Construction of 1-Deletion-Correcting Ternary Codes

Zhiyuan Li

Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

© 2011

Abstract

Finding large deletion correcting codes is an important issue in coding theory. Many researchers have studied this topic over the years. Varshamov and Tenegolts constructed the Varshamov-Tenengolts codes (VT codes) and Levenshtein showed the Varshamov-Tenengolts codes are perfect binary one-deletion correcting codes in 1992. Tenegolts constructed T codes to handle the non-binary cases. However the T codes are neither optimal nor perfect, which means some progress can be established. Latterly, Bours showed that perfect deletion-correcting codes have a close relationship with design theory. By this approach, Wang and Yin constructed perfect 5-deletion correcting codes of length 7 for large alphabet size. For our research, we focus on how to extend or combinatorially construct large codes with longer length, few deletions and small but non-binary alphabet especially ternary. After a brief study, we discovered some properties of T codes and produced some large codes by 3 different ways of extending some existing good codes.

Acknowledgement

I wish to express my greatest appreciation to my supervisor, Professor Sheridan Houghten. She guided me into this interesting area and provided me precious experience. Without her encouragement and suggestions, I could not finish my research.

I would like to acknowledge the help of my supervisory committee: Professor Ke Qiu and Professor Michael Winter. I also want to thank the staff of the computer science department for various supports.

Finally, I am forever grateful to my parents and all my family members for their endless understanding. I would like to deliver my acknowledge to all my friends.

Contents

Abstract 1

Acknowledgement 2

List of Tables 6

List of Figures 7

1 Introduction 8

1.1 Codes defined using Hamming distance 9

1.2 Codes Defined Using Insertion/Deletion Distance 13

2 Background 15

2.1 Deletion Correcting Codes 15

2.2 Varshamov-Tenegolts codes 22

2.3 Tenegolts Codes 26

2.4 Design Theory 31

2.5 Two Trivial Construction and Extensions 33

3 Difficulties in Construction of Perfect Codes **38**

4 A Simple Construction for Ternary Codes **46**

4.1 Definitions of S Codes 47

4.2 Correctness of S Codes 48

4.3 Decoding of S Codes 57

5 The Extension of T Codes **61**

5.1 Properties and Hypothesis 61

5.2 One Extendable Codeword 78

6 Results and Implementations **83**

6.1 Results 83

6.2 Implementation 86

7 Conclusion and Future Work **89**

Bibliography **91**

List of Tables

3.1 Different Types of Words of Length 3 41

3.2 The Construction of Perfect Codes of Length 3. 43

4.1 The Size of S Codes 60

5.1 $T_{0,0}(5)$ Code 65

5.2 The Sizes of $T(3)$ 66

5.3 The Sizes of $T(4)$ 66

5.4 The Sizes of $T(5)$ 66

5.5 The Sizes of $T(6)$ 67

5.6 The Sizes of $T(7)$ 67

5.7 The Sizes of $T(8)$ 68

5.8 The Sizes of $T(9)$ 68

5.9 The Sizes of $T(10)$ 68

5.10 The Uncovered Errors of $T_s(4)$, $T_s(5)$ and $T_s(6)$ 74

5.11 7 Examples of Extendable Errors 76

5.12 The Number of Uncovered Errors and Extendable Errors. . . . 77

6.1 The Comparison of Different Codes 84

6.2 The Optimal Extensions of $T(5)$ 85

6.3 The Optimal Extensions of $T(6)$ 85

6.4 The Optimal Extensions of $T(7)$ 85

6.5 The Optimal Extensions of $T(8)$ 86

6.6 The Optimal Extensions of $T(9)$ 86

6.7 The Optimal Extensions of $T(10)$ 87

List of Figures

1.1 The procedure of encoding and decoding. 9

1.2 Hamming Space 13

1.3 Synchronization Error 14

2.1 Deletion Space 16

2.2 Spheres 20

2.3 Perfect and Optimal Codes of Length 3 21

2.4 Backtracking Algorithm 36

3.1 The Words of Type "AAB" 40

3.2 The Words of Type "ABA" 40

3.3 The Words of Type "ABC" 40

3.4 Perfect Codes of Length 3 and 4 45

Chapter 1

Introduction

In this chapter we introduce the basics of coding theory and give a brief picture of how a code works. Some traditional codes which are based on *Hamming distance* will be discussed.

Coding theory is a study that focuses on correcting errors during data transmission and storage. It adds some *redundancy* symbols to a *message* to correct errors. A *code* is defined by a code book, which is a set of codewords. A *codeword* is a string of information symbols and redundancy symbols. We will provide the formal definitions in the next section. A *word* is a string, a sequence or a vector. A code sometimes defines an *encoding* and *decoding* algorithm, but not necessarily.

Suppose we wish to send a message over a channel. *Channel* is a general idea of hardware. For example, it can be a telephone line or a CD. When sending a message, the message will first be encoded to produce a codeword. Since

the channel is noisy, some errors might be added to the codeword during transmission. Finally, the received string will be decoded back to the original message by the decoding algorithm. See Figure 1.1 for a depiction of this process.

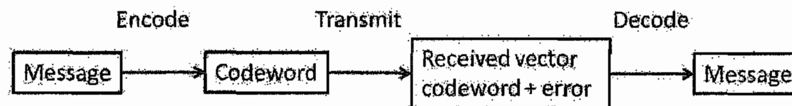

Figure 1.1: The procedure of encoding and decoding.

1.1 Codes defined using Hamming distance

In 1950, Hamming constructed the first error correcting code, named the *Hamming code* which corrects a single substitution error. Codes correcting substitution errors are based on *Hamming distance*, which is defined as follows.

Definition 1.1.1. The *Hamming distance* between two words is the number of different corresponding symbols.

Example 1.1.1. 1001 and 1101 are two words with distance 1. They differ at the second symbol.

And we further define *vector space*, *errors*, *minimum distance*, and *code* as follows. After the definitions, we will provide an example. In general all possible symbols belong to a given *alphabet*. For Hamming codes, the alphabet

is a finite field $\mathbb{F}_q$. When $q = 2$ all symbols are either 0 or 1 so we have a *binary* code.

Definition 1.1.2. The *vector space* of all n-tuples over the finite field $\mathbb{F}_q$ is denoted by $\mathbb{F}_q^n$.

Definition 1.1.3. All the words in the vector space except the codewords are *error vectors* or *errors.*

Definition 1.1.4. The *minimum distance* of a code is the minimal distance between any two codewords.

Definition 1.1.5. A q-ary code of length n is a subset of $\mathbb{F}_q^n$. Each n-tuple is a codeword. If the minimum distance of a code is d, then the code is denoted by $(n, d)_q$ code.

Definition 1.1.6. If the minimum distance of a code is d, then the *sphere* of *radius* $\lfloor (d-1)/2 \rfloor$ about a codeword is a set of words such that every word inside the sphere has distance less than or equal to $\lfloor (d-1)/2 \rfloor$ to the codeword.

Every word e in the sphere for a given codeword w will be corrected to w, because e has distance less than or equal to $\lfloor (d-1)/2 \rfloor$ to w and distance larger than $\lfloor (d-1)/2 \rfloor$ to other codewords.

Definition 1.1.7. If the minimum distance of a code is measured by Hamming distance, then this code is *substitution code.*

So, a sphere of a word in a substitution code is a set of vectors within the same vector space.

We now define *optimal* and *perfect* and provide an example of a code with these attributes.

Definition 1.1.8. A $(n, d)_q$ code is *perfect* if every q-ary vector of length n has distance less than or equal to $\lfloor (d-1)/2 \rfloor$ to exactly one codeword. This means every possible vector of length n can be corrected by exactly one codeword.

Definition 1.1.9. The size of a code is the number of codewords. An $(n, d)_q$ code is *optimal* if there is no code that has the same length and alphabet and corrects the same number of errors, but has larger size.

Example 1.1.2. Consider the binary code {000,111}. This code has length 3 and minimum distance 3, and thus can correct one error. It has two codewords 000 and 111, so its size is 2. If the received message is one of {000,001,010,100}, then it is corrected to 000, and if the received message is one of {110,101,011,111} then it is corrected to 111. Therefore the sphere of radius 1 about 000 contains {000,001,010,100}, while the sphere about 111 contains {111,110,101,011}. Since all possible words of length 3 are in these spheres, the code is perfect. Also, there is no code with the same attributes and larger size, so this code is optimal.

A perfect $(n, d)_q$ code has the following attributes:

1. The number of substitution errors it corrects is $\lfloor (d-1)/2 \rfloor$.

2. The size of a sphere of radius $\lfloor (d-1)/2 \rfloor$ is :

$$(1 * (q-1) + \binom{n}{1} * (q-1) + \binom{n}{2} * (q-1) + \binom{n}{3} * (q-1) + ... + \binom{n}{\lfloor (d-1)/2 \rfloor} * (q-1) \quad (1.1)$$

There is a relationship between perfect codes and optimal codes. Being perfect is a sufficient condition for the code to be optimal.

Theorem 1.1.1. *If a code is a perfect substitution code, then it is also optimal.*

Proof. Assume C is a perfect $(n,d)_q$ code. The size of the entire space is q^n, which is fixed. The size of a sphere is also fixed by Equation 1.1. Every q-ary word is in exactly one sphere. Furthermore, there are no gaps between spheres.

$\therefore$ We cannot add other sphere into the space of C.

$\therefore$ There is no another code which has larger size.

$\therefore$ C is optimal. □

We need to notice that optimal codes are not always perfect. For some length and alphabets, perfect codes may not exist, but we can still find the optimal one by exhaustive search.

Consider the graph in which the set of vertices are all the binary words of length n, and in which there is an edge from word x to word y if and only if x is Hamming distance 1 from y. This graph is a hypercube. Figure 1.2 shows

the Hamming space for binary words of length 3. Given code $\{000,111\}$, the four words on the top right are in one sphere and the other four are in the other sphere.

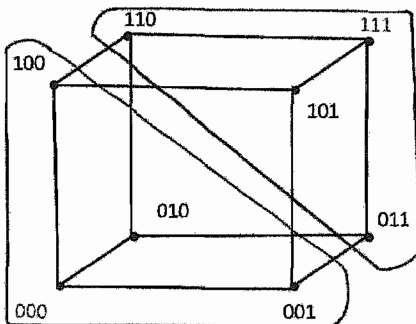

Figure 1.2: Hamming Space

Many of the best substitution codes are *linear codes.* This type of code is defined as follows.

Definition 1.1.10. A code is linear if every linear-combination of codewords is also a codeword.

The Hamming code is an example of a linear code.

1.2 Codes Defined Using Insertion/Deletion Distance

Substitution errors are not the only type of errors. For example, when the sender and receiver are not well synchronized, this causes deletion or insertion

of symbols. If the received word is shorter or longer than the sent codeword, the traditional codes, which are based on Hamming distance, will not work any more. As a result, *deletion correcting codes* are studied. These codes are based on *insertion/deletion distance.*

Definition 1.2.1. The *insertion/deletion distance* is the minimum number of deletions or insertions required to transform one string into another.

An example is shown in Figure 1.3. A sender transmits a word "0100100111", but the receiver is not well synchronized with the sender. It starts at the third bit. The first two bits are not read by the receiver. The fundamentals of deletion correcting codes are explained in Chapter 2.

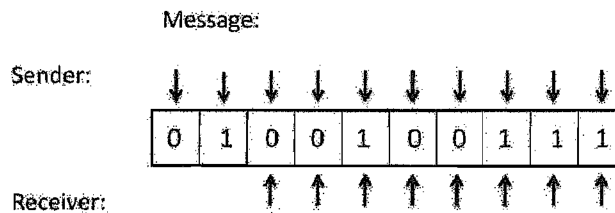

Figure 1.3: Synchronization Error

Chapter 2

Background

2.1 Deletion Correcting Codes

In this section we will introduce the fundamentals of deletion correcting codes. For further information, see[7, 8, 9].

The insertion/deletion distance, as defined in Definition 1.2.1, measures the distances between strings. It is a metric on the set of strings from the alphabet.

Example 2.1.1. Word $x = 1010$ and $y = 1011$ have distance 2. In this case there are multiple ways to transform x into y, such as deleting all symbols in 1010 and inserting all symbols of 1011 back. This takes 4 deletions and 4 insertions. But the shortest transformation is removing the last 0 and adding 1 at the end.

This distance can be shown as a graph named the insertion/deletion space.

Vertices are words and two vertices are adjacent if and only if they have insertion/deletion distance 1 (see Figure 2.1). Since an insertion is the inverse of

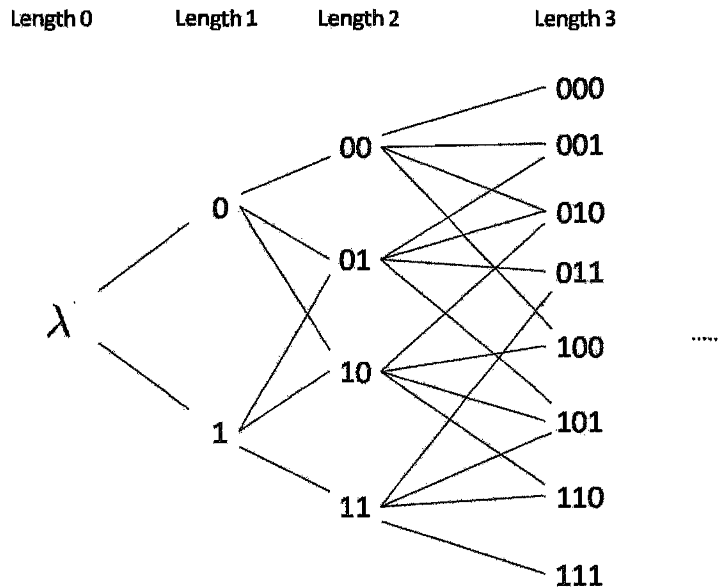

Figure 2.1: The deletion space, λ means a string with length zero.

a deletion, we will only discuss deletion errors and deletion correcting codes. The deletion correcting codes that we studied are not generally linear codes. There are two kinds of deletion correcting codes: variable length and fixed length. The variable length means the codewords in a code may have different lengths with the length of the code defined as the length of its longest codeword. If all the codewords must have the same length, then the code is fixed length.

Example 2.1.2. The code $\{000{,}101\}$ is a 1-deletion correcting fixed length code. It corrects all single deletions that create words of length 2. The words

$\{01, 10, 11\}$ are all corrected to 101, and 00 is corrected to 000. Since the two codewords have the same length, the code is a fixed length code.

Definition 2.1.1. An error for an $(n-t, n, q)$-code is a word of length $\geq n-t$.

Definition 2.1.2. *Correcting* an error is the process of replacing an error by a codeword.

Example 2.1.3. The code $\{0000,101\}$ is a deletion correcting variable length code, since the word 101 has length 3, while the word 0000 has length 4.

