01010}$, $\underline{00100}$ cannot occur as they cannot be matched with the required prefix 00111 through any valid edits for segment $i+1$, whether or not y_{p_i+b+1} is inserted.

(d) If $\underline{Y(p_i+b+1:p_i+b+5) = 01001}$, then y_{p_i+b+1} is an insertion for segment i as this is the only option consistent with the prefix 00111.

(e) If $\underline{Y(p_i+b+1:p_i+b+5) = 0011u \text{ or } 01011}$, then $y_{p_i+b+1} = 0$ is not an insertion for segment i, and is the starting bit for decoding segment $(i+1)$.

(f) If $\underline{Y(p_i+b+1:p_i+b+5)=00101}$, we need to compare the VT syndromes of two sequences to determine the status of y_{p_i+b+1}. We will also decode S_{i+1} in the process. If $y_{p_i+b+1}=0$ is inserted, then $y_{p_i+b+3}=1$ should also be inserted, therefore $S_{i+1}=Z_1$ where $Z_1=[00,Y(p_i+b+5:p_i+2b+2)]$. On the other hand, if y_{p_i+b+1} is not inserted then $y_{p_i+b+4}=0$ should be an inserted bit, therefore, $S_{i+1}=Z_2$ where $Z_2=[001,Y(p_i+b+5:p_i+2b+1)]$. However, Z_1 and Z_2 will always produce different syndromes and only one of them will be equal to a_0, the correct syndrome for segment $(i+1)$. Thus we can correctly identify whether y_{p_i+b+1} was an insertion for segment i or not.

Hence we have shown that whenever $\mathsf{syn}(Y(p_i+1:p_i+b))=a_0$, we can uniquely decode S_i and determine the correcting starting position for the next segment.

The decoding algorithm described above was simulated in Matlab to confirm that the code is indeed zero error. The Matlab files for implementing the codes proposed for all three binary segmented edit models are available at [105].

3.6.3 Non-binary code construction

We now construct segmented insertion-deletion correcting codes for alphabet size $q>2$. For $a=0,\ldots,b-1$, and $c=0,\ldots,q-1$, define following sets:

$$\begin{aligned}\mathcal{A}^0_{a,c} \triangleq \{&S\in\mathcal{X}^b:\mathsf{syn}(A_S)=a,\mathsf{sum}(S)=c,\\ &s_1s_2s_3s_4s_5=00111,\, s_{b-2}=s_{b-1}=s_b\},\end{aligned} \tag{3.6.6}$$

$$\begin{aligned}\mathcal{A}^1_{a,c} \triangleq \{&S\in\mathcal{X}^b:\mathsf{syn}(A_S)=a,\mathsf{sum}(S)=c,\\ &s_1s_2s_3s_4s_5=11000,\, s_{b-2}=s_{b-1}=s_b\}.\end{aligned} \tag{3.6.7}$$

For $j=0,1$ define

$$\{a_j,c_j\}=\underset{\substack{0\le a\le b-1\\0\le c\le q-1}}{\arg\max}|\mathcal{A}^j_{a,c}|. \tag{3.6.8}$$

We use the sets $\mathcal{A}^0_{a_0,c_0}$ and $\mathcal{A}^1_{a_1,c_1}$ to construct the codebook by alternating depending on the last symbol of the previous segment. We set $M_s=\min\{\mathcal{A}^0_{a_0,c_0},\mathcal{A}^1_{a_1,c_1}\}$ and construct the sets $\mathcal{A}^0,\mathcal{A}^1$ by choosing M_s sequences from $\mathcal{A}^0_{a_0,c_0},\mathcal{A}^1_{a_1,c_1}$, respectively. The codeword for the first segment is chosen from $\mathcal{A}^0$. For $i\in\{2,\ldots,k\}$, if the last symbol of segment $(i-1)$ is an *even* number, then the codeword for segment i is drawn

from $\mathcal{A}^1$; if the last symbol of segment $(i-1)$ is an *odd* number, the codeword is drawn from $\mathcal{A}^0$.

The number of codewords per segment satisfies

$$M_s \geq \frac{q^{b-7}}{qb} \tag{3.6.9}$$

and thus a lower bound on the rate is

$$R \geq \log_2 q - \frac{1}{b}\log_2 b - \frac{8}{b}\log_2 q. \tag{3.6.10}$$

The decoding is almost identical to the binary case. As with previous decoders, to decode segment i, it is assumed that the first $(i-1)$ segments have been decoded correctly. Let $Z = Y(p_i+1 : p_i+b)$, where p_i+1 is the starting position of the ith segment. Compute

$$\hat{a} = \mathsf{syn}(A_Z), \quad \hat{c} = \mathsf{sum}(Z). \tag{3.6.11}$$

The decoder checks whether $\{\hat{a} = a_0 \text{ and } \hat{c} = c_0\}$ or $\{\hat{a} \neq a_0 \text{ or } \hat{c} \neq c_0\}$. In the first case, the decoder sets $\hat{S}_i = Y(p_i+1 : p_i+b)$ and in order to find the starting point of segment $i+1$, follows the same case breakdown as in the binary decoder (see case 2 of the binary decoder). On the other hand, if $\{\hat{a} \neq a_0 \text{ or } \hat{c} \neq c_0\}$, thanks to the prefix-suffix code structure being the same as the binary one, the decoder follows exactly the same case breakdown (see case 1 of the binary decoder) in order to identify the type of edit and correct it.

3.6.4 The Liu-Mitzenmacher conditions

Let $\mathcal{I}_1(X)$ denote the set of all sequences obtained by adding one bit to the binary sequence X. Then $\mathcal{C} \subseteq \{0,1\}^b$ is a binary zero error code for both the segmented insertion channel and the segmented deletion channel (with segment length b) if the following conditions are satisfied.

1. For any $U, V \in \mathcal{C}$, with $U \neq V$, $\mathcal{I}_1(U) \cap \mathcal{I}_1(V) = \emptyset$;

2. For any $U, V \in \mathcal{C}$, with $U \neq V$, $\text{prefix}(\mathcal{I}_1(U)) \cap \text{suffix}(\mathcal{I}_1(V)) = \emptyset$;

3. Any string of the form $y^*(zy)^*$ or $y^*(zy)^*z$, where $y, z \in \{0,1\}$, is not in $\mathcal{C}$.

Here $\text{prefix}(X)$ denotes the subsequence of X obtained excluding the last bit, $\text{suffix}(X)$ the subsequence obtained excluding the first bit, and X^* is the regular expression

notation referring to 0 or more copies of sequence X. The set $\text{prefix}(\mathcal{I}_1(U))$ is defined as $\{\text{prefix}(X) : X \in \mathcal{I}_1(U)\}$. The set $\text{suffix}(\mathcal{I}_1(V))$ is defined similarly.

We now show that the insertion correcting code $\mathcal{A}_{a_0}$ defined in Section 3.5.1 satisfies these conditions. Since $\mathcal{A}_{a_0}$ is a subset of a VT code and is hence a single insertion correcting code, the first condition is satisfied.

We next verify the third condition. All the codewords in $\mathcal{A}_{a_0}$ start with 01. It is easy to see that any sequence starting with 01 and violating the third condition in either of the two ways must have 0101 as its first four bits. But these sequences are excluded from $\mathcal{A}_{a_0}$, so each codeword in $\mathcal{A}_{a_0}$ satisfies the third condition.

It remains to prove that the second condition is satisfied. Assume towards contradiction that there exist codewords $U, V \in \mathcal{A}_{a_0}$ such that $U \neq V$ and the set $\mathcal{W} = \text{prefix}(\mathcal{I}_1(U)) \cap \text{suffix}(\mathcal{I}_1(V))$ is non-empty. Suppose that the sequence $Z \in \mathcal{W}$, and $Z_1 \in \mathcal{I}_1(U)$ and $Z_2 \in \mathcal{I}_1(V)$ are length $(b+1)$ sequences such that that $Z = \text{prefix}(Z_1) = \text{suffix}(Z_2)$.

Since $U \in \mathcal{A}_{a_0}$ and $Z_1 \in \mathcal{I}_1(U)$, $\text{prefix}(Z_1)$ will start with a 0, unless the inserted bit in Z_1 is a 1 and is inserted exactly at the beginning of U, i.e., unless $Z_1 = [1, U]$. Also, since $Z_2 \in \mathcal{I}_1(V)$, $\text{suffix}(Z_2)$ will start with 1 unless Z_2 is obtained by adding a bit at the beginning of V, i.e. $Z_2 = [h, V]$, for $h \in \{0, 1\}$. Since $Z = \text{prefix}(Z_1) = \text{suffix}(Z_2)$, clearly one of the above two cases should hold. First, assume that Z starts with 1 and therefore we have $Z_1 = [1, U]$. Now since $U \in \mathcal{A}_{a_0}$ starts with 01, we have

$$Z = \text{prefix}(Z_1) \tag{3.6.12}$$
$$= Z_1(1:b) \tag{3.6.13}$$
$$= [1, U(1:b-1)] \tag{3.6.14}$$
$$= [101, U(3:b-1)]. \tag{3.6.15}$$

Now we also know that $Z = \text{suffix}(Z_2)$, so $\text{suffix}(Z_2) = [101, U(3:b-1)]$. Now, notice that $Z_2 \in \mathcal{I}_1(V)$ and first bit of V is 0, so the first two bits of Z_2 cannot be 11. We therefore have

$$Z_2 = [0101, U(3:b-1)]. \tag{3.6.16}$$

But we know that $V \in \mathcal{A}_{a_0}$ cannot start with 0101, so either the third or the fourth bit in Z_2 is the inserted bit. Therefore, we know that

$$V = [01z, U(3:b-1)], \tag{3.6.17}$$

for $z \in \{0,1\}$. We also know that

$$U = [01, U(3:b-1), u_b], \tag{3.6.18}$$

where $u_b \in \{0,1\}$. But this contradicts condition 1 (which has already been verified) because we obtain the same length $(b+1)$ sequence by: i) inserting u_b to the end of V, and ii) inserting z after the second bit of U.

Next consider the second case where Z starts with a 0. As explained above, we then have $Z_2 = [h, V]$, and hence, $Z = \text{suffix}(Z_2) = V$. Therefore $\text{prefix}(Z_1) = V$, so one can obtain Z_1 by adding the last bit of Z_1 to V. Therefore $Z_1 \in \mathcal{I}_1(U) \cap \mathcal{I}_1(V)$, which is a contradiction. This completes the proof that $\mathcal{A}_{a_0}$ satisfies all the three conditions.

Chapter 4

Multilayer codes

4.1 Introduction

Consider two remote nodes having binary sequences X and Y, respectively, where Y is an edited version of X. The goal in this chapter is to construct a message M such that the second node (decoder) can reconstruct X using M and Y. In the majority of this chapter, we consider the edits to be deletions. In Section 4.8, we briefly consider the case with both insertions and deletions. Let the length of X be n bits, and the number of deletions be k. Thus, Y is a sequence of length $m = (n-k)$, obtained by deleting k bits from X. As discussed in Chapter 1, in the synchronization model shown in Fig. 4.1, the node with X (the "encoder") sends a message M via an error-free link to the other node (the "decoder"), which attempts to reconstruct X using M and Y. The goal is to design a scheme so that the decoder can reconstruct X with minimal communication, i.e., we want to minimize the number of bits used to represent the message M.

The deletion model is a simplified version of the general file synchronization model where the edits can be a combination of deletions, insertions, and substitutions. The general synchronization model has a number of applications including file backup (e.g., Dropbox) and file sharing. Various forms of the synchronization model have been studied in previous works; see, e.g., [60–62, 64–67]. A number of these works allow for *two-way* interaction between the encoder and decoder. In contrast, we seek codes for *one-way* synchronization: the message M is produced by the encoder using only X, with no knowledge of Y except its length m. We assume that the decoder knows n, so it can infer the number of deletions $k = (n-m)$. The message M belongs to a finite set $\mathcal{M}$ with cardinality $|\mathcal{M}|$.

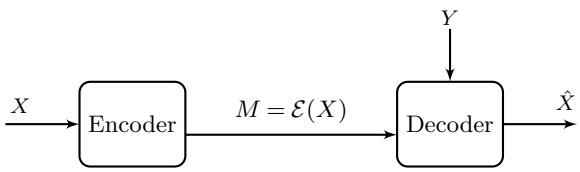

Figure 4.1: Synchronization Model

In this chapter, we construct a code based on multiple layers of VT codes for synchronization from deletions when the number of deletions k is small compared to n. The output of the decoder is a small list of sequences that is guaranteed to contain the correct sequence X. We observe from simulations that with a careful choice of the code parameters, the list size rarely exceeds 2 or 3; for reasonably large n, the list size can be made 1, i.e., X is exactly reconstructed. For example, we construct a code of length $n = 378$ that can synchronize from $k = 7$ deletions with $R = 0.365$, and a length $n = 2800$ code which can synchronize from $k = 10$ deletions with $R = 0.135$. (Details are provided in Section 4.4.) We later show how to modify the decoder (keeping the same encoding scheme) to reconstruct a combination of up to k insertions and deletions. Also in Section 4.9 we discuss how the proposed multilayer code can be used in a deletion channel setup.

4.1.1 Overview of the code construction

The starting point for our code construction is VT codes. The VT code gives an elegant way to exactly synchronize from a single deletion: the encoder simply sends the VT syndrome of the sequence X. The decoder then uses the single deletion correcting property of the VT code to recover the deleted bit.

Remark 4.1. *The VT syndrome is known to be optimal for recovering one deletion. This is because there are exactly $n+1$ distinct sequences that can be obtained by inserting one bit into a length $n-1$ sequence (see* (2.3.11)*). Therefore, the encoder in order to signal these sequences needs at least $n+1$ messages.*

In our model, the code needs to synchronize from $k > 1$ deletions. The encoder sends the VT syndromes of various substrings of X to the decoder. Specifically, the length n sequence X is divided into smaller chunks of n_c bits each. The encoder then computes VT syndromes for two kinds of substrings: *blocks* which are composed of adjacent chunks, and *chunk-strings* which are composed of well-separated chunks. Fig. 4.2 shows an example where X of length 12 is divided into 4 length-3 chunks. The blocks B_1 and B_2 are each formed by combining two adjacent chunks, while the

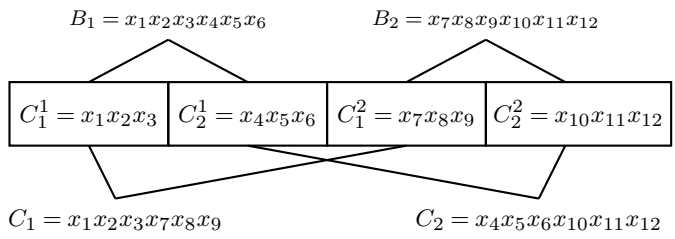

Figure 4.2: Blocks and chunk-strings structure for the example where $l_1 = l_2 = 2$

chunk-strings C_1 and C_2 are each formed by combining two alternate chunks. In this case, the encoder sends the VT syndromes of B_1, B_2, C_1, and C_2.

The intersecting VT constraints of blocks and chunk-strings help the decoder to estimate locations of the edits. The VT syndromes serve a dual purpose: i) they can be used to recover deleted bits in blocks or chunk-strings inferred to have a single deletion; this recovery may result in new blocks and chunk-strings with a single deletion; ii) the VT syndromes also act as checks that eliminate a large number of potential deletion patterns, allowing the decoder to localize the deletions to a relatively small set of chunks. The final element of the message is a parity check syndrome of X using a linear code. This is used to recover the deletions in chunks that still remain uncertain at the decoder after processing the intersecting VT constraints. We refer to this code construction as a two-layer code as the chunks are combined to form two kinds of intersecting substrings. The construction can be generalized to combine chunks in multiple ways to form many layers of intersecting substrings (see Section 4.10.2). Increasing the number of constraints in the code improves its synchronization capability at the cost of increasing the rate.

For decoding, we use a list decoder. The output of the decoder is the list of all length n sequences that can be obtained by inserting k bits into sequence Y, and satisfy VT constraints and the parity check constraints that are imposed via message M. The correct sequence X is always in the list. List decoding has also been recently used in [106] for the case of multiple deletions. Specifically, a lower bound is found for the maximum list size when the code has a single VT constraint, and it is shown that the list size can grow exponentially with the length of the code.

4.1.2 Comparison between erasure and deletion errors

Using multiple VT constraints for recovering multiple deletions seems a natural approach if one compares the two cases of deletion and erasure errors. Assume that the sequence Y (at the decoder) suffers from only a single erasure, and encoder wants to send the

message M such that the decoder recovers the erasure. In this case, as the position of the error is known it suffices to send the binary summation of bits in X to the decoder. Now consider the case when Y suffers from multiple erasures; linear codes are the simplest codes that can be used in this setup (e.g., LDPC codes). That means we will send the binary summation of various subsequences of X to the decoder. Now in comparison with the deletion error, when there is only one deletion in Y, VT syndrome of X suffices for recovering the single deletion. In this work we are considering using multiple VT constraints of various subsequences of X for the case when Y suffers from multiple deletions.

4.1.3 Structure of the chapter

The remainder of this chapter is structured as follows. In Section 4.2, we explain the encoding scheme which can be used for synchronizing from deletion or insertions. Then in Section 4.3, we describe the decoding algorithm for synchronization from deletions. In Section 4.4, we provide the simulation results for the proposed encoding and decoding algorithm. In Section 4.5, we analyze the complexity of the encoding and decoding schemes. In Section 4.6 we explain how the decoding algorithm can be modified in order to work without the linear parity check constraints in message M. In Section 4.7, we introduce a new simple method for recovering deletions. Using this method we can provide a new interpretation of the decoder which we use to give an upper bound for the list size. In Section 4.8, a modified decoder is described that can correct a combination of insertions and deletions. The next section studies two approaches to use multilayer code in a deletion channel setup. Section 4.10 discusses two extensions of the proposed code construction and a short summary of the work.

4.2 Code construction and encoding

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

The first step is to divide $X = x_1 x_2 \cdots x_n$ into l_1 equal-sized blocks (assume that n is divisible by l_1). The length of each block is denoted by $n_b = \frac{n}{l_1}$. For $1 \leq i \leq l_1$, the ith block is denoted by $B_i = X((i-1)n_b + 1 : in_b)$, and its VT syndrome is $s_{B_i} = \mathsf{syn}(B_i)$. The first part of the message is the collection of VT syndromes for the l_1 blocks, i.e.,

$M_1 = \{s_{B_1}, s_{B_2}, \cdots, s_{B_{l_1}}\}$. Since each s_{B_i} is an integer between 0 and n_b, the number of bits required to represent the VT syndromes of the l_1 blocks is $l_1\lceil\log(n_b+1)\rceil$.

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

$$C_j^i = X((i-1)n_b + (j-1)n_c + 1 : (i-1)n_b + jn_c). \tag{4.2.1}$$

The jth *chunk-string* is then formed by concatenating the jth chunk from each of the l_1 blocks. That is, the jth chunk string $C_j = [C_j^1, C_j^2, \cdots, C_j^{l_1}]$, for $1 \le j \le l_2$. Fig. 4.2 shows the blocks and the chunk-strings in an example where X of length $n = 12$ is divided into $l_1 = 2$ blocks, each of which is divided into $l_2 = 2$ chunks of $n_c = 3$ bits.

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

The final part of the message is the parity check syndrome of X with respect to a linear code. Consider a linear code of length n with parity check matrix $\boldsymbol{H} \in \{0,1\}^{z\times n}$. Then $M_3 = \boldsymbol{H}X$ is the third component of M. The coset of the linear code containing X will be used as an erasure correcting code. In our experiments in Section 4.4, the linear code is chosen to be either a Reed-Solomon code, or a random linear code defined by a random binary parity check matrix. The number of bits in M_3 is equal to the number of rows of $\boldsymbol{H}$, i.e., number of binary parity checks in the code, z. The overall number of bits required to represent the message $M = [M_1, M_2, M_3]$ is $l_1\lceil\log(n_b+1)\rceil + l_2\lceil\log(n_c l_1+1)\rceil + z$.

