2-Deletion Codes: Beyond Binary

Zhen Zhou

CMU-CS-20-103
May 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Venkatesan Guruswami (Chair)
Bernhard Haeupler

*Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in Computer Science.*

Copyright © 2020 Zhen Zhou

Keywords: Coding Theory, Error-Correcting Codes, List Decoding, Non-binary, Indicator Sequence

Abstract

The problem of constructing codes resilient to *deletions*, where bits go missing and a subsequence is received, has a long history. Optimal binary single-deletion codes, which have size $\Theta(2^n/n)$ (or redundancy $\log n + O(1)$), for length n, have been known since the 60s. Regardless of the alphabet size of the code, the redundancy of a code is measured in bits. Although the optimal binary single-deletion codes have been established, the situation even for two deletions is a lot more complex. A non-constructive greedy argument shows the existence of binary codes of redundancy $4\log_2 n$, whereas redundancy $2\log_2 n$ is necessary. Compared with binary deletion codes, the topic of deletion codes over an alphabet of an arbitrary size q (or q-ary deletion codes) received much less attention. In this work, we describe 2-deletion codes over an alphabet of size q of redundancy $3\log_2 n + O_q(\log\log n) + r_2(n)$, assuming $r_2(n)$ is the minimum redundancy needed for binary 2-deletion code, thus opening up a new direction of study. Combining with Håstad's list-decodable binary 2-deletion codes, which have redundancy $3\log_2 n + O(\log\log n)$ and allow the received word to be list-decoded into at most two possibilities from 2 deletions, we obtain a list-decodable q-ary 2-deletion code with $6\log_2 n + O_q(\log\log n)$ redundancy and a list size 2. Our approach in the proof uses indicator strings and the run numbers in the string to compute various check sums that together enable recovery of the two missing bits. We hope this new perspective will be a useful way to think about and construct further deletion codes.

Acknowledgments

I would like to thank my professor Venkatesan Guruswami for giving me countless valuable advice on this project as well as technical writing. I have gotten support from professor Guruswami even in the most difficult times of my research. I have also learned many valuable lessons from his methods of approaching research questions.

I would like to thank professor Johan Håstad for sharing and discussing some of his unpublished results on the related topics, which inspired me to write a detailed proof for his list-decodable binary 2-deletion code, which is crucial to the second half of my thesis. I would also like to thank professor Bernhard Haeupler for being my committee member.

I would like to thank all the professors and staff I met in School of Computer Science for being supportive and understanding. Finally, I would like to thank my family and friends for their continuous support.

Contents

1 Introduction **1**
1.1 Main Contributions . . . 2
1.2 Organization of Article . . . 3
1.3 Related Works . . . 3
1.3.1 Existential Bounds of Code Size . . . 3
1.3.2 k-Deletion Codes . . . 4
1.3.3 List-Decodable Codes . . . 6

2 Preliminaries **7**
2.1 Coding Via Sketches . . . 7
2.2 Review of VT-Code and Its q-ary Generalizations . . . 8
2.3 Sketch Construction . . . 9
2.4 Overview of Proof Strategy . . . 10

3 Reduction From q-ary 2-Deletion to Binary 2-Deletion **13**
3.1 Naive q-ary Single-Burst 2-Deletion Code . . . 13
3.2 Decoding q-ary 2-Deletion Code with Known α-Indicator . . . 13
3.3 Proof of Unique-Decoding Lemma . . . 14
3.3.1 Generalization and Limitation . . . 17

4 Completing Sketch by Håstad's List-Decodable Code **19**
4.1 Håstad's Single-Deletion Code . . . 19
4.2 List-Decoding α-Indicator . . . 21
4.2.1 Proof of Håstad's List-Decoding Theorem . . . 22
4.3 Extensions of Current Result . . . 25
4.4 Encoding Algorithm . . . 26
4.5 Decoding Algorithm . . . 26

5 Conclusion and Future Directions **29**
5.1 Conclusion . . . 29
5.2 Future Work . . . 30

Bibliography **31**

List of Figures

3.1 2-deletion projection from q-ary alphabet to binary alphabet 14
3.2 Graphic representation for the case $\lambda_1 = \lambda_2$ 16

4.1 When $r_{n-1}(x') = r_n(x)$, graphic representation of $R_1(y_i)$ as a function of i . . . 20
4.2 When $r_n(x) - r_{n-1}(x') = 2$, graphic representation of $R_1(y_i)$ as a function of i . 21
4.3 When $r_n(x) - r_{n-2}(x') = 4$, graphic representation of $R_2(y_{s,t})$ as a function of s 24
4.4 When $r_n(x) - r_{n-2}(x') = 2$, graphic representation of $R_2(y_{s,t})$ as a function of s 25

Chapter 1

Introduction

An error-correcting code is an encoding scheme used in communication over a noisy channel, which allows the recovery of an altered message. Here, *noisy* means that the channel might be unstable and would potentially change or delete the content of the message. In general, the goal of error-correcting codes is to provide an efficient method of encoding and decoding the message, such that the intended message can be encoded efficiently with small redundancy, as well as decoded in a small amount of time. Meanwhile, it is crucial that the code is robust against the error that might occur, such that even in the worst-case scenarios, information can still be retrieved from the partially corrupted message. In general, different encoding schemes may need to deal with different types of *noise* in the communication channel, such as erasure (when a codeword bit is missing), error (when a codeword bit is flipped), deletion (when a codeword bit is deleted, and the rest are concatenated) and so on. In this work, we focus on the problem of recovering messages from deletions. Furthermore, we focus on the case where the number of deleted bits is a constant and study how to minimize the redundancy and maximize the efficiency in our code.

Let $\Sigma_q = \{0, 1, \cdots q-1\}$ be an alphabet of size q. A *k-deletion code* over Σ_q maps an n-symbol message $x \in \Sigma_q^n$ into a codeword $c \in \Sigma_q^N$, such that x can be recovered from any subsequence of c of length $N-k$. A code C is a list of codewords c, and $|C|$ is the size of the code. The *rate* of the code is the ratio $R = \frac{n}{N}$ that measures the efficiency of the code. Such a code allows for recovery from k worst-case deletions (where the locations of the deleted bits are *not* known). Similarly, a *list-decodable k-deletion code* of message length n over Σ_q allows a list L_x to be recovered from any subsequences of c of length $N-k$, such that $x \in L_x$. The quantity $\max_x |L_x|$ is the *list size* of the list-decodable code. In general, a k-deletion code over Σ_q is also referred to as *q-ary* k-deletion code, and when $q=2$, it is referred to as a *binary* k-deletion code. We will use these terms interchangeably.

To measure efficiency of the k-deletion codes in another metric, we define the *redundancy* of a code as follows: We assume that all codes we construct are *systematic codes*, where the message sequence x is a substring of the corresponding codeword c. Without the loss of generality, we assume that any codeword $c \in \Sigma_q^N$ can be written as the concatenation of two sequences, $c = x_1 \cdots x_n s(x)_1 \cdots s(x)_{N-n}$, where the sequence $x_1 \cdots x_n = x \in \Sigma_q^n$ is the message, and the binary sequence $s(x)_1 \cdots s(x)_{N-n} = s(x) \in \Sigma_2^{N-n}$ is the additional $(N-n)$-bits of information

needed. We define the binary sequence $s(x)$ as the *sketch* of the message x, and define the quantity $r = N - n$ as the *redundancy* of the code. Note that in this thesis, the redundancy of a code is always measured in bits, rather than q-ary symbols.

Early in the 1960s, Levenshtein [1] provided the very first result on binary k-deletion codes. Let us denote $C^*(n, k)$ as the largest binary k-deletion code for strings of length n. Levenshtein showed the following bounds on the size of largest binary k-deletion code:

$$\frac{2^k (k!)^2 2^n}{n^{2k}} \lessapprox |C^*(n,k)| \lessapprox \frac{k! 2^n}{n^k}$$

However, the construction of binary k-deletion code from Levenshtein's result [1] was not efficient because it was based on a greedy algorithm and required exponential decoding time with respect to k. Recent works by [2], [3], [4] and [5] showed much more efficient constructions with reasonable redundancy. In comparison, the q-ary extension of k-deletion code has been rarely explored. The q-ary extension of single-deletion codes was studied early in the work of [6], but the general case where k is a constant is far from being studied. Only a recent work of [7] has come up with efficient construction for this extension.

1.1 Main Contributions

In this thesis, we focus on the case $k = 2$ and the encoding over q-ary alphabet. We present the following results:

- We present a reduction from q-ary 2-deletion codes to binary 2-deletion codes, which allows a construction of q-ary 2-deletion code with redundancy of $3 \log_2 n + O_q(\log \log n) + r_2(n)$ bits, assuming $r_2(n)$ is the minimum redundancy needed for the optimal binary 2-deletion code.

- We present a detailed proof of Håstad's list-decodable binary 2-deletion code [8] with list size 2, that uses $3 \log_2 n + O(\log \log n)$ redundancy bits. This is based on Johan Håstad's result in 2015 [9] and 2017 [8].

- Combining the first two results, we present a list-decodable q-ary 2-deletion code with list size of 2, that uses $6 \log_2 n + O_q(\log \log n)$ redundancy bits. It is the best-known result on 2-deletion over q-ary alphabet.

In fact, Håstad's result [9] contains a $4 \log_2 n + O(\log \log n)$ redundancy construction of a binary 2-deletion code, which if combined with our reduction result, would provide a q-ary 2-deletion code with $7 \log_2 n + O_q(\log \log n)$ redundancy. However, since Håstad's list-decoding result from [8] has a much cleaner proof, we are including it in this thesis.

