Beyond Single-Deletion Correcting Codes: Substitutions and Transpositions

Ryan Gabrys
ECE Department, University of California, San Diego, CA, USA

Venkatesan Guruswami
EECS Department, University of California, Berkeley, CA, USA

João Ribeiro
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Ke Wu
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We consider the problem of designing low-redundancy codes in settings where one must correct deletions in conjunction with substitutions or adjacent transpositions; a combination of errors that is usually observed in DNA-based data storage. One of the most basic versions of this problem was settled more than 50 years ago by Levenshtein, who proved that binary Varshamov-Tenengolts codes correct one arbitrary edit error, i.e., one deletion *or* one substitution, with nearly optimal redundancy. However, this approach fails to extend to many simple and natural variations of the binary single-edit error setting. In this work, we make progress on the code design problem above in three such variations:

- We construct linear-time encodable and decodable length-n non-binary codes correcting a single edit error with nearly optimal redundancy $\log n + O(\log\log n)$, providing an alternative simpler proof of a result by Cai, Chee, Gabrys, Kiah, and Nguyen (IEEE Trans. Inf. Theory 2021). This is achieved by employing what we call *weighted VT sketches*, a new notion that may be of independent interest.
- We show the existence of a binary code correcting one deletion *or* one adjacent transposition with nearly optimal redundancy $\log n + O(\log\log n)$.
- We construct linear-time encodable and list-decodable binary codes with list-size 2 for one deletion *and* one substitution with redundancy $4\log n + O(\log\log n)$. This matches the existential bound up to an $O(\log\log n)$ additive term.

2012 ACM Subject Classification Theory of computation → Error-correcting codes

Keywords and phrases Synchronization errors, Optimal redundancy, Explicit codes

Digital Object Identifier 10.4230/LIPIcs.APPROX/RANDOM.2022.8

Category RANDOM

Related Version *Full Version*: https://arxiv.org/abs/2112.09971

Funding *Venkatesan Guruswami*: Research supported in part by the NSF grants CCF-1814603 and CCF-2107347. Part of the work was done while at Carnegie Mellon University.
João Ribeiro: Research supported in part by the NSF grants CCF-1814603 and CCF-2107347 and by the NSF award 1916939, DARPA SIEVE program, a gift from Ripple, a DoE NETL award, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.
Ke Wu: Research supported in part by a DARPA SIEVE award, SRI Subcontract Number 53978, and DARPA Prime Contract Number HR00110C0086.

© Ryan Gabrys, Venkatesan Guruswami, João Ribeiro, and Ke Wu;
licensed under Creative Commons License CC-BY 4.0
Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2022).
Editors: Amit Chakrabarti and Chaitanya Swamy; Article No. 8; pp. 8:1–8:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Deletions, substitutions, and transpositions are some of the most common types of errors jointly affecting information encoded in DNA-based data storage systems [27, 14]. Therefore, it is natural to consider models capturing the interplay between these types of errors, along with the best possible codes for these settings. More concretely, one usually seeks to pin down the optimal redundancy required to correct such errors, and also to design fast encoding and decoding procedures for low-redundancy codes. It is well-known that deletions are challenging to handle even in isolation, since they cause a loss of synchronization between sender and receiver. The situation where one aims to correct deletions in conjunction with other reasonable types of errors is even more difficult. Our understanding of this interplay remains scarce even in basic settings where only one or two such worst-case errors may occur.

One of the most fundamental settings where deletions interact with the other types of errors mentioned above is that of correcting a single *edit* error (i.e., a deletion, insertion, or substitution) over a *binary* alphabet. In this case, linear-time encodable and decodable binary codes correcting a single edit error with nearly optimal redundancy have been known for more than 50 years. Levenshtein [13] showed that the binary Varshamov-Tenengolts (VT) code [24] defined as

$$\mathcal{C} = \left\{ x \in \{0,1\}^n : \sum_{i=1}^{n} i \cdot x_i = a \mod (2n+1) \right\} \tag{1}$$

corrects one arbitrary edit error. For an appropriate choice of a, this code has redundancy at most $\log n + 2$, and it is not hard to see that at least $\log n$ bits of redundancy are required to correct one edit error. Remarkably, a greedy Gilbert-Varshamov-type argument only guarantees the existence of single-edit correcting codes with redundancy $2 \log n$ – much higher than what can be achieved with the VT code. We recommend Sloane's excellent survey [18] for a more in-depth overview of binary VT codes and their connections to combinatorics.

Although the questions of determining the optimal redundancy and giving nearly-optimal explicit constructions of codes in the binary single-edit setting have been settled long ago, the underlying approach fails to extend to many simple, natural variations of this setting combining deletions with substitutions and transpositions. In this work, we make progress on these questions in three such fundamental variations, which we proceed to describe next.

1.1 *Non-binary* single-edit correcting codes

We begin by considering the problem of correcting a single arbitrary edit error over a non-binary alphabet. This setting is especially relevant due to its connection to DNA-based data storage, which requires coding over a 4-ary alphabet. In this case, the standard *VT sketch*

$$f(x) = \sum_{i=1}^{n} i \cdot x_i \mod N, \tag{2}$$

which allows us to correct one binary edit error in (1) with an appropriate choice of N, is no longer enough. Instead, we present a natural extension of the binary VT code to a non-binary alphabet via a new notion of *weighted VT sketches*, which yields an order-optimal result.

▶ **Theorem 1.** *There exists a 4-ary*[1] *single-edit correcting code* $\mathcal{C} \subseteq \{0,1,2,3\}^n$ *with* $\log n + \log\log n + 7 + o(1)$ *bits of redundancy, where* $o(1) \to 0$ *when* $n \to \infty$*. Moreover, there exists a single edit-correcting code* $\mathcal{C} \subseteq \{0,1,2,3\}^n$ *with* $\log n + O(\log\log n)$ *redundant bits that supports linear-time encoding and decoding.*

[1] A 4-ary alphabet is relevant for DNA-based data storage.

This problem was previously considered by Cai, Chee, Gabrys, Kiah, and Nguyen [2], who proved an analogous result. Our existential result requires 6 fewer bits of redundancy than the corresponding result from [2], and our explicit code supports linear time encoding and decoding procedures, while the explicit code from [2] requires $\Theta(n \log n)$ time encoding [22]. However, we believe that our more significant contribution in this setting is the simpler approach we employ to prove Theorem 1 via weighted VT sketches. The technique of weighted VT sketches seems quite natural and powerful and may be of independent interest.

We note that the existential result in Theorem 1 extends to arbitrary alphabet size q with $\log n + O_q(\log \log n)$ redundant bits, but we focus on $q = 4$ since it is the most interesting setting and provides the clearest exposition of our techniques. More details can be found in Section 3, where we also present a more in-depth discussion on why the standard VT sketch (2) does not suffice in the non-binary case.

1.2 Binary codes correcting one deletion *or* one adjacent transposition

As our second contribution, we consider the interplay between deletions and adjacent transpositions, which map 01 to 10 and vice-versa. An adjacent transposition may be seen as a special case of a burst of two substitutions. Besides its relevance to DNA-based storage, the interplay between deletions and transpositions is an interesting follow-up to the single-edit setting discussed above because the VT sketch is highly ineffective when dealing with transpositions, while it is the staple technique for correcting deletions and substitutions. The issue is that, if $y, y' \in \{0,1\}^n$ are obtained from $x \in \{0,1\}^n$ via any two adjacent transpositions of the form $01 \mapsto 10$, then $f(y) = f(y') = f(x) - 1$, where we recall $f(z) = \sum_{i=1}^{n} i \cdot z_i \mod N$ is the VT sketch. This implies that knowing the VT sketch $f(x)$ reveals almost no information about the adjacent transposition, since correcting an adjacent transposition is equivalent to finding its location.