Since $n_b = n_c l_2$, normalizing by $n = n_c l_1 l_2$ gives the *synchronization rate* R_{sync} of our scheme

$$R_{\text{sync}} = \frac{z}{n} + \frac{\lceil\log(n_c l_2+1)\rceil}{n_c l_2} + \frac{\lceil\log(n_c l_1+1)\rceil}{n_c l_1}. \tag{4.2.2}$$

Example 4.1. *Suppose that we want to design a code for synchronizing a binary sequence of length $n = 60$ from $k = 4$ deletions. Choose the chunk length $n_c = 4$, so that there are 15 chunks in the string. Divide the string into $l_1 = 5$ blocks, each comprising $l_2 = 3$ chunks. Thus there are 5 blocks each consisting of 3 adjacent chunks, and 3 chunk-strings each consisting of 5 separated chunks.*

Noting that each chunk of $n_c = 4$ bits corresponds to a symbol in $GF(2^4)$, we use a Reed-Solomon code defined over $GF(2^4)$ with length $2^4 - 1 = 15$. We also choose the

parity check matrix to have 4 *parity check equations in* $GF(2^4)$*, so we can recover* 4 *erased chunks using this Reed-Solomon code.*

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

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

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

We similarly compute M_2*. The first chunk-string* $[4\ 0\ 7\ 0\ 4]$ *in binary is*

$$C_1 = 0100\ 0000\ 0111\ 0000\ 0100,$$

with VT syndrome $s_{C_1} = 11$*. Computing the VT syndromes of the other chunk-strings in a similar manner, we get* $M_2 = \{11, 20, 4\}$*.*

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

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

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

4.3 Decoding algorithm

The goal of the decoder is to recover X given Y, n and the message $M = [M_1, M_2, M_3]$. From M_1, M_2, the decoder knows the VT syndrome of each block and each chunk-string. Using this, the decoder first finds all possible configurations of deletions across blocks, and then for each of these configurations, it finds all possible chunk deletion patterns. Since each chunk is the intersection of a block and a chunk-string, each chunk plays a role in determining exactly two VT syndromes. The intersecting construction of blocks and chunk-strings enables the decoder to iteratively recover the deletions in a large

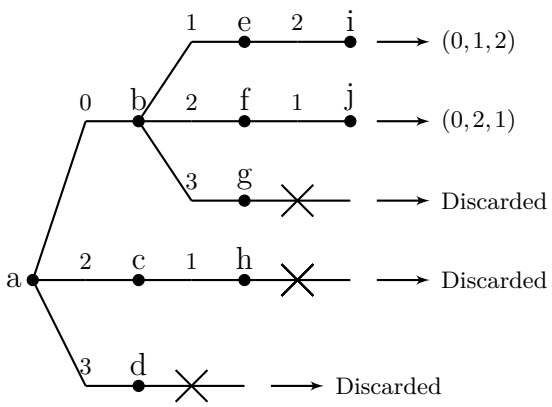

Figure 4.3: Tree representing the valid block vectors for Example 4.2.

number of cases. The decoder is then able to localize the positions of the remaining deletions to within a few chunks. These chunks are considered erased, and are finally recovered by the erasure-correcting code.

The decoding algorithm consists of six steps, as described below.

Step 1: Block boundaries

In the first step, the decoder produces a list of candidate *block-deletion patterns* $V = (a_1, \cdots, a_{l_1})$ compatible with Y, where a_i is the number of deletions in the ith block. Each pattern in the list should satisfy $\sum_{i=1}^{l_1} a_i = k$ with $0 \leq a_i \leq k$. The list of candidates always includes the true block-deletion pattern. It is convenient to represent the candidate block-deletion patterns as branches on a tree with l_1 levels, as shown in Fig. 4.3. At every level (block) $i = 1, \ldots, l_1$, branches are added and labeled with all possible values of a_i. Specifically, the tree is constructed as follows.

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

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

$\underline{01}00\ 1111\ \ldots$, with the underlined bits deleted we get $Y = 001111\ldots$. In this case $u = s_{B_1} = 2$. The decoder thus adds a branch for $a_1 = 0, 2, \ldots, k$.

2. $\underline{u \neq s_{B_1}}$: Block one contains one or more deletions and the decoder adds a branch for $a_1 = 1, 2, \ldots, k$.

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

$$p_{i+1} = n_b i - d_i + 1. \tag{4.3.1}$$

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

1. $\underline{u = s_{B_{i+1}}}$: If $(k - d_i) < 2$ then the only possibility is that $a_{i+1} = 0$. If $(k - d_i) \geq 2$, $k - d_i - 1$ branches are added for $a_{i+1} = 0, 2, \ldots, k - d_i$.

2. $\underline{u \neq s_{B_{i+1}}}$: If $(k - d_i) > 0$ then there are $(k - d_i)$ possibilities at this branch: the ith block can have $1, 2, \cdots, (k - d_i)$ deletions. If $(k - d_i) = 0$, it is assumed this is an invalid branch, and the path is discarded.

Example 4.2. *Assume $k = 3$ deletions, $l_1 = 3$ blocks, and that the true deletion pattern is (0,2,1), i.e., there are zero deletions in the first block, two deletions in second block, and one deletion in third block. The tree constructed by the decoder depends on the underlying sequences X and Y. In Fig. 4.3, we illustrate one possible tree constructed for this scenario without explicitly specifying X and Y.*

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

Step 2: Primary fixing of blocks

Denote by $\mathcal{L}_1$ the list of the block-deletion pattern candidates after the first step and denote the corresponding block-deletion patterns by $V_1, \cdots, V_{|\mathcal{L}_1|}$. In this second step, for each of the block-deletion patterns, we restore the deleted bit in blocks containing a single deletion by using the VT decoder. Specifically, for a block-deletion pattern $V = (a_1, \cdots, a_{l_1})$, let the ith block of Y with respect to V be $S = Y(p_i : p_i + n_b - 1)$ where p_i is the starting position of the ith block in Y, defined analogously to (4.3.1). If $a_i = 1$, feed the sequence S to the VT decoder and in Y, replace S with the decoded sequence. After this, the ith block in Y is deletion free, thus, the decoder updates the block-deletion pattern V by setting $a_i = 0$. We carry out this procedure for all blocks with one deletion in V. This results in a sequence $\hat{Y}$, which is obtained from Y by recovering the single-deletion blocks corresponding to block-deletion pattern V. Denote the updated version of block-deletion pattern V by $\hat{V}$. Thus at the end of this step, we have $|\mathcal{L}_1|$ updated candidate sequences $\hat{Y}_1, \cdots, \hat{Y}_{|\mathcal{L}_1|}$ with corresponding block-deletion patterns $\hat{V}_1, \cdots, \hat{V}_{|\mathcal{L}_1|}$.

Example 4.3. *Consider the code of Example 4.1 with $l_1 = 5$ blocks, and $k = 4$ deleted bits. If the list of block-deletion patterns at the end of the first step is*

$$V_1 = (1,1,1,1,0), \quad V_2 = (1,1,2,0,0),$$
$$V_3 = (1,2,1,0,0), \quad V_4 = (2,0,2,0,0),$$

then the updated list of block-deletion patterns is

$$\hat{V}_1 = (0,0,0,0,0), \quad \hat{V}_2 = (0,0,2,0,0),$$
$$\hat{V}_3 = (0,2,0,0,0), \quad \hat{V}_4 = (2,0,2,0,0).$$

Step 3: Chunk boundaries

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

For example, some of the possible matrices for $\hat{V}_4 = (2,0,2,0,0)$ in Example 4.3 are

$$\boldsymbol{A}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ \boldsymbol{A}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ \boldsymbol{A}_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{4.3.2}$$

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

Level 1 *of the tree*: Construct a sequence S by concatenating the first n_c bits of each block in $\hat{Y}$ and compute its VT syndrome $u = \mathsf{syn}(S)$. There are two possibilities:

1. $\underline{u = s_{C_1}}$: For the first chunk-string, list all valid chunk-deletion patterns of the form $(a_{11}, \ldots, a_{l_1 1})$, where $0 \le a_{i1} \le a_i$, and $\sum_{i=1}^{l_1} a_{i1} \ne 1$, since a single deletion in the chunk-string would result in $u \ne s_{C_1}$.

2. $\underline{u \ne s_{C_1}}$: List all valid chunk-vectors for the first chunk-string of the form $(a_{11}, \ldots, a_{l_1 1})$, where $0 \le a_{i1} \le a_i$, and $\sum_{i=1}^{l_1} a_{i1} \ge 1$.

Level j, $1 < j \le l_2$: Assume that we have constructed the tree up to layer $j-1$. Thus, we know the number of deletions in each chunk of the first $(j-1)$ chunk-strings. From this, we can determine the total number of deletions in the first $(j-1)$ chunks of each block. Let $d_{i,j-1}$ denote the number of deletions in the first $(j-1)$ chunks of block i. Then along this path, the jth chunk of ith block in $\hat{Y}$ is

$$S_{ij} = \hat{Y}\Big(p_i + (j-1)n_c - d_{i,j-1} : p_i + jn_c - d_{i,j-1} - 1\Big).$$

Form the jth chunk-string, $S_j = [S_{1j}, \cdots, S_{l_1 j}]$, compute its VT syndrome $u = \mathsf{syn}(S_j)$, and compare it with the correct syndrome s_{C_j}. There are two possibilities.

1. $\underline{u = s_{C_j}}$: List all valid chunk-deletion patterns for the jth chunk-string of the form $(a_{1j}, \ldots, a_{l_1 j})$, where $0 \le a_{ij} \le a_i - d_{i,j-1}$, and $\sum_{i=1}^{l_2} a_{ij} \ne 1$.

2. $\underline{u \ne s_{C_j}}$: List all valid chunk-deletion patterns for the jth chunk-string of the form $(a_{1j}, \ldots, a_{l_1 j})$, where $0 \le a_{ij} \le a_i - d_{i,j-1}$, and $\sum_{i=1}^{l_2} a_{ij} \ge 1$. If the list is empty, discard the branch. The list will be empty when there are no more deletions to assign to jth chunk-string.

At the end of step 3, the decoder provides a list of pairs $(\hat{Y}, \boldsymbol{A})$, where $\hat{Y}$ is a candidate sequence to be decoded using the chunk-deletion pattern matrix $\boldsymbol{A}$, with a_{ij} being the number of deletions in the jth chunk of the ith block. Denote the number of such pairs in the list by $|\mathcal{L}_3|$.

Step 4: Iterative correction of blocks and chunk-strings

Similarly to step 2, in step 4 we use the VT syndromes (known from M_1 and M_2) to recover deletions in blocks and chunk-strings for which the matrix $\boldsymbol{A}$ indicates a single deletion. Whenever a deletion recovered using a VT decoder lies in a chunk different from the one indicated by $\boldsymbol{A}$, the candidate is discarded. Simulations results in Section 4.4 indicate that this is an effective way of discarding several invalid candidates (see Table 4.2). The iterative algorithm is described below. For each pair $(\hat{Y}, \boldsymbol{A})$:

i) For each column of $\boldsymbol{A}$ containing a single 1 (indicating a single deletion in the corresponding chunk-string), recover the deleted bit in the chunk-string using its VT syndrome. With some abuse of notation we still refer to the restored sequence as $\hat{Y}$. If the restored bit does not lie in the expected chunk indicated by the 1, discard the pair $(\hat{Y}, \boldsymbol{A})$ and move to the next candidate pair. Otherwise, update the matrix $\boldsymbol{A}$ by replacing the 1s corresponding to the restored chunks by 0s. If there is a row in the updated matrix $\boldsymbol{A}$ with a single 1, proceed to step 4.ii).

ii) For each row of $\boldsymbol{A}$ containing a single 1 (indicating a single deletion in the corresponding block), recover the deleted bit in the block using its VT syndrome. Again, with some abuse of notation we still refer to the restored sequence as $\hat{Y}$. If the restored bit does not lie in the expected chunk indicated by the 1, discard the pair $(\hat{Y}, \boldsymbol{A})$ and move to the next pair. Otherwise, update the chunk-deletion matrix by replacing the 1s corresponding to the restored chunks to 0s. If there is a column in the updated matrix $\boldsymbol{A}$ with a single 1, go to step 4.i).

Denote the updated candidate pairs at the end of this procedure by $(\tilde{Y}, \tilde{\boldsymbol{A}})$, and assume there are $|\mathcal{L}_4|$ of them.

As an illustrative example, consider the three chunk-matrices given in (4.3.2). In $\boldsymbol{A}_1$, we can successfully recover all the deletions. In $\boldsymbol{A}_2$, we can only fix two deletions in the third block. However, for $\boldsymbol{A}_3$, we cannot recover any of the deletions. Thus, the

updated $\tilde{\boldsymbol{A}}$ matrices are

$$\tilde{\boldsymbol{A}}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ \tilde{\boldsymbol{A}}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ \tilde{\boldsymbol{A}}_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{4.3.3}$$

In Section 4.6, we discuss a method to recover remaining deletions using VT constraints and bypassing the fifth step (where we use linear codes).

Step 5: Replacing deletions with erasures

In this step, for each of the $|\mathcal{L}_4|$ surviving pairs $(\tilde{Y}, \tilde{\boldsymbol{A}})$, we replace each chunk of $\tilde{Y}$ that still contains deletions with n_c erasures. Hence, if there are ν chunks with deletions (where $1 \leq \nu \leq k$), the resulting sequence will have length n, with $n_c\nu$ erasures and no deletions. Notice that this operation of replacing with erasures can be performed without ambiguity since $\tilde{\boldsymbol{A}}$ precisely indicates the starting position of each chunk and also the number of deletions within that chunk.

The purpose of the linear code is to recover from the remaining erasures. The minimum distance of the linear code should be large enough to guarantee that we can resolve all the νn_c erased bits. In Example 4.1, as there are four deletions, we will have at most $\nu = 4$ erased chunks, so we choose a Reed-Solomon code with 4 parity check equations in $GF(2^4)$. The chunk-matrix $\tilde{\boldsymbol{A}}_3$ in (4.3.3) shows that a smaller number of parity check symbols will not suffice if we want to correct all deletion patterns.

Some invalid candidates may be discarded in the process of correcting the erasures as we may find that the parity check equations cannot be solved. We denote the number of remaining candidates at the end of this step by $|\mathcal{L}_5|$.

Remark 4.2. *Note that when the decoder uses the linear code to recover an erased chunk, it can check that whether the recovered chunk is a supersequence of the erased chunk. We can discard a candidate if this is not the case for one of the erased chunks. We do not check this condition in our simulations but we will use this condition in Section 4.7.3 to derive an upper bound for the list size.*

Step 6: Discarding invalid/identical candidates

The reconstructed sequences at the end of Step 5, denoted by $\hat{X}$, all have length n and are deletion free. For each of the $|\mathcal{L}_5|$ sequences $\hat{X}$, we check the VT and parity-check constraints for each of the block and chunk-strings and discard those not meeting any of the constraints. At the end of Step 5 it is possible to have multiple copies of the same sequence. This is due to a deletion occurring in a run that intersects two chunks (or more); this deletion can be interpreted as a deletion in either chunk, and each interpretation leads to seemingly different candidates which will turn out to be the same at the end of the process. The surviving $|\mathcal{L}_6|$ distinct sequences comprise the final list produced by the decoder.

The final list of reconstructed sequences comprises all length-n sequences that can be obtained by adding k bits to Y and also satisfy all the VT and parity check constraints. The correct sequence is always among the $|\mathcal{L}_6|$ candidates. The synchronization algorithm is said to be zero error if and only if $|\mathcal{L}_6|= 1$ for all sequences and deletion patterns. When $|\mathcal{L}_6|> 1$, the list size can be further reduced if additional hash functions or cyclic redundancy checks are available from the encoder.

4.4 Numerical examples

In this section, we present numerical results illustrating the performance of the synchronization code for various choices of the system parameters. The different setups considered for the numerical examples are shown in Table 4.1. For each setup, the performance was recorded over 10^6 trials. In each trial, the sequence X and the locations of the k deletions were chosen independently and uniformly at random. For the first five setups, we used parity check constraints from a Reed-Solomon code over $GF(2^{n_c})$ with code length $(2^{n_c}-1)$. For example, in setup 5 we used 7 parity check constraints from a Reed-Solomon code over $GF(2^6)$, hence $z = 42$ bits are needed to represent the parity check syndrome. In the last two setups, denoted with an asterisk, we used a random binary linear code, i.e., z binary parity check constraints drawn equiprobably.

Table 4.2 shows the list sizes of the number of candidates at the end of various steps of the decoding process. Recall that $|\mathcal{L}_1|$ is the number of candidate block-deletion patterns at the end of step 1, $|\mathcal{L}_3|$ is the number of pairs $(\hat{Y}, \boldsymbol{A})$ at the end of step 3, $|\mathcal{L}_4|$ is the number of pairs $(\tilde{Y}, \tilde{\boldsymbol{A}})$ at the end of step 4, and $|\mathcal{L}_6|$ is the number of sequences $\hat{X}$ in the final list. The average of $|\mathcal{L}_i|$ over one million trials is denoted by

Table 4.1: Number of deletions k, code length n, and code parameters for each setup.

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

Table 4.2: List size after each step.

	$\mathbb{E}\|\mathcal{L}_1\|$	$\mathbb{E}\|\mathcal{L}_3\|$	$\mathbb{E}\|\mathcal{L}_4\|$	$\mathbb{E}\|\mathcal{L}_6\|$	$\max\|\mathcal{L}_6\|$	$\mathbb{P}[\|\mathcal{L}_6\|> 1]$
Setup 1	1.87	1.92	1.42	1.003	3	0.003
Setup 2	1.87	1.92	1.42	1.000	2	2.5×10^{-5}
Setup 3	1.87	1.92	1.42	1	1	0
Setup 4	3.39	6.18	2.53	1	1	0
Setup 5	11.51	74.43	3.42	1	1	0
Setup 6	11.20	28.64	2.55	1	1	0
Setup 7	12.76	26.16	1.57	1	1	0

$\mathbb{E}|\mathcal{L}_i|$. The column $\max|\mathcal{L}_6|$ shows the maximum size of the final list across the one million trials. The column $\mathbb{P}[|\mathcal{L}_6|> 1]$ shows the fraction of trials for which $|\mathcal{L}_6|> 1$.

The first three setups have identical parameters, except for the number of Reed-Solomon parity checks. This shows the effect of adding parity check constraints on the list size and the rate. Adding more parity check constraints improves the decoder performance by reducing the number of trials with list size greater than one, at the expense of a rate increase.

The fourth setup is precisely the code described in Example 4.1. It has the same values of (n_c, l_1, l_2) as the first three setups but with a larger number of deletions and of parity check constraints. We observe that increasing the number of deletions (with n_c, l_1, l_2 unchanged) increases the average number of candidates in the different decoding steps. In general, choosing $l_1 \geq k$ ensures that the average list size after step 1 is small.

The fifth setup is a larger code with length $n = 378$ and can handle a larger number of deletions ($k = 7$). Though the final list size is always one, there are a large number of candidates at the end of the third step; this increases the decoding complexity. Comparing this with setup six, we observe that increasing l_2 significantly reduces the number of candidates at the end of the third step. This is because increasing

l_2 increases the number of chunk-string VT constraints, which allows the decoder to eliminate more candidates while determining chunk boundaries.

The last setup is a relatively long code. Although the average number of candidates in each of the decoding steps is not very high, we found that a small fraction of trials have a very large number of candidates, resulting in considerably slower decoding for these trials.

4.5 Complexity analysis

In this section we discuss the number of required operations for constructing the encoded message $M = [M_1, M_2, M_3]$ and for the decoding algorithm. Computing the VT syndrome of a length m sequence needs $O(m)$ arithmetic operations. For constructing M_1 we need to compute VT syndrome of l_1 blocks, each of length $n_b = n_c l_2$. This can be done with $O(n_c l_1 l_2)$ operations. Similarly, for M_2 we need $O(n_c l_2 l_1)$ arithmetic operations. Recalling that $n = n_c l_1 l_2$, both M_1 and M_2 can be computed with $O(n)$ operations. Recall that $M_3 = \boldsymbol{H}X$ is constructed via multiplication of a $z \times n$ matrix into a length n vector. This requires $O(zn)$ operations. This is the dominant term in the encoding complexity, therefore, the overall complexity of the encoder is $O(zn)$.

In the following, we analyze the decoding complexity by finding an upper bound for the complexity of each of the six decoding steps.

Step 1: Block boundaries

In the first step, we construct a tree for finding all the candidates for block boundaries. At each node of the tree, a VT syndrome of a length n_b sequence is computed and compared with the syndrome known from M_1. The tree has l_1 levels. The number of nodes in level i $(1 \leq i \leq l_1 - 1)$ can be upper bounded by the number of non-negative integer solutions of the following inequality:

$$a_1 + \cdots + a_i \leq k. \tag{4.5.1}$$

Recall that a_j shows the number of deletions in the jth block. For the last level of the tree, the number of nodes is upper bounded by the number of non-negative integer solutions of the following equation:

$$a_1 + \cdots + a_{l_1} = k. \tag{4.5.2}$$

The number of integer solutions for each equation in (4.5.1) is bounded by the number of solution to the equation in (4.5.2), which is $\binom{k+l_1-1}{k}$. Hence, there are at most $l_1\binom{k+l_1-1}{k}$ nodes in the tree. Therefore, an upper bound for the number of required operations in this step is:

$$\binom{k+l_1-1}{k} \times l_1 \times O(n_b) = O\left(n\binom{k+l_1-1}{k}\right). \tag{4.5.3}$$

Note that in the above upper bound, we did not take into account that the solutions of (4.5.1) and (4.5.2) should be consistent with the VT syndrome of blocks. As explained in the decoding section, many of the branches of the tree will get discarded because of inconsistency with VT constraints. The average number of nodes at the final level of the tree (denoted by $\mathbb{E}|\mathcal{L}_1|$) is particularly important since it will determine the average complexity of the next steps of the decoding. In table 4.3, we compare the empirical value of $\mathbb{E}|\mathcal{L}_1|$ (from Table 4.2) with $\binom{k+l_1-1}{k}$, the upper bound for $|\mathcal{L}_1|$ obtained from (4.5.2). The considerable difference between these two numbers shows the importance of using VT codes, which not only recover deletions but also decrease the complexity of the decoding algorithm by reducing the number of possibilities that need to be considered in the next steps. This lower complexity allows us to have relatively long code lengths like the code in setup 7 (see Table 4.1).

Table 4.3: Comparison of average number of surviving paths.

	Setup 1 to 3	Setup 4	Setup 5	Setup 6	Setup 7
$\mathbb{E}\lvert\mathcal{L}_1\rvert$	1.87	3.39	11.51	11.20	12.76
$\binom{k+l_1-1}{k}$	35	70	6.4×10^3	6.4×10^3	2.0×10^7

In Figures 4.4 and 4.5, we show how the empirical average $\mathbb{E}|\mathcal{L}_1|$ changes with the parameters of the code. The contributing parameters on $\mathcal{L}_1$ are k, l_1 and n_b (recall that $n_b = l_2 n_c$). In Figure 4.4, where k and n_b are fixed, it can be seen that although $\binom{k+l_1-1}{k}$ (the upper bound on $|\mathcal{L}_1|$) increases with l_1, $\mathbb{E}|\mathcal{L}_1|$ decreases. This is because a given solution of equation (4.5.2) will not be in the list $\mathcal{L}_1$ when it is not consistent with a block VT constraint. In particular, if $a_i = 0$, the VT syndrome of the sequence corresponding to ith block should match with the correct syndrome known from M_1.