Formally, we summarize our contribution as the following theorems:

Theorem 1. *Let q be a constant with respect to n, and assume that the optimal binary 2-deletion code uses $r_2(n)$ redundancy bits and requires $O(t(n))$ time to encode and decode. Then there exists a 2-deletion code over an alphabet of size q, that uses at most $3\log_2 n + O_q(\log\log n) + r_2(n)$ bits of redundancy, and allows encoding and decoding in $O(n^2 + t(n))$ time.*

Theorem 2. *(Håstad's List-Decoding Theorem) There exists a list-decodable binary 2-deletion code with list size 2, that uses at most $3\log_2 n + O(\log\log n)$ bits of redundancy, and allows encoding and decoding in $O(n^2)$ time.*

As a corollary of **Theorem 1** and **Theorem 2**, we obtain the following theorem:

Theorem 3. *Let q be a constant with respect to n. Then there exists a list-decodable 2-deletion code over an alphabet of size q with list size 2, that uses at most $6\log_2 n + O_q(\log\log n)$ bits of redundancy, and allows encoding and decoding in $O(n^2)$ time.*

1.2 Organization of Article

In Chapter 1.3, we provide an overview of the literatures on the topics that are related to this thesis. In Chapter 2, we review the construction of the VT-code [10] and its q-ary extension [6], describe the sketch we use in our construction, and introduce all the necessary notations we need for the proof. In Chapter 3 and Chapter 4 we prove our main results by first proving our reduction from q-ary 2-deletion code to binary 2-deletion code, then use Håstad's list-decodable binary 2-deletion code [8] to complete the proof. In Chapter 5, we summarize our results and discuss some salient open problems.

1.3 Related Works

1.3.1 Existential Bounds of Code Size

As we have mentioned earlier, Levenshtein's result [1] showed the limits of the optimal binary k-deletion codes. It implied that $(2k + o(1))\log_2 n$ bits of redundancy is sufficient for a binary k-deletion code, and $(k + o(1))\log_2 n$ bits of redundancy is necessary for the code. Therefore, the bounds in [1] are often referred to as the *existential bounds* of binary k-deletion codes, because the bounds only implied the existence of such codes. Later works such as [11], [12] and [13] improved the upper bounds of Levenshtein's bounds. However, since the main focus of this thesis is to find an efficient construction of 2-deletion codes, we will not go into details regarding the existential bounds.

1.3.2 k-Deletion Codes

single-deletion codes. When the number of deleted bits $k = 1$, the k-deletion code is often referred to as *single-deletion code*. In 1960s, Varshamov and Tenengolts [10] constructed a concise binary single-deletion code, usually referred to as VT-code, that achieved a code size of $\frac{2^n}{n+1}$. According to Levenshtein's upper bound($\Theta(\frac{2^n}{n})$) [1], VT-code has a code size that is asymptotically optimal. It was later shown by Sloan's survey [14] that VT-code is truly optimal for message length up to 10. We will review the details of VT-code in later chapters since it is crucial in our constructions.

k**-deletion codes**. While constructive codes for single-deletion has been established since 1960s, efficient k-deletion codes for fixed k have not been discovered until recent years. A naive solution for constructing a k-deletion code would be using $(k+1)$-repetition code, which simply repeats the message $k+1$ times and requires $O_k(n)$ redundancy bits. Inspired by VT-codes, construction by Helberg and Ferreira [15] used Fibonacci sequence v_i to replace weights i in the VT-code for their binary k-deletion codes (usually known as the Helberg codes), and work by [16] studied the decodablility of run-length limited sequences. However, the code rate of both results are far from optimal. In the work of [15], the weights grow exponentially, therefore the construction requires $\Omega(n)$ bits of redundancy. Similarly, the result of [16] has code rate bounded away from 1. An improvement was made by Guruswami and Wang [17] in 2014, whose work implicitly gave a $O(\sqrt{kn})$ redundancy construction of binary k-deletion code, but the redundancy was still bounded away from the existential bounds.

In 2015, a breakthrough was made by Brakensiek, Guruswami and Zbarsky [2] to finally bring down the redundancy to the level of $O_k(\log n)$ for binary k-deletion codes. They showed a construction for a binary k-deletion code that only required $O(k^2 \log k \log n)$ redundancy bits and allowed efficient decoding in $O_k(n(\log n)^4)$ time. In 2018, Cheng, Jin, Li and Wu [3] and Haeupler [4] simultaneously and independently presented an amazing algorithm that can be adapted to a binary k-deletion code with $O(k \log n)$ redundancy and $\texttt{poly}(n)$ decoding time. Their results were already asymptotically optimal with respect to the existential bound [1]. To take a step further, recent result by Sima and Bruck in 2019 [18] and result by Sima, Gabrys and Bruck in 2020 [5] showed two k-deletion codes with $8k \log n$ and $4k \log n$ redundancy respectively. These results further reduce the constant factor in redundancy, but their drawbacks are clear as well: construction of [18] is *non-systematic*, and both results require exponential decoding time with respect to k. In the upcoming sections, we will make the distinction between *systematic* and *non-systematic* codes.

After result of [2] was posted, many results were discovered on k *edit-errors*. Naturally, *insertion* errors is defined by addition of symbols into the message sequence, and *edit-errors* include both *insertion* and *deletion* errors. On this topic, construction by Belazzougui [19] showed a $O(k^2 + k \log^2 n)$ redundancy one-pass protocol of document exchange with k edit-errors. Results of [3] and [4] both provided $O(k^2 \log \frac{n}{k})$ redundancy solution to solve the document exchange problem with binary alphabet over a noisy channel that causes up to k edit-errors for any k. An important technique introduced by Haeupler and Shahrasbi in [20] called *synchronization*

string transformed the edit-error type problems to erasure/corruption error problems. Further works of [21] and [22] improved upon this reduction technique. For the scope of this thesis, we will mainly focus on 2-deletion code, although some extension to 2 edit-errors will be discussed at the end of the proof section.

2-deletion codes. Although the general techniques for binary k-deletion code have pushed the number of redundancy bits to the existential bounds, those techniques usually perform poorly on cases such as $k = 2$. When applying existing k-deletion code constructions such as [2] on binary 2-deletion problem, the resulting redundancy can be as large as $128 \log n$ [23]. Observe that each constant factor introduced in the number of redundancy bits shrink the code size by *poly(n)* factor. Therefore, alternative solutions other than general techniques on k-deletion are needed for 2-deletion codes.

To solve this problem, work by Gabrys and Sala [24] and work by Sima, Raviv and Bruck [23] showed two different constructions of binary 2-deletion codes with $8 \log n$ and $7 \log n$ redundancy respectively. Both constructions were efficient, but not optimal when compared with the existential bound of $4 \log n$. So far, the best result on binary 2-deletion code came from Hastad's unpublished results in 2015 [9] and 2017 [8], which showed a $4 \log n$ redundancy construction. Hastad's proof in [8] included a nice construction of a list-decoding result, which will be presented in detail in the second half of the thesis.

q-ary k-deletion codes. As part our focus, the q-ary extension of deletion codes has been rarely studied. After the discovery of VT-codes [10], Tenengolts proposed a q-ary single-deletion code in [6]. He reduced the problem of q-ary single-deletion code to binary single-deletion code, which inspired us greatly in this thesis. Later work by [15] was originally proposed for binary k-deletion codes, but was later proven in the work of Le and Nguyen [25] that the same technique is also applicable to q-ary k-deletion codes. However, the code in [25] has an unsatisfactory rate. Result of [4] on k edit-errors can be adapted to q-ary as well, but it suffers from an extra $\log n$ factor from adaptation, resulting in a $O(k \log^2 n)$ redundancy. Only most recently, work by Sima, Gabrys and Bruck [7] showed a $4k \log n + o_q(\log n)$ redundancy construction for q-ary k-deletion code, which is the only known result that asymptotically matches the existential upper-bound among all q-ary k-deletion codes.

Related to the topic of q-ary deletion codes, there also has been some results on q-ary *burst deletion* codes. A *burst deletion* code can recover message x from substrings of c when adjacent bits are deleted. In particular, a single-burst k-deletion code allows the recovery from deleting k adjacent symbols from c. Work by Schoeny, Wachter-Zeh, Gabrys and Yaakobi [26] constructed single-burst binary k-deletion codes for $k = 3, 4$, and further work by [27] studied the extension of the problem on q-ary alphabet. However, since burst deletion is not the main topic of the thesis, we will only be mentioning it again when we use burst deletion results in the construction of our sketches and in the proof.

1.3.3 List-Decodable Codes

Although one of our main focus of the thesis, Håstad's list-decodable code [8] focused on list-decodable results for small number of deletions, literature in the past on list-decodable codes mainly focused on the case where a large fraction of bits is deleted. For example, work by Guruswami and Wang [17] was the first to study the list-decodability of binary deletion codes on high rate deletions. Later work by Haeupler, Shahrasbi and Sudan [28] further investigated the potential of list-decoding using a larger alphabet, and work by Liu, Tjuawinata and Xing [29] showed other bounds for large rate deletion codes. However, the case where k is a fixed integer has received much less attention in list-decodable k-deletion codes. When k is treated as a fixed constant, work by Wachter-Zeh [30] constructed a binary list-decodable k-deletion code of list size n^{k-1} based on VT-code and parity checks. As we will show in the thesis, Håstad's list-decodable 2-deletion code [8] is an improvement to this result for the case $k = 2$.

