Electronic Colloquium on Computational Complexity, Report No. 116 (2015)

Efficient Low-Redundancy Codes for Correcting Multiple Deletions

Joshua Brakensiek* Venkatesan Guruswami† Samuel Zbarsky‡

Carnegie Mellon University

Abstract

We consider the problem of constructing binary codes to recover from k–bit deletions with efficient encoding/decoding, for a fixed k. The single deletion case is well understood, with the Varshamov-Tenengolts-Levenshtein code from 1965 giving an asymptotically optimal construction with $\approx 2^n/n$ codewords of length n, i.e., at most $\log n$ bits of redundancy. However, even for the case of two deletions, there was no known explicit construction with redundancy less than $n^{\Omega(1)}$.

For any fixed k, we construct a binary code with $c_k \log n$ redundancy that can be decoded from k deletions in $O_k(n \log^4 n)$ time. The coefficient c_k can be taken to be $O(k^2 \log k)$, which is only quadratically worse than the optimal, non-constructive bound of $O(k)$. We also indicate how to modify this code to allow for a combination of up to k insertions and deletions.

We also note that among *linear* codes capable of correcting k deletions, the $(k+1)$-fold repetition code is essentially the best possible.

1 Introduction

A k-bit binary deletion code of length N is some set of strings $C \subseteq \{0,1\}^N$ so that for any $c_1, c_2 \in C$, the longest common subsequence of c_1 and c_2 has length less than $N-k$. For such a code, a codeword of C can be uniquely identified from any of its subsequences of length $N-k$, and therefore such a code enables recovery from k adversarial/worst-case deletions.

In this work, we are interested in the regime when k is a fixed constant, and the block length N grows. Denoting by $\mathsf{del}(N,k)$ the size of the largest k-bit binary deletion code of length N, it is known that

$$a_k \frac{2^N}{N^{2k}} \leqslant \mathsf{del}(N,k) \leqslant A_k \frac{2^N}{N^k} \tag{1}$$

for some constants $a_k > 0$ and $A_k < \infty$ depending only k [1].

For the special case of $k = 1$, it is known that $\mathsf{del}(N,1) = \Theta(2^N/N)$. The Varshamov-Tenengolts code [2] defined by

$$\left\{(x_1,\ldots,x_N) \in \{0,1\}^N \mid \sum_{i=1}^{N} i x_i \equiv 0 \pmod{(N+1)}\right\} \tag{2}$$

[*] jbrakens@andrew.cmu.edu. Research supported in part by an REU supplement to NSF CCF-0963975.

[†] guruswami@cmu.edu. Research supported in part by NSF grants CCF-0963975 and CCF-1422045.

[‡] szbarsky@andrew.cmu.edu

ISSN 1433-8092

is known to have size at least $2^N/(N+1)$, and Levenshtein [1] shows that this code is capable of correcting a single deletion. An easy to read exposition of the deletion correcting property of the VT code can be found in the detailed survey on single-deletion-correcting codes [3].

The bound (1) shows that the asymptotic number of redundant bits needed for correcting k-bit deletions in an N-bit codeword is $\Theta(k \log N)$ (i.e., one can encode $n = N - \Theta(k \log N)$ message bits into length N codewords in a manner resilient to k deletions). Note that the codes underlying this result are obtained by an exponential time greedy search — an efficient construction of k-bit binary deletion codes with redundancy approaching $O(k \log N)$ was not known, except in the single-deletion case where the VT code gives a solution with optimal $\log N + O(1)$ redundancy.

The simplest code to correct k worst-case deletions is the $(k+1)$-fold repetition code, which maps n message bits to $N = (k+1)n$ codewords bits, and thus has $\frac{k}{k+1}N$ redundant bits. A generalization of the VT code for the case of multiple deletions was proposed in [4] and later proved to work in [5]. These codes replace the weight i given to the i'th codeword bit in the check constraint of the VT code (2) by a much larger weight, which even for the $k = 2$ case is related to the Fibonacci sequence and thus grows exponentially in i. Therefore, the redundancy of these codes is $\Omega(N)$ even for two deletions, and equals $c_k N$ where the constant $c_k \to 1$ as k increases. Some improvements were made for small k in [6], which studied run-length limited codes for correcting insertions/deletions, but the redundancy remained $\Omega(N)$ even for two deletions.

Allowing for $\Theta(N)$ redundancy, one can in fact efficiently correct a constant *fraction* of deletions, as was shown by Schulman and Zuckerman [7]. This construction was improved and optimized recently in [8], where it was shown that one could correct a fraction $\zeta > 0$ of deletions with $O(\sqrt{\zeta}N)$ redundant bits in the encoding. One can deduce codes to correct a constant k number of deletions with redundancy $O_k(\sqrt{N})$ using the methods of [8] (we will hint at this in Section 1.2).

In summary, despite being such a natural and basic problem, there were no known explicit codes with redundancy better than $\sqrt{N}$ even to correct from two deletions. Our main result, stated formally as Theorem 1 below, gives an explicit construction with redundancy $O_k(\log N)$ for any fixed number k of deletions, along with a near-linear time decoding algorithm.

For simplicity, the above discussion focused on the problem of recovering from deletions alone. One might want codes to recover from a combination of deletions and insertions (i.e., errors under the edit distance metric). Levenshtein [1] showed that any code capable of correcting k deletions is in fact also capable of correcting from any combination of a total of k insertions and deletions. But this only concerns the combinatorial property underlying correction from insertions/deletions, and does not automatically yield an algorithm to recover from insertions/deletions based on a deletion-correcting algorithm. For our main result, we are able to extend our construction to efficiently recover from an arbitrary combination of worst-case insertions/deletions as long as their total number is at most k.

