5-18-2016

Decoding of non-binary multiple insertion/deletion error correcting codes

Tuan Anh Le
Rowan University

DECODING OF NON-BINARY MULTIPLE INSERTION/DELETION ERROR CORRECTING CODES

by

Tuan Anh Le

A Thesis

Submitted to the
Department of Mathematics
College of Science and Mathematics
In partial fulfillment of the requirement
For the degree of
Master of Arts in Mathematics
at
Rowan University
April 26, 2016

Thesis Chair: Hieu D. Nguyen

Dedication

This thesis is dedicated to my loving wife, Ty Nguyen, who without a doubt is the most intelligent and beautiful lady I ever met.

Acknowledgments

I wish to express my profound gratitude to Prof. Hieu D. Nguyen for his fatherly counsel both in and out of schoolwork, and for his patience, motivation, and immense knowledge. I especially thank him for inspiring me towards completing this thesis. I could not have imagined having a better mentor.

I wish to thank Prof. Umashanger Thayasivam in the Rowan Math Department immensely for also advising me in research. I also want to thank Prof. Abdulhakir Hassen for caring about my study and giving me helpful advice. I also wish to give a big "thank you" to Profs. Marcus Wright, Ming-Sun Li, Thomas Osler, and Olcay Fatma Ilicasu. My professors in the Math Department are unique people. They are the best teachers that anyone can dream of, and I want to thank them immensely for their instruction during my stay at Rowan.

Thang Nguyen and Hieu Truong have been great friends since I came to the U.S. I would like to use this opportunity to thank you both for everything that we have been through together.

I wish to also recognize my beloved family at home in Vietnam who have always been there for me. Lastly, I would like to thanks my new family here in the U.S., namely my mother-in-law and my younger brother-in-law.

I wish you all the best.

Thank you!

Abstract

Tuan Anh Le
DECODING OF NON-BINARY MULTIPLE INSERTION/DELETION ERROR CORRECTING CODES
2015-2016
Hieu D. Nguyen, Ph.D.
Master of Arts in Mathematics

Data that is either transmitted over a communication channel or stored in memory is not always completely error free. Many communication channels are subject to noise, and thus errors may occur during transmission from transmitter to receiver. For example, DRAM memory cell contents can change spuriously due to electromagnetic interference while magnetic flux can cause one or more bits to flip in magnetic storage devices. To combat these errors, codes capable of correcting insertion/deletion errors have been investigated.

Levenshtein codes are the foundation of this thesis. His codes, first constructed by Varshamov-Tenengol'ts, are capable of correcting one insertion/deletion error. Helberg codes are based on Levenshtein codes and are able to correct multiple insertion/deletion errors. Even though there are codes with better rates, Helberg codes are still important because they can be constructed number-theoretically. In addition, Helberg codes also allow us to correct errors with certainty. However, prior to this thesis, there was no known efficient algorithm to decode Helberg codes.

In this thesis, we first present an algorithm to decode deletion errors in Helberg codes as well as a way to generalize Helberg code to non-binary alphabets. Our algorithm recursively corrects one error at a time. As a result, this algorithm is much more efficient compared to exhaustive search. Secondly, we introduce a new class of non-binary codes capable of correcting multiple insertion/deletion errors by generalizing the construction for Helberg codes. Our decoding algorithm is also applicable to correcting deletion errors in these new non-binary codes as well. All the results in this thesis have been published in [9].

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

List of Tables

Chapter 1

Introduction

Research Background

"Is there anything of which one can say 'Look, this is new'?

No, it has already existed, long before our time."

Ecclesiastes 1:9-11

For over many decades, man-made satellites have been transmitting information from deep space back to earth, yet the power of their radio transmitters is only a few watts. How is it that this information can be reliably transmitted across planets, through millions of miles without being completely drowned out by noise? This was made possible through the use of error correcting codes, a branch of coding theory that concerns itself with reliably and efficiently transmitting data across noisy channels. Coding theory was born in 1945 when C. Shannon wrote his landmark paper [3] on the mathematical theory of communication. The theory of error correcting codes not only allow us to discover the universe, but also to develop technologies for data storage and media devices, such as DVD players.

The simplest method for detecting errors in data is parity-checking where extra bits are added to the source message in order to allow us to detect and correct errors. In 1947, Richard W. Hamming [18] described his original ideas on error correcting codes and constructed the Hamming code.

Reed-Solomon codes are the most widely used type of error correcting codes. They were first described in 1960 by Reed and Solomon [17]. Since then they have been applied to CD-ROMs, wireless communications, space communications, and digital television. Unlike Hamming codes which allows us to correct one bit of a time, Reed-Solomon codes can correct groups of bits. However, encoding of data is relatively straightforward in Reed-Solomon codes, but decoding is time consuming.

In 1965, Levenshtein [15] introduced an elegant decoding algorithm to decode a single insertion/deletion error in Varshamow-Tenengol'ts (VT) codes. In 1995, A. S. J. Helberg [1] generalized VT codes to construct codes capable of correcting multiple insertion/deletion errors.

Even though there are codes with better rates, Helberg codes are still important because they can be constructed number-theoretically. Other codes such as watermark codes [13] consist of a nonlinear inner code that allow for probabilistic resynchronization and allow for the correction of errors with high probability. On the other hand, Helberg codes are deterministic and allow for correction of errors with certainty.

Problem Statement

For many years since its conception in 1993, Helberg codes were known to have been the only codes constructed explicitly capable of multiple insertion/deletion error correction. Even though there are now other explicit code constructions with better code rates, Helberg codes are still important since they are a natural generalization of the celebrated Levenshtein codes. However, prior to this thesis, there was no known efficient algorithm for decoding Helberg codes.

Our recursively reduces the number of errors, it recovers errors one by one instead of figure out all of error at the same time. This unique characteristic of our algorithm allows us to work with any number of errors while the need of work linearly increases.

Main Results

Our contributions are two-fold. First, we develop a decoding algorithm for Helberg codes to correct codewords that suffer only deletions. This algorithm, which we call the Deletion Decoding Algorithm, recursively corrects one error at a time and has linear run-time. Secondly, we generalize Helberg's construction to generate a new non-binary error correcting codes. Our proof that these codes are capable of correcting multiple insertion/deletion errors

follows the one given by Abdel-Ghaffar et al. [10]. We also generalize our Deletion Decoding Algorithm to decode these generalized Helberg codes for deletion errors as well.

Structure of Thesis

Chapter 2 gives a literature review on codes that can correct substitution errors by appending parity-check bits and Levenshtein codes. It also reviews two generalization of Levenshtein codes: Helberg codes and Tenengol'ts codes. Chapter 3 describes our Deletion Decoding Algorithm, which is the heart of this thesis, for binary Helberg codes. Chapter 4 presents the construction to generalize Helberg codes to non-binary alphabets. Chapter 5 gives a more general version of the Deletion Decoding Algorithm for non-binary Helberg codes. Chapter 6 discusses and compares this algorithm to exhaustive search. This chapter also describes open problems for future research.

Chapter 2

Error Correcting Codes

Preliminaries

Communication and storage channels may cause synchronization errors due to corruption of data. Suppose a person sends a message and the intended recipient receives the message, but with errors, which we call the corrupted message. These errors may be due to noise over transmission channel. A figure describing the encoding, transmission, and decoding of such a message is given below: For example, let our original message be

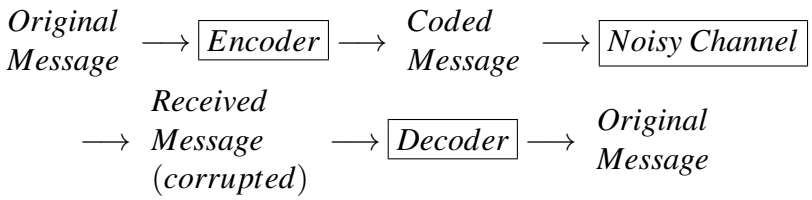

Figure 1. Transmission of Data over Noisy Channel

the sequence $(0,0,1)$ and suppose it is sent over a noisy channel where a single-bit error occurred. There are three possibilities that this could happen.

The first possibility is due to the loss of a bit which we call a deletion error. This is illustrated in Figure 2. The second possibility is due to the receipt of an extra symbol which

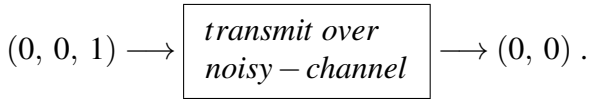

Figure 2. Deletion Error

we call an insertion error. This is illustrated in Figure 3. The third possibility is due to the

$$(0,\ 0,\ 1) \longrightarrow \boxed{\begin{array}{l} transmit\ over \\ noisy-channel \end{array}} \longrightarrow (0,\ 1,\ 0,\ 1)$$

Figure 3. Insertion Error

reversal of a bit, i.e., $0 \to 1$ or $1 \to 0$, which we call a substitution error. This is illustrated in Figure 4 In this thesis, we focus on how to correct deletion errors in a corrupted message

$$(0,\ 0,\ 1) \longrightarrow \boxed{\begin{array}{l} transmit\ over \\ noisy-channel \end{array}} \longrightarrow (0,\ 1,\ 1)$$

Figure 4. Reversal or Substitution Error

to obtain the original message.

We define a *codebook C* to be a set of sequences called *codewords* whose elements are chosen from a given alphabets of symbols. *Error detection* in a codebook is the ability to detect errors in its codewords while *error correction* in a codebook is the ability to not only detect but also correct error. Error detection always precedes error correction. A code is *d-deletion correcting* if no sequence can be obtained from more than one codeword in the codebook by deleting at most d symbols. The code is *d-deletion/insertion correcting* if no sequence can be obtained from more than one codeword in the codebook by inserting α symbols and deleting β symbols where $\alpha + \beta \leq d$.

For example, suppose we have a codebook C_1 that contains 4 codewords:

$$C_1 = \{\alpha_1,\ \alpha_2, \alpha_3,\ \alpha_4\},$$

where

$$\alpha_1 = (1,\ 0,\ 1,\ 0,\ 0,\ 0,\ 1)$$

$$\alpha_2 = (1,\ 0,\ 0,\ 1,\ 0,\ 0,\ 1)$$

$$\alpha_3 = (1,\ 1,\ 0,\ 0,\ 1,\ 0,\ 0)$$

$$\alpha_4 = (1,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0)$$

Let us assume that a codeword $\mathbf{x} \in C_1$ is transmitted and that the codeword $\mathbf{x}'$ is received where one bit has been deleted from $\mathbf{x}$:

$$\mathbf{x}' = (1,\ 0,\ 1,\ 0,\ 0,\ 1)$$

We observe that $\mathbf{x}'$ can be obtained from $\mathbf{x} = \alpha_1$ by deleting one of its bits at position 4, 5, or 6. Moreover, $\mathbf{x}'$ can also obtained from $\mathbf{x} = \alpha_2$ by deleting one of its bits at position 2 or 3. Therefore we do not know whether α_1 or α_2 was transmitted since $\mathbf{x}'$ can be obtained from more than one codeword. Thus this codebook can not correct a single deletion error.

Let us consider a second codebook C_2:

$$C_2 = \{\alpha_5,\ \alpha_6,\ \alpha_7,\ \alpha_8\}$$

where

$$\alpha_5 = (1,\ 0,\ 1,\ 1,\ 0,\ 0,\ 0)$$

$$\alpha_6 = (0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0)$$

$$\alpha_7 = (0,\ 1,\ 1,\ 1,\ 0,\ 1,\ 1)$$

$$\alpha_8 = (1,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1)$$

Again, let us assume that a codeword $\mathbf{y} \in C_2$ was transmitted and that $\mathbf{y}'$ was received where one bit has been deleted from $\mathbf{y}$:

$$\mathbf{y}' = (0,\ 0,\ 0,\ 1,\ 0,\ 0)$$

However, in this case we see that $\mathbf{y}'$ could have only been obtained from a single codeword, namely α_6 by deleting one of its bits at position 1,2,3, or 4. Thus, this codebook is able to correct a single deletion error since $\mathbf{y}'$ can only be obtained from a unique $\mathbf{y}$.

In error-correcting codes, only certain sequences are used. This is similar to having a dictionary of allowable words. If an error occurs in a codeword, we then find the most similar codeword in our codebook that matches the codeword that we received. This raises the question: "How do we construct such a codebook?". Moreover, this raises the follow question: "How do we decode to correct such errors?". Much research has been done on substitution-correcting codes. In addition, various codes have been proposed to correct insertion/deletion errors but not all of them have efficient decoding algorithms. One such family of codes, called Helberg codes and capable of correcting only insertion/deletion error, is the focus of this thesis.

Throughout this thesis, we refer to C as a codebook, q is the alphabet size, n is the length of each codeword, and d is the maximum number of errors that can occur. We define the *information rate* of the codebook C by

$$I_n(q,d) = \frac{\log_q |C|}{n}$$

where $|C|$ is the cardinality of the codebook C.

The following sections will introduce some of the oldest and simplest error-correcting codes.

Repetition codes. Repetition of each bit is a very simple approach to tackling substitution errors. By repeating each bit twice we can make our codebook detect errors. For example, instead of transmitting the codeword $(0,1,0)$, we transmit $(0,0,1,1,0,0)$. If the received codeword is $(0,1,1,1,0,0)$, then we know that an error occurred since the first zero bit was not repeated. This yields an error-detecting code, it allows one error to be detected but it is not error-correcting code because the original codeword may be been $(0,0,1,1,0,0)$

or $(1,1,1,1,0,0)$.

On the other hand, if we repeat each bit three times then the receiver can correct a single substitution error. This is illustrated in the following example.

Example 2.1. Suppose we construct a codebook that contains 4 words which gives directional commands: North (N), East (E), West (W), and South (S). These directions can be represented by the following codewords:

$$C_1 = \{(0,0),(0,1),(1,0),(1,1)\},$$

respectively. This code however is unable to detect (and therefore unable to correct) substitution errors since we can not determine whether a received codeword has any errors or not. This is because any substitution error in a codeword yields another codeword from the same codebook.

Therefore in order to be able to correct a single substitution error, we shall repeat each bit three times by converting C_1 into a new codebook C_2:

$$C_2 = \{(0,0,0,0,0,0),(0,0,0,1,1,1),(1,1,1,0,0,0),(1,1,1,1,1,1)\},$$

where each codeword corresponds to $(0,0),(0,1),(1,0),(1,1)$, respectively.

Now assume that a codeword

$$\mathbf{x}' = (0,0,1,1,1,1)$$

is received. We then know that it must come from $\mathbf{x} = (0,0,0,1,1,1)$ since $\mathbf{x}'$ differs from $\mathbf{x}$ (and only $\mathbf{x}$) in exactly one position.

The information rate of C_1 and C_2 are

$$I_3(2,0) = \frac{\log_2 |C_1|}{2} = \frac{\log_2 4}{2} = 1$$

$$I_6(2,1) = \frac{\log_2 |C_2|}{6} = \frac{\log_2 4}{6} = \frac{1}{3}.$$

Thus, error correction came at a cost of adding redundancy bits.

Parity check codes. The simplest and oldest error detection method is parity check. In communication, parity checking refers to the use of parity bits to check that data has been transmitted accurately. The parity bit is appended to every word that are transmitted. The parity bit for each word is set so that the extended word has either an even number or an odd number of one bits, which is equivalent to the sum of all the bits being even or odd, respectively. If the received codeword has parity different from the transmitted codeword then we know a transmission error occurred.

The following example illustrates how to detect a one-bit substitution error.

Example 2.2. Let us consider the scheme of even parity where we append either a 0 or a 1 bit to make the total of number of 1 bit be even:

$$\underset{\text{Data}}{(0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1)} \rightarrow \underset{\text{Data + parity}}{(0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1)}$$

$$\underset{\text{Data}}{(1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 1)} \rightarrow \underset{\text{Data+parity}}{(1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 1,\ 0)}$$

Notice that a one-bit error can occur in data bit or parity check bit.

Example 2.3. Suppose we consider the codebook C_1 from Example 2.1. Let C_3 be the codebook that is obtained from C_1 by adding a parity check bit:

$$C_3 = \{(0,0,0),(0,1,1),(1,0,1),(1,1,0)\}$$

where the third bit is added so that the sum of all bits is always even. This code can detect a bit reversal since if such an error occurred, the sum of all bits will be odd. However, this code can not correct the error since the position of the bit reversal is not uniquely determine.

For example, if we received a codeword $\mathbf{x}' = (0,1,0)$ and we know that there was exactly one substitution error, then there are two possibilities for the original codeword: $\mathbf{x} = (0,0,0)$ and $\mathbf{x} = (1,1,0)$.

The following example illustrates how we can correct one-bit substitution error:

Example 2.4. We want to continue working on the problem from previous examples: Example 2.1 and 2.3. A better approach which allows error correction is to construct a codebook C_4 from C_3 as follows:

$$C_4 = \{(0,0,0,0,0),(0,1,1,0,1),(1,0,1,1,0),(1,1,0,1,1)\}$$

where the last two bits in each word of C_4 is a repeat of the first two bits.

Clearly this code can detect a substitution error if such an error occurred, then either the sum of the first three bits is odd or the sum of the last three bits is odd. This code, however, can correct the error. For example, suppose we received a codeword $\mathbf{y}' = (0,1,1,0,0)$. We know $\mathbf{y}'$ has to come from the second codeword in C_4. This is because there is at most one codeword differing in at most one bit from any possible receivable codeword.

We calculate the information rate of C_4 as follows:

$$I_5(2,1) = \frac{\log_2 |C_4|}{5} = \frac{\log_2 4}{5} = 0.464.$$

We observe that parity check bit code C_4 has a higher information rate than the repetition code C_2.

Hamming distance. It is the number used to denote the different between two strings with equal lengths.

Definition: The Hamming distance [18] between two codewords $\mathbf{x}$ and $\mathbf{y}$ is defined to be

the number of position in which they differ in value, i.e.,

$$d_H(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|.$$

Example 2.5. Let

$$\mathbf{x} = (0,\ 0,\ 1,\ 1,\ 1)$$
$$\mathbf{y} = (1,\ 1,\ 0,\ 0,\ 1)$$

then

$$d_H(\mathbf{x},\mathbf{y}) = 4.$$

If the distance between $\mathbf{x}$ and $\mathbf{y}$ is d, then d bit-substitutions are required to convert $\mathbf{x}$ into $\mathbf{y}$. Also notice that Hamming distance $d_H(\mathbf{x},\mathbf{y})$ is a metric.

Theorem 2.6.

1. $d_H(\mathbf{x},\mathbf{y}) \geq 0$ *and* $d_H(\mathbf{x},\mathbf{y}) = 0$ *if and only if* $\mathbf{x} = \mathbf{y}$. *(Non-negativity)*

2. $d_H(\mathbf{x},\mathbf{y}) = d_H(\mathbf{y},\mathbf{x})$. *(Symmetry)*

3. $d_H(\mathbf{x},\mathbf{z}) \leq d_H(\mathbf{x},\mathbf{y}) + d_H(\mathbf{y},\mathbf{z})$. *(Triangle inequality)*

Definition: The minimum Hamming distance of a code C, denoted $d_H(C)$, is defined by

$$d_H(C) = \min_{\substack{\mathbf{x},\mathbf{y}\in C \\ \mathbf{x}\neq\mathbf{y}}} (d_H(\mathbf{x},\mathbf{y}))$$

Example 2.7. Let us consider $C = \{(0,\ 0),\ (0,\ 1),\ (1,\ 0),\ (1,\ 1)\}$. Then: $d_H(C) = 1$.

Minimum Hamming distance is a crucial property of a code, since it allows us to obtain the following simple but very important result.

Theorem 2.8. *Let C be a codebook.*

1. *If $d_H(C) \geq k+1$, then C can detect up to k substitution errors.*

2. *If $d_H(C) \geq 2k+1$, then C can correct up to k substitution errors.*

Proof. Assume $\mathbf{x} \in C$ was sent and $\mathbf{x}' \in C$ was received such that $\mathbf{x}$ is subject to at most k substitution errors, thus $d_H(\mathbf{x}, \mathbf{x}') \leq k$.

1. Since $d_H(C) \geq k+1$, it follows that $d_H((\mathbf{x}, \mathbf{x}') < d_H(C)$ which contradict to the definition of $d_H(C)$. Hence, $\mathbf{x}'$ is not a codeword of C and thus we have detected that an error occurred.

2. Let $\mathbf{y}$ be any codeword different from $\mathbf{x}$. We have:

$$\begin{aligned} d_H(\mathbf{x}', \mathbf{y}) + k &\geq d_H(\mathbf{x}', \mathbf{y}) + d_H(\mathbf{x}, \mathbf{x}') && \text{(Since } d_H(\mathbf{x}, \mathbf{x}') < k\text{)} \\ &\geq d_H(\mathbf{x}, \mathbf{y}) && \text{(Since } d_H(\mathbf{x}, \mathbf{x}') \text{ is a metric)} \\ &\geq d_H(C) \\ &\geq 2k+1. && \text{(Since } d_H(C) \geq 2k+1\text{)} \end{aligned}$$

Hence, $d_H(\mathbf{x}', \mathbf{y}) \geq k+1$. Since $d_H(\mathbf{x}, \mathbf{x}') \leq k$, this means $\mathbf{x}$ is the unique nearest neighbor of $\mathbf{x}'$, so $\mathbf{x}$ must have been the original codeword. This completes the proof.

Example 2.9. Let

$$C = \{(0,\ 0,\ 0,\ 0,\ 0),\ (1,\ 1,\ 1,\ 1,\ 1)\}$$

be the repetition code of length 5. Then

$$d_H(C) = 5.$$

By Theorem 2.8, C can detect up to 4 substitution errors and correct up to 2 substitution errors.