In our research, fixed length codes are studied. So, in the following chapters, when we say deletion correcting codes, it simply means fixed length codes. Now we define notation for deletion correcting codes:

Definition 2.1.3. An $(n-t, n, q)$-code is a deletion correcting code of length n, alphabet size q and correcting t deletions.

Definition 2.1.4. A sphere of a codeword w in an $(n-t, n, q)$-code is a set of words such that each of the words has deletion distance $\leq t$ to w.

The definition of perfect and optimal can be applied from traditional codes to deletion correcting codes as follows:

Definition 2.1.5. An $(n-t, n, q)$-code is perfect if all of the words of length $n-t$ are in exactly one sphere about a codeword.

Definition 2.1.6. The minimum distance of a deletion correcting code is the minimum insertion/deletion distance between any two codewords.

Example 2.1.4. $\{000, 101\}$ is a perfect $(2, 3, 2)$-code. There are 4 possible errors: 00 corrected by 000 and $01, 10, 11$ all corrected by 101.

Definition 2.1.7. A deletion correcting code is optimal if there is no another code which has the same length and alphabet and corrects the same number of deletions but has larger size.

We also need to provide the formal definitions for *subsequence, supersequence* and a *run* of a codeword.

Definition 2.1.8. Let w be a word of length n and w' be a word of length $n - t$. If w' has t deletion distance to w, then w' is a *subsequence* of w and w is a *supersequence* of w'.

Example 2.1.5. Suppose we have two words 10110 and 100. These two words have deletion distance 2. So 100 is a subsequence of 10110 and 10110 is a supersequence of 100.

Definition 2.1.9. A *run* in a word is a maximal contiguous sequence of the same symbol.

Example 2.1.6. The word 100110 has four runs: 1, 00, 11 and 0. For the second run, 00 is maximal, but 0 is not, because it can be extended. If we remove one 0 from the second run and one 1 from the third run, then we have a subsequence: 1010. Whether the first or second 0 is removed from the second run, we create the same the result. The same occurs when removing the first or second 1 from the third run: we will still get the same subsequence.

Therefore, we have the following observation:

Observation 2.1.1. *Deletions in the same run but at different positions produce the same subsequence.*

Proof. Let $C = \{c_1c_2c_3...c_mc_{m+1}...c_{m+p}...c_n\}$ be a string and $c_mc_{m+1}...c_{m+p}$ be a run of the string, in which each symbol is x.

Let $C_a = \{c_1c_2c_3...c_mc_{m+1}...c_{m+a-1}c_{m+a+1}...c_{m+p}...c_n\}$ and

$C_b = \{c_1c_2c_3...c_mc_{m+1}...c_{m+b-1}c_{m+b+1}...c_{m+p}...c_n\}$ be two substrings of $C(m \leq a, b \leq p$ and $a \neq b)$.

Therefore the beginning and ending of C_a and C_b are the same, which are $\{c_1c_2c_3...c_{m-1}\}$ and $\{c_{m+p+1}...c_n\}$.

By the definition of a run, the symbols of $c_mc_{m+1}...c_{m+p}$ are the same, which is a string of x's of length p.

Therefore $\{c_mc_{m+1}...c_{m+a-1}c_{m+a+1}\}$ and $c_mc_{m+1}...c_{m+b-1}c_{m+b+1}...c_{m+p}$ are two strings of x's of length $p-1$, which are the same.

Therefore C_a is the same as C_b.

Since $a \neq b$, deletions in the same run but at different positions provide the same subsequence. □

This observation also shows the number of subsequences of a word is closely related to the number of the runs. Therefore the number of error vectors and the size of a code is related to the number of runs. If it is 1-deletion correcting code, then the size of a sphere about a codeword is equal to the number of runs in the codewords. Since the number of runs in different codewords are

not necessarily equal, the sizes of the spheres are also not necessarily equal. Even the vectors in a sphere have different lengths. The codewords are always longer than the errors. Figure 2.2 shows an example of the spheres of different sizes. If 101 is selected as a codeword, it corrects 10, 01 and 11. This sphere has size 4. In the sphere of 000, it has only two words: 000 and 00.

This is totally different from the traditional (Hamming) codes. As we previously mentioned, the size of the spheres of substitution codes are the same.

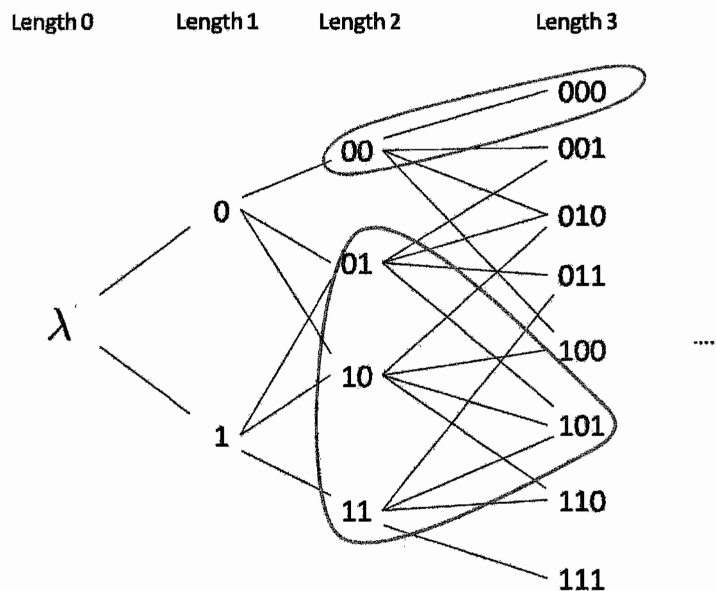

Figure 2.2: Spheres

As a result, Theorem 1.1.1 cannot be applied to deletion correcting codes. For a particular perfect deletion correcting code, some codewords with many runs may be replaced by more codewords with fewer runs. Also, some error

vectors, which can be corrected in the perfect code may not be corrected when this happens.

Figure 2.3 shows an example of such a case. C_1, C_2 and C_3 are three 1-deletion correcting codes of length 4. C_1 corrects all words of deletion distance 1, so it is perfect. However it is not optimal. If we want to extend the size of C_1, then we find the size of the spheres of codeword 0010 and 1011 are too large. We can replace 0010 by 0000 and 1011 by 1111. Each of them has 2 fewer runs and the size of the sphere shrinks by two. As a result, {001,010,011,101} are not covered. Then we can add another codeword 0011, which corrects 001 and 011. So, this is code C_2. After searching the entire deletion space, we will find there is no code that has larger size than C_2, which means C_2 is optimal. But it is not perfect, words 010 and 101 are not covered. We can also make some changes in order to cover all the error vectors. 0011 and 1100 are replaced by 1001 and 0110. And now we have C_3 which is perfect and optimal at the same time.

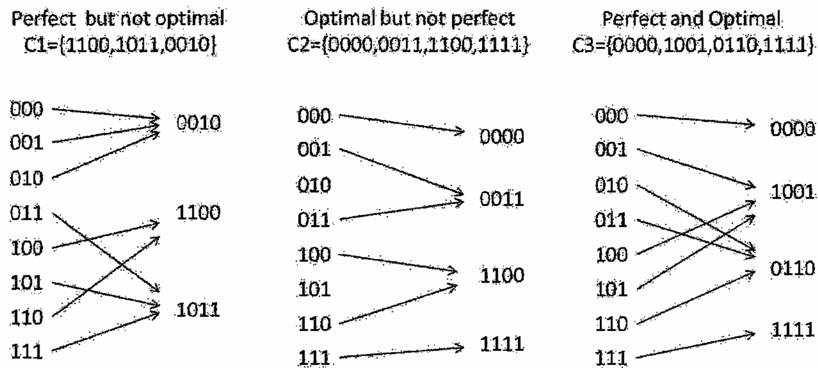

Figure 2.3: Perfect and Optimal Codes of Length 3

2.2 Varshamov-Tenegolts codes

In this chapter, we will introduce *Varshamov-Tenegolts codes* (VT codes for short). Varshamov and Tenegolts constructed these codes in 1965 [11] and Levenshtein did a lot of research, gave bounds for the codes and proved that all VT codes are perfect in 1992 [6]. However, a conjecture about the size of the VT codes is still unproven, as mentioned later in this section. This motivation drives us to further study the codes. The definition of *checksum* and VT codes is as follows.

Definition 2.2.1. Let n and a be two integers, and $0 \leq a \leq n$. The *checksum* σ of a binary word $(x_1, x_2, ..., x_n)$ is $\sum_{i=1}^{n} ix_i$. The *Varshamov-Tenegolts code* $VT_a(n)$ contains all binary words such that

$$\sigma \equiv a(mod\, n+1)$$

.

Example 2.2.1. Let a=1 and n=4. Then $VT_1(4)$ is $\{1000, 1110, 0101\}$. For each of the codewords:

$$1000: \sum_{i=1}^{4} ix_i = 1+0+0+0 = 1 \equiv 1(mod\, 4+1)$$

$$1110: \sum_{i=1}^{4} ix_i = 1+2+3+0 = 6 \equiv 1(mod\, 4+1)$$

$$0101: \sum_{i=1}^{4} ix_i = 0+2+0+4 = 6 \equiv 1(mod\, 4+1)$$

As we can see, 1000 and 1110 each have 2 runs and 0101 has 4 runs. The total is $8 = 2^3$ runs, which is the number of binary strings of length 3. Recall that for a 1-deletion correcting code, the size of the sphere about the codewords is equal to the number of runs in the codewords. So all words of length 3 can be corrected. Therefore $VT_1(4)$ is a perfect 1-deletion correcting code, because it corrects 1 deletion and the number of runs is equal to the number of errors that can be corrected.

Abdel-Ghaffar and Ferreira [2] showed the VT codes can be provided by systematic encoding. A message of length i is encoded into a $VT_0(n)$ codeword of length $n = i + \lfloor log(i+2) \rfloor + 1$. The algorithm is as follows.

1. Find $\lfloor log(i+2) \rfloor + 1$ redundance bits at position 2^r, where $r = 1, 2, 3..., \lfloor log(i+2) \rfloor + 1$

2. Let the message be $(m_1 m_2 m_3 ... m_i)$ and the corresponding codeword be $(c_1 c_2 m_1 c_4 m_2 ... m_i)$ of length n. And $c_1 + 2c_2 + 4c_4 + ... \equiv a - (3m_1 + 5m_2 + 6m_3 + ... + nm_i) \equiv 1 \bmod n+1$. There might be multiple choices for $(c_1 c_2 c_4 ...)$, and any one works. So, we can use the lexicographically least one.

3. Then put c and m together to get the codeword.

Example 2.2.2. Suppose the message is 011001. Then the codeword is

$c = (c_1c_2 0 c_4 110 c_8 01)$ and $n = 10$. Let $a = 0$, by equation

$$c_1 + 2c_2 + 4c_4 + 8c_8 \equiv$$
$$0 - (3 \times 0 + 5 \times 1 + 6 \times 1 + 7 \times 0 +$$
$$9 \times 0 + 10 \times 1) \equiv 1 \ (mod\ 11)$$

We have $c_1c_2c_4c_8 = 1000$. Therefore the codeword is 1000110001.

The decoding algorithm, given by Levenshtein in 1965 [5], is as follows.

1. Assume a word $c = (c_1c_2...c_n)$ is transmitted and $c' = (c'_1c'_2...c'_{n-1})$ is received, where symbol c_p is missing. Suppose the checksum of c is σ, the checksum of c' is $\sigma' = \sum_{i=1}^{n-1} ic'_i$ and the weight of c' is $w = \sum_{i=1}^{n-1} c'_i$.

2. Since the deletion position p is no larger than the original code length n, therefore the deficiency in the checksum is no smaller than 0 and no larger than n. So we can get the original checksum $\sigma = \lceil \sigma'/(n+1) \rceil (n+1)$ and deficiency in the checksum $\Delta\sigma = \sigma - \sigma'$.

3. If $\Delta\sigma$ is less than or equal to w, that is a 0 was deleted, then we insert it to the left of the rightmost $\Delta\sigma$ 1's. Otherwise, a 1 was deleted, then we insert it to the right of the leftmost $(\Delta\sigma - w - 1)$ 0's.

Following the previous example of encoding, assume after transmission, the second bit of the codeword is missing, so that the receiver only received 100110001. The new checksum is $\sigma' = 1+4+5+9 = 19$ and the weight is $w =$

1+1+1+1 = 4. Then, the original checksum is $\sigma = \lceil 19/(10+1) \rceil (10+1) = 22$. The deficiency is $\Delta\sigma = 3$. Since $\Delta\sigma \leq w$, a 0 was deleted, insert 0 to the left of the rightmost 3 1's, that is between the 3rd and 4th bits. So, the received vector is decoded into 1000110001.

Here are some properties of VT codes.

Theorem 2.2.1. *(Levenshtein 1992 [6])*

$VT_a(n)$ are perfect 1-deletion correcting binary codes for any $n, a \in \mathcal{N}$ and $0 \leq a \leq n$.

Studying optimality of deletion-correcting codes is much harder than for Hamming distance codes, since, as we mentioned, the spheres of the codewords are not of fixed size. But still, some upper bounds are obtained for the VT codes.

From the definition of VT codes, the sum of every string is taken modulus $n+1$. This means the binary space is divided into $n+1$ subsets. Therefore, if the size of a VT code is denoted by $|VT_a(n)|$, then for a fixed n, the size of the largest VT code has

$$|VT_a(n)| \geq \frac{2^n}{n+1}$$

It is also known that

Theorem 2.2.2. *(Varshamov 1965 [11])*

$$|VT_0(n)| \geq |VT_a(n)| \text{ for all } 0 \leq a \leq n.$$

Therefore

$$|VT_0(n)| \geq \frac{2^n}{n+1}.$$

In contrast, Levenshtein [9] provided an approximate size for optimal deletion correcting codes. Let $A(n,t)$ be the size of the optimal t-deletion correcting binary code of length n. Then

$$A(n,t) \sim \frac{2^n}{n}, \ as \ n \rightarrow \infty$$

As a result, for large n, the $VT_0(n)$ codes are close to optimal. But it has yet to be proven that $VT_0(n)$ are always optimal. This remains a conjecture. See [9] for further discussions.

2.3 Tenegolts Codes

In the previous chapter, we introduced a perfect and nearly optimal 1-deletion correcting binary code. However, even though the VT codes have many nice properties, they can only handle the binary cases. In this chapter we will demonstrate *Tenegolts codes* [10](T codes for short), which can be viewed as an extension of VT codes correcting a single deletion for non-binary words. Indeed, we can translate non-binary words to binary ones to avoid the difficult construction of different codes. But the non-binary space is still worth studying, since it has various applications. For instance, DNA codes in bioin-

formatics are quaternary codes.