1.1 Our result

In this work, we construct, for each fixed k, a binary code of block length N for correcting k insertions/deletions on which all relevant operations can be done in polynomial (in fact, near-linear) time and that has $O(k^2 \log k \log N)$ redundancy. We stress that this is the first efficient construction with redundancy smaller than $N^{\Theta(1)}$ even for the 2-bit deletion case. For simplicity of exposition, we go through the details on how to construct an efficient deletion code, and then indicate how to modify it to turn it into an efficient deletion/insertion code.

Theorem 1 (Main)**.** *Fix an integer $k \geqslant 2$, For all sufficiently large n, there exists a code length $N \leqslant n +$*

$O(k^2 \log k \log n)$, an injective encoding map $\mathsf{Enc} : \{0,1\}^n \to \{0,1\}^N$ and a decoding map $\mathsf{Dec} : \{0,1\}^{N-k} \to \{0,1\}^n \cup \{Fail\}$ both computable in $O_k(n(\log n)^4)$ time, such that for all $s \in \{0,1\}^n$ and every subsequence $s' \in \{0,1\}^{N-k}$ obtained from $\mathsf{Enc}(s)$ by deleting k bits, $\mathsf{Dec}(s') = s$.

Note that the decoding complexity in the above result has a FPT (fixed-parameter tractable) type dependence on k, and a near-linear dependence on n.

Our encoding function in Theorem 1 is non-linear. This is inherent; in Appendix A we give a simple proof that among *linear* codes capable of correcting k deletions, the $(k+1)$-fold repetition code is essentially the best possible.

1.2 Our approach

We describe at a high level the ideas behind our construction of k-bit binary deletion codes with logarithmic redundancy. The difficulty with the deletion channel is that we don't know the location of deletions, and thus we lose knowledge of which position a bit corresponds to. Towards identifying positions of some bits in spite of the deletions, we can break a codeword into blocks $a_1, a_2, \ldots, a_m$ of length b bits each, and separate them by introducing dummy buffers (consisting of a long enough run of 0's, say). If only k bits are deleted, by looking for these buffers in the received subsequence, we can identify all but $O(k)$ of the blocks correctly (there are some details one must get right to achieve this, but these are not difficult). If the blocks are protected against $O(k)$ errors, then we can recover the codeword. In terms of redundancy, one needs at least m bits for the buffers, and at least $\Omega(kb) \geqslant b$ bits to correct the errors in the blocks. As $mb = n$, such a scheme needs at least $\Omega(\sqrt{n})$ redundant bits. Using this approach, one can in fact achieve $\approx \sqrt{kn}$ redundancy; this is implicit in [8].

To get lower redundancy, our approach departs from the introduction of explicit buffers, as they use up too many redundant bits. Our key idea is to use patterns that occur frequently in the string *themselves* as "implicit" buffers, so we have no redundancy wasted for introducing buffers. Since an adversary could delete a portion of an implicit buffer thereby foiling our approach, we use multiple implicit patterns and form a separate "hash" for each pattern (which will protect the intervening blocks against k errors). Since many strings have very few short patterns (such as as the all 0's string), we first use a *pattern enriching* encoding procedure to ensure that there are sufficiently many patterns. The number of implicit patterns is enough so that less than half of them can be corrupted by an adversary in any choice of k deletions. Then, we can decode the string using each pattern and take the majority vote of the resulting decodings. The final transmitted string bundles the *pattern rich* string, a hash describing the pattern enriching procedure, and the hash for each pattern. The two hashes are protected with a less efficient k-bit deletion code (with $o(n)$ redundancy) so that we can immediately decode them.

1.3 Deletion codes and synchronization protocols

A related problem to correcting under edit distance is the problem of synchronizing two strings that are nearby in edit distance or *document exchange* [9]. The model here is that Alice holds a string $x \in \{0,1\}^n$ and Bob holds an arbitrary string y at edit distance at most k from x — for simplicity let us consider the deletions only case so that $y \in \{0,1\}^{n-k}$ is a subsequence of x. The existential result for deletion codes implies that there is a short message $g(x) \in \{0,1\}^{O(k \log n)}$ that Alice can send to Bob, which together with y enables him to recover x (this is also a special case of a more general communication problem considered in [10]). However, the function g takes exponential time to compute. We note that if we had an efficient to compute g with output length $O(k\log n)$, then one can also get deletion codes with small redundancy by protecting $g(x)$ with a deletion code (that is shorter and therefore easier to construct). Indeed, this is in effect what our approach outline above does, but only when x is a pattern rich string. Our methods don't yield a deterministic protocol for this problem when x is arbitrary, and constructing such a protocol with $n^{o(1)}$ communication remains open.

If we allow randomization, sending a random hash value $h(x)$ of $O(k\log n)$ bits will allow Bob to correctly identify x among all possible supersequences of y; however, this will take $n^{O(k)}$ time. Randomized protocols that enable Bob to efficiently recover x in near-linear time are known, but these require larger hashes of size $O(k\log^2 n\log^* n)$ [11] or $O(k\log(n/k)\log n)$ [12]. Very recently, a randomized protocol with a $O(k^2\log n)$ bound on the number of bits transmitted was given in [13]. But the use of randomness makes these synchronization protocols unsuitable for the application to deletion codes in the adversarial model.