In this setting, the best known redundancy lower bound is $\log n$ (the same as for single-deletion correcting codes), while the best known existential upper bound is $2 \log n$, obtained by naively intersecting a single-deletion correcting code and a single-transposition correcting code. A code with redundancy $\log n + O(1)$ was claimed in [7, Section III], but the argument there is flawed. In this work, we determine the optimal redundancy of codes in this setting up to an $O(\log \log n)$ additive term via a novel marker-based approach. More precisely, we prove the following result, more details of which can be found in Section 4.

▶ **Theorem 2.** *There exists a binary code $\mathcal{C} \subseteq \{0,1\}^n$ correcting one deletion or one transposition with redundancy $\log n + O(\log \log n)$.*

Since we know that every code that corrects one deletion also corrects one insertion [13], we also conclude from Theorem 2 that there exists a binary code correcting one deletion, one insertion, or one transposition with nearly optimal redundancy $\log n + O(\log \log n)$.

1.3 Binary codes for one deletion *and* one substitution

To conclude, we make progress on the study of single-deletion single-substitution correcting codes. Recent work by Smagloy, Welter, Wachter-Zeh, and Yaakobi [19] constructed efficiently encodable and decodable binary single-deletion single-substitution correcting codes with redundancy close to $6 \log n$. On the other hand, it is known that $2 \log n$ redundant bits are required, and a greedy approach shows the *existence* of a single-deletion single-substitution correcting code with redundancy $4 \log n + O(1)$.

In this setting, we ask what improvements are possible if we relax the unique decoding requirement slightly and instead require that the code be *list-decodable with list-size* 2. There, our goal is to design a low-redundancy code $\mathcal{C} \subseteq \{0,1\}^n$ such that for any corrupted string $y \in \{0,1\}^{n-1} \cup \{0,1\}^n$ there are at most two codewords $x, x' \in \mathcal{C}$ that can be transformed into y via some combination of at most one deletion and one substitution. This is the strongest possible requirement after unique decoding, which corresponds to lists of size 1.

The best known *existential* upper bound on the optimal redundancy in the list-decoding setting is still $4 \log n + O(1)$ via the Gilbert-Varshamov-type greedy algorithm. We give an explicit list-decodable code with list-size 2 correcting one deletion and one substitution with redundancy matching the existential bound up to an $O(\log \log n)$ additive term. At a high level, this code is obtained by combining the standard VT sketch (2) with *run-based sketches*, which have been recently used in the design of two-deletion correcting codes [9]. More precisely, we have the following result, details of which can be found in Section 5.

▶ **Theorem 3.** *There exists a linear-time encodable and decodable binary* list-size 2 *single-deletion single-substitution correcting code* $\mathcal{C} \subseteq \{0,1\}^n$ *with* $4 \log n + O(\log \log n)$ *bits of redundancy.*

Subsequently to the appearance of our work online, Song, Cai, and Nguyen [20] constructed a list-decodable code with list-size 2 for one deletion and one substitution with redundancy $3 \log n + O(\log \log n)$.

1.4 Related work

Recently, there has been a flurry of works making progress in coding-theoretic questions analogous to the ones we consider here in other extensions of the binary single-edit error setting. A line of work culminating in [1, 9, 17] has succeeded in constructing explicit low-redundancy codes correcting a constant number of worst-case deletions. Constructions focused on the two-deletion case have also been given, e.g., in [17, 6, 9]. Explicit binary codes correcting a sublinear number of edit errors with redundancy optimal up to a constant factor have also been constructed recently [3, 10]. Other works have considered the related setting where one wishes to correct a burst of deletions or insertions [15, 12, 26], or a combination of duplications and edit errors [23]. Following up on [19], codes correcting a combination of more than one deletion and one substitution were given in [21] with sub-optimal redundancy. List-decodable codes in settings with indel errors have also been considered before. For example, Wachter-Zeh [25] and Guruswami, Haeupler, and Shahrasbi [8] study list-decodability from a linear fraction of deletions and insertions.

Most relevant to our result in Section 1.3, Guruswami and Håstad [9] constructed an explicit list-size two code correcting two deletions with redundancy $3 \log n + O(\log \log n)$, thus beating the greedy existential bound in this setting.

With respect to the interplay between deletions and transpositions, Gabrys, Yaakobi, and Milenkovic [7] constructed codes correcting a single deletion *and* many adjacent transpositions. In an incomparable regime, Schulman and Zuckerman [16], Cheng, Jin, Li, and Wu [4], and Haeupler and Shahrasbi [11] constructed explicit codes with good redundancy correcting a linear fraction of deletions and insertions and a nearly-linear fraction of transpositions.

2 Preliminaries

2.1 Notation and conventions

We denote sets by uppercase letters such as S and T or uppercase calligraphic letters such as $\mathcal{C}$, and define $[n] = \{0, 1, \ldots, n-1\}$, $S^{\leq k} = \bigcup_{i=0}^{k} S^i$, and $S^* = \bigcup_{i=0}^{\infty} S^i$ for any set S. The symmetric difference between two sets S and T is denoted by $S \triangle T$. We use the notation

$\{\{a, a, b\}\}$ for multisets, which may contain several copies of each element. Given two strings x and y over a common alphabet Σ, we denote their concatenation by $x\|y$ and write $x[i:j] = (x_i, x_{i+1}, \dots, x_j)$. We say $y \in \Sigma^k$ is a *k-subsequence* of $x \in \Sigma^n$ if there are k indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $x_{i_j} = y_j$ for $j = 1, \dots, k$, in which case we also call x an *n-supersequence* of y. Moreover, we say $x[i:j]$ is an *a-run* of x if $x[i:j] = a^{j-i+1}$ for a symbol $a \in \Sigma$. We denote the base-2 logarithm by log. A length-n *code* $\mathcal{C}$ is a subset of Σ^n for some alphabet Σ which will be clear from context. In this work, we are interested in the *redundancy* of certain codes (measured in bits), which we define as $n \log |\Sigma| - \log |\mathcal{C}|$.

2.2 Error models and codes

Since we will be dealing with three distinct but related models of worst-case errors, we begin by defining the relevant standard concepts in a more general way. We may define a worst-case error model over some alphabet Σ by specifying a family of *error balls* $\mathcal{B} = \{B(y) \subseteq \Sigma^* : y \in \Sigma^*\}$, where the $B(y)$ can be arbitrary sets. Usually, $B(y)$ contains all strings that can be corrupted into y by applying an allowed error pattern. We proceed to define unique decodability of a code $\mathcal{C} \subseteq \Sigma^n$ with respect to an error model.

▶ **Definition 4** (Uniquely decodable code). *We say a code $\mathcal{C} \subseteq \Sigma^n$ is* uniquely decodable (with respect to $\mathcal{B}$) *if $|B(y) \cap \mathcal{C}| \leq 1$ for all $y \in \Sigma^*$.*

Throughout this work the underlying error model will always be clear from context, so we do not mention it explicitly. We will also consider *list-decodable* codes with small list size in Section 5, and so we require the following more general definition.

▶ **Definition 5** (List-size t decodable code). *We say a code $\mathcal{C} \subseteq \Sigma^n$ is* list-size t decodable (with respect to $\mathcal{B}$) *if $|B(y) \cap \mathcal{C}| \leq t$ for all $y \in \Sigma^*$.*

Note that uniquely decodable codes correspond exactly to list-size 1 codes. Moreover, we remark that for the error models considered in this work and constant t, the best existential bound for list-size t codes coincides with the best existential bound for uniquely decodable codes up to a constant additive term.