Hamming codes. Hamming codes are a type of error-correcting codes that use more than one parity check bit. In general, a Hamming code contains codewords of length $n = m + r$ where each codeword consists of m data bits and r parity check bits.

As the name suggests, Hamming codes have the following error correcting abilities due to Theorem 2.8: they can detect up to $d_H(C) - 1$ substitution-bit errors and can correct up to $\frac{d_H(C)-1}{2}$ substitution-bit errors.

We need to determine the number of parity bits r to append to m data bits such that the corresponding codebook has the ability to correct a single-bit substitution error. An error could occur in any of the $n = m + r$ bits of each codeword, so each codeword can be associated with n erroneous words at a Hamming distance of 1. For example: $(1,1,1)$ can become $(1,1,0)$ or $(1,0,1)$ or $(0,1,1)$ due to single bit error. Therefore we have $n+1$ possible corrections for each codeword: The valid codeword itself and the n erroneous words. Since there are 2^m data words (same as the number of codewords), the total number of possible corrections is $(n+1) \cdot 2^m$, which must be less (in terms of information theory) than the total number of words of length n. Thus we have the inequality

$$(n+1) \cdot 2^m \leq 2^n.$$

But $n = m + r$, therefore

$$(m+r+1) \cdot 2^m \leq 2^{m+r}$$

$$(m+r+1) \leq 2^r.$$

The last inequality is known as the Hamming rule. It gives a lower bound on the number of parity bits that we need in our codeword.

The information rate of a Hamming code H is defined as the proportion of useful data bits which respect to the codeword length:

$$I_n(H) = \frac{m}{n}.$$

Example 2.10. Suppose we want to calculate the information rate for a Hamming code H which encodes data generated from 8 bits.

Since there are 8 data bits, we have $m = 8$. Applying the Hamming rule, we have $(8 + r + 1) \leq 2^r$. Therefore $r = 4$. The information rate of H, refer to $Hamming(12,8)$, is

$$I_{12}(H) = \frac{8}{12} = \frac{2}{3}.$$

We see that to build a code with 8 data bits that will correct single substitution error, we need to add 4 parity check bits.

The following example will demonstrate how to generate $Hamming(12,8)$.

Example 2.11. We illustrate how to append 4 parity check bits to the data word $(1,0,1,1,1,1,0,0)$ to obtain a Hamming codeword $\mathbf{x}$ of length $n = m + r = 12$.

We number each bit position of $\mathbf{x}$ from right to left where position 1 correspond to the right most bit. Each bit position corresponding to a power of 2 will be occupied by a parity bit and all other bit positions are occupied by the data bits:

$$\mathbf{x} = (1,0,1,1,p_4,1,1,0,p_3,0,p_2,p_1)$$

Each parity bit calculates the parity for a certain subset of bits in the codeword. We first express each bit position in binary as a sum of powers of 2:

$1 = 2^0$	$5 = 2^2 + 2^0$	$9 = 2^3 + 2^0$
$2 = 2^1$	$6 = 2^2 + 2^1$	$10 = 2^3 + 2^1$
$3 = 2^1 + 2^0$	$7 = 2^2 + 2^1 + 2^0$	$11 = 2^3 + 2^1 + 2^0$
$4 = 2^2$	$8 = 2^3$	$12 = 2^3 + 2^2$

We see that bits at positions $1,2,4,$ and 8 correspond to the power of 2; therefore these

positions will be occupied by a parity bit. Our picture looks like this:

$$\begin{array}{cccccccccccc} & & & & p_4 & & & & p_3 & & p_2 & p_1 \\ \overline{12} & \overline{11} & \overline{10} & \overline{9} & 8 & \overline{7} & \overline{6} & \overline{5} & 4 & \overline{3} & 2 & 1 \end{array}$$

To determine the value of p_1 at position $1 = 2^0$, we consider the positions of those data bits whose binary expansion contains 2^0, namely $3, 5, 7, 9,$ and 11. Similarly, for p_2 at position $2 = 2^1$, we consider the positions whose binary expansion contains 2^1, namely $3, 6, 7, 10,$ and 11. This is repeated for p_3 and p_4.

To assign values to these parity bits we calculate the parity sum of its corresponding data bits as follows:

- Bit p_1 checks the position $3, 5, 7, 9,$ and 11: $p_1 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 0$
- Bit p_2 checks the position $3, 6, 7, 10,$ and 11: $p_2 = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1$
- Bit p_3 checks the position $5, 6, 7,$ and 12: $p_3 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$
- Bit p_4 checks the position $9, 10, 11,$ and 12: $p_4 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$

Our complete generated 12-bit codeword is

$$\begin{array}{cccccccccccc} 1, & 0, & 1, & 1, & 1, & 1, & 1, & 0, & 1, & 0, & 1, & 0 \\ \overline{12} & \overline{11} & \overline{10} & \overline{9} & 8 & \overline{7} & \overline{6} & \overline{5} & 4 & \overline{3} & 2 & 1 \end{array}$$

Suppose an error occurred at bit position 5:

$$\begin{array}{cccccccccccc} 1, & 0, & 1, & 1, & 1, & 1, & 1, & \mathbf{1}, & 1, & 0, & 1, & 0 \\ \overline{12} & \overline{11} & \overline{10} & \overline{9} & 8 & \overline{7} & \overline{6} & \overline{5} & 4 & \overline{3} & 2 & 1 \end{array}$$

We compare each parity bit with its parity sum:

- For bit p_1, we find its parity sum at positions$3, 5, 7, 9,$ and 11 to be $0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1$.

But this differs from p_1, which equal 0. Therefore we know an error has occurred at either p_1 or one of these positions.

- For bit p_2, we find its parity sum at positions $3, 6, 7, 10,$ and 11 to be $0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1$, which equal p_2. Therefore the bits at these positions are correct.

- For bit p_3, we find its parity sum at positions $5, 6, 7,$ and 12 to be $1 \oplus 1 \oplus 1 \oplus 1 = 0$. But this differs from p_3, which equal 0. Therefore we know an error has occurred at either p_3 or one of these positions.

- For bit p_4, we find its parity sum at positions$9, 10, 11,$ and 12 to be $1 \oplus 1 \oplus 0 \oplus 1 = 1$, which equal p_4. Therefore the bits at these positions are correct.

We now determine the position of the bit error by a process of elimination. Since the parity bit p_1 and p_3 indicate that there is an error and they both correspond to positions 5 and 7, we know that the error must be at one of these two positions. But parity bit p_2 indicates that position 7 is correct, therefore the error must have occurred at position 5.

If we now change the bit at position 5 to 0, then all parity bits agree with their parity sum and our codeword is restored.

The following table gives the number of parity bits corresponding to the number of data bits and gives the information rate of the corresponding Hamming code.

Table 1

Hamming Code Rate

Data Bits m	Parity Bits r	Total Bits n	Hamming Code	Rate $I_n(H)$
1	2	3	Hamming$(3,1)$	$1/3 \approx 0.333$
2	3	5	Hamming$(5,2)$	$2/5 = 0.4$
3	3	6	Hamming$(6,3)$	$3/6 = 0.5$
4	3	7	Hamming$(7,4)$	$4/7 \approx 0.571$
5	4	9	Hamming$(9,5)$	$5/9 \approx 0.556$
6	4	10	Hamming$(10,6)$	$6/10 = 0.6$
7	4	11	Hamming$(11,4)$	$7/11 = 0.636$
8	4	12	Hamming$(12,8)$	$8/12 \approx 0.667$
9	4	13	Hamming$(13,9)$	$9/13 \approx 0.692$
10	4	14	Hamming$(14,10)$	$10/14 \approx 0.714$

Observe that the code rate increases with the number of data bits.

Even though Hamming codes for single-bit error correction are the most commonly used type of codes, there are limitations. First, Hamming codes can only correct a single substitution error. Other codes such as Reed-Solomon codes are capable of correcting multiple substitution errors. Secondly, Hamming codes cannot correct deletion or insertion errors. Codes that can correct such errors will be discussed next.

Levenshtein Codes

Varshamov-Tenengol'ts codes. In 1965, Varshamov and Tenengol'ts [20] proposed a code construction to correct a single asymmetrical error where the probability of the bit 1 turns into 0 is considerably less than the probability that the bit 0 turns into 1, or vice versa.

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $x_i \in \{0,1\}$, denote a binary codeword. We define a family of codebooks, each denoted by $C(n,m,a)$, where n is the length of each codeword, m is the modulus, and a is the residue.

Varshamov and Tenenvol'ts required that

$$\mathbf{x} \in C(n,m,a) \Longleftrightarrow \sum_{i=1}^{n} i x_i \equiv a \pmod{m} \tag{2.1}$$

where $m \geq n+1$.

Notice that the set of all possible codewords, denoted by C, is partitioned into m different codebooks $C(n,m,a)$. We shall prove later that each codebook is capable of correcting a single insertion/deletion error.

Example 2.12. Let length $n = 4$ and $m = 5$. All 16 possible codewords of length $n = 4$ is given by

$$\begin{aligned} C = \{&(0,0,0,0),(0,1,1,0),(1,0,0,1),(1,1,1,1),(1,0,0,0),(1,1,1,0),\\ &(0,1,0,1),(0,1,0,0),(1,1,0,1),(0,0,1,1),(1,1,0,0),(0,0,1,0),\\ &(1,0,1,1),(1,0,1,0),(0,0,0,1),(0,1,1,1)\}. \end{aligned}$$

We assign these codewords to different codebooks using condition 2.1 above.

Let $\mathbf{x} = (0,0,0,0)$. Since

$$\sum_{i=1}^{4} i x_i \equiv 0 \pmod{5}$$

then $\mathbf{x} \in C(4,5,0)$.

Similarly, let $\mathbf{y} = (1,1,0,1)$. Since

$$\sum_{i=1}^{4} i y_i = (1\cdot 1 + 2\cdot 1 + 3\cdot 0 + 4\cdot 1) = 7 \equiv 2 \pmod{5}$$

then $\mathbf{y} \in C(4,5,2)$.

By repeating this for all other codewords, we obtain the following codebooks in Table 2

Table 2

Varshmov- Tenengolts Codebooks with $n = 4$

a	$C(4,5,a)$
0	{(0,0,0,0), (0,1,1,0), (1,0,0,1), (1,1,1,1)}
1	{(1,0,0,0), (1,1,1,0), (0,1,0,1)}
2	{(0,1,0,0), (1,1,0,1), (0,0,1,1)}
3	{(1,1,0,0), (0,0,1,0), (1,0,1,1)}
4	{(1,0,1,0), (0,0,0,1), (0,1,1,1)}

Observe that residue $a = 0$ gives the largest codebook.

Levenshtein distance. It is the number used to denote the different between two strings that may have different lengths.

Definition: The Levenshtein distance between two strings $\mathbf{x}$ and $\mathbf{y}$, denoted by $d_L(\mathbf{x},\mathbf{y})$, is defined as the minimum number of operations needed to transform one string (source) into the other (target), with the allowable operations being insertion, deletion, or substitution of a single character.

It is known that the Levenshtein distance is also a metric. However, while Hamming distance is the measurement between two strings of equal length , Levenshtein distance also be used for strings with different lengths. Another advantage of Levenshtein distance is that it allows us to assign costs to different operations. For example, we can define the cost of inserting a character to be twice as much as the cost of deleting a character. For simplicity, in this thesis we will assume that all costs are the same.

Example 2.13. Let

$$\mathbf{x}_1 = (0,\ 1,\ 1,\ 0,\ 1)$$

$$\mathbf{x}_2 = (0,\ 1,\ 1,\ 0,\ 1)$$

$$\mathbf{x}_3 = (1,\ 0,\ 1,\ 1,\ 0).$$

Then $d_L(\mathbf{x}_1, \mathbf{x}_2) = 0$, since $\mathbf{x}_1$ and $\mathbf{x}_2$ are identical. Moreover, $d_L(\mathbf{x}_1, \mathbf{x}_3) = 4$, since 4 bit-substitutions are required to change $\mathbf{x}_1$ into $\mathbf{x}_3$.

Levenshtein distance has many applications in spell checking, speech recognition, communications, genetics, and in plagiarism detection.

Example 2.14. Let the source string and the target string be denoted by s and t, respectively.

1. If $s =$ "*math*" and $t =$ "*math*", then $d_L(s,t) = 0$, since the two strings are identical.

2. If $s =$ "*kitten*" and $t =$ "*sitting*", then $d_L(s,t) = 3$, since the following three operations will change s into t:

   (a) "*kitten*" $\rightarrow$ "*sitten*" (substitute "k" with "*s*").

   (b) "*sitten*" $\rightarrow$ "*sittin*" (substitute "*e*" with "*i*").

   (c) "*sittin*" $\rightarrow$ "*sitting*" (insert "*g*" at last position).

Insertion/deletion correcting codes. We say that a codebook C can correct d deletions, insertions, and/or substitutions if any errorneous codeword can be obtained from no more than one codeword in C by d or fewer deletions, insertions, or substitutions. Levenshtein [15] proved the following important result, which allows us to treat any deletion error-correcting code as an insertion/deletion error-correcting codes.

Theorem 2.15. *(V. I. Levenshtein [15])*

Any code that can correct d deletions (or any code that can correct d insertions) can correct d deletions and insertions.

Proof. Assume that the same codeword **z** is obtained from a codeword **x** of length n by i_1 deletions and j_1 insertions $(i_1 + j_1 \leq d)$, and from a codeword **y** of the same length n by i_2 deletions and j_2 insertions $(i_2 + j_2 \leq d)$. If the bits that were inserted (deleted) from at least one of the codewords **x** or **y** to obtain **z** are now deleted from (inserted into) the codeword **z**, then we get a codeword that can be obtained from both **x** and **y** by no more than $max(i_2 + j_1, j_2 + i_1)$ deletions (insertions). Since **x** and **y** have the same length, $j_1 - i_1 = j_2 - i_2$ and $i_2 + j_1 = j_2 + i_1 = \frac{1}{2}(i_1 + i_2 + j_1 + j_2) \leq d$, it follows that $\mathbf{x} = \mathbf{y}$. This show there is a unique decoding for **z**, which proves the theorem.

Levenshtein codes. In 1966, Levenshtein [14] observed that Vashamov-Tenengol'ts codes could be used for correcting a single deletion or insertion error. He was able to prove this by giving the following elegant decoding algorithm.

Levenshtein one deletion decoding algorithm. Suppose a codeword $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in C(n, m, a)$ is transmitted (called the original codeword), the symbol $x_k = s$ in position k is deleted, and $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_{n-1})$ is received (called the deleted codeword). Let there be L_0 $0's$ and L_1 $1's$ to the left of s, and R_0 $0's$ and R_1 $1's$ to the right of s. Let $|x'|$ denote the number of ones in $\mathbf{x}'$and let $a' = a - \sum_{i=1}^{n-1} i x'_i \pmod{m}$, where $m \geq n+1$.

1. If $a' \leq |x'|$, then a 0 was deleted and $a' = R_1$. We insert $s = 0$ to the left of the rightmost R_1 $1's$ to restore **x**.

2. If $a' > |x'|$, then a 1 was deleted and $a' = |x'| + 1 + L_0$. We insert $s = 1$ the right of the leftmost L_0 $0's$ to restore **x**.

Proof. We calculate

$$\begin{aligned}
a' &= a - \sum_{i=1}^{n-1} ix_i' \pmod m \\
&= \sum_{i=1}^{n} ix_i - \sum_{i=1}^{n-1} ix_i' \pmod m \\
&= \sum_{i=1}^{k-1} ix_i + x_k.k + \sum_{i=k+1}^{n} ix_i - \left(\sum_{i=1}^{k-1} ix_i' + \sum_{i=k}^{n-1} ix_i' \right) \\
&= x_k \cdot k + \sum_{i=k}^{n-1} (i+1)x_{i+1} - \sum_{i=k}^{n-1} ix_i' \\
&= x_k \cdot k + \sum_{i=k}^{n-1} \left[i(x_{i+1} - x_i') + x_{i+1} \right] \\
&= x_k \cdot k + \sum_{i=k}^{n-1} x_{i+1} \\
&= x_k \cdot k + \sum_{i=k+1}^{n} x_i \\
&= x_k \cdot k + (\text{\# of 1's to the right } x_k).
\end{aligned}$$

Case I: Assume $x_k = 1$. Then

$$\begin{aligned}
a' &= x_k \cdot k + (\text{\#1's to the right}) \\
&= k + (\text{\#1's to the right}) \\
&= (k-1) + 1 + (\text{\#1's to the right}) \\
&= (\text{\#0's to the left}) + (\text{\#1's to the left}) + 1 + (\text{\#1's to the right}) \\
&= \left| x' \right| + 1 + L_0.
\end{aligned}$$

Case II: Assume $x_k = 0$. Then

$$\begin{aligned} a' &= x_k \cdot k + (\#1\text{'s to the right}) \\ &= (\#1\text{'s to the right}) \\ &= R_1. \end{aligned}$$

Example 2.16. Consider the codebook

$$\begin{aligned} C(6,7,0) = \{&(0,0,0,0,0,0),(1,1,0,1,0,0),(0,0,1,1,0,0),(0,1,0,0,1,0),(0,1,1,1,1,0), \\ &(1,0,0,0,0,1),(1,0,1,1,0,1),(1,1,0,0,1,1),(0,0,1,0,1,1),(1,1,1,1,1,1)\} \end{aligned}$$

a) Assume that we received the deleted codeword $\mathbf{x}' = (0,\ 0,\ 1,\ 0,\ 1)$.

Since the original codeword $\mathbf{x} \in C(6,7,0)$, we know that $a = 0$ and $|x'| = 2$. We have

$$\begin{aligned} a' &= a - \sum_{i=1}^{5} i x_i' \pmod 7 \\ &= 0 - 8 \pmod 7 \\ &= 6. \end{aligned}$$

Since $a' > |x'|$, this implies a 1 was deleted. Also,

$$a' = \left|x'\right| + 1 + L_0.$$

Therefore

$$\begin{aligned} L_0 &= a' - \left|x'\right| - 1 \\ &= 6 - 2 - 1 \\ &= 3. \end{aligned}$$

According to the algorithm, we insert a 1 the the right of the leftmost L_0 number of 0.

Therefore we obtain

$$\mathbf{x} = (0,\ 0,\ 1,\ 0,\ \mathbf{1},\ 1).$$

We have successfully recovered the 9^{th} codeword in $C(6,7,0)$.

b) Assume we received the deleted codeword $\mathbf{x}' = (0,\ 1,\ 0,\ 1,\ 0)$. We have $a = 0$, and $|x'| = 2$. And

$$\begin{aligned} a' &= a - \sum_{i=1}^{5} i x_i' \pmod 7 \\ &= 0 - 6 \pmod 7 \\ &= 1. \end{aligned}$$

Since $a' < |x'|$, this implies a 0 was deleted. Also

$$a' = R_1 = 1.$$

According to the algorithm, we insert a 0 to the left of the rightmost R_1 number of 1. Hence, we get

$$\mathbf{x} = (0,\ 1,\ 0,\ \mathbf{0},\ 1,\ 0).$$

We have successfully recovered the 4^{th} codeword in $C(6,7,0)$.

From (2.1), we see that for a given length n, the Levenshtein code construction generates 2^n codewords and divides them into $m = n+1$ codebooks. The Pigeonhole Principle assures that there is at least one codebook whose cardinality is greater than the proportion between the total number of codewords over the total number of codebooks. In other word, there exists a residue a such that

$$|C(n,m,a)| \geq \frac{2^n}{n+1}.$$

This inequality gives a lower bound on the cardinality of the largest Levenshtein codebook which we denote by L_n.

The information rate of the largest Levenshtein codebook is given by

$$\begin{aligned} I_n(q,d) &= \frac{\log_2 L_n}{n} \\ &\geq \frac{\log_2(\frac{2^n}{n+1})}{n} \\ &\geq \frac{n-\log_2(n+1)}{n} \end{aligned}$$

It follows that $I_n(q,d) \to 1$ as $n \to \infty$. We say that Levenshtein code is asymptotically optimal.

Helberg Codes

Code construction. Levenshtein codes are easy to decode and have good code rates. Its limitation is that it can only correct a single insertion/deletion error. In order to correct multiple deletions and insertions, Helberg [1] generalized the aforementioned construction by replacing the moment $\sum_{i=1}^{n} ix_i$ of the codeword $\mathbf{x}$ with the generalized moment $M_{\mathbf{x}} = \sum_{n=1}^{n} v_i x_i$ where v_i, called the i-th weight, is a modified version of the Fibonacci number. Fix d to be a positive integer (corresponding to the maximum number of deletions that our codebook is able to correct), and n to be the length of each codeword. We define

$$C_H(n,d,a) = \left\{ (x_1,x_2,\ldots,x_n) \in \{0,1\}^n : \sum_{i=1}^{n} v_i x_i \equiv a \pmod{m} \right\}, \tag{2.2}$$

where $m \geq v_{n+1}$, and

$$v_i = \begin{cases} 0, & \text{for } i \leq 0; \\ 1 + \sum_{j=1}^{d} v_{i-j}, & \text{for } i \geq 1, \end{cases} \tag{2.3}$$

We shall prove later that $C_H(n,m,a)$ is capable of correcting up to d insertion/deletion errors.

For $d = 2$, the first few weights are given by

$$\begin{aligned} v_{-1} &= 0 \\ v_0 &= 0 \\ v_1 &= 1 + v_0 + v_{-1} = 1 \\ v_2 &= 1 + v_1 + v_0 = 2 \\ v_3 &= 1 + v_2 + v_1 = 4 \end{aligned}$$

The Table 3 gives a listing of the first 10 weights.