1.4 Organization

In Section 2, we define the notation and describe some simple or well-known codes which will be used throughout the paper. Section 3 demonstrates how to efficiently encode and decode pattern rich strings against k-bit deletions using a hashing procedure. Section 4 describes how to efficiently encode any string as a pattern rich string. Section 5 combines the results of the previous sections to prove Theorem 1. Section 6 describes how to modify the code so that it works efficiently on the k-bit insertion and deletion channel. Section 7 suggests what would need to be done to improve redundancy past $O(k^2\log k\log n)$ using our methods. Appendix A proves that essentially the best linear k-bit deletion code is the $(k+1)$-repetition code.

2 Preliminaries

A subsequence in a string x is any string obtained from x by deleting one or more symbols. In contrast, a substring is a subsequence made of several consecutive symbols of x.

Definition 1. Let k be a positive integer. Let $\sigma_k : \{0,1\}^n \to 2^{\{0,1\}^{n-k}}$ be the function which maps an binary string s of length n to the set of all subsequences of s of length $n-k$. That is, $\sigma_k(s)$ is the set of all possible outputs through the k-bit deletion channel.

Definition 2. A two n-bit strings s_1 and s_2 are *k-confusable* if and only if $\sigma_k(s_1)\cap\sigma_k(s_2)\neq\emptyset$.

We now state and develop some basic ingredients that our construction builds upon. Specifically, we will see some simple constructions of hash functions such that the knowledge $\mathsf{hash}(x)$ and an arbitrary string $y\in\sigma_k(x)$ allows one to reconstruct x. Our final deletion codes will use these basic hash functions, which are either inefficient in terms of size or complexity, to build hashes that are efficient both in terms of size and computation time. These will then be used to build deletion codes, after protecting those hashes themselves with some redundancy to guard against k deletions, and then including them also as part of the codeword.

We start with an asymptotically optimal hash size which is inefficient to compute. For runtimes, we adopt the notation $O_k(f(n))$ to denote that the runtime may depend on a hidden function of k.

Lemma 2. *Fix an integer $k\geqslant 1$. There is a hash function $\mathsf{hash}_1 : \{0,1\}^n\to\{0,1\}^m$ for $m\leqslant 2k\log n+O(1)$, computable in $O_k(n^{2k}2^n)$ time, such that for all $x\in\{0,1\}^n$, given $\mathsf{hash}_1(x)$ and an arbitrary $y\in\sigma_k(x)$, the string x can be recovered in $O_k(n^{2k}2^n)$ time.*

Proof. This result follows from an algorithmic modification of the methods of [1]. It is easy to see that for any n-bit string x, $|\sigma_k(x)| \leqslant n^k$. Additionally, for any $(n-k)$-bit string y, the number of n-bit strings s for which $y \in \sigma_k(s)$ is at most $2^k\binom{n}{k} \leqslant 2n^k$. Thus, any n-bit string x is confusable with at most $2n^{2k}$ others strings. Thus, there exists a deterministic procedure in $O_k(n^{2k}2^n)$ time which $(2n^{2k}+1)$-colors these strings. We can define $\mathsf{hash}_1(x)$ to be the color of x.

Given such a hash and a $(n-k)$ bit received subsequence $y \in \sigma_k(x)$, the receiver can in $O_k(n^{2k}2^n)$ determine the color of all strings s for which $y \in \sigma_k(s)$. By design, exactly one of these stings has the color of the hash, so the receiver will be able to successfully decode x, as desired. □

We now modify the above result to obtain a larger hash that is however faster to compute (and also allows recovery from deletions in similar time).

Lemma 3. *Fix an integer $k \geqslant 1$. There is a hash function $\mathsf{hash}_2 : \{0,1\}^n \to \{0,1\}^m$ for $m \approx 2kn\log\log n/\log n$ computable in $O_k(n^2(\log n)^{2k})$ time, such that for all $s \in \{0,1\}^n$, given $\mathsf{hash}_2(s)$ and an arbitrary $y \in \sigma_k(s)$, the string s can be recovered in $O_k(n^2(\log n)^{2k})$ time.*

Proof. We describe how to compute $\mathsf{hash}_2(s)$ for an input $s \in \{0,1\}^n$. Break up the string into consecutive substrings $s_1,\ldots,s_{n'}$ of length $\lceil \log n \rceil$ except possibly for $s_{n'}$ which is of length at most $\lceil \log n \rceil$. For each of these strings, by Lemma 2 we can compute in $O_k(n(\log n)^{2k})$ time the string $\mathsf{hash}_1(s_i)$ of length $\sim 2k\log\log n$. Concatenating each of these hashes, we obtain a hash of length $\sim 2kn\log\log n/\log n$ which takes $O_k(n^2(\log n)^{2k})$ time to compute. The decoder can recover the string s' in $O_k(n^2(\log n)^{2k})$ time by using the following procedure. For each of $i \in \{1,\ldots,n'\}$ if j_i and j'_i are the starting and ending positions of s_i in s, then the substring between positions j_i and $j'_i - k$ in s' must be a subsequence of s_i. Thus, applying the decoder described in 2, we can in $O_k(n(\log n)^{2k})$ time recover s_i. Thus, we can recover s in $O_k(n^2(\log n)^{2k})$ time, as desired. □

We will also be using Reed-Solomon codes to correct k symbol *errors*. For our purposes, it will be convenient to use a systematic version of Reed-Solomon codes, stated below. The claimed runtime follows from near-linear time implementations of unique decoding algorithms for Reed-Solomon codes, see for example [14].