We proceed to describe the type of errors we consider. A deletion transforms a string $x \in \Sigma^n$ into one of its $(n-1)$-subsequences. An insertion transforms a string $x \in \Sigma^n$ into one of its $(n+1)$-supersequences. A substitution transforms $x \in \Sigma^n$ into a string $x' \in \Sigma^n$ that differs from x in exactly one coordinate. An adjacent transposition transforms strings of the form ab into ba. More formally, a string $x \in \Sigma^n$ is tranformed into a string $x' \in \Sigma^n$ with the property that $x'_k = x_{k+1}$ and $x'_{k+1} = x_k$ for some k, and $x'_i = x_i$ for $i \neq k, k+1$.

We can now instantiate the above general definitions under the specific error models considered in this paper. In the case of a single edit, $B(y)$ contains all strings which can be transformed into y via at most one deletion, one insertion, or one substitution. In the case of one deletion *and* one substitution, $B(y)$ contains all strings that can be transformed into y by applying at most one deletion and at most one substitution. Finally, in the case of one deletion or one adjacent transposition, $B(y)$ contains all strings that can be transformed into y by applying either at most one deletion or at most one transposition.

3 Non-binary single-edit correcting codes

In this section, we describe and analyze the code construction used to prove Theorem 1. Before we do so, we provide some intuition behind our approach.

3.1 The binary alphabet case as a motivating example

It is instructive to start off with the binary alphabet case and the VT code described in (1), which motivates our approach for non-binary alphabets. More concretely, we may wonder whether a direct generalization of $\mathcal{C}$ to larger alphabets also corrects a single edit error, say

$$\mathcal{C}' = \left\{ x \in [q]^n \,\middle|\, \sum_{i=1}^{n} i x_i = s \mod (1+2qn), \quad \forall c \in [q] : |\{i : x_i = c\}| = s_c \mod 2 \right\},$$

where $[q] = \{0,1,\dots,q-1\}$ and $s, s_0, \dots, s_{q-1}$ are appropriately chosen integers. However, this approach fails already over a ternary alphabet $\{0,1,2\}$. In fact, $\mathcal{C}'$ cannot correct worst-case deletions of 1's because it does not allow us to distinguish between $\dots\underline{1}02\dots$ and $\dots02\underline{1}\dots$, which can be obtained one from the other by deleting and inserting a 1 in the underlined positions. More generally, there exist codewords $x \in \mathcal{C}'$ with substrings $(x_j = 1, x_{j+1}, \dots, x_k)$ not consisting solely of 1's satisfying

$$\sum_{i=j+1}^{k} (x_i - 1) = 0. \tag{3}$$

This is problematic since the string x' obtained by deleting $x_j = 1$ from x and inserting a 1 between x_k and x_{k+1} is also in $\mathcal{C}'$. In order to avoid the problem encountered by $\mathcal{C}'$, we instead consider a *weighted VT sketch* of the form

$$f_w(x) = \sum_{i=1}^{n} i \cdot w(x_i) \mod N \tag{4}$$

for some weight function $w : [q] \to \mathbb{Z}$ and an appropriate modulus N. Using f_w instead of the standard VT sketch $f(x) = \sum_{i=1}^{n} i x_i \mod N$ in the argument above causes the condition (3) for an uncorrectable 1-deletion to be replaced by $\sum_{i=j+1}^{k}(w(x_i) - w(1)) = 0$. Then, choosing $0 \le w(0) < w(1) < w(2) < \dots < w(q-1)$ appropriately allows us to correct the deletion of a 1 in x given knowledge of $f_w(x)$ provided that x satisfies a simple runlength constraint. In turn, encoding an arbitrary message z into a string x satisfying this constraint can be done very efficiently via a direct application of the simple *runlength replacement* technique from [15] using few redundant bits. Theorem 1 is then obtained by instantiating the weighted VT sketch (4) with an appropriate weight function and modulus.

3.2 Code construction

In this section, we present our construction of a 4-ary single-edit correcting code which leads to Theorem 1. As discussed in Section 3.1, given an arbitrary string $x \in \{0,1,2,3\}^n$ we consider a weighted VT sketch

$$f(x) = \sum_{i=1}^{n} i \cdot w(x_i) \mod [1 + 2n \cdot (2\log n + 12)],$$

where $w(0) = 0$, $w(1) = 1$, $w(2) = 2\log n + 11$, and $w(3) = 2\log n + 12$, along with the count sketches $h_c(x) = |\{i : x_i = c\}| \mod 2$ for $c \in \{0,1,2\}$. Intuitively, the count sketches allow us to cheaply narrow down exactly what type of deletion or substitution occurred (but not its position). As we shall prove later on, successfully correcting the deletion of an a boils down to ensuring that

$$\sum_{i=j}^{k}(w(x_i) - w(a)) \neq 0 \tag{5}$$

for all $1 \le j \le k \le n$ such that there is $i \in [j,k]$ with $x_i \neq a$. We call strings x that satisfy this property for every a *regular*, and proceed to show that enforcing a simple runlength constraint on x is sufficient to guarantee that it is regular.

▶ **Lemma 6.** *Suppose $x \in \{0,1,2,3\}^n$ satisfies the following property:* If x' denotes the subsequence of x obtained by deleting all 1's and 3's and x'' denotes the subsequence obtained by deleting all 0's and 2's, it holds that all 0-runs of x' and all 3-runs of x'' have length at most $\log n + 3$. *Then, x is regular.*

Proof. See the full version [5]. ◀

Let $\mathcal{G} \subseteq \{0,1,2,3\}^n$ denote the set of regular strings. Given the above definitions, we set our code to be

$$\mathcal{C} = \mathcal{G} \cap \{x \in \{0,1,2,3\}^n : f(x) = s, h_c(x) = s_c, c \in \{0,1,2\}\} \tag{6}$$

for appropriate choices of $s \in \{0,\dots,1+2n\cdot(2\log n+12)\}$ and $s_c \in \{0,1\}$ for $c = 0,1,2$. A straightforward application of the probabilistic method shows that most strings are regular.

▶ **Lemma 7.** *Let X be sampled uniformly at random from $\{0,1,2,3\}^n$. Then, we have* $\Pr[X \text{ is regular}] \ge 7/8$.

As a result, by the pigeonhole principle there exist choices of s, s_0, s_1, s_2 such that

$$|\mathcal{C}| \ge \frac{7\cdot 4^n}{8\cdot 2^3\cdot(1+2n\cdot(2\log n+12))}.$$

This implies that we can make it so that $\mathcal{C}$ has $\log n + \log\log n + 6 + o(1)$ bits of redundancy, where $o(1) \to 0$ when $n \to \infty$, as desired. If n is not a power of two, then taking ceilings yields at most one extra bit of redundancy for a total of $\log n + \log\log n + 7 + o(1)$ bits, as claimed.

It remains to show that $\mathcal{C}$ corrects a single edit in linear time and that a standard modification of $\mathcal{C}$ admits a linear time encoder. Observe that if a codeword $x \in \mathcal{C}$ is corrupted into a string y by a single edit error, we can tell whether it was a deletion, insertion, or substitution by computing $|y|$. Therefore, we treat each such case separately. Since correcting one substitution in our code is analogous to correcting one substitution in the original binary VT code, and since correcting one insertion is similar to correcting one deletion, we consider only the case of one deletion here and leave the remaining cases to the full version [5].