Table 3

Weight v_i for $d = 2$

i	1	2	3	4	5	6	7	8	9	10
v_i	1	2	4	7	12	20	33	54	88	143

Example 2.17. Fix $d = 2$, we compute the moment of the codeword $\mathbf{x} = (0,\ 0,\ 1,\ 0,\ 1,\ 1)$ using Table 3:

$$\sum_{i=1}^{n} v_i x_i = 1 \cdot 0 + 2 \cdot 0 + 4 \cdot 1 + 7 \cdot 0 + 12 \cdot 1 + 20 \cdot 1 = 36 \equiv 3 \pmod{33}.$$

Therefore $\mathbf{x} \in C_H(6, 2, 3)$.

Proof of multiple error correction. In this section, we prove that Helberg codes are capable of correcting multiple insertion/deletion errors. For many years since its conception in 1993, Helberg codes have been the only known explicit construction intended for multiple insertion/deletion corrections. By exhaustive testing, Helberg and Ferreira verified that such a construction gave codes capable of corrections d insertion/deletion errors for up to length

of 14 and for d up to 5. In 2011, Abdel-Ghaffar et al. [10] proved that the code as defined in (2.2) is indeed an d insertion/deletion correcting codes. We shall describe their proof next.

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a codeword with length n and D be a subset of $\{1, 2, \ldots, n\}$. We denote $|D| = d$ and $\{1, 2, \ldots, n\} \setminus D = \{i_1, i_2, \ldots, i_{n-d}\}$, where $1 \leq i_1 < i_2 < \cdots < i_{n-d} < n$. We define $\mathbf{x}^{(D)} = (x_{i_1}, x_{i_2}, \ldots, x_{i_{n-d}})$ to be the codeword obtained by deleting all bits with indices in D. If $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ is also a codeword with length n with $E \subseteq \{1, 2, \ldots, n\}$, and $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$, then $|D| = |E| = d$ and

$$x_{i_l} = y_{i_l} \tag{2.4}$$

for $l = 1, 2, \ldots, n-d$, where

$$\{1, 2, \ldots, n\} \setminus E = \{j_1, j_2, \ldots, j_{n-d}\}$$

and

$$1 \leq j_1 < j_2 < \cdots < j_{n-d} < n.$$

We write $\mathbf{x} \equiv \mathbf{y}$ if and only if $\sum_{i=1}^{n} v_i x_i \equiv \sum_{i=1}^{n} v_i y_i \pmod{m}$.

The following lemma will allow us to replace the right most 1 by 0 in any two equivalent codewords and have the same deleted codeword.

Lemma 2.18. *(Abdel-Ghaffar, Paulunčić, Ferreira, Clarke [10])*

Let $\mathbf{x}$ *and* $\mathbf{y}$ *be two codeword of length n such that* $\mathbf{x} \equiv \mathbf{y}$ *and* $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$ *with* $|D| = |E| = d$. *Suppose there exists a positive number* i_{max} *such that* $x_{i_{max}} = y_{i_{max}} = 1$ *and* $x_i = y_i = 0$ *for all* $i > i_{max}$. *Define* $\tilde{\mathbf{x}} = (\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n)$ *and* $\tilde{\mathbf{y}} = (\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_n)$, *where* $\tilde{x}_{i_{max}} = \tilde{y}_{i_{max}} = 0$, $\tilde{x}_i = x_i$, *and* $\tilde{y}_i = y_i$ *for* $i \neq i_{max}$. *Then* $\tilde{\mathbf{x}} \equiv \tilde{\mathbf{y}}$ *and* $\tilde{\mathbf{x}}^{(D_0)} = \tilde{\mathbf{y}}^{(E_0)}$ *for some subset of* D_0 *and* E_0 *of the same size of D and E*.

This lemma allows us to prove the following two lemmas:

Lemma 2.19. *(Abdel-Ghaffar, Paulunčić, Ferreira, Clarke [10])*

Let $\mathbf{x}$ *and* $\mathbf{y}$ *be any two codeword of length n such that* $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$ *for subset D and E of* $\{1,2,\ldots,n\}$ *of size at most equal to d. We have*

$$\left| \sum_{i=1}^{n} v_i(x_i - y_i) \right| < m.$$

Lemma 2.20. *(Abdel-Ghaffar, Paulunčić, Ferreira, Clarke [10])*

Let $\mathbf{x}$ *and* $\mathbf{y}$ *be any two codeword of length n such that* $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$ *for subset D and E of* $\{1,2,\ldots,n\}$ *of size at most equal to d. We have*

$$\sum_{i=1}^{n} v_i(x_i - y_i) \neq 0.$$

The previous three lemmas imply the following result.

Theorem 2.21. *(Abdel-Ghaffar, Paulunčić, Ferreira, Clarke [10])*

The Helberg code $C_H(n,d,a)$ *is an d insertions-deletions correcting code.*

Proof. Assume $C_H(n,d,a)$ is not an d insertion/deletion correcting code. Therefore it has two distinct codeword $\mathbf{x}$ and $\mathbf{y}$ such that $\mathbf{x} \equiv \mathbf{y}$ and $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$ for some subset D and E of size at most d. However, Lemma 2.19 and 2.20 state that no such sequences exist. Hence, $C_H(n,d,a)$ is an d-deletion correcting code. By the result of Levenshtein [15], Helberg code is also an d insertion/deletion correcting code.

Cardinality of Helberg codes. Beside lacking an algorithm to decode insertions (we give an algorithm to decode deletion in Chapter 3), Helberg codes also suffer from a low cardinality [5]. As defined in (2.2), for a given n, Helberg codes generate 2^n codewords and divide them into $m = v_{n+1}$ codebooks. Hence, the lower bound of the largest Helberg code

is given by

$$|C_H(n,d,r)| \geq \frac{2^n}{v_{n+1}}$$

for some residue r. Thus the information rate of the largest Helberg code is bounded below by

$$I_H \geq \frac{\log_2 |C_H(n,d,r)|}{n} = \frac{n - \log_2 v_{n+1}}{n}.$$

On the other hand, the information rate of Helberg codes is upper bounded asymptotically by $1/d$ [5]:

$$I_H \leq \frac{1}{d}$$

We denotes the size of the largest Helberg code in term of code length by:

$$N_n(d) = \max\{|C_H(n,d,a)| : a = 0,1,\ldots,v_{n+1}-1\}$$

We also denote $R_n(d)$ be the set of values of a such that $|C_H(n,d,a)| = N_n(d)$. By exhaustive computer search, we computed the values $N_n(d)$ and $R_n(d)$ for certain values of n and d. Table 4 gives values for $N_n(2)$ and $R_n(2)$ for 2-deletions Helberg Codes. Table 5 gives similar values for 3-deletions Helberg codes.

Table 4

Binary 2-deletion Helberg Codes: Values of $N_n(2)$ and $R_n(2)$

n	$N_n(2,2)$	$R_n(2,2)$
1	1	0,1
2	1	0,1,2,3
3	2	0
4	2	0,1,2,7
5	2	0,1,2,3,4,5,6,7,12,13,14,19
6	3	0,1,6,7, 12,13
7	4	12,13
8	5	12,33
9	6	12,33,39,45,66
10	8	66
11	9	65,66,99,100,120,121,154,155
12	11	65,66,99,154,155,175,176,181,182,187,188, 208,209,264,297,298
13	15	297,298
14	18	297,441,475,496,530,674
15	22	297,441,674,763,784,790,796,817,906,1139,1283
16	30	1283

Table 5

Binary 3-deletion Helberg Codes:Values of $N_n(3)$ and $R_n(3)$

n	$N_n(2,3)$	$R_n(2,3)$
1	1	0,1
2	1	0,1,2,3
3	1	0,1,2,3,4,5,6,7
4	2	0
5	2	0,1,2,15
6	2	0,1,2,3,4,5,6,115,28,29,30,43
7	2	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,28,29, 30,43,52,53,54,55,56,57,58,67,80,81,82,95
8	3	0,1,14,15,28,29
9	4	28,29
10	4	28,29,52,53,54,55,56,57,80,81,354,355
11	5	28,177
12	6	28,177,191,205,354

Let the asymptotic lower and upper bounds for the cardinality of the largest code possible that is capable of correcting up to d deletions be defined by [16]

$$L_n(d) = \frac{(d!)^2 2^{n+d}}{n^{2d}},$$

$$U_n(d) = \frac{d!2^n}{n^d},$$

Table 6 and 7 compare the cardinalities $N_n(d)$ with $L_n(d)$ and $U_n(d)$ for 2-deletions and 3-deletions, respectively.

Table 6

Cardinality Binary 2-deletion Helberg Codes

n	$L_n(2)$	$N_n(2)$	$U_n(2)$
4	1	2	2
5	0.8192	2	2.56
6	0.790123	3	3.55556
7	0.852978	4	5.22449
8	1	5	8
9	1.24859	6	12.642
10	1.6384	8	20.48
11	2.2381	9	33.8512
12	3.16049	11	56.8889
13	4.5892	15	96.9467
14	6.82382	18	167.184
15	10.3563	22	291.271
16	16	30	512

Table 7

Cardinality Binary 3*-deletion Helberg Codes*

n	$L_n(3)$	$N_n(3)$	$U_n(3)$
4	1.125	2	6
5	0.589824	2	7.68
6	0.395062	2	10.6667
7	0.313339	2	15.6735
8	0.28125	3	24
9	0.277464	4	37.9259
10	0.294912	4	61.44
11	0.33294	5	101.554
12	0.395062	6	170.667

Non-binary Error Correcting Codes

Tenengol'ts code. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a codeword be defined over q-ary alphabets where $0 \leq x_i \leq q-1$. Tenegol'ts associates to $\mathbf{x}$ a binary codeword $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ define by

$$\alpha_i = \begin{cases} 1, \text{ if } x_i \geq x_{i-1} \\ 0, \text{ if } x_i < x_{i-1} \end{cases} \tag{2.5}$$

for $2 \leq i \leq n$.

In this construction, we let $\alpha_1 = 1$ and define the following system of congruences:

$$\sum_{i=1}^{n} x_i \equiv \beta \pmod{q} \tag{2.6}$$

$$\sum_{i=1}^{n} (i-1)\alpha_i \equiv \gamma \pmod{n}. \tag{2.7}$$

The purpose of (2.6) is to determine the value of inserted or deleted bit while (2.7) essentially defines$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ to be a Levenshtein codeword of length n.

We next demonstrate how the non-binary Tenengol'ts code construction gives a single insertion/deletion correcting code. In the case of deletion error, let $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_{n-1})$ denote the deleted codeword and $\alpha' = (\alpha'_1, \alpha'_2, \ldots, \alpha'_{n-1})$ denote the associated deleted codeword. Let $|\alpha'|$ denote the number of 1's in α'.

We define S_1 and S_2 as the least nonnegative residues of the congruences

$$S_1 \equiv \beta - \sum_{i=1}^{n-1} x'_i \pmod{q}$$

$$S_2 \equiv \gamma - \sum_{i=1}^{n-1} (i-1)\alpha' \pmod{n}.$$

It follows that S_1 is equal to the value of deleted element in $\mathbf{x}$. The corresponding deleted element in α' is determined by the Levenshtein decoding algorithm:

1. If $S_2 \geq |\alpha'|$, then insert 1 into α' so that the number of zeros to the left of the inserted element will be equal to $S_2 - |\alpha'|$.

2. If $S_2 < |x'|$, then insert 0 into α' so that the number of ones to the right of the inserted element will be equal to S_2.

In the case of insertion error, let $\mathbf{x}'' = (x''_1, x''_2, \ldots, x''_n, x''_{n+1})$ denote the inserted codeword and $\alpha'' = (\alpha''_1, \alpha''_2, \ldots, \alpha''_n, \alpha''_{n+1})$ denote the associated inserted codeword. We define S_1

and S_2 as the least nonnegative residues of the congruences

$$S_1 \equiv \sum_{i=1}^{n+1} x_i'' - \beta \pmod{q}$$
$$S_2 \equiv \sum_{i=1}^{n+1} (i-1)\alpha'' - \gamma \pmod{n}.$$

It follows that S_1 is equal to the value of the inserted symbol in $\mathbf{x}$. The correspondent inserted element in α'' is determined by:

1. If $S_2 = 0$, then delete the last element in α''.
2. If $0 < S_2 < |\alpha'| - 1$, then delete any zero so that the number of 1 on the right of this element in α'' is equal to S_2.
3. If $S_2 = |\alpha'| - 1$, then delete the second element in α''.
4. If $S_2 > |\alpha'| - 1$, then delete any 1 so that the number of zeros on the right of this element is equal to $n - S_2$.

We define the Tenengol'ts code $C_T(n,q,\beta,\gamma)$ to be all such codeword $\mathbf{x}$ satisfy the congruences (2.6) and (2.7).

Theorem 2.22. *(Grigory Tenengol'ts [7])*

$C_T(n,q,\beta,\gamma)$ is a single insertion/deletion correcting code.

We present an example of a non-binary Tenengol'ts codes.

Example 2.23. Let $q = 4$, $n = 8$ and let $\beta = \gamma = 0$ in (2.6) and (2.7). We show that the codeword $\mathbf{x} = (0,1,2,2,3,2,2,0) \in C_T(8,4,0,0)$. Since

$$\sum_{i=1}^{n} x_i = 0+1+2+2+3+2+2+0 = 12 \equiv 0 \pmod{4}$$

and the associated binary sequence $\alpha = (1,1,1,1,1,0,1,0)$ satisfies

$$\sum_{i=1}^{n} (i-1)\alpha_i = 1+2+3+4+6 = 16 \equiv 0 \pmod{4},$$

It follows that $\mathbf{x} \in C_T(8,4,0,0)$.

Suppose we received the deleted codeword $\mathbf{x}' = (0,1,2,2,2,2,0)$. The associated binary codeword is $\alpha' = (1,1,1,1,1,1,0)$.

We have:

$$\begin{aligned} &|\alpha'| = 6, \\ &S_1 = -(0+1+2+2+2+2+0) = -9 \equiv 3 \pmod 4, \\ &S_2 = -(1+2+3+4+5) = -15 \equiv 1 \pmod 8. \end{aligned}$$

We now apply Levenshtein decoding on the associated binary codeword α'. Since $S_2 < |\alpha'|$, we know that a 0 was deleted which we insert to the left of the first 1 from the right since $S_2 = 1$. As a result we have $\alpha = (1,1,1,1,1,0,1,0)$.

Since $S_1 = 3$, it follows that the symbol 3 in $\mathbf{x}$ has been deleted. Since a 0 was inserted in the first run of 1's, we conclude that the symbol 3 should be inserted in either the corresponding run in $\mathbf{x}'$ or in the preceding run. In this case, it's located in the first run. We simply move the symbol 3 along this run in $\mathbf{x}'$ until it associated binary codeword matches with α. As a result, the correct decoding for $\mathbf{x}$ is

$$\mathbf{x} = (0,1,2,2,3,2,2,0).$$

Generalization of Tenengol'ts code. Palunčić et al. [6] have showed how the Tenengol'ts approach can be extended to construct multiple insertion/deletion error correcting codes. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a q-ary codeword and define its associated binary codeword α as in (2.5). But unlike the Tenengol'ts construction, we assume that α comes from a codebook that can correct d insertion/deletion errors. In this thesis, we assume that α comes from a Helberg codebook.

Fix d to be a positive integer corresponding to the maximum number of insertion/deletion errors in a codebook. Given a set of residues $\beta = \{\beta_1, \ldots, \beta_d\}$ and γ. We define $C_T(n,q,\beta,\gamma)$

to be the set of codeword $\mathbf{x}$ that satisfies the following system of congruences:

$$\begin{aligned}
\sum_{i=1}^{n} x_i &\equiv \beta_1 \pmod{p}, \\
\sum_{i=1}^{n} {x_i}^2 &\equiv \beta_2 \pmod{p}, \\
&\vdots \\
\sum_{i=1}^{n} {x_i}^d &\equiv \beta_d \pmod{p},
\end{aligned} \tag{2.8}$$

where p is any prime satisfying $p > \max(q-1,\ d)$ and

$$\sum_{i=1}^{n} (i-1)\alpha_i \equiv \gamma \pmod{v_n}. \tag{2.9}$$

In this thesis, we always choose p to be the smallest such prime. Observe that $\alpha \in C_H(n,d,\gamma)$.

Next, we construct a codebook $\hat{C}_T(n,q,\beta,\gamma)$ by purging certain codewords from $C_T(n,q,\beta,\gamma)$ in order to turn it into a d insertion/deletion correcting codes. The prove relies on the following lemmas.

Lemma 2.24. *[6]*

The set of congruences in (2.8) *for fixed values of* $(\beta_1,\beta_2,\ldots,\beta_d)$ *can uniquely determine the values of* d *or fewer symbols deleted from* $\mathbf{x} \in C_T(n,q,\beta,\gamma)$.

Lemma 2.25. *[6]*

If d *symbols are deleted from a codeword* $\mathbf{x} = (x_1,x_2,\ldots,x_n) \in C_T(n,q,\beta,\gamma)$*, then any deleted codeword* $\mathbf{x}' = (x'_1,x'_2,\ldots,x'_{n-d})$ *can be obtained by at most* $d!$ *codewords from* $C_T(n,q,\beta\gamma)$.

This lemma shows that in general $C_T(n,q,\beta,\gamma)$ is not capable of correcting d insertion/deletion errors as illustrated in following example.

Example 2.26. Assume $\mathbf{x} \in C_2(6,6,(3,2))$ is transmitted and its associated binary codeword is $\alpha \in C_H(6,2,3)$. Suppose two deletions occurred so that $\mathbf{x}' = (5,5,3,4)$ is received. We want to show that there are two possibilities for $\mathbf{x}$.

First, we determine the value of the deleted bits. Since $\mathbf{x} \in C_2(6,6(3,2))$, we have $d = 2$, $q = 6$, $n = 6$, $\beta_1 = 3$, and $\beta_2 = 2$. We denote the deleted bits by x_j and x_k where $0 \leq x_j, x_k \leq q-1$.

We apply the system of congruences (2.8) to determine β_1' and β_2' where $p = 7 > \max(q-1,d) = \max(5,2)$:

$$\beta_1' \equiv \sum_{i=1}^{4} (x_i')^1 = 5+5+3+4 = 17 \equiv 3 \pmod 7$$
$$\beta_2' \equiv \sum_{i=1}^{4} (x_i')^2 = 5^2+5^2+3^2+4^2 = 75 \equiv 5 \pmod 7$$

Hence, $\beta_1' = 3$ and $\beta_2' = 5$. We set up a system of congruences to determine the values x_j and x_k:

$$x_j^1 + x_k^1 \equiv \beta_1 - \beta_1' \equiv 0 \pmod 7$$
$$x_j^2 + x_k^2 \equiv \beta_2 - \beta_2' \equiv -3 \pmod 7$$

Since $0 \leq x_j, x_k \leq 5$. it follows that there are two possibilities: either $x_j = 3$ and $x_k = 4$ or $x_j = 4$ and $x_k = 3$.

Next, we want to show that we can determine the positions of the two deleted bits. From $\mathbf{x}' = (5,5,3,4)$, the associated binary codeword is $\alpha' = (0,1,0,1)$. Since $\alpha \in C_H(6,2,3)$, we can decode α' to recover $\alpha = (0,0,1,0,1,1)$ (explained in Chapter 3). Thus the bits 0 and 1 were deleted from say positions 2 and 5 in α.

However, the problem is that we do not know which symbols (whether x_j or x_k) is associated with which bits (0 or 1) in the associated binary codeword. The original codeword could be

$$\mathbf{x}_1 = (5,3,5,3,4,4)$$

Or

$$\mathbf{x}_2 = (5,4,5,3,3,4)$$

since both $\mathbf{x}_1, \mathbf{x}_2 \in C_2(6,6,(3,2)$. However, the number of codewords we can recover from $\mathbf{x}'$ is at most $d!$, which equals 2 in this case.

Therefore, to construct $\hat{C}_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ from $C_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ so that $\hat{C}_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ is a d insertion/deletion error correcting codes, we apply the following purging process:

1. Choose any $\mathbf{x} = (x_1,x_2,\ldots,x_n) \in C_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$. For all possible combinations of deleting d symbols from $\mathbf{x}$, determine the set $D = \{x_n^{(1)}, x_n^{(2)}, \ldots, x_n^{(\sigma)}\}$, where $\sigma \leq d!$. D is called the decoding set. Purge all codewords in D from $C_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ except $\mathbf{x}$. Insert $\mathbf{x}$ into $\hat{C}_T(n,q,\beta,\gamma)$.

2. Choose $\mathbf{y} = (y_1,y_2,\ldots,y_n) \in C_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ such that $\mathbf{y}$ is not equal to any selected or purged codewords. Repeat the purging process for $\mathbf{y}$ as in Step 1.

3. Repeat Step 2 until there are no codewords left in $C_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ that can be selected.

Theorem 2.27. *[6]*

The set $\hat{C}_T(q,n,(\beta_1,\beta_2,\ldots,\beta_d))$ is an d insertion/deletion correcting code.

Example 2.28. From the previous example, suppose we select $\mathbf{x}_1 = (5,3,5,3,4,4) \in C_2(6,6,(3,2))$. Then observe that given that $\mathbf{x}_2 = (5,4,5,3,3,4) \in D$, the decoding set for $\mathbf{x}_1$. Thus, we purge $\mathbf{x}_2$ from $C_2(6,6,(3,2))$ since we saw earlier that both $\mathbf{x}_1$ and $\mathbf{x}_2$ yield the same deleted codeword $\mathbf{x}'$. Repeating this purging process yield the codebook $\hat{C}_2(6,6,(3,2))$ capable of correcting 2 insertion/deletion errors.