Figure 4.5 shows that $\mathbb{E}|\mathcal{L}_1|$ decreases with n_b as well. The reason for this is that, as explained in the decoding section, sometimes a VT syndrome of a sequence accidentally matches with the correct VT syndrome. These accidental matches can increase the number of compatible deletion patterns at the end of the tree search. A block VT

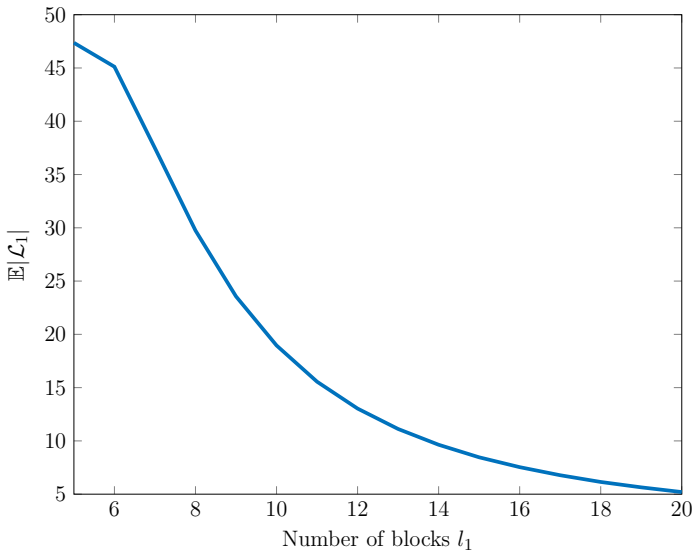

Figure 4.4: Empirical average $\mathbb{E}|\mathcal{L}_1|$ for different values of l_1 when $k = 8$, $l_2 = 7$, and $n_c = 6$.

syndrome is a number between 0 and n_b, therefore, the probability of an accidental match decreases as n_b increases.

Step 2: Primary fixing of blocks

In the second step, we use block VT syndromes to recover deletions in single-deletion blocks, there are at most l_1 such blocks. Since the VT decoding complexity is linear in n_b (the length of each block), the complexity for this step is

$$|\mathcal{L}_1| \times l_1 \times O(n_b) = O\left(n|\mathcal{L}_1|\right). \tag{4.5.4}$$

Step 3: Chunk boundaries

In this step, we find all possibilities for the number of deletions in each chunk by performing the tree search on each of the block deletion patterns produced in the first step. Consider $V = (a_1, \cdots, a_{l_1})$ to be one of these deletion patterns. After the second step of the decoding, assume that there are s blocks with non-zero number of deletions. Without loss of generality, assume that a_1 to a_s are non-zero. Since they are not recovered in the second step of the decoding we know that $a_i > 1$, also $\sum_{i=1}^{s} a_i \leq k$. Similar to the first step, the number of non-negative integer solutions of the following set of equations is an upper bound for the number of nodes in the last level of the tree which can also serve as an upper bound for the other levels. Recall that a_{ij} represents

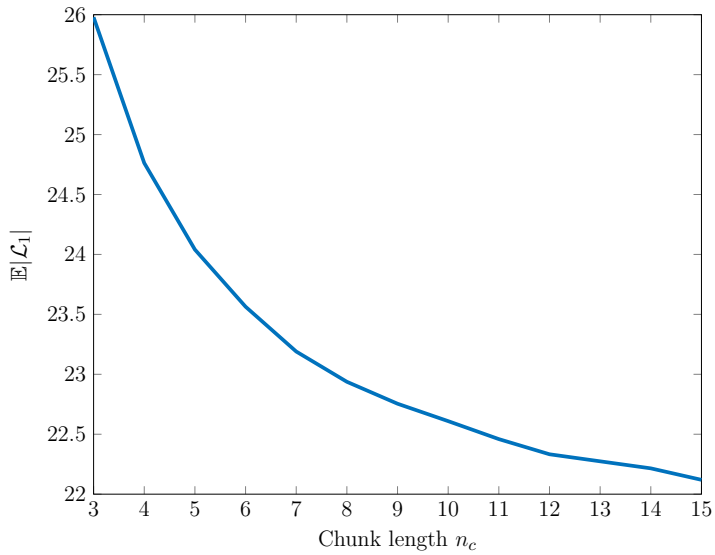

Figure 4.5: Empirical average $\mathbb{E}|\mathcal{L}_1|$ for different values of n_c when $k=8$, $l_1=9$, and $l_2=7$.

the number of deletions in the jth chunk of the ith block.

$$\begin{aligned} a_{11}+a_{12}+\cdots+a_{1l_2} &= a_1 \\ a_{21}+a_{22}+\cdots+a_{2l_2} &= a_2 \\ &\vdots \\ a_{sl_2}+a_{sl_2}+\cdots+a_{sl_2} &= a_s, \end{aligned} \tag{4.5.5}$$

To bound the complexity of this step we need the following definition and lemma.

Definition 4.1. *For a real number x and an integer number a we define $\binom{x}{a}$ as follows:*

$$\binom{x}{a} \triangleq \frac{x(x-1)\cdots(x-a+1)}{a!}. \tag{4.5.6}$$

When x is an integer, this definition reduces to the normal binomial coefficient.

Lemma 4.1. *The number of non-negative integer solutions of the set of equations in (4.5.5) when $\sum_{i=1}^{s} a_i = k$, $a_i \geq 0$, and s and l_2 are positive integers, is upperbounded by*

$$\binom{k/s+l_2-1}{l_2-1}^{s}. \tag{4.5.7}$$

Proof. See Section 4.11.3. □

Lemma 4.1 shows that (4.5.7) is an upperbound for the number of nodes in each level of the tree. Since the summation of a_i's is k and $a_i \geq 2$ for each i, s is a number

between 1 and $\frac{k}{2}$. The derivative of (4.5.7) with respect to s is positive when $s > 1$. Therefore, $s = \frac{k}{2}$ maximizes (4.5.7). Thus an upper bound for the number of nodes in each level of the tree is

$$\max_{1 \le s \le \frac{k}{2}} \binom{k/s + l_2 - 1}{l_2 - 1}^s = \binom{l_2+1}{l_2-1}^{\frac{k}{2}} = \binom{l_2+1}{2}^{\frac{k}{2}}. \tag{4.5.8}$$

Therefore we have

$$|\mathcal{L}_3| \le |\mathcal{L}_1| \binom{l_2+1}{2}^{\frac{k}{2}}. \tag{4.5.9}$$

At each node of the tree, we compute the VT syndrome of a length $n_c l_1$ sequence and compare it with the syndrome known from M_2. Therefore, the complexity of this step is

$$|\mathcal{L}_3| \times l_2 \times O(n_c l_1) \le O\left(n|\mathcal{L}_1| \binom{l_2+1}{2}^{\frac{k}{2}}\right) \tag{4.5.10}$$

$$= O\left(n|\mathcal{L}_1|(l_2/\sqrt{2})^k\right). \tag{4.5.11}$$

Similar to the first step, many of the solutions of (4.5.5) are not compatible with VT syndromes of chunk-strings. The following table compares the empirical value of $\mathbb{E}|\mathcal{L}_3|$ with the upper bound in (4.5.9), and shows the importance of VT constraints in eliminating the number of compatible deletion patterns.

Table 4.4: Comparison of average number of surviving paths after the third step.

	Setup 1 to 3	Setup 4	Setup 5	Setup 6	Setup 7
$\mathbb{E}\lvert\mathcal{L}_2\rvert$	1.92	6.18	74.43	28.64	26.16
$\mathbb{E}\lvert\mathcal{L}_1\rvert\binom{l_2+1}{2}^{k/2}$	27.48	122.04	1.3×10^6	6.8×10^6	5.2×10^{12}

Step 4: Iterative correction of blocks and chunk-strings

In this step, we iteratively use the VT decoder for blocks and chunk strings to recover deletions. Each of the VT checks will be used at most once. Since there are l_1 blocks and l_2 chunk-strings an upper bound for the complexity is

$$|\mathcal{L}_3| \times (l_1 \times O(n_c l_2) + l_2 \times O(n_c l_1)) = O\left(n|\mathcal{L}_3|\right). \tag{4.5.12}$$

Recall from the decoding algorithm that some of the candidates will be discarded in this step, therefore, $|\mathcal{L}_4| \leq |\mathcal{L}_3|$.

Step 5: Replacing deletions with erasures

In this step, we use the linear equations for recovering the erased chunks. There are at most k erased chunks and hence kn_c bits erased. Hence, the complexity of finding solutions for the set of linear equations can be upper bounded by

$$O\left(n^3|\mathcal{L}_4|\right). \tag{4.5.13}$$

We discard a candidate if there is no solution for the linear equations; therefore $|\mathcal{L}_5| \leq |\mathcal{L}_4|$.

Step 6: Discarding invalid/identical candidates

In this step, we compute the VT syndrome of blocks and chunk-strings for all the candidates on the list and compare them with the known syndromes. Hence the complexity is

$$\mathbb{E}|\mathcal{L}_4| \times O(n). \tag{4.5.14}$$

Summary

In this section, we have computed the complexity of the encoding and the different steps of the decoding. An upper bound for the decoding complexity (not considering the effect of VT codes in eliminating candidates) is $O\left(n^3\binom{k+l_1-1}{k}(\frac{l_2}{\sqrt{2}})^k\right)$. If one assumes that k, l_2, and l_1 are fixed and the length of the code is increased by increasing n_c, then the complexity of the decoding is $O(n^3)$ while the complexity of the encoding is $O(n)$.

4.6 Guess-based VT decoding

In this section, we suggest an alternative way of decoding which does not rely on linear parity checks. The purpose of linear codes is to recover deletions that cannot be directly recovered using the intersecting VT constraints. Here we first characterize these deletions, and then show how to recover them using only VT constraints. The main advantage of this approach is the rate reduction, which comes at the expense of an increased list size.

$$\boldsymbol{A}_0 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \iff$$

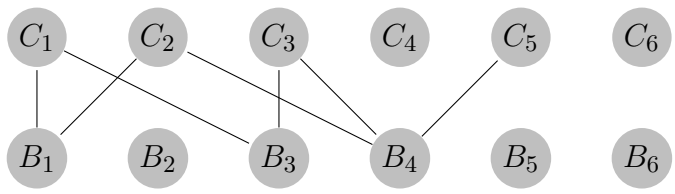

Figure 4.6: Bipartite graph corresponding to Matrix $\boldsymbol{A}_0$

4.6.1 Unresolved deletions in step four

Here we explain which deletions cannot be recovered by means of the iterative algorithm in the fourth step of decoding, for a given chunk-deletion matrix produced in the third step. We use a graph representation for the chunk-deletion matrix to illustrate this.

Definition 4.2. *Define a bipartite graph $\mathcal{G} = (\mathcal{B}, \mathcal{C})$ associated with each chunk-deletion matrix $\boldsymbol{A}$. There is a vertex in set $\mathcal{B}$ corresponding to each block (rows of $\boldsymbol{A}$) and a vertex in set $\mathcal{C}$ corresponding to each chunk-string (columns of $\boldsymbol{A}$). For any non-zero element of $\boldsymbol{A}$, like a_{ij}, there will be a_{ij} edges between the ith vertex in $\mathcal{B}$ and jth vertex in $\mathcal{C}$.*

After each iteration in the fourth step, we update the graph with respect to the updated deletion matrix. We will adopt usual definitions of paths and cycles from graph theory. In particular, if there are two edges between two vertices, we consider that as a cycle of length 2. For example, consider the matrix $\boldsymbol{A}_0$ defined below, The bipartite graph $\mathcal{G}$ given in Figure 4.6 represents this matrix. Here vertex C_i represents the ith column (chunk-string) of the matrix and B_i represents ith row (block). The deletion corresponding to edge between B_4 and C_5 will be recovered as it is the only deletion in the fifth chunk-string. But the other deletions will remain unresolved as they form a cycle in the graph. In general, in a given step of the iterative algorithm, we will recover a deletion if and only if the corresponding edge in the graph is connected to a vertex of degree one. Consequently, when the iterative algorithm finishes, there will be no degree one vertex. The following result determines the graph configurations that result in deletions that cannot be recovered.

Proposition 4.1. *A deletion occurring in the jth chunk of the ith block will not be recovered by means of the iterative algorithm if and only if, the corresponding edge, B_iC_j, in the graph $\mathcal{G}$ belongs to a cycle, or belongs to a path between two cycles.*

Proof. Consider an unrecovered edge B_iC_j which does not belong to a cycle. As the degree of B_i is greater than one, we can find a vertex other than C_j connected to B_i. Similarly, the degree of that vertex is greater than one, hence we can continue this

procedure. Since the graph is finite we revisit a vertex which means there is a path from B_i to a cycle. By repeating this argument for C_j, we conclude that B_iC_j is in a path which connects two cycles. □

4.6.2 Guess-based decoding

Here we show how to recover the remaining deletions from step four of the decoding using VT constraints only by guessing bits to break cycles in the bipartite graph. Since there are no parity check constraints, the new rate is:

$$R = \frac{\lceil \log(n_c l_2 + 1) \rceil}{n_c l_2} + \frac{\lceil \log(n_c l_1 + 1) \rceil}{n_c l_1}, \tag{4.6.1}$$

which is a saving of $\frac{z}{n}$ over the rate in (4.2.2). Consider the matrix $\boldsymbol{A}_0$ and its corresponding graph given in Figure 4.6. If we recover one of the deletions in the cycle, then we can recover all other remaining deletions using the iterative algorithm (as there is no other cycle in the graph). This motivates us to guess the deleted bit in one of the chunks in the cycle. For instance, we can guess the deleted bit in the first chunk. Since the first chunk length is $(n_c - 1)$, there are $(n_c + 1)$ distinct sequences that can be obtained by inserting one bit into this chunk. In general, number of distinct supersequences that can be obtained by inserting a bits in a length $(n_c - a)$ binary sequence is [91]

$$\sum_{j=0}^{a} \binom{n_c}{j} \leq (n_c + 1)^a. \tag{4.6.2}$$

The decoder can run the iterative algorithm of the fourth step on each of the $n_c + 1$ obtained sequences. Since there are no other cycles in the graph, the iterative algorithm will either successfully find all the remaining deletions, or discard the sequence due to the incompatibility of the position of the recovered bits with their expected chunk (known from $\boldsymbol{A}_0$). The decoder then forwards the remaining sequences, which now are of length n, to the sixth step of the decoding algorithm (bypassing the fifth step).

In general, in order to find which deletions need to be guessed, Proposition 4.1 indicates that it is necessary and sufficient to remove a number of edges such that there are no more cycles in the graph. Hence, the minimum number of edges that need to be removed to make the graph acyclic is equal to the minimum number of bits that need to be guessed. Denote this number by q. If there are c connected components in the graph with $\alpha_1, \cdots, \alpha_c$ vertices, then $a^* = e - (\sum_{i=1}^{c} \alpha_i) + c$, where e is the total number of edges in the graph (total number of deletions). Using equation (4.6.2), $(n_c + 1)^{a^*}$ is an upper bound for the number of different possibilities for the guessed

Table 4.5: Number of deletions and code parameters for each setup.

	k	n	l_1	l_2	n_c	z	R_{sync} (R'_{sync})
Setup 8	3	60	5	3	4	0 (4)	0.583 (0.650)
Setup 9	3	60	5	3	6	0 (6)	0.444 (0.511)
Setup 10	4	60	5	3	4	0 (16)	0.583 (0.850)
Setup 11	4	60	5	3	6	0 (24)	0.444 (0.711)

Table 4.6: List size distribution.

	$\lvert\mathcal{L}_6\rvert = 1$	$\lvert\mathcal{L}_6\rvert = 2$	$\lvert\mathcal{L}_6\rvert = 3$	$\lvert\mathcal{L}_6\rvert > 3$	$\max\lvert\mathcal{L}_6\rvert$
Setup 8	92.7%	6.3%	0.91%	0.09%	6
Setup 9	85.9%	10.2%	2.8%	1.1%	10
Setup 10	84.3%	12.8%	2.3%	0.6%	13
Setup 11	71.9%	18.2%	6.1%	3.8%	25

sequence. In our algorithm, the decoder chooses one of the edges in a cycle uniformly at random, removes it by guessing a bit in the corresponding chunk, and then performs the iterative algorithm on the updated graph and discard inconsistent candidates. If there are any unresolved deletions, it chooses another edge from a cycle uniformly at random and repeat the algorithm until there are no more edges in the graph.

In the following, we present the result of simulations for the setups listed in Table 4.5. The performance was recorded over 10^6 simulation trials. Setup 8 and 10 are similar to setups 1 and 4 in Section 4.4 respectively. The only difference is that there is no linear code in setups 8 and 10. The rate R'_{sync} in brackets is the higher overall rate when linear codes were used. The first three columns in Table 4.6 show the percentage of trials in which the final list size was exactly 1, 2, and 3 respectively. The fourth column shows the number of trials with more than 3 candidates on the final list. Finally, the last column shows the largest list size over all 10^6 trials. Comparing setups 8 and 10 in Table 4.6 with setups 1 and 4 in Table 4.2 shows a significant increase in the probability of having more than one candidate on the final list. Comparing setups 8 and 9 shows that increasing n_c decreases the rate (see (4.6.1)), but increases the average list size. The reason for this is that, in the cases where deletion pattern contains cycles, the number of possible guesses increases with n_c. The same effect can be observed by comparing setups 10 and 11. Also comparing the setup 8 with 10 (and also 9 with 11) shows that increasing k will increase the list size. Guess-based iterative decoding is effective if we are willing to tolerate list sizes greater tan one with non-negligible probability.

4.7 Recovering deletions using erasure codes and hashes

In this section we first propose a simple code for recovering deletions using erasure codes and random hashes. Then we show the connection between Guess and Check codes [76] and this method. This method also provides an alternative interpretation to the multilayer code which we will use to find a bound for the list size.

Let us revisit the main problem. We want to construct a message M for recovering length n binary sequence $X = x_1x_2\cdots x_n$ from its length $(n-k)$ subsequence Y. We can interpret this problem as having k erasures in some unknown positions of the sequence Y. There are $\binom{n}{k}$ ways for choosing positions of k erasures. Given the positions of the erasures in Y, one can use an erasure code to reconstruct X. For recovering k erasures we need a code with minimum distance of $k+1$. For different values of n and k various codes are suggested in the literature with minimum distance of at least $k+1$ and lower bounds are also available on the rate of such a code (e.g. see Gilbert-Varshamov bound [107, Chapter 4]). Here we denote the rate of a code with minimum distance of at least $k+1$ with R_{k+1}.

Notice that there are $\binom{n}{k}$ possibilities for guessing the positions of erasures which gives us up to $\binom{n}{k}$ different reconstructed sequences. Denote these sequences by $X_1, X_2\cdots, X_t$ where $t \leq \binom{n}{k}$. Since we are considering all possibilities, the correct position of the erasures will be also considered, hence, X is among X_1 to X_t. A natural way to find X is using a number of random hash functions to discard other sequences. This hash function can have different forms. For now consider a universal hash function which is define for the sequence X as follows.

$$h(X) = \sum_{i=1}^{n} g_i x_i \pmod{s}, \tag{4.7.1}$$

where g_i's are uniformly distributed over $\{0,1,\cdots,s\}$, and s is an arbitrary positive integer. If the jth reconstructed sequence $X_j \neq X$, then $h(X_j)$ will be a random number in $\{0,1,\cdots,s\}$. Assume that $X_j = x_{j1}\cdots x_{jn}$,

$$h(X_j) - h(X) = \sum_{i=1}^{n} g_i(x_{ji} - x_i) \pmod{s}, \tag{4.7.2}$$

since $X_j \neq X$ there exist some i where $x_{ji} \neq x_i$, thus $h(X_j) - h(X)$ is uniformly distributed over $\{0,1,\cdots,s\}$. Hence, the probability of $h(X_j) = h(X)$ is $\frac{1}{s}$.

Consider r different hashes $h_1, h_2, \cdots, h_r$ which are independently produced (i.e. their coefficients are independently and uniformly produced from $\{0, 1, \cdots, s\}$). Similarly one can show that if $X_j \neq X$, the probability of $h_p(X_j) = h_p(X)$ for all $1 \leq p \leq r$, is $\left(\frac{1}{s}\right)^r$.

For the sequence X, we construct the message M by concatenating R_{k+1} bits of the erasure code and r hashes $h_1(X), \cdots, h_r(X)$. Therefore, the number of required bits to represent M is $R_{k+1} + r\lceil \log s \rceil$.

At the decoder we consider all $\binom{n}{k}$ possibilities for positions of erasures in Y. For each of them we use the first k bits of M to recover the erasures. This will give us sequences $X_1, \cdots, X_t$. Now for all $1 \leq j \leq t$ we compute $h_1(X_j), \cdots, h_r(X_j)$ and compare it with $h_1(X), \cdots, h_r(X)$ (known from M). If X_j satisfies all r constraints imposed by the hashes, we put X_j in the final list. We know that X will be in the list. If the list size is 1, then we have successfully decoded X.