Chapter 2

Preliminaries

2.1 Coding Via Sketches

Recall that when we work under *systematic code* setting, any codeword c can be written as a concatenation of the message x and the sketch $s(x)$. We claim that there exists a reduction between an efficient sketch and an efficient 2-deletion code. As a result, we should focus on finding the sketch of the minimum size for the 2-deletion problem. We will formalize our claim as the following lemma:

Lemma 1. *For every message $x \in \Sigma_q^n$, assume that there exists an encoding algorithm* `Enc` *that produce the q-ary 2-deletion sketch $s(x)$, such that $|s(x)| \leq t \log n$, where t is a constant, and there exists an decoding algorithm* $\texttt{Dec}(x', s(x))$ *that takes in any subsequence of x of length $n-2$ and the sketch $s(x)$, and outputs x. Then, there exists a q-ary 2-deletion code with redundancy $t \log n + O(\log \log n)$.*

Proof. Observe that when $s(x)$ is an effective q-ary 2-deletion sketch for x, we can apply the encoding algorithm `Enc` on $s(x)$ again and obtain $s(s(x))$. Then the size of $s(s(x))$ has a lower asymptotic order: $|s(s(x))| < t \log t \log n = O(\log \log n)$. Therefore, we are free to use 3-repetition code on $s(s(x))$ to protect it from 2-deletion. The total number of redundancy bits will be $t \log n + 3t \log t \log n = t \log n + O(\log \log n)$.

For the decoding algorithm, we can decode the strings in the following order: $s(s(x)) \rightarrow s(x) \rightarrow x$. The sketch $s(s(x))$ can be decoded by the decoding algorithm of 3-repetition code, and $s(x)$ and x can be decoded subsequently by using the 2-deletion decoding algorithm `Dec`.

□

By proving the above theorem, we shift our focus to constructing an efficient 2-deletion sketch, rather than constructing the entire 2-deletion code. We can also assume that any problem in decoding 2-deletion comes down to decoding the message with a subsequence of length $n-2$ and the sketch. We will refer to such a sketch as an effective 2-deletion sketch.

2.2 Review of VT-Code and Its q-ary Generalizations

For any message x of length n, we can write it as a sequence of n characters $x_1, x_2, \cdots, x_n$. We recall that in VT-code [10], a binary single-deletion code is constructed as the set of all binary sequences $x_1, x_2, \cdots, x_n$ that satisfy the following equation for some fixed integer a:

$$\sum_{i=1}^{n} i x_i \equiv a \mod n+1 \tag{2.1}$$

Therefore, it is equivalent to think of VT-code as a sketch defined by $\sum_{i=1}^{n} i x_i \mod n+1$ on any message x, which requires $\log_2 n + 1$ number of redundancy bits.

Example 1. For a binary sequence $x = 110010$, the associated sketch is $\sum_{i=1}^{n} i x_i \mod n+1 = 1+2+5 \mod 7 = 1$.

To generalize VT-code to an alphabet of size q, Tenengolts proposed a construction of q-ary single-deletion code [6], which we reiterate as follows: For any $q - ary$ sequence $x_1, x_2, \cdots x_n$, where $x_i \in \Sigma_q$, associate it with a sequence $\alpha_1, \alpha_2, \cdots, \alpha_n$, where $\alpha_i \in \Sigma_2$. Each number α_i is defined by the following rule:

$$\alpha_i = \begin{cases} 1 & \text{if } i = 1 \text{ or } x_i \geq x_{i-1} \\ 0 & \text{if } x_i < x_{i-1} \end{cases} \tag{2.2}$$

Example 2. For a q-ary sequence $x = 123321$, the associated α_i's are 1,1,1,1,0,0.

Tenengolts' paper [6] showed that a q-ary single-deletion code can be constructed as a set of all binary sequences of length n that satisfy $\sum_{i=1}^{n} x_i \mod q = a$ and $\sum_{i=1}^{n}(i-1)\alpha_i \mod n = b$ for some fixed integers a, b. Tenengolts' code is slightly different from VT-code because $\alpha_1 = 1$ is fixed in the definition. Using Tenengolts' code is equivalent to having the following sketch $T(x)$:

$$\begin{aligned} T(x) &= (a, b), \text{where} \\ a &= \sum_{i=1}^{n} x_i \mod q \\ b &= \sum_{i=1}^{n} (i-1)\alpha_i \mod n \end{aligned} \tag{2.3}$$

Example 3. For a ternary sequence $x = 123321$, the associated sketch is $T(x) = (0, 0)$.

In this thesis, for any q-ary sequence $x = x_1 x_2 \cdots x_n$, denote the string $\alpha(x)$ as the concatenation of the α_i as defined previously. We will refer to the sequence $\alpha(x)$ as the *α-indicator sequence* of message x. To distinguish between characters in α-indicator sequence of different

messages, we use $\alpha_i(x)$ to refer to the ith character of $\alpha(x)$.

2.3 Sketch Construction

Sketch For Håstad's List-Decodable Code: Many properties of the *run numbers* of a sequence are essential in Håstad's proof [8] of his list-decodable binary 2-deletion code. We define *run numbers* of a sequence as follows: For any sequence $x_1, x_2, \cdots, x_n \in \Sigma_q$, associate it with a sequence $r_1, r_2 \cdots, r_n$, where $r_i \in \mathbb{N}^+$. Each number r_i is defined recursively by the following rule:

$$r_i = \begin{cases} 1 & \text{if } i = 1 \\ r_{i-1} & \text{if } i \neq 1 \text{ \& } x_i = x_{i-1} \\ r_{i-1} + 1 & \text{if } i \neq 1 \text{ \& } x_i \neq x_{i-1} \end{cases} \tag{2.4}$$

Example 4. For a sequence $x = 123321$, the associated r_i's are 1,2,3,3,4,5.

For any string $x = x_1 x_2 \cdots x_n$, we define $r_i(x)$ as the *run number* of string x at index i. The consecutive characters are in the same *run* when their run numbers are the same. Using the definition of run numbers, the sketches $R_1(x), R_2(x)$ used in Håstad's list-decodable binary 2-deletion code are defined as follows:

$$\begin{aligned} R_1(x) &= \sum_{i=1}^{n} r_i(x) \mod 2n + 1 \\ R_2(x) &= \sum_{i=1}^{n} (r_i(x))^2 \mod 2n^2 + 1 \end{aligned} \tag{2.5}$$

Example 5. For a sequence $x = 123321$, the associated sketches are $R_1(x) = 5, R_2(x) = 64$.

Observation 4. As a side note, we observe that by using our notion of run numbers, the VT-code [10] and its q-ary extension [6] do not in fact decode the exact index of deleted bit. Instead, they decode the exact run number of the deleted bit. An example would be deleting a 0 from the string 0011. It would result in the same string 011 regardless of the choice of deleting first or second 0. Therefore, the decoder should not worry about decoding exact index in many cases, but rather focus on decoding the run number of deleted bits.

Sketch For Reduction From q-ary to Binary: For the reduction, we first define a run-number-weighted sketch on α-indicator of x, which we denote as $rw_\alpha(x)$, as follows:

$$rw_\alpha(x) = \sum_{i=1}^{n} r_i(\alpha(x)) \cdot x_i \mod qn \tag{2.6}$$

We will also use a sketch for binary single-burst 2-deletion code. For sequence $x = x_1 x_2 \cdots x_n$, let x_{odd} be the sequence $x_1 x_3 \cdots$ that consists of odd-indexed characters in x, and x_{even} be the

sequence $x_2x_4\cdots$ that consists of even-indexed characters in x. Our sketch for single-burst 2-deletion code will be applying Tenengolts' q-ary single-deletion sketch on x_{odd} and x_{even} respectively, which we denote as $T(x_{odd}), T(x_{even})$. We will formally prove that it is a single-burst 2-deletion sketch in the upcoming chapter.

To simplify, we will use $Red(x)$ to denote the concatenation of all the sketches we use in the reduction step:

$$Red(x) = (T(x_{odd}), T(x_{even}), rw_\alpha(x)) \tag{2.7}$$

Other Constant-Sized Sketches: For simplicity, we assume the following constant-sized sketches are known in advance, but we omit them in our analysis:

1. For each character $i \in \Sigma_q$, we keep a counter $c_i \mod 3$ in our sketch.
2. For the sequence x, we keep $r_n(x) \mod 5$ in our sketch.

The first sketch helps the decoder recover the identity of deleted characters, and the second sketch helps the decoder compute the change in number of runs of the message and its subsequence.

Sketch Notations and Redundancy Recap:

- Sketch for reduction from q-ary to binary: $Red(x) = (T(x_{odd}), T(x_{even}), rw_\alpha(x))$
  Redundancy: $2 * (\log_2 \frac{n}{2} + \log_2 q) + \log_2 qn = 3\log_2 n + O_q(1)$
- Sketch for Håstad's list-decodable code: $R_1(x), R_2(x)$
  Redundancy: $\log_2{(2n+1)} + \log_2{(2n^2+1)} = 3\log_2 n + O(1)$
- Sketch known in advance (constant-sized): $c_i \mod 3$, $r_n(x) \mod 5$
  Redundancy: $q * \log_2 3 + \log_2 5 = O_q(1)$

Additional Notation:

In many works related to k-deletion codes, $\sigma_k(c)$ denotes the deletion ball of c, consisting of all the subsequences of c of length $N-k$, we will stick with this notation as well.