Chapter 3

Decoding Algorithm for Helberg Codes

Deletion Decoding Algorithm

In this chapter, we present a linear Deletion Decoding Algorithm to decode deleted Helberg codewords. The algorithm works very well for deletion errors but unfortunately, it does not work for insertion errors. Therefore this algorithm is unable to correct both insertion/deletion errors. Hence, as we will see in some lemmas and theorems, we intend to ignore the insertion feature. We will focus on proving lemmas and theorems for deletion codes only. Why does the algorithm work for deletion errors but not for insertion errors in the same class of codes? We will discuss this in detail in Chapter 6.

Another thing we assume is that Helberg codes are synchronized. This means that we know the beginning and ending of each codeword in the whole message we received.

Before we go into the details of the algorithm, let's describe our problem first: We have received a message containing codewords where some bits in the codewords were deleted. We wish to recover the original message. The only available information for us is the deleted codeword we received and the codebook it comes from, which in turn gives us the length of codeword, the residue, and the modulus.

The idea of this algorithm is very simple: We add some bits to the deleted codeword to try to get back the original codeword.

Define:

- The binary codeword $\mathbf{x}$ with fixed length n: $\mathbf{x} = (x_1, x_2, x_3, \ldots, x_n)$.
- The deleted codeword $\mathbf{x}'$ with d deleted bits: $\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{n-d})$.
- Let $\tilde{\mathbf{x}}$ be our initial decoding for $\mathbf{x}'$ where we insert d variable symbols $\delta_1, \ldots, \delta_d$ at the right most position: $\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{n-d}, \delta_1, \delta_2, \ldots, \delta_d)$.

- v_i is the weight of x_i at position i.
- The *original moment*: $M_{\mathbf{x}} = \sum_{i=1}^{n} v_i x_i$.
- The *deleted moment*: $M_{\mathbf{x}'} = \sum_{i=1}^{n} v_i x'_i$.
- The *modolus*: $m = v_{n+1}$.
- The *residue*: $r = \sum_{i=1}^{n} v_i x_i \mod v_{n+1}$.
- The *index*: $I = M_{\mathbf{x}} - M_{\mathbf{x}'}$.

The Deletion Decoding Algorithm is a technique of tracking the difference between the moment of the original codeword and the deleted codeword. Therefore it requires knowing the moment of the original codeword without knowing the value of each bit in it. The following lemmas will allow us to determine $M_{\mathbf{x}}$ from $M_{\mathbf{x}'}$.

Lemma 3.1 ([5]). *For $d \geq 2$,*

$$\sum_{i=1}^{n} v_i = \frac{\sum_{i=0}^{d-1} (d-i) v_{n-i} - n}{d-1}.$$

Lemma 3.2 ([5]). *For $d \geq 2$,*

$$\sum_{i=1}^{n} v_i < \frac{d}{d-1} v_{n+1} - 1.$$

Lemma 3.3. *If $M_{\mathbf{x}'} > r$, then $M_{\mathbf{x}} = r + m$. Otherwise, if $M_{\mathbf{x}'} < r$, then $M_{\mathbf{x}} = r$.*

Proof. By definition,

$$M_{\mathbf{x}} = \sum_{i=1}^{n} v_i < \frac{d}{d-1} v_{n+1} - 1,$$

where the inequality follows from Lemma 3.2.

Since $d \geq 2$, this implies

$$\begin{aligned} d &\geq 2 \\ 2d - d &\geq 2 \\ 2(d-1) &\geq d \\ 2 &\geq \frac{d}{d-1}. \end{aligned}$$

Therefore,

$$\begin{aligned} M_{\mathbf{x}} &< \frac{d}{d-1} v_{n+1} - 1 \\ &< 2v_{n+1} - 1 \\ &< 2m. \end{aligned}$$

Recall that $M_{\mathbf{x}} \equiv r \pmod{m}$, therefore $M_{\mathbf{x}} = r + tm$ where $t = 0, 1, 2, \ldots$.

Since $M_{\mathbf{x}} < 2m$, this implies $M_{\mathbf{x}} = r$ or $M_{\mathbf{x}} = r + m$. We consider two cases:

- Case I: $M_{\mathbf{x}'} > r$. Since $M_{\mathbf{x}} > M_{\mathbf{x}'} > r$, therefore $M_{\mathbf{x}} = r + m$.
- Case II: $M_{\mathbf{x}'} < r$, then $M_{\mathbf{x}} = r$ or $M_{\mathbf{x}} = r + m$.

  Suppose $M_{\mathbf{x}} = r + m$. By definition, we have:

$$v_n = 1 + \sum_{j=1}^{d} v_{n-j}.$$

  Let $\Delta_{\mathbf{x}}$ is the maximum change in the moment from $M_{\mathbf{x}}$ to $M_{\mathbf{x}'}$. We have

$$\Delta_{\mathbf{x}} = \sum_{j=1}^{d} v_{n-j} < v_n < v_{n+1} = m$$

  We calculate the deleted moment as follow:

$$\begin{aligned} M_{\mathbf{x}'} &= M_{\mathbf{x}} - \Delta_{\mathbf{x}} \\ &= r + m - \Delta_{\mathbf{x}} > r \end{aligned}$$

since $m > \Delta_{\mathbf{x}}$. But this is a contradiction since $M_{\mathbf{x}'}$ is supposed to be less than r. Hence, $M_{\mathbf{x}} = r$.

Therefore, if $M_{\mathbf{x}'} > r$, then $M_{\mathbf{x}} = r + m$; otherwise, if $M_{\mathbf{x}'} < r$, then $M_{\mathbf{x}} = m$.

The Deletion Decoding Algorithm works as follow: We assume the deleted elements are at the right end and try to insert values for those positions. If these positions are not correct, then shift the symbols to the left. Hence, with one deletion error, this algorithm basically performs an exhaustive search. But with two or more errors, this algorithm will require less work (compare to exhaustive search) as the following sections will illustrate.

One Deletion Method

Suppose we have a deleted codeword with length $n-1$. A bit x_i at position i has the corresponding weight v_i. Initialize $p = n$. Assume the deleted element is at the right end, i.e., position p, denoted as δ. We have:

$$\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{p-1}).$$
$$\widetilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{p-1}, \delta).$$

We adjust the deleted codeword moment by inserting a value or shifting this deleted element to the left using following conditions:

Algorithm D1 (Decode One Deletion).

1. If $I = 0$, then insert 0 for δ, stop.

2. If $I = v_p$, then insert 1 for δ, stop.

3. Otherwise, we shift x'_{p-1} to the right of δ, and update the index so that $I = I - x'_{p-1} \cdot (v_p - v_{p-1})$. Repeat through steps 1 to 3.

Since this algorithm is essentially an exhaustive search, it will correctly decode $\mathbf{x}'$.

Example 3.4. Assume our codebook $C \subset C_H(8,2,25)$ is as follows:

$$C = \{(0,1,1,1,1,0,0,0),\ (1,0,1,0,0,1,0,0),$$
$$(0,1,1,0,0,1,1,1),\ (1,1,1,1,1,0,1,1)\}.$$

Suppose we receive a word $\mathbf{x}'$ where one bit has been deleted from $\mathbf{x}$

$$\mathbf{x}' = (1,1,1,1,0,1,1).$$

We wish to decode $\mathbf{x}'$ to recover $\mathbf{x}$.

Even though this involves only one deletion error, our code is capable of correcting up to two errors. Hence, we shall apply the weight from Table 3.

The modulus is

$$m = v_{n+1} = 88.$$

The residue is given as:

$$r = 25.$$

The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= \sum_{i=1}^{n-1} v_i x_i' \\ &= 1\cdot 1 + 2\cdot 1 + 4\cdot 1 + 7\cdot 1 + 12\cdot 0 + 20\cdot 1 + 33\cdot 1 \\ &= 67. \end{aligned}$$

Since $M_{\mathbf{x}'} > r$, we follow Lemma 3.3 and calculate the original moment as follows:

$$\begin{aligned} M_{\mathbf{x}} &= m + r \\ &= 88 + 25 \\ &= 113. \end{aligned}$$

Therefore the index is

$$I = 113 - 67 = 46.$$

Assume the deleted element is at the right end so that our initial decoding is

$$\tilde{\mathbf{x}} = (1,1,1,1,0,1,1,\delta).$$

Since $I \neq 0$ and $I \neq 54$ we shift δ to the left one position as follows:

$$\tilde{\mathbf{x}} = (1,1,1,1,0,1,\delta,1).$$

We update the index as follows:

$$I = 46 - (54 - 33) = 25.$$

Since $I \neq 0$ and $I \neq 33$ we shift δ again so that

$$\tilde{\mathbf{x}} = (1,1,1,1,0,\delta,1,1).$$

We update the index so that

$$I = 25 - (33 - 20) = 12.$$

Since $I \neq 0$ and $I \neq 20$ we shift δ yet again to get

$$\tilde{\mathbf{x}} = (1,1,1,1,\delta,0,1,1).$$

Notice that by shifting δ to the left one position is the same as shifting 0 (bit at the firth

position) to the right. But by shifting 0, the index does not change, namely

$$I = 12.$$

Since $I \neq 0$ but $I = 12 = v_5$, we insert $\delta = 1$ at position 5. Thus the original codeword is

$$\tilde{\mathbf{x}} = (1,1,1,1,1,0,1,1) = \mathbf{x}.$$

The Deletion Decoding Algorithm with one deletion error is simply an exhaustive search. It is not efficient but this algorithm improve when more than one error occurs as the next section will demonstrate.

Two Deletions Method

Suppose we have a deleted codeword with length $n-2$. Assume our 2 deleted bits are at the right end, represented by δ_1 and δ_2. Initialize $p = n$, so that we have:

$$\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{p-2}).$$
$$\widetilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{p-2}, \delta_1, \delta_2).$$

If the deleted elements are not in their correct positions, then we want to shift them to the left as far as possible. Recall that we denote the deleted moment by $M_{\mathbf{x}'}$, the current working codeword moment which contains δ_1 and δ_2 by $M_{\tilde{\mathbf{x}}}$, and the original moment by $M_{\mathbf{x}}$. We are looking for values of δ_1 and δ_2 such that $M_{\tilde{\mathbf{x}}} = M_{\mathbf{x}}$. We keep track of the index I by subtracting from it the change in the *weight* after shifting or inserting a bit.

We determine the values of δ_1, δ_2, and their positions using the following algorithm:

Algorithm D2 (Decode Two Deletions). If

$$I = \sigma_1 \cdot v_{p-1} + \sigma_2 \cdot v_p$$

for a set of values $\sigma_1,\ \sigma_2\ \in\ \{0,1\}$, then δ_1, δ_2 are in their correct positions. To decode, let $\delta_1 = \sigma_1, \delta_2 = \sigma_2$. Otherwise:

1. For $x'_{p-2} = 0$:

   (a) If $v_p > I$, then shift the last known bit x'_{p-2} to the right two positions and decrement p.

   (b) If $v_p < I$, then insert $\delta_2 = 1$. Update the index $I = I - v_p$. Apply Algorithm D1.

2. For $x'_{p-2} = 1$:

   (a) If $v_p > I$:

      i. If $(v_p - v_{p-2}) \leq I$, then shift x'_{p-2} to the right two positions and decrement p. Update the index $I = I - (v_p - v_{p-2})$.

      ii. If $(v_p - v_{p-2}) > I$, then insert $\delta_2 = 0$. Apply Algorithm D1.

   (b) If $v_p < I$, then shift x'_{p-2} to the right two positions and decrement p. Update the index $I = I - (v_p - v_{p-2})$.

Proof. We have:

$$\widetilde{\mathbf{x}} = (x'_1,\ x'_2,\ x'_3,\ \ldots,\ x'_{p-2},\ \delta_1,\ \delta_2).$$

1. Suppose $x'_{p-2} = 0$. We consider two cases:

   (a) $v_p > I$. Either δ_2 is in the correct position or not. If δ_2 is in the correct position then it has to be a 0; otherwise, if $\delta_2 = 1$, then the moment of $\tilde{\mathbf{x}}$ will exceed $\mathbf{x}$, regardless of the position and value of δ_1:

$$\begin{aligned} \min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + \delta_2 \cdot v_p \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}} \end{aligned}$$

But observe that inserting $\delta_2 = 0$ is equivalent to shifting x'_{p-2} to the right of δ_2. The other situation is that δ_2 is NOT in the correct position. This forces us to shift x'_{p-2} to the right of δ_2.

(b) $v_p < I$. We consider two situations regarding the position of δ_2 again. If δ_2 is NOT in the correct position then we are forced to shift x'_{p-2} to the right of δ_2 so that

$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{p-3}, \delta_1, \delta_2, 0).$$

But then we are not able to shrink the index since

$$\begin{aligned} \max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + v_{p-1} + v_{p-2} \\ &< M_{\mathbf{x}'} + v_p \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}}. \end{aligned}$$

Hence, δ_2 is in the correct position. If $\delta_2 = 0$, we cannot recover $\mathbf{x}$ since

$$\begin{aligned} \max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + M_{\delta_1} \\ &= M_{\mathbf{x}'} + v_{p-1} \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}}. \end{aligned}$$

Therefore, δ_2 has to be a 1.

2. Suppose $x'_{p-2} = 1$. We consider two cases:

(a) $v_p > I$. We consider two sub-cases:

i. $I \geq (v_p - v_{p-2})$. Suppose δ_2 is in the correct position. If $\delta_2 = 1$, the increase in the moment for $\tilde{\mathbf{x}}$ will exceed the index since $v_p > I$. If $\delta_2 = 0$, the we

will not able to shrink the index since

$$\begin{aligned}\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + v_{p-1} \\ &< M_{\mathbf{x}'} + (v_p - v_{p-2}) \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}}.\end{aligned}$$

Therefore, δ_2 is NOT in the correct position and we are forced to shift x'_{p-2} to the right of δ_2.

ii. $I < (v_p - v_{p-2})$. Assume δ_2 is NOT in the correct position, therefore we are forced to shift x'_{p-2} to the right of δ_2 so that

$$\begin{aligned}M_{\tilde{\mathbf{x}}} &= M_{\mathbf{x}'} + (v_p - v_{p-2}) \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}}.\end{aligned}$$

The increase in the moment for $\tilde{\mathbf{x}}$ exceeds the index. Therefore, δ_2 is in the correct position. δ_2 cannot be 1 since $v_p > I$. Hence, δ_2 has to be 0.

(b) $v_p < I$. There are two situations regard the position of δ_2. If δ_2 is in the correct position, then it has to be 1. Otherwise; if $\delta_2 = 0$, then we will not able to correct the index since

$$\begin{aligned}\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + v_{p-1} \\ &< M_{\mathbf{x}'} + v_p \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}}.\end{aligned}$$

But observe that inserting $\delta_2 = 1$ is equivalent to shifting x'_{p-2} to the right of δ_2. The other situation is that δ_2 is NOT in the correct position then we are forced to shift x'_{p-2} to the right of δ_2.

Example 3.5. Assume our code $C \subset C_H(10,2,62)$ is as follows:

$$C = \{(0,1,0,1,0,1,1,0,0,0),(1,0,0,1,0,0,0,1,0,0),$$
$$(1,1,0,1,0,1,1,0,1,1),(0,1,0,1,0,0,0,1,1,1)\}.$$

Suppose we have received the codeword $\mathbf{x}'$ where 2 bits have been deleted from $\mathbf{x}$:

$$\mathbf{x}' = (1,1,0,1,0,1,0,1).$$

We wish to decode $\mathbf{x}'$ to recover $\mathbf{x}$.

Since the length of $\mathbf{x}'$ is 8 and $\mathbf{x} \in C_H(10,2,62)$, we know two deletion errors have occurred. We shall apply the weights in Table 3.

The modulus is

$$m = v_{n+1} = 232.$$

The residue is given as:

$$r = 62.$$

The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= \sum_{i=1}^{n-2} v_i x_i' \\ &= 1\cdot 1 + 2\cdot 1 + 4\cdot 0 + 7\cdot 1 + 12\cdot 0 + 20\cdot 1 + 33\cdot 0 + 54\cdot 1 \\ &= 84. \end{aligned}$$

Since $M_{\mathbf{x}'} > r$, we follow Lemma 3.3 and calculate the original moment as follows:

$$\begin{aligned} M_{\mathbf{x}} &= m + r \\ &= 232 + 62 \\ &= 294. \end{aligned}$$

The index is

$$I = 294 - 84 = 210.$$

Assume the deleted elements are at the right end and are represented by δ_1 and δ_2. We have

$$\tilde{\mathbf{x}} = (1,1,0,1,0,1,0,1,\delta_1,\delta_2).$$

Since $x'_{p-2} = x'_8 = 1$, $v_n = 143 < I = 210$. We follow case $2(b)$ and shift x'_{p-2} two positions to the right, $p = p-1 = 9$. We update the index: $I = 210 - (143 - 54) = 121$, so

$$\tilde{\mathbf{x}} = (1,1,0,1,0,1,0,\delta_1,\delta_2,1).$$

Now we have $x'_{p-2} = x'_7 = 0$, $v_9 = 88 < I = 121$. We follow case $1(b)$ and insert $\delta_2 = 1$ at position 9. We update the index: $I = I - v_9 = 121 - 88 = 33$, so

$$\tilde{\mathbf{x}} = (1,1,0,1,0,1,0,\delta_1,1,1).$$

Now we apply Algorithm D1 to determine δ_1.

Since $I \neq 0$ and $I \neq 54$, we shift δ_1 and update $I = 33$, so $\tilde{\mathbf{x}}$ becomes

$$\tilde{\mathbf{x}} = (1,1,0,1,0,1,\delta_1,0,1,1).$$

Since $I \neq 0$ but $I = 33 = v_7$, we insert $\delta_1 = 1$ at position 7.

Thus the original codeword is

$$\mathbf{x} = (1,1,0,1,0,1,1,0,1,1).$$

As we have just shown, our algorithm decodes errors one by one. As a comparison, for

a codeword of length n, the complexity of exhaustive search for 2 deletions is

$$\binom{n}{2} = \frac{n(n-1)}{2} \sim O(n^2).$$

More generally, the complexity of exhaustive search for d deletions is equal to the number of ways of choosing two positions out of n:

$$\binom{n}{d} = \frac{n!}{d!(n-d)!} \sim O(n^d),$$

whereas for the Deletion Decoding Algorithm, the complexity is still of order n:

$$n \sim O(n).$$

Multiple Deletions Method

Let d be the number of deletions. Suppose we have a deleted codeword with length $n-d$. We assume the deleted bits are at the right end, represented by δ_i's. Initialize $p = n$. We have

$$\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{p-d}),$$
$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{p-d}, \delta_1, \delta_2, \ldots, \delta_{d-1}, \delta_d).$$

If the deleted elements are not in the correct positions, then we want to shift them to the left. We determine the values of $\delta_1, \delta_2, \ldots, \delta_d$, and their positions using the following algorithm:

Algorithm DM (Decode Multiple Deletions). If

$$I = \sigma_1 \cdot v_{p-d+1} + \sigma_2 \cdot v_{p-d+2} + \ldots + \sigma_d \cdot v_p$$

for a set of values $\sigma_1, \ldots, \sigma_d \in \{0,1\}$, then $\delta_1, \ldots, \delta_d$ are in the correct positions. To decode, set $\delta_1 = \sigma_1, \ldots, \delta_d = \sigma_d$. Otherwise:

1. For $x'_{p-d} = 0$:

   (a) If $v_p > I$, then shift x'_{p-d} to the right by d positions and decrement p.

   (b) If $v_p < I$, then insert $\delta_d = 1$. Apply DM to decode $d-1$ errors.

2. For $x'_{p-d} = 1$:

   (a) If $v_p > I$:

      i. If $(v_p - v_{p-d}) \leq I$, then shift x'_{p-d} to the right by d positions and decrement p.

      ii. If $(v_p - v_{p-d}) > I$, then insert $\delta_d = 0$. Apply DM to decode $d-1$ errors.

   (b) If $v_p < I$, then shift x'_{p-d} to the right by d positions and decrement p.

Proof. We will argue by contradiction, similar to the proof of the Two Deletions Method. We have

$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{p-d}, \delta_1, \delta_2, \ldots, \delta_{d-1}, \delta_d).$$

1. If $x_{p-d} = 0$. We consider two cases:

   (a) $v_p > I$. Either δ_d is in the correct position or not. If δ_d is in the correct position then it has to be a 0; otherwise, the increase in the moment for $\tilde{\mathbf{x}}$ will exceed the index, regardless of the positions and values of $\delta_1, \ldots, \delta_{d-1}$:

$$\begin{aligned} \min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + v_p \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}}. \end{aligned}$$

   But observe that inserting $\delta_d = 0$ is equivalent to shifting x'_{p-d} to the right of δ_d. The other situation is δ_d is *not* in the correct position. This forces us to shift x'_{p-d} to the right of δ_d.

   (b) $v_p < I$. Assume δ_d is NOT in the correct position. This forces us to shift x'_{p-d} to the right of δ_d so that

$$\tilde{\mathbf{x}} = (x'_1, x'_2, \ldots, x'_{p-d-1}, \delta_1, \delta_2, \ldots, \delta_{d-1}, \delta_d, 0).$$

Notice that $M_{\tilde{\mathbf{x}}}$ is maximum when the other δ_i's are 1, and that

$$\begin{aligned}
\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + (M_{\delta_1} + \ldots + M_{\delta_d}) \\
&= M_{\mathbf{x}'} + (v_{p-d} + \ldots + v_{p-1}) \\
&< M_{\mathbf{x}'} + v_p \\
&< M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

so we are not able to shrink the index.