Denote the final list by $\mathcal{L}_r$. As discussed earlier for any $X_j \neq X$ the probability of $X_j \in \mathcal{L}_r$ is $\left(\frac{1}{s}\right)^r$. Thus we have the following upper bound for the expected list size,

$$\mathbb{E}|\mathcal{L}_r| \leq \binom{n}{k}\left(\frac{1}{s}\right)^r + 1. \tag{4.7.3}$$

The number of required hashes in order to constraint the expected value of the list size to be less than two can be derived by imposing that

$$\binom{n}{k}\left(\frac{1}{s}\right)^r \leq 1. \tag{4.7.4}$$

Notice that $\binom{n}{k}$ can be upper bounded by $\left(\frac{en}{k}\right)^k$ (the proof is simple noting that $e^k > \frac{k^k}{k!}$). Hence, a sufficient condition for s and r is

$$\left(\frac{en}{k}\right)^k \leq s^r. \tag{4.7.5}$$

This gives $r \log s \geq k \log(en/k)$. Thus the number of required bits to represent M is

$$R_{k+1} + \lceil r \log s \rceil \leq R_{k+1} + k\lceil \log \frac{en}{k} \rceil. \tag{4.7.6}$$

One should note that we need to consider all $\binom{n}{k}$ possibilities for the decoding which is computationally expensive. The following subsection explains how to decrease complexity in expense of rate.

4.7.1 Rate, complexity tradeoff

Consider again the problem of recovering X from Y. Divide the sequence X into chunks of length n_c, thus we have $n = ln_c$. Here, $X((i-1)n_c+1 : in_c)$ is the ith chunk of the sequence. Now assume that Y can be obtained from X by deleting α_i bits from ith chunk of X. Therefore,

$$\sum_{i=1}^{l} \alpha_i = k. \tag{4.7.7}$$

We call any integer vector $(\alpha_1, \cdots, \alpha_k)$ which satisfies (4.7.7) with $\alpha_i \geq 0$, a deletion pattern. The number of different deletion patterns can be upper bounded by $\binom{l+k-1}{k}$ which is the number of integer (non-negative) solutions of (4.7.7) (the bound is exact if $n_c \geq k$).

In contrast with the previous section where we guess the positions of deletions, here we guess the deletion pattern. Thus, we will not know the exact position of a deletion but just the chunk that contains it. For the decoding, when a chunk contains deletions, we replace the entire chunk with n_c erasures and then use an erasure code to recover erased chunks. We should modify the encoding process of the previous section, by considering each chunk as a symbol sending k parity check symbols from an MDS code for recovering erasures. Note that the alphabet size is 2^{n_c}. Thus, we need kn_c bits for sending these parity check symbols. Here again, we use r random hashes defined in (4.7.1).

If we use the same estimation method for the number of required hashes, we have

$$\binom{l+k-1}{k}\left(\frac{1}{s}\right)^r \leq 1. \tag{4.7.8}$$

Here, the LHS is an upper bound for the expected number of the sequences that survive the hashes. We can upper bound the binomial term using Lemma 4.2,

$$\binom{l+k-1}{k} \leq \left(\frac{en}{n_c k}+1\right)^k. \tag{4.7.9}$$

Therefore, the number of required bits to represent M is

$$k(n_c - \log n_c) + k \log \frac{e(n+kn_c)}{k}. \tag{4.7.10}$$

By comparing (4.7.6) and (4.7.10), one can notice that by increasing n_c the rate also increases almost linearly. In exchange, number of possibilities that need to be checked (and thus the complexity) is decreasing by the factor of n_c^k.

4.7.2 Relationship with guess and check code

The guess and check code proposed in [76] can be compared with the method we introduced in this section. In that work, the length n sequence X is divided into chunks of length $\log n$. Therefore, we have $l = \frac{n}{\log n}$. Then the message M is consist of c MDS parity checks where $c > k$. Hence, we can consider k of the parity checks as the erasure code and the other $c-k$ as the hashes ($r = c-k$).

The decoding algorithm is similar to what we suggest in the previous subsection. The decoder consider all $\binom{k+l-1}{k}$ possibilities for the deletion positions. Use k erasure code to recover the erased chunk and then check whether the recovered sequence satisfies the other $c-k$ parity checks (hashes).

Note that each hash is a symbol with $\log n$ bits, thus it represents a number between 0 and $(n-1)$. Therefore, using the notation in this section we have $s = n$. Now (4.7.8) suggests to choose $r \geq k$ which means $c \geq 2k$, this is exactly the condition found in [76]. Of course, the importance of the analysis given in [76] is that it proves the vanishing error (by bounding probability of error) when $c > 2k$ while the argument proposed here is an approximation. Since it was for a random hash and it is not clear that can be generalized the MDS parity checks used in [76].

In general, an advantage of multilayer codes in comparison with the code in [76] is the lower complexity of the decoding. As discussed in Section 4.5, different layers of VT codes significantly decrease the number of cases that the decoder needs to check (see Table 4.3), whereas in [76] all possibilities should be visited by the decoder. The existence of rigorous analysis for error probability of the coding scheme in [76] is an advantage for guess and check method. In the following we give an upper bound for the list size of Multilayer code.

4.7.3 List size analysis for Multilayer code

Assume that the linear code in a multilayer code is capable of recovering k erased chunks. This means that the minimum distance of the linear code is at least $kn_c + 1$ (which implies $z \geq kn_c$). In the multilayer construction we have $l = l_1 l_2$ chunks. Therefore, if we use the decoding procedure that we described in this section, there are $\binom{k+l_1 l_2-1}{k}$ solutions for the following equation (deletion patterns) which the decoder needs to check.

$$a_{11} + a_{12} + \cdots + a_{l_1 l_2} = k, \tag{4.7.11}$$

where a_{ij} is the number of deletions in jth chunk of ith block. Then the decoder erases the chunks with deletions according to the guessed deletion pattern, and uses

the linear code to recover the erased chunks. Note that the recovered bits should be a supersequence of the erased chunk, otherwise the decoder can discard the candidate. Here, we use this fact to obtain an upper bound on the list size. We need the following assumption:

Assumption 1. When the deletion pattern is not correct, if the set of parity check constraints has a solution for the erased chunks corresponding to the deletion pattern, then the recovered bits will be uniformly random.

We illustrate this assumption in the following example.

Example 4.4. *Assume that* $n_c = 3$ *and* $l_1 = l_2 = 2$*, also there are* $k = 3$ *deletions. Let* $X = \underline{1}01\ 100\ 011\ 1\underline{00}$ *and* $Y = 011000111$*. The underlined bits are deleted from* X *to get* Y*. Consider an incorrect deletion pattern* $(0,2,0,1)$*. According to this pattern, the decoder erases the second and fourth chunks to get* $Y' = 011\ xxx\ 001\ xxx$*, where* x *indicated an erased bit. Then, it uses the linear code to recover the erasures. Assumption 1 states that the recovered bits in second and fourth chunks of* Y' *are uniformly random and hence independent from the erased bits, namely* 0 *in the second chunk and* 11 *in the fourth chunk.*

Assumption 2. Given an incorrect deletion pattern, if the decoder uses $z' < z$ linear constraints to recover the erasures imputed with respected to the incorrect pattern, the recovered sequence satisfies each of the remaining constraints independently with probability $\frac{1}{2}$. Therefore, the probability that the resulting sequence satisfies all the remaining linear equations is $\left(\frac{1}{2}\right)^{z-z'}$.

The motivation for this assumption is that when an incorrect pattern is considered, the ith bit in the recovered sequence is not representing the actual ith bit of the sequence, furthermore, using Assumption 1, the recovered erased bits are uniformly random. Hence, it is plausible to assume that the probability of the satisfaction of a binary linear constraint will be 1/2. Given the two above assumptions, we prove the following proposition:

Proposition 4.2. *Under Assumptions 1 and 2, the average final list size of a multilayer code, when the minimum distance of the linear code is more than* kn_c*, satisfies the following inequality:*

$$\mathbb{E}|\mathcal{L}_6| \leq 1 + \left(\frac{e(k+l_1l_2)(n_c+1)}{k2^{n_c}}\right)^k. \tag{4.7.12}$$

Proof. Consider an incorrect deletion pattern $(a_{11}, \cdots, a_{l_1l_2})$, where the deletions are in $k' \leq k$ chunks. Now consider the jth chunk of the ith block (assume that $a_{ij} > 0$).

Using the first assumption, the probability that the n_c bits recovered using the linear code in this chunk are a supersequence of the current $(n_c - a_{ij})$ bits in this chunk is:

$$\frac{\sum_{m=0}^{a_{ij}} \binom{n_c}{m}}{2^{n_c}} \leq \frac{(n_c+1)^{a_{ij}}}{2^{n_c}}. \tag{4.7.13}$$

The numerator on the LHS is the number of supersequences of length n_c for the erased chunk, and the denominator is the total number of length n_c binary sequences. Now $k'n_c$ independent linear equations are enough to recover $k'n_c$ erasures. Using the second assumption, the probability that the recovered sequence satisfies the remaining parity check equations is

$$\frac{1}{2^{z-k'n_c}} \leq \left(\frac{1}{2^{n_c}}\right)^{k-k'}. \tag{4.7.14}$$

Using (4.7.13) and (4.7.14), we have the following upper bound on the probability that the assumed incorrect deletion pattern ends up on the final list:

$$\left(\frac{1}{2^{n_c}}\right)^{k-k'} \prod_{i,j:a_{ij}\geq 1} \left(\frac{(n_c+1)^{a_{ij}}}{2^{n_c}}\right) = \left(\frac{1}{2^{n_c}}\right)^{k-k'} \left(\frac{(n_c+1)^k}{2^{k'n_c}}\right) \tag{4.7.15}$$

$$= \left(\frac{n_c+1}{2^{n_c}}\right)^k, \tag{4.7.16}$$

where (4.7.15) holds because $\sum_{i,j} a_{ij} = k$ and there are k' chunks with deletions. Now using this upper bound for each of the guess patterns, and noting that the correct pattern will always be on the list we have

$$\mathbb{E}|\mathcal{L}_6| \leq 1 + \binom{k+l_1l_2-1}{k}\left(\frac{n_c+1}{2^{n_c}}\right)^k \tag{4.7.17}$$

$$\leq 1 + \left(\frac{e(k+l_1l_2)}{k}\right)^k \left(\frac{n_c+1}{2^{n_c}}\right)^k. \tag{4.7.18}$$

□

As n_c increases the second term goes to zero (when other parameters of the code is fixed), and the average of the list size goes to one. For having $\mathbb{E}|\mathcal{L}_6| < 2$ we need to choose parameters such that:

$$n_c - \log(n_c+1) > \log(k+l_1l_2) + \log k + \log e. \tag{4.7.19}$$

In the above argument we did not consider VT constraints for obtaining the bound. In the multilayer code, we can think of VT constraints as hashes. Taking this into account can give a better bound for the list size. There is a major difference between the structure of hashes defined in (4.7.1) and VT codes used in the multilayer code. A hash function in (4.7.1) is defined over the sequence X, while a VT code is defined over a block or chunk-string which is a subsequence of X. As a result some of the VT constraints may be dependent in the multilayer construction. Further analysis of the list size that take into account the effect of VT constraints is a direction for future work.

4.8 Decoding insertions and deletions

In this section, we discuss the required modifications in the decoding algorithm such that it can recover a combination of up to k deletions and insertions. We use the same message M constructed in Section 4.2 with a modified decoder. First notice that for the case where we have only insertions we can use almost the same decoding algorithm we used for the deletion only case (recall that VT codes can also recover a single insertion in a sequence).

For the case that both insertions and deletions are possible, assume that the sequence Y is of length m (where $n-k \leq m \leq n+k$) and can be obtained from X by a deletions and b insertions where $a+b \leq k$ (thus $m = n-a+b$). We will use a similar six-step decoding for reconstructing sequence X.

Step 1

In this step, we perform a tree search to find the number of insertions and deletions in each of the blocks. The output of this step is a list of edit vector candidates where each edit vector comprises a deletion pattern vector V_d and an insertion pattern vector V_i.

$$V_d = (a_1, a_2, \cdots, a_{l_1}), \quad V_i = (b_1, b_2, \cdots, b_{l_1}). \tag{4.8.1}$$

Each entry of V_d and V_i shows the number of deletions and insertions respectively in the corresponding block. We know that

$$\begin{aligned}\sum_{i=1}^{l_1}(a_i+b_i) &\leq k,\\ \sum_{i=1}^{l_1}(a_i-b_i) &= n-m.\end{aligned} \tag{4.8.2}$$

At each node of the tree we compute the VT syndrome of the next block. Assume that for a given node at level j, in total d_j deletions and ι_j insertions occurred in $j-1$ previous blocks. The starting point of the jth block is

$$p_j = (j-1)n_b - d_j + \iota_j + 1. \tag{4.8.3}$$

Note that given (4.8.2), we know that a_j and b_j should satisfy

$$a_j \leq \frac{k+n-m}{2} - d_j, \quad b_j \leq \frac{k+m-n}{2} - \iota_j. \tag{4.8.4}$$

We can compute the VT syndrome of the jth block, $\mathsf{syn}(Y(p_j : p_j + n_b - 1))$. If it does not match with s_{B_j}, the correct VT syndrome of this block, the possible values for a_j and b_j are all the pairs which satisfy (4.8.4) and also $a_j + b_j \neq 0$. If $d_j + \iota_j = k$, then we discard the correspond branch of the tree. If the VT syndrome of the block matches with s_{B_j}, then the possible values for a_j and b_j are all the pairs which satisfy (4.8.4) and also $a_j + b_j \neq 1$.

The total number of solutions of (4.8.2) can be upper bounded by

$$\sum_{i=1}^{k/2}\binom{i+l_2-1}{i}^2. \tag{4.8.5}$$

This upper bound will be achieved when $m = n$ and $a + b = k$. We expect many of these solutions to be inconsistent with the VT syndrome and thus does not appear on the first step list. Comparing (4.8.5) with the upper bound for the deletion only case shows that the complexity of the decoding is higher when both insertions and deletions are possible.

Step 2

For any given edit vector produced from step one, we can recover the edit in the blocks with single insertion or deletion, i.e. when $a_i + b_i = 1$. After recovering the edit in a block we should update the edit vector accordingly.

Step 3

The goal of this step is to create a list of edit matrices where each edit matrix comprises a deletion matrix $\boldsymbol{A}$, and an insertion matrix $\boldsymbol{B}$. a_{ij} in $\boldsymbol{A}$ shows the number of deletions in the jth chunk of ith block, and b_{ij} in $\boldsymbol{B}$ shows the number of insertions in this chunk. Similar to step 3 in the deletion decoding, we construct this matrix for any edit vector candidate via a tree search.

For each node of the tree at level j,we compute the VT syndrome of jth chunk-string and compare it with the correct syndrome known from M. Note that for each node at level j of the tree, we know a_{ih} and b_{ih} for all i and $h < j$. Thus we know the starting position of the jth chunks of blocks, and therefore we can form the jth chunk-string and compute its VT syndrome. Consider the following two possibilities:

1. If the VT syndrome of the chunk-string matches with s_{C_j} (the correct syndrome of the jth chunk-string), the possible values for a_{ij} and b_{ij} are all non-negative integers that satisfy $\sum_{i=1}^{l_1}(a_{ij} + b_{ij}) \neq 1$ and also:

$$a_{ij} \leq a_i - \sum_{h=1}^{j-1} a_i h \quad \text{and} \quad b_{ij} \leq b_i - \sum_{h=1}^{j-1} b_i h. \tag{4.8.6}$$

2. If the VT syndrome of the chunk-string does not match with s_{C_j}, the possible values for a_{ij} and b_{ij} are all non-negative integers that satisfy (4.8.6), and also $\sum_{i=1}^{l_1}(a_{ij} + b_{ij}) \neq 0$. The node will be discarded if

$$a_i = \sum_{h=1}^{j-1} a_i h \quad \text{and} \quad b_i = \sum_{h=1}^{j-1} b_i h, \quad \text{for} \quad 1 \leq i \leq l_1. \tag{4.8.7}$$

Since there are up to k edits in the sequence, number of non-zero value among a_i's and b_i's is at most k. If we denote this number wit s then with an argument similar to step 3 of the deletion only case in Section 4.5, we can give this upperbound for the

number of edit matrix candidates

$$\binom{k/s+l_2-1}{l_2-1}^s. \tag{4.8.8}$$

Note that, here unlike the deletion only case we cannot claim that a_i's and b_i's are greater than one. This is because if there is exactly one deletion and one insertion in a block we will not be able to recover them in the second step of the decoding. Therefore $s=k$ is a possibility and since the derivative of (4.8.8) with respect to s is positive, $s=k$ maximizes (4.8.8). Therefore, an upper bound for the number of solutions is l_2^k. Similar to the deletion case, we expect many of the solutions to be discarded due to inconsistency with VT syndrome.

Steps 4 to 6

The last three steps are very similar to the deletion only case. In step 4, perform the iterative algorithm on all candidates i.e., if there exist a block or chunk-string with single edit we recover the edit using VT decoder an update the edit matrices. Then we repeat this until there is no more single edits in blocks or chunk-strings. Similar to the deletion only case, we will discard a candidate if the recovered bit lies in a wrong chunk.

In step 5, we replace any chunk which still contains edits with n_c erasures, and use the linear code to recover erasures. This will be done for all the candidates, and any candidate which is inconsistent with the linear constraints will be discarded. Finally, at the sixth step we check all the constraints for all candidates and discard inconsistent candidates.

4.9 Multilayer codes for deletion channels

The multilayer construction can be adapted to the deletion channel which introduces k deletions in a block of n bits. Here the goal is to transmit one of the 2^{nR} messages, where $R \leq 1$ is the communication rate. A codebook can be produced for the deletion channel based on multilayer construction for message M. The channel codebook consists of length n sequences where blocks and chunk-strings satisfy a pre-specified VT syndrome, and also a pre-specified linear code coset. For a length $n = n_c l_1 l_2$ sequence, there are $(n_c l_2+1)^{l_1}(n_c l_1+1)^{l_2}$ possibilities for the VT syndromes of l_1 blocks and l_2 chunk-strings and also 2^z possibilities for the coset of z linear codes. Therefore, there

exists a set of syndromes and a coset of linear code for which there are at least

$$\frac{2^n}{2^z(n_c l_2+1)^{l_1}(n_c l_1+1)^{l_2}} \tag{4.9.1}$$

sequences with these parameters. We can use this set of sequences as the codebook, and use the same decoding algorithm described in Section 4.3 for decoding. The rate of this code will be at least

$$R \geq 1-\frac{\log_2(n_c l_2+1)}{n_c l_2}-\frac{\log_2(n_c l_2+1)}{n_c l_1}-\frac{z}{n_c l_1 l_2}. \tag{4.9.2}$$

From the codebook definition we know the VT syndrome of the blocks and chunk-strings, and also the linear code coset. Hence, the decoding is exactly the same as the decoding process explained in Section 4.3. However, this code construction does not immediately suggest an efficient method for mapping information bits onto codewords. Also, finding the codebook becomes unfeasible even for a relatively small n, this is because we need to calculate syndromes and linear cosets for all length n sequences in order to find the codewords. In the following we suggest a heuristic method for systematically mapping the data bits into a codeword. The codewords are designed based on the multilayer and repetition codes. The rate of the proposed code is less than the lower bound in (4.9.2).

4.9.1 A heuristic encoding scheme for the deletion channel

First recall the systematic encoding of the VT codes from Chapter 2. The idea was to reserve bits in dyadic positions for adjusting the VT syndrome of the sequence, and assign data bits to the rest of the bits. The rate of the systematic code is:

$$R=1-\frac{\lceil \log(n+1)\rceil}{n}. \tag{4.9.3}$$

We explain our proposed encoding scheme via an example. Recall the parameters of setup 7 from Table 4.5. Assume that we want to have the same number of blocks and chunk-strings in our codewords. We use the above systematic encoding method to encode data bits into all the blocks but block 3 and 15 (assume that our desired syndrome is 0). Therefore we embed

$$(l_1-2)(n_c l_2-\lceil \log(n_c l_2+1)\rceil)=2376 \tag{4.9.4}$$

bits. And the VT syndrome of all the blocks except for the third and the 15th are set to be 0. Now in each chunk-string there are two chunks that are yet to be filled, namely, the third and 15th chunks. The idea is to use these two chunks to adjust the VT syndrome of the chunk-strings. The reason that we chose these two specific chunks is that we can find appropriate $2n_c$ remaining bits in these chunks to adjust the VT syndrome of the chunk-strings (this has been checked by computer search). Assume that we want all the chunk-strings to have VT syndrome 0. We can compute the current VT syndrome (excluding the two chunks) and find the deficiency (the difference between the calculated syndrome and 0), then use a look up table that give us the two appropriate chunks for that deficiency.

Up to this point we have embedded the data bits into the sequence, and also adjusted the VT syndrome of all chunk-strings, and all but two of the blocks. Now we compute the VT syndromes of the two remaining blocks, and also the coset of the linear code for the length $n_c l_1 l_2$ sequence. The VT syndrome is a number between 0 and 140 and can be represented with 8 bits. Also, we have $z = 60$ hence if we concatenate the VT syndrome of two blocks and the linear coset of the sequence we have a message of length 76 bits. We will use a repetition code with $k+1$ repetitions on this message and append the resulting sequence at the end of the length $n_c l_1 l_2 = 2800$ sequence. Therefore, the total length of the sequence (codeword) will be 3636 bits. The decoder will first decode the repetition code and find the VT syndromes of two blocks and the coset of linear code. Then we can use the decoder that we described in Section 4.3 to decode the rest of the codeword. The rate of the code is $0.65 = \frac{2376}{2800+76\times 11}$. The rate lower bound of the impractical code which is given in (4.9.2) is 0.87.