Lemma 4. *Let $k < n$ be positive integers, and q be a power of two satisfying $n + 2k \leqslant q \leqslant O(n)$. Then there exists a map $\mathsf{RS} : \mathbb{F}_q^n \to \mathbb{F}_q^{2k}$, computable in $O_k(n(\log n)^4)$ time, such that the set $\{(x, \mathsf{RS}(x)) \mid x \in \mathbb{F}_q^n\}$ is an error-correcting code that can correct k errors in $O_k(n(\log n)^4)$ time. In particular, given $\mathsf{RS}(x)$ and an arbitrary z at Hamming distance at most k from x, one can compute x in $O_k(n(\log n)^4)$ time.*

3 Deletion-correcting hash for mixed strings

In this section, we will construction a short, efficiently computable hash that enables recovery of a string x from k-deletions, when x is typical in the sense that each short pattern occurs frequently in x (we call such strings *mixed*).

3.1 Pattern-rich strings

We will use n for the length of the (mixed) string to be hashed, and as always k will be the number of deletions we target to correct. The following parameters will be used throughout:

$$d = \lfloor 20000k(\log k)^2 \log n \rfloor \qquad \text{and} \qquad m = \lceil \log k + \log\log(k+1) + 5 \rceil \ . \tag{3}$$

It is easy to see that the choice of m satisfies

$$2^m > 2k(2m-1) \ . \tag{4}$$

(Indeed, we have

$$2^m \geqslant 32k\log(k+1) > 2k(15\log(k+1)) > 2k(2\log k + 2\log\log(k+1) + 11) > 2k(2m-1) \ .)$$

We now give the precise definition of mixed strings.

Definition 3. Let p and s be binary strings of length m and n, respectively, such that $m < n$. Define a *p-split point* of s be an index i such that $p = s_i s_{i+1} \dots s_{i+m-1}$.

Definition 4. We say that a string $s \in \{0,1\}^n$ is *k-mixed* if for every $p \in \{0,1\}^m$, every substring of s of length d contains a *p-split point*. Let $\mathscr{M}_n$ be the set of k-mixed strings of length n.

3.2 Hashing of Mixed Strings

The following is our formal result on a short hash for recovering mixed strings from k deletions.

Theorem 5. *Fix an integer $k \geqslant 2$. Then for all large enough n, there exists $b = O(k^2 \log k \log n)$ and a hash function $H_{\mathsf{mixed}} : \mathscr{M}_n \to \{0,1\}^b$ and a deletion correction function $G_{\mathsf{mixed}} : \{0,1\}^{n-k} \times \{0,1\}^b \to \{0,1\}^n \cup \{\mathsf{Fail}\}$, both computable in $O_k(n(\log n)^4)$ time, such that for any k-mixed $s \in \{0,1\}^n$, and any $s' \in \sigma_k(s)$, we have $G_{\mathsf{mixed}}(s', H_{\mathsf{mixed}}(s)) = s$.*

Definition 5. If $s \in \{0,1\}^n$, $s' \in \sigma_k(s)$, and $p \in \{0,1\}^m$, we say that s' is *p-preserving with respect to s* if there are some $1 \leqslant i_1 \leqslant \ldots \leqslant i_k \leqslant n$ such that s' is obtained from s by deleting $s_{i_1}, \ldots, s_{i_k}$ and so that:

1. no substring of s equal to p contains any of the bits at positions i_j
2. s and s' have an equal number of instances of substrings equal to p

Intuitively, s' is p-preserving with respect to s if we can obtain s' from s by deleting k bits without destroying or creating any instances of the pattern p.

We first prove the following lemma.

Lemma 6. *Fix an integer $k \geqslant 2$. Then for sufficiently large n, there exists a hash function*

$h_{\mathsf{pattern}} : \mathscr{M}_n \times \{0,1\}^m \to \{0,1\}^{2k(\lceil \log n \rceil + 1)}$ *and deletion correction function*

$g_{\mathsf{pattern}} : \{0,1\}^{n-k} \times \{0,1\}^{2k(\lceil \log n \rceil + 1)} \times \{0,1\}^m \to \{0,1\}^n \cup \{\mathit{Fail}\}$,

both computable in $O_k(n(\log n)^4)$ time, such that for every pattern $p \in \{0,1\}^m$, every k-mixed $s \in \{0,1\}^n$, and an arbitrary $s' \in \sigma_k(s)$ that is p-preserving with respect to s, one has

$$g_{\mathsf{pattern}}(s', h_{\mathsf{pattern}}(s,p), p) = s \ .$$

Proof. We first define the hash function h_{pattern} as follows.

Assume we are given a mixed string $r \in \mathscr{M}_n$ and a pattern $p \in \{0,1\}^m$. Let $a_1, \ldots, a_u$ be the p-split points of r. Then we let strings $w_0, \ldots, w_u$ be defined by $w_0 = r_0 \cdots r_{a_1 - 1}$, $w_u = r_{a_u} \cdots r_{n-1}$, and for $1 \leqslant i \leqslant u-1$, $w_j = r_{a_j} \cdots r_{a_{j+1}-1}$. Thus $\{w_j\}$ are the strings that r is broken into by splitting it at the split points. By the definition of a mixed string, each w_j has length at most d (as defined in (3)).

We let ℓ_j be the length of w_j, let v_j be w_j padded to length d by leading 0's, and let $y_j = \mathsf{hash}_2(v_j)$ as defined in Lemma 3, with the binary representation of the length of ℓ_j appended. We can compute y_j in time $O_k((\log n)^2 (\log\log n)^{2k})$ and y_j has length v satisfying

$$v = O\left(2k(k \log k)^2 \log n \frac{\log\log\log n}{\log\log n} + \lceil \log d \rceil\right) < \log n \tag{5}$$

for large enough n.