2.4 Overview of Proof Strategy

The crux of our approach is to take the viewpoint of a decoder in the protocol. Whenever two different messages can be decoded, we use the information from sketch to eliminate at least one. Our main proof shows a reduction from the decoding of q-ary 2-deletion problem to the decoding of binary 2-deletion problem, as well as showing a proof on a list-decodable binary 2-deletion code by Håstad [8]. For the reduction, we will first show that the α-indicator of the message can be recovered by using an existing binary 2-deletion code, by using a reduction from [6]. We then prove **Theorem 1** by showing that α-indicator and the sketch $Red(x)$ are sufficient in fixing the

positions of deleted bits by a case analysis on the run where character was deleted. In our proof of **Theorem 2**, we present a detailed proof for Håstad's list-decodable binary 2-deletion code by analyzing some monotonicity results of run numbers of the message.

After proving the first two theorems, we will combine the results to present an efficient encoding and decoding algorithm for a q-ary 2-deletion code as our **Theorem 3**. The encoding algorithm of this code is based on the sketches we presented earlier in this chapter. The decoding algorithm breaks down the decoding procedure into a two-stage problem, which involves solving a binary 2-deletion problem on α-indicator first, then solves the q-ary 2-deletion problem using our reduction.

Chapter 3

Reduction From q-ary 2-Deletion to Binary 2-Deletion

3.1 Naive q-ary Single-Burst 2-Deletion Code

In the previous chapter, we claimed that Tenengolts' sketch on x_{odd}, x_{even} are effective single-burst 2-deletion sketches. We will prove the claim here since it will be helpful in our proof of the main theorem.

Lemma 2. *$T(x_{odd}), T(x_{even})$ are effective q-ary single-burst 2-deletion sketches for any $x \in \Sigma_q^n$.*

Proof. Any q-ary single-burst 2-deletion problem on x can be reduced to a q-ary single-deletion problem on x_{odd} and a q-ary single-deletion problem on x_{even}, because deleting two adjacent bits in x would cause exactly one bit to be deleted from x_{odd} and x_{even} each. Since $T(x)$ is an effective q-ary single-deletion sketch for x as shown in [6], $T(x_{odd}), T(x_{even})$ can help recover x_{odd} and x_{even} respectively, and therefore help recover x. □

3.2 Decoding q-ary 2-Deletion Code with Known α-Indicator

In order to demonstrate the effectiveness of our technique, we will show that it is compatible with all existing binary 2-deletion codes. To do so, we will show a reduction from q-ary 2-deletion codes to binary 2-deletion codes. Assume that there exist some binary 2-deletion codes, we claim that we can uniquely recover the α-indicator sequence of x.

Claim 1. *Let s be a positive integer. Let x be a q-ary sequence of length $n > s$. For every $x' \in \sigma_s(x)$, $\alpha(x') \in \sigma_s(\alpha(x))$.*

Proof. Instead of proving the claim directly, we will show a reduction from this claim to a result that was proven in [6]: For all sequences $x' \in \sigma_1(x)$, $\alpha(x') \in \sigma_1(\alpha(x))$. Observe that when $x' \in \sigma_s(x)$, we can construct a sequence of q-ary sequences $y_0, y_1, \cdots, y_s$, such that $y_0 = x'$ and $y_s = x$, and for all $1 \leq i \leq s$, $y_{i-1} \in \sigma_1(y_i)$. Then by applying the claim from [6], we can show that for all $1 \leq i \leq s$, $\alpha(y_i) \in \sigma_1(y_{i-1})$. Therefore, we obtain that $\alpha(y_0) \in \sigma_s(\alpha(y_s))$, which is exactly $\alpha(x') \in \sigma_s(\alpha(x))$.

□

From the claim proven, we know that having a binary 2-deletion code allows us to recover $\alpha(x)$ uniquely. Then we claim that proving **Theorem 1** requires us to prove the following lemma, which we refer to as the Unique-Decoding Lemma. Recall that $Red(x) = (T(x_{odd}), T(x_{even}), rw_\alpha(x))$.

Lemma 3. (Unique-Decoding Lemma) *For sufficiently large n, if $x \in \Sigma_q^n$ is a q-ary sequence of symbols, and $x' \in \sigma_2(x)$ is a subsequence of x of length $n-2$, then there exists an algorithm* `DecRed`$(x', Red(x), \alpha(x))$ *that computes x in $O(n^2)$ time.*

3.3 Proof of Unique-Decoding Lemma

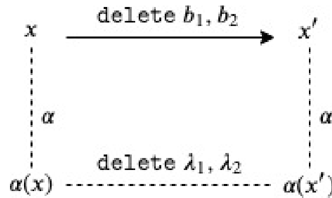

Figure 3.1: 2-deletion projection from q-ary alphabet to binary alphabet

Proof. When $\alpha(x)$ is recovered, the identities of the two bits deleted from $\alpha(x)$ to $\alpha(x')$ are also recovered by counting the difference in number of $0, 1$ between $\alpha(x)$ and $\alpha(x')$. Denote these two bits deleted from $\alpha(x)$ as $\lambda_1, \lambda_2 \in \{0, 1\}$, corresponding to the characters $b_1, b_2 \in \Sigma_q$ deleted from x. The values of λ_i and b_i are known in advance by using our constant-sized sketches. Without the loss of generality, assume that $\lambda_1 \leq \lambda_2$. In order to analyze how to recover x, we first analyze how $\alpha(x)$ is recovered by inserting λ_1, λ_2 into $\alpha(x')$ by a case analysis on λ_i:

Case 1: $\lambda_1 = \lambda_2$. When $\lambda_1 = \lambda_2$, we claim that we can recover the runs i, j in which two bits are inserted from $\alpha(x)$ by a simple linear-time procedure `DecodePosition`: Recall that $\alpha_i(x)$ is the ith character of $\alpha(x)$. For each $1 \leq i \leq n$, we greedily match the bits $\alpha_i(x)$ and $\alpha_i(x')$. If they are not equal, output the run number of the current character $\alpha_i(x')$, and delete the bit $\alpha_i(x)$ from $\alpha(x)$ to continue matching. If we finished matching, and there are still characters left in $\alpha(x)$, we can out put their run numbers.

By applying this algorithm, we recover at least one set of positions (i, j) to insert λ_1, λ_2 to recover $\alpha(x)$. We make the following claim:

Claim 2. *If $\lambda_1 = \lambda_2$, there exists a unique pair of runs (i, j) to insert λ_1, λ_2 to recover $\alpha(x)$ from $\alpha(x')$.*

Proof. Let (i, j) be the solution obtained from running `DecodePosition`$(\alpha(x), \alpha(x')$. Without the loss of generality, assume $\lambda_1 = \lambda_2 = 0$, and $i \leq j$. Assume for the sake of contradiction that

Algorithm 1 Algorithm for Finding the Runs Where Two bits Are Deleted

```
1: function DECODEPOSITION(α(x), α(x'))
2:     n ← |α(x)|, index ← 0, offset ← 0, Runs ← [r_{n−3}(x'), r_{n−2}(x')]
3:     while index ≠ n − 2 do
4:         if α_index(x') ≠ α_{index+offset}(x) then
5:             Runs[offset] ← [r_index(x')]
6:             offset ← offset + 1
7:         else
8:             index ← index + 1
9:         end if
10:    end while
11:    return Runs                ▷ Two entries indicates two runs to insert the bits
12: end function
```

there exists another set of runs $(i', j') \neq (i, j)$ to insert λ_1, λ_2 respectively to recover $\alpha(x)$ from $\alpha(x')$. Let s_1 denote the string obtained from inserting two bits at run (i, j), and let s_2 denote the string obtained from inserting two bits at run (i', j'). By our construction, $s_1 = \alpha(x)$. We want to show that $(i', j') \neq (i, j)$ implies $s_1 \neq s_2$.

Since $(i', j') \neq (i, j)$, either we have $i' \neq i$, or we have $i' = i$ and $j' \neq j$. In the first case, let $i^* = \min(i, i')$, and compare the number of zeroes in run i^* in s_1 and s_2 respectively. Notice that because $\lambda_1 = \lambda_2$, the character inserted in run j, j' would not decrease the number of zeroes in run i^*, and therefore, in s_1 and s_2 the number of zeroes in run i^* must differ by at least 1, so we obtain that $s_1 \neq s_2$. Similarly, in the second case, let $j^* = \min(j, j')$, we can show that in s_1 and s_2, the number of zeroes in run j^* must differ by 1, and therefore $s_1 \neq s_2$. Since the contradiction is shown in both cases, we have shown that s_2 must differ from s_1, which indicates that the set of runs to insert must be unique when $\lambda_1 = \lambda_2$.

□

After fixing the run i, j, we know one of the following cases should happen:

A. λ_1 is inserted in run i, λ_2 is inserted in run j. (Fig 4.2, Case A)
B. λ_1 is inserted in run j, λ_2 is inserted in run i. (Fig 4.2, Case B)

In the above two cases, the same $\alpha(x)$ is obtained, but the strings obtained from inserting $b_1, b_2 \in \Sigma_q$ to recover x still might be different. To show that we can obtain exactly one string from each case, we provide a simple observation to facilitate the proof:

Observation 5. When the run i to insert λ_1 is fixed, the index to insert b_1 is uniquely fixed. Similarly, when the run j to insert λ_2 is fixed, the index to insert b_1 is also uniquely fixed.