Thus δ_d must be in the correct position. If $\delta_d = 0$, we will not able to shrink the index since

$$\begin{aligned}
\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + (M_{\delta_1} + \ldots + M_{\delta_{d-1}}) \\
&< M_{\mathbf{x}'} + v_p \\
&< M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

Therefore, δ_d is in the correct position and it has to be 1.

2. If $x_{p-d} = 1$. We consider two cases:

   (a) $v_p > I$. We consider two sub-cases:

      i. $(v_p - v_{p-d}) \leq I$. Assume δ_d is in the correct position. δ_d cannot be 1 since $v_p > I$ implies the increase in the moment of $\tilde{\mathbf{x}}$ will exceed the index. So δ_d has to be 0. But then we will not able to shrink the index since

$$\begin{aligned}
\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + (M_{\delta_1} + \ldots + M_{\delta_{d-1}}) \\
&= M_{\mathbf{x}'} + (v_{p-d+1} + \ldots + v_{p-1}) \\
&< M_{\mathbf{x}'} + (v_p - v_{p-d}) \\
&< M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

      Therefore, δ_d is NOT in the correct position. We are forced to shift x'_{p-d} to

the right of δ_d.

ii. $(v_p - v_{p-d}) > I$. Assume δ_d is NOT in the correct position. Then we are forced to shift x'_{p-d} to the right of δ_d so that

$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, \delta_1, \delta_2, \ldots, \delta_{d-1}, \delta_d, 1).$$

Then the change in moment of $\tilde{\mathbf{x}}$ exceeds the index since

$$\begin{aligned} M_{\tilde{\mathbf{x}}} &= M_{\mathbf{x}'} + (v_p - v_{p-d}) \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}}. \end{aligned}$$

Hence, δ_d is in the correct position. Now, δ_d cannot be 1 since $v_p > I$. Therefore, δ_d is in the correct position and it has to be 0.

(b) $v_p < I$. We consider two situations regard the position of δ_d. If δ_d is in the correct position then it has to be 1; for otherwise, we will not able to shrink the index, regardless of the positions and values of $\delta_1, \ldots, \delta_{d-1}$:

$$\begin{aligned} \max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + (M_{\delta_1} + \ldots + M_{\delta_{d-1}}) \\ &= M_{\mathbf{x}'} + (v_{p-d+1} + \ldots + v_{p-1}) \\ &< M_{\mathbf{x}'} + v_p \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}} \end{aligned}$$

But observe that inserting $\delta_d = 1$ is equivalent to shifting x'_{p-d} to the right of δ_d. The other situation is δ_d is NOT in the correct position, which forces us to shift x'_{p-d} to the right of δ_d.

Example 3.6. Assume our code $C \subset C_H(10,4,30)$ is as follows:

$$C = \{(1,1,1,1,1,0,1,1,1,1),\ (0,0,0,0,0,1,1,1,1,1),\ (0,1,1,1,1,0,0,0,0,0)\}.$$

Suppose we have received a codeword $\mathbf{x}'$ where 4 bits have been deleted from $\mathbf{x}$:

$$\mathbf{x}' = (1,1,1,0,0,0).$$

We wish to decode $\mathbf{x}'$ to recover $\mathbf{x}$.

Since the length of $\mathbf{x}'$ is 6 and $\mathbf{x} \in C_H(10,4,30)$, four deletion errors ($d = 4$) have occurred. The weight v_i associated with $d = 4$ is given in following table:

Table 8

Weight v_i for $d = 4$

i	1	2	3	4	5	6	7	8	9	10
v_i	1	2	4	8	16	31	60	116	224	432

The modulus is

$$m = v_{n+1} = 833.$$

The residue is given as

$$r = 30.$$

The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= 1\cdot 1 + 1\cdot 2 + 1\cdot 4 + 0\cdot 8 + 0\cdot 16 + 0\cdot 31 \\ &= 7. \end{aligned}$$

Since $M_{\mathbf{x}'} < r$, Lemma 3.3 states that

$$M_{\mathbf{x}} = r = 30.$$

The index is

$$I = 30 - 7 = 23.$$

Assume the deleted elements are at the right end and are represented by δ's. We have:

$$\tilde{\mathbf{x}} = (1,1,1,0,0,0,\delta_1,\delta_2,\delta_3,\delta_4).$$

Since $x'_{p-d} = x'_6 = 0,\ v_p = v_{10} = 432 > I = 23$, we follow case $1(a)$ and shift x'_{p-d} four positions to the right, $p = p - 1 = 9$. By shifting a zero, the index does not change.

$$\tilde{\mathbf{x}} = (1,1,1,0,0,\delta_1,\delta_2,\delta_3,\delta_4,0).$$

Similarly, $x'_{p-d} = x'_5 = 0,\ v_p = v_9 = 224 > I = 23$, case $1(a)$ suggests we shift x'_{p-d} four positions to the right, $p = p - 1 = 8$. By shifting a zero, the index does not change.

$$\tilde{\mathbf{x}} = (1,1,1,0,\delta_1,\delta_2,\delta_3,\delta_4,0,0).$$

Again, $x'_{p-d} = x'_4 = 0,\ v_p = v_8 = 116 > I = 23$, we follow case $1(a)$ and shift x'_{p-d} four positions to the right, $p = p - 1 = 7$. By shifting a zero, the index does not change.

$$\tilde{\mathbf{x}} = (1,1,1,\delta_1,\delta_2,\delta_3,\delta_4,0,0,0).$$

We have $x'_{p-d} = x'_3 = 1,\ v_p = v_7 = 60 > I = 23$. Also $(v_p - v_{p-d}) = 60 - 8 = 52 > I$, case 2(a)(ii) suggests us to insert $\delta_d = \delta_4 = 0$, $p = p - 1 = 6$. By inserting a zero, the index does not change. We update the number of deletions: $d = d - 1 = 3$.

$$\tilde{\mathbf{x}} = (1,1,1,\delta_1,\delta_2,\delta_3,0,0,0,0).$$

Since $x'_{p-d} = x'_3 = 1,\ v_p = v_6 = 31 > I = 23$. Also $(v_6 - v_3) = (31 - 4) = 27 > I = 23$,

we follow 2(a)(ii) and insert $\delta_d = \delta_3 = 0$, $p = p-1 = 5$. By inserting a zero, the index does not change. We update the number of deletions: $d = d-1 = 2$.

$$\tilde{\mathbf{x}} = (1,1,1,\delta_1,\delta_2,0,0,0,0,0).$$

And $x'_{p-d} = x'_3 = 1,\ v_p = v_5 = 16 < I = 23$, we follow case 2(b) and shift x'_{p-d} two positiosa to the right, $p = p-1 = 4$. We update the index: $I = I-(v_5 - v_3) = I-12 = 11$.

$$\tilde{\mathbf{x}} = (1,1,\delta_1,\delta_2,1,0,0,0,0,0).$$

Similarly, $x'_{p-d} = x'_2 = 1,\ v_p = 8 < I = 11$, case 2(b) tells us to shift x'_{p-d} two positions to the right, $p = p-1 = 3$. We update the index: $I = I-(v_4 - v_2) = I-6 = 5$.

$$\tilde{\mathbf{x}} = (1,\delta_1,\delta_2,1,1,0,0,0,0,0).$$

We have $x'_{p-d} = x'_1 = 1,\ v_p = 4 < I = 5$, we follow $2(b)$ and shift x'_{p-d} two positions to the right, $p = p-1 = 2$. We update the index: $I = I-(v_3 - v_1) = I-3 = 2$.

$$\tilde{\mathbf{x}} = (\delta_1,\delta_2,1,1,1,0,0,0,0,0).$$

Now, we see that $v_p = v_2 = 2 = I$, therefore $\delta_2 = 1,\ \delta_1 = 0$. We have obtained the original codeword.

$$\tilde{\mathbf{x}} = (0,1,1,1,1,0,0,0,0,0) = \mathbf{x}.$$

Chapter 4

Generalization of Helberg Codes

Non-Binary Helberg Codes

Let $A = (0, 1, \ldots, q-1)$ be a q-ary alphabet and $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in A^n$ a codeword of length n. Following [1], we define the sequence of weights $W(q,d) = w_1, w_2, \ldots$ as:

$$w_i = \begin{cases} 0, & \text{for } i \leq 0 \\ 1 + p \sum_{j=1}^{d} w_{i-j}, & \text{otherwise} \end{cases} \tag{4.1}$$

where d is the number of errors, r is the residue, $p = q-1$. Then the non-binary generalization of Helberg codes is given by:

$$C_n(q,d,m,r) = \left\{ \mathbf{x} \in A^n : \sum_{i=1}^{n} w_i x_i \equiv r \pmod{m} \right\}, \qquad m \geq (w_{n+1}). \tag{4.2}$$

For $d = 3$ and $q = 3$:

$$\begin{aligned} w_{-2} &= 0. \\ w_{-1} &= 0. \\ w_0 &= 0. \\ w_1 &= 1 + 2 \cdot (w_0 + w_{-1} + w_{-2}) = 1 + 2 \cdot (0+0+0) = 1. \\ w_2 &= 1 + 2 \cdot (w_1 + w_0 + w_{-1}) = 1 + 2 \cdot (1+0+0) = 3. \\ w_3 &= 1 + 2 \cdot (w_2 + w_1 + w_0) = 1 + 2 \cdot (3+1+0) = 9. \\ &\vdots \end{aligned}$$

Table 9

Weight w_i for $d = 3$, $q = 3$

i	1	2	3	4	5	6	7	8	9	10
w_i	1	3	9	27	79	231	675	1971	5755	16803

Example 4.1. Suppose we have a quaternary codeword that can correct 3 deletions:

$$\mathbf{y} = (0,1,3,0,2,1,0).$$

Since it is quaternary and can correct up to 3 deletions, we compute its moment using weight w_i from Table 9. So

$$\begin{aligned}\sum_{i=1}^{n} w_i x_i &= 1\cdot 0 + 3\cdot 1 + 9\cdot 3 + 27\cdot 0 + 79\cdot 2 + 231\cdot 1 + 675\cdot 0 \\ &= 419 \equiv 419 \pmod{1971}.\end{aligned}$$

therefore

$$\mathbf{y} \in C_7(3,3,w_8,419).$$

Proof of Multiple Errors Correction

By adapting the argument in [10], we prove that $C_n(q,d,m,r)$ is a d-deletions/insertions error correcting code. Recall that the moment of the codeword $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of length n is defined as $M_{\mathbf{x}} = \sum_{i=1}^{n} w_i x_i$. Let D be a nonempty subset of $S(n) = \{1,2,\ldots,n\}$. We define $\mathbf{x}^{(D)} = (x_{i_1}, x_{i_2}, \ldots, x_{i_{n'}})$ to be a deleted codeword obtained by deleting the element of $\mathbf{x}$ indexed by D where $S' = S(n) - D = \{i_1, i_2, \ldots, i_{n'}\}$. Given two codeword $\mathbf{x}$ and $\mathbf{y}$, we write $\mathbf{x} \equiv \mathbf{y}$ if and only if $M_{\mathbf{x}} \equiv M_{\mathbf{y}}$. If we define $\Delta(\mathbf{x},\mathbf{y}) = M_{\mathbf{x}} - M_{\mathbf{y}}$, then $\mathbf{x} \equiv \mathbf{y}$ is equivalent to $\Delta(\mathbf{x},\mathbf{y}) \equiv 0 \pmod{m}$. The index with respect to $\mathbf{x}$ is $I = M_{\mathbf{x}} - M_{\mathbf{x}^{(D)}}$. Set $n' = n - |D|$.

We start with the following lemma, which allows us to replace the rightmost nonzero bit with the value of 0 in any two codewords that are equivalent and have the same deleted codeword.

Lemma 4.2. *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be two codewords of length n such that* $\mathbf{x} \equiv \mathbf{y}$ *and* $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$ *for some subset D and E of* $\{1,2,\ldots,n\}$ *with* $|D| = |E| \leq d$. *Suppose there exists a positive index L such that* $x_L = y_L > 0$ *and* $x_i = y_i = 0$ *for all* $i > L$. *Then there exist codewords* $\tilde{\mathbf{x}}$ *and* $\tilde{\mathbf{y}}$ *where* $\tilde{x}_i = x_i$, $\tilde{y}_i = y_i$ *for all* $i \neq L$ *and* $\tilde{x}_L = \tilde{y}_L = 0$ *such that* $\tilde{\mathbf{x}} \equiv \tilde{\mathbf{y}}$ *and* $\tilde{\mathbf{x}}^{(\tilde{D})} = \tilde{\mathbf{y}}^{(\tilde{E})}$ *for some sets* $\tilde{D}$ *and* $\tilde{E}$ *having the same size as D and E*.

Proof. Since $x_i - y_i = \tilde{x}_i - \tilde{y}_i$ for all $i = 1,2,\ldots,n$ so that $M_{\mathbf{x}} - M_{\mathbf{y}} = M_{\tilde{\mathbf{x}}} - M_{\tilde{\mathbf{y}}}$. But $\mathbf{x} \equiv \mathbf{y}$, hence $\tilde{\mathbf{x}} \equiv \tilde{\mathbf{y}}$. To complete the proof, we consider 4 cases:

Case I: Assume $L \in D \cap E$. The non-zero bit at the position L is deleted from $\mathbf{x}$ and $\mathbf{y}$ to obtain $\mathbf{x}^{(D)}$ and $\mathbf{y}^{(E)}$. Define $D = \tilde{D}$ and $E = \tilde{E}$. The zero bits $\tilde{x}_L$ and $\tilde{y}_L$ are also deleted from $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$, respectively. Since $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$, it implies $\tilde{\mathbf{x}}^{(\tilde{D})} = \tilde{\mathbf{y}}^{(\tilde{E})}$.

Case II: Assume $L \notin D \cup E$. In this case, the bit at position L appear in $\mathbf{x}^{(D)}$ and $\mathbf{y}^{(E)}$ as the right most non-zero bit, respectively since $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$. But then x_L and y_L are replaced by $\tilde{x}_L$ and $\tilde{y}_L$, respectively. Therefore $\tilde{\mathbf{x}}^{(\tilde{D})} = \tilde{\mathbf{y}}^{(\tilde{E})}$ given that $D = \tilde{D}$ and $E = \tilde{E}$.

Case III: Assume $L \in D \setminus E$. In this case, the bit at position L is deleted from $\mathbf{x}$ to obtain $\mathbf{x}^{(D)}$, but it is not deleted from $\mathbf{y}$ therefore it appears in $\mathbf{y}^{(E)}$. Let z denotes the number of bits to the right of y_L in $\mathbf{y}^{(E)}$ which all be 0 since $y_i = 0$ for $i > L$. Number of bits to the right of y_L that are deleted from $\mathbf{y}$ to obtain $\mathbf{y}^{(E)}$ equal $z' = n - L - z$. Let x_K denotes the rightmost nonzero bit of $\mathbf{x}^{(D)}$. Since $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$, implies $x_K = x_L = y_L$ and the number of zeros to the right of x_K in $\mathbf{x}^{(D)}$ is also equal z. Hence, the number of bits to the right of x_K that are deleted from $\mathbf{x}$ to obtain $\mathbf{x}^{(D)}$ equals $n - K - z$. Let $D' = \{K, K+1, ..., L-1, L+1, ..., L+z'\}$. It follows that $\mathbf{x}^{(D')} = \mathbf{x}^{(D)}$ with $|D'| = |D|$. Since $L \notin D' \cup E$, the result follows from case II where D is replaced by D'.

Case IV: Assume $L \in E \setminus D$. This case is similar to case III.

Theorem 4.3. *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be two codewords of length n such that* $\mathbf{x} \equiv \mathbf{y}$ *and* $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$ *for some subset D and E of* $\{1,2,\ldots,n\}$ *with* $|D| = |E| \leq d$. *Then*

$$0 < |\Delta(\mathbf{x},\mathbf{y})| < m.$$

Proof. As proved in [9], we first prove that $|\Delta(\mathbf{x},\mathbf{y})| < m$ as follow:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) &= M_{\mathbf{x}} - M_{\mathbf{y}} \\
&= \sum_{i\in D} w_i x_i - \sum_{j\in E} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\leq \sum_{i\in D} w_i x_i + \sum_{\substack{k\in S(n') \\ i_k \leq j_k}} (w_{i_k} - w_{j_k}) x_{i_k} + \sum_{\substack{k\in S(n') \\ i_k > j_k}} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\leq \sum_{i\in D} w_i x_i + \sum_{\substack{k\in S(n') \\ i_k > j_k}} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\leq \sum_{i\in D} p w_i + \sum_{\substack{k\in S(n') \\ i_k > j_k}} p(w_{i_k} - w_{j_k}) \\
&= \sum_{i\in D} p w_i + \sum_{\substack{k\in S(n') \\ i_k > j_k}} p w_{i_k} - \sum_{\substack{k\in S(n') \\ i_k > j_k}} p w_{j_k}
\end{aligned}$$

Hence:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) &\leq \sum_{i\in D} pw_i + \sum_{\substack{k\in S(n')\\ i_k > j_k}} pw_{i_k} + \sum_{\substack{k\in S(n')\\ i_k \leq j_k}} pw_{i_k} - \sum_{\substack{k\in S(n')\\ i_k \leq j_k}} pw_{i_k} - \sum_{\substack{k\in S(n')\\ i_k > j_k}} pw_{j_k} \\
&= \sum_{i=1}^{n} pw_i - \sum_{k=1}^{n'} pw_{min(w_{i_k}, w_{j_k})} \\
&\leq \sum_{i=1}^{n} pw_i - \sum_{k=1}^{n'} pw_k \\
&\leq \sum_{i=n'+1}^{n} pw_i \\
&= \sum_{j=1}^{n-n'} pw_{n+1-j} \\
&\leq p \sum_{j=1}^{d} w_{n+1-j} \\
&= w_{n+1} - 1 \\
&\leq m - 1 \\
&< m.
\end{aligned}$$

By symmetry, it also follows that $R(x,y) = M_{\mathbf{y}} - M_{\mathbf{x}} < m$. Hence:

$$|\Delta(\mathbf{x},\mathbf{y})| < m.$$

Next, we will prove that $|\Delta(\mathbf{x},\mathbf{y})| > 0$ by considering 4 different cases. By Lemma 4.2.1 we can assume without loss of generality that there exists $L \in \{1,2,\ldots,n\}$ such that $x_L > y_L$ and $x_i = y_i = 0$ for all $i > L$.

Case I: Assume $L \in D \cap E$. Then $i_k \neq L$ for all $k = 1,2,\ldots,n'$. It follows that:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) = M_{\mathbf{x}} - M_{\mathbf{y}} &= \sum_{i\in D} w_i x_i - \sum_{j\in E} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\
&= w_L x_L + \sum_{\substack{i\in D \\ i\le L-1}} w_i x_i - w_L y_L - \sum_{\substack{j\in E \\ j\le L-1}} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\ge w_L (x_L - y_L) - \sum_{\substack{j\in E \\ j\le L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k}} (w_{i_k} - w_{j_k}) x_{i_k} + \sum_{\substack{k\in S(n') \\ i_k \ge j_k}} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\ge w_L - \sum_{\substack{j\in E \\ j\le L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} (w_{i_k} - w_{j_k}) x_{i_k}
\end{aligned}$$

where $i_k \neq L$ and $x_{i_k} = 0$ if $i_k > L$. Since $x_{i_k} \le p$, we have:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) &\ge w_L - \sum_{\substack{j\in E \\ j\le L-1}} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p(w_{i_k} - w_{j_k}) \\
&= w_L - \sum_{\substack{j\in E \\ j\le L-1}} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p w_{i_k} - \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p w_{j_k}.
\end{aligned}$$

We now add and subtract as follow:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) &\ge w_L - \sum_{\substack{j\in E \\ j\le L-1}} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p w_{i_k} - \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p w_{j_k} \\
&\qquad + \sum_{\substack{k\in S(n') \\ i_k \ge j_k, i_k \le L-1}} p w_{i_k} - \sum_{\substack{k\in S(n') \\ i_k \ge j_k, i_k \le L-1}} p w_{j_k} \\
&\ge w_L - \sum_{j=1}^{L-1} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p w_{i_k} + \sum_{\substack{k\in S(n') \\ i_k \ge j_k, i_k \le L-1}} p w_{j_k} \\
&= w_L - \sum_{j=1}^{L-1} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \le L-1}} p w_{min(i_k, j_k)} \\
&\ge w_L - \sum_{j=1}^{L-1} p w_j + \sum_{k=1}^{min(n', L-1)} p w_k.
\end{aligned}$$

Since $k \leq min(i_k, j_k)$, we have:

$$\begin{aligned}\Delta(\mathbf{x},\mathbf{y}) &= w_L - \sum_{i=min(n',L-1)}^{L-1} pw_i \\ &\leq w_L - \sum_{i=L-d}^{L-1} pw_i \\ &= 1\end{aligned}$$

since $L-d \leq min(n', L-1)$ where $L \leq n = n'+d$ and $d \geq 1$.