Definition 2.3.1. [10] Let $A = a_1 a_2 a_3 ... a_n$ be a non-binary string of length n. It is translated into binary string $\alpha_1 \alpha_2 \alpha_3 ... \alpha_n$ by the rule

$$\alpha_i = \begin{cases} 1, & \text{if } a_i \geq a_{i-1},\ i \geq 2 \\ 0, & \text{if } a_i < a_{i-1},\ i \geq 2 \end{cases}$$

α_1 can be either 0 or 1. In this thesis, the first bit will be set as 1.

Word $a : \{a_1, a_2, ..., a_n\}$ is a codeword of a $T_{\beta,\gamma}(n)$ code if

$$\sum_{i=1}^{n} a_i \equiv \beta \ (mod\ q)$$

and

$$\sum_{i=1}^{n} (i - l)\alpha_i \equiv \gamma \ (mod\ n),$$

where β, γ are integers and $0 \leq \beta < q$, $0 \leq \gamma < n$.

The last congruence shows how a T code relates to a VT code. If C is a codeword of $T_{\beta,\gamma}(n)$ and $B = b_1 b_2 ... b_n$ is the binary string translated from C, then removing the first bit from B obtains $B' = b'_1 b'_2 ... b'_{n-1}$ of length $n - 1$, in other words, B' is a subsequence of B. By the definition,

$$b_2 * 1 + b_3 * 2 + ... + b_n * (n - 1) \equiv \gamma \ (mod\ n)$$

and

$$b'_1 * 1 + b'_2 * 2 + ... + b'_{n-1} * n - 1 \equiv \gamma \,(mod\,(n-1)+1).$$

Therefore B' is a codeword of $VT_\gamma(n-1)$.

To ensure this code does work, Tenegolts proved its correctness.

Theorem 2.3.1. *(Tenegolts 1984 [10]) T codes are* 1*-deletion correcting non-binary codes.*

Proof. The value of the lost symbol and the position are two terms which decide the deletion. Suppose the alphabet has size q, the length of received word is $n-1$, the weight of the original word is w, the weight of the received word is w' and the value of the lost symbol is S.

Then $S = w - w'$ and $S < q$. Therefore $S \equiv \beta - w' \,(mod\, q)$, which is unique. Since the related binary string is a superstring of a codeword of $VT_\gamma(n-1)$, then the position of the deletion can be decided by the decoding algorithm of VT codes, which is also unique.

So, the correction is unique. As a result, T codes are 1-deletion correcting codes. ☐

From the proof, the decoding algorithm of T codes can be described as follows:

1. Let the weight of the received word be w', and the lost symbol $S \equiv \beta - w' \,(mod\, q)$.

2. Let $A_\alpha = \alpha_2\alpha_3...\alpha_{n-1}$ be the associated binary word, with the first bit

removed, determined by the received word. Suppose the weight of A_α is w_α and the checksum of A is $T\sigma_\alpha$.

3. Calculate the deficiency in the checksum $\Delta T\sigma = \lceil T\sigma_\alpha/(n-1)\rceil(n-1) - T\sigma_\alpha$.

4. If $\Delta T\sigma$ is less than or equal to w_α, that is a 0 was deleted, then we insert it to the left of the rightmost $\Delta T\sigma$ 1's. Otherwise, a 1 was deleted, then we insert it to the right of the leftmost $(\Delta T\sigma - w_\alpha - 1)$ 0's.

5. Suppose the insertion in the binary word is at position p, then the insertion in the q-ary word will be in the run which is in the binary word and contains p.

Example 2.3.1. Let $q = 3$, $n = 7$, $\beta = 2$ and $\gamma = 0$. Then $A = 1101200$ is a codeword since the weight w is

$$w = \sum_{i=1}^{7} = 1 + 1 + 2 + 2 = 2\ (mod\, 3)$$

and the checksum of associated binary word $A_\alpha = 1101101$ is

$$T\sigma = \sum_{i=1}^{7}(i-1)\alpha = 1 + 3 + 4 + 6 = 0\ (mod\, 7).$$

Assume the 1 at position 4 is deleted, so that $A' = 110200$ is received. The associated binary word is $A'_\alpha = 110101$, $w' = 1+1+2 = 4$, $w_\alpha = 1+1+1 = 3$

and checksum $T\sigma' = 1 + 3 + 5 = 9$. The symbol deleted is

$$S = 2 - w' = -2 = 1 \ (mod\, 3)$$

and the deficiency is

$$\Delta T\sigma = \lceil 9/(7-1) \rceil (7-1) - 9 = 12 - 9 = 5$$

which is greater than $w_\alpha = 3$. So a 1 is deleted. We insert it to the right of the leftmost $5 - 3 - 1 = 1$ 0's which is between the third and fourth bits. For the 3-ary word, a 1 is inserted in the third run which may be either between 0 at the third bit and 2 at the fourth bit or 2 at the fourth bit and 0 at the fifth bit. As the third run in the binary word has 2 bits, then we need a 2-bit non-decreasing sequence. Therefore, the only possible insertion is between the third bit and the fourth bit. Then we have 1101200 back again.

Tenengolts also discussed the size T codes. He showed that the lower bound for the size of the best code in the class of T codes is close to optimal.

Theorem 2.3.2. *(Tenegolts 1984 [10]) For fixed q and $n \to \infty$, the size of the optimal 1-deletion correcting code is*

$$M(q, n) \lesssim \frac{q^n}{(q-1) * n}$$

From the construction of the codes, we know the weight is taken modulus q and the checksum is taken modulus n. This means the entire deletion space is divided into $q * n$ subsets. Each subset is a T code. Therefore,

Theorem 2.3.3. *(Tenegolts 1984 [10]) The largest code has size:*

$$|T_{\beta,\gamma}(n)| \geqslant \frac{q^n}{q * n}$$

From the two inequalities, the largest T code is close to asymptotically optimal.

However, based on the T codes we have already constructed, non of them is optimal or perfect. Furthermore, Tenengolts did not mention how β and γ affect the size of the codes and how to get the best T code. These issues will be discussed in the next chapter.

2.4 Design Theory

In this section constructing codes from design theory is presented briefly, since some special designs provide perfect codes. This is a part of combinatorial mathematics. Bours [4] showed the close relationship between deletion correcting codes and design theory. Yin and his colleagues [12, 13, 14, 15] constructed some codes from designs. The definition of a *design* is given as follows:

Definition 2.4.1. A directed τ*-design* of type $\tau-(q,n,\lambda)$ is a pair $(Q,\mathcal{B})$ where Q is an alphabet of size q, $\mathcal{B}$ is a set of directed subsets(blocks) with size $n(n \leq q)$ of Q and every ordered subset of Q of size t is contained in exactly λ blocks.

Definition 2.4.2. A *directed balanced incomplete block design (DBIBD)* is a directed design of type $2-(q,n,\lambda)$.

Example 2.4.1. Let the alphabet $Q=(0,1,2,3)$ and let $\mathcal{B}=\{$(0,1,2,3), (1,0,3,2),(2,0,3,1),(3,0,2,1),(2,1,3,0),(3,1,2,0)$\}$. So $(Q,\mathcal{B})$ is a $3-(4,4,1)$ directed design. The alphabet has size 4, the size of each block is 4 and every ordered subset of size 3 is contained in exactly 1 block. The block (0,1,2,3) has four ordered subsets of size 3: (0,1,2),(0,1,3),(0,2,3) and (1,2,3).
Let $\mathcal{B}'$ =$\{$(0,1,2,3),(3,2,1,0)$\}$. Then, $(Q,\mathcal{B}')$ is a $(4,4,1)$ DBIBD, since every ordered set of size 2 over the alphabet Q is contained in exactly one block.

If we take $\lambda=1$, every ordered subset of size 2 is contained in exactly one block. This implies a DBIBD can be converted to a perfect deletion correcting code by adding all codewords which have only one symbol. So, we can define another type of perfect deletion correcting code.

Definition 2.4.3. A $(q,n,1)DBIBD$ is a perfect $(2,n,q)$ deletion correcting code T such that for every codeword c in T, c has no repeated symbols. Each directed subset in the DBIBD is a codeword in T. In other words, a

$(q, n, 1)DBIBD$ is a perfect deletion correcting code in which the coordinates are different.

Since a block in a DBIBD is a set and a set cannot have two repeated symbols, then the perfect code implied by the design has no repeated symbols in one word. If the set of all the codewords having only one symbol is C, then we have $T^*(2, n, q) = T(2, n, q) + C_1$, in which T^* is a perfect code with same coordinates.

Example 2.4.2. If we take the DBIBD in example 2.4.1, $\mathcal{B}'$ ={0123,3210} and let $C = \{0000, 1111, 2222, 3333\}$, then $T^* = \{0000, 1111, 2222, 3333, 0123, 3210\}$ is a perfect 2-deletion correcting quaternary code of length 4.

By this idea, Wang and Yin constructed perfect 5-deletion correcting codes of length 7 for some alphabets and Wang and Ji [13] constructed perfect 1-deletion correcting codes of length 4 for any arbitrary alphabet size.
As we can see, the perfect codes implied by DBIBDs have to correct $n-2$ deletions, and have a large alphabet, short length and small size ($n-2$ deletions means spheres are probably large and the size of the code is fairly small). However, this situation is not common in reality. So, this thesis will not focus on the construction using designs.

2.5 Two Trivial Construction and Extensions

This section will introduce two algorithms: the greedy algorithm and the backtracking algorithm. These two algorithms can construct or extend any

type of codes. But the codes produced by these two algorithms will not have an efficient encoding or decoding algorithm. The algorithms can only ensure the spheres will not intersect with each other. The *greedy* algorithm is as follows.

1. Initialize $d =$ minimum distance, $q =$ the size of the alphabet, $l =$ the length, and $value = 0$ to be the increment value. Also initialize a code C: If the program is constructing a new code, C is initialized as empty. If the program is extending, C is initialized as the code which will be extended;
2. For $value < q^l$ repeat Steps 3 and 4;
3. Translate $value$ to a q-ary l-length word w;
4. If w has distance no smaller than d to all codewords in C, add w to C, increase $value$ by the increment and the size of C;
5. The code C is the result.

The second step repeats q^l times, so the time complexity of the algorithm is $O(q^l)$. The complexity is exponential, based on length. If it is based on the size of the space, then it becomes linear.

The *backtracking* algorithm is shown as follows.

1. Initialize $d =$ the minimal distance, $q =$ the size of the alphabet, $l =$ the length, two empty codes $C_{cur.}$ and C_{best}, a code $C_{exi.}$(If the program

is constructing a new code, $C_{exi.}$ is initialized as empty. If the program is extending, $C_{exi.}$ is initialized as the code which will be extended.) and *value* is the decimal value of the last word in $C_{exi.}$ plus 1(If $C_{exi.}$ is empty, $value = 0$);

2. Repeat Steps 3-8;

3. For $value < q^l$ repeat Steps 4 and 5;

4. Translate *value* to a q-ary l-length word w;

5. If w has distance no smaller than d to all codewords in $C_{cur.}$ and all codewords in $C_{exi.}$, add w into $C_{cur.}$, increase the size of $C_{cur.}$ and *value*;

6. If $C_{cur.}$ is "better" than C_{best}, copy $C_{cur.}$ to C_{best};

7. If $C_{cur.}$ is empty, the program stops and C_{best} is the solution;

8. Set *value* to be the decimal value of the last word in $C_{cur.}$ plus 1. Remove the last word from $C_{cur.}$.

$C_{exi.}$ is the existing code which is being extended. C_{best} provides the solution of the algorithm. $C_{cur.}$ is the code which is being processed. If the program is constructing an optimal code, then "better" means the size of $C_{cur.}$ is larger than the size of C_{best}. If the program is constructing a perfect code, "better" means the $C_{cur.}$ corrects more errors than C_{best}.

The process of the algorithm is isomorphic to the depth-first search because the structure of a code can be shown as a tree. The root of the tree is the

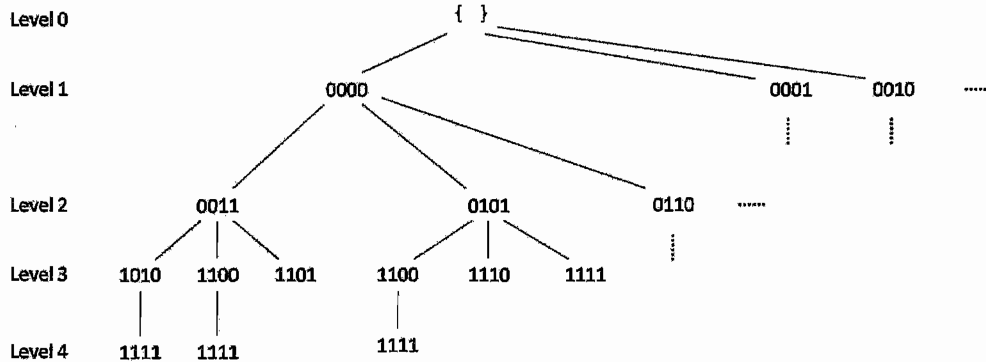

Figure 2.4: A part of a process of a backtracking algorithm.

word λ, which is the word of length zero. Each of the words on the path from the root downwards has distance no smaller than the minimum distance of the code. And each path of the tree is a code. So, the longest path of the tree provided by a depth-first search is the goal of the backtracking algorithm. Suppose $q = 2$, $l = 4$ and $d = 4$. Figure 2.4 shows an example of the structure. At level 1, all words are candidates since each of them has distance 4 to λ. At level 2, on the branch of 0000, some words like 0011, 0101, 0110 are candidates. When the process is at one leaf node, it will rolled back to the nearest fork and search the next branch. The search will rollback at the leaf node of path $\lambda \rightarrow 0000 \rightarrow 0101 \rightarrow 1111$, because the candidates are lexicographically increasing, even though the word 1100 has distance no smaller than d to each of the codewords.

The time complexity of this algorithm is $O(2^{q^l})$, which is extremely large, because it is an exhaustive search. This algorithm never constructs an optimal code of length 5 during the experiment, even after a whole month of

computing time. Thus for longer lengths, we only use this algorithm to extend the T codes. Since the T codes are very close to optimal, there are not very many words that can be extended. The time used for extension is acceptable. The program successfully extends the T codes, up to length 14. The results will be discussed in chapter 5.

Chapter 3

Difficulties in Construction of Perfect Codes

This chapter will demonstrate that finding a perfect code is usually very complicated.

The first example is the construction of ternary perfect codes of length 3 that corrects a single error. In order to make a code perfect, we need to analyze the structure of the errors and the candidates of the codewords. Here is a list of all words of length 2, which are the errors.

$$\begin{array}{ccc} 00 & 01 & 02 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \end{array}$$

There are three words with 1 run and six words with 2 runs. Let $R_{n,r}$ be the set of all words of length n with r runs and let $|R_{n,r}|$ be the size of the set.