To prove this observation, we simply notice that each run in $\alpha(x)$ represents a monotone sequence in x. A sequence of 1's in $\alpha(x)$ represents a monotonically non-decreasing sequence in x, and a sequence of 0's in $\alpha(x)$ represents a monotonically decreasing sequence in x. Therefore, if we know the identity of b_1 and the monotone sequence to insert it, its position in the sequence is fixed. Therefore, we obtain exactly one string in each case A, B. Denote the strings obtained

from case A, B as s_A, s_B respectively. Given that at least one out of the two strings s_A, s_B is the message x, we want to show that either $s_A = s_B = x$, or one of the string does not produce the same sketch as x. We will do so by a case analysis by comparing b_1, b_2:

- Special case 1. When $b_1 = b_2$, we will recover the same string in the two cases, $s_A = s_B$. Therefore, we recover x uniquely (Fig 4.2, Special 1).

- Special case 2. If $i = j$, we will recover the same string in the two cases again, and therefore we recover x uniquely (Fig 4.2, Special 2).

- In the general case, when $b_1 \neq b_2$ and $i \neq j$, we may check the string obtained with our sketch $rw_\alpha(x) = \sum_{i=1}^{n} r_i(\alpha(x)) \cdot x_i \mod qn$. We can treat the sketch $rw_\alpha(x)$ as a potential function and calculate the value of the function on these two strings s_A, s_B. Notice that:

  $$|rw_\alpha(s_A) - rw_\alpha(s_B)| = |(j-i) \cdot (b_1 - b_2)| \mod qn \neq 0$$

  Therefore, the sketch of at most one string out of s_A, s_B can match the sketch $rw_\alpha(x)$. Since we know at least one string out of s_A, s_B is x, we recover x by choosing the string with the matching sketch value.

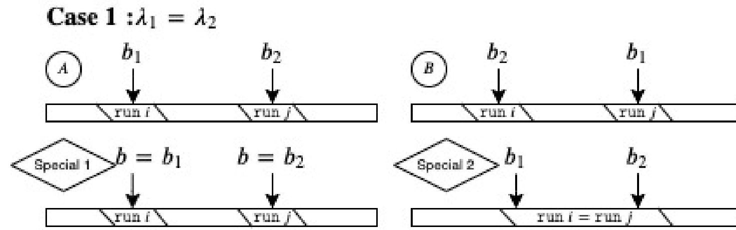

Figure 3.2: Graphic representation for the case $\lambda_1 = \lambda_2$

Case 2: $\lambda_1 \neq \lambda_2$**.** When $\lambda_1 = 0$ and $\lambda_2 = 1$, we can use the same argument as above in most cases to determine a unique solution. One caveat here is that now the uniqueness of the sequence $\alpha(x)$ does not guarantee the uniqueness of which runs to insert the two characters. Particularly, we will show how the previous **Claim 2** fails to hold in some special cases when $\lambda_1 \neq \lambda_2$ in the following example:

Example 6. To recover the string $x = 19281919$ from the subsequence $x' = 191919$, we first recover the indicator sequences $\alpha(x') = 110101$ and $\alpha(x) = 11010101$. By running our algorithm `DecodePosition` on $\alpha(x')$ and $\alpha(x)$ as input, we would obtain that we need to insert the numbers $2, 8$ at the last two positions. We then will obtain the string 19191928 instead of the desired result. Similarly, our algorithm would fail only if two adjacent bits are deleted. In general, if some $\alpha(x)$ contains $O(n)$ length of alternating 01 pattern, then up to $O(n)$ strings can share the same α-indicator sequence. In such cases, we obtain up to $O(n)$ different strings.

To resolve this issue, we notice that **Claim 2** only fails to hold when we are in the special case where two adjacent symbols are deleted, which is exactly the problem of single-burst 2-deletion. Since we have shown in **Lemma 2** that $T(x_{odd}), T(x_{even})$ are effective q-ary single-burst 2-deletion sketches, x can be uniquely decoded by using these sketches in the special cases.

After showing the correctness of the code above, a decoding algorithm `DecRed` works as follows: First, determine the identity of λ_1, λ_2 by $\alpha(x')$. Then, determine the positions to insert both bits to complete $\alpha(x)$. It takes $O(n^2)$ time to run brute-force algorithm over all valid insert positions to compute the sketch for each position, and check if it is equal to $Red(x)$.

□

3.3.1 Generalization and Limitation

We have presented in this chapter on how to construct a q-ary 2-deletion code assuming there exists an efficient binary 2-deletion code. Notice that the same technique also applies to all list-decoding results on binary 2-deletion code, without increasing the size of the resulting list. One would only need to apply the reduction on each of the α-indicators obtained by the binary 2-deletion code. Therefore, our result can be generalized to adapt to any list-decodable binary 2-deletion code.

Meanwhile, it is tempting to generalize our reduction technique, which uses α-indicator sequence, to other k-deletion questions with $k = 3, 4$ or larger. However, one main challenge in doing so has been discussed in our proof: Up to polynomially many different strings might share the same indicator sequence $\alpha(x)$. This sharing issue is a fundamental problem of using the α-indicator. In some cases such as $k = 2$, one can use an existing code such as the single-burst 2-deletion code to distinguish between the strings that share the same α-indicator string, but it is much harder to do so in the general setting when k is a larger constant. In the general setting, it may require complex case analysis, as well as more complex k-burst deletion codes, since the burst can be split up to multiple locations.

Chapter 4

Completing Sketch by Håstad's List-Decodable Code

In this chapter, we present a detailed proof of correctness of Håstad's construction [8] of list-decodable binary 2-deletion codes, with a list size 2. Combining this list-decoding result with our reduction, we obtain a list-decodable 2-deletion code on q-ary alphabet, with list size up to 2. We will first show that part of Håstad's sketch is effective against binary single-deletion, then show that the entire sketch is effective against binary 2-deletion. Since now we are working with binary sequences, we will now be working with binary alphabet $\{0, 1\}$ in this section instead of q-ary alphabet.

For simplicity, we may assume that $x_1 = 1$ and $x_n = 0$, as we may pay extra constant number of bits in our sketch to compensate for this. The following lemma will be useful in our proof, and we will be proving the lemma at the end of the chapter:

Lemma 4. *(**Run Number Lemma**) For any $x \in \Sigma_2^n$, if bit b is deleted from x to obtain x', then one of the following two cases applies:* **1.** *$r_n(x) - r_{n-1}(x') = 0, 2$.* **2.** *$r_n(x) - r_{n-1}(x') = 1$, and b is either at the beginning or at the end of x.*

4.1 Håstad's Single-Deletion Code

Before showing how effective the entire sketch is, we present a lemma that shows Håstad's sketch is equally efficient as the VT-code sketch [10], up to an additive constant factor:

Lemma 5. *For any $x \in \Sigma_2^n$, $R_1(x)$ and $r_n(x) \mod 5$ are effective binary single-deletion sketches.*

Proof. Recall that $R_1(x)$ is defined as $R_1(x) = \sum_{i=1}^{n} r_i(x) \mod 2n + 1$. Since we know the sketch $r_n(x) \mod 5$, we can use it to compute $r_n(x) - r_{n-1}(x')$ for any $x' \in \sigma_1(x)$. By using the **Run Number Lemma**, we know that we can prove this lemma by a case analysis on the value of $r_n(x) - r_{n-1}(x')$:

- $r_{n-1}(x') = r_n(x)$. Denote y_i as the string obtained from inserting b at index i. Then we can define a potential function $f(i) = R_1(y_i)$, by calculating the sketch value on each y_i. Let $f(i)$ be only defined on the values of i such that $r_{n-1}(y_i) = r_n(x)$, then we observe that $f(i)$ is a monotonically increasing function if the mod is not taken as in $R_1(x)$. Additionally, all the $f(i)$ values differ by at most r, where r is the total number of runs in y_i. The total number of runs can not exceed the number of bits, therefore $r \leq n$. So if we take $\mod 2n+1$ on $f(i)$, only one position has the matching sketch value as $R_1(x)$.

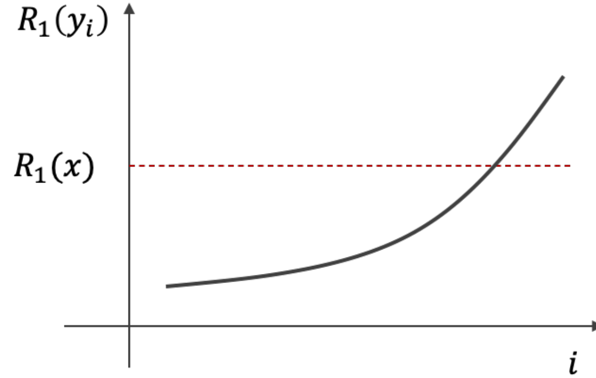

Figure 4.1: When $r_{n-1}(x') = r_n(x)$, graphic representation of $R_1(y_i)$ as a function of i

Example 7. Assume $x' = 10100$, and the sketch $R_1(x) = 5$. Then the list of all the possible y_i obtainable from x' and their corresponding sketches $R_1(y_i)$ (without the mod) are as follows:

$\mathbf{y_i}$	$\mathbf{R_1(y_i)}$
$\underline{1}10100$	5
$1\underline{0}0100$	6
$10\underline{1}100$	7
$101\underline{0}00$	8

- $r_n(x) - r_{n-2}(x') = 2$. Similar to previous case, define y_i and $f(i) = R_1(y_i)$. In this case, let $f(i)$ be only defined on the values of i such that $r_{n-1}(y_i) = r_n(x) - 2$, then observe that $f(i)$ is a monotonically decreasing function if the mod is not taken as in $R_1(x)$. Additionally, all the $f(i)$ values differ by at most $2r$, where r is the total number of runs in y_i, and again we have $r \leq n$. Therefore, with the $\mod 2n + 1$ taken, only one position has the matching sketch value as $R_1(x)$.

