Deletion codes for a constant number of errors

David Kewei Lin Jensen Jinhui Wang Tan Siah Yong

May 3, 2022

Contents

1 An introduction to deletion codes

In the conventional setting for error correcting codes, typically a codeword can be susceptible to erasures of symbols or substitution of symbols. One feature of this model of errors is that the alignment of symbols is usually preserved: a symbol unaffected by errors will remain in the same position.

However, in some real-world scenarios (e.g. genetic mutations, document editing, packet transmission), there are errors that may completely delete a symbol with no indication of the misalignment. This presents much difficulty to the usual methods used in error correction.

We define error correction as follows:

Definition 1

Given a binary strings $x = x_1x_2...x_n \in \{0,1\}^n$ and $y = y_1y_2...y_{n-1} \in \{0,1\}^{n-1}$, we say that y is x **deleted** at index i if

$$\begin{cases} x_j = y_j & j < i \\ x_{j+1} = y_j & j \geq i \end{cases}$$

for $j = 1, 2..., n-1$. We also say that y can be produced from a **deletion** on x.

Informally, given a string of length n, the deletion of t characters from it results in a message of length $n - t$ that consists of the undeleted characters in the order that they appeared in.

Given this model of error, we can set up error correction for this problem:

Definition 2

A (n,t)**-deletion code** is, informally, a way to encode of length n messages such that t deletions can be corrected.

Formally, this consists of an encoder $\mathsf{enc} : \{0,1\}^n \to \{0,1\}^\ell$ and a decoder $\mathsf{dec} : \{0,1\}^{\ell-t} \to \{0,1\}$ such that

$$\mathsf{dec}(y) = x$$

for any y produced from t deletions on $\mathsf{enc}(x)$.

The **redundancy** of the code is $\ell - n$. For a systematic deletion code (i.e. $\mathsf{enc}(x) = x \circ f(x)$ for some f), the redundancy are the number of additional bits sent by the encoder (i.e. length of output of f).

1.1 Hamming and GV bounds

We can follow the usual logic of both the Hamming and GV bounds to get rate-optimal (albeit non-constructive) constructions and lower bounds for length of a code.

Theorem 1 (GV construction analogue)

There exists a deletion code $\mathcal{C} \subset \{0,1\}^n$ that can correct t deletions where

$$\log |\mathcal{C}| \geq n - 2t \log n - O(1).$$

Proof. Let $\mathcal{B}_{\mathsf{edit}}(x) \subset \{0,1\}^n$ denote all strings that can be produced from x by performing t deletions and t insertions.

We proceed by greedily including x into our code. Whenever we include x, we have to strike off all elements of $\mathcal{B}_{\mathsf{edit}}(x)$ from $\{0,1\}^n$, since for any $z \in \mathcal{B}_{\mathsf{edit}}(x)$, we can make x and z collide by performing t deletions.

Now we claim that $|\mathcal{B}_{\mathsf{edit}}(x)| \leq 2n^{2t}$. The number of ways to delete t-symbols from x is $\binom{n}{t}$, while the number of ways to add t-symbols is less $2^t\binom{n}{t}$, so altogether

$$|\mathcal{B}_{\mathsf{edit}}(x)| \leq \binom{n}{t} \cdot 2^t \binom{n}{t} \leq n^{2t} \cdot \frac{2^t}{(t!)^2} \leq 2n^{2t}.$$

Thus our greedily constructed code has at least

$$|\mathcal{C}| \geq \frac{2^n}{2n^{2t}} = 2^{n-2t\log n - O(1)}.$$

□

Theorem 2 (c.f. Lemma 2 in [7], Hamming bound analogue)
Any deletion code $\mathcal{C} \subset \{0,1\}^n$ able to correct t deletions satisfies

$$\log|\mathcal{C}| \leq n - t\log n + O_t(1)$$

Proof. We follow the proof in [7]. Given $x \in \{0,1\}^n$, define $\mathcal{B}_{\mathsf{del}}(x) \subset \{0,1\}^{n-t}$ to be the set of strings obtainable from deleting t symbols of x.