Therefore $R_{2,1} = \{00, 11, 22\}$ and $R_{2,2} = \{01, 02, 10, 12, 20, 21\}$.

Here is a list of all words of length 3, which are the candidates to be codewords.

000	001	002	010	011	012	020	021	022
100	101	102	110	111	112	120	121	122
200	201	202	210	211	212	220	221	222

So, $|R_{3,1}| = 3$, $|R_{3,2}| = 12$ and $|R_{3,3}| = 12$. However, not all words in $R_{3,3}$ are of the same type. For example, 020 corrects 2 words with 2 runs (02,20) and 1 word with 1 run (00) while in contrast 012 corrects 3 words with 2 runs (01,02,12). We summarize the types of candidates in Table 3. The symbols in the "Type" column are not fixed, but represented by a letter to show a pattern, forwards or backwards. For example, 020 and 212 are both type ABA. We do not separate types that are the reverse of each other. For example, types AAB and ABB are not separate since both of them have 2 substrings, one with 1 run, and the other with 2 runs.

However, not all the candidates with the same type can be chosen at the same time. We also need to study the distance between each pair of candidates.

In Figures 3.1, 3.2 and 3.3, two words are connected if they have a common substrings by one deletion. In other words, they have distance 2 and so they cannot be selected as codewords together. The type of "AAA" is ignored, because it is trivial: all substrings are the same. In Figure 3.1, there are 3 cliques of size 4, which implies at most three words (one from each clique)

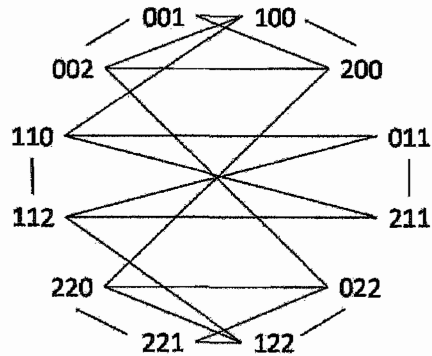

Figure 3.1: The Words of Type “AAB”

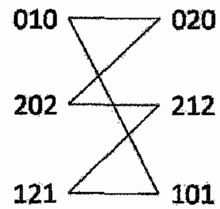

Figure 3.2: The Words of Type “ABA”

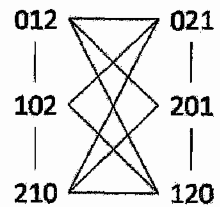

Figure 3.3: The Words of Type “ABC”

	Type	Size	Words	
$R_{3,1}$	AAA	3	000	111
			222	
$R_{3,2}$	AAB	12	001	002
			110	112
			220	221
			100	200
			011	211
			022	122
$R_{3,3}$	ABA	6	010	020
			101	121
			202	212
	ABC	6	012	021
			102	120
			201	210

Table 3.1: Different types of words of length 3.

can be chosen as codewords together. Similarly, no more than 3 words can be chosen from type "ABA", because there are 3 cliques of size 2, and no more than 2 from type "ABC", since there are 2 cliques of size 3.

Suppose the construction strategy is to consider the candidates with the most runs first, i.e. "ABA","ABC","AAB" then "AAA" and follow the backtracking algorithm. The stages are shown in Table 3.2.

Stage	Codeword Types	Covered errors	Uncovered errors
1	3×"ABA"	6×"AB" 3×"AA"	0-Perfect
	As shown in Figure 3.2, type "ABA" has 6 words and 3 can be chosen at the same time. Two perfect codes are found: $\{010, 202, 121\}$ and $\{020, 212, 101\}$		

2	2×"ABC"	6×"AB"	3×"AA"-Not perfect
	2 words of type "ABC" can be chosen.		
3	2×"ABC" 3×"AAA"	6×"AB" 3×"AA"	0 Perfect
	Type "ABA" or "AAB" cannot be chosen, since all "AB"'s are corrected. This stage has three different codes: $\{012, 210, 000, 111, 222\}$, $\{102, 201, 000, 111, 222\}$ and $\{120, 021, 000, 111, 222\}$.		
4	2×"ABA"	4×"AB" 2×"AA"	2×"AB" 1×"AA"
	The case of 2×"ABC" is finished.		
5	2×"ABA" 1×"AAB"	5×"AB" 3×"AA"	1×"AB"
	This code will never be perfect, since no word of length 3 can correct only a single type "AB" word.		
6	2×"ABA" 1×"AAA"	4×"AB" 3×"AA"	2×"AB"
	This case is replacing a type of "ABA" by "AAA" from the case of stage 1. As a result, if the "AAA" is 000, the remaining "AB" will be 12 and 21; if the "AAA" is 111, the remaining "AB" will be 02 and 20; and if the "AAA" is 222, the remaining "AB" will be 10 and 01. As we can see, using a single word of type "AAB" to handle this case is impossible. So, this case will never be perfect.		
7	1×"ABC"	3×"AB"	3×"AB" 3×"AA"
	The case of selecting two words with 3 runs is finished at step 6.		
8	1×"ABC" 2×"AAB"	5×"AB" 2×"AA"	1×"AB" 1×"AA"

<table>
<tr><td></td><td colspan="3">We cannot choose the third “AAB” because whatever the case is, the third one will have distance 2 to the “ABC”.</td></tr>
<tr><td rowspan="2">9</td><td>1×“ABC” 2×“AAB” 1×“AAA”</td><td>5×“AB” 3×“AA”</td><td>1×“AB”</td></tr>
<tr><td colspan="3">Never to be perfect, since no word of length 3 can correct only a single type “AB” word.</td></tr>
<tr><td rowspan="2">10</td><td>3×“AAB”</td><td>3×“AB” 3×“AA”</td><td>3×“AB”</td></tr>
<tr><td colspan="3">We have finished the cases of choosing codewords with 3 runs. And this case will never to be perfect, because the only way to handle 3 “AB”s is adding one word of “ABC”. But non of the “ABC” has distance larger than 2 to the chosen codewords.</td></tr>
<tr><td rowspan="2">11</td><td>2×“AAB”</td><td>2×“AB” 2×“AA”</td><td>4×“AB” 1×“AA”</td></tr>
<tr><td colspan="3">This case can never be perfect. It will not generate a code with enough total runs to correct all the errors. Therefore, the whole process stops.</td></tr>
</table>

Table 3.2: The construction of perfect codes of length 3.

A total of five perfect codes are found.

The process considers all cases of length 3 and discovers five perfect codes. However, this process cannot be performed if the length is longer than 3. Not only must the types of candidates be discussed, but also the types of errors. As the length grows linearly, the number of cases grows exponentially and

the situation becomes extremely complicated. Due to this reason, we did not manage to find a perfect code of length 4 by hand. Fortunately, the program found one by exhaustive search. Compared to other perfect codes (VT codes and the perfect code of length 3), this perfect code of length 4 does not have a regular structure. And it is totally different to the optimal code as shown in Figure 3.4. Therefore, as we can see, constructing a perfect code is very difficult. The first column is all the vectors of length 2, the second is all the vectors of length 3 with codewords of a perfect code in the solid blocks and the third one is all the codewords of a perfect code and an optimal code. The words in blocks are codewords of the perfect code, the shaded words are codewords of the optimal code. The word 0000 is in both the perfect code and the optimal code.

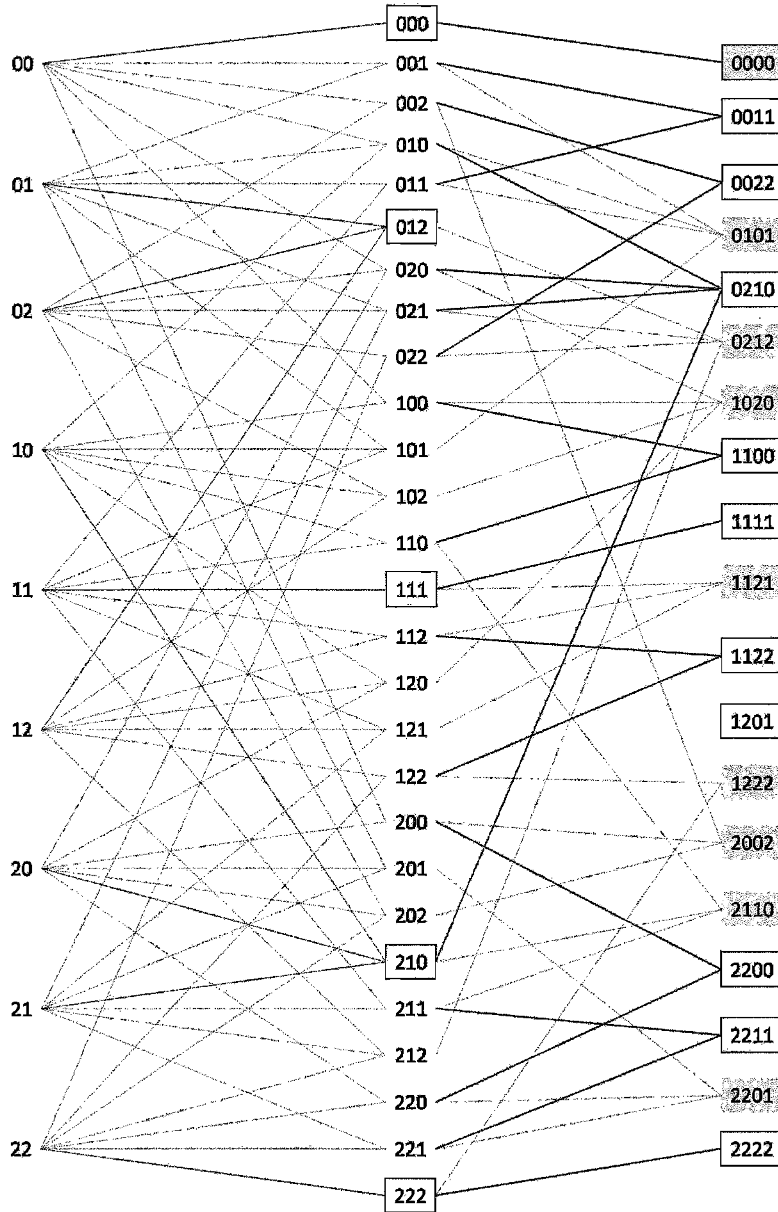

Figure 3.4: Perfect Codes of Length 3 and 4

Chapter 4

A Simple Construction for Ternary Codes

In this chapter, we will illustrate a simple construction of a class of deletion-correcting codes. The codes, named S codes, are ternary and correct 1 deletion error. The size of the alphabet will be fixed at 3 for most of the codes, since ternary codes are the main target of my research.

The methodology is similar to the construction of VT codes, but this code is neither perfect nor optimal. Furthermore, the size is much smaller than the size of T codes, which will be studied in Chapter 5.

4.1 Definitions of S Codes

Firstly, we need to define the *s-checksum* and *s-weight*, since the alphabet is no longer binary.

Definition 4.1.1. Let n be the length of a ternary word $\{v_1, v_2, v_3, ..., v_n\}$. The *s-checksum*

$$s\sigma = \sum_{i=1}^{n} m_i \quad , where \;\; m_i = \begin{cases} 0, & \text{if } v_i = 0 \\ n * i, & \text{if } v_i = 1 \\ n^2 + (n+1) * i, & \text{if } v_i = 2 \end{cases}$$

Definition 4.1.2. Let k_i be the number of i's in the word $v = \{v_1, v_2, v_3, ..., v_n\}$, for $i = 1$ or 2. The *s-weight* of v is

$$sw = k_1 * n + k_2 * (n+1)$$

The S codes are defined as follows:

Definition 4.1.3. A word $\{v_1, v_2, v_3, ..., v_n\}$ is a codeword of the *S code* of length n if s-checksum

$$s\sigma \equiv 0 \,(mod\,(2 * n^2 + a * n)), where\ a\ is\ a\ non-negative\ integer\ and\ a < n.$$

4.2 Correctness of S Codes

In this section, we are going to follow the proof of VT codes [11] to prove the S codes are 1-deletion correcting. The overall idea of the proof of the VT codes is to show the deficiency in the checksum is unique. Unfortunately, this cannot be applied to S codes directly, since different deletions can provide the same deficiency in the s-checksum. But we can still handle the exceptions by the following lemmas. The lemmas show that the smallest s-checksum of a supersequence created by inserting symbol 1 is always larger than the largest s-checksum of a supersequence created by inserting symbol 0 and the smallest s-checksum of a supersequence created by inserting symbol 2 is always larger than the largest s-checksum of a supersequence created by inserting symbol 1. This result ensures the codes are working even if some exceptions occur.

Let V be a ternary word of length $n-1$ and s-checksum (defined by Definition 4.1.1) $s\sigma$. Let V^* be a supersequence of V created by a single insertion. If a 0 is inserted at position p, $1 \leq p \leq n$, the s-checksum of V^* is $s\sigma_{0,p}$. "At position p" means after the insertion, the inserted symbol is the pth bit of V^*. In other words, the insertion is before the pth symbol in the original word. If a 1 is inserted, the s-checksum is $s\sigma_{1,p}$. If a 2 is inserted, the s-checksum is $s\sigma_{2,p}$. Since the insertion can happen at n different positions, $max(s\sigma_x)$ and $min(s\sigma_x)$, where $x = 0, 1, 2$, denote the maximum and minimum s-checksum created by inserting x. Suppose there are L_0 0's, L_1 1's and L_2 2's to the left of the insertion, and R_0 0's, R_1 1's and R_2 2's to the right of the insertion.

So, $p = L_0 + L_1 + L_2 + 1$ and $n = p + R_0 + R_1 + R_2$.

Lemma 4.2.1. $s\sigma_{0,1} = max(s\sigma_0)$.

Proof. Suppose the supersequence V^* is $v_1, v_2, ..., v_{p-1}, 0, v_p, v_{p+1}, ..., v_n$, created by inserting 0 at position p of V and suppose its s-checksum is $s\sigma_{0,p}$. Let $v_1, v_2, ..., v_{p-1}, v_p, 0, v_{p+1}, ..., v_n$ be another supersequence created by inserting 0 at $p + 1$ and let its s-checksum be $s\sigma_{0,p+1}$.

Therefore $s\sigma_{0,p} = s\sigma + n * R_1 + (n + 1) * R_2$.

If $v_p = 0$, then $s\sigma_{0,p+1} = s\sigma + 0 * (p + 1) + n * R_1 + (n + 1) * R_2 = s\sigma_{0,p}$.

If $v_p = 1$, then R_1 is decreased by 1 and $s\sigma_{0,p+1} = s\sigma + n * (R_1 - 1) + (n + 1) * R_2 < s\sigma_{0,p}$.