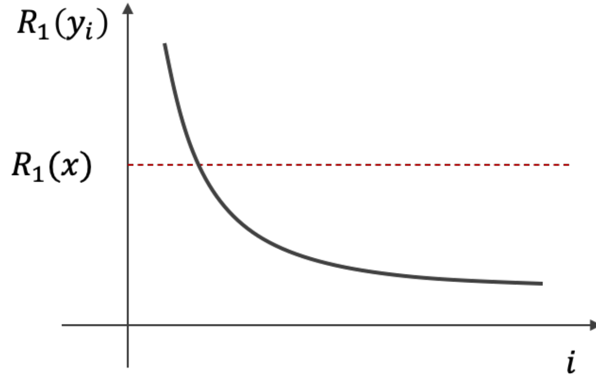

Figure 4.2: When $r_n(x) - r_{n-1}(x') = 2$, graphic representation of $R_1(y_i)$ as a function of i

- $r_n(x) - r_{n-2}(x') = 1$. By the **claim 3** we know that the bit is deleted either from the beginning or the end, then we can use the fact that $x_1 = 1$ and $x_n = 0$ to decode x uniquely.

□

The technique we used in this proof involves using the sketch as a potential function. We then showed some monotone property of the potential function, while proving that the function values can only vary within a certain range. This technique will appear again in the upcoming main proof of Håstad's list-decodable binary 2-deletion code.

4.2 List-Decoding α-Indicator

To prove **Theorem 2**, we will need to show that $R_1(x), R_2(x)$ are effective sketches for list-decoding binary 2-deletion codes. Recall that when we can list-decode a binary 2-deletion code, we can list-decode α-indicator for any q-ary sequence and construct a q-ary 2-deletion code. We claim that the proof of Håstad's code [8] can be done by a case analysis on the value $r_n(x) - r_{n-2}(x')$, which indicates the difference in total number of runs between the message x and the subsequence x'. For the trivial cases not covered by the case analysis, we show that decoding the message can be reduced to a binary single-deletion problem, which is easily solved with our sketch $R_1(x)$. We formalize our claim as the following:

Claim 3. *If $x \in \Sigma_2^n$ is recovered from $x' \in \sigma_2(x)$ by inserting $b_1, b_2 \in \{0, 1\}$ into x', then each inserted bit b_i increases the total number of runs by 0 or 2; otherwise, recovering x is reduced to a single-deletion problem.*

Proof. By using our previous **Run Number Lemma**, we can simply break down the problem into the following cases:

- Deleting b_1, b_2 both decreased number of runs by 0 or 2.

- Exactly one of b_i decreased number of runs by 1. Then by applying **Run Number Lemma**, the bit decreased number of runs by 1 must be deleted at the beginning or the end of x. We then use the fact that $x_1 = 1$ and $x_n = 0$ to recover that bit, so only one bit's position remains unknown. As we have shown earlier, Håstad's code can deal with the remaining single-deletion problem.

- Both b_i decrease number of runs by 1. Then the fact $x_1 = 1$ and $x_n = 0$ allows us to recover x directly.

□

Notation 6. We can take the perspective of a decoder trying to insert b_i into $x' \in \sigma_2(x)$ to recover x. Since the inserted bits can either add 0 or 2 runs, to simplify our language, we will use b_c (create) to denote the bit *creating* 2 runs and use b_{nc} (not create) to denote the bit *not creating* a run.

4.2.1 Proof of Håstad's List-Decoding Theorem

After proving the **Claim 3** above, it is not hard to verify that proving the following two lemmas proves that we can list-decode message x with sketches x', $R_1(x), R_2(x)$, which then proves **Theorem 2**. Since the first lemma studies relatively simple cases, we refer to it as **Lemma Simple**, and we denote the second lemma as **Lemma Complex**:

Lemma 6. *(***Lemma Simple***) For all $x \in \sigma_2^n$, and for all $x' \in \sigma_2(x)$, if $r_n(x) - r_{n-2}(x') = 0, 4$, then $R_1(x), R_2(x)$ are effective binary 2-deletion sketches that allows recovery of x in $O(n^2)$ time.*

Lemma 7. *(***Lemma Complex***) For all $x \in \sigma_2^n$, and for all $x' \in \sigma_2(x)$, if $r_n(x) - r_{n-2}(x') = 2$, then $R_1(x), R_2(x)$ are effective list-decodable binary 2-deletion sketches, that allows a list $L \subseteq \sigma_2^n$ of at most size 2 to be decoded in $O(n^2)$ time, such that $x \in L$.*

Proof of Lemma Simple

Proof. **Case 1:** $r_n(x) - r_{n-2}(x') = 0$. Therefore, both bits inserted increase $r_n(x)$ by 0. Assume that runs r_i, r_j are the correct runs to insert b_{nc_1} and b_{nc_2} to recover x, and without the loss of generality, assume $i \leq j$. By definition of sketch $R_1(x), R_2(x)$, we obtain equations (4.1) with r_i, r_j as unknowns. Since equations (4.1) have a unique solution, x can be uniquely recovered by solving this set of equation.

$$\begin{aligned} R_1(x) - R_1(x') &= r_i + r_j \mod 2n+1 \\ R_2(x) - R_2(x') &= r_i^2 + r_j^2 \mod 2n^2+1 \end{aligned} \tag{4.1}$$

Case 2: $r_n(x) - r_{n-2}(x') = 4$. Both bits increase $r_n(x)$ by 2. To recover the correct positions to insert b_{c1}, b_{c2}, we can insert two bits at some positions to make the sketch of current string matches the sketch of x. When two bits are inserted after index s, t respectively, denote the

string we obtain from such insertion as $y_{s,t} = x'_0 x'_1 \cdots x'_s b_{c1} x'_{s+1} \cdots x'_t b_{c2} x'_{t+1} \cdots x'_n$. Then, we would move two bits from some positions until both sketches match the sketch of x. To reason over the valid positions, define the functions $f(x', s, t) = R_1(y_{s,t})$ and $g(x', s, t) = R_2(y_{s,t})$. Expand f by the definition and we obtain: (We omit the mod taken here and later for simplicity)

$$\begin{aligned} f(x', s, t) = R_1(y_{s,t}) = \sum_{i=1}^{n} r_i(y_{s,t}) \\ =(\sum_{i=1}^{s} r_i(x')) + (r_{s^*}(x) + 1)[\sum_{i=s+1}^{t} (r_i(x') + 2)] + (r_{t^*}(x') + 3) + \sum_{i=t+1}^{n} (r_i(x') + 4) \end{aligned} \tag{4.2}$$

Similarly, expanding out the expression for g and obtain:

$$\begin{aligned} g(x', s, t) = R_2(y_{s,t}) = \sum_{i=1}^{n} (r_i(y_{s,t}))^2 \\ =(\sum_{i=1}^{s} (r_i(x'))^2) + (r_s(x') + 1)^2 + [\sum_{i=s+1}^{t} (r_i(x') + 2)^2] + (r_t(x') + 3)^2 + \sum_{i=t+1}^{n} (r_i(x') + 4)^2 \end{aligned} \tag{4.3}$$

Notice that the goal now is to not only find a solution (s, t) for the following equations (6), but also prove that the solution is unique up to the proper mod taken as specified in the sketch.

$$\begin{aligned} f(x', s, t) = R_1(x) \\ g(x', s, t) = R_2(x) \end{aligned} \tag{4.4}$$

An iterative method to solve this set of equations is to find a pair (s', t'), such that $(s', t') = \arg\min_{(s,t):f(x',s,t)=R_1(x)} |s - t|$. Then we keep $f(x', s, t) = R_1(x)$ as an invariant, and modify the values of s, t from s', t' until both constraints are satisfied. By keeping $f(x', s, t) = R_1(x)$ as an invariant, one observes that $t > t' \implies s < s'$.

To show that function g changes *monotonically* under the invariant, one can calculate the partial derivative of the ratio between g and f, with respect to the positions s, t. After some calculation, we see that $\frac{\partial |\frac{g(x',s,t)}{f(x',s,t)}|}{\partial t} < \frac{\partial |\frac{g(x',s,t)}{f(x',s,t)}|}{\partial s}$, indicating the following:

$$\forall (s, t) \neq (s', t'), f(x', s, t) = R_1(x) \implies g(x', s, t) < g(x', s', t')$$

Example 8. Assume $x' = 1111111$, and the sketch $R_1(x) = 2$. Then the list of all the possible $y_{s,t}$ obtainable from x' while keeping $f(x', s, t) = R_1(x)$ as an invariant, and their corresponding sketches $R_2(y_{s,t})$ (without the mod) are as follows:

$\mathbf{y_{s,t}}$	$\mathbf{R_2(y_{s,t})}$
$111\underline{0}1\underline{0}111$	107
$11\underline{0}111\underline{0}11$	99
$1\underline{0}11111\underline{0}1$	91

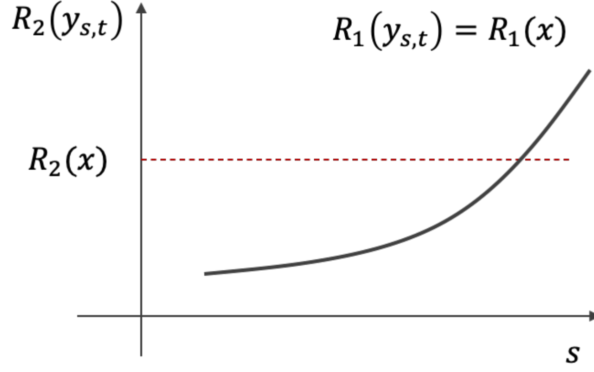

Figure 4.3: When $r_n(x) - r_{n-2}(x') = 4$, graphic representation of $R_2(y_{s,t})$ as a function of s

From here, we know that the solution is unique because $\forall(s,t)$, $f(x', s, t) = R_1(x) \implies |g(x', s, t) - g(x', s', t')| < 2n^2$, and such solution exists because we can always decode x. Therefore, with the appropriate mod taken, there exists a unique solution (s,t) that satisfies both equations. To obtain the solution in $O(n^2)$ time, we can run a brute-force algorithm on all positions (s,t), and checks if sketch of $y_{s,t}$ of current position is equal to the sketch of x. The *monotonicity* of g under the invariant will be useful in the upcoming proof of the **Lemma Complex**.