Let x_j be the number whose binary representation is y_j. Then based on the length of y_j, we have that $x_j < n$. Let q be the smallest power of 2 that is at least $n + 2k$. We then apply lemma 4 to $x = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$ (with all those that are not defined being assigned value 0) to obtain $(y_1, \ldots, y_{2k}) = \mathsf{RS}(x) \in \mathbb{F}_q^{2k}$. For $1 \leqslant j \leqslant 2k$, let S_j be the binary representation of y_k, padded with leading 0's so that its length is $\lceil \log n \rceil + 1$.

Finally, we define the hash value

$$h_{\mathsf{pattern}}(s, p) = S_1 \cdots S_{2k} \ .$$

Clearly, the length of $h_{\mathsf{pattern}}(s,p)$ equals $2k(\lceil \log n \rceil + 1)$.

To compute $g_{\mathsf{pattern}}(s', \tilde{h}, p)$, where s' is a subsequence of s that is p-pattern preserving with respect to s, we split $\tilde{h}$ into $2k$ equal-length blocks, calling them $S_1, \ldots, S_{2k}$. We compute $(x'_1, \ldots, x'_n)$ from s' in the same way that we computed $(x_1, \ldots, x_n)$ from s when defining $h_{\mathsf{pattern}}(s,p)$. Now, assuming $\tilde{h} = h_{\mathsf{pattern}}(s,p)$, there are at most k values of j such that $x'_j \neq x_j$, since there are at most k deletions. We can use Lemma 4 and $S_1, \ldots, S_{2k}$ to correct these k errors. From a corrected value of x_j, we can obtain the value of w'_j and ℓ_j. Since ℓ_j is the length of w_j, we can use it to remove the proper number of leading zeroes from w'_j and obtain w_j. Thus we can restore the original s in $O_k(n(\log n)^4 + n(\log n)^2(\log\log n)^{2k})$ time. Since $(\log\log n)^{2k} \leqslant O_k(1) + O(\log n)$,[1] the overall decoding time is $O_k(n(\log n)^4)$. □

With the above lemma in place, we are now ready to prove the main theorem of this section.

Proof. (of Theorem 5) Given a mixed string $s \in \mathscr{M}_n$, the hash $H_{\mathsf{mixed}}(s)$ is computed by computing $h_{\mathsf{pattern}}(s,p)$ from Lemma 6 for each pattern $p \in \{0,1\}^m$ and concatenating those hashes in order of increasing p. For the decoding, to compute $G_{\mathsf{mixed}}(s', H_{\mathsf{mixed}}(s))$ for a $s' \in \sigma_k(s)$, we run g_{pattern} from Lemma 6 on each of the 2^m subhashes corresponding to each $p \in \{0,1\}^m$, and then take the majority (we can perform the majority bitwise so that it runs in $O_k(n)$ time). When deleting a bit from s, at most $(2m-1)$ patterns p are affected (since at most m are deleted and at most $m-1$ are created). Thus k deletions will affect at most $k(2m-1)$ patterns of length m. Since m was chosen such that $2^m > 2k(2m-1)$, we have that s' is p-preserving with respect to s for a majority of patterns p. Therefore, we will have $g_{\mathsf{pattern}}(s', h_{\mathsf{pattern}}(s,p), p) = s$ for a majority of patterns p, and thus G_{mixed} reconstructs the string s correctly. □

[1] For instance, $(\log\log n)^{2k} \leqslant O(2^{2k^2} + \log n)$, because either $\log\log n \leqslant 2^k$ in which case $(\log\log n)^{2k} \leqslant 2^{2k^2}$, or $(\log\log n)^{2k} \leqslant (\log\log n)^{2\log\log\log n} \leqslant O(\log n)$.

4 Encoding into Mixed Strings

The previous section describes how to protect mixed strings against deletions. We now turn to the question of encoding an arbitrary input string $s \in \{0,1\}^n$ into a mixed string in $\mathscr{M}_n$.

Definition 6. Let $\mu : \{0,1\}^n \times \{0,1\}^L \to \{0,1\}^n$ be the function which takes a string s of length n and a string t, called the *template*, of length L, and outputs the bit-wise XOR of s with t concatenated $\lceil n/L \rceil$ times and truncated to a string of length n.

We will apply the above function with the parameter choice

$$L = \lceil m2^m(\log(n2^m)+1) \rceil \leqslant \lceil 10000k(\log k)^2 \log n \rceil - 1 \ . \tag{6}$$

Equation 6 follows since for $k \geqslant 2$

$$\begin{aligned}
\lceil m2^m(\log(n2^m)+1) \rceil &\leqslant (\log k + \log\log(k+1)+6)(64k\log(k+1))(\log n + m + 1) \\
&\leqslant (8\log k)(128k\log k)(2\log n) \\
&\leqslant \lceil 10000k(\log k)^2(\log n) \rceil - 1
\end{aligned}$$

Notice that for all $s \in \{0,1\}^n$ and $t \in \{0,1\}^L$, $\mu(\mu(s,t),t) = s$. Notice also that μ is computable in $O(n)$ time. It is not hard to see that for any $s \in \{0,1\}^n$, the string $\mu(s,t)$ for a random template $t \in \{0,1\}^L$ will be k-mixed with high probability. We now show how to find one such template t that is suitable for s, deterministically in near-linear time.