3.3 Correcting one deletion

Suppose that y is obtained from $x \in \mathcal{C}$ by deleting an a at position i. First, note that we can find a by computing $h_c(y) - h_c(x)$ for $c = 0,1,2$. Now, let $y^{(j)}$ denote the string obtained by inserting an a to the left of y_j (when $j = n$ this means we insert an a at the end of y). We have $x = y^{(i)}$ and our goal is to find i. Consider $n \ge j \ge i$ and observe that

$$f(x) - f(y^{(j)}) = f(y^{(i)}) - f(y^{(j)}) = \sum_{\ell=i+1}^{j} (w(x_\ell) - w(a)),$$

because $y_{\ell-1} = x_\ell$ for $\ell > i$. Since x is regular, it follows that $\sum_{\ell=i+1}^{j}(w(x_\ell) - w(a)) \neq 0$ unless $x_{i+1} = \cdots = x_j = a$. This suggests the following decoding algorithm: Successively compute $f(x) - f(y^{(j)})$ for $j = n, n-1, \dots, 1$ until $f(x) - f(y^{(j)}) = 0$, in which case the above argument ensures that $y^{(j)} = x$ since we must be inserting a into the same a-run of x from which an a was deleted. This procedure runs in overall time $O(n)$, since we can compute $f(x) - f(y^{(j-1)})$ given $f(x) - f(y^{(j)})$ with $O(1)$ operations.

3.4 A linear-time encoder

We have described a linear-time decoder that corrects a single edit error in regular strings x assuming knowledge of the weighted VT sketch $f(x)$ and the count sketches $h_c(x)$ for $c = 0, 1, 2$. It remains to describe a low-redundancy linear-time encoding procedure for a slightly modified version of our code $\mathcal{C}$ defined in (6). Fix an arbitrary message $z \in \{0, 1, 2, 3\}^m$. We proceed in two steps:

1. We encode z into a regular string $x \in \{0, 1, 2, 3\}^{m+4}$ in linear time by exploiting the runlength replacement technique from [15];
2. We append an appropriate encoding of the sketches (which we now see as binary strings) to x that can be recovered even if the final string is corrupted by an edit error. This adds only $O(\log \log n)$ bits of redundancy, and allows x (and thus z) to be recovered in linear time.

The complete analysis can be found in the full version [5].

4 Binary codes correcting one deletion or one transposition

In this section, we describe and analyze the code construction used to prove Theorem 2. As discussed in Section 1.2, the adjacent transposition precludes the use of the standard VT sketch. Therefore, we undertake a radically different approach.

4.1 Code construction and high-level overview of our approach

Our starting point is a marker-based segmentation approach considered by Lenz and Polyanskii [12] to correct bursts of deletions. We then introduce several new ideas. Roughly speaking, our idea is to partition a string $x \in \{0, 1\}^n$ into consecutive short substrings $z_1^x, \dots, z_\ell^x$ for some ℓ according to the occurrences of a special marker string in x. Then, by carefully embedding hashes of each segment z_i^x into a VT-type sketch, adding information about the *multiset* of hashes, and exploiting specific structural properties of deletions and adjacent transpositions, we are able to determine a short interval containing the position where the error occurred. Once this is done, a standard technique allows us to recover the true position of the error by slightly increasing the redundancy.

We now describe the code construction in detail. For a given integer $n > 0$, let $\Delta = 50 + 1000 \log n$ and $m = 1000\Delta^2 = O(\log^2 n)$. For the sake of readability, we have made no efforts to optimize constants, and assume n is a power of two to avoid using ceilings and floors. Given a string $x \in \{0, 1\}^n$, we divide it into substrings split according to occurrences of the marker 0011. To avoid edge cases, assume that x ends in 0011 – this will only add 4 bits to the overall redundancy. Then, this marker-based segmentation induces a vector $z^x = (z_1^x, \dots, z_{\ell_x}^x)$, where $1 \leq \ell_x \leq n$, and each string z_i^x has length at least 4, ends with 0011, and 0011 only occurs once in each such string. We may assume that $|z_i^x| \leq \Delta$ for all i. This will only add 1 bit to the overall redundancy, as captured in the following simple lemma.

▶ **Lemma 8.** *Suppose X is uniformly random over $\{0, 1\}^n$. Then,* $\Pr[|z_i^X| \leq \Delta, i = 1, \dots, \ell_X] \geq \frac{1}{2}$.

Our goal now will be to impose constraints on z^x so that (i) We only introduce $\log n + O(\log \log n)$ bits of redundancy, and (ii) If x is corrupted by a deletion or transposition in z_i^x, we can then locate a window $W \subseteq [n]$ of size $|W| = O(\log^4 n)$ such that $z_i^x \subseteq W$. This will then allow us to correct the error later on by adding $O(\log \log n)$ bits of redundancy.

Since each z_i^x has length at most $\Delta = O(\log n)$, we will exploit the fact that there exists a hash function h with short output that allows us to correct a deletion, substitution, or transposition in all strings of length at most 3Δ. This is guaranteed by the following lemma.

▶ **Lemma 9.** *There exists a hash function $h : \{0,1\}^{\leq 3\Delta} \to [m]$ with the following property:* If z' is obtained from z by at most two transpositions, two substitutions, or at most a deletion and an insertion, then $h(z) \neq h(z')$.

Proof. We can construct such a hash function h greedily. Let $A(z)$ denote the set of such strings obtained from $z \in \{0,1\}^{\leq 3\Delta}$. Since $|A(z)| < m$, we can set $h(z)$ so that $h(z) \neq h(z')$ for all $z' \in A(z) \setminus \{z\}$. ◀

With the intuition above and the hash function h guaranteed by Lemma 9 in mind, we consider the VT-type sketch

$$f(x) = \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot m + h(z_j^x)) \mod (L = 10n \cdot \Delta \cdot m + 1)$$

along with the count sketches $g_1(x) = \ell_x \mod 5$ and $g_2(x) = \sum_{i=1}^n \overline{x}_i \mod 3$, where $\overline{x}_i = \sum_{j=1}^i x_j \mod 2$. At a high level, the sketch $f(x)$ is the main tool we use to approximately locate the error in x. The count sketches $g_1(x)$ and $g_2(x)$ are added to allow us to detect how many markers are created or destroyed by the error, and to distinguish between the cases where there is no error or a transposition occurs. Thus, we define the preliminary code

$$\mathcal{C}' = \left\{ x \in \{0,1\}^n \;\middle|\; \begin{array}{l} x[n-3,n] = (0,0,1,1), f(x) = s_0, g_1(x) = s_1, g_2(x) = s_2, \\ \forall i \in [\ell_x] : |z_i^x| \leq \Delta \end{array} \right\}$$

for appropriate choices of s_0, s_1, s_2. Taking into account all constraints, the choice of Δ and m, and Lemma 8, the pigeonhole principle implies that we can choose s_0, s_1, s_2 so that this code has at most $4 + \log(10n \cdot \Delta \cdot m + 1) + 1 + 2 + 2 + 1 = \log n + O(\log\log n)$ bits of redundancy.

However, it turns out that the constraints imposed in $\mathcal{C}'$ are not enough to handle a deletion or a transposition. Intuitively, the reason for this is that, in order to make use of the sketch $f(x)$ when decoding, we will need additional information both about the hashes of the segments of x that were affected by the error and the hashes of the corresponding corrupted segments in the corrupted string y. Therefore, given a vector z^x and the hash function h guaranteed by Lemma 9, we will be interested in the associated *hash multiset* $H_x = \{\{h(z_1^x), \dots, h(z_{\ell_x}^x)\}\}$ over $[m]$. As we shall see, a deletion or transposition will change this multiset by at most 4 elements. Therefore, we will expurgate $\mathcal{C}'$ so that any pair of remaining codewords x and x' satisfy either $H_x = H_{x'}$ or $|H_x \triangle H_{x'}| \geq 10$. This will allow us to recover the true hash multiset of x from the hash multiset of the corrupted string. The following lemma shows that this expurgation adds only an extra $O(\log m) = O(\log\log n)$ bits of redundancy.