If $v_p = 2$, then R_2 is decreased by 1 and $s\sigma_{0,p+1} = s\sigma + n * R_1 + (n + 1) * (R_2 - 1) < s\sigma_{0,p}$.

Therefore $s\sigma_{0,p} \geq s\sigma_{0,p+1}$ for all $1 \leq p \leq n$.

Therefore $s\sigma_{0,1} = max(s\sigma_0)$. □

Example 4.2.1. Let $V = 1202$ and 0 is inserted,

if $p = 1$, then $V^* = 01202$, $n = 5$ and $s\sigma = n * 2 + n^2 + (n + 1) * 3 + n^2 + (n + 1) * 5 = 108$;

if $p = 2$, then $V^* = 10202$, $n = 5$ and $s\sigma = n * 1 + n^2 + (n + 1) * 3 + n^2 + (n + 1) * 5 = 103$;

if $p = 3$, then $V^* = 12002$, $n = 5$ and $s\sigma = n * 1 + n^2 + (n + 1) * 2 + n^2 + (n + 1) * 5 = 97$;

if $p = 4$, then $V^* = 12002$, $n = 5$ and $s\sigma = n * 1 + n^2 + (n + 1) * 2 + n^2 +$

$(n+1)*5 = 97$;

if $p = 5$, then $V^* = 12020$, $n = 5$ and $s\sigma = n*1 + n^2 + (n+1)*2 + n^2 + (n+1)*4 = 91$;

So the maximum s-checksum is 108, when $p = 1$.

Lemma 4.2.2. *v_p is the leftmost 0 $\iff$ $s\sigma_{1,p} = min(s\sigma_1)$.*

Proof. $s\sigma_{1,p} = s\sigma + n*p + n*R_1 + (n+1)*R_2$

If $v_p = 0$, then $s\sigma_{1,p+1} = s\sigma + n*(p+1) + n*R_1 + (n+1)*R_2 = s\sigma_{1,p} + n > s\sigma_{1,p}$.

If $v_p = 1$, then $s\sigma_{1,p+1} = s\sigma + n*(p+1) + n*(R_1 - 1) + (n+1)*R_2 = s\sigma_{1,p}$.

If $v_p = 2$, then $s\sigma_{1,p+1} = s\sigma + n*(p+1) + n*R_1 + (n+1)*(R_2 - 1) = s\sigma_{1,p} - 1 < s\sigma_{1,p}$.

Therefore if R_0 is decreased by 1 then the s-checksum is increased by n and if R_2 is decreased by 1 then the s-checksum is decreased by 1.

However, the length of the word is n. Thus R_2 will decrease by $n-1$ at most and the s-checksum will decrease by $n-1$ at most by the fact, which is smaller than increment of s-checksum, if R_0 is decreased by 1. In other words, assume the right neighbor of the insertion is 0. If we switch the insertion with the 0, R_0 is decreased by 1 and the s-checksum is increased by n. If we want to reduce the s-checksum, the only way is switching with 2's to the right. But each switching with 2 reduces the s-checksum by 1, and there are $n-1$ 2's in the word at most. Therefore the s-checksum will never be decreased back to the value before switching with 0.

Therefore if $s\sigma_{1,p} = min(s\sigma_1)$, then $L_0 = 0$, and

if v_p is the leftmost 0, then $s\sigma_{1,p} = min(s\sigma_1)$. □

Lemma 4.2.3. *v_{p-1} is the rightmost 0 $\iff$ $s\sigma_{1,p} = max(s\sigma_1)$.*

Proof. From the proof of Lemma 4.2.2, similarly, R_2 will decrease by $n-1$ at most and the s-checksum will decrease by $n-1$ at most, which is smaller than increment of s-checksum, if R_0 is decreased by 1.
Therefore if $s\sigma_{1,p} = max(s\sigma_1)$, then $R_0 = 0$, and
if v_{p-1} is the rightmost 0, then $s\sigma_{1,p} = min(s\sigma_1)$. □

Example 4.2.2. Let $V = 1202$ and 1 is insterted,
if $p = 1$, then $V^* = 11202$, $n = 5$ and $s\sigma = n*1+n*2+n^2+(n+1)*3+n^2+(n+1)*5 = 113$;
if $p = 2$, then $V^* = 11202$, $n = 5$ and $s\sigma = n*1+n*2+n^2+(n+1)*3+n^2+(n+1)*5 = 113$;
if $p = 3$, then $V^* = 12102$, $n = 5$ and $s\sigma = n*1+n^2+(n+1)*2+n*3+n^2+(n+1)*5 = 112$;
if $p = 4$, then $V^* = 12012$, $n = 5$ and $s\sigma = n*1+n^2+(n+1)*2+n*4+n^2+(n+1)*5 = 117$;
if $p = 5$, then $V^* = 12021$, $n = 5$ and $s\sigma = n*1+n^2+(n+1)*2+n^2+(n+1)*4+n*5 = 116$.
So the minimum s-checksum is 112, when 1 is inserted to the left of the leftmost 0, and the maximum s-checksum is 117, when 1 is inserted to the right of the rightmost 0. If there is no 0 in a word, $p = 1$ provides the largest s-checksum and $p = n+1$ provides the smallest s-checksum.

Lemma 4.2.4. *$s\sigma_{2,1} = min(s\sigma_2)$.*

Proof. $s\sigma_{2,p} = s\sigma + n^2 + (n+1)*p + n*R_1 + (n+1)*R_2$.

If $v_p = 0$, then $s\sigma_{2,p+1} = s\sigma + n^2 + (n+1)*(p+1) + n*R_1 + (n+1)*R_2 > s\sigma_{2,p}$.

If $v_p = 1$, then R_1 is decreased by 1 and $s\sigma_{2,p+1} = s\sigma + n^2 + (n+1)*(p+1) + n*(R_1 - 1) + (n+1)*R_2 > s\sigma_{2,p}$.

If $v_p = 2$, then R_2 is decreased by 1 and $s\sigma_{2,p+1} = s\sigma + n^2 + (n+1)*(p+1) + n*R_1 + (n+1)*(R_2 - 1) = s\sigma_{2,p}$.

Therefore $s\sigma_{2,p} \leq s\sigma_{2,p+1}$.

Therefore $s\sigma_{2,1} = min(s\sigma_2)$. □

Example 4.2.3. Let $V = 1202$ and 2 is insterted,

if $p = 1$, then $V^* = 21202$, $n = 5$ and $s\sigma = n^2 + (n+1)*1 + n*2 + n^2 + (n+1)*3 + n^2 + (n+1)*5 = 139$;

if $p = 2$, then $V^* = 12202$, $n = 5$ and $s\sigma = n*1 + n^2 + (n+1)*2 + n^2 + (n+1)*3 + n^2 + (n+1)*5 = 140$;

if $p = 3$, then $V^* = 12202$, $n = 5$ and $s\sigma = n*1 + n^2 + (n+1)*2 + n^2 + (n+1)*3 + n^2 + (n+1)*5 = 140$;

if $p = 4$, then $V^* = 12022$, $n = 5$ and $s\sigma = n*1 + n^2 + (n+1)*2 + n^2 + (n+1)*4 + n^2 + (n+1)*5 = 146$;

if $p = 5$, then $V^* = 12022$, $n = 5$ and $s\sigma = n*1 + n^2 + (n+1)*2 + n^2 + (n+1)*4 + n^2 + (n+1)*5 = 146$.

So the maximum s-checksum is 146, when $p = 5$.

Lemma 4.2.5. *$max(s\sigma_0) < min(s\sigma_1)$.*

Proof. By Lemma 4.2.1, $max(s\sigma_0) = s\sigma_{0,1} = s\sigma + n*k_1 + (n+1)*k_2$.

By Lemma 4.2.2,

$$\begin{aligned}
min(s\sigma_1) &= s\sigma + n*p + n*R_1 + (n+1)*R_2 \\
&= s\sigma + n*(L_0 + L_1 + L_2 + 1) + n*R_1 + (n+1)*R_2 \\
&= s\sigma + n*(L_1 + R_1) + (n+1)*(L_2 + R_2) - L_2 + n*L_0 + n \\
&= s\sigma + n*k_1 + n*k_2 + L_2 + n*L_0 + n
\end{aligned}$$

So,

$$\begin{aligned}
& min(s\sigma_1) - max(s\sigma_0) \\
= \;& (s\sigma + n*k_1 + n*k_2 + n*L_0 + L_2 + n) - (s\sigma + n*k_1 + (n+1)*k_2) \\
= \;& n*L_0 + L_2 + n - k_2 \\
= \;& n*L_0 + L_2 + n - (L_2 + R_2) \\
= \;& n*L_0 + n - R_2 \\
> \;& 0 \;\; (since \;\; n = L_0 + L_1 + L_2 + 1 + R_0 + R_1 + R_2 > R_2)
\end{aligned}$$

Therefore, $max(s\sigma_0) < min(s\sigma_1)$. □

Lemma 4.2.6. $max(s\sigma_1) < min(s\sigma_2)$

Proof. By Lemma 4.2.4, $min(s\sigma_2) = s\sigma_{2,1} = s\sigma + n^2 + (n+1)*p + n*k_1 + (n+1)*k_2$.

By Lemma 4.2.3, $max(s\sigma_1) = s\sigma + n*p + n*R_1 + (n+1)*R_2$.

So,

$$\begin{aligned}
& min(s\sigma_2) - max(s\sigma_1) \\
= \; & (s\sigma + n^2 + (n+1) * p + n * k_1 + (n+1) * k_2) \\
& -(s\sigma + n * p + n * R_1 + (n+1) * R_2) \\
= \; & n^2 + p + n * (k_1 - R_1) + (n+1) * (k_2 - R_2) \\
= \; & n^2 + p + n * L_1 + (n+1) * L_2 \\
> \; & 0 \;\; (since \;\; p = L_0 + L_1 + L_2 + 1 > 0)
\end{aligned}$$

Therefore, $max(s\sigma_1) < min(s\sigma_2)$. □

Example 4.2.4. From examples 4.2.1, 4.2.2 and 4.2.3, we can see $max(s\sigma_0) = 108$, $min(s\sigma_1) = 112$, $max(s\sigma_1) = 117$ and $min(s\sigma_2) = 146$.
So, $max(s\sigma_0) < min(s\sigma_1) < max(s\sigma_1) < min(s\sigma_2)$.

Lemma 4.2.7. *There is no word that has two supersequences created by one insertion, such that they have the same s-checksum.*

Proof. Assume we construct a function $f : p \to s\sigma$, then:
If 0 is inserted, f is monotonically decreasing (by Lemma 4.2.1). Therefore, it is impossible to produce two different supersequence by inserting 0.
If 2 is inserted, f is monotonically increasing (by Lemma 4.2.4). Therefore, it is also impossible.
If 1 is inserted, f is not monotonic. However, increasing R_0 by 1 contributes n to the total s-checksum while increasing R_2 by 1 contributes 1 (by Lemma

4.2.2). Therefore, if the two supersequences have the same s-checksum, then one of them must have n 2's and the other one has a single 0, which is impossible, because in this case the length is $n+1$, but not n.

Therefore, it is impossible to generate two supersequences with the same s-checksum by one insertion. □

Now, it is time to prove these codes correct a single deletion.

Theorem 4.2.1. *S codes are* 1*-deletion correcting.*

Proof. If 0 is deleted, the deficiency in s-checksum is $\Delta s\sigma = n * R_1 + (n+1) * R_2$, which is no larger than weight $sw = k_1 * n + k_2 * (n+1)$. If 1 is deleted, the deficiency is

$$\begin{aligned}
\Delta s\sigma &= n * p + n * R_1 + (n+1) * R_2 \\
&= n * (L_0 + L_1 + L_2 + 1) + n * R_1 + (n+1) * R_2 \\
&= n * (L_1 + R_1) + (n+1) * (L_2 + R_2) - L_2 + n * L_0 + n \\
&= n * k_1 + (n+1) * k_2 - L_2 + n * L_0 + n \\
&= sw - L_2 + n * L_0 + n
\end{aligned}$$

This value $\Delta s\sigma$ can be larger than n^2 and smaller than sw.

If 2 is deleted, the deficiency is $\Delta s\sigma = n^2 + (n+1) * p + n * R_1 + (n+1) * R_2$, which is no smaller than n^2.

Therefore, if $w < \Delta s\sigma < n^2$, the deletion must be 1.

If $\Delta s\sigma \leq w$ and $max(s\sigma_0) < s\sigma' + \Delta s\sigma$, then the deletion is 1, by Lemma

4.2.5.

If $max(s\sigma_0) \geq s\sigma' + \Delta s\sigma$, then 0 is deleted.

If $\Delta s\sigma \geq n^2$ and $max(s\sigma_1) < s\sigma' + \Delta s\sigma$, the 2 is deleted, by Lemma 4.2.6.

If $max(s\sigma_1) \geq s\sigma' + \Delta s\sigma$, then 1 is deleted.

To correct the error, we also need to decide the position of the deletion.

If 0 is deleted, the position can be decided by R_1 and R_2.

$$\begin{aligned} \Delta s\sigma &= n * R_1 + (n+1) * R_2 \\ &\equiv R_2 \ (mod\ n) \end{aligned}$$

So, $R_1 = \lfloor \Delta s\sigma / n \rfloor - R_2$.

If 1 is deleted, the position can be decided by R_0 and R_2.

$$\begin{aligned} \Delta s\sigma &= n * p + n * R_1 + (n+1) * R_2 \\ &= n * (p + R_1 + R_2) + R_2 \end{aligned}$$

So, $p + R_1 + R_2 = \lfloor \Delta s\sigma / n \rfloor$,

$R_0 = n - (p + R_1 + R_2) = n - \lfloor \Delta s\sigma / n \rfloor$ and

$R_2 = \Delta s\sigma - \lfloor \Delta s\sigma / n \rfloor * n$.

If 2 is deleted, the position can be decided by R_0 and R_1.

$$\begin{aligned}\Delta s\sigma &= n^2 + (n+1) * p + n * R_1 + (n+1) * R_2 \\ \Delta s\sigma - n^2 &= (n+1) * p + n * R_1 + (n+1) * R_2 \\ \Delta s\sigma - n^2 &= (n+1) * (p + R_1 + R_2) - R_1\end{aligned}$$

So, $R_1 = \lceil(\Delta s\sigma - n^2)/(n+1)\rceil * (n+1) - (\Delta s\sigma - n^2)$,

$p + R_1 + R_2 = \lceil(\Delta s\sigma - n^2)/(n+1)\rceil$ and

$R_0 = n - (p + R_1 + R_2) = n - \lceil(\Delta s\sigma - n^2)/(n+1)\rceil$.

Therefore, S codes can correct 1 deletion.

The uniqueness of the correction is ensured by Lemma 4.2.7.