Lemma 7. *There exists a function $T : \{0,1\}^n \to \{0,1\}^L$ such that for all $s \in \{0,1\}^n$, $\mu(s,T(s)) \in \mathscr{M}_n$. Also, T is computable in $O(k^3(\log k)^3 \, n\log n) = O_k(n\log n)$ time.*

Proof. For a given string s and template t, we say that a pair $(i,p) \in \{0,1,\ldots,\lfloor n/L \rfloor - 1\} \times \{0,1\}^m$ is an *obstruction* to $r = \mu(s,t)$ being mixed if the substring of $r_{iL+1}\cdots r_{iL+L}$ does not include the pattern p as a substring.

We will choose $T(s)$ algorithmically. For $0 \leqslant j \leqslant \lfloor L/m \rfloor - 1$, at the beginning of step j we have we will have b_j potential obstructions. Clearly, $b_0 = \lfloor n/L \rfloor 2^m$. At step j, we will specify the values of $t_{jm+1}\cdots t_{jm+m}$. If we chose these bits randomly, then $\mathbb{E}[b_{j+1}] = (1-2^{-m})b_j$. Thus we can check all of the 2^m possibilities and find some way to specify the bits so that $b_{j+1} \leqslant (1-2^{-m})b_j$. This will then give us that

$$b_{\lfloor L/m \rfloor} \leqslant (1-2^{-m})^{\lfloor L/m \rfloor}(\lfloor n/L \rfloor 2^m) \leqslant e^{-\lfloor L/m \rfloor 2^{-m}} n2^m < 1$$

by our choice of L in (6).

Thus we can find a template $T(s)$ with no obstructions in $\lfloor L/m \rfloor$ steps. The definition of an obstruction tells us that then every substring of $r = \mu(s,T(s))$ of length $2L \leqslant \lfloor 20000k(\log k)^2 \log n \rfloor = d$ contains every pattern of length m, so the string $r \in \mathscr{M}_n$.

Let us estimate the time complexity of finding $T(s)$. Fix a step j, $0 \leqslant j < \lfloor L/m \rfloor$. Going over all possibilities of $t_{jm+1}\cdots t_{jm+m}$ takes 2^m time, and for each estimating the reduction in number of potential obstructions takes $O(n2^m)$ time. The total runtime is thus $O(n4^m L/m) = O(n8^m \log(n2^m)) = O(k^3(\log k)^3 n \log n)$. □

5 The Encoding/Decoding Scheme: Proof of Theorem 1

Combining the results from Sections 3 and 4, we can now construct the encoding/decoding functions Enc and Dec which satisfy Theorem 1. But first, as a warm-up, we consider a simple way to obtain such maps for codewords of slightly larger length $N \leqslant n + O(k^3 \log k \log n)$ (i.e., the redundancy has a cubic rather than quadratic dependence on k). Let s be the message string we seek to encode. First compute $t = T(s)$ and $r = \mu(s,t)$ as per Lemma 7. Then, define the encoding of s as

$$\mathsf{Enc}(s) = \langle r,\ \mathrm{rep}_{k+1}(t),\ \mathrm{rep}_{k+1}(H_{\mathsf{mixed}}(r)\ \rangle\ ,$$

where $H_{\mathsf{mixed}}(\cdot)$ is the deletion-correcting hash function for mixed strings from Theorem 5, and rep_{k+1} is the $(k+1)$-repetition code which repeats each bit $(k+1)$ times. Since $H_{\mathsf{mixed}}(r)$ is the most intensive computation, $\mathsf{Enc}(s)$ can be computed in $O_k(n(\log n)^4)$ time and its length is

$$\begin{aligned} n + (k+1)L + (k+1)|H_{\mathsf{mixed}}(r)| &\leqslant n + (k+1)O(k(\log k)^2 \log n) + (k+1)O(k^2 \log k \log n) \\ &\leqslant n + O(k^3 \log k \log n). \end{aligned}$$

We now describe the efficient decoding function Dec. Suppose we receive a subsequence $s' \in \sigma_k(\mathsf{Enc}(s))$, First, we can easily decode t and $H_{\mathsf{mixed}}(r)$, since we know the lengths of t and $H_{\mathsf{mixed}}(r)$ beforehand. Also the first $n-k$ symbols of s' yield a subsequence r' of r of length $n-k$. Then, as shown in Theorem 5, $r = G_{\mathsf{mixed}}(r', H_{\mathsf{mixed}}(r))$ and this can be computed in $O_k(n(\log n)^4)$ time. Finally, we can compute $s = \mu(r,t)$, so we have successfully decoded the message s from s', as desired.

Now, we demonstrate how to obtain the improved encoding length of $n + O(k^2 \log k \log n)$. The idea is to use Lemma 3 to protect t and $H_{\mathsf{mixed}}(r)$ with less redundancy than the naive $(k+1)$-fold repetition code.

Proof. (of Theorem 1) Consider the slightly modified encoding

$$\mathsf{Enc}(s) = \langle r,\ t,\ \mathrm{rep}_{k+1}(\mathsf{hash}_2(t)),\ H_{\mathsf{mixed}}(r),\ \mathrm{rep}_{k+1}(\mathsf{hash}_2(H_{\mathsf{mixed}}(r)))\ \rangle\ . \tag{7}$$

The resulting codeword can be verified to have length $O(k^2(\log k)(\log n))$ for large enough n; the point is that $\mathsf{hash}_2()$ applied to t and $H_{\mathsf{mixed}}(r)$, which are $O_k(\log n)$ long strings, will result in strings of length $o_k(\log n)$, so we can afford to encode them by the redundancy $(k+1)$ repetition code, without affecting the dominant $O_k(\log n)$ term in the overall redundancy.