▶ **Lemma 10.** *There exists a code $\mathcal{C} \subseteq \mathcal{C}'$ of size $|\mathcal{C}| \geq \frac{|\mathcal{C}'|}{m^{10}}$ such that for any $x, x' \in \mathcal{C}$ we either have $H_x = H_{x'}$ or $|H_x \triangle H_{x'}| \geq 10$.*

We will take our *error-locating* code to be the expurgated code $\mathcal{C}$ guaranteed by Lemma 10. By the redundancy of $\mathcal{C}'$ above and the choice of m, it follows that there exists a choice of s_0 and s_1 such that $\mathcal{C}$ has $\log n + O(\log\log n)$ bits of redundancy. We prove the following result, which states that, given a corrupted version of $x \in \mathcal{C}$, we can identify a small interval containing the position where the error occurred.

▶ **Theorem 11.** *If $x \in \mathcal{C}$ is corrupted into y via one deletion or transposition, we can recover from y a window $W \subseteq [n]$ of size $|W| \leq 10^{10} \log^4 n$ that contains the position where the error occurred (in the case of a transposition, we take the error location to be the smallest of the two affected indices).*

We can use Theorem 11 to prove our main Theorem 2 via standard methods (see the full version [5]).

Fix $x \in \mathcal{C}$ and suppose y is obtained from x via one deletion or one transposition. To prove Theorem 11, we consider several independent cases based on the fact that a marker cannot overlap with itself, that we can identify whether a deletion occurred by computing $|y|$, and that we can identify whether a transposition occurred by comparing $g_2(x)$ and $g_2(y)$. Since the arguments are similar, we show how to locate one deletion and leave the case of one adjacent transposition to the full version [5].

4.2 Locating one deletion

In this section, we show how we can locate one deletion appropriately. Fix $x \in \mathcal{C}$ and suppose that a deletion is applied to z_i^x. The following lemma holds due to the marker structure.

▶ **Lemma 12.** *A deletion either (i) Creates a new marker and does not delete any existing markers, in which case $\ell_y = \ell_x + 1$, (ii) Deletes an existing marker and does not create any new markers, in which case $\ell_y = \ell_x - 1$, or (iii) Neither deletes existing markers nor creates new markers, in which case $\ell_y = \ell_x$.*

Note that we can distinguish between the cases detailed in Lemma 12 by comparing $g_1(x)$ and $g_1(y)$. Thus, we analyze each case separately:

1. $\ell_y = \ell_x$: In this case, we have $z^y = (z_1^x, \dots, z_{i-1}^x, z_i', z_{i+1}^x, \dots, z_{\ell_x}^x)$, where z_i' is obtained from z_i^x by a deletion (in particular, $|z_i'| = |z_i^x| - 1$). Therefore, it holds that

$$\begin{aligned} f(x) - f(y) &= \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot m + h(z_j^x)) - \sum_{j=1}^{\ell_y} j(|z_j^y| \cdot m + h(z_j^y)) \mod L \\ &= i(|z_i^x| \cdot m + h(z_i^x) - |z_i'| \cdot m - h(z_i')) \\ &= i(m + h(z_i^x) - h(z_i')), \end{aligned}$$

where the second equality uses $\ell_y = \ell_x$. Let H_y denote the hash multiset of y. Then, we know that $|H_x \triangle H_y| \leq 2$. Therefore, we can recover H_x from H_y, which means that we can recover $h(z_i^x) - h(z_i')$. Indeed, if $h(z_i^x) - h(z_i') = 0$ then $H_x = H_y$. On the other hand, if $h(z_i^x) - h(z_i') \neq 0$ then $|H_x \triangle H_y| = 2$ and we recover both $h(z_i^x)$ (the element in H_x but not in H_y) and $h(z_i')$ (the element in H_y but not in H_x). As a result, we know $m + h(z_i^x) - h(z_i')$. Since it also holds that $m + h(z_i^x) - h(z_i') \neq 0$ (because $|h(z_i^x) - h(z_i')| < m$), we can recover i from $f(x) - f(y)$. This gives a window W of length at most $\Delta = O(\log n)$.

2. $\ell_y = \ell_x - 1$: In this case, the marker at the end of z_i^x is destroyed, merging z_i^x and z_{i+1}^x. Observe that if $i = \ell_x$ then we can simply detect that the last marker in x was destroyed. Therefore, we assume that $i < \ell_x$, in which case we have $z^y = (z_1^x, \dots, z_{i-1}^x, z_i', z_{i+2}^x, \dots, z_{\ell_x}^x)$, where $|z_i'| = |z_i^x| + |z_{i+1}^x| - 1$. Consequently, it holds that

$$
\begin{aligned}
f(x)-f(y) &= \sum_{j=1}^{\ell_x} j(|z_j^x|\cdot m + h(z_j^x)) - \sum_{j=1}^{\ell_y} j(|z_j^y|\cdot m + h(z_j^y)) \mod L \\
&= i(|z_i^x|\cdot m + h(z_i^x)) + (i+1)(|z_{i+1}^x|\cdot m + h(z_{i+1}^x)) - i(|z_i'|\cdot m + h(z_i')) \\
&+ \sum_{j=i+2}^{\ell_x} (|z_j^x|\cdot m + h(z_j^x)) \\
&= \sum_{j=i+2}^{\ell_x} (|z_j^x|\cdot m + h(z_j^x)) + i(m + h(z_i^x) + h(z_{i+1}^x) - h(z_i')) \\
&+ (|z_{i+1}^x|\cdot m + h(z_{i+1}^x)).
\end{aligned}
$$

Note that, since $|H_x \triangle H(y)| \leq 3$, we can recover H_x from H_y. In particular, this means that we know $h(z_i^x) + h(z_{i+1}^x) - h(z_i')$. Therefore, for $i' = \ell_y - 1, \ell_y - 2, \ldots, i$ we can compute the *potential function*

$$
\begin{aligned}
\Phi(i') &= \sum_{j=i'+1}^{\ell_y} (|z_j^y|\cdot m + h(z_j^y)) + i'(m + h(z_i^x) + h(z_{i+1}^x) - h(z_i')) \\
&= \sum_{j=i'+2}^{\ell_x} (|z_j^x|\cdot m + h(z_j^x)) + i'(m + h(z_i^x) + h(z_{i+1}^x) - h(z_i')).
\end{aligned}
$$

Note that

$$
|\Phi(i) - (f(x) - f(y))| = ||z_{i+1}^x|\cdot m + h(z_{i+1}^x)| \leq \Delta \cdot m + m \leq 10^7 \log^2 n. \tag{7}
$$

Moreover, we also have

$$
\begin{aligned}
\Phi(i'-1) - \Phi(i') = |z_{i'+1}^x|\cdot m + h(z_{i'+1}^x) - (m + h(z_i^x) + h(z_{i+1}^x) - h(z_i')) \\
\geq 4m - 3m = m.
\end{aligned} \tag{8}
$$

This suggests the following procedure for recovering the window W. Sequentially compute $\Phi(i')$ for i' starting at $\ell_y - 1$ until we find $i^\star \geq i$ such that $|\Phi(i') - (f(x) - f(y))| \leq 10^6 \log^2 n$. This is guaranteed to exist since $i' = i$ satisfies this property. We claim that $i^\star - i \leq 10^7 \log n$. In fact, if this is not the case then the monotonicity property in (8) implies that $|\Phi(i) - (f(x) - f(y))| > m \cdot 10^7 \log n > 10^7 \log^2 n$, contradicting (7). Since $|z_j^x| \leq \Delta$ for every j, recovering $i^\star$ also yields a window $W \subseteq [n]$ of size $|W| = 10^6 \log n \cdot \Delta = 10^9 \log^2 n$ containing the error position, as desired.