The main question in this scheme is whether we can find two chunks that can generate all the syndromes in a chunk-string. For a fixed n_c as number of chunks in a block increases (l_1 increases) finding two chunks to produce all the syndromes becomes less likely. A pair of chunks can produce at most 2^{2n_c} different syndromes. Therefore, when $n_c l_l$ (the length of a chunk-string) is greater than 2^{2n_c} we cannot find a pair of chunks that can produce all syndromes (this is a necessary condition). Using computer search we found that, for example, when $n_c = 4$, we can always find a pair of chunks that produces all the syndromes for $l_1 \leq 24$, when $n_c = 5$, we can find a pair for $l \leq 42$. If we fail to find a pair of chunks (due to large l_1) we can attempt to produce all syndromes with more chunks, and consequently reserve more blocks for adjusting chunk-strings syndrome, this will reduce the rate.

4.10 Extensions and discussion

In this section we describe two extenstions to the basic multilayer code described in the previous sections, and conclude the chapter with a short discussion.

4.10.1 Non-binary multilayer codes

Non-binary VT codes over q-ary alphabets (for $q > 2$) were introduced in [17]. Similar to the binary VT codes these codes also partition the length n sequences into qn classes. Each of the classes is a single deletion (or insertion) correcting code with linear encoding and decoding complexity [77]. In our code construction we can replace the binary VT codes with the non-binary version, and the binary linear codes with a suitable non-binary linear codes to get a non-binary version of the multilayer codes.

4.10.2 Multilayer codes with more than two layers

Here we show how one can generalize the proposed two-layer construction of multilayer codes to a three-layer construction, the generalization to more than three layers is similarly possible. First consider this example. Divide the sequence $X = [V_1, V_2, \cdots, V_8]$ into eight chunks of length n_c. There are three kinds of intersecting VT constraints in the code. Two block constraints are the VT syndromes of $[V_1, V_2, V_3, V_4]$ and $[V_5, V_6, V_7, V_8]$. Two chunk-string constraints are the VT syndrome of $[V_1, V_2, V_5, V_6]$ and $[V_3, V_4, V_7, V_8]$. The third set of constraints are VT syndromes of $[V_1, V_3, V_5, V_7]$ and $[V_2, V_4, V_6, V_8]$. In Figure 4.7, we use factor-graph convention to illustrate the construction of the code. Each of the chunks is represented via a variable node, and each of the VT constraints is represented via a check node. We denoted block constraints by letter B and chunk-string constraints by letter C and third layer constraints by letter T. This graph is essentially equivalent of Figure 4.2, but for a three layers construction. The fact that there are three layers in the code can be noted from the degree of variable nodes.

In general assume a length $n = n_c l_1 l_2 l_3$ sequence. We construct the message $M = [M_1, M_2, M_3, M_4]$, by first dividing the sequence into l_1 equal-size blocks. Similar to the two-layer code, M_1 contains the VT syndrome of these blocks. Then encoder splits each block into l_2 chunks and forms the chunk-strings as in Section 4.2. The VT syndrome of all the chunk-strings are contained in M_2. Divide each chunk to l_3 equal-size sub-chunks each of length n_c. Then form l_3 sequences each of length $n_c l_1 l_2$ by concatenating sub-chunks of different $l_1 l_2$ chunks. The VT syndrome of all these

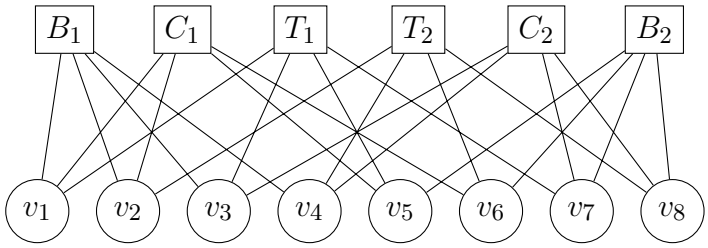

Figure 4.7: Factor graph representation for a three-layer code

sequences come to M_3. Finally M_4 is the coset of z parity check equations. The rate of the code will be

$$R_{\text{sync}} = \frac{\lceil \log(n_c l_1 l_2 + 1) \rceil}{n_c l_1 l_2} + \frac{\lceil \log(n_c l_1 l_3 + 1) \rceil}{n_c l_1 l_3} + \frac{\lceil \log(n_c l_2 l_3 + 1) \rceil}{n_c l_2 l_3} + \frac{z}{n}. \tag{4.10.1}$$

It is possible to devise a decoding algorithm for this code similar to the six-steps algorithm for two-layer code which we described in Section 4.3. For this code two extra steps are needed in the decoding due to the extra layer of VT constraints in the code. Here we only provide a high level idea of the decoder. The first four steps are similar to the steps in two-layer decoder, where we find all the acceptable boundaries for blocks and chunks and also recover deletions in blocks or chunk-strings with single deletion for all of the candidates. In the fifth step, the decoder determines the boundaries of sub-chunks, similarly to steps one and three of the two-layer decoder. This can be done via a tree search. Then in the sixth step, the decoder uses an iterative algorithm (similar to the fourth step of two-layer decoder) to recover any single deletion in any of the three layers of VT constraints. The seventh step will be replacing sub-chunks with deletions with erasures and using the linear code to recover the erasures. The final step is to check all the constraints for the remaining candidates on the list and discard any inconsistent candidate.

One of regimes where this construction (more than two layers of VT constraints) can be useful is when there are few deletions in a very long sequence, in this regime by having more than two layers we can split the sequence into small enough chunks more efficiently. Further investigation in this direction is for future work.

The representation in Figure 4.7, motivates the generalization of the code using factor graphs.

Factor graph construction

In general, one can divide the sequence into some chunks of length n_c, and then choose l different arbitrary subsets of chunks like $\mathcal{C}_1, \cdots, \mathcal{C}_l$. Now l subsequences can be formed

by concatenating the chunks within each subset (with the order they appear in X), and the message M will be the VT syndrome of these l sequences, M can be appended with an erasure code. Note that the multilayer code with two layers is a special case of this code when $l = l_1 + l_2$. Also, the three layers code is a special case when $l = l_1 + l_2 + l_3$. The decoding algorithm in Section 4.3 cannot be used for this code as it heavily depends on the the way that VT constraints are formed, for example, the fact that blocks are concatenation of consecutive chunks is important to the decoding process described in Section 4.3. However, a decoder similar to the hash decoder explained in Section 4.7.1 can be used for this encoder. The decoder can guess the number of deletions in each of the chunks and then form the constraints accordingly. Then it can attempt to decode by iteratively recover single deletions connected to a check node. This is similar to iterative decoding of erasures with a linear code. Investigating decoders for multilayer codes designed via a general factor graph construction is an interesting direction for future work.

4.10.3 Discussion

In this work we introduced a new method for one-way synchronization problem based on using multiple VT constraints concatenated with linear codes, and a list decoder. We showed that VT constraints can serve for both recovering multiple deletions and also reducing the complexity of the decoding algorithm. Thanks to the efficient decoding algorithm we are able to recover significant number of deletions (e.g. 10 deletions) in relatively long codes (few thousands). There are several direction for the future work, for example, it is interesting to investigate how multilayer codes perform in a two-way synchronization model. In a two-way model, the decoder may be allowed to request a small amount of extra information to disambiguate a list with multiple reconstructed sequences. Another direction is that the decoder can ask for additional information in complex cases when decoding is time consuming, for instance when there are atypical number of candidates after the first or third step of the decoding.

4.11 Proofs

4.11.1 Proof of Lemma 4.1

Proof. Define the function $p(x) = \binom{x+l_2-1}{l_2-1}$ (p is a polynomial of degree (l_2-1)). We first show that $p(x)p(y) \leq p(\frac{x+y}{2})^2$ for any two real positive numbers x, y. To show

this, we need to prove

$$\prod_{i=1}^{l_2-1}(y+i)(x+i) \leq \prod_{i=1}^{l_2-1}(\frac{x+y}{2}+i)^2, \tag{4.11.1}$$

which clearly follows from $xy \leq (\frac{x+y}{2})^2$. Now if we define $g(x) \triangleq \ln(p(x))$, we have $g(x)+g(y) \leq 2g(\frac{x+y}{2})$ which means g is mid-point concave, and since it is continuous, it is generally concave. Hence, we have $g(x_1)+g(x_2)+\cdots+g(x_n) \leq ng(\sum_{i=1}^n x_i/n)$ for any integer n and positive x_i's. Therefore, we have

$$p(x_1)p(x_2)\cdots p(x_n) \leq p\left(\frac{\sum_{i=1}^n x_i}{n}\right)^n. \tag{4.11.2}$$

Choosing $n = s$ and $x_i = a_i$ will prove the result. □

4.11.2 Derivative of (4.5.7)

Here we show that the derivative of $f(s) = \binom{k/s+l_2-1}{l_2-1}^s$ with respect to s is positive for $s > 0$. Define $g(s) = \ln(f(s))$, then we have:

$$\frac{dg}{ds} = \frac{1}{f(s)}f'(s) \tag{4.11.3}$$

Also, define

$$h(s) = \binom{k/s+l_2-1}{l_2-1} \tag{4.11.4}$$

We get

$$\frac{dg}{ds} = \ln(h(s)) + \frac{s}{h(s)}h'(s) \tag{4.11.5}$$

Therefore, to prove $df/ds > 0$ for $s > 0$ we have to show that

$$\ln(h(s)) + \frac{s}{h(s)}h'(s) > 0 \tag{4.11.6}$$

Recall the definition in (4.5.6), $h(s)$ can be shown as $p(s^{-1})/(l_2-1)!$, where $p(s)$ is a polynomial of degree (l_2-1).

$$p(s) = (ks+1)(ks+2)\cdots(ks+l_2-1) \tag{4.11.7}$$

Rewrite (4.11.6)

$$\ln(h(s)) > \frac{-s}{p(s^{-1})}\left(p(s^{-1})\right)' \tag{4.11.8}$$

$$\ln(h(s)) > \frac{1}{sp(s^{-1})}p'(s^{-1}) \tag{4.11.9}$$

We prove (4.11.9) by induction on $l_2 - 1$. For $l_2 - 1 = 1$, we get

$$\ln(ks^{-1}+1) > \frac{k}{(k+s)} \tag{4.11.10}$$

If we denote ks^{-1} by x, then we can rewrite (4.11.10) as

$$\ln(x+1) > \frac{x}{(x+1)}. \tag{4.11.11}$$

Which holds for all $x > 0$. Assuming that (4.11.9) holds for $w = l_2 - 1$, we prove it for $w+1$.

$$\ln(h(s)) = \left(\ln(ks^{-1}+w+1) - \ln(w+1)\right) + \sum_{i=1}^{w}(\ln(ks^{-1}+i) - \ln(i)) \tag{4.11.12}$$

$$> \left(\ln(ks^{-1}+w+1) - \ln(w+1)\right) + \frac{1}{s\prod_{i=1}^{w}(ks^{-1}+i)}\left(\prod_{i=1}^{w}(ks^{-1}+i)\right)' \tag{4.11.13}$$

Where the last inequality holds by using the hypothesis of induction. Note that we have:

$$p'(s^{-1}) = k\left(\prod_{i=1}^{w}(ks^{-1}+i)\right) + (ks^{-1}+w+1)\left(\prod_{i=1}^{w}(ks^{-1}+i)\right)' \tag{4.11.14}$$

Hence,

$$\frac{1}{sp(s^{-1})}p'(s^{-1}) = \frac{k}{k+(w+1)s} + \frac{\left(\prod_{i=1}^{w}(ks^{-1}+i)\right)'}{s\prod_{i=1}^{w}(ks^{-1}+i)} \tag{4.11.15}$$

Comparing (4.11.13) and (4.11.15) we have to prove

$$\ln\left(\frac{(ks^{-1}+w+1)}{w+1}\right) > \frac{k}{k+(w+1)s} \tag{4.11.16}$$

If we denote ks^{-1} by x, then we have to prove

$$\ln\left(\frac{(x+w+1)}{w+1}\right) > \frac{x}{x+(w+1)}. \tag{4.11.17}$$

This holds for all $x > 0$.

4.11.3 Upper bound on the binomial coefficients

Lemma 4.2. *For any real $x \geq \frac{1}{2}$ and positive integer a we have*

$$\binom{x+a}{a} \leq \min\left\{\left(\frac{e(x+a)}{a}\right)^a, \left(\frac{e(x+a)}{x}\right)^x\right\} \tag{4.11.18}$$

Proof. Recall that Gamma function is defined as follows for any complex number z,

$$\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}\,dx. \tag{4.11.19}$$

Since $\Gamma(x+1) = x\Gamma(x)$, and $\Gamma(a+1) = a!$, one can rewrite (4.5.6) as

$$\binom{x}{a} = \frac{\Gamma(x+1)}{\Gamma(a+1)\Gamma(x-a+1)}. \tag{4.11.20}$$

We will use the following inequalities for the Gamma function which can be found in [108, Thm. 5]

$$\sqrt{2e}\left(\frac{x+1/2}{e}\right)^{x+1/2} \leq \Gamma(x+1) \leq \sqrt{2\pi}\left(\frac{x+1/2}{e}\right)^{x+1/2}. \tag{4.11.21}$$

Using (4.11.20) and (4.11.21) we have

$$\binom{x+a}{a} = \frac{\Gamma(x+a+1)}{\Gamma(a+1)\Gamma(x+1)} \tag{4.11.22}$$

$$\leq \frac{\sqrt{2\pi}\left(\frac{x+a+1/2}{e}\right)^{x+a+1/2}}{\sqrt{2e}\left(\frac{a+1/2}{e}\right)^{a+1/2}\sqrt{2e}\left(\frac{x+1/2}{e}\right)^{x+1/2}} \tag{4.11.23}$$

$$= \sqrt{\frac{\pi}{2e}}\,\frac{(x+a+1/2)^{x+a+1/2}}{(a+1/2)^{a+1/2}(x+1/2)^{x+1/2}} \tag{4.11.24}$$

$$\leq \sqrt{\frac{1}{a+1/2}}\,\frac{(x+a+1/2)^{a}}{(a+1/2)^{a}}\,\frac{(x+a+1/2)^{x+1/2}}{(x+1/2)^{x+1/2}} \tag{4.11.25}$$

$$\leq \left(\frac{x+a}{a}\right)^{a}\,\frac{(x+1/2+a)^{x+1/2}}{(x+1/2)^{x+1/2}} \tag{4.11.26}$$

$$= \left(\frac{x+a}{a}\right)^{a}\,\frac{(x'+a)^{x'}}{(x')^{x'}} \tag{4.11.27}$$

$$\leq \left(\frac{x+a}{a}\right)^{a}\,e^{a} \tag{4.11.28}$$

Where in (4.11.27) $x' = x+1/2$ and in (4.11.28) we used the fact that $\ln(1+y) \leq y$ for positive y. The other inequality in (4.11.18) can be proved similarly, noting that $\left(\frac{x+(a+1/2)}{a+1/2}\right)^{a+1/2} \leq e^{x}$. □

Chapter 5

Deletion Channels with Multiple Traces

5.1 Introduction

In this chapter we consider the problem of coding for multiple independent deletion channels. Each output sequence (known as a *traces*) is produced by deleting at most k symbols from the length n input sequence. We assume that the positions of the deletions are chosen uniformly and independently. As we discussed in the first chapter, the problem of recovering coded information from multiple traces is relevant in DNA-based storage systems [11, 109]. While retrieving information by sequencing stored DNA, each trace may contain errors that are a combination of deletions, insertions, and substitutions. In this chapter, using the stylized model of a channel that introduces only deletions, we aim to understand the coding benefits obtained by having multiple traces. In particular, it is shown how one can use VT codes to achieve small probability of error for multiple deletions under suitable assumptions.

Here we assume that there are t traces available to the decoder. Each of the traces undergoes at most k deletions where k is a fixed number. The positions of these deletions are uniformly distributed. Also we assume that the deletion channels are independent which means the deletion patterns of different traces are independent. In particular, there is a chance that some of the t traces be equal. Here is an example of the channel. Assume that we have two traces, $t = 2$, and k, maximum number of deletions is also 2. If $X = 000111$ be the transmitted codeword, the decoder may receive $Y_1 = 00111$ and $Y_2 = 0011$. The goal is to design a codebook $\mathcal{C}$ which X belongs to, such that the decoder can reconstruct X with high probability when Y_1 and Y_2 are received.

5.1.1 Related works

Trace reconstruction is a well-studied problem relevant to this coding problem. In trace reconstruction, the assumption is that there are several traces available where each of them is an edited version of a target sequence. The task is to reconstruct the target sequence using the available traces. Several algorithms are suggested for recovering the sequence from its traces, e.g. see [55, 51, 53]. In all of these works, the assumption is that the target sequence is an i.i.d sequence. In contrast, in our model the target sequence is drawn from a predefined set, i.e. a codebook which can be designed. As stated in the first chapter, the motivation for this model is DNA storage system. In DNA sequencing, it is possible to have several erroneous reads of the DNA sequence, one can think of each of these reads as one of the traces. Nanopore sequencer motivated a new line of research in this area, see [110]. In DNA-based data storage systems, we embed data into the DNA sequence by synthesizing a desired DNA string and we read the data back by sequencing the DNA. As we are in control of the synthesizing, it can be assumed that the sequence belongs to a codebook that we design. Very recently in [58, 111] a relevant problem is studied. There, a codebook is designed for a version of k-deck problem. In k-deck problem, the goal is to reconstruct a sequence X of length n, when all subsequences of X of length k are known. Also, recently in [59] the multiple deletion channel is studied and a characterization for the capacity is given when the probability of deletion goes to zero.

5.1.2 Graph model

Recall the edit graph model in Chapter 2, where there is a vertex corresponding to every length n sequence, and there is an edge between two vertices like X and Y if $|\mathcal{D}_k(X) \cap \mathcal{D}_k(Y)| \geq 1$. Here we want to define a weighted version of confusability graph. If $\mathcal{X}$ is the alphabet set, the vertices in $\mathcal{G}$ correspond to elements of $\mathcal{X}^n$. Two vertices of $\mathcal{G}$, like X and Y, are connected with an edge of weight t, if $t = |\mathcal{D}_k(X) \cap \mathcal{D}_k(Y)|$. We call this graph, *weighted deletion graph*. Similarly one can define weighted insertion graph. In [50], Levenshtein found the greatest weight in both of these graphs, i.e., he found the minimum number of traces required for guaranteed reconstruction for both deletion and insertion graphs (in the graph related models the assumption is that traces are different). In [56], the authors found the greatest weight among a subset of vertices in the deletion graph. The subset was defined by codewords of a single-deletion-correcting code. In [57], the greatest weight is found for the insertion graph in a subset of sequences. In [112] the number of required traces for exact reconstruction

is studied when there are different types of traces, i.e., number of edits in the traces are different.

Another approach is to view the problem as a traditional code design problem, where one wants to find a set of codewords for reliable communication. Instead of finding the greatest weight in the graph or a subset of it, for a fixed t, one can find a subset of the graph (a codebook) where there is no edge in that subset with a weight greater or equal to t. Such a subset is a codebook capable of recovering k deletions when t (distinct) traces are available to the decoder. This is the zero error version of the problem that we are studying in this chapter and to the best of our knowledge is not studied yet.

5.1.3 Overview of the coding scheme

Our code construction is based on VT codes. Each codeword is a concatenation of blocks, with each block drawn from a predetermined VT codebook (the syndrome of the VT code in each block can be chosen arbitrary). Since our codes are based on VT codebooks, they can be constructed for any finite alphabet $q \geq 2$.

We illustrate the idea using the following binary example, which shows a codeword X of length 15 with three blocks, each of which is a sequence from a length 5 binary VT code. The channel produces two traces, Y_1 and Y_2, by deleting the underlined bits:

$$1\underline{0}001\ \ 110\underline{1}1\ \ 01010 \longrightarrow Y_1 = 1001110101010$$
$$10001\ \ 1\underline{10}11\ \ 0101\underline{0} \longrightarrow Y_2 = 100011110101$$

The decoder operates in two phases. In phase 1, it identifies blocks that are deletion-free in at least one of the traces. Each block for which a deletion-free copy is identified in one of the traces is recovered by inserting the required bits in the other traces. In the example above, block 1 has no deletions in Y_2, so Y_2 is used to correct the first block of Y_1; similarly, block 3 has no deletions in Y_1. Assuming that there were no errors, at the end of this phase the decoder has corrected all blocks which are deletion-free in at least one trace. We call the remaining blocks 'congested'. In the example, block 2 is congested as both traces have bits deleted in this block.

In phase 2, the decoder attempts to correct the congested blocks, i.e., blocks for which no clean copy was found in phase 1. In the example, since Y_1 has one deletion in block 2, the block can be corrected using the VT decoder. Since blocks 1 and 3 were corrected in phase 1, the entire codeword is recovered.

However, decoding errors may occur in either phase. In phase 1, we may wrongly identify a block as deletion-free in a trace, which leads to errors in the starting positions of other blocks. In phase 2, a congested block (or set of consecutive congested blocks) may not be correctable with the VT code, because of too many deletions in each trace. In phase 1, wrongly identifying a trace as having a deletion-free copy of a block will lead to an unusually large number of insertions when correcting the other traces using this copy. This can be used to discard accidental matches in phase 1. We show via numerical simulations that the probability of phase 1 error decreases with n_b, the length of each block. For the phase 2 error, under the assumption that the locations of the deletions within each trace are uniformly random, we obtain a bound that decreases exponentially with the number of traces. We provide simulation results which confirm that when n_b is large and the number of traces is small, decoding errors are mostly due to phase 2 errors (unresolvable congestion). On the other hand, as t grows with n_b fixed, phase 1 errors are the dominant source of decoding errors.