Since we know beforehand, the starting and ending positions of each of the five segments in the codeword (7), we can in $O(n)$ time recover subsequences of r, t, $\mathrm{rep}_{k+1}(\mathsf{hash}_2(t))$, $H_{\mathsf{mixed}}(r)$, and $\mathrm{rep}_{k+1}(\mathsf{hash}_2(H_{\mathsf{mixed}}(r))$ with at most k deletions in each. By decoding the repetition codes, we can recover $\mathsf{hash}_2(t)$ and $\mathsf{hash}_2(H_{\mathsf{mixed}}(r))$ in $O(n)$ time. Then, using the algorithm described in Lemma 3, we can recover t and $H_{\mathsf{mixed}}(r)$ in $O_k(n'^2(\log n')^{2k})$ time where $n' = \max(L, |H_{\mathsf{mixed}}(r)|) = O(k^2 \log k \log n)$. Once t and $H_{\mathsf{mixed}}(r)$ are recovered, we can proceed as in the previous argument and decode s in $O_k(n(\log n)^4)$ time, as desired. □

6 Efficient Algorithm for Correcting Insertions and Deletions

By a theorem of Levenshtein [1], we have that our code works not only on the k-bit deletion channel but also on the k-bit insertion and deletion channel. The caveat though with this theorem is that the decoding algorithm may not be as efficient. In this section, we demonstrate a high-level overview of a proof that, with some slight modifications, the code we constructed for k deletions can be efficiently decoded on the k-bit insertion and deletion channel. Although the redundancy will be slightly worse, its asymptotic behavior will remain the same.

To show that our code works, we argue that suitable modifications of each of our lemmas allow the result to go through.

- Lemma 2 works for the k-bit insertion and deletion channel by Levenshtein's result [1]. Since encoding/decoding were done by brute force the efficiency will not change by much.
- To modify Lemma 3 we show that the code which corrects $3k$ deletions can also correct k insertions and k deletions nearly as efficiently. If the codeword transmitted is $s_1, \ldots, s_{n'}$ where each s_i is of length at most $\lceil \log n \rceil$ then in the received word, if s_i was supposed to be in positions i_a to i_b, then positions $i_a + k$ to $i_b - k$ must contain bits from s_i except possibly for k spurious insertions. Using Lemma 2 modified for insertions, we can restore s_i using brute force, and thus we can restore the original string with about the same runtime as before.
- Lemma 4 and Lemma 6 do not change because the underlying error-correcting code does not depend on deletions or insertions.
- Theorem 5 does not change because the hash works in an error-correcting way and the number of affected "foiled" patterns is roughly the same.
- Lemma 7 does not change.
- Theorem 1 needs some modifications. The encoding has the same general structure except we use the hash functions of the modified lemmas and we use a $(3k+1)$-fold repetition code instead of a $(k+1)$-fold repetition code. That is, our encoding is

  $$\phi(s) = \langle r,\, t,\, \mathrm{rep}_{3k+1}(\mathsf{hash}_2(t)),\, H_{\mathsf{mixed}}(r),\, \mathrm{rep}_{3k+1}(\mathsf{hash}_2(H_{\mathsf{mixed}}(r)))\rangle\ .$$

  In the received codeword we can identity each section with up to k bits missing on each side and k spurious insertions inside. In linear time we can correct the $(3k+1)$-repetition code by taking the majority vote on each block of length $3k+1$. Thus, we will have $\mathsf{hash}_2(t)$ and $\mathsf{hash}_2(H_{\mathsf{mixed}}(r))$ from which we can obtain t, $H_{\mathsf{mixed}}(r)$, and finally r and s in polynomial (in fact, near-linear) time as in the deletions-only case.

Thus, we have exhibited an efficient code encoding n bits with $O(k^2 \log k \log n)$ redundancy and which can be corrected in near-linear time against any combination of insertions and deletions totaling k in number.

7 Concluding remarks

In this paper, we exhibit a first-order asymptotically optimal efficient code for the k-bit deletion channel. Note that to improve the code length past $n + O(k^2 \log k \log n)$, we would need to modify our hash function $H_{\mathsf{mixed}}(r)$ so that either it would use shorter hashes for each particular pattern p or it would require using fewer patterns p. The former would require distributing the hash information between different patterns, which may not be possible since the patterns do not synchronize with each other. An approach through the latter route seems unlikely to improve past $n + O(k^2 \log n)$ since an adversary is able to "ruin" k essentially independent patterns because the string being transmitted is k-mixed.

Another interesting challenge is to give a deterministic one-way protocol with $\mathrm{poly}(k\log n)$ communication for synchronizing a string $x \in \{0,1\}^n$ with a subsequence $y \in \sigma_k(x)$ (the model discussed in Section 1.3). The crux of our approach is such a protocol when the string x is mixed, but the problem remains open when x can be an arbitrary n-bit string.

Another intriguing question is whether there is an extension of the Varshamov-Tenengolts (VT) code for the multiple deletion case, possibly by using higher degree coefficients in the check condition(s) (for example, perhaps one can pick the code based on $\sum_{i=1}^{n} i^a x_i$ for $a = 0,1,2,\ldots,d$ for some small constant d). Note that this would also resolve the above question about a short and efficient deterministic hash for the synchronization problem. However, for the case of two deletions there are counterexamples for $d \leqslant 4$, and it might be the case that no such bounded-degree polynomial hash works even for two deletions.