Case II: Assume $L \in D \setminus E$. Recall that $x_L > y_L$ and $x_i = y_i = 0$ for $i > L$. We have:

$$\begin{aligned}\Delta(\mathbf{x},\mathbf{y}) &= M_{\mathbf{x}} - M_{\mathbf{y}} \\ &= \sum_{i\in D} w_i x_i - \sum_{j\in E} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\ &= w_L x_L + \sum_{\substack{i\in D \\ i\leq L-1}} w_i x_i - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k}.\end{aligned}$$

We again partition $S(n')$ into those elements k such that $i_k < j_k$ and $i_k \geq j_k$ to obtain

$$\begin{aligned}\Delta(\mathbf{x},\mathbf{y}) &\geq w_L x_L - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k}} (w_{i_k} - w_{j_k}) x_{i_k} + \sum_{\substack{k\in S(n') \\ i_k \geq j_k}} (w_{i_k} - w_{j_k}) x_{i_k} \\ &\geq w_l x_L - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k}} (w_{i_k} - w_{j_k}) x_{i_k} \\ &\geq w_L - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} (w_{i_k} - w_{j_k}) x_{i_k}.\end{aligned}$$

Now, using the same reasoning as in Case I, $\Delta(\mathbf{x},\mathbf{y}) \geq 1$.

Case III: Assume $L \in E \setminus D$. The argument in this case is the same as in Case II, through switching the roles of D and E.

Case IV: Assume $L \notin D \cup E$. Then $i_K = L$ for some $i_K \in S'$. We claim that $j_K \leq i_K - 1$.

Since $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$, it follows that $x_{i_K} = y_{i_K}$. On the other hand, we have $y_{i_K} < x_{i_K}$ and $y_i = 0$ for all $i \geq L = i_K$. Therefore $j_K \leq i_K - 1$.

We now proceed as previously:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) &= M_{\mathbf{x}} - M_{\mathbf{y}} \\
&= \sum_{i\in D} w_i x_i - \sum_{j\in E} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\
&= \sum_{i\in D} w_i x_i - \sum_{j\in E} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\
&= \sum_{\substack{i\in D \\ i\leq L-1}} w_i x_i - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{k=1}^{n'} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\geq - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k}} (w_{i_k} - w_{j_k}) x_{i_k} + \sum_{\substack{k\in S(n') \\ i_k \geq j_k}} (w_{i_k} - w_{j_k}) x_{i_k} \\
&\geq - \sum_{\substack{j\in E \\ j\leq L-1}} w_j y_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} (w_{i_k} - w_{j_k}) x_{i_k} + (w_{i_K} - w_{i_K}) x_{i_K}.
\end{aligned}$$

Since $x_i \leq p$, we have:

$$\begin{aligned}
\Delta(\mathbf{x},\mathbf{y}) &\geq w_L - p w_{j_K} - \sum_{\substack{j\in E \\ j\leq L-1}} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} p(w_{i_k} - w_{j_k}) \\
&= w_L - p w_{j_K} - \sum_{\substack{i\in E \\ j\leq L-1}} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} p w_{i_k} - \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} p w_{j_k} \\
&= w_L - p w_{j_K} - \sum_{\substack{i\in E \\ j\leq L-1}} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} p w_{i_k} - \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} p w_{j_k} \\
&\qquad + \sum_{\substack{k\in S(n') \\ i_k \geq j_k, i_k \leq L-1}} p w_{j_k} - \sum_{\substack{k\in S(n') \\ i_k \geq j_k, i_k \leq L-1}} p w_{j_k} \\
&= w_L - \sum_{j=1}^{L-1} p w_j + \sum_{\substack{k\in S(n') \\ i_k < j_k, i_k \leq L-1}} p w_{i_k} + \sum_{\substack{k\in S(n') \\ i_k \geq j_k, i_k \leq L-1}} p w_{j_k}.
\end{aligned}$$

The rest of the proof is the same as that in Case I. Therefore, $\Delta(\mathbf{x},\mathbf{y}) \geq 1$. Hence:

$$0 < |\Delta(\mathbf{x},\mathbf{y})| < m$$

as desired.

Theorem 4.4. *The code $C_n(q,d,m,r)$ is a d-deletion/insertion correcting code.*

Proof. Assume that $C_n(q,d,m,r)$ is not capable of correcting up to d deletions. Then there exist at least two codewords $\mathbf{x},\mathbf{y} \in C_n(q,d,m,r)$ and subsets D and E with $|D| = |E| \leq d$ such that $\mathbf{x}^{(D)} = \mathbf{y}^{(E)}$. By Theorem 4.2.2, we have $0 < |\Delta(\mathbf{x},\mathbf{y})| < m$, which means $\mathbf{x} \not\equiv \mathbf{y}$, a contradiction since $\mathbf{x}$ and $\mathbf{y}$ are from the same codebook. Thus, $C_n(q,d,m,r)$ is capable of correcting up to d deletions. So, due to result of Leveinstein [15], it can correct up to d insertion/deletion errors as well.

Example 4.5. By brute force, we generated all possible codewords of length 8 of generalized Helberg codes with 4 symbols, following formula (4.1) with $d = 3$. We then assigned each codeword to different codebooks using formula (4.2) above. As result, the codebook with residue $r = 506$ contains:

$$C_8(4,3,61705,506) = \{(1,3,3,3,1,0,0,0),(0,0,0,0,2,0,0,0),$$
$$(2,3,3,3,1,3,3,3),(1,0,0,0,2,3,3,3)\}$$

where $m = w_9 = 61705$ is the modulus. These codewords are able to correct up to 3 insertion/deletion errors. We will use one of these codewords in the next example to demonstrate some properties of non-binary Helberg codes.

Properties of Non-binary Helberg Codes

Useful lemmas. The following lemmas will allows us to determine the moment of the original codeword based on the residues and the moment of the deleted codeword. Recall

that w_i is the weight of non-binary Helberg codes at position i.

Lemma 4.6. *For $d \geq 2$,*

$$\sum_{i=1}^{n} w_i = \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right) - n}{pd-1}.$$

Proof. We use induction to prove this lemma as follows:

For $n = 1$. Since $w_1 = 1$ for all $s \geq 2$, we have:

$$\begin{aligned} w_1 &= \frac{p\left(d(w_1) + (d-1)w_0 + (d-2)w_{-1} + \cdots\right) - 1}{pd-1} \\ &= \frac{pdw_1 - 1}{pd-1} \\ &= \frac{pd-1}{pd-1} = 1, \end{aligned}$$

where the second equality follows Equation (4.1), and the last equality follows from the fact that

$w_1 = 1$.

The statement holds for $n = 1$. Assume that it holds for n, i.e.,

$$\sum_{i=1}^{n} w_i = \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right) - n}{pd-1}.$$

Then for $n+1$, we have:

$$\begin{aligned}
\sum_{i=1}^{n+1} w_i &= \sum_{i=1}^{n} w_i + w_{n+1} \\
&= \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right)-n}{pd-1} + w_{n+1} \\
&= \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right)-n}{pd-1} + \frac{pd(w_{n+1})-w_{n+1}}{pd-1} \\
&= \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right)-n}{pd-1} + \frac{pd(w_{n+1})-\left(1+p\sum_{i=1}^{d} w_{n+1-i}\right)}{pd-1} \\
&= \frac{pd(w_{n+1})-(n+1)+p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right)-p\left(\sum_{i=1}^{d} w_{n+1-i}\right)}{pd-1} \\
&= \frac{pd(w_{n+1})-(n+1)+p\left(\sum_{i=0}^{d-1}(d-i-1)w_{n-i}\right)}{pd-1} \\
&= \frac{p\left(\sum_{i=0}^{d-1}(d-i-1)w_{n-i}+dw_{n+1}\right)-(n+1)}{pd-1} \\
&= \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n+1-i}\right)-(n+1)}{pd-1}.
\end{aligned}$$

Lemma 4.7. *For $d \geq 2$,*

$$\sum_{i=0}^{n} w_i < \frac{d}{pd-1} w_{n+1}.$$

Proof. Following the previous lemma:

$$\begin{aligned}\sum_{i=1}^{n} w_i &= \frac{p\left(\sum_{i=0}^{d-1}(d-i)w_{n-i}\right)-n}{pd-1} \\ &\leq \frac{p\left(\sum_{i=0}^{d-1} dw_{n-i}\right)-n}{pd-1} \\ &= \frac{pd\left(\sum_{i=0}^{d-1} w_{n-i}\right)-n}{pd-1} \\ &= \frac{d\left(w_{n+1}-1\right)-n}{pd-1} \\ &< \frac{d}{pd-1} w_{n+1}.\end{aligned}$$

Lemma 4.8. *If $M_{\mathbf{x}'} > r$, then $M_{\mathbf{x}} = r + m$. Otherwise, if $M_{\mathbf{x}'} < r$, then $M_{\mathbf{x}} = r$ (Given that r is the residue and m is the modulus.)*

Proof. By definition,

$$M_{\mathbf{x}} = p\sum_{i=1}^{n} w_i < \frac{pd}{pd-1} w_{n+1}$$

where the inequality follows from the previous lemma.

Since $d \geq 2,\ p \geq 2$ implies

$$\begin{aligned} pd &\geq 2 \\ 2pd - pd &\geq 2 \\ 2pd - 2 &\geq pd \\ 2 &\geq \frac{pd}{pd-1}. \end{aligned}$$

It follows that

$$M_{\mathbf{x}} < \frac{pd}{pd-1} w_{n+1} \leq 2w_{n+1}$$

which is the same as saying

$$M_{\mathbf{x}} < 2m.$$

Recall that $M_{\mathbf{x}} \equiv r \pmod{m}$, therefore $M_{\mathbf{x}} = r + tm$ where $t = 0, 1, 2, \ldots$.

Since $M_{\mathbf{x}} < 2m$, implies $M_{\mathbf{x}} = r$ or $M_{\mathbf{x}} = r + m$. We consider two cases:

- Case I: $M_{\mathbf{x}'} > r$. Since $M_{\mathbf{x}} > M_{\mathbf{x}'} > r$, therefore $M_{\mathbf{x}} = r + m$.

- Case II: $M_{\mathbf{x}'} < r$, then $M_{\mathbf{x}} = r$ or $M_{\mathbf{x}} = r + m$.

  Suppose $M_{\mathbf{x}} = r + m$. But by definition, we have:

$$w_n = 1 + p \sum_{j=1}^{d} w_{n-j}$$

  Let Δ_x be the maximum change in the moment from $M_{\mathbf{x}}$ to $M_{\mathbf{x}'}$ so that

$$\begin{aligned} \Delta_{\mathbf{x}} &= p \sum_{j=1}^{d} w_{n-j} \\ &< w_n \\ &< w_{n+1} = m. \end{aligned}$$

  The deleted moment is given as follows:

$$\begin{aligned} M_{\mathbf{x}'} &= M_{\mathbf{x}} - \Delta_{\mathbf{x}} \\ &= r + m - \Delta_x > r \end{aligned}$$

  since $m > \Delta_{\mathbf{x}}$. But this is a contradiction since $M_{\mathbf{x}'}$ is supposed to be less than r. Therefore $M_{\mathbf{x}} = r$.

This completes the proof.

Example 4.9. Let $\mathbf{x} = (2,3,3,3,1,3,3,3)$, $\mathbf{x} \in C_8(4,3,61705,506)$ as stated in the previous example. Let $\mathbf{x}'$ be what remains after deleting the first and last bits of $\mathbf{x}$. Let's assume

we received a codeword $\mathbf{x}' = (3,3,3,1,3,3)$, and pretend we do not know $\mathbf{x}$ but we know $\mathbf{x}'$ comes from $C_8(4,3,61705,506)$. What is the original moment $M_{\mathbf{x}}$?

Since the length of $\mathbf{x}'$ is 6 and it comes from a codebook with length 8, we know there were deletion errors. This codebook is able to correct up to 3 deletions. The weights associated with this code is given in Table 10.

Table 10

Weight w_i for $d = 3$, $q = 4$

i	1	2	3	4	5	6	7	8	9	10
w_i	1	4	16	64	253	1000	3952	15616	61705	243820

We have $r = 506$. Since $n = 8$, therefore $m = w_{n+1} = w_9 = 61705$. The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= \sum_{i=1}^{n} w_i x_i' \\ &= 1\cdot 3 + 4\cdot 3 + 16\cdot 3 + 64\cdot 1 + 253\cdot 3 + 1000\cdot 3 \\ &= 3886 \equiv 3886 \pmod{61705} \end{aligned}$$

Since $M_{\mathbf{x}'} > r$, Lemma 4.8 suggests that

$$\begin{aligned} M_{\mathbf{x}} &= m + r \\ &= 61705 + 506 \\ &= 62211. \end{aligned}$$

We can check the result by computing the moment of $\mathbf{x}$ as follows:

$$\begin{aligned} M_{\mathbf{x}} &= \sum_{i=1}^{n} w_i x_i \\ &= 1\cdot 2 + 4\cdot 3 + 16\cdot 3 + 64\cdot 3 + 253\cdot 1 + 1000\cdot 3 + 3952\cdot 3 + 15616\cdot 3 \\ &= 62211. \end{aligned}$$

Code cardinality. Recall that the size of the largest codebook in terms of code length is defined by

$$N_n(q,d) = \max\{|C_n(q,d,w_{n+1},r)| : r = 0,1,\ldots,w_{n+1}-1\}$$

We also denote $R_n(q,d)$ to be the set contains values of r such that

$$|C_n(q,d,w_{n+1},r)| = N_n(q,d).$$

Table 11 and 12 give values for ternary 2-deletion Helberg codes and quaternary 2-deletion Helberg codes, respectively.

Table 11

Ternary 2-deletion Helberg Codes: Values of $N_n(3,2)$ and $R_n(3,2)$

n	$N_n(3,2)$	$R_n(3,2)$
1	1	0,1,2
2	1	0,1,2,3,4,5,6,7,8
3	2	0,1,2,3,4,5,6,7,25,26,50,51
4	2	0,1,2,3,4,5,6,7,25,26,50,51
5	3	0,25
6	4	25,50
7	4	24,25,50,69,70,71,72,73,74,75,94,119,138,139 140,141,142,143,144,163,188,189,542,567,1059,1084
8	5	24,25,49,50,69,70,71,73,73,74,188,189,213,214,377, 378, 402,403,517,518,519,520,521,522,541,542,566,567
9	7	541,542,566,567,1058,1059,1083,1084
10	8	517,518,519,520,521,541,542,566,567,1437,1482, 1483,1484,1485,1486,1487,1551,1552,1553,1554,1555, 1556,1601,2850,2895,2896,2897,2898,2899,2900, 2964,2965,2966,2967,2968,2969,3014,3884,3885, 3909,3910,3930,3931,3932,3933,3934

Table 12

Quaternary 2-deletion Helberg Codes: Values of $N_n(4,2)$ and $R_n(4,2)$

n	$N_n(4,2)$	$R_n(4,2)$
1	1	0,1,2,3
2	1	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
3	2	0,1,2
4	2	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,61 62,63,122,123,124,183,184,185
5	3	0,1,61,62
6	4	61,62,122,123,183,184
7	5	61,880
8	6	61,122,183,880,941,1760,1821,2640,2701,3398,3459,3520

Since $N_n(q,d)$ is the size of the largest codebook, the information rate of the largest generalized Helberg non-binary code is defined by

$$I_n(q,d) = \frac{\log_q N_n(q,d)}{n}.$$

Table 13 gives the information rate for binary 2-deletion, binary 3-deletion, ternary 2-deletion, and quaternary 2-deletion Helberg codes. We have seen that binary Helberg codes suffer from low cardinality. In this table, it is clear that generalized Helberg codes have lower information rate than binary Helberg codes.

Table 13

Information Rate

n	$I_n(2,2)$	$I_n(2,3)$	$I_n(3,2)$	$I_n(4,2)$
4	0.25	0.25	0.158	0.125
5	0.2	0.2	0.2	0.158
6	0.264	0.167	0.210	0.167
7	0.286	0.143	0.180	0.166
8	0.290	0.198	0.183	0.162
9	0.287	0.222	0.197	-
10	0.3	0.2	0.189	-
11	0.288	0.211	-	-
12	0.288	0.215	-	-
13	0.300	-	-	-
14	0.298	-	-	-
15	0.297	-	-	-
16	0.307	-	-	-

Also notice that there is no known lower bound or upper bound for Helberg codes. There are only known lower bounds and upper bounds for the largest possible codebook. Let the asymptotic lower and upper bounds for the cardinality of the largest codebook that can correct up to d deletions be denoted by $L_n(q,d)$ and $U_n(q,d)$, respectively. We define

$$L_n(q,d) = \frac{(d!)^2 q^{n+d}}{(q-1)^{2d} n^{2d}},$$

$$U_n(q,d) = \frac{d! q^n}{(q-1)^d n^d},$$

and let $M_n(q,d)$ denote the size of the largest possible q-ary code of length n that can correct

up to d deletions. As in [16], Levenshtein proved that

$$L_n(q,d) \lesssim M_n(q,d) \lesssim U_n(q,d)$$

as $n \to \infty$, where $f_n \lesssim g_n$ is defined to be $\lim_{n\to\infty} f_n/g_n = 1$.

Chapter 5

Decoding Algorithm of Generalized Helberg Codes

We have seen how non-binary Helberg codes are able to correct multiple errors. The natural question to ask is how to correct, or how to decode them. Our situation here is that we have received an deleted codeword, which we assume to come from the codebook $C_n(q,d,m,r)$. The available information includes the moment of the deleted codeword $M_{\mathbf{x}'}$, the number of deletions d, and the residue r. We adapt the procedure we have applied to binary Helberg codes. That is, we determine the moment of the original codeword $M_{\mathbf{x}}$ to figure out the index I. Then, we assume the deleted symbols are at the right end and decide whether to insert a value for the right most unknown symbol or to shift it to the left.

The first section will describe an algorithm to correct one deletion after which we will provide a recursive algorithm to correct two or more deletions.

Decoding One Deletion

As we have seen in one deletion method for binary Helberg codes, the Deletion Decoding Algorithm is essentially an exhaustive search. Let the original codeword is denoted by $\mathbf{x} \in C_n(q,d,m,r)$ and the deleted codeword is denoted by $\mathbf{x}'$ which obtained from $\mathbf{x}$ by deleting one symbol. Assume the deleted element is at the right end, representative by δ. Initialize $P = n$, so that we have:

$$\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{P-1}),$$
$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{P-1}, \delta).$$

We then try to recover the original codeword $\mathbf{x}$ so that $M_{\tilde{\mathbf{x}}} = M_{\mathbf{x}}$ by either inserting a value for δ or shifting it to the left of x'_{P-1}. The decision is made by using following conditions:

Algorithm DN1 (Decode Non-binary One Deletion). Define: $\sigma \in \{0,\ 1,\ \ldots,\ p\}$

1. If $I = 0$, insert 0 for δ, stop.

2. If $I = \sigma \cdot w_P$, insert σ for δ, stop.

3. Otherwise, we shift x'_{P-1} to the right of δ, and update the index

$$I = I - x'_{P-1} \cdot (w_P - w_{P-1}).$$

Repeat through steps 1 to 3.

Since this algorithm is an exhaustive search, it will correctly decode $\mathbf{x}'$. In the following example, we will assume that our codes come from codebook that capable of correcting two errors, i.e., $d = 2$. Otherwise, if $d = 1$, then non-binary Tenengol'ts decoding should be used.

Example 5.1. Suppose the ternary codeword $\mathbf{x} = (2,2,2,0,2,2,1,2) \in C_8(3,2,w_9,24)$ was used to transmit and $\mathbf{x}' = (2,2,2,0,2,1,2)$ was received so that one deletion had occurred. We want to decode $\mathbf{x}'$ to recover $\mathbf{x}$.

The weight corresponding to this codebook is defined as $w_i = 1 + 2 \cdot (w_{i-1} + w_{i-2})$. The first 10 weights are given in Table 14.

Table 14

Weight w_i for $d = 2$, $q = 3$

i	1	2	3	4	5	6	7	8	9	10
w_i	1	3	9	25	69	189	517	1413	3861	10549

The modulus is

$$w_9 = 3861.$$

The residue is

$$r = 24.$$

The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= \sum w_i x_i' \\ &= 2\cdot 1 + 2\cdot 3 + 2\cdot 9 + 0\cdot 25 + 2\cdot 69 + 1\cdot 189 + 2\cdot 517 \\ &= 1387. \end{aligned}$$

Since $M_{\mathbf{x}'} > r$, we follow Lemma 4.8 and calculate the original moment as follows:

$$\begin{aligned} M_{\mathbf{x}} &= m + r \\ &= 3861 + 24 \\ &= 3885. \end{aligned}$$

Therefore the index is

$$\begin{aligned} I &= 3885 - 1387 \\ &= 2498. \end{aligned}$$

Assume the deleted element is at the right end. We have

$$\tilde{\mathbf{x}} = (2,2,2,0,2,1,2,\delta)$$

According to the Algorithm DN1, since $I \neq \sigma \cdot w_8$ for all $\sigma \in \{0,1,2\}$, we shift $x_7' = 2$ to the right of δ. We should update the position $P \to P - 1 = 7$, and update the index $I \to I - x_7'(w_8 - w_7) = 706$.

$$\tilde{\mathbf{x}} = (2,\ 2,\ 2,\ 0,\ 2,\ 1,\ \delta,\ 2)$$

Similarly, since $I \neq \sigma \cdot w_7$ for all $\sigma \in \{0,1,2\}$, we shift x_6' to the right of δ. We update

$P \to P-1 = 6$, and we update $I \to I - x'_6 \cdot (w_7 - w_6) = 378$. Then we have

$$\tilde{\mathbf{x}} = (2,\ 2,\ 2,\ 0,\ 2,\ \delta,\ 1,\ 2)$$

We now find that $I = \sigma \cdot w_6$ for $\sigma = 2$. We set $\delta = 2$. The original codeword is

$$\tilde{\mathbf{x}} = (2,\ 2,\ 2,\ 0,\ 2,\ 2,\ 1,\ 2) = \mathbf{x}$$

Decoding Two Deletions

Supposes $\mathbf{x}'$ is obtained from $\mathbf{x} \in C_n(q,2,m,r)$ after deleting two symbols. Assume the deleted symbols are at the right end, represented by δ_1 and δ_2. Initialize $P = n$, so that we have

$$\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{P-2})$$
$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{P-2}, \delta_1, \delta_2)$$

We then try to insert a value for δ_2 and reduce it to one deletion algorithm DN1, or shift x'_{P-2} to the right of δ_2. The following conditions describe when each action is executed:

Algorithm DN2 (Decoding Non-binary Two Deletions). If

$$I = \sigma_1 \cdot w_{P-1} + \sigma_2 \cdot w_P$$

for a set of value $\sigma_1,\ \sigma_2 \in \{0,1,\ldots,p\}$, then δ_1 and δ_2 are in their correct positions. To decode, set $\delta_1 = \sigma_1$, $\delta_2 = \sigma_2$. Otherwise, we assume at least one of the δs is NOT in its correct position. Define:

$$\sigma_{max} = \max\{\sigma : \sigma \cdot (w_P - w_{P-2}) < I,\ \sigma = 0,1,\ldots,p\}$$

Then:

1. For $w_P > I$:

(a) If $x'_{P-2} = 0$, then shift x'_{P-2} to the right δ_2, update $P \to P-1$. Repeat the algorithm.