3. $\ell_y = \ell_x + 1$: This case is very similar to the previous one (see the full version [5]).

5 Binary list-size two code for one deletion and one substitution

In this section, we describe and analyze a binary list-size two decodable code for one deletion and one substitution, which yields Theorem 3. Departing from the approach of [19], our construction makes use of *run-based sketches* combined with the standard VT sketch. Run-based sketches have thus far been exploited in the construction of multiple-deletion correcting codes, including list-decodable codes with small list size [9].

5.1 Code construction

We begin by describing some required concepts: Given a string $x = (x_1, \dots, x_n) \in \{0,1\}^n$, we define its *run string* r^x by first setting $r^x_0 = 0$ along with $x_0 = 0$ and $x_{n+1} = 1$, and then iteratively computing $r^x_i = r^x_{i-1}$ if $x_i = x_{i-1}$ and $r^x_i = r^x_{i-1}+1$ otherwise for $i = 1, \dots, n, n+1$. Note that every string x is uniquely determined by its run string r^x and vice-versa. Moreover, it holds that r^x defines a non-decreasing sequence and $0 \leq r^x_i \leq i$ for every $i = 1, \dots, n, n+1$. As an example, the run string corresponding to $x = 011101000$ is $r^x = 0111234445$. We call r^x_i the *rank* of index i in x. We will denote the total number of runs in x by $r(x)$.

The main component of our code is a combination of the standard VT sketch

$$f(x) = \sum_{i=1}^{n} i x_i \mod (3n+1) \tag{9}$$

with the run-based sketches

$$f^r_1(x) = \sum_{i=1}^{n} r^x_i \mod (12n+1), \tag{10}$$

$$f^r_2(x) = \sum_{i=1}^{n} r^x_i (r^x_i - 1) \mod (16n^2+1) \tag{11}$$

originally considered in [9]. Additionally, we also consider the count sketches

$$h(x) = \sum_{i=1}^{n} x_i \mod 5 \quad \text{and} \quad h_r(x) = r(x) \mod 13. \tag{12}$$

Intuitively, the count sketches are used to distinguish different error patterns. The sketch $h(x)$ is used to determine the value of the bit deleted and the value of the bit flipped, while $h_r(x)$ is used to identify how the number of runs was affected by the errors. For each possible error pattern, we use the standard VT-sketch and the run-based sketches to decode. Given the above, our code is defined to be

$$\mathcal{C} = \{x \in \{0,1\}^n : f(x) = s, f^r_1(x) = s^r_1, f^r_2(x) = s^r_2, h(x) = u, h_r(x) = u_r\}, \tag{13}$$

for an appropriate choice of $s \in [3n+1]$, $s^r_1 \in [12n+1]$, $s^r_2 \in [16n^2+1]$, $u \in [5]$, and $u_r \in [13]$. By the pigeonhole principle, there is such a choice which ensures $\mathcal{C}$ has redundancy $4\log n + O(1)$.

In the remainder of this section, we first provide a high-level overview of our approach towards showing that $\mathcal{C}$ admits linear-time list-decoding from one deletion and one substitution with list-size 2. Then, we analyze a special case which exemplifies our more general approach. The remainder of our argument appears in the full version [5]. We remark that linear-time decoding and encoding of a slightly modified version of $\mathcal{C}$ (which has redundancy $4\log n + O(\log\log n)$ instead) follow without difficulty from this analysis via standard methods. These algorithms are presented and analyzed in the full version [5].

5.2 High-level overview of our approach

Fix $x \in \mathcal{C}$, and let y be the string obtained from x after one deletion at index d and one substitution at index e. We use x_e to denote the bit flipped, and x_d to denote the bit deleted in x. When $d = e$, we have one deletion and no substitution. Our goal is to recover x from y.

We begin with some simple but useful remarks. First, we observe that one deletion and no substitution can be equivalently transformed to one deletion and one substitution. Thus, we will only consider the case in which we have one deletion and one substitution, i.e., $d \neq e$. We present a proof of this fact in the full version [5]. Second, the following structural lemma about the number of runs in a corrupted string will prove useful in our case analysis.

▶ **Lemma 13.** *If x' is obtained from x via one deletion, then either $r(x') = r(x)$ or $r(x') = r(x) - 2$. On the other hand, if x' is obtained from x via one substitution, then either $r(x') = r(x)$, $r(x') = r(x) - 2$, or $r(x') = r(x) + 2$.*

Combining Lemma 13 with the count sketches $h(x)$ and $h_r(x)$ and knowledge of y ensures that we can identify not only the values of x_d and x_e, but also $r(x)$. As a result, this allows us to split our analysis into several independent cases.

The process of decoding can be thought of as inserting a bit x_d before the d-th bit in y and flipping the $(e - \delta)$-th bit in y, where $\delta \in \{0, 1\}$ is the indicator variable of whether $e > d$. Our goal is to find d and e. We will begin with a candidate position pair $(\widetilde{d}, \widetilde{e})$ with $\widetilde{d}$ is as small as possible with the property that, if $\widetilde{x}$ denotes the string obtained from y by inserting x_d before the d-th bit in y and flipping the bit at position $\widetilde{e} - \widetilde{\delta}$ in y, where $\widetilde{\delta}$ indicates whether $\widetilde{d} < \widetilde{e}$, then $f(\widetilde{x}) = f(x)$, $h_r(\widetilde{x}) = h_r(x)$, and $h_r(x') = h_r(\widetilde{x}')$, where x' (resp. $\widetilde{x}'$) denotes the string obtained from x (resp. $\widetilde{x}$) by deleting x_d (resp. $\widetilde{x}_{\widetilde{d}}$). We call such pairs *valid*. Intuitively, valid pairs are indistinguishable from the true error pattern (d, e) from the perspective of the VT sketch and the count sketches, and there may be several of them. However, crucially, many are ruled out via the run-based sketches. Note that the true error pattern (d, e) is a valid pair, so such pairs always exist.

Roughly speaking, our strategy is to start with some valid pair $(\widetilde{d}, \widetilde{e})$ and sequentially move to the *next* valid pair. This is done by moving $\widetilde{d}$ one index to the right and checking whether the *unique* index $\widetilde{e}$ that ensures $f(\widetilde{x}) = f(x)$ forms a valid pair $(\widetilde{d}, \widetilde{e})$. If this does not hold, then we move $\widetilde{d}$ one more index to the right, and repeat the process. We call this an *elementary move*. Note that since inserting a bit b into a b-run at any position gives the same output, we may always move $\widetilde{d}$ to the end of the next x_d-run in y (which may be empty). Figure 1 shows an example of an elementary move.