The rate of the code is equal to the rate of a VT code of length n_b, which is close to $\log q - \log n_b/n_b$, where $q \geq 2$ is the alphabet size. (The precise values are given in Section 5.2.) The decoding complexity is $O(t^2k^2n)$. Therefore the proposed scheme offers an explicit, efficient technique for recovering from deletions using multiple traces. Due to its low-complexity, it can be well suited for a variety of applications, particularly in DNA-based storage.

The remainder of this chapter is structured as follows. In Section 5.2 we explain the code construction for both binary and non-binary cases. Then in Section 5.3, we introduce the decoding algorithm and its complexity. In Section 5.4, we provide the simulation results and also an upperbound for one of the error events. In the last section, alternative encoding and decoding schemes are introduced that can reduce error probability at the expense of higher complexity or lower rate.

5.2 Code construction

Codewords of length n are constructed by concatenating l blocks of VT codewords from the relevant alphabet. Each block has length n_b (therefore $n = ln_b$).

Binary code: Each block i ($1 \leq i \leq l$) is a binary VT codeword with a predetermined VT syndrome a_i, known to both the encoder and the decoder (a_i's can be chosen arbitrarily). To encode each block, one can use the systematic VT encoder in [82] that maps $n_b - \lceil \log(n_b+1) \rceil$ bits to a length n_b VT sequence with the desired syndrome.

The rate of the code will be

$$R = 1 - \frac{1}{n_b}\lceil \log(n_b+1) \rceil. \tag{5.2.1}$$

Non-binary code: The code construction is very similar to the binary case. Each block is encoded separately, and belongs to a known non-binary VT class, as defined in (2.2.25). The systematic encoder for non-binary VT codes which is discussed in Chapter 2 can be used for the mapping of messages bits to each of the blocks. Alternatively, one can use a look up table for the mapping. There are qn_b non-binary VT classes, so there exists a class with at least $\frac{q^{n_b}}{qn_b}$ sequences. Using this class for encoding each block induces the following lower bound for the rate of the code

$$R \geq \log q - \frac{1}{n_b}\log n_b - \frac{1}{n_b}\log q. \tag{5.2.2}$$

5.3 Decoding

The goal of the decoder is to reconstruct X using the traces $Y_1, Y_2, \cdots, Y_t$, each obtained by deleting up to k symbols from X. We next describe the binary decoder, and then outline the main differences for the non-binary case. We first explain the main ideas using examples, and then specify the decoding algorithm in detail.

5.3.1 Phase 1

Consider block i of the codeword, for $1 \leq i \leq l$. If the starting position of block i within each trace is known, then the decoder can compute the VT syndrome of the length n_b sequence from the starting position, for each trace. If there is a trace for which the computed syndrome matches with a_i (the correct syndrome for block i), then the trace can be used to correct this block within other traces. The following example illustrates this idea.

Example 5.1. *Let the number of blocks be $l = 3$, with each block of length $n_b = 5$. Thus $n = 15$. Let*

$$X = 01001\ \ 11001\ \ 11111$$

be the transmitted codeword. The VT syndromes of the blocks are $a_1 = 1$, $a_2 = 2$, and $a_3 = 3$. Suppose that the decoder receives two traces, each with $k = 2$ deletions. The

underlined bits are deleted from X to produce Y_1 and Y_2:

$$01\underline{00}1\ \ 11001\ \ 11111 \longrightarrow Y_1 = 0111100111111$$
$$01001\ \ \underline{1}100\underline{1}\ \ 11111 \longrightarrow Y_2 = 0100110011111$$

The decoder first computes the VT syndromes of $Y_1(1:5)$ and $Y_2(1:5)$. We have

$$\mathsf{syn}(Y_1(1:5)) = 2 \quad \textit{and} \quad \mathsf{syn}(Y_2(1:5)) = 1.$$

Since $a_1 = 1$, the decoder assumes that $Y_2(1:5)$ is the first block of X, and uses it to correct the first block of Y_1 by inserting the two missing bits. The decoder then considers the second block, whose starting position is now known for each trace. Finding that $\mathsf{syn}(Y_1(4:8)) = a_2 = 2$, it assumes this sequence is the second block of X, and uses it to correct Y_2 by inserting two bits. Since there are no deletions in the third block, the decoder finds $\mathsf{syn}(Y_1(9:13)) = \mathsf{syn}(Y_2(9:13)) = a_3 = 3$, and stops.

A decoding error may occur if the VT syndrome of a block in a trace accidentally matches the correct value. In this case, an incorrect sequence will be used to correct the block in all other traces, potentially introducing multiple errors. The following example shows that how other traces can help to identify and discard such accidental matches.

Example 5.2. *Let the number of blocks be $l = 2$, with each block of length $n_b = 10$. The VT syndromes of the blocks are $a_1 = a_2 = 5$. Let the transmitted codeword be*

$$X = 1000111100\ \ 0011101100.$$

There are two traces, each with $k = 2$ deletions (underlined bits are deleted):

$$100\underline{0}111\underline{1}00\ \ 0011011100 \longrightarrow Y_1 = 100111000011011100$$
$$\underline{1}000111100\ \ 001\underline{1}011100 \longrightarrow Y_2 = 000111100001011100$$

The decoder finds that $\mathsf{syn}(Y_1(1:10)) = 5$ (this is an instance of an accidental match), and $\mathsf{syn}(Y_2(1:10)) = 0$. It assumes that $Y_1(1:10)$ is the correct block, and uses it to fix Y_2. It does this by comparing $Y_1(1:10)$ with Y_2, and inserting the required bits to get $\hat{Y}_2(1:10) = \underline{1}00\underline{11}1\underline{0}\underline{000}$. Since there are two deletions in Y_2, exactly two inserted bits are required to recover the codeword. However, since 7 bits need to be inserted into Y_2 to get $Y_1(1:10)$ and $k = 2$, the decoder realizes that $Y_1(1:10)$ is an accidental match.

The above example shows that when an accidentally matched block is used as the model to correct other traces, the number of inserted bits is likely to be large. Hence the decoder can distinguish between an accidental match and a correct match in most cases.

Congested blocks and resynchronization. There may be blocks that have undergone at least one deletion in each of the traces. These blocks are called *congested*, as a correct match for them cannot be found in any of the traces. In Example 5.2, the first block is congested as there are deleted bits in both of the traces. As the decoder proceeds from left to right in phase 1, it needs to resynchronize whenever it identifies a congested block. It does so by testing all possible starting positions for the next block in each trace.

Assume block i is congested, and consider a trace for which the decoder has inferred that there are $d < k$ deletions up to block $(i-1)$. The decoder needs to test $(k-d)$ possible starting positions for block $(i+1)$ in this trace. It computes the VT syndromes of the length n_b sequences starting from each of these positions, and checks for a match with the correct syndrome a_{i+1}. If a match is found, it is used to correct the other traces. It repeats this process for each trace, testing all possible starting positions for block $(i+1)$, and checking whether a match is found for the correct VT syndrome. If the decoder finds one or more syndrome matches among those tested, it chooses the one that requires the minimum number of insertions (across all traces) for correcting block $(i+1)$.

When block i is identified as congested, it is possible that block $(i+1)$ is also congested (i.e., has deletions in all the traces). In this case, no matches may be found among all the tested starting positions for block $(i+1)$. The decoder then tries to synchronize by testing all possible starting positions for block $(i+2)$.

5.3.2 Phase 2

At the end of phase 1, if there are no errors, the decoder has corrected all the blocks for which there is at least one trace with a deletion-free copy of the block. Each remaining block is congested, and is either: i) isolated, i.e., the bits corresponding to the block in each trace are known, or ii) part of an isolated set of consecutive congested blocks.

In the second phase, the decoder uses the VT syndromes to correct as many congested blocks as possible. For each congested set of r consecutive blocks ($r \geq 1$, with $r = 1$ corresponding to a single congested block), the decoder can infer the number of deleted bits within each trace. It uses this information, and attempts to correct the congested blocks as follows. For a congested set of r consecutive blocks ($r \geq 1$),

the decoder looks for a trace with exactly r deletions. If such a trace exists, then this set of blocks can be corrected using that trace and the known VT syndromes of the r blocks. On the other hand, a congested set of r consecutive blocks cannot be corrected if it has at least $(r+1)$ deletions in each trace.

Example 5.3. *Let the number of blocks be $l = 4$, each with length $n_b = 5$. The VT syndromes of all blocks are $a_1 = a_2 = a_3 = a_4 = 0$. Let the transmitted codeword be*

$$X = 11100\ 10001\ 10001\ 01010.$$

There are two traces, with 4 deletions in the first trace and 3 in the second:

$$\begin{aligned} &1110\underline{0}\ 1\underline{0}001\ 10001\ \underline{0}101\underline{0} \rightarrow Y_1 = 1110100110001101 \\ &11100\ 100\underline{0}1\ 10001\ 0\underline{1}0\underline{1}0 \rightarrow Y_2 = 11100100110001000. \end{aligned}$$

The first block is recovered using Y_2, using which the block is corrected in Y_1. The second block is congested, and neither trace provides a match for its VT syndrome. The decoder therefore tests the possible starting positions for the third block. Consider the first trace, which has a total of 4 deletions. Since there was one deletion in the first block, there are three possible starting positions for the third block: bits $7, 8$ and 9 of Y_1. Similarly, bits $8, 9$ and 10 of Y_2 are the possible starting positions for the third block.

The decoder therefore computes the VT syndrome of $Y_1(9:13), Y_1(8:12), Y_1(7:12)$, and $Y_2(10:14)$, $Y_2(9:13)$, $Y_2(8:12)$. Among these, the only one that satisfies the correct syndrome $a_3 = 0$ is $Y_1(9:13) = Y_2(10:14) = 10001$. This indicates that there is one deletion in the second block, in each of the traces. Thus the second block can be recovered using the VT decoder in phase 2. With the first three blocks synchronized, the decoder attempts to correct the fourth. The fourth block has two deletions in both traces. As the VT decoder can only correct a single deletion, the decoder declares an error due to an unresolvable congestion.

In the next section (Propositon 5.1), we derive a bound on the probability of phase 2 error, caused by an unresolvable congestion like the one above.

5.3.3 Decoding algorithm (for binary alphabet)

We now describe the decoder in detail. Denote the number of deletions in the jth trace by k_j, recalling that $k_j \leq k$ for $1 \leq j \leq t$.

<u>*Phase 1*</u>

Block 1: Compute the VT syndrome of $Y_j(1:n_b)$, for $1 \leq j \leq t$. If the computed syndrome for trace j is equal to a_1, consider $Y_j(1:n_b)$ as a candidate for the first block of the codeword, and use it to correct the other traces. In the process, if the total number of bits inserted into any trace exceeds the number of deletions in it, discard Y_j from the list of candidates. If the final list of candidates is non-empty, pick one that leads to the fewest total insertions in the other traces. If the final list of candidates is empty, declare block 1 congested and proceed to the second block.

Block $i > 1$: There are two possibilities:

1. If block $(i-1)$ is not congested: The starting position of the ith block is known in each trace. As in block 1, for each trace compute the VT syndrome for the length n_b sequence from the starting position, and compare with a_i. Each sequence whose VT syndrome matches a_i is a candidate. Use each candidate sequence to correct the other traces; if the total number of bits inserted in any trace (up to this point in decoding) exceeds the number of deletions in it, discard the sequence from the list of candidates. If the final list of candidates is non-empty, pick one that leads to the fewest total insertions in the other traces. If the final list list of candidates is empty, declare block i congested, and proceed to the next block.

2. If block $(i-1)$ is congested: The starting position of block i is not known. Suppose that blocks $(i-1)$ to $(i-c)$ are congested (where $c \geq 1$). Since block $(i-c-1)$ is not congested, for each trace the decoder can infer the total number of deletions up to block $(i-c-1)$. Denote this number by d_j for trace j. Then the starting position of the block i in trace j is a number between $(i-1)n_b - c - d_j + 1$ and $(i-1)n_b - k_j + 1$, where k_j is the total number of deletions in trace j. Compute the VT syndrome for each of these $(k_j - c - d_j + 1)$ possibilities, and compare with a_i. If there is a sequence whose syndrome matches, add it to the list of candidates and correct the other traces using this sequence. Since the starting position of block i is not known, when correcting using a candidate sequence, we need to consider all the possible starting positions of block i in the other traces. Pick the starting position that results in the minimum number of inserted bits. (If there is more than one starting position that gives the minimum, we pick the rightmost one.) As before, discard a candidate if the number of bits inserted in trace j is larger than $k_j - c - d_j$ for some j.

   If the final list of candidates is non-empty, pick one that leads to the fewest total insertions in the other traces. This process also gives the starting positions for

block $(i+1)$ in each trace. If the final list of candidates is empty, declare block i congested, and proceed to the next block.

Phase 2

Consider each congested set of r consecutive blocks separately, for $1 \leq r \leq k$. For each of these congested sets, the decoder knows the number of deletions in each trace. For a congested set with r blocks, if each trace has more than r deletions in the congested set, the decoder declares an error. Otherwise the decoder finds a trace with exactly r deletions in the congested set, i.e., exactly one deletion per block. The decoder corrects these blocks using the VT decoder, and uses them to correct the other traces by inserting the appropriate bits. During this process, if the number of inserted bits does not match the number of deletions in the trace within the congested set, the decoder declares an error.

5.3.4 Non-binary alphabet

The decoding is similar to the binary case. The only difference is that the VT syndrome of a non-binary sequence is a pair of numbers. Therefore, when we comparing VT syndromes of two sequences in the first phase, both numbers in the pair should be compared. In the second phase, the decoder uses the non-binary VT decoder from [17] to recover a block with a single deletion.

5.3.5 Decoding complexity

In the first phase, for a block for which the starting position is unknown, the decoder computes at most VT syndromes of k length n_b sequences in each of the t traces. For each matched syndrome, the decoder needs to check inserted bits in at most k blocks in the other $(t-1)$ traces. Since there are l blocks, and $n = n_b l$, the complexity for the first phase is $\mathcal{O}(t^2 k^2 n)$.

In the second phase, the decoder uses the VT decoder in at most l blocks (each of length n_b), and then use the recovered sequence to correct the block in the other traces. Since the VT decoder has linear complexity, the complexity for phase 2 is $\mathcal{O}(tn)$.

5.4 Error probability and simulation results

5.4.1 Phase 1 errors

In the first phase of decoding, an error can occur in two ways. First, an accidental match may lead to a block being wrongly identified as deletion-free in a trace; this is then used to correct the block in other traces. Second, when a congested block is identified, the decoder may pick a wrong starting position for the next block. As shown in Example 5.2, an accidental match in a trace can be often detected by the decoder when it leads to a large number of inserted bits in the other traces. This detection feature makes it hard to derive a rigorous bound for the phase 1 error.

Without the detection feature, the probability of an accidental VT match in the binary case will be inversely proportional to n_b, the length of the block. Indeed, the family of (n_b+1) VT codes (in the binary case) partitions the space of length-n_b binary sequences into approximately equal-sized sets of size $\sim 2^{n_b}/(n_b+1)$ (see Section 2.2.2). Hence the probability that a binary sequence picked uniformly at random will match a given VT syndrome is close to $\frac{1}{(n_b+1)}$. (Although one should notice that an accidentally matched sequence is not a uniformly random sequence and here we merely provided a heuristic argument.)

5.4.2 Phase 2 errors

Errors in the second phase of the decoding are due to unresolvable congestion. Recall that unresolvable congestion occurs if, for some $1 \le r \le k$, there is a set of r consecutive congested blocks with at least $(r+1)$ deletions in each trace. The following proposition bounds the probability of phase 2 error, denoted by P_{e_2}.

Proposition 5.1. *Consider a code with l blocks, and a channel that introduces at most k deletions in each of the t traces. If $k < l$ and the locations of deletions within each trace are uniformly random, the probability of phase 2 error satisfies*

$$\mathsf{P}_{e_2} \le l\Big((1-p_0-p_1)^t + \Big((1-p_0)^2 - p_1p_1'\Big)^t + \sum_{r=3}^{k-1}(1-p_0)^{rt}\Big) \tag{5.4.1}$$

$$\le l\Big((1-p_0-p_1)^t + \Big((1-p_0)^2 - p_1p_1'\Big)^t + \frac{(1-p_0)^{3t}}{1-(1-p_0)^t}\Big) \tag{5.4.2}$$

where, for $s = 0, 1$,

$$p_s = \frac{\binom{(k-s)+(l-1)-1}{k-s}}{\binom{k+l-1}{k}}, \tag{5.4.3}$$

and

$$p_1' = \frac{\binom{(k-2)+(l-2)-1}{k-2}}{\binom{(k-1)+(l-1)-1}{k-1}}. \tag{5.4.4}$$

We note that the phase 2 error probability (and the bound) depends only on k and l. It does not depend on either n_b or the alphabet size. The restriction $k < l$ is natural, since otherwise the expected number of deletions per block would be greater than 1.

Proof. For $1 \le i \le l$ and $1 \le r \le k$, let $Z_{i,r}$ be an indicator random variable with $Z_{i,r} = 1$ if the ith block is in an unresolvable congestion of exactly r consecutive blocks, and $Z_{i,r} = 0$ otherwise. Let

$$Z = \sum_{r=1}^{k-1} \sum_{i=1}^{l} \frac{1}{r} Z_{i,r}. \tag{5.4.5}$$

For each r, the inner sum in (5.4.5) counts the number of distinct sets of r consecutive congested blocks. Therefore Z is the total number of distinct congested sets, where a congested set is a set of of r consecutive congested blocks, for some $r \ge 1$. Hence $\mathsf{P}_{e_2} = \mathbb{P}(Z \ge 1)$. Using Markov's inequality,

$$\mathbb{P}(Z \ge 1) \le \mathbb{E}[Z] = \sum_{i=1}^{l} \sum_{r=1}^{k-1} \frac{1}{r} \mathbb{E}[Z_{i,r}] \tag{5.4.6}$$

The probability that a given block has exactly s deletions (for $0 \le s \le k$) is given by p_s in (5.4.3). Indeed, since the locations of the k deletions are uniformly random, the probability of a block having s deletions is the proportion of non-negative integer solutions of $x_1 + x_2 + \cdots + x_l = k$ with $x_1 = s$.

A block can be in an unresolvable congestion only if it has more than one deletion in each of the traces. Therefore,

$$\mathbb{E}[Z_{i,1}] \le (1 - p_0 - p_1)^t. \tag{5.4.7}$$

To find an upperbound for $\mathbb{E}[Z_{i,r}]$ for $r \ge 2$, we need the following lemma.

Lemma 5.1. *For a given trace and r blocks ($1 \le r \le k$), denote by q_r the probability of at least one deletion occurring in each of the r blocks. Then*

$$q_r \le (1 - p_0)^r. \tag{5.4.8}$$

Proof. We prove this by using induction on r. For $r = 1$, we have $q_1 = (1 - p_0)$. Now assume that (5.4.8) holds for q_u, for some $u < r$. For $s \ge u$, we write $q_u(s)$ for the

probability of s deletions occurring in a given set of u consecutive blocks, with at least one deletion in each of them. Clearly, $q_u = \sum_{s=u}^{k} q_u(s)$. We then have

$$q_{u+1} = \sum_{s=u}^{k} q_u(s) \left(1 - \frac{\binom{(k-s)+(l-u)-2}{k-s}}{\binom{(k-s)+(l-u-1)}{k-s}}\right) \tag{5.4.9}$$

$$= \sum_{s=u}^{k} q_u(s) \left(1 - \frac{l-u-1}{(k-s)+(l-u-1)}\right) \tag{5.4.10}$$

$$\leq \sum_{s=u}^{k} q_u(s) \left(1 - \frac{l-1}{k+l-1}\right) \tag{5.4.11}$$

$$= \sum_{s=u}^{k} q_u(s)\,(1-p_0) \tag{5.4.12}$$

$$\leq (1-p_0)^{u+1}. \tag{5.4.13}$$

In the chain above, it can be verified that (5.4.11) holds when $l > k$ and $s \geq r$. Eq. (5.4.13) is obtained using the induction hypothesis: $\sum_{s=u}^{k} q_u(s) = q_u \leq (1-p_0)^u$. □

Using the lemma, the probability that two consecutive blocks, say i and $(i+1)$, have at least three deletions in each trace (and are hence unresolvable) is bounded by $[(1-p_0)^2 - p_1 p_1']^t$. Here p_1' defined in (5.4.4) is the probability of block $(i+1)$ having one deletion *given* that block i has one deletion. Therefore, considering the event of unresolvable congestion either in the pair of blocks $\{(i-1), i\}$ or in blocks $\{i, (i+1)\}$, we have

$$\mathbb{E}[Z_{i,2}] \leq 2\left((1-p_0)^2 - p_1 p_1'\right)^t. \tag{5.4.14}$$

For $r > 2$, consider a set of r consecutive blocks, say $i, \ldots, (i+r-1)$. The probability of congestion in this set of blocks is q_r^t, which by (5.4.8) is bounded by $(1-p_0)^{rt}$. Hence,

$$\mathbb{E}[Z_{i,r}] \leq r(1-p_0)^{tr}, \tag{5.4.15}$$

where we use the fact that a given block i is part of (up to) r different sets of r consecutive blocks. Using (5.4.7), (5.4.14), and (5.4.15) in (5.4.6) yields the result of the proposition. □

5.4.3 Numerical results

Figure 5.1 shows the empirical error probability of the code for different values of n_b, for $q = 2$ (binary) and for $q = 4$. Each codeword consists of $l = 7$ blocks, each of length

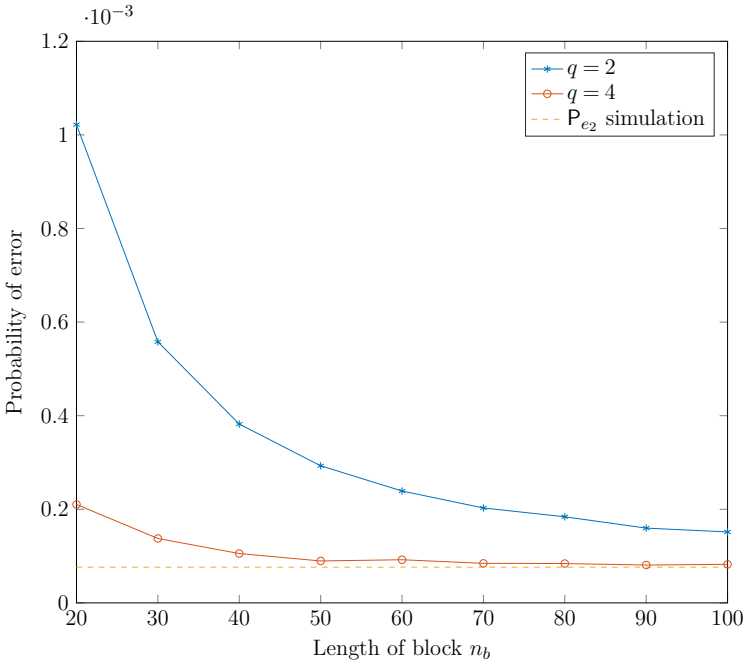

Figure 5.1: Probability of error for different block lengths when $l = 7, k = 4$, and $t = 5$.

n_b. There are $t = 5$ traces, each with $k = 4$ deletions at uniformly random locations. We note that both the code length and the rate (cf. (5.2.1), (5.2.2)) increase with n_b.