Therefore, S codes are 1-deletion correcting. □

4.3 Decoding of S Codes

From the proof, we can conclude the decoding algorithm as follows:

1. Calculate the s-checksum $s\sigma'$ by using Definition 4.1.1.
2. Calculate the s-weight sw by using definition 4.1.2.
3. Calculate the original s-checksum $s\sigma = \lceil s\sigma'/(2 * n^2 + a * n)\rceil * (2 * n^2 + a * n)$, by using Definition 4.1.3 and the deficiency $\Delta s\sigma = s\sigma - s\sigma'$.
4. If $\Delta s\sigma \leq w$ and $max(s\sigma_0) < s\sigma' + \Delta s\sigma$, then the deletion is 0, by Lemma 4.2.5.

If $\Delta s\sigma \geq n^2$ and $max(s\sigma_1) < \sigma' + \Delta s\sigma$, then the deletion is 2, by Lemma 4.2.6.

Otherwise, the deletion is 1.

5. If the deletion is 0, get R_1 and R_2 by proof 4.2. And insert 0 to the left of the rightmost R_1 1's and rightmost R_2 2's.

   If the deletion is 1, get R_0 and R_2 by proof 4.2. And insert 1 to the left of the rightmost R_0 0's and rightmost R_2 2's.

   If the deletion is 2, get R_0 and R_1 by proof 4.2. And insert 0 to the left of the rightmost R_0 0's and rightmost R_1 1's.

Let us demonstrate this code by an example.

Example 4.3.1. Let $q = 3$, $n = 4$ and $a = 3$. 0102 is a codeword, since

$$\begin{aligned} s\sigma &= 0 * 1 + 4 * 2 + 0 * 3 + 4^2 + (4+1) * 4 \\ &= 44 \\ &\equiv 0 \, (mod\, 2 * 4^2 + 3 * 4) \end{aligned}$$

Suppose 012 is received after transmission. The s-checksum is $s\sigma' = 4 * 2 + 4^2 + (4+1) * 3 = 39$. The s-weight is $sw = 4 + 5 = 9$. And the original s-checksum is $s\sigma = \lceil 39/44 \rceil * 44 = 44$. The deficiency is $\Delta s\sigma = 44 - 39 = 5$, which is smaller than sw. So, we know the deletion is not 2. And we compute $max(s\sigma_0)$, which is the s-checksum of 0012 by inserting 0 at the first position. Then $max(s\sigma_0) = 4*3+4^2+(4+1)*4 = 48$, which is larger than the original

s-checksum. Therefore, we know that 0 was deleted. Then by following the last step of the algorithm, $R_2 = \Delta s\sigma \, mod \, n = 1$ and $R_1 = \lfloor \Delta s\sigma / n \rfloor - R_2 = 0$. The insertion is to the left of the rightmost 0 1's and 1 2's. Therefore, we decode to 0102.

After we construct the codes and prove the codes work, the size of S codes is studied. Unlike other codes, we do not have an efficient encoding algorithm. The codewords have to be generated by exhaustive search, which is scanning all possible words in the vector space to see which ones follow the definition. Table 4.3 shows the size of some of the S codes and the size of the codes constructed by a greedy algorithm (see Section 2.5). As we can see, S codes have size far smaller than the size of codes constructed by the greedy algorithm. This small size is caused by the definition of S codes. In Definition 4.1.3, the modulus we defined is $n^2 + a * n$, which means the entire vector space is divided into $n^2 + a * n$ subsets. The size of the whole vector space is q^n. As a result, we can never expect the size of the S codes to exceed the size of the T codes or the codes constructed by the greedy algorithm. Although S codes have a better decoding algorithm than greedy codes, we must conclude that the construction of the S codes is a failure from the point of view of trying to construct codes of large size. Table 4.3 shows the size of the S codes with length from 4 to 14. The parameter a is the coefficient of the formula in Definition 4.1.3. The column "Greedy" shows the size of the codes constructed by the greedy algorithm.

Length	$a = 2$	$a = 3$	Greedy
4	3	3	11
5	5	5	20
6	12	10	50
7	25	27	116
8	57	53	298
9	133	138	769
10	308	297	2010
11	733	731	5303
12	1832	1752	14120
13	4607	4487	38008
14	11833	11453	102704

Table 4.1: The Size of S Codes

Chapter 5

The Extension of T Codes

After the failure of S codes, the research turns in another direction. This approach is based on the construction of the T codes. Tenegolts discovered these nearly optimal codes in [10]. However the codes are neither optimal nor perfect, which is shown in Chapter 2. In this chapter, we will introduce some further properties of the T codes, which are found during the research, and present several ways to extend the size of the T codes by using these properties. As before, we are concentrating on ternary codes.

5.1 Properties and Hypothesis

Before considering the properties we would like to define *equivalent*, *symbol permutation* and *reverse* first. Similar definitions of equivalent can be found in [3].

Definition 5.1.1. A q-ary word $w = w_1, w_2, ..., w_n$ of length n is created by a *symbol permutation* of another q-ary word $w' = w'_1, w'_2, ..., w'_n$, if w can be transformed into w' by a function $g(w') = (f(w_1), f(w_2), ..., f(w_n))$, where f is a bijection $f : Q \mapsto Q$ and Q is the alphabet of w and w'.

Definition 5.1.2. A word $w = w_1, w_2, ..., w_n$ is the *reverse* of w', if $w' = w_n, w_{n-1}, ..., w_1$.

Example 5.1.1. Suppose we have two codes $C_1 = \{000, 101\}$ and $C_2 = \{010, 111\}$. The word 101 is the reverse of itself. It is a symbol permutation of 010, by changing the 1's and 0's. Because the word is binary, only 1 symbol permutation is possible other than the identity permutation. If it is q-ary, then it has $q! - 1$ symbol permutations other than the identity permutation.

Definition 5.1.3. Two codes C_1 and C_2 are *equivalent* if and only if each sphere in C_1 can be mapped to one and only one sphere in C_2. And two spheres s_1 and s_2 are mapped, if and only if the codeword of s_1 is mapped to the codeword of s_2 and each word in the sphere s_1 can be mapped to one and only one word in the sphere s_2. The mapping of two words can only be symbol permutation or reverse.

Example 5.1.2. Suppose there are two codes $C_1 = \{001, 110\}$ and $C_2 = \{200, 122\}$. In C_1, 001 corrects $\{00, 01\}$(so in the sphere of 001, there are three words $001, 00, 01,$) and 110 corrects $\{10, 11\}$ while in C_2, 122 corrects $\{12, 22\}$ and 200 corrects $\{20, 00\}$. We can establish a mapping:

Sphere of 001	$\leftrightarrow$	Sphere of 122	Sphere of 110	$\leftrightarrow$	Sphere of 200
00	$\leftrightarrow$	11	11	$\leftrightarrow$	00
01	$\leftrightarrow$	12	10	$\leftrightarrow$	20

As a result C_1 is equivalent to C_2.

With the above definitions, the following theorems are proved.

Theorem 5.1.1. *The symbol permutation of $VT_0(n)$ is itself if n is even or $VT_{\frac{n+1}{2}}(n)$ if n is odd.*

Proof. Let w be a codeword in $VT_0(n)$ with t 1's at position p_i, where $1 \leq i \leq t \leq n$ and w' be a codeword in the symbol permutation. The checksum for w is σ and for w' is σ'.

By the definition of symbol permutation, in w all the 1's are replaced by 0's and 0's are replaced by 1's. So w' has $n-t$ 1's. And $\sigma + \sigma' = \sum_{i=1}^{n} i$.

By the definition of VT codes, $\sigma = \sum_{i=1}^{t} p_i = l * (n+1) \equiv 0 \ (mod\, n+1)$, where l is a non-negative integer. Therefore, if n is even, then

$$\sigma' = \sum_{i=i}^{n} i - \sigma = \frac{n}{2}(n+1) - l(n+1) = (\frac{n}{2} - l)(n+1)$$

where $n/2$ and l are two integers. So $\sigma' \equiv 0 \ (mod\, n+1)$.

If n is odd then

$$\sigma' = \sum_{i=1}^{n} i - \sigma = \frac{n-1}{2}(n+1) - l(n+1) + \frac{n+1}{2} = (\frac{n-1}{2} - l)(n+1) + \frac{n+1}{2}$$

where $(n-1)/2$ and l are two integers. So $\sigma' \equiv \frac{n+1}{2} \ (mod\, n+1)$.

Therefore, the symbol permutation of $VT_0(n)$ is itself if n is even or $VT_{\frac{n+1}{2}}(n)$ if n is odd. □

Theorem 5.1.2. *The symbol permutation of $VT_0(n)$ is equivalent to $VT_0(n)$.*

Proof. If n is even, then the symbol permutation of $VT_0(n)$ is itself. And it is equivalent to itself.

If n is odd, each word w in $VT_0(n)$ has a symbol permuted word w' in $VT_{\frac{n+1}{2}}(n)$. The words w and w' have the same number of runs and same length for each pair of runs r from w and r' from w'. So the sphere of w is equivalent to the sphere of w'. Therefore the two codes are equivalent.

As a result, the symbol permutation of $VT_0(n)$ is equivalent to $VT_0(n)$. □

Then, we must ensure that some improvement can be applied to the T codes. So the optimality of the codes is studied.

Theorem 5.1.3. *Not all of the T codes are optimal.*

Theorem 5.1.4. *Not all of the T codes are perfect.*

These two theorems can be easily proved by counter examples. Table 5.1 gives the list of all the codewords in $T_{0,0}(5)$, all the uncovered errors and extended words discovered by the greedy algorithm.

The code $T_{0,0}(5)$ has 17 codewords. And 24 errors of length 4 cannot be corrected by the code, so it is not perfect. The two words $\{11001, 12200\}$ can be

Codewords		Errors			Extra words
00000	11112	0021	1012	2111	11001
00012	12102	0101	1100	2122	12200
00111	12222	0102	1101	2200	
00222	20010	0120	1121	2202	
01122	20121	0201	1200	2212	
02100	20220	0202	1201	2221	
10020	20220	0211	1211		
10110	22101	0212	1220		
10221		1001	2002		

Table 5.1: The codewords of $T_{0,0}(5)$, the uncovered errors and extra words that can be added into the code.

added into the code to extend the size, where 11001 corrects $\{1100, 1101, 1001\}$ while 12200 corrects $\{1220, 1200, 2200\}$. Alternatively, the word 12200 could be replaced by 22002, which corrects $\{2200, 2202, 2002\}$. So, the code $T_{0,0}(5) \cup \{11001, 12200\}$ has larger size than $T_{0,0}(5)$. This means $T_{0,0}(5)$ is not optimal. Actually, most of the codes generated during the experiment are neither optimal nor perfect.

After this, the sizes of different T codes with the same length is studied. Theorem 2.2.2 shows that different VT codes of the same length have different sizes. This property can be applied to the T codes.

Theorem 5.1.5. *Not all ternary T codes with the same length have the same size except that length is a power of 3.*

Proof. Theorem 2.3.3 shows the T codes actually divide the vector space into $q * n$ subsets. And for ternary we have $q = 3$. And if n is not a power of 3, then $3^n \not\equiv 0 \ (mod\, n)$. So $3^n \not\equiv 0 \ (mod\, 3 * n)$, and the sizes of the $3 * n$

subsets are not the same. □

This theorem can also be applied to an arbitrary alphabet, such that $q \neq 0 \ (mod\, n)$.

Now it comes to another question: how can we generate a ternary T code with largest size? To study this issue, we first construct the ternary T codes of length from 4 to 10 and all combinations of β and γ. The sizes of the codes are shown in Tables 5.2 - 5.9.

From Tables 5.3, 5.4, 5.6, 5.7 and 5.9, we see that the value of β does

$\gamma \backslash \beta$	0	1	2
0	5	3	3
1	2	3	3
2	2	3	3

Table 5.2: $n = 3$, β affects the size of codes.

$\gamma \backslash \beta$	0	1	2
0	7	7	7
1	6	6	6
2	8	8	8
3	6	6	6

Table 5.3: $n = 4$

$\gamma \backslash \beta$	0	1	2
0	17	17	17
1	16	16	16
2	16	16	16
3	16	16	16
4	16	16	16

Table 5.4: $n = 5$

$\gamma\backslash\beta$	0	1	2
0	41	39	39
1	40	42	42
2	38	39	39
3	46	42	42
4	38	39	39
5	40	42	42

Table 5.5: $n = 6$, β affects the size of codes.

$\gamma\backslash\beta$	0	1	2
0	105	105	105
1	104	104	104
2	104	104	104
3	104	104	104
4	104	104	104
5	104	104	104
6	104	104	104

Table 5.6: $n = 7$

not affect the size of the codes, although this is not the case in Tables 5.5 or 5.8. We conjecture that

Hypothesis 5.1.1. *The value of β does not affect the size of the T codes, if $\beta \not\equiv 0 \ (mod\ 3)$.*

The reason that β cannot be a multiple of 3 is that, by the association rule in the definition of the T codes, it is looking at decreasing and non-decreasing sequences, and for a ternary word, the longest decreasing sequence is 210 which has length 3. If this hypothesis is true, then it can be applied to non-ternary codes as follows.

Hypothesis 5.1.2. *The value of β does not affect the size of the q-ary T*

$\gamma\backslash\beta$	0	1	2
0	277	277	277
1	207	270	270
2	276	276	276
3	270	270	270
4	278	278	278
5	270	270	270
6	276	276	276
7	270	270	270

Table 5.7: $n = 8$

$\gamma\backslash\beta$	0	1	2
0	737	729	729
1	726	729	729
2	726	729	729
3	734	729	729
4	726	729	729
5	726	729	729
6	734	729	729
7	726	729	729
8	726	729	729

Table 5.8: $n = 9$, β affects the size of codes.

$\gamma\backslash\beta$	0	1	2
0	1961	1961	1961
1	1976	1976	1976
2	1960	1960	1960
3	1976	1976	1976
4	1960	1960	1960
5	1978	1978	1978
6	1960	1960	1960
7	1976	1976	1976
8	1960	1960	1960
9	1976	1976	1976

Table 5.9: $n = 10$

codes, if $\beta \neq 0 \pmod q$.

For the studied cases, code size is the largest when $\beta = 0$ and $\gamma = 2$ if $n = 4$ (Table 5.3); $\beta = 0$ and $\gamma = 0$ if $n = 5$ (Table 5.4); $\beta = 0$ and $\gamma = 3$ if $n = 6$ (Table 5.5); $\beta = 0$ and $\gamma = 0$ if $n = 7$ (Table 5.6); $\beta = 0$ and $\gamma = 4$ if $n = 8$ (Table 5.7); $\beta = 0$ and $\gamma = 0$ if $n = 9$ (Table 5.8); and $\beta = 0$ and $\gamma = 5$ if $n = 10$ (Table 5.9). We now have an hypothesis about the largest ternary T codes.