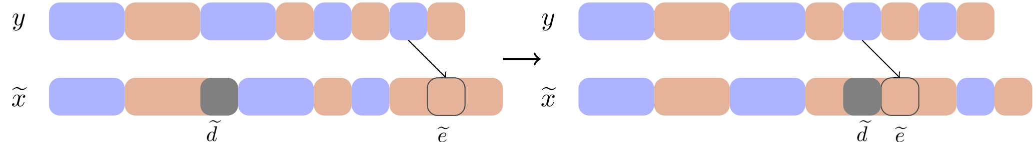

Figure 1 Example of an elementary move. Suppose that the error pattern indicates that $x_d = 1$, $x_e = 1$, and the deletion does not reduce the number of runs while the substitution increases the number of runs by two. The process starts with the left figure in which a bit 1 is inserted at position $\widetilde{d}$, the end of a 1-run and the bit 1 at position $\widetilde{e} - 1$ is flipped. After an elementary move, $\widetilde{d}$ moves to the end of the next 1-run, and e moves to the next position that matches the error pattern $y_{\tilde{e}-\tilde{\delta}+1} = y_{\tilde{e}-\tilde{\delta}-1} = 0$.

Considering this step-by-step process with elementary moves proves useful because it turns out to be feasible to track how the different sketches change in each such move. In particular, the following equations will be useful to determine how $\widetilde{d}$ and $\widetilde{e}$ change in each elementary move. Recall that we regard y as a string obtained via one substitution at index $e - \delta$ from $x' \in \{0, 1\}^{n-1}$, where x' is obtained via one deletion from x at index d. Note that

$$f(x) - f(x') = dx_d + \sum_{d}^{n-1} x'_i \quad \text{and} \quad f(x') - f(y) = (e - \delta)[x_e - (1 - x_e)].$$

Moreover, we have $\sum_d^{n-1} x'_i = \sum_d^{n-1} y_i + \delta(2x_e - 1)$. Combining these three observations yields

$$f(x) - f(y) = dx_d + \sum_{i=d}^{n-1} y_i + e(2x_e - 1). \tag{14}$$

We prove that, during this sequential process, either f_1^r is monotonic and hence rules out all but one valid pair $(\widetilde{d}, \widetilde{e})$, or a convexity-type property of f_2^r, which implies that it takes on each value at most twice, rules out all but at most two valid pairs. The convexity of $f_2^r(x)$ is a consequence of the following lemma.

▶ **Lemma 14** ([9, Lemma 4.1]). *Let a_i and a'_i be two sequences of non-negative integers such that $\sum_{i=1}^n a_i = \sum_{i=1}^n a'_i$ and there is a value t such that for all i satisfying $a_i < a'_i$ it holds that $a'_i \leq t$, and for all i satisfying $a_i > a'_i$ it holds that $a'_i \geq t$. Then, either $a_i = a'_i$ for all i, or $\sum_{i=1}^n a_i(a_i - 1) > \sum_{i=1}^n a'_i(a'_i - 1)$.*

Finally, we note that, in the high level overview above, we ignored the fact that we do not have access to the intermediate string x', but we need to know $h_r(x')$. For example, if $r(y) = r(x)$, then there is uncertainty about $h_r(x')$. In fact, it could be that both errors do not change the number of runs, or that both errors do change the number of runs but these cancel each other out. Since we are aiming for list-size 2 decoding, this is not problematic, and we handle it in the final decoding procedure.

Below, we consider one special case which exemplifies how our high level approach above can be realized. The remaining cases are analyzed in the full version [5].

5.3 Special case – Unique decoding when the number of runs increases by two

If $r(y) = r(x) + 2$, then it must be that $r(x) = r(x')$ and $r(y) = r(x') + 2$. This means that the deletion does not change the number of runs (and thus occurred in a run of length at least 2 in x), while the substitution affects a bit in the middle of a run of length at least 3. In particular, we have $y_{e-\delta-1} = y_{e-\delta+1} = 1 - y_{e-\delta}$. In this case, it follows that

$$f_1^r(x) - f_1^r(x') = r_d^x, \qquad f_1^r(x') - f_1^r(y) = -(1 + 2(n - e + \delta)).$$

Therefore, for the run-based sketch $f_1^r(x)$ it holds that

$$f_1^r(x) - f_1^r(y) = r_d^x - (1 + 2(n - e + \delta)). \tag{15}$$

We now proceed by case analysis on the value of x_d and x_e.

5.3.1 If $x_e = x_d = b$

In this case, when $\widetilde{d}$ makes an elementary move to the right, it must pass across a $(1 - b)$-run of some length $\ell \geq 1$. According to (14), position $\widetilde{e}$ has to move to the left by ℓ so that $f(\widetilde{x}) = f(x)$. If we have $\widetilde{d} < \widetilde{e}$ before one elementary move but $\widetilde{d} > \widetilde{e}$ after that move, we call it a *take over step*. For each elementary move:

- If the move is not a take over step: Then, $r^{\widetilde{x}}_{\widetilde{d}}$ increases by 2 while $2(n-\widetilde{d}+\widetilde{\delta})+1$ increases by 2ℓ. Therefore, (15) implies that $f_1^r(\widetilde{x})$ strictly decreases after such a move whenever $\ell > 1$. If $\ell = 1$, then $\widetilde{e}$ moves by 1 to the left and $f_1^r(\widetilde{x})$ remains unchanged. However, since we need $1-b = y_{\widetilde{e}-\widetilde{\delta}} = 1 - y_{\widetilde{e}-\widetilde{\delta}}$ it follows that $\widetilde{e}$ cannot move only 1 position to the left, and so $\ell > 1$ necessarily.
- If the move is a take over step: Before the move, $\widetilde{d}$ is on the left of a $(1-b)$-run of length $\ell \geq 1$ while $\widetilde{e} > \widetilde{d}$ satisfies $y_{\widetilde{e}-1} = 1-b$ and $y_{\widetilde{e}-2} = y_{\widetilde{e}} = b$. After the move, $\widetilde{d}$ moves to the right of the $(1-b)$-run of length ℓ, while $\widetilde{e}$ is to the left of $\widetilde{d}$. Moreover, it must be that $y_{\widetilde{e}} = 1-b$ and $y_{\widetilde{e}-1} = y_{\widetilde{e}+1} = b$. To match the error pattern, the only possible case is that $\ell = 1$. To see why this is the case, note that when $\ell \geq 2$ the index $\widetilde{e}$ has to move to the left by at least $\ell + 2$ to match the error pattern $y_{\widetilde{e}-\widetilde{\delta}-1} = y_{\widetilde{e}-\widetilde{\delta}+1} = 1 - y_{\widetilde{e}-\widetilde{\delta}}$. However, this move leads to $f(\widetilde{x}) \neq f(x)$, and thus does not yield a valid pair $(\widetilde{d}, \widetilde{e})$. When $\ell = 1$, let $(\widetilde{d}_1, \widetilde{e}_1)$ and $(\widetilde{d}_2, \widetilde{e}_2)$ denote the position pair before and after the move, respectively. Then, these two pairs yield the same candidate solution $\widetilde{x}_1 = \widetilde{x}_2$. See Figure 2 for an example.

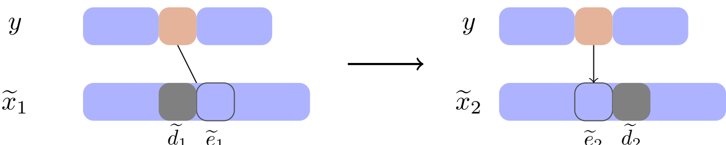

Figure 2 An example of a take over step. If the take over happens, it must be that $\ell = 1$. The resulting $\widetilde{x}_1$ and $\widetilde{x}_2$ are the same.