Figure 5.1 also shows the empirical phase 2 error (dashed line), which does not depend on either n_b or the alphabet. Proposition 5.1 gives an upper bound of 3.41×10^{-4} for the phase 2 error, while the empirical value is 7.63×10^{-5}. The difference between the overall and the phase 2 error probabilities can be (roughly) interpreted as the phase 1 error probability. The phase 1 error caused by wrong matches of VT syndrome decreases with n_b, as explained in Section 5.4.1. Furthermore, we observe that the phase 1 error is smaller (and decreases faster with n_b) for $q = 4$ than for $q = 2$. There are two reasons for this. First, the number of potential VT syndromes for $q > 2$ is qn, in contrast to the binary case where there are $(n+1)$ VT syndromes. Thus the probability of an accidental match is smaller for the non-binary code. Second, as q increases we expect an accidental match to produce more insertions in the other traces, making it is less likely to be accepted as the correct block. This is because matching of two symbols is less likely in a larger alphabet.

Figure 5.2 shows how the error probability decreases with the number of traces t, for a rate $\frac{5}{6}$ binary code with code parameters held fixed. Each trace has 4 deletions. As shown in Proposition 5.1, the phase 2 error decays exponentially with t. The overall error probability decays more slowly. Hence for larger values of t, phase 1 error becomes the dominant contribution to the overall error probability.

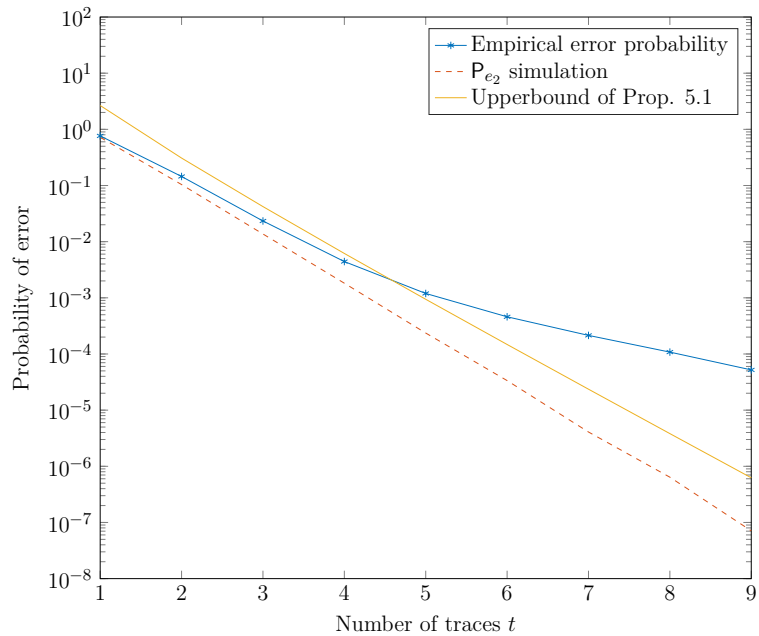

Figure 5.2: Probability of error for a binary code for different values of t when $l = 6, k = 4$, and $n_b = 30$. The code length $n = 180$, and the rate is $5/6$.

Figure 5.3 shows the probability of error for different k for a rate $\frac{5}{6}$ binary code with $t = 5$ traces. No errors were observed for $k = 2$.

5.5 List decoding approach

In this section we develop ideas to improve the error performance of the coding scheme by using more complex decoding algorithms in both phases.

Improving Phase 1: When there are multiple syndrome matches for a block, the current decoder picks the match that produces the fewest insertions in other traces. Instead, the decoder could accept up to a certain number of matches (based on the number of insertions they produce), and run the decoding procedure in parallel for each of the accepted candidates. As decoding proceeds, multiple matches may be produced within each of these parallel scenarios; the decoder uses the "fewest insertion" rule to decide which candidates to discard. The extreme version of this decoder will only discard a candidate if it is inconsistent with the received traces. This would be a list decoder that produces all the codewords that are consistent with the received traces.

Improving Phase 2: In phase 2, when there is an unresolvable congestion in a block (i.e., more than one deletion in each trace), an error is declared. Instead, the decoder could choose the trace with the minimum number of deletions in the block, guess all bits except one, and use the VT decoder to find the remaining bit. If the trace has d

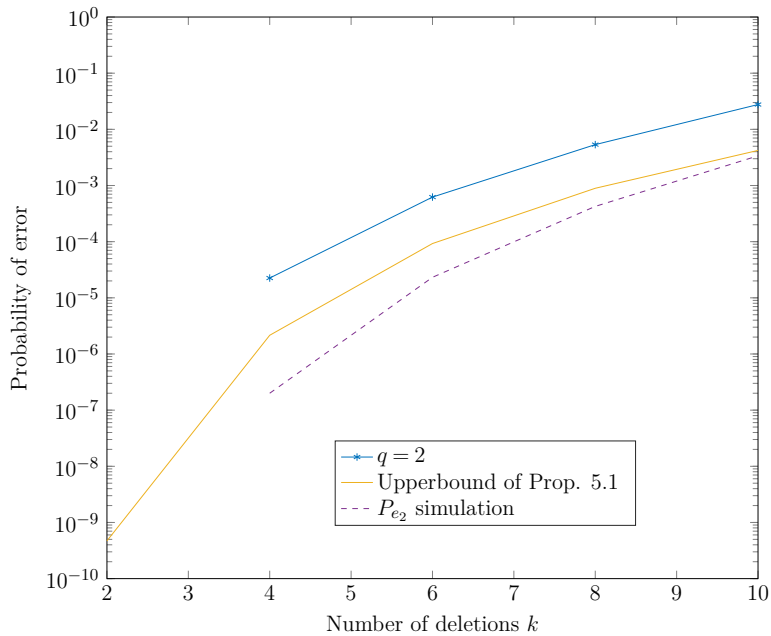

Figure 5.3: Probability of error of a binary rate 5/6 code for different values of k, the number of deletions. Code parameters are $n_b = 30$, $l = 10$, and the number of traces $t = 5$.

bits deleted in the block, there are

$$\sum_{i=1}^{d-1} \binom{n_b - 1}{i} \leq n_b^{d-1} \tag{5.5.1}$$

ways to insert $d-1$ bits into the trace. Among these, it chooses the one that produces the correct number of insertions in the other traces. Note that there might be more than one candidate which produce the correct number of insertions in other traces. In this case the decoder keeps all the compatible candidates and runs the rest of the algorithm on these candidates separately. This is in a way similar to the guess based VT decoding which we discussed in Section 4.6.

The complexity of list decoding can be an issue. For the phase 1 improvement that we discussed above, one can choose the number of parallel scenarios that the decoder considers as a tradeoff between error probability and the complexity of the decoder. For the guess-based decoder to handle unresolvable congestion in phase 2, there are different ways for adjusting the complexity. One is limiting the number guesses per block that the decoder is allowed to have. Another way that we discuss here is to only attempt to fix a congested block if there are a limited number of consecutive congested blocks. For a fixed number i, the decoder that attempts to recover a congestion of at most i consecutive blocks is called i-guess decoder. For instance, 1-guess decoder will only guess bits in a trace when there is only single congested blocks (i.e. there are

no consecutive congested blocks). The following propositions shows that the average complexity of the 1-guess decoder remains $O(t^2k^2n)$ when there are enough traces available at the decoder and also $l > k+1$. Recall that in the upper bound proposed Proposition 5.1 for phase 2 error, the first term in (5.4.1) is the dominant term. This term is the upper bound for the cases where there is a single unresolvable congested block, and should be removed for the upper bound of phase 2 errors for the 1-guess decoder.

Proposition 5.2. *The average complexity of the 1-guess decoder is $O(t^2k^2n)$ under the following conditions on the code parameters.*

$$t \geq \log n_b \tag{5.5.2}$$

$$l \geq k+2 \tag{5.5.3}$$

Proof. Phase 1 in the 1-guess decoder is identical to the normal decoder described in Section 5.3.3. Thus its complexity is at most $O(t^2k^2n)$. For the phase 2, if the decoder guesses u bits for a given block, then the complexity of recovering that block is $O(tn_b^{u+1})$. This is because the complexity of the guessing u bits is $O(n_b^u)$, for all of these guesses the decoder recover one deleted bit and compare the recovered block with other traces, the complexity of this is $O(tn_b)$.

Define $p(u)$ as the probability of an unresolvable congestion in a block where there are at least u deletions in this block across all the t traces. Recall p_i from (5.4.3), we have

$$p(u) = \left(1 - \sum_{i=0}^{u-1} p_i\right)^t. \tag{5.5.4}$$

Therefore, the average complexity of the decoding of a block is bounded by

$$\sum_{u=0}^{k} p(u)\ O(tn_b^{u+1}) \tag{5.5.5}$$

This is an upper bound since the decoder will not attempt to decode the block if it belongs to a consecutive congestion (we are not considering this in (5.5.5)). Therefore, can be upper bounded by

Using (5.4.3) we have

$$\frac{p_{i+1}}{p_i} = \frac{k-i}{k-i+l-2}, \tag{5.5.6}$$

therefore, $l \geq k+2$ will guarantee that $\frac{p_{i+1}}{p_i} \leq \frac{1}{2}$ for every $i \geq 0$. Using this in (5.5.4) we get

$$p(u) \leq \left(\frac{1}{2^u}\right)^t. \tag{5.5.7}$$

Therefore, an upper bound for the average complexity of the phase 2 is

$$\begin{aligned} l \sum_{u=0}^{k} p(u)\ O(tn_b^{u+1}) &\leq l \sum_{u=0}^{k} \left(\frac{1}{2^u}\right)^t O(tn_b^{u+1}) && (5.5.8)\\ &= \left(\frac{1}{2^t}\right)^u O(tn_b^{u+1}) && (5.5.9)\\ &\leq \left(\frac{1}{n_b}\right)^u O(tn_b^{u+1}) && (5.5.10)\\ &= O(tn). && (5.5.11) \end{aligned}$$

Here (5.5.10) follows from the assumption that $t \geq \log_2 n_b$. □

Both of the conditions in the above propositions are intuitive. The condition $l \geq k+2$ ensures less than one deletion per block on average, assuming that the locations of the deletions are uniformly random. The other constraint shows that the decoder needs more traces as the length of the block increases. This is because as n_b increases the number of guesses also increases. Finding similar constraints for general guess-based decoders is an interesting direction for future work.

5.6 Discussion

The coding scheme in this chapter demonstrates that single-deletion correcting codes can be effective in correcting multiple deletions when several traces are available. The availability of multiple traces helps the decoder in two ways: to identify deletion-free copies of a block, and to avoid mis-synchronization (by examining the number of bits inserted while correcting each trace).

In this work, each block of the codeword is chosen from a VT code which is capable of recovering one deletion. One can replace the VT code with any other deletion correcting code. For instance, if we choose blocks from codebook $\mathcal{C}$ which is capable of recovering two deletions the resulting code will have a lower phase two error in comparison with the suggested code in this work. This is because when the blocks are drawn from $\mathcal{C}$ in order to have unresolvable congestion, all the traces should have more than two deletions. It is also expected to have a lower phase 1 error: since the number of codewords in a two deletion correcting codebook is less than a VT codebook, the

probability of an accidental match is expected to be lower. Clearly the trade off is that the rate of such a code is less than VT codes.

Note the decoding algorithm suggested in this chapter can be modified to work in association with any codebook $\mathcal{C}$. The difference in the first phase of the decoding is that for VT codes, to consider a sequence as a candidate, the decoder compares its VT syndrome with the known syndrome of the block. For a general codebook $\mathcal{C}$, a sequence is considered as a candidate for a block if it simply belongs to $\mathcal{C}$. In the second phase, we need to use the decoder associated with $\mathcal{C}$ instead of using the VT decoder.

The code presented in this chapter is the first step in designing codes for multiple traces. Many of the codes that are suggested for recovering edits from one trace can potentially work with a modified decoder in our setup where multiple edited traces are available. Specifically, codes in [3, 23] are promising candidates to consider for this direction of future work.

Chapter 6

Conclusion and Future work

In this thesis we considered various models pertaining to edit channels and edit synchronization.

- In Chapter 2, we studied VT codes in detail. These codes are near optimal in terms of rate and can correct one edit with zero error probability. We also described the confusability graph for edit channels, and showed that the VT family is the only integer code that can offer a vertex coloring for the confusability graph. Furthermore, we reviewed the existing lower and upper bounds for zero error codes. One future work can be exploring if lower bounds for the zero error code based on tight upper bounds on the chromatic number of the graph can improve on the existing bounds. Another question is whether edit graphs are weakly perfect, i.e., whether clique number and chromatic number are the same for these graphs. Moreover, a new systematic encoder for non-binary codes was introduced in this chapter. Improvement of this scheme may be possible, especially in the fifth step of the encoding where we assign symbols to dyadic positions. In most of the cases, there is more than one suitable choice for these symbols. It might be possible to use this to embed more information bits. Such an improvement will also give a tighter lower bound on the size of the non-binary VT classes.

- In Chapter 3, we have considered three segmented edit channel models and proposed zero error codes for each of them over alphabets of size $q \geq 2$. The proposed codes are constructed using carefully chosen subsets of VT codes, and can be decoded in a segment-by-segment fashion in linear time. The rate scaling for the codes is shown to be the same as that of the maximal code; the upper bound of Theorem 1 shows that the rate penalty is of order $1/b$.

One direction for future work is to obtain tighter non-asymptotic upper and lower bounds on the cardinality of these codes. For tighter upper bounds, the linear programming technique from [100] is a promising approach. For tighter lower bounds, one approach would be to use the known formulas for the cardinality of VT codes (2.2.12), and adapt them to our setting where the prefix and/or suffix constraints are added. In particular, one may be able to generalize the proof given in [16, Theorem 2.2] for the exact size ov VT classes.

Also, in the three segmented models that we discussed here only one edit per segment was allowed. A natural question is how to generalize the coding scheme for models where $k > 1$ edits are allowed per segment. Assuming that there exists a codebook $\mathcal{C}$ which is a zero error k deletion correcting code, it may be possible to determine appropriate prefix and suffix for the codewords of $\mathcal{C}$ (similar to the techniques proposed for one edit per segment) to construct a zero error code for segmented model. Carefully exploring this idea would constitute another interesting line for future work.

- In Chapter 4, we introduced multilayer codes for one-way file synchronization. We have used multiple layers of VT codes, which could localize the the deletions and also recover many of them and also some linear code to recover erasures. Using the proposed structure reduces the complexity of the decoding in comparison with the method suggested in [76] as the decoder does not consider all the patterns for the deletions. We also introduced a guess-based decoding algorithm which only uses VT codes. Further studies on this decoding method is one direction to work con. For example, exploring how much adding random hashes to the multilayer construction can reduce the list size given in Table 4.6. We also gave an analysis on the list size in Section 4.7.3, improving that analysis by also considering VT constraints is another future work.

  We proposed a modified decoder to decode a combination of insertions and deletions. Implementing this decoder and evaluating its empirical performance is an important next step. In Section 4.10 we introduced the multilayer codes with more than two layers. A better understanding of this code and the tradeoffs it presents is key. For example, an interesting direction is to investigate the tradeoffs in a three layer code compared to a two layer one, with respect to rate, complexity an error probability. Adapting multilayer codes for a two-way setup is another interesting direction. For example, the decoder can ask for additional information from the encoder when there is more than one candidate on the

final list. Another idea is asking for additional information when the number of candidates in one of the decoding steps is larger than a threshold. The additional information can be used to discard some of the candidates, and this will reduce the complexity of the decoding.

The final suggestion for continuing this work is the study of burst deletion models. There are different models suggested for burst deletions (see for example [113, 114]). One interesting model considered in [114] is that all of the k deletions occur in a window of at most w bits in the input sequence. The intersecting constraints in the multilayer construction allow us to localize the deletion. If we choose $n_b \geq w$, then we know that the deletions will occur in one block or two consecutive blocks. This will simplify the decoding algorithm. Nevertheless, one may choose not to use the VT codes as the intersecting constraints. This is because in a burst deletion model, it is less likely to have a single deletion in a constraint. Therefore, we may use other random constraints (like a random binary hash) to get a better rate. Formalizing these ideas and finding the performance of a multilayer code in the burst model can be an important contribution.

- In Chapter 5, we considered the problem of coding for the deletion channel when several traces are available to the decoder. The codeword consisted of a number of blocks each of them drawn from a VT code. For each block, the decoder searches for a clean copy across the traces and use that copy to fix other traces. Multiple traces not only increase the probability of receiving a clean copy but also help the decoder to identify the clean copy when there are several candidates. List decoding was suggested as future work for improving the performance of the code in terms of error probability. In phase 1, when there are multiple syndrome matches for a block, the decoder could run the decoding procedure for a fixed number of these matches in parallel, using the "fewest insertions" rule to decide which candidates to retain. In phase 2, when an unresolvable congestion occurs, the decoder could guess a subset of the bits in the block, use the VT decoder to recover the rest, and choose a guess that produces the correct number of insertions in other traces. Implementing these methods is a very promising direction for future work. The number of required traces in order to have a low complexity of decoding and short list size is needed to be investigated.

  As suggested at the end of Chapter 5, we can also use a more sophisticated code in the blocks of the proposed construction. This will reduce the rate of the overall code but will improve the probability of error. One example of these

codes can be our multilayer code. Furthermore, in this work we only considered the deletion errors, generalizing of the method for both insertions and deletions errors is another direction for future work. For such a generalization in the first phase of the decoding instead of looking for the number of insertions the decoder needs to find the edit distance between the blocks of different traces.

Finally, finding bounds for the zero error capacity of the the trace deletion channel is an interesting combinatorial problem. For example, assume that $t = 2$ distinct traces are available to the decoder, and there are at most k deletions. The decoder may confuse two sequences if they share at least two subsequences. Therefore, two sequences X and Y in the confusability graph will be connected to each other if $|\mathcal{D}_k(X) \cap \mathcal{D}_k(Y)| \geq 2$. As discussed in Section 2.3, finding upper bounds for the degree of this graph yields a lower bound on the zero error capacity.

Small code construction

In the end, we want to highlight the importance of designing codes for a specific regime we call *small code regime*. This problem is related to the different chapters of this thesis. As mentioned earlier we may be able to use a code that can correct multiple deletions with a suitable prefix and suffix in the construction of the segmented model when multiple errors can occur in a segment. Also, such codes can be used in the construction we proposed in Chapter 5. Note that in both of these constructions the case where the length of the block (or segment) is not very long is also interesting. Say, when the length of the block (or segment) is less than 100. This is because we can concatenate different blocks to construct a code with the desired length. For example, in the code proposed in Chapter 5, we have the assumption $k \leq l$, where k is the number of deletions and l is the number of blocks. This means that $\frac{k}{n} \leq \frac{1}{n_b}$ (recall that n_b is the length of the block). Hence, if the fraction of deleted bits is more than 0.01 (which is a reasonable assumption, especially in DNA storage model), we have $n_b \leq 100$.

Motivated by this, we want to suggest a method for constructing a code capable of correcting multiple deletions in the small code regime, where the size of the codebook is not very large. For example, there are a few hundred codewords in the codebook. For this regime, finding the codebook even without an efficient encoding and decoding is acceptable. Since the number of codewords is not large, a look up table can be used for the mapping of the messages into codewords. Also for the decoding, the edit distance between the received sequence and all the codewords can be computed and the codeword with the least distance will be the output of the decoder. Notice that by

using the known bounds when the code length is not very large we expect to be in the small code regime for many of the cases. For example, from (2.3.17), for $n = 50$ and $k = 6$ the best lower bound for the size of the codebook is 217.