□

Proof of Lemma Complex:

Proof. Similar to proof of **Lemma Simple**, we can insert two bits at some positions to make the sketch of current string matches the sketch of x. We insert b_{nc} after index s of x' and insert b_c after index t of x'. Define $f(x', s, t)$ and $g(x', s, t)$ as the same potential function in **Lemma Simple**. To solve the equations, we use the same iterative method by finding a pair (s', t') such that $(s', t') = \arg\min_{(s,t): f(x', s, t) = R_1(x)} s + t$. We then keep $f(x', s, t) = R_1(x)$ as an invariant, and modify the value of s, t until both constraints are satisfied. Similarly, by keeping $f(x', s, t) = R_1(x)$ as an invariant, if $s > s' \implies t > t'$.

One may verify that $|\frac{\partial f(x', s, t)}{\partial s}| \leq |\frac{\partial f(x', s, t)}{\partial t}|$, which indicates that when changing the value of s, t while keeping the first constraint satisfied, s changes *faster* than t. To establish the list-decoding result, we prove the uniqueness of solution by casing on whether $s \leq t$:

A. $s \leq t$. From previous proof, we obtain that $\frac{\partial |\frac{g(x', s, t)}{f(x', s, t)}|}{\partial t} > \frac{\partial |\frac{g(x', s, t)}{f(x', s, t)}|}{\partial s}$, indicating that as long as $s \leq t$ and $f(x', s, t) = R_1(x)$, the value of $g(x', s, t)$ changes *monotonically* with the increase of s. Therefore, there exists at most one solution (s, t) that satisfies both equations when $s \leq t$.

B. $s > t$. We can apply the same analysis as above. Since the value of $g(x', s, t)$ changes *monotonically* in the opposite direction as case A, there exists at most one solution (s, t) for both equations as long as $s > t$.

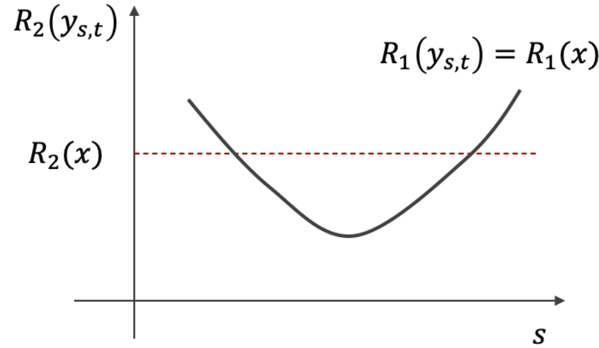

Figure 4.4: When $r_n(x) - r_{n-2}(x') = 2$, graphic representation of $R_2(y_{s,t})$ as a function of s

Since at most one solution can be obtained from each case, at most 2 strings can be obtained from solving the equation. Additionally, we are guaranteed that one of the strings is x, which proves the list-decoding result. To understand how to decode in $O(n^2)$ time, one can calculate the value of f, g on all valid (s, t) pairs, and check the values obtained in each position with sketch of x with brute-force algorithm. For up to two solutions obtained, output a unique string corresponding to each solution.

□

Now we come back to prove the **Run Number Lemma**.

Proof of Run Number Lemma:

Proof. If the bit b is not deleted from the beginning or the end of x, then b has two neighbors in x. In binary, it must be one of the following cases: If at least one neighbor is the same as b, then $r_n(x)$ is decreased by 0. If both neighbors are different from b, then $r_n(x)$ is decreased by 2. Otherwise, when at least one bit is inserted at the beginning or end of x', it can happen that $r_n(x)$ is decreased by 1.

Example 9. If $x = 10\underline{1}110$ and $x' = 10110$, then $r_6(x) = 4$ and $r_5(x') = 4$.

Example 10. If $x = 1\underline{0}1110$ and $x' = 11110$, then $r_6(x) = 4$ and $r_5(x') = 2$.

□

4.3 Extensions of Current Result

There are several extensions and claims that can be made about this result: First, the list-decodable q-ary 2-deletion code specified in this paper also guards against single-deletion or single-insertion error over q-ary alphabet. This simply follows from the fact that Håstad's sketch $R_1(x)$ corrects single-deletion and single-insertion error in linear time as well. Meanwhile, $R_1(x)$ is also an effective sketch against single-error (substitution).

Secondly, the sketch $R_1(x), R_2(x)$ used in Håstad's binary 2-deletion code can also guard against binary 2-insertion error, allowing a list-decoding result up to list size 2. Combined with the first extension, this presents a binary code that list-decodes against 2 edit-errors with $3\log_2 n + O(\log\log n)$ redundancy, with list size 2. Combining this 2 edit-error extension with our main result, we obtain a list-decodable code that corrects 2-edit errors with $6\log_2 n + O_q(\log\log n)$ redundancy, with list size 2.

4.4 Encoding Algorithm

Overall, the encoding and decoding algorithm of our q-ary 2-deletion code can be both split into two stages: the first stage for binary 2-deletion code, and the second stage for the reduction. Assume $\texttt{Enc}_2$ and $\texttt{Dec}_2$ are the encoding and decoding algorithms of some systematic binary 2-deletion code, and the sketch obtained by the same code is $s_2(x)$. Similarly, let $\texttt{EncRed}_q$ and $\texttt{DecRed}_q$ be the encoding and decoding algorithms of our reduction step. We will also use $\texttt{Rep}_i$ to denote the function that calculate i-repetition code.

Algorithm 2 Algorithm for Encoding q-ary 2-deletion code

```
 1: function EncodeHelper(x, Enc_2)
 2:     αSeq ← α(x)
 3:     sketch_2 ← Enc_2(αSeq)
 4:     sketch_red ← EncRed_q(x)
 5:     sketch_q ← (sketch_2, sketch_red)
 6:     return sketch_q
 7: end function

 8: function EncodeQ(x, Enc_2)
 9:     k ← EncodeHelper(x)
10:     k' ← EncodeHelper(k)
11:     return (x, k, Rep_3(k'))
12: end function
```

If we are working with an optimal binary 2-deletion code, the sketch $s_2(x)$ uses $r_2(n)$ bits. The reduction sketch $\texttt{sketch}_{red}$ uses a total of $3\log_2 n + O_q(\log\log n)$ bits. Notice that we can compress any binary sketch into a q-ary sketch with constant overhead. Therefore, the overall redundancy is bounded by $(3 + o(1))\log_2 n + r_2(n)$.

4.5 Decoding Algorithm

For decoding, we can assume that there exists an algorithm to parse the input to the format we desire, since we are working under the *systematic code* setting. We will first use the decoding algorithm for repetition code to recover k' as the sketch of the sketch. Then solving the 2-deletion

Algorithm 3 Algorithm for Decoding q-ary 2-deletion code

```
1: function DecodeHelper(x', sketch_q, Dec_2)
2:     (sketch_2, sketch_red) ← Parse(sketch_q)
3:     α(x) ← Dec_2(x', sketch_2)
4:     x ← DecRed(x', sketch_red, α(x))
5:     return x
6: end function

7: function DecodeQ(y, Dec_2)
8:     (x', s, s') ← Parse(y)
9:     k' ← RepDec(s')
10:    k ← DecodeHelper(s, k', Dec_2)
11:    x ← DecodeHelper(x', k, Dec_2)
12:    return x
13: end function
```

problem on our sketch to obtain k, eventually solve the 2-deletion problem on $x' \in \sigma_2(x)$ to recover x. Any parsing can be done in linear time, therefore the entire procedure takes $O(n^2)$ time overall, assuming $\mathtt{Dec}_2$ uses $O(n^2)$ time. If we set $s_2(x) = (R_1(x), R_2(x))$ as in Håstad's code, then we obtain a list-decodable code that allows $O(n^2)$ time decoding, and list size up to 2. This code uses $6\log_2 n + O_q(\log\log n)$ redundancy bits.

Chapter 5

Conclusion and Future Directions

5.1 Conclusion

In this thesis, we presented a construction of q-ary 2-deletion codes efficiently, as well as a detailed proof for Håstad's list-decodable binary 2-deletion codes [8]. The correctness of the codes was based on many combinatorial properties of the numeric sequences. The application of such properties in the proof was successful in arriving at the results of this thesis, but there are many drawbacks to use them as well.