Taking into account both cases above, we see that $f_1^r(\widetilde{x})$ decreases during each elementary move, and decreases by at most $2n$ during the whole process. Since the value of $f_1^r(x)$ is taken modulo $12n+1$, there is only a unique pair $(\widetilde{d}, \widetilde{e})$ that yields a solution such that $f_1^r(\widetilde{x}) = f_1^r(x)$. Hence, $f(x)$ and $f_1^r(x)$ together with y uniquely determine one valid pair $(\widetilde{d}, \widetilde{e})$, which in turn yields a unique candidate solution $\widetilde{x} = x$.

5.3.2 If $x_d = 1 - x_e = b$

In this case, when $\widetilde{d}$ makes an elementary move to the right, it must pass across a $(1-b)$-run of some length $\ell \geq 1$. Then, $\widetilde{e}$ has to move to the right by ℓ so that $f(\widetilde{x}) = f(x)$. During each such move $f_1^r(\widetilde{x})$ strictly increases. For the whole process, $f_1^r(\widetilde{x})$ increases by at most $2n$. By a similar argument as above, we have that $f(x)$ and $f_1^r(x)$ together with y uniquely determine one valid pair $(\widetilde{d}, \widetilde{e})$ which yields the correct solution $\widetilde{x} = x$.

6 Open problems

Our work leaves open several natural avenues for future research. We highlight a few of them here:

- Given the effectiveness of weighted VT sketches in the construction of nearly optimal non-binary single-edit correcting codes in Section 3 with fast encoding and decoding, it would be interesting to find further applications of this notion.
- We believe that the code we introduce and analyze in Section 5 is actually uniquely decodable under one deletion and one substitution. Proving this would be quite interesting, since then we would also have explicit uniquely decodable single-deletion single-substitution correcting codes with redundancy matching the existential bound, analogous to what is known for two-deletion correcting codes [9].

- The code we designed in Section 4 fails to correct an arbitrary substitution. Roughly speaking, the reason behind this is that one substitution may simultaneously destroy and create a marker with a different starting point. As the clear next step, it would be interesting to show the existence of a binary code correcting one *edit* error or one transposition with redundancy $\log n + O(\log\log n)$.

References

1 Joshua Brakensiek, Venkatesan Guruswami, and Samuel Zbarsky. Efficient low-redundancy codes for correcting multiple deletions. *IEEE Transactions on Information Theory*, 64(5):3403–3410, 2018. doi:10.1109/TIT.2017.2746566.

2 Kui Cai, Yeow Meng Chee, Ryan Gabrys, Han Mao Kiah, and Tuan Thanh Nguyen. Correcting a single indel/edit for DNA-based data storage: Linear-time encoders and order-optimality. *IEEE Transactions on Information Theory*, 67(6):3438–3451, 2021. doi:10.1109/TIT.2021.3049627.

3 Kuan Cheng, Zhengzhong Jin, Xin Li, and Ke Wu. Deterministic document exchange protocols, and almost optimal binary codes for edit errors. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 200–211, 2018. doi:10.1109/FOCS.2018.00028.

4 Kuan Cheng, Zhengzhong Jin, Xin Li, and Ke Wu. Block edit errors with transpositions: Deterministic document exchange protocols and almost optimal binary codes. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 37:1–37:15, 2019. doi:10.4230/LIPIcs.ICALP.2019.37.

5 Ryan Gabrys, Venkatesan Guruswami, João Ribeiro, and Ke Wu. Beyond single-deletion correcting codes: Substitutions and transpositions. *arXiv e-prints*, December 2021. doi:10.48550/arXiv.2112.09971.

6 Ryan Gabrys and Frederic Sala. Codes correcting two deletions. *IEEE Transactions on Information Theory*, 65(2):965–974, 2019. doi:10.1109/TIT.2018.2876281.

7 Ryan Gabrys, Eitan Yaakobi, and Olgica Milenkovic. Codes in the Damerau distance for deletion and adjacent transposition correction. *IEEE Transactions on Information Theory*, 64(4):2550–2570, 2018. doi:10.1109/TIT.2017.2778143.

8 Venkatesan Guruswami, Bernhard Haeupler, and Amirbehshad Shahrasbi. Optimally resilient codes for list-decoding from insertions and deletions. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 524–537, 2020. doi:10.1145/3357713.3384262.

9 Venkatesan Guruswami and Johan Håstad. Explicit two-deletion codes with redundancy matching the existential bound. *IEEE Transactions on Information Theory*, 67(10):6384–6394, 2021. doi:10.1109/TIT.2021.3069446.

10 Bernhard Haeupler. Optimal document exchange and new codes for insertions and deletions. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 334–347, 2019. doi:10.1109/FOCS.2019.00029.

11 Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization strings: Explicit constructions, local decoding, and applications. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 841–854, 2018. doi:10.1145/3188745.3188940.

12 Andreas Lenz and Nikita Polyanskii. Optimal codes correcting a burst of deletions of variable length. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 757–762, 2020. doi:10.1109/ISIT44484.2020.9174288.

13 Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk*, 163(4):845–848, 1965.

14 Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, et al. Random access in large-scale DNA data storage. *Nature biotechnology*, 36(3):242, 2018. doi:10.1038/nbt.4079.

15 Clayton Schoeny, Antonia Wachter-Zeh, Ryan Gabrys, and Eitan Yaakobi. Codes correcting a burst of deletions or insertions. *IEEE Transactions on Information Theory*, 63(4):1971–1985, 2017. doi:10.1109/TIT.2017.2661747.

16 Leonard J. Schulman and David Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory*, 45(7):2552–2557, 1999. doi:10.1109/18.796406.

17 Jin Sima and Jehoshua Bruck. On optimal k-deletion correcting codes. *IEEE Transactions on Information Theory*, 67(6):3360–3375, 2021. doi:10.1109/TIT.2020.3028702.

18 Neil J. A. Sloane. On single-deletion-correcting codes. *arXiv*, 2002. doi:10.48550/arXiv.math/0207197.

19 Ilia Smagloy, Lorenz Welter, Antonia Wachter-Zeh, and Eitan Yaakobi. Single-deletion single-substitution correcting codes. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 775–780, 2020. doi:10.1109/ISIT44484.2020.9174213.

20 Wentu Song, Kui Cai, and Tuan Thanh Nguyen. List-decodable codes for single-deletion single-substitution with list-size two, January 2022. doi:10.48550/arXiv.2201.02013.

21 Wentu Song, Nikita Polyanskii, Kui Cai, and Xuan He. On multiple-deletion multiple-substitution correcting codes. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2655–2660, 2021. doi:10.1109/ISIT45174.2021.9517878.

22 Daniel Tan. Implementation of single-edit correcting code, 2020. URL: https://github.com/dtch1997/single-edit-correcting-code.

23 Yuanyuan Tang and Farzad Farnoud. Error-correcting codes for short tandem duplication and edit errors. *IEEE Transactions on Information Theory*, 68(2):871–880, 2022. doi:10.1109/TIT.2021.3125724.

24 Rom R. Varshamov and Grigory M. Tenengolts. Codes which correct single asymmetric errors. *Autom. Remote Control*, 26(2):286–290, 1965.

25 Antonia Wachter-Zeh. List decoding of insertions and deletions. *IEEE Transactions on Information Theory*, 64(9):6297–6304, 2018. doi:10.1109/TIT.2017.2777471.

26 Shuche Wang, Jin Sima, and Farzad Farnoud. Non-binary codes for correcting a burst of at most 2 deletions. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2804–2809, 2021. doi:10.1109/ISIT45174.2021.9517917.

27 S. M. Hossein Tabatabaei Yazdi, Ryan Gabrys, and Olgica Milenkovic. Portable and error-free DNA-based data storage. *Scientific reports*, 7(1):5011, 2017. doi:10.1038/s41598-017-05188-1.