(b) If $x'_{P-2} \neq 1$ and

i. $I < (w_P - w_{P-2})$, then insert $\delta_2 = 0$ and apply Algorithm DN1.

ii. $I \geq (w_P - w_{P-2})$, then shift x'_{P-2} to the right of δ_2, update

$$I \to I - x'_{P-2} \cdot (w_P - w_{P-2}),$$

and $P \to P-1$. Repeat the algorithm.

2. For $w_P < I$:

(a) If $x'_{P-2} > \sigma_{max}$, then insert $\delta_2 = \sigma_{max}$, update $I \to I - \sigma_{max} \cdot w_P$, and apply Algorithm DN1.

(b) If $x'_{P-2} < \sigma_{max}$ and

i. $\sigma_{max} \cdot w_P \leq I$, then insert $\delta_2 = \sigma_{max}$, update $I \to I - \sigma_{max} \cdot w_P$, and apply Algorithm DN1.

ii. $\sigma_{max} \cdot w_P > I$, then shift x'_{P-2} to the right of δ_2, update $P \to P-1$. Repeat the algorithm.

(c) If $x'_{P-2} = \sigma_{max}$, then shift x'_{P-2} to the right of δ_2, update $I \to I - \sigma_{max} \cdot w_P$ and $P \to P-1$. Repeat the algorithm.

Proof. We want to prove that the conditions 1. and 2. in DN2 give a correct decoding of $\mathbf{x}'$.

1. Suppose $w_P > I$. We consider two cases:

(a) $x'_{P-2} = 0$. There are two situations whether δ_2 is in the correct position as the right most deleted symbol or not. If δ_2 is in the correct position then it has to be

a 0; otherwise, if $\delta_2 \geq 1$, then the moment of $\tilde{\mathbf{x}}$ will exceed $\mathbf{x}$, regardless of the position and value of δ_1:

$$\begin{aligned} \min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + \delta_2 \cdot w_P \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}} \end{aligned}$$

But observe that inserting $\delta_2 = 0$ is equivalent to shifting x'_{P-2} to the right of δ_2. The other situation is δ_2 is NOT in the correct position. This forces us to shift x'_{p-2} to the right of δ_2.

(b) $x'_{P-2} \neq 0$. We consider two sub-cases:

i. $I < (w_P - w_{P-2})$. Suppose δ_2 is NOT in the correct position, therefore we are forced to shift x'_{P-2} to the right of δ_2. But then the increase in moment of $\tilde{\mathbf{x}}$ will exceed $\mathbf{x}$:

$$\begin{aligned} \min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + x'_{P-2} \cdot (w_P - w_{P-2}) \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}} \end{aligned}$$

Hence, δ_2 is in the correct position, and $\delta_2 = 0$ since $w_P > I$.

ii. $I \geq (w_P - w_{P-2})$. Suppose δ_2 is in the correct position. Since $w_P > I$, δ_2 has to be a 0 and so

$$\tilde{\mathbf{x}} = (x'_1,\ x'_2,\ \ldots,\ x'_{P-2},\ \delta_1,\ 0)$$

But then we will not albe to shrink the index:

$$\begin{aligned}\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + p \cdot w_{P-1} \\ &= M_{\mathbf{x}'} + w_P - p \cdot w_{P-2} - 1 \\ &< M_{\mathbf{x}'} + w_P - w_{P-2} \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}}\end{aligned}$$

Hence, δ_2 is NOT in the correct position. Therefore we are forced to shift x'_{P-2} to the right of δ_2.

2. Suppose $w_P < I$. First, we will prove that if δ_2 is in the correct position then $\delta_2 = \sigma_{max}$. Otherwise, we will not able to shrink the index:
   If $\delta_2 = \sigma < \sigma_{max}$, then:

$$\begin{aligned}\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + p \cdot w_{P-1} + \sigma \cdot w_P \\ &= M_{\mathbf{x}'} + (w_P - p \cdot w_{P-2} - 1) + \sigma \cdot w_P \\ &= M_{\mathbf{x}'} + (\sigma + 1) \cdot w_P - p \cdot w_{P-2} - 1 \\ &< M_{\mathbf{x}'} + \sigma_{max} \cdot (w_P - w_{P-2}) \\ &< M_{\mathbf{x}'} + I = M_{\mathbf{x}}\end{aligned}$$

   If $\delta_2 = \sigma > \sigma_{max}$, then:

$$\begin{aligned}\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + \sigma \cdot w_P \\ &> M_{\mathbf{x}'} + \sigma \cdot (w_P - w_{P-2}) \\ &\geq M_{\mathbf{x}'} + I = M_{\mathbf{x}}\end{aligned}$$

   Next, we consider three cases:

   (a) $x'_{P-2} > \sigma_{max}$. Suppose δ_2 is NOT in the correct position therefore we shift x'_{P-2}

to the right of δ_2 and we will exceed the index:

$$\begin{aligned}\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + x'_{P-2} \cdot (w_P - w_{P-2}) \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}}.\end{aligned}$$

Hence, δ_2 is in the correct position and as we proved previously, $\delta_2 = \sigma_{max}$.

(b) $x'_{P-2} < \sigma_{max}$. Considering two sub-cases:

i. $\sigma_{max} \cdot w_P \leq I$. Suppose δ_2 is NOT in the correct position therefore we shift x'_{P-2} to the right of δ_2 and we will not able to shrink the index:

$$\begin{aligned}\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + p \cdot w_{P-1} + x'_{P-2} \cdot (w_P - w_{P-2}) \\ &= M_{\mathbf{x}'} + (w_P - p \cdot w_{P-2} - 1) + x'_{P-2} \cdot (w_P - w_{P-2}) \\ &< M_{\mathbf{x}'} + (1 + x'_{P-2}) \cdot w_P \\ &\leq M_{\mathbf{x}'} + \sigma_{max} \cdot w_P \\ &\leq M_{\mathbf{x}'} + I = M_{\mathbf{x}}\end{aligned}$$

Hence, δ_2 is in the correct position. Therefore, $\delta_2 = \sigma_{max}$.

ii. $\sigma_{max} \cdot w_P > I$. Suppose δ_2 is in the correct position, then as we have seen above, $\delta_2 = \sigma_{max}$, so that we will exceed the index:

$$\begin{aligned}\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + \sigma_{max} \cdot w_P \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}}\end{aligned}$$

Hence, δ_2 is NOT in the correct position. We are forced to shift x'_{P-2} to the right of δ_2.

(c) $x'_{P-2} = \sigma_{max}$. In this case, if δ_2 is in the correct position then $\delta_2 = \sigma_{max}$. but observe that this same result can be obtained by shifting x'_{P-2} to the right of δ_2. Thus, we choose to shift instead. This completes the proof.

Example 5.2. Suppose the quaternary codedword $\mathbf{x} = (3,0,0,1,3,3,2,3) \in C_8(4,2,w_9,62)$ was used to transmit and $\mathbf{x}' = (3,0,0,3,3,3)$ was received so that two deletions had occurred. We want to decode $\mathbf{x}'$ to recover $\mathbf{x}$.

The weight corresponding to this codebook is defined as $w_i = 1 + 3(w_{i-1} + w_{i-2})$. The first 10 weights are given in Table 15.

Table 15

Weight w_i for $d = 2$, $q = 4$

i	1	2	3	4	5	6	7	8	9	10
w_i	1	4	16	61	232	880	3337	12652	47968	181861

The modulus is

$$m = w_{n+1} = w_9 = 47968.$$

The residue is

$$r = 62.$$

The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= \sum w_i x_i' \\ &= 3 \cdot 1 + 0 \cdot 4 + 0 \cdot 16 + 3 \cdot 61 + 3 \cdot 232 + 3 \cdot 880 \\ &= 3522. \end{aligned}$$

Since $M_{\mathbf{x}'} > r$, the Lemma 4.8 suggests the original moment is

$$\begin{aligned} M_{\mathbf{x}} &= m + r \\ &= 47968 + 62 \\ &= 48030 \end{aligned}$$

Therefore the index is

$$\begin{aligned} I &= 48030 - 3522 \\ &= 44508 \end{aligned}$$

Assume the deleted symbols are at the right end. We have

$$\tilde{\mathbf{x}} = (3,0,0,3,3,3,\delta_1,\delta_2)$$

As $w_8 = 12652 < I$, we compute $\sigma_{max} = 3$. Since $x'_6 = 3 = \sigma_{max}$, we follow case $2(c)$ and shift x'_6 to the right of δ_2. We update the index $I \to I - x'_6 \cdot (w_8 - w_6) = 9192$ so that

$$\tilde{\mathbf{x}} = (3,\ 0,\ 0,\ 3,\ 3,\ \delta_1,\ \delta_2,\ 3)$$

Next, as $w_7 = 3337 < I$, we compute $\sigma_{max} = 2$. Since $x'_5 = 3 > \sigma_{max}$, case $2(a)$ suggests us to insert $\delta_2 = \sigma_{max} = 2$ and update the index $I \to I - \sigma_{max} \cdot w_7 = 2518$. It yields

$$\tilde{\mathbf{x}} = (3,\ 0,\ 0,\ 3,\ 3,\ \delta_1,\ 2,\ 3)$$

It remains to apply Algorithm DN1 on the truncated codeword $\tilde{\mathbf{x}} = (3,\ 0,\ 0,\ 3,\ 3,\ \delta_1)$ with $I = 2518$ to determine δ_1. Following example 5.1.1, we find that $\delta_1 = 1$ should be insert at position 4. Hence, our final decoding codeword is

$$\tilde{\mathbf{x}} = (3,\ 0,\ 0,\ 1,\ 3,\ 3,\ 2,\ 3) = \mathbf{x}$$

Decoding Multiple Deletions

Suppose we have a deleted generalized Helberg codeword with length $n-d$, where d is integer. Assume our deleted symbols are at the right end, represented by $\delta_1,\ldots,\delta_d$. Initialize $P=n$, so that we have:

$$\mathbf{x}' = (x'_1, x'_2, x'_3, \ldots, x'_{P-d}).$$
$$\tilde{\mathbf{x}} = (x'_1, x'_2, x'_3, \ldots, x'_{P-d}, \delta_{P-d+1}, \delta_{P-d+2}, \ldots, \delta_{P-1}, \delta_P).$$

Recall $M_{\mathbf{x}}$, $M_{\mathbf{x}'}$, $M_{\tilde{\mathbf{x}}}$ are the moment of original codeword, the deleted codeword, and the current working codeword contains δ_i's, respectively. We determine the values of δ_i's and their positions by using following conditions:

Algorithm DNM (Decoding Non-binary Multiple Deletions). If

$$I = \sigma_1 \cdot w_{P-d+1} + \sigma_2 \cdot w_{P-d+2} + \ldots + \sigma_{d-1} \cdot w_{P-1} + \sigma_d \cdot w_P$$

for a set of value $\sigma_1,\ \ldots,\ \sigma_d\ \in\ \{0,1,\ldots,p\}$, then $\delta_{P-d+1},\ldots,\delta_P$ are in their correct positions. To decode, set $\delta_{P-d+1}=\sigma_1,\ldots,\delta_P=\sigma_d$. Otherwise, we assume at least one of the δs is NOT in its correct position. Define:

$$\sigma_{max} = \max\{\sigma : \sigma \cdot (w_P - w_{P-d}) < I,\ \sigma = 0,\ 1,\ 2,\ \ldots,\ p\}$$

Then:

1. For $w_P > I$:
   (a) If $x'_{P-d}=0$, then shift x'_{P-d} to the right δ_P, update $P \to P-1$. Repeat the algorithm.
   (b) If $x'_{P-d} \neq 0$ and
      i. $I < (w_P - w_{P-d})$, then insert $\delta_P = 0$ and repeat the algorithm for $d-1$

errors.

ii. $I \geq (w_P - w_{P-d})$, then shift x'_{P-d} to the right of δ_P, update

$$I \to I - x'_{P-d} \cdot (w_P - w_{P-d}),$$

and $P \to P-1$. Repeat the algorithm.

2. For $w_P < I$:

(a) If $x'_{P-d} > \sigma_{max}$, then insert $\delta_P = \sigma_{max}$, update $I \to I - \sigma_{max} \cdot w_P$, and repeat the algorithm for $d-1$ errors.

(b) If $x'_{P-d} < \sigma_{max}$ and

i. $\sigma_{max} \cdot w_P \leq I$, then insert $\delta_P = \sigma_{max}$, update $I \to I - \sigma_{max} \cdot w_P$, and repeat the algorithm for $d-1$ errors.

ii. $\sigma_{max} \cdot w_P > I$, then shift x'_{P-d} to the right of δ_P, update $P \to P-1$. Repeat the algorithm.

(c) If $x'_{P-d} = \sigma_{max}$, then shift x'_{P-d} to the right of δ_P, update

$$I \to I - \sigma_{max} \cdot (w_P - w_{P-2}),$$

and $P \to P-1$. Repeat the algorithm.

Proof. We will argue by contradiction, similar to the proof of the DM2 Algorithm.

1. Suppose $w_P > I$. Let consider two cases:

(a) $x'_{P-d} = 0$. There are two situations: First, we suppose that δ_P is in the correct position, therefore $\delta_P = 0$ since $w_P > I$; otherwise, the moment increase in $M_{\tilde{\mathbf{x}}}$ will exceed the index. But observe that inserting $\delta_P = 0$ is equivalent to shifting x'_{P-d} to the right of δ_P. The other situation is δ_P is NOT in the correct position. In this case, we are forced to shift x'_{P-d} to the right of δ_P.

(b) $x'_{P-d} \neq 0$. Let consider two sub-cases:

i. $I < (w_P - w_{P-d})$. Suppose δ_P is NOT in the correct position. We shift x'_{P-d} to the right of δ_P and we will exceed the index:

$$\begin{aligned}\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + x'_{P-d} \cdot (w_P - w_{P-d}) \\ &> M_{\mathbf{x}'} + I = M_{\mathbf{x}}.\end{aligned}$$

Hence, δ_P is in the correct position, and $\delta_P = 0$ since $w_P > I$.

ii. $I \geq (w_P - w_{P-d})$. Suppose δ_P is in the correct position, therefore $\delta_P = 0$ since $w_P > I$.

$$\tilde{\mathbf{x}} = (x'_1,\, x'_2,\, \ldots,\, x'_{P-d},\, \delta_{P-d+1},\, \delta_{P-d+2},\, \ldots,\, \delta_{P-1},\, \delta_P = 0).$$

Then we are not able to shrink the index:

$$\begin{aligned}\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + p \cdot (w_{P-d+1} + \ldots + w_{P-1}) \\ &= M_{\mathbf{x}'} + w_P - p \cdot w_{P-d} - 1 \\ &< M_{\mathbf{x}'} + w_P - w_{P-d} \\ &\leq M_{\mathbf{x}'} + I = M_{\mathbf{x}}.\end{aligned}$$

Hence, δ_P is NOT in the correct position. We are forced to shift x'_{P-d} to the right of δ_P.

2. Suppose $w_P < I$. First, we prove that if δ_P is in its correct position, then $\delta_P = \sigma_{max}$. Otherwise, we will not able to shrink the index:

If $\delta_P = \sigma < \sigma_{max}$, then:

$$\begin{aligned}
\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + p \cdot (w_{P-1} + \ldots + w_{P-d+1}) + \sigma \cdot w_P \\
&= M_{\mathbf{x}'} + (w_P - w_{P-d} - 1) + \sigma \cdot w_P \\
&= M_{\mathbf{x}'} + (\sigma + 1) \cdot w_P - p \cdot w_{P-d} - 1 \\
&< M_{\mathbf{x}'} + \sigma_{max} \cdot (w_P - w_{P-d}) \\
&< M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

If $\delta_P = \sigma > \sigma_{max}$, then:

$$\begin{aligned}
\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + \sigma \cdot w_P \\
&> M_{\mathbf{x}'} + \sigma \cdot (w_P - w_{P-d}) \\
&> M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

Next, we consider three cases:

(a) $x'_{P-d} > \sigma_{max}$. Suppose δ_P is NOT in the correct position, therefore we shift x'_{P-d} to the right of δ_P. Observe that the change in moment of $M_{\tilde{\mathbf{x}}}$ exceeds the index:

$$\begin{aligned}
\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + x'_{P-d} \cdot (w_P - w_{P-d}) \\
&> M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

Hence, δ_P is in the correct position and as we showed previously, $\delta_P = \sigma_{max}$.

(b) $x'_{P-d} < \sigma_{max}$. Considering two sub-cases:

i. $\sigma_{max} \cdot w_P \leq I$. Assume that δ_P is NOT in the correct position. Therefore we

shift x'_{P-d} to the right of δ_P. We will not able to shrink the index since:

$$\begin{aligned}
\max(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + p \cdot (w_{P-d+1} + \ldots + w_{P-1}) + x'_{P-d} \cdot (w_P - w_{P-d}) \\
&= M_{\mathbf{x}'} + (w_P - w_{P-d} - 1) + x'_{P-d} \cdot (w_P - w_{P-d}) \\
&< M_{\mathbf{x}'} + (1 + x'_{P-d}) \cdot w_P \\
&\leq M_{\mathbf{x}'} + \sigma_{max} \cdot w_P \\
&\leq M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

Hence, δ_P is in the correct position and $\delta_P = \sigma_{max}$.

ii. $\sigma_{max} \cdot w_P > I$. Assume δ_P is in the correct position. Hence, $\delta_P = \sigma_{max}$. Then we will exceed the index since:

$$\begin{aligned}
\min(M_{\tilde{\mathbf{x}}}) &= M_{\mathbf{x}'} + \sigma_{max} \cdot w_P \\
&> M_{\mathbf{x}'} + I = M_{\mathbf{x}}.
\end{aligned}$$

Therefore δ_P is NOT in the correct position. We are foreced to shift x'_{P-d} to the right of δ_P.

(c) $x'_{P-d} = \sigma_{max}$. In this case, inserting σ_{max} for δ_P is the same as shifting x'_{P-d} to the right of δ_P. Hence, we choose to shift instead. This complete the proof.

Example 5.3. Suppose the quaternary codeword $\mathbf{x} = (2,3,3,3,0,0,0,0) \in C_8(4,3,w_9,254)$ was used to transmit and $\mathbf{x}' = (3,3,0,0,0)$ was received so that three deletions had occurred. We want to decode $\mathbf{x}'$ to recover $\mathbf{x}$.

The weigt corresponding to this codebook is defined as $w_i = 1 + 3(w_{i-1} + w_{i-2} + w_{i-3})$. The first 10 weights are given in Table 16.

Table 16

Weight w_i for $d = 3$, $q = 4$

i	1	2	3	4	5	6	7	8	9	10
w_i	1	4	16	64	253	1000	3952	15616	61705	243820

The modulus is

$$m = w_{n+1} = w_9 = 61705.$$

The residue is

$$r = 254.$$

The deleted moment is

$$\begin{aligned} M_{\mathbf{x}'} &= \sum w_i x'_i = 3 \cdot 1 + 3 \cdot 4 + 0 \cdot 16 + 0 \cdot 64 + 0 \cdot 253 \\ &= 15. \end{aligned}$$

Since $M_{\mathbf{x}'} < r$, we follow Lemma 4.8 and calculate the original moment as follows:

$$M_{\mathbf{x}} = r = 254.$$

Therefore the index is

$$\begin{aligned} I &= 254 - 15 \\ &= 239. \end{aligned}$$

Assume the deleted symbols are at the right end. We have

$$\tilde{\mathbf{x}} = (3, 3, 0, 0, 0, \delta_1, \delta_2, \delta_3)$$

As $w_8 = 15616 > I$, and $x'_5 = 0$, case $1(a)$ suggests us to shift x'_5 to the right of δ_3.