Hypothesis 5.1.3. *The size of a ternary T code is largest if* $\beta = 0$ *and*

$$\gamma = \begin{cases} 0, & \textit{if } n \textit{ is odd} \\ n/2, & \textit{if } n \textit{ is even} \end{cases}$$

This hypothesis is closely related to the property of VT codes. Theorem 2.2.2 shows that $VT_0(n)$ has the largest size. This means if n is even then the all 1-word is in the code, and if n is odd then the all 1 word is in the symbol permutation code which is equivalent to $VT_0(n)$, by Theorem 5.1.2. The association rule of the T codes translates a non-binary word into a word of a VT code. So if the associated VT code is maximal, the T code probably is. As a result, if a codeword of a T code is associated with the all 1-word in a VT code, then that code has maximal size. In other words, γ has to be 0 if n is odd and $n/2$ if n is even.

This hypothesis helps us to specify a special class of T codes.

Definition 5.1.4. $T_s(n)$ is a ternary T code, such that $\beta = 0$ and

$$\gamma = \begin{cases} 0, & \text{if } n \text{ is odd} \\ n/2, & \text{if } n \text{ is even} \end{cases}$$

If the size of this kind of special T codes can be extended, then some larger codes can be constructed.

The tables also show that the values in Tables 5.4 and 5.6 are very regular. This relates to Fermat's little theorem.

Theorem 5.1.6. *(Fermat's little theorem) For any integer a and prime number p, $a^p \equiv a \pmod p$.*

So, for length 5, the entire vector space has size 3^5 and it is divided into $3 * 5$ subsets. By the theorem, $3^5 \equiv 3 \, (mod\, 5)$. And by Hypothesis 5.1.1, β does not affect the size of the codes, that is, all the codes with the same γ will have the same size, so all the codes should have the same size of $(3^5 - 3)/(3 * 5) = 16$ except for three codes(all 0 word, all 1 word and all 2 word) of size 17, which have $\gamma = 0$, by Hypothesis 5.1.2.

Here we partly prove Hypothesis 5.1.3.

Theorem 5.1.7. *If β does not affect the size of T codes and n is a prime number with exception 3, then*

$$|T_{\beta,\gamma}(n)| = \frac{3^{n-1}-1}{n} + 1, \text{ where}$$

$$\gamma = \begin{cases} 0, & \text{if } n \text{ is odd} \\ n/2, & \text{if } n \text{ is even} \end{cases}$$

and

$$|T_{\beta,\gamma}(n)| = \frac{3^{n-1}-1}{n}, \text{ where}$$

$$\gamma \neq \begin{cases} 0, & \text{if } n \text{ is odd} \\ n/2, & \text{if } n \text{ is even} \end{cases}$$

This theorem assumes a part of the hypothesis, which is β does not affect the size, is correct and discusses the case, such that n is a prime number other than 3. The exception happens, since after we remove the all 0, all 1 and all 2 words, there are $3^3 - 3 = 24$ words left and $24 \not\equiv 0 \mod q * n$. If we can find a bijection between different T codes with the same length, then we can prove the theorem. So, the theorem can be rewritten as follows.

Theorem 5.1.8. *If $w = w_1w_2...w_n$ (w is not one of all 0, all 1 and all 2 words) is a codeword of $T_{\beta,\gamma}(n)$ (n is a prime except 3), then the cyclic word of w: $w' = w_i...w_nw_1...w_{i-1}$ $(1 < i \leq n)$ is a codeword of $T_{\beta',\gamma'}(n)$, $\beta = \beta'$ and $\gamma \neq \gamma'$.*

If $i = 1$, then $w' = w_1w_2...w_n$, which is the same as w. So the range of i ensures that w and w' are different.

Proof. By definition 2.3.1,

$$\begin{aligned}
\beta' &\equiv \sum_{t=i}^{n} w_t + \sum_{t=1}^{i-1} w_t \bmod q \\
&\equiv \sum_{t=1}^{n} w_t \bmod q \\
&= \beta
\end{aligned}$$

Let the associated binary string of w be $\alpha = \alpha_1\alpha_2...\alpha_n$ with checksum σ, and the associated binary string of w' be $\alpha' = \alpha_i...\alpha_n\alpha_1...\alpha_{i-1}$ with checksum σ'. Since all the symbols of $\alpha_i...\alpha_n$ are moved $n-i+1$ positions to the left and all the symbols of $\alpha_1...\alpha_{i-1}$ are moved $i-1$ positions to the right from α,

$$\begin{aligned}
\sigma' - \sigma &= (n-i+1)\sum_{t=1}^{i-1} \alpha_t - (i-1)\sum_{t=i}^{n} \alpha_t \\
&= n\sum_{t=1}^{i-1} \alpha_t - (i-1)\sum_{t=1}^{i-1} \alpha_t - (i-1)\sum_{t=i}^{n} \alpha_t \\
&= n\sum_{t=1}^{i-1} \alpha_t - (i-1)\sum_{t=1}^{n} \alpha_t
\end{aligned}$$

Since the all 0, all 1 and all 2 words are excluded, not all symbols in α are 1's.

$\therefore \sum_{t=1}^{n} \alpha_t < n$

$\therefore \sum_{t=1}^{n} \alpha_t \not\equiv \bmod n$

$\because n$ is a prime, $1 < i \leq n$.

Also, $i - 1 \not\equiv 0 \bmod n$ $\therefore (i-1)\sum_{t=1}^{n} \alpha_t \not\equiv 0 \bmod n$

$\because \gamma \equiv \sigma \bmod n$, $\gamma' \equiv \sigma' \bmod n$

$\therefore \gamma \neq \gamma'$. □

Also we need to prove the reduction of the two theorems is correct.

Theorem 5.1.9. *Theorem 5.1.8 implies Theorem 5.1.7.*

Proof. By Fermat's little theorem, $(3^n - 3) \equiv 0 \bmod n$ when n is a prime. And $(3^n - 3) \equiv 0 \bmod 3$, so $(3^n - 3) \equiv 0 \bmod 3 * n$. This implies there is an even partition of the vector space after we remove the all 0, all 1 and all 2 words. Also by Theorem 5.1.8, a word w has $n - 1$ cyclic words. No two of these n words(w and its cyclic words) are in the same T code. So these n words are separated evenly by different values of γ. Therefore if the three words are excluded, and if β does not affect the size, then all of the codes have the same size.

For the cases of the all 0, all 1 and all 2 words, n is a prime and larger than 3. Then $\gamma = 0$ which is the same, $\beta = 0$ for the all 0 word, $\beta \equiv n \bmod 3$ for the all 1 word and $\beta \equiv 2n \bmod 3$ for the all 2 word, which are different. So each of the three word belongs to different $T_{\beta,0}(n)$ codes.

After a summary, we have Theorem 5.1.7. □

Returning to the $T_s(n)$ codes again, we construct all the codes from length 4 to length 18 and consider all the error vectors that cannot be corrected. One very interesting property is found. The uncovered errors of $T_s(4)$, $T_s(5)$ and $T_s(6)$ are listed in Table 5.10.

From the table we can find that 021 cannot be corrected by $T_s(4)$, 0021 cannot

$T_s(4)$	$T_s(5)$		$T_s(6)$			
021	0021	1200	00021	02012	11120	21000
101	0101	1201	00102	02111	11201	21001
110	0102	1211	00120	02122	11210	21002
202	0120	1220	00201	02200	12000	21011
220	0201	2002	00210	02210	12011	21012
	0202	2111	01002	02212	12022	21022
	0211	2122	01100	02220	12202	21112
	0212	2200	01110	10001	12210	22002
	1001	2202	01202	10012	12211	22112
	1012	2212	01210	10222	12221	22201
	1100	2221	01211	11001	20002	22210
	1101		01221	11012	20111	
	1121		02001	11102	20122	

Table 5.10: The uncovered errors of $T_s(4)$, $T_s(5)$ and $T_s(6)$.

be corrected by $T_s(5)$ and 00021 cannot be corrected by $T_s(6)$. Indeed, as the length grows longer, the run of 0's in the error increases simultaneously. Similarly 101, 1001 and 10001 also cannot be corrected. With this discovery, an *extendable error* is defined as

Definition 5.1.5. If a word $w = w_1w_2...w_{n-1}$ is an *extendable error* of $T_s(n)$ at position p, then the word $w = w_1w_2...w_{p-1}\overbrace{00...0}^{i}w_pw_{p+1}...w_{n-1}$ is an error vector that cannot be corrected by $T_s(n+i)$.

If non-zero symbols are inserted, the weight of the words and value of β are changed. Then the word will no longer be a codeword of $T_s(n)$. This is the reason that the symbol 0 is the only possible extension.

To discuss the extendable errors, we have to define *palindromes*.

Definition 5.1.6. A *palindrome* is a word $w_1w_2w_3...w_{n-1}w_n$, such that $w_i = w_{n-i+1}$ for every $1 \leq i \leq n$.

In other words, a palindrome is a word in which the suffix of any given length i is the inverse of the prefix of the same length. For example, 1001001 and 010010 are two palindromes.

After an examination of all uncovered errors of $T_s(6)$, 7 types of extendable errors are discovered (see Table 5.11). In the table, "Word" means the words are codewords of VT codes. "Decode" means the decoding process of the VT codes. "Binary" means the associated binary string. "Asso." means the association rule by the definition of T codes. "Error" means the error vectors which cannot be corrected by $T_s(n)$ codes. The error vectors are just some examples. Usually more than one error that can be associated with the same binary word.

From the table we can see that after the decoding process of VT codes, the first and the third cases are already palindromes without any extension; the second case is also a palindrome with no extension, but after one extension it becomes a different palindrome and it returns to the previous type of palindrome again after two extensions; the fourth and the seventh are always non-palindromes; the fifth is a non-palindrome at the beginning, but turns to a palindrome after one extension and the sixth is a palindrome originally, but becomes a non-palindrome later. Even with these many cases, one thing is unchanged. After several extensions, the binary codeword is always extended by a single one at the middle run. As a result, we have the following

		n=6	extension	n=7	extension	n=8
1.	Word	101110	⟶	1011110	⟶	10111110
	Decode	↑		↑		↑
	Binary	11110	⟶	111110	⟶	1111110
	Asso.	↑		↑		↑
	Error	01221	⟶	001221	⟶	0001221
2.	Word	110101	⟶	1110011	⟶	11101011
	Decode	↑		↑		↑
	Binary	11011	⟶	111011	⟶	1111011
	Asso.	↑		↑		↑
	Error	02001	⟶	002001	⟶	0002001
3.	Word	110101	⟶	1101101	⟶	11011101
	Decode	↑		↑		↑
	Binary	11011	⟶	110111	⟶	1101111
	Asso.	↑		↑		↑
	Error	02012	⟶	020012	⟶	0200012
4.	Word	111000	⟶	1110100	⟶	11101100
	Decode	↑		↑		↑
	Binary	11100	⟶	111010	⟶	1110110
	Asso.	↑		↑		↑
	Error	02210	⟶	022010	⟶	0220010
5.	Word	111000	⟶	1001100	⟶	10011100
	Decode	↑		↑		↑
	Binary	11100	⟶	101100	⟶	1011100
	Asso.	↑		↑		↑
	Error	11210	⟶	101210	⟶	1001210
6.	Word	110101	⟶	1001011	⟶	10011011
	Decode	↑		↑		↑
	Binary	11011	⟶	101011	⟶	1011011
	Asso.	↑		↑		↑
	Error	12011	⟶	102011	⟶	1002011
7.	Word	100011	⟶	1001011	⟶	10011011
	Decode	↑		↑		↑
	Binary	10011	⟶	101011	⟶	1011011
	Asso.	↑		↑		↑
	Error	21000	⟶	201000	⟶	2001000

Table 5.11: 7 examples of extendable errors.

	Uncovered Error	Extendable Errors
$T_s(4)$	5	3
$T_s(5)$	24	14
$T_s(6)$	51	37
$T_s(7)$	215	172
$T_s(8)$	620	514
$T_s(9)$	1915	1651
$T_s(10)$	5845	5098
$T_s(11)$	17910	15958
$T_s(12)$	53833	48228
$T_s(13)$	163548	148791
$T_s(14)$	492932	448938
$T_s(15)$	1487723	1370498
$T_s(16)$	4482358	4128126
$T_s(17)$	13504926	12548478
$T_s(18)$	40649351	37748386

Table 5.12: The Number of Uncovered Errors and Extendable Errors.

hypothesis.

Hypothesis 5.1.4. *If w is an extendable error of length n ($n \to \infty$), t is the VT code of the associated string of w, w' is the error by one extension from w and t' is the VT code of the associated string of w', then w' is a supersequence of w created by inserting 1 between the $\lceil n/2 \rceil th$ bit and the $\lceil n/2 \rceil + 1th$ bit.*

With this hypothesis, our program exhaustively searched all the uncovered errors from $T_s(4)$ to $T_s(18)$. The results are listed in Table 5.12. The table shows that the numbers of uncovered errors and extendable errors are increasing exponentially at the same time. And if we can construct a code-

word from these extendable errors then the codeword can also be extended. So, a larger code can be constructed by adding this codeword to $T_s(n)$.

5.2 One Extendable Codeword

This section will demonstrate the first extendable codeword: 2222202211, which can be added into $T_s(n)$, for $n \geq 10$. This is the only extendable codeword which is discovered so far. The process of looking for such extendable codewords includes: 1) search all the extendable errors; 2) search among the extendable errors of length n to see if there is a word of length $n+1$ such that all of its subsequences by 1-deletion are extendable errors; 3) check if the extension positions are in the same run. We will also discuss why it can be extended and how to extend.

Theorem 5.2.1. *The word* 2222202211 *is an extendable codeword for* $T_s(n)$, *where* $n \geq 10$ *and the extension position is between the first and second runs.*

As a result, 2222202211 can be added to $T_s(10)$, 22222002211 can be added to $T_s(11)$, 222220002211 can be added to $T_s(12)$ and so on.

In order to prove this theorem, we need to prove that all substrings (namely 222202211, 222222211, 222220211 and 222220221) of the word are extendable errors which are uncovered by $T_s(n)$ and that the extension positions are the same.

Lemma 5.2.1. 222202211 *cannot be corrected by* $T_s(10)$.

Proof. $w_9 = 222202211$ will be transformed into $w_{s9} = 111101101$ by the association rule (2.3.1).

$w_{s9} = 111101101$ will be decoded into $w_{v9} = 1111011010$ by the decoding algorithm of VT codes. The insertion is a 0 at the end of the word.

By the decoding algorithm of T codes, 1 will be added into the last run of the ternary word. Since the last run is a run of 1's, 2222022111 will be the only possibility.

However, the associated binary string is 1111011011, which is not the same as $w_{v9} = 1111011010$.