The proposed method has two steps. Assume that we want to construct a code that can correct k deletions. In the first step, we use a known code which can be referred as the core code to construct a set of codewords that are known to have a small error probability when used over a channel with k deletions. This can be any of the known code with small error probability, for example multilayer codes. Codes suggested in [3, 39, 41] are among other candidates for construction of the core code. Denote the codebook of the core code by $\mathcal{C}_1$. Now we construct the confusability graph $\mathcal{G}$ for this codebook. We will have a graph with $|\mathcal{C}_1|$ vertices corresponding to codewords in $\mathcal{C}_1$, and two sequences $X, Y \in \mathcal{C}_1$ are connected if $|\mathcal{D}_k(X) \cap \mathcal{D}_k(Y)| > 0$. In the second step, we can use a greedy algorithm on $\mathcal{G}$ to find an independence set. In other words, we choose the vertex in the graph with the minimum degree and add the corresponding sequence to the final codebook (if there are more than one vertex we randomly choose one of them). Then we remove the neighbours of the chosen vertex and repeat the algorithm for the rest of the graph. The resulting codebook will be zero error code capable of correcting k deletions.

The most computationally expensive part of the above algorithm is the constructing of confusability graph. Constructing this graph when there are 2^n sequences (if we do not use a core code) soon become impractical as n increases. Using a core code reduces this complexity since we need to find the edges only among $|\mathcal{C}|$ vertices. Also, when the core code $\mathcal{C}_1$ is has a good performance in terms of error probability (i.e. when codewords have large edit distance), the second step is expected to retain many of the codewords. For example, for $k = 2$ and $n = 20$, we first constructed $\mathcal{C}_1$ by enforcing the VT syndrome of $X[1:10]$ and $X[11:20]$ to be 0. Also, we split the sequence into four chunks of length 5 and enforce the summation of the chunks in $GF(2^5)$ to be zero. In the first step, $\mathcal{C}_1$ had 596 codewords and the final zero error code had 249 codewords. The lower bound from (2.3.17) for $n = 20$ and $k = 2$ is 94. Exploring which of the small error codes will perform better as a core code is a direction for future work.

Bibliography

[1] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[2] C. Shannon, "The zero error capacity of a noisy channel," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 8–19, 1956.

[3] E. A. Ratzer, "Marker codes for channels with insertions and deletions," *Annals of Telecommunications*, vol. 60, no. 1, pp. 29–44, 2005.

[4] D. Slepian and J. Wolf, "Noiseless coding of correlated information sources," *IEEE Transactions on information Theory*, vol. 19, no. 4, pp. 471–480, 1973.

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

[6] K. A. S. Immink, *Codes for mass data storage systems.* Shannon Foundation Publisher, 2004.

[7] H. Mercier, V. K. Bhargava, and V. Tarokh, "A survey of error-correcting codes for channels with symbol synchronization errors," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 1, pp. 87–96, 2010.

[8] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. LeProust, B. Sipos, and E. Birney, "Towards practical, high-capacity, low-maintenance information storage in synthesized DNA," *Nature*, vol. 494, no. 7435, p. 77, 2013.

[9] G. M. Church, Y. Gao, and S. Kosuri, "Next-generation digital information storage in DNA," *Science*, vol. 337, no. 6102, pp. 1628–1628, 2012.

[10] R. Heckel, I. Shomorony, K. Ramchandran, and N. David, "Fundamental limits of DNA storage systems," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2017, pp. 3130–3134.

[11] S. M. S. Tabatabaei Yazdi, 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 Commun.*, vol. 1, no. 3, pp. 230–248, Sep 2015.

[12] S. M. S. Tabatabaei Yazdi, R. Gabrys, and O. Milenkovic, "Portable and error-free DNA-based data storage," *Scientific reports*, vol. 7, no. 1, p. 5011, 2017.

[13] W. Mao, S. N. Diggavi, and S. Kannan, "Models and information-theoretic bounds for nanopore sequencing," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 3216–3236, 2018.

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

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

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

[17] G. Tenengolts, "Nonbinary codes, correcting single deletion or insertion," *IEEE Transactions on Information Theory*, vol. 30, no. 5, pp. 766–769, 1984.

[18] A. S. Helberg and H. C. Ferreira, "On multiple insertion/deletion correcting codes," *IEEE Transactions on Information Theory*, vol. 48, no. 1, pp. 305–308, 2002.

[19] K. A. Abdel-Ghaffar, F. Paluncic, H. C. Ferreira, and W. A. Clarke, "On Helberg's generalization of the Levenshtein code for multiple deletion/insertion error correction," *IEEE Transactions on Information Theory*, vol. 58, no. 3, pp. 1804–1808, 2012.

[20] F. Sellers, "Bit loss and gain correction code," *IRE Transactions on Information theory*, vol. 8, no. 1, pp. 35–38, 1962.

[21] V. Levenshtein, "Binary codes capable of correcting spurious insertions and deletion of ones," *Problems of information Transmission*, vol. 1, no. 1, pp. 8–17, 1965.

[22] W. Ferreira, W. A. Clarke, A. S. J. Helberg, K. A. S. Abdel-Ghaffar, and A. H. Vinck, "Insertion/deletion correction with spectral nulls," *IEEE Transactions on Information Theory*, vol. 43, no. 2, pp. 722–732, 1997.

[23] F. Wang, D. Fertonani, and T. M. Duman, "Symbol-level synchronization and ldpc code design for insertion/deletion channels," *IEEE Transactions on Communications*, vol. 59, no. 5, pp. 1287–1297, 2011.

[24] J. Brakensiek, V. Guruswami, and S. Zbarsky, "Efficient low-redundancy codes for correcting multiple deletions," *IEEE Transactions on Information Theory*, vol. 64, no. 5, pp. 3403–3410, 2018.

[25] R. Gabrys and F. Sala, "Codes correcting two deletions," *IEEE Transactions on Information Theory*, vol. 65, no. 2, pp. 965–974, 2019.

[26] J. Sima, N. Raviv, and J. Bruck, "Two deletion correcting codes from indicator vectors," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2018, pp. 421–425.

[27] Z. Liu and M. Mitzenmacher, "Codes for deletion and insertion channels with segmented errors," *IEEE Transactions on Information Theory*, vol. 56, no. 1, pp. 224–232, 2010.

[28] M. Mitzenmacher, "Capacity bounds for sticky channels," *IEEE Transactions on Information Theory*, vol. 54, no. 1, pp. 72–77, 2008.

[29] R. L. Dobrushin, "Shannon's theorems for channels with synchronization errors," *Problemy Peredachi Informatsii*, vol. 3, no. 4, pp. 18–36, 1967.

[30] S. N. Diggavi and M. Grossglauser, "On transmission over deletion channels," in *In Proceedings of the Annual Allerton Conference on Communication Control and Computing*, vol. 39, no. 1. Citeseer, 2001, pp. 573–582.

[31] Y. Kanoria and A. Montanari, "On the deletion channel with small deletion probability," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2010, pp. 1002–1006.

[32] S. Diggavi, M. Mitzenmacher, and H. D. Pfister, "Capacity upper bounds for the deletion channel," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2007, pp. 1716–1720.

[33] E. Drinea and M. Mitzenmacher, "On lower bounds for the capacity of deletion channels," *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4648–4657, 2006.

[34] D. Fertonani and T. M. Duman, "Novel bounds on the capacity of the binary deletion channel," *IEEE Transactions on Information Theory*, vol. 56, no. 6, pp. 2753–2765, 2010.

[35] E. Drinea and M. Mitzenmacher, "Improved lower bounds for the capacity of iid deletion and duplication channels," *IEEE Transactions on Information Theory*, vol. 53, no. 8, pp. 2693–2714, 2007.

[36] A. Kalai, M. Mitzenmacher, and M. Sudan, "Tight asymptotic bounds for the deletion channel with small deletion probabilities," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2010, pp. 997–1001.

[37] M. Rahmati and T. M. Duman, "Bounds on the capacity of random insertion and deletion-additive noise channels," *IEEE Transactions on Information Theory*, vol. 59, no. 9, pp. 5534–5546, 2013.

[38] D. Fertonani, T. M. Duman, and M. F. Erden, "Bounds on the capacity of channels with insertions, deletions and substitutions," *IEEE Transactions on Communications*, vol. 59, no. 1, pp. 2–6, 2010.

[39] R. Venkataramanan, S. Tatikonda, and K. Ramchandran, "Achievable rates for channels with deletions and insertions," *IEEE Transactions on Information Theory*, vol. 59, no. 11, pp. 6990–7013, 2013.

[40] M. Rahmati and T. M. Duman, "Upper bounds on the capacity of deletion channels using channel fragmentation," *IEEE Transactions on Information Theory*, vol. 61, no. 1, pp. 146–156, 2015.

[41] R. G. Gallager, "Sequential decoding for binary channels with noise and synchronization errors," MIT Lincoln Lab., Tech. Rep., Oct. 1961.

[42] L. J. Schulman and D. Zuckerman, "Asymptotically good codes correcting insertions, deletions, and transpositions," *IEEE transactions on information theory*, vol. 45, no. 7, pp. 2552–2557, 1999.

[43] M. C. Davey and D. J. C. MacKay, "Reliable communication over channels with insertions, deletions, and substitutions," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 687–698, 2001.

[44] T. A. Le and H. D. Nguyen, "New multiple insertion/deletion correcting codes for non-binary alphabets." *IEEE Transactions on Information Theory*, vol. 62, no. 5, pp. 2682–2693, 2016.

[45] V. Guruswami and C. Wang, "Deletion codes in the high-noise and high-rate regimes," *IEEE Transactions on Information Theory*, vol. 63, no. 4, pp. 1961–1970, 2017.

[46] T. G. Swart, "Coding and bounds for correcting insertion/deletion errors," Ph.D. dissertation, University of Johannesburg, 2001.

[47] K. Tian, A. Fazeli, A. Vardy, and R. Liu, "Polar codes for channels with deletions," in *55th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2017, pp. 572–579.

[48] K. Tian, A. Fazeli, and A. Vardy, "Polar coding for deletion channels: Theory and implementation," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2018, pp. 1869–1873.

[49] M. Mitzenmacher, "A survey of results for deletion channels and related synchronization channels," *Probability Surveys*, vol. 6, pp. 1–33, 2009.

[50] V. I. Levenshtein, "Efficient reconstruction of sequences," *IEEE Transactions on Information Theory*, vol. 47, no. 1, pp. 2–22, 2001.

[51] T. Holenstein, M. Mitzenmacher, R. Panigrahy, and U. Wieder, "Trace reconstruction with constant deletion probability and related results," in *In Proceedings of the ACM-SIAM SODA*, 2008.

[52] A. De, R. O'Donnell, and R. A. Servedio, "Optimal mean-based algorithms for trace reconstruction," in *In Proceedings of the STOC*, 2017.

[53] Y. Peres and A. Zhai, "Average-case reconstruction for the deletion channel: subpolynomially many traces suffice," in *In Proceedings of the IEEE Annual Symposium on Foundations of Computer Science*, 2017.

[54] N. Holden, R. Pemantle, and Y. Peres, "Subpolynomial trace reconstruction for random strings and arbitrary deletion probability," *arXiv preprint arXiv:1801.04783*, 2018.

[55] S. R. Srinivasavaradhan, M. Du, S. Diggavi, and C. Fragouli, "On maximum likelihood reconstruction over multiple deletion channels," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2018.

[56] R. Gabrys and E. Yaakobi, "Sequence reconstruction over the deletion channel," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 2924 – 2931, Apr. 2018.

[57] F. Sala, R. Gabrys, C. Schoeny, and L. Dolecek, "Exact reconstruction from insertions in synchronization codes," *IEEE Transactions on Information Theory*, vol. 63, no. 4, pp. 2428–2445, 2017.

[58] R. Gabrys and O. Milenkovic, "Unique reconstruction of coded strings from multiset substring spectra," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2018.

[59] B. Haeupler and M. Mitzenmacher, "Repeated deletion channels," in *In Proceedings of the IEEE ITW*, 2014.

[60] A. V. Evfimievski, "A probabilistic algorithm for updating files over a communication link," in *In Proceedings of the ACM-SIAM Symp. on Discrete Algorithms*, 1998, pp. 300–305.

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

[62] A. Orlitsky and K. Viswanathan, "Practical protocols for interactive communication," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2001.

[63] A. Orlitsky, "Interactive communication of balanced distributions and of correlated files," *SIAM Journal on Discrete Mathematics*, vol. 6, no. 4, pp. 548–564, 1993.

[64] S. Agarwal, V. Chauhan, and A. Trachtenberg, "Bandwidth efficient string reconciliation using puzzles," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 11, pp. 1217–1225, 2006.

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

[66] S. M. S. Tabatabaei Yazdi and L. Dolecek, "Synchronization from deletions through interactive communication," *IEEE Transactions on Information Theory*, vol. 60, no. 1, pp. 397–409, Jan. 2014.

[67] N. Ma, K. Ramchandran, and D. Tse, "Efficient file synchronization: A distributed source coding approach," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2011.

[68] L. Su and O. Milenkovic, "Synchronizing rankings via interactive communication," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2014, pp. 1056–1060.

[69] U. Irmak, S. Mihaylov, and T. Suel, "Improved single-round protocols for remote file synchronization," in *In Proceedings the 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3. IEEE, 2005, pp. 1665–1676.

[70] R. Venkataramanan, H. Zhang, and K. Ramchandran, "Interactive low-complexity codes for synchronization from deletions and insertions," in *In Proceedings of the 48th Annual Allerton Conf. on Comm., Control, and Computing*, 2010.

[71] N. Bitouzé and L. Dolecek, "Synchronization from insertions and deletions under a non-binary, non-uniform source," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2013.

[72] N. Bitouzé, F. Sala, S. M. S. Tabatabaei Yazdi, and L. Dolecek, "A practical framework for efficient file synchronization," in *In Proceedings of the 51st Annual Allerton Conf. on Comm., Control, and Computing*, 2013, pp. 1213–1220.

[73] A. Trigdell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University, February 1999.

[74] N. Ma, K. Ramchandran, and D. Tse, "Efficient file synchronization: A distributed source coding approach," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2011, pp. 583–587.

[75] Q. Wang, S. Jaggi, M. Médard, V. R. Cadambe, and M. Schwartz, "File updates under random/arbitrary insertions and deletions," *IEEE Transactions on Information Theory*, vol. 63, no. 10, pp. 6487–6513, 2017.

[76] S. K. Hanna and S. El Rouayheb, "Guess & check codes for deletions, insertions, and synchronization," *IEEE Transactions on Information Theory*, vol. 65, no. 1, pp. 3–15, 2019.

[77] M. Abroshan, R. Venkataramanan, Fabregas, and A. Guillén i Fàbregas, "Efficient systematic encoding of non-binary VT codes," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2018, pp. 91–95.

[78] M. Abroshan, R. Venkataramanan, and A. Guillén i Fàbregas, "Coding for segmented edit channels," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 3086–3098, 2017.

[79] M. Abroshan, R. Venkataramanan, and A. Guillén i Fàbregas, "Codes for channels with segmented edits," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2017.

[80] M. Abroshan, R. Venkataramanan, and A. Guillén i Fabregas, "Multilayer codes for synchronization from deletions," in *In Proceedings of the IEEE Information Theory Workshop*, 2017.

[81] M. Abroshan, R. Venkataramanan, L. Dolecek, and A. Guillén i Fàbregas. (2019) Coding for deletion channels with multiple traces. (Online) http://link.eng.cam.ac.uk/foswiki/pub/Main/RV285/trace.pdf.

[82] K. A. S. Abdel-Ghaffar and H. C. Ferreira, "Systematic encoding of the Varshamov-Tenengolts codes and the Constantin-Rao codes," *IEEE Transactions on Information Theory*, vol. 44, no. 1, pp. 340–345, Jan 1998.

[83] K. Saowapa, H. Kaneko, and E. Fujiwara, "Systematic deletion/insertion error correcting codes with random error correction capability," in *Int. Symp. Defect and Fault Tolerance in VLSI Systems*, Nov. 1999, pp. 284–292.

[84] F. Sala, C. Schoeny, N. Bitouzé, and L. Dolecek, "Synchronizing files from a large number of insertions and deletions," *IEEE Transactions on Communications*, vol. 64, no. 6, pp. 2258–2273, 2016.

[85] R. R. Varshamov, "On an arithmetic function with an application in the theory of coding (in Russian)," *Doklady Akademii Nauk SSSR*, vol. 161, no. 3, pp. 540–543, 1965.

[86] B. D. Ginzburg, "A number-theoretic function with an application in the theory of coding," *Problemy Kibernet.*, pp. 249–252, 1967.

[87] S. Martirossian, "Single error-correcting, close packed and perfect codes," in *In Proceedings of the First INTAS International Seminar on Coding Theory and Combinatorics, Thahkadzor, Armenia*, 1996.

[88] R. L. Graham, D. Knuth, and O. Patashnik, *Concrete mathematics: A foundation for computer science.* Addison-Wesley, Reading, 1994.

[89] H. AJ and H. Morita, "Codes over the ring of integers modulo m," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 81, no. 10, pp. 2013–2018, 1998.

[90] U. Tamm, "On perfect integer codes," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2005, pp. 117–120.

[91] V. I. Levenshtein, "Efficient reconstruction of sequences from their subsequences or supersequences," *Journal of Combinatorial Theory, Series A*, vol. 93, no. 2, pp. 310–332, 2001.

[92] L. Calabi and W. Hartnett, "Some general results of coding theory with applications to the study of codes for the correction of synchronization errors," *Information and Control*, vol. 15, no. 3, pp. 235–249, 1969.

[93] D. S. Hirschberg and M. Regnier, "Tight bounds on the number of string subsequences," *Journal of Discrete Algorithms*, vol. 1, no. 1, pp. 123–132, 2000.

[94] J. Gu and T. Fuja, "A generalized Gilbert-Varshamov bound derived via analysis of a code-search algorithm," *IEEE transactions on information theory*, vol. 39, no. 3, pp. 1089–1093, 1993.

[95] L. M. Tolhuizen, "The generalized Gilbert-Varshamov bound is implied by turan's theorem [code construction]," *IEEE Transactions on Information Theory*, vol. 43, no. 5, pp. 1605–1606, 1997.

[96] V. I. Levenshtein, "Bounds for deletion/insertion correcting codes," in *In Proceedings of the IEEE International Symposium on Information Theory,*, 2002, p. 370.

[97] I. Landjev and K. Haralambiev, "On multiple deletion codes," *Serdica Journal of Computing*, vol. 1, no. 1, pp. 13p–26p, 2007.

[98] T. G. Swart and H. C. Ferreira, "A note on double insertion/deletion correcting codes," *IEEE Transactions on Information Theory*, vol. 49, no. 1, pp. 269–273, 2003.

[99] L. Tolhuizen, "Upper bounds on the size of insertion/deletion-correcting codes," in *In Proceedings of the 8th Workshop on ACCT, Tsarskoe Selo, Russia*, 2002, pp. 242–246.

[100] A. A. Kulkarni and N. Kiyavash, "Nonasymptotic upper bounds for deletion correcting codes," *IEEE Transactions on Information Theory*, vol. 59, no. 8, pp. 5115–5130, Aug. 2013.

[101] A. Fazeli, A. Vardy, and E. Yaakobi, "Generalized sphere packing bound," *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2313–2334, 2015.

[102] F. Wang, T. M. Duman, and D. Aktas, "Capacity bounds and concatenated codes over segmented deletion channels," *IEEE Transactions on Communications*, vol. 61, no. 3, pp. 852–864, 2013.

[103] A. Kulkarni, "Insertion and deletion errors with a forbidden symbol," in *In Proceedings of the IEEE Information Theory Workshop*, 2014.

[104] M. Mitzenmacher and E. Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis.* Cambridge University Press, 2005.

[105] "Matlab scripts for implementing codes for segmented edit channels." Available at: https://github.com/MahedAb/Segmented_Edit_Channels.

[106] A. Wachter-Zeh, "List decoding of insertions and deletions," *IEEE Transactions on Information Theory*, vol. 64, no. 9, pp. 6297–6304, 2018.

[107] R. Roth, *Introduction to coding theory.* Cambridge University Press, 2006.

[108] N. Batir, "Inequalities for the gamma function," *Archiv der Mathematik*, vol. 91, no. 6, pp. 554–563, 2008.

[109] H. M. Kiah, G. J. Puleo, and O. Milenkovic, "Codes for DNA sequence profiles," *IEEE Transactions on Information Theory*, vol. 62, no. 6, pp. 3125–3146, 2016.

[110] W. Mao, S. Diggavi, and S. Kannan, "Models and information-theoretic bounds for nanopore sequencing," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 3216–3236, 2018.

[111] R. Gabrys and O. Milenkovic, "The hybrid k-deck problem: Reconstructing sequences from short and long traces," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2017, pp. 1306–1310.

[112] M. Horovitz and E. Yaakobi, "Reconstruction of sequences over non-identical channels," in *In Proceedings of the IEEE International Symposium on Information Theory*, 2017.

[113] C. Schoeny, A. Wachter-Zeh, R. Gabrys, and E. Yaakobi, "Codes correcting a burst of deletions or insertions," *IEEE Transactions on Information Theory*, vol. 63, no. 4, pp. 1971–1985, 2017.

[114] S. K. Hanna and S. El Rouayheb, "Correcting bursty and localized deletions using guess & check codes," in *55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2017, pp. 9–16.