Notice that by shifting 0, the index does not change so that:

$$\tilde{\mathbf{x}} = (3,\ 3,\ 0,\ 0, \delta_1,\ \delta_2,\ \delta_3,\ 0).$$

Next, $w_7 = 3952 > I$, and $x_4' = 0$, we follow case $1(a)$ again and shift x_4' to the right of δ_3:

$$\tilde{\mathbf{x}} = (3,\ 3,\ 0,\ \delta_1,\ \delta_2,\ \delta_3,\ 0,\ 0).$$

Similarly, as $w_6 = 1000 > I$, and $x_3' = 0$, we shift x_3' to the right of δ_3:

$$\tilde{\mathbf{x}} = (3,\ 3,\ \delta_1,\ \delta_2,\ \delta_3,\ 0,\ 0,\ 0).$$

Next, as $w_5 = 253 > I$, and $x_3' = 3\ \neq 0$. We calculate $w_5 - w_2 = 249$. Since

$$I = 239 < (w_5 - w_2),$$

we follow case $1(b)(i)$ and insert $\delta_3 = 0$ so that

$$\tilde{\mathbf{x}} = (3,\ 3,\ \delta_1,\ \delta_2,\ 0,\ 0,\ 0,\ 0).$$

From here, applying DNM for 2 deletions which is the same as DN2. As $w_4 = 64 < I$, we calculate $\sigma_{max} = 3$. Since $x_2' = 3 = \sigma_{max}$, case $2(c)$ suggests us to shift x_2' to the right of δ_2 and update $I \to I - \sigma_{max}(w_4 - w_2) = 59$.

$$\tilde{\mathbf{x}} = (3,\ \delta_1,\ \delta_2,\ 3,\ 0,\ 0,\ 0,\ 0).$$

Next, as $w_3 = 16 < I$, we compute $\sigma_{max} = 3$. Since $x_1' = 3 = \sigma_{max}$, case $2(c)$ tells us to shift x_1' to the right of δ_2 and update $I \to I - \sigma_{max}(w_3 - w_1) = 14$. We have

$$\tilde{\mathbf{x}} = (\delta_1,\ \delta_2,\ 3,\ 3,\ 0,\ 0,\ 0,\ 0).$$

Then we have

$$I = 2 \cdot w_1 + 3 \cdot w_2.$$

Hence, $\delta_1 = 2$ and $\delta_2 = 3$. Our final decoding codeoword is

$$\tilde{\mathbf{x}} = (2,\ 3,\ 3,\ 3,\ 0,\ 0,\ 0,\ 0) = \mathbf{x}.$$

Chapter 6

Conclusion

The Levenshtein code, which is based on Varshamov-Tenengol'ts' construction, is remarkable for a number of reasons. Levenshtein's results are the foundation for Tenengol'ts' work on the non-binary single insertion/deletion error correcting code. Moreover, Helberg generalized the aforementioned construction in the Levenshtein code to obtain a new class of codes which are capable of correcting multiple insertion/deletion errors. Prior to this thesis, there was no known efficient algorithm to correct Helberg codes. Swart [21] informally gave an algorithm for correcting deletions; however his algorithm is more complicated and less efficient than the algorithm presented in this thesis, which we have called the Deletion Decoding Algorithm.

To decode Helberg codes, our recursive algorithm decodes one symbol error each time. As a result, our algorithm is much more efficient in term of the length of codeword and the number of deletion errors. However, for the correction of one deletion error $(d = 1)$, the Deletion Decoding Algorithm essentially performs an exhaustive search. But for two or more deletion errors, the algorithm attempts to correct the rightmost deletion symbol by shifting the deleting symbol from right to left, or inserting a value for the rightmost deletion symbol, after with the decoding reduces to the algorithm for correcting $(d-1)$ deletion errors. Therefore when the length of the codeword increases, the complexity of our algorithm is linear, namely $O(n)$. This is a drastic improvement over exhaustive search, which has exponential complexity.

As previously mentioned, the Deletion Decoding Algorithm is only able to correct deletion errors. We have not been able to adapt this algorithm to correct insertion errors due to the fact that such errors are intrinsically different from deletion errors. In the case of deletion errors, we insert d deletion symbols, denoted by δ_i's, and for each symbol, we try to determine whether to shift it or insert a value. These two possibilities give us more

freedom in correcting a codeword than in the case of insertion error, where we can only delete symbols. In particular, we can only determine the position of the inserted symbols; we have no control over the value of the symbols, which make it more difficult to decode.

In this thesis, we also constructed a new class of non-binary codes capable of correcting multiple insertion/deletion errors, generalizing Helberg codes. Since these new codes apply to ternary and quaternary alphabets, they will find applications in biology, e.g., DNA bar coding.

Since our decoding algorithm is not capable of correcting insertion errors and our generalized Helberg codes have poor rates, there remains work to find a better algorithm as well as codes with better rates.

References

[1] A. S. J. Helberg. (1993). *Coding for the Correction of Synchronization Errors* (Ph.D. dissertation). Rand Afrikaans University, Johannesburg, South Africa.

[2] A. S. J. Helberg, H. C. Ferreira. (2002). On Multiple Insertion/Deletion Correcting Codes. *IEEE Trans. Inf. Theory, vol(48)*, 305-308.

[3] C. E. Shannon. (1949). *A Mathematical Theory of Communication*. University of Illinois Press.

[4] D. Kracht, S. Schober (2015). Insertion And Deletion Correcting DNA Barcodes Based On Watermarks. *BMC Bioinformatic, vol(10)*, 16-50.

[5] Filip Palunčić, Khaled A. S. Abdel-Ghaffar, Hendrik C. Ferreira, A. Clarke(2012). A Multiple Insertion/Deletion Correcting Code for Run-Length Limited Sequences. *IEEE Trans. Information Theory vol(58)*, 1809-1824.

[6] F. Palunčić, T. G. Swart, J. H. Weber, H. C. Ferreira, W. A. Clarke (2011). A Note on Non-binary Multiple Insertion/Deletion Correcting Codes. *Information Theory Workshop, IEEE*, 683-687.

[7] Grigory Tenengol'ts (1994). Nonbinary Code, Correcting Single Deletion or Insertion. *IEEE Trans. Information Theory, vol(IT-30)*, 766-769.

[8] I. Landjev and K. Haralambiev (2007). On Multiple Deletion Codes. *Serdica J. Comput, vol(1)*, 13-26.

[9] T. A. Le, H. D. Nguyen (2016).New Multiple insertion/deletion Correcting Codes For Non-Binary Alphabets. *IEEE Trans. Information Theory, vol(62)*, 2682-2693 .

[10] K. A. S. Abdel-Ghaffar, F. Palunčić, H. C. Ferreira, W. A. Clarke (2012). On Helberg's Generalization of the Levenshtein Code for Multiple Deletion/Insertion Error Correction. *IEEE Trans. Inf. Theory, vol(58)*, 1804-1808.

[11] L. Dolecek, V. Anatharam (2010). Repetition Error Correcting Sets: Explicit Constructions And Prefixing Methods. *SIAM J. Discrete Math, vol(23)*, 687-698.

[12] L. J. Schulman, D. Zuckerman (1999). A Symptotically Good Codes Correcting Insertions, Deletiions, and Transpositions. *IEEE Trans. Inf. Theory, vol(45)*, 2552-2557.

[13] M. C. Davey, D. J. C. Mackay 92001). Reliable Communication Over Channels With Insertion, Deletions, And Substitution. *IEEE Trans. Inf. Theory, vol(47)*, 687-698.

[14] N. J. A. Sloane (2000). *One Single-Deletion-Correcting Codes*. Information Sciences Research, AT&T Shannon Labs.

[15] V. I. Levenshtein (1996). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Sov. Phys.-Dokl., vol(10)*, 707-710.

[16] V. I. Levenshtein (2002). Bonds For Deletion/Insertion Correcting Codes, *Proceeding of the 2002 IEEE International Symposium on Information Theory (pp. 370)*. Lausanne, Switzerland.

[17] I. S. Reed, G. Solomon (1957). Polynomial Codes Over Certain Finite Fields. *Journal of Society for Industrial and Applied Mathematics, vol(2)*, 300-304.

[18] R. W. Hamming (1950). *Error Dectecting and Error Correcting Codes*. The Bell System Technical Journal.

[19] R. R. Varshamov, G. M. Tenengol'ts (1965). Codes Which Correct Single Asymmetric Errors. *Autom. Remote Control, vol(26)*, 286-290.

[20] R. R. Varshamov, G. M. Tenengolt's (1965). Correction Code For Single Asymmetrical Errors. *Avtom. Telemekh, vol(26)*, 288-292.

[21] T. G. Swart (2001). *Coding And Bounds For Correcting Insertion/Deletion Errors* (M.S. Thesis). Rand Afrikaans University. Johannesburg, South Africa.

Appendix A

Decoding Algorithm for Binary Helberg Codes

Algorithm D1

procedure ONE DELETION METHOD FOR BINARY

$\tilde{x} = x'_1 x'_2 x'_3 ... x'_{n-1} \delta$

▷ Initialize $\tilde{x}$ with unknown bit δ at position n, where δ is to be determined.

for Position $P = n$ to $P = 1$ **do**

if $I = 0$ **then**

$\delta = 0$

else if $I = v_P$ **then**

$\delta = 1$

else

$x = x'_1 x'_2 x'_3 ... x'_{P-2} \delta x'_{P-1} x'_{P+1} ... x'_{n-1}$. ▷ Shift x'_{P-1} to the right of δ.

$I = I - x'_{P-1}(v_P - v_{P-1})$. ▷ Update the index.

end if

end for

end procedure

Algorithm D2

procedure TWO DELETIONS METHOD FOR BINARY

$x = x'_1 x'_2 x'_3 ... x'_{n-2} \delta_1 \delta_2$ ▷ Initialize x with unknown bits δ_1 and δ_2 are at position $n-1$ and n respectively, where δ's are to be determined

for Position $P = n$ to $P = 1$ **do**

if $d = 1$ **then**

One Deletion Procedure ▷ Apply D1

end if

7: **for** $\sigma_1, \sigma_2 = 1$ to 0 **do**

8: **if** $I = \sigma_1 w_{P-1} + \sigma_2 w_P$ **then**

9: $\delta_1 = \sigma_1, \delta_2 = \sigma_2$

10: STOP

11: **end if**

12: **end for**

13: **if** $x'_{P-2} = 0$ **then**

14: **if** $v_P > I$ **then**

15: $x' = x'_1 x'_2 ... x'_{P-3} \delta_1 \delta_2 0 x'_{P+1} ... x'_{n-2}$ ▷ Shift x'_{P-2} to the right of δ_2

16: **else if** $v_P < I$ **then**

17: $x' = x'_1 x'_2 ... x'_{P-3} 0 \delta_1 1 x'_{P+1} ... x'_{n-2}$ ▷ Insert 1 for δ_2

18: $I = I - v_P$ ▷ Update the index

19: $d = 1$ ▷ Update number of deletions

20: **end if**

21: **else if** $x'_{P-2} = 1$ **then**

22: **if** $v_P < I$ **then**

23: $x' = x'_1 x'_2 ... x'_{P-3} \delta_1 \delta_2 1 x'_{P+1} ... x'_{n-2}$ ▷ Shift x'_{P-2} to the right of δ_2

24: $I = I - (v_P - v_{P-2})$ ▷ Update the index

25: **else if** $v_P > I$ **then**

26: **if** $(v_P - v_{P-2}) \leq I$ **then**

27: $\tilde{x} = x'_1 x'_2 ... x'_{P-3} \delta_1 \delta_2 1 x'_{P+1} ... x'_{n-2}$ ▷ Shift x'_{P-2} to the right of δ_2

28: $I = I - (v_P - v_{P-2})$ ▷ Update the index

29: **else if** $(v_P - v_{p-2}) > I$ **then**

30: $x' = x'_1 x'_2 ... x'_{P-3} \delta_1 10 x'_{P+1} ... x'_{n-2}$ ▷ shift x'_{P-2} to the right of δ_1 and insert $\delta_2 = 0$

31: $d = 1$ ▷ Update number of deletions

32: **end if**

end if

end if

end for

end procedure

Algorithm DM

procedure MULTIPLE DELETIONS METHOD FOR BINARY

$\tilde{x}x = x'_1x'_2x'_3...x'_{n-d}\delta_1\delta_2...\delta_{d-1}\delta_d$ ▷ Initialize $\tilde{x}$ with unknown bits $\delta_1, \delta_2...\delta_d$ are at position $n-d+1$, $n-d+2$,...,n respectively, where δ's are to be determined

for Position $P = n$ to $P = 1$ **do**

if $d = 1$ **then**

One Deletion Procedure ▷ Apply D1

end if

for $\sigma_1, \ldots, \sigma_d = 1$ to 0 **do**

if $I = \sigma_1 w_{P-1} + \cdots + \sigma_d w_P$ **then**

$\delta_1 = \sigma_1, \cdots, \delta_d = \sigma_d$

STOP

end if

end for

if $v_P > I$ **then**

if $x'_{P-d} = 0$ **then**

$\tilde{x} = x'_1x'_2...x'_{P-d-1}\delta_1\delta_2...\delta_d 0...x'_{n-d-1}x'_{n-d}$ ▷ Shift last known bit to the right of δ_d

$P = P - 1$. ▷ Update new position for δ's

else

$\tilde{x} = x'_1x'_2...x'_{P-d}\delta_1\delta_2...\delta_{d-1}0...x'_{n-d-1}x'_{n-d}$ ▷ Inserting 0 for δ_d

19: $d = d - 1$. ▷ Update number of deletions

20: **end if**

21: **else** ▷ $v_P \leq I$

22: **if** $x'_{P-d} = 1$ **then**

23: $x = x'_1 x'_2 \ldots x'_{P-d-1} \delta_1 \delta_2 \ldots \delta_d 1 \ldots x'_{n-d-1} x'_{n-d}$ ▷ Shift last known bit to the right of δ_d

24: $I = I - (v_P - v_{P-d})$ ▷ Update the index

25: $P = P - 1$. ▷ Update new position for δ's

26: **else**

27: $x = x'_1 x'_2 \ldots x'_{P-d} \delta_1 \delta_2 \ldots \delta_{d-1} 1 \ldots x'_{n-d-1} x'_{n-d}$ ▷ Inserting 1 for δ_d

28: $I = I - v_P$ ▷ Update the index

29: $d = d - 1$. ▷ Update number of deletions

30: **end if**

31: **end if**

32: **end for**

33: **end procedure**

Appendix B

Decoding Algorithm for Non-Binary Helberg Codes

Algorithm DN1

procedure ONE DELETION METHOD FOR NON-BINARY

$\tilde{x} = x'_1 x'_2 x'_3 \ldots x'_{n-1} \delta$

▷ Initialize $\tilde{x}$ with unknown bit δ at position n, where δ is to be determined.

for Position $P = n$ to $P = 1$ **do**

for $\sigma = q-1$ to 0 **do**

if $I = \sigma \cdot w_P$ **then**

$\delta = \sigma$

STOP

end if

end for

$\tilde{x} = x'_1 x'_2 x'_3 \ldots x'_{P-2} \delta x'_{P-1} x'_p \ldots x'_{n-1}$. ▷ Shift x'_{P-1} to the right of δ one unit.

$I = I - x'_{P-1}(w_P - w_{P-1})$. ▷ Update the index.

end for

end procedure

Algorithm DN2

procedure TWO DELETIONS METHOD FOR NON-BINARY

$\tilde{x} = x'_1 x'_2 x'_3 \ldots x'_{n-d} \delta_1 \delta_2$▷ Initialize $\tilde{x}$ with unknown bits δ_1, δ_2 are at position $n-1$, n respectively, where δ's are to be determined

for Position $P = n$ to $P = 1$ **do**

if $d = 1$ **then**

One Deletion Procedure ▷ Apply DN1

end if

7: **for** $\sigma_1, \sigma_2 = q-1$ to 0 **do** ▷ d-nested for loop

8: **if** $I = \sigma_1 w_{P-1} + \sigma_2 w_P$ **then**

9: $\delta_1 = \sigma_1, \delta_2 = \sigma_2$

10: STOP

11: **end if**

12: **end for**

13: **if** $w_P > I$ **then**

14: **if** $(x'_{P-d} = 0)$ or $(x'_{P-d} \geq 1$ and $I \geq w_P - w_{P-d})$ **then**

15: $\tilde{x} = x'_1 x'_2 ... x'_{P-3} \delta_1 \delta_2 x'_{p-2} ... x'_{n-3} x'_{n-2}$ ▷ Shift last known bit to the right of δ_2

16: $I = I - x'_{P-2}(w_P - w_{P-2})$ ▷ Update the index

17: **else**

18: $\tilde{x} = x'_1 x'_2 ... x'_{P-2} \delta_1 0 ... x'_{n-3} x'_{n-2}$ ▷ Inserting 0 for δ_2

19: $d = 1.$ ▷ Update number of deletions

20: **end if**

21: **else** ▷ $w_P \leq I$

22: **for** $\sigma = p$ to $\sigma = 0$ **do** ▷ Compute σ_{max}

23: **if** $\sigma(w_P - w_{P-2}) \leq I$ **then**

24: $\sigma_{max} = \sigma$

25: **end if**

26: **end for**

27: **if** $x'_{P-2} > \sigma_{max}$ **then**

28: $\tilde{x} = x'_1 x'_2 ... x'_{P-2} \delta_1 \sigma_{max} ... x'_{n-3} x'_{n-2}$ ▷ Insert σ_{max} for δ_2

29: $I = I - \sigma_{max} w_P$ ▷ Update the index

30: $d = 1.$ ▷ Update number of deletions

31: **else**

32: **if** $\sigma_{max} w_P > I$ or $x'_{P-2} = \sigma_{max}$ **then**

$\tilde{x} = x'_1 x'_2 ... x'_{P-3} \delta_1 \delta_2 ... \delta_d x'_{P-2} ... x'_{n-3} x'_{n-2}$ ▷ Shift x'_{P-2} to the right of δ_2

$I = I - x'_{P-2}(w_P - w_{P-2})$ ▷ Updatethe index

else

$\tilde{x} = x'_1 x'_2 ... x'_{P-2} \delta_1 \sigma_{max} ... x'_{n-3} x'_{n-2}$ ▷ Insert σ_{max} for δ_2

$I = I - \sigma_{max} w_P$ ▷ Update the index

$d = 1$ ▷ Update number of deletions

end if

end if

end if

end for

end procedure

Algorithm DNM

procedure MULTIPLE DELETIONS METHOD FOR NON-BINARY

$\tilde{x} = x'_1 x'_2 x'_3 ... x'_{n-d} \delta_1 \delta_2 ... \delta_{d-1} \delta_d$ ▷ Initialize x with unknown bits $\delta_1, \delta_2 ... \delta_d$ are at position $n-d+1$, $n-d+2$,...,n respectively, where δ's are to be determined

for Position $P = n$ to $P = 1$ **do**

if $d = 1$ **then**

One Deletion Procedure ▷ Apply DN1

end if

for $\sigma_1, \ldots, \sigma_d = q-1$ to 0 **do** ▷ d-nested for loop

if $I = \sigma_1 w_{P-d+1} + \cdots + \sigma_d w_P$ **then**

$\delta_1 = \sigma_1, \ldots, \delta_d = \sigma_d$

STOP

end if

12: **end for**

13: **if** $w_P > I$ **then**

14: **if** $(x'_{P-d} = 0)$ or $(x'_{P-d} \geq 1$ and $I \geq w_P - w_{P-d})$ **then**

15: $\tilde{x} = x'_1 x'_2 ... x'_{P-d-1} \delta_1 \delta_2 ... \delta_d x'_{p-d} ... x'_{n-d-1} x'_{n-d}$ ▷ Shift last known bit to the right of δ_d

16: $I = I - x'_{P-d}(w_P - w_{P-d})$ ▷ Update the index

17: **else**

18: $\tilde{x} = x'_1 x'_2 ... x'_{P-d} \delta_1 \delta_2 ... \delta_{d-1} 0 ... x'_{n-d-1} x'_{n-d}$ ▷ Inserting 0 for δ_d

19: $d = d - 1$. ▷ Update number of deletions

20: **end if**

21: **else** ▷ $w_P \leq I$

22: **for** $\sigma = p$ to $\sigma = 0$ **do** ▷ Compute σ_{max}

23: **if** $\sigma(w_P - w_{P-d}) \leq I$ **then**

24: $\sigma_{max} = \sigma$

25: **end if**

26: **end for**

27: **if** $x'_{P-d} > \sigma_{max}$ **then**

28: $\tilde{x} = x'_1 x'_2 ... x'_{P-d} \delta_1 \delta_2 ... \delta_{d-1} \sigma_{max} ... x'_{n-d-1} x'_{n-d}$ ▷ Insert σ_{max} for δ_d

29: $I = I - \sigma_{max} w_P$ ▷ Update the index

30: $d = d - 1$. ▷ Update number of deletions

31: **else**

32: **if** $\sigma_{max} w_P > I$ or $x'_{P-d} = \sigma_{max}$ **then**

33: $\tilde{x} = x'_1 x'_2 ... x'_{P-d-1} \delta_1 \delta_2 ... \delta_d x'_{P-d} ... x'_{n-d-1} x'_{n-d}$ ▷ Shift x'_{P-d} to the right of δ_d

34: $I = I - x'_{P-d}(w_P - w_{P-d})$ ▷ Updatethe index

35: **else**

36: $\tilde{x} = x'_1 x'_2 ... x'_{P-d} \delta_1 \delta_2 ... \delta_{d-1} \sigma_{max} ... x'_{n-d-1} x'_{n-d}$ ▷ Insert σ_{max} for δ_d

$I = I - \sigma_{max} w_P$ ▷ Update the index

$d = d - 1$ ▷ Update number of deletions

end if

end if

end if

end for

end procedure