Therefore, 222202211 cannot be corrected by $T_s(10)$. □

Lemma 5.2.2. *Let* $w_n = 2222\underbrace{0...0}_{m}2211$ *be the extendable error after m extensions and let the length of w be n. Then w cannot be corrected by* $T_s(n)$.

Proof. w_n will be associated with $w_{sn} = 1111\underbrace{01...1}_{m}1101$ by the association rule.

Let the checksum of w_s be $T\sigma_n$, so $T\sigma_9 = 25$ by Definition 2.3.1. Every extension inserts a single 1 between the $(n-4)$th bit and the $(n-3)$th bit. Thus each bit inserted contributes $n-4$ to the checksum. Since the suffix has three 1's, the checksum is increased by $n-4+3$ from length n to length $n+1$.

As a result,

$$\begin{aligned} T\sigma_n &= T\sigma_{n-1} + n - 2 \\ &= T\sigma_9 + \sum_{i=8}^{n-2} i \\ &= T\sigma_9 + \frac{(n+6)(n-9)}{2} \\ &= 25 + \frac{n^2 - 3n - 54}{2} \\ &= \frac{n^2 - 3n - 4}{2} \\ &= \frac{(n+1)(n-4)}{2} \end{aligned}$$

By the definition of T_s codes, $\gamma = 0$ if the length of the codeword is odd and $\gamma = (n+1)/2$ otherwise. The length of the error vector is n, so the length of the codeword is $n + 1$.

If $n+1$ is even, then

$$\begin{aligned}
& T\sigma_n \bmod (n+1) \\
\equiv\ & (T\sigma_n - \frac{n+1}{2} + \frac{n+1}{2}) \bmod (n+1) \\
\equiv\ & (\frac{(n+1)(n-4)}{2} - \frac{n+1}{2} + \frac{n+1}{2}) \bmod (n+1) \\
\equiv\ & (\frac{(n+1)(n-5)}{2} + \frac{n+1}{2}) \bmod (n+1) \\
\equiv\ & (\frac{(n+1)(n-5)}{2} \bmod (n+1) + \frac{n+1}{2} \bmod (n+1)) \bmod (n+1) \\
\equiv\ & (0 + \frac{n+1}{2}) \bmod (n+1) \\
\equiv\ & \frac{n+1}{2} \bmod (n+1) \\
=\ & \gamma
\end{aligned}$$

If $n+1$ is odd, then

$$\begin{aligned}
T\sigma_n \bmod (n+1) &\equiv \frac{(n+1)(n-4)}{2} \bmod (n+1) \\
&\equiv 0 \bmod (n+1) \\
&= \gamma
\end{aligned}$$

Therefore, $T\sigma_n \bmod (n+1) - \gamma \equiv T\sigma_9 \bmod (9+1) - 0$.

So, after the decoding algorithm of VT codes, w_{sn} will be decoded into $w_{vn} = 1111\underbrace{01...1}_{m}11010$. The insertion is the same as w_{v9}, which means if w_9 cannot be decoded by $T_s(10)$, then w_n cannot be decoded by $T_s(n)$.

Therefore, w_n cannot be decoded by $T_s(n)$. $\square$

To summarize, we have

Lemma 5.2.3. 222202211 *is an extendable error, and the extension position is between the 4th bit and the 5th bit.*

With similar proofs, we have the following lemmas.

Lemma 5.2.4. 222222211 *is an extendable error, and the extension position is between the 5th bit and the 6th bit.*

Lemma 5.2.5. 222220211 *is an extendable error, and the extension position is between the 5th bit and the 6th bit.*

Lemma 5.2.6. 222220221 *is an extendable error, and the extension position is between the 5th bit and the 6th bit.*

Since all the substrings are extendable and the extension positions are in the same run, Theorem 5.2.1 is proved.

Chapter 6

Results and Implementations

In this chapter, we will illustrate some results produced by our programs and the implementations of our research.

6.1 Results

Table 6.1 shows the size of ternary 1-deletion correcting codes, including the codes constructed by the greedy algorithm in 2nd row, the number of errors which cannot be covered by greedy in 3rd row, the T_s codes and uncovered errors in 4th and 5th row, the number of words extended by the greedy algorithm and the backtracking algorithm based on T_s codes in 6th and 8th row. The seventh row is the number of errors which are not covered by the codes after extension by the greedy algorithm. The last row is the number of errors which are not covered by the codes after extension by the backtracking

Length	5	6	7	8	9	10	11	12	13
Greedy	20	50	116	298	769	2010	5303	14120	38008
Un. Gre.	13	51	181	568	1853	6070	19526	62598	197808
T_s	17	46	105	278	737	1978	5369	14800	40881
Un. T_s	24	50	215	620	1914	5845	17910	53832	163548
E. Gre.	2	0	4	5	14	17	28	38	68
SU	18	50	203	596	737	5756	17765	53636	163164
E. BT	2	0	4	5	14	17	28	41	68
SU	18	50	203	596	737	5756	17765	53621	163164

Table 6.1: The Comparison of Different Codes

algorithm. As we can see, the size of T_s codes is larger than the size of the codes constructed by the greedy algorithm after the length is longer than 11. The size of T_s codes is also growing exponentially. This ensures that the size of T codes is getting close to optimal as the length increases (see Theorems 2.3.2 and 2.3.3). This table also shows the two ways of extension, greedy and backtracking, based on T_s codes do not have any difference except when the length is 12. The reason is that the T_s codes have covered most of the errors already and there is no space for a large extension. As a result, if we need to extend the T codes, the backtracking algorithm provides the optimal extension, but with a time cost that is extremely large, while the greedy algorithm provides almost the same extensions with very little time cost.

Tables 6.2- 6.7 show the size of the backtracking extension based on different T codes. The extensions in Table 6.7 are still unknown when $\beta = 0$. Our program has run more than one month but with no result. The equations in the tables mean that the size of T codes plus the size of the backtracking extension. After the extension, the extended T_s codes are usually the largest,

$\gamma\backslash\beta$	0	1	2
0	17+2=19	17+2=19	17+2=19
1	16+1=17	16+1=17	16+1=17
2	16+3=19	16+2=18	16+2=18
3	16+2=18	16+3=19	16+2=18
4	16+1=17	16+1=17	16+1=17

Table 6.2: The Optimal Extensions of $T(5)$

$\gamma\backslash\beta$	0	1	2
0	41+3=44	39+5=44	39+5=44
1	40+3=43	42+1=43	42+1=43
2	38+6=44	39+5=43	39+5=44
3	46+0=46	42+1=43	42+1=43
4	38+6=44	39+5=44	39+5=44
5	40+3=43	42+1=43	42+1=43

Table 6.3: The Optimal Extensions of $T(6)$

but not always. When the length is 8, the extended $T_{0,0}$ and $T_{0,1}$ codes have larger size than the extended T_s code, even though the $T_{0,0}$ and $T_{0,1}$ codes are smaller than the T_s code.

$\gamma\backslash\beta$	0	1	2
0	105+4=109	105+4=109	105+4=109
1	104+2=106	104+2=106	104+2=106
2	104+9=113	104+5=109	104+5=109
3	104+4=108	104+4=108	104+4=108
4	104+4=108	104+4=108	104+4=108
5	104+5=109	104+5=109	104+9=113
6	104+2=106	104+2=106	104+2=106

Table 6.4: The Optimal Extensions of $T(7)$

$\gamma\backslash\beta$	0	1	2
0	277+8=285	277+8=285	277+6=283
1	270+11=281	270+9=279	270+9=279
2	276+2=278	276+3=279	276+4=280
3	279+6=279	270+13=283	270+10=280
4	278+5=283	278+5=283	278+2=280
5	270+13=283	270+9=279	270+10=280
6	276+3=279	276+2=278	276+4=280
7	270+9=279	270+11=281	270+9=279

Table 6.5: The Optimal Extensions of $T(8)$

$\gamma\backslash\beta$	0	1	2
0	737+14=751	729+17=746	729+17=746
1	726+10=736	729+14=743	729+12=741
2	726+16=742	729+10=739	729+9=738
3	734+21=755	729+9=738	729+6=735
4	726+8=734	729+12=741	729+12=741
5	726+8=734	729+12=741	729+12=741
6	734+21=755	729+6=735	729+9=738
7	726+16=742	729+9=738	729+10=739
8	726+10=736	729+12=741	729+14=743

Table 6.6: The Optimal Extensions of $T(9)$

6.2 Implementation

All the programs are developed in C programming language based on MinGW and WindowsXP. The main achievements of our programs are as follows.

1. The greedy algorithm for constructing and extending codes;

2. The backtracking algorithm for constructing and extending codes;

3. A program for S codes, which contains the exhaustive encoding algo-

$\gamma \backslash \beta$	0	1	2
0	Unknown	Unknown	Unknown
1	1976+12=1988	1976+11=1987	1976+17=1993
2	1960+19=1979	1960+26=1986	1960+22=1982
3	1976+19=1995	1976+13=1989	1976+14=1990
4	1960+25=1985	1960+20=1980	1960+22=1982
5	1978+17=1995	1978+14=1992	1978+17=1995
6	1960+22=1982	1960+20=1980	1960+25=1985
7	1976+14=1990	1976+13=1989	1976+19=1995
8	1960+22=1982	1960+26=1986	1960+19=1979
9	1976+17=1993	1976+11=1987	1976+12=1988

Table 6.7: The Optimal Extensions of $T(10)$

rithm, the decoding algorithm, one recorder for the size of the codes, one detector for finding the uncovered errors, and one extension of the S codes by using the greedy algorithm;

4. One program for T codes, which includes the encoding and decoding algorithm of VT codes (since by Definition 2.3.1, the decoding algorithm of T codes is based on the decoding algorithm of VT codes), the encoding and decoding algorithm of T codes, one function to generate the size of T codes with all possible values of β and γ, one detector for the uncovered errors, and one generator for generating all permutations;

5. A program for extendable errors, which searches all the uncovered errors (from the program of T codes) and checks all the possible extension positions in order to discover the extendable errors.

The time complexity of the backtracking program is $O(2^{q^l})$. It did not finish in more than 1 month of computing time when $l = 5$ and $q = 3$. So, we cannot construct a perfect or optimal ternary code of length 5.

Chapter 7

Conclusion and Future Work

In this thesis, we have studied three different approaches of constructing good deletion correcting codes. The first one is a construction from design theory. However, designs only produce $n-2$ deletion correcting codes over a large alphabet, which is very restrictive and not generally useful in the real world. Thus we chose not to pursue this approach. The second approach is the S codes. We constructed the codes and proved that they are 1-deletion correcting codes. But the size of the codes are extremely small.

In this case, we turned to the third approach: extending the T codes. The greedy algorithm and the backtracking algorithm are two trivial ways to extend the size. The greedy extension is usually as large as the backtracking extension, but uses much less time. Besides these two algorithms, we manage to extend the code by some properties and hypotheses, which are discovered during our research.We show that we can never expect the T codes to be

perfect or optimal. And we guess the T_s codes are the largest among the T codes. This strategy extends only one codeword 2222202211 right now, but this word can be added into any T_s code if length is longer than 10. And we believe, that more words can be extended as the length grows longer. Only a small part of the hypotheses can be proved at the moment. The rest of them have to be remained as future work.

For future work, the first thing that can be done is separating the backtracking program into multiple smaller tasks. The whole search space can be separated into subspaces and each of the subspaces can be searched simultaneously. The bottle neck of this process is that how to partition the tasks evenly to different programs. Before search, we cannot predict which branch of a tree has the shortest depth.

The second thing is proving the hypotheses, which is not easy. In the previous chapter, we have already pointed out that the hypotheses have a close relationship with Fermat's little theorem. So, studying the Number Theory will be a good start.

The third one is continuing the study of extendable errors. Searching more T_s codes will produce more extendable errors, and allow us to construct more extendable codewords.

The last one is studying the codes for other alphabets. All of my studies are focused on ternary, so some other alphabets are worth to try, especially when q is a power of 2. Some interesting properties or special cases may be discovered.

Bibliography

[1] R.J.R. Abel and F.E. Bennett, "Existence of directed BIBDs with block size 7 and related perfect 5-deletion-correcting codes of length 7", Designs, Codes and Cryptography, vol. 57, pp. 383-397, 2010.

[2] K.A.S. Abdel-Ghaffar and H.C. Ferreira, "Systematic Encoding of the Varshamov-Tenengol'ts Codes and the Constantin-Rao Codes", IEEE Transactions on Information Theory, vol.44, pp. 340-345, 1998.

[3] S. Baker, R. Flack and S. Houghten, "Optimal Variable-Length Insertion-Deletion Correcting Codes and Edit Metric Codes", Congressus Numerantium 186(2007), pp. 65-80, 2007.

[4] P.A.H. Bours, "On the Construction of Perfect Deletion-Correcting Codes using Design Theory", Designs, Codes and Cryptography, 1995 Kluwer Academic Publishers, vol. 6, pp. 5-20, 1995.

[5] V.I. Levenshtein, "Binary codes capable of correcting spurious insertions and deletions of ones", Problemy Peredachi Informatsii, vol. 1(No. 1), pp. 12-25, 1965(in Russian).

[6] V.I. Levenshtein, "On perfect codes in deletion and insertion metric", Discrete Mathematics and Applications, vol2, pp. 241-258, 1992.

[7] H. Mercier, V.K. Bhargava and V. Tarokh, "A Survey of Error-Correcting Codes for Channels With Symbol Synchronization Errors", IEEE Communications surveys & Tutorials, vol. 12, pp. 87-96, 2010.

[8] M. Mitzenmacher, "A survey of results for deletion channels and related synchronization channels", Probability Surveys, vol. 6, pp. 1-33, 2009.

[9] N.J.A. Sloane, "On Single-Deletion-Correcting Codes", Codes and Designs, Ohio State University, pp. 273-296, 2002.

[10] G. Tenengolts, Nonbinary Codes, "Correcting Single Deletion or Insertion", IEEE Transactions on Information Theory, vol. IT-30, pp. 766-769, 1984.

[11] R.R. Varshamov and G.M. Tenegolz, "One asymmetrical error correcting codes", Automatics and Telemechanics, vol. 26, pp. 288-292, 1965(in Russian).

[12] J. Wang, "Some combinatorial constructions for optimal perfect deletion-correcting codes", Designs, Codes and Cryptography, vol. 48, pp. 331-347, 2008.

[13] J. Wang and L. Ji, "Existence of T*(3,4,v)-Codes", The Journal of Combinatorial Designs, vol. 13, pp. 42-53, 2005.

[14] J. Wang and J. Yin, "Constructions for Perfect 5-Deletion-Correcting Codes of Length 7", IEEE Transactions on Information Theory, vol.52, pp. 3676-3685, 2006.

[15] J. Yin, "A Combinatorial Construction for Perfect Deletion-Correcting Codes", Designs, Codes and Cryptography, vol 23, pp. 99-110, 2001.