The combinatorial properties, such as the properties of the indicator strings used in this thesis, prove to be elegant and yield deterministic and concise results. The VT-code [10] and its extension [6] are great examples for the elegance of these methods. We have discussed their effectiveness in the introduction chapter as well. On the other hand, it is hard to ignore the difficulty in discovering such nice combinatorial properties. For example, it is natural to wonder if sketches like $\sum_i ix_i$ could result in effective binary k-deletion code, but all such candidate sketches so far have met with counterexamples. Similarly, many attempts that were made during this project to find similar properties of 2-deletion codes failed as well. Therefore, it is worthwhile for anyone who wants to find a efficient k-deletion code to consider more general frameworks, and find a balance between general frameworks and the combinatorial properties.

There are many existing frameworks and techniques for k-deletion codes for k as a constant, as we have introduced in the related works section. There are frameworks for k as a fraction of n as well, and these frameworks are potentially better in generalizing the k-deletion constructions to q-ary case. For example, a framework called *concatenated codes* was used in [31], [28] and [32] as well as other literatures. Concatenated code uses an alphabet of large size to encode an *outer code*, where each symbol of the outer code is encoded with an *inner code* by a sequence of symbols in a smaller alphabet. This technique opens up the opportunity to apply binary results onto large alphabet results and is successful in dealing with deletion error being a constant fraction of the message in [32]. This technique has disadvantages as well, such as having too much overhead when k values are small. Therefore, finding a balance between the combinatorial techniques and the general frameworks might be another great direction to explore.

5.2 Future Work

From the result of this thesis, we identify several major open questions on the related topics:

The first question is that, is it possible to find the optimal binary k-deletion code, with both redundancy $(2k + o(1)) \log n$ that matches the existential bound, and an efficient decoding algorithm? Regarding the redundancy, Håstad's binary 2-deletion code [9] matched the existential bound for $k = 2$, while Sima, Gabrys and Bruck's result [5] showed that it is possible for the general case to get a $4k \log n$ redundancy construction. However, in order for the code to be useful in practice, it is crucial to have an efficient decoding algorithm as well. In this sense, result of [5] is not ideal because the decoding algorithm runs in exponential time with respect to k. Meanwhile, works of Cheng, Jin, Li and Wu [3] and Haeupler [4] showed $O(k \log n)$ redundancy construction of a binary k-deletion code with $poly(n)$ decoding time. Their code have a much more efficient decoding algorithm than [5], but the redundancy of the code is less optimal. Therefore we ask: Is there an algorithm that allows both optimal redundancy and efficient decoding?

The second question relates to our q-ary extension of the k-deletion code. As we mentioned at the end of Chapter 3, one major standing issue is that α-indicator sequence can be shared across many messages, so our result does not directly apply to k-deletion problems in general. One direction we can take is to generalize α-indicator to the case for constant k. One major challenge in continue to use α-indicator is that, to uniquely decode the message out of polynomially many candidate that share the same α-indicator, one may need to create or make use of k-burst deletion codes, with each burst of up to k-deletions. Another direction one might choose is to avoid using α-indicator sequence and resort to a general framework, as we have mentioned in our conclusion. The major challenge in this direction would be finding another framework to complete the reduction.

The third question is related to an observation on Håstad's list-decodable binary 2-deletion code [8]. One notice that if the receiver of Håstad's code is allowed to ask a $\log_2 n$ bits long question to the sender of message and sender is allowed to reply 1 bit, then the receiver can just query the first position that the two strings in the list differ, and the total bits communicated is at most $4 \log n + O(1)$, which is exactly the existential bound of binary 2-deletion code. Our question is: In general, if we allow constant rounds of communication, is there an efficient communication algorithm that matches the existential bound up to $o(\log n)$ factor? We have seen that constant round communication is easily more powerful than a single-round procedure, but the limitations of using constant rounds of communication in deletion codes is still unknown. Since the question that receiver asks might be depended upon the position of deletion, the constant-round procedure is related to *oblivious* model of this problem, which assumes the deletions are not adversarial, but random. Papers like [31] explored the potential of using oblivious deletion model, rather than the worst-case adversarial deletion model. However, the effectiveness of *oblivious* model on the constant-round procedures is still unknown.

Bibliography

[1] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Dokl.*, 10:707–710, 1965.

[2] Joshua Brakensiek, Venkatesan Guruswami, and Samuel Zbarsky. Efficient low-redundancy codes for correcting multiple deletions. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1884–1892. ACM, New York, 2016.

[3] Kuan Cheng, Zhengzhong Jin, Xin Li, and Ke Wu. Deterministic document exchange protocols, and almost optimal binary codes for edit errors. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 200–211. IEEE Computer Society, 2018.

[4] Bernhard Haeupler. Optimal document exchange and new codes for small number of insertions and deletions. *CoRR*, abs/1804.03604, 2018.

[5] Jin Sima, Ryan Gabrys, and Jehoshua Bruck. Optimal systematic t-deletion correcting codes. *submitted to 2020 IEEE International Symposium on Information Theory (ISIT)*, 2020.

[6] Grigory Tenengolts. Nonbinary codes, correcting single deletion or insertion. *IEEE Trans. Inform. Theory*, 30(5):766–769, 1984.

[7] Jin Sima, Ryan Gabrys, and Jehoshua Bruck. Optimal codes for the q-ary deletion channel. *submitted to 2020 IEEE International Symposium on Information Theory (ISIT)*, 2020.

[8] Johan Håstad. v5. *unpublished*, 2017.

[9] Johan Håstad. Ver3. *unpublished*, 2015.

[10] R. R. Varšamov and G. M. Tenengolts. A code which corrects single asymmetric errors. *Avtomat. i Telemeh.*, 26:288–292, 1965.

[11] Ankur A. Kulkarni and Negar Kiyavash. Non-asymptotic upper bounds for deletion correcting codes. *CoRR*, abs/1211.3128, 2012.

[12] Daniel Cullina and Negar Kiyavash. An improvement to levenshtein's upper bound on the cardinality of deletion correcting codes. *CoRR*, abs/1302.6562, 2013.

[13] Daniel Cullina and Negar Kiyavash. Generalized sphere-packing and sphere-covering bounds on the size of codes for combinatorial channels. *CoRR*, abs/1405.1464, 2014.

[14] N. J. A. Sloane. On single-deletion-correcting codes. In *Codes and designs (Columbus, OH, 2000)*, volume 10 of *Ohio State Univ. Math. Res. Inst. Publ.*, pages 273–291. de Gruyter,

Berlin, 2002.

[15] Albertus S. J. Helberg and Hendrik C. Ferreira. On multiple insertion/deletion correcting codes. *IEEE Trans. Inf. Theory*, 48(1):305–308, 2002.

[16] Filip Paluncic, Khaled A. S. Abdel-Ghaffar, Hendrik C. Ferreira, and Willem A. Clarke. A multiple insertion/deletion correcting code for run-length limited sequences. *IEEE Trans. Inf. Theory*, 58(3):1809–1824, 2012.

[17] Venkatesan Guruswami and Carol Wang. Deletion codes in the high-noise and high-rate regimes. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2015*, volume 40 of *LIPIcs*, pages 867–880. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[18] Jin Sima and Jehoshua Bruck. Optimal k-deletion correcting codes. *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 847–851, 2019.

[19] Djamal Belazzougui. Efficient deterministic single round document exchange for edit distance. *CoRR*, abs/1511.09229, 2015.

[20] Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization strings: explicit constructions, local decoding, and applications. In *STOC'18—Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 841–854. ACM, New York, 2018.

[21] Kuan Cheng, Xin Li, and Ke Wu. Synchronization strings: Efficient and fast deterministic constructions over small alphabets. *CoRR*, abs/1710.07356, 2017.

[22] Kuan Cheng, Bernhard Haeupler, Xin Li, Amirbehshad Shahrasbi, and Ke Wu. Synchronization strings: Efficient and fast deterministic constructions over small alphabets. *CoRR*, abs/1803.03530, 2018.

[23] Jin Sima, Netanel Raviv, and Jehoshua Bruck. Two deletion correcting codes from indicator vectors. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 421–425. IEEE, 2018.

[24] Ryan Gabrys and Frederic Sala. Codes correcting two deletions. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 426–430. IEEE, 2018.

[25] Tuan A. Le and Hieu D. Nguyen. New multiple insertion/deletion correcting codes for non-binary alphabets. *IEEE Trans. Inf. Theory*, 62(5):2682–2693, 2016.

[26] Clayton Schoeny, Antonia Wachter-Zeh, Ryan Gabrys, and Eitan Yaakobi. Codes for correcting a burst of deletions or insertions. *CoRR*, abs/1602.06820, 2016.

[27] Clayton Schoeny, Frederic Sala, and Lara Dolecek. Novel combinatorial coding results for DNA sequencing and data storage. In *51st Asilomar Conference on Signals, Systems, and Computers, ACSSC 2017*, pages 511–515. IEEE, 2017.

[28] Bernhard Haeupler, Amirbehshad Shahrasbi, and Madhu Sudan. Synchronization strings: List decoding for insertions and deletions. *Proceeding of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 76:1–76:14, 2018.

[29] Shu Liu, Ivan Tjuawinata, and Chaoping Xing. On list decoding of insertion and deletion errors. *CoRR*, abs/1906.09705, 2019.

[30] Antonia Wachter-Zeh. List decoding of insertions and deletions. *IEEE Trans. Inf. Theory*, 64(9):6297–6304, 2018.

[31] Venkatesan Guruswami and Ray Li. Coding against deletions in oblivious and online models. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 625–643. SIAM, 2018.

[32] Venkatesan Guruswami, Bernhard Haeupler, and Amirbehshad Shahrasbi. Optimally resilient codes for list-decoding from insertions and deletions. *Electronic Colloquium on Computational Complexity (ECCC)*, 26:153, 2019.