Acknowledgment

The second author thanks Michael Saks for valuable discussions about the problem of recovering from multiple deletions and in particular about the possibility of finding a VT-like code for correcting two deletions.

References

[1] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966. 1, 2, 5, 9, 10

[2] R. R. Varshamov and G. M. Tenengol'ts. Codes which correct single asymmetric errors. *Autom. Remote Control*, 26(2):286–290, 1965. 1

[3] Neil J. A. Sloane. On single-deletion-correcting codes. In *Ohio State University*, pages 273–291, 2001. 2

[4] Albertus S. J. Helberg and Hendrik C. Ferreira. On multiple insertion/deletion correcting codes. *IEEE Transactions on Information Theory*, 48(1):305–308, 2002. 2

[5] Khaled A. S. Abdel-Ghaffar, Filip Paluncic, Hendrik C. Ferreira, and Willem A. Clarke. On Helberg's generalization of the levenshtein code for multiple deletion/insertion error correction. *IEEE Transactions on Information Theory*, 58(3):1804–1808, 2012. 2

[6] F. Paluncic, Khaled A.S. Abdel-Ghaffar, H.C. Ferreira, and W.A. Clarke. A multiple insertion/deletion correcting code for run-length limited sequences. *IEEE Transactions on Information Theory*, 58(3):1809–1824, March 2012. 2

[7] Leonard J. Schulman and David Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *Information Theory, IEEE Transactions on*, 45(7):2552–2557, Nov 1999. 2

[8] Venkatesan Guruswami and Carol Wang. Deletion codes in the high-noise and high-rate regimes. In *Proceeding of RANDOM*, 2015. To appear; available from arXiv:1411.6667. 2, 3

[9] Graham Cormode, Mike Paterson, Süleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 197–206, 2000. 3

[10] Alon Orlitsky. Interactive communication of balanced distributions and of correlated files. *SIAM J. Discrete Math.*, 6(4):548–564, 1993. Preliminary version in FOCS'91. 3

[11] Hossein Jowhari. Efficient communication protocols for deciding edit distance. In *20th Annual European Symposium on Algorithms*, pages 648–658, 2012. 4

[12] Utku Irmak, Svilen Mihaylov, and Torsten Suel. Improved single-round protocols for remote file synchronization. In *Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1665–1676, 2005. 4

[13] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucky. Low distortion embedding from edit to Hamming distance using coupling. *Electronic Colloquium on Computational Complexity (ECCC)*, 22:111, 2015. 4

[14] Shuhong Gao. A new algorithm for decoding Reed-Solomon codes. In *Communications, Information and Network Security, V.Bhargava, H.V.Poor, V.Tarokh, and S.Yoon*, pages 55–68. Kluwer, 2002. 5

[15] Khaled A.S. Abdel-Ghaffar, Hendrik C. Ferreira, and Ling Cheng. On linear and cyclic codes for correcting deletions. In *Information Theory, 2007. ISIT 2007. IEEE International Symposium on*, pages 851–855, June 2007. 12

A Linear deletion codes cannot have rate better than $\frac{1}{k+1}$

In this section, we generalize the work of [15] (who did the $k = 1$ case) to demonstrate that the best asymptotic rate achievable by a linear k-bit binary deletion code is $1/(k+1)$. Note that this is achieved by the $k+1$-repetition code.

Theorem 8. *Let C be a n-bit linear code for the k-bit deletion channel. Then, dim*$(C) \leqslant n/(k+1)+(k+1)^2$.

Proof. Define $C^0 = C, C^1, \dots C^k$ to be subspaces of $\mathbb{F}_2^n$ of dimension $\dim(C)$ such that

$$C^i = \{(x_{i+1},\dots,x_n,x_1,\dots,x_i) \,|\, (x_1,\dots,x_n) \in C\}.$$

Lemma 9. *For all* $0 \leqslant i < j \leqslant k$,

$$\dim(C^i \cap C^j) \leqslant \gcd(j-i,n).$$

Proof. If $z \in C^i \cap C^j$, then there exists $x, y \in C$ such that

$$z = (x_{i+1},\dots,x_n,x_1,\dots,x_n) = (y_{j+1},\dots,y_n,y_1,\dots,y_j).$$

Since $j > i$, we have that $(x_{i+1},\dots x_{n-j+i}) = (y_{j+1},\dots,y_n)$. Thus, when x and y are passed through the k-deletion channel, they could output the same result since $k \geqslant j$. Thus, $x = y$. Hence, $x_\ell = x_{\ell+j-i}$ (indices modulo n) for all $\ell \in \{1,\dots,n\}$. Thus, there are at most $2^{\gcd(j-i,n)}$ choices for x, so $\dim(C^i \cap C^j) \leqslant \gcd(j-i,n)$. □

From the lemma, we can see that

$$\begin{aligned} n &\geqslant \dim\left(\sum_{i=0}^{k} C^i\right) \\ &\geqslant (k+1)\dim(C) - \sum_{i=0}^{k}\sum_{j=i+1}^{k} \dim(C^i \cap C^j) \\ &\geqslant (k+1)\dim(C) - (k+1)^3. \end{aligned}$$

Thus, $\dim(C) \leqslant (k+1)^2 + n/(k+1)$, as desired. □

Remark. If we let n be a prime, then $\dim(C^i \cap C^j) \leqslant \gcd(j-i,n) = 1$. Furthermore, the only possible nontrivial intersection is $11\ldots1$. Thus, the sum of these $k+1$ vector spaces would have to have dimension at least $(k+1)(\dim(C)-1)+1$, from which we can get that $\dim(C) \leqslant (n+k)/(k+1)$.