Thus, a valid code $\mathcal{C}$ must have each $\mathcal{B}_{\mathsf{del}}(x)$ being disjoint across all $x \in \mathcal{C}$. By a count,

$$\sum_{x\in\mathcal{C}} |\mathcal{B}_{\mathsf{del}}(x)| \leq 2^{n-t}$$

Now we analyze $|\mathcal{B}_{\mathsf{del}}(x)|$. In the worst case where x is a string of 0's, $|\mathcal{B}_{\mathsf{del}}(x)| = 1$ since the only string obtainable by deletion is also a string of 0's.

To this end, we define $\|x\|$ to be the number of consecutive runs in x (e.g. 011001 has 4 consecutive runs). Note that

$$|\mathcal{B}_{\mathsf{del}}(x)| \geq \binom{\|x\| - t + 1}{t}$$

since for each choice of t pairwise non-adjacent runs, deleting one symbol from each run gives a different string in $\{0,1\}^{n-t}$. Here, we have used the bijection between t sorted non-adjacent elements $(x_1, ..., x_n)$ of $[n]$ and t sorted distinct elements $(y_1, ..., y_n)$ of $[n-t+1]$ via $x_i \leftrightarrow y_i + (i-1)$. Let $C_\delta = \{x : \|x\| < (1-\delta)\frac{n}{2}\}$, then assuming $\delta = o(1)$ as $n \to \infty$,

$$(|\mathcal{C}| - |C_\delta|) \cdot \binom{(1-\delta)\frac{n}{2} - t + 1}{t} \leq 2^{n-t}$$

$$\Leftrightarrow |\mathcal{C}| - |C_\delta| \leq \frac{2^n}{n^t}(t!)(1 + o_t(1))$$

It remains to show that $|C_\delta|/2^n = o_t(n^{-t})$, but this follows easily from the Chernoff bound and picking $\delta = \Theta_t(\sqrt{\log n/n})$. □

Theorems 1 and 2 suggests that the optimal encoding on $\{0,1\}$ able to correct for t-deletions must have $\Theta(t\log n)$ extra bits compared to the original message. However, the question remains whether we can find (1) a systematic encoding and (2) a polynomial (in n) time computable encoding.

As a remark to the significance of the order $O(t\log n)$, this is precisely the number of bits required to encode the indices and values of the deleted bits (if we could predict which bits were going to be deleted).

1.2 The document exchange problem

The document exchange problem is another problem that involves deletion: given the binary string x, suppose we have made a small number of edits (we will assume, only in this paper, that they are all deletions) to it to produce y, but now we would like to recover to the original string x. We can keep a small amount of additional information about x, called a **sketch**, that is guaranteed to be unchanged.

Formally, we may define it as follows:

Definition 3

A function $\{0,1\}^n \to \{0,1\}^\ell$ is a **sketch** of length ℓ for t deletions if there exists a decoding function $\mathsf{dec} : \{0,1\}^{n-t} \times \{0,1\}^\ell \to \{0,1\}^n$ such that given any input x,

$$\mathsf{dec}(y, \mathsf{sk}(x)) = x$$

for any y produced by t deletions from x (i.e. $y \in \mathcal{B}_{\mathsf{del}}(x)$).

We will denote the **length** of the sketch by $|\mathsf{sk}| := \ell$.

It may seem that the assumption that the sketch is not modified by the deletion process makes it strictly easier than the deletion coding problem. Specifically, a systematic deletion code is immediately a sketch for the document exchange problem.

Surprisingly, it turns out that the converse is true. For this, we require a positive rate encoder:

Theorem 3 (Schulman and Zuckerman, [9])

For some $c, \delta > 0$, there exists an encoder $\mathsf{enc} : \{0,1\}^n \to \{0,1\}^{cn}$ able to correct for δn deletions.

The immediate implication is that for a sufficiently long sketch (i.e. with length $|\mathsf{sk}| > t/\delta$), we can encode it and make it at most c times longer to correct for t deletions. So we can turn any sketch sk into an encoder $\mathsf{enc}_{\mathsf{sk}}$ defined by

$$\mathsf{enc}_{\mathsf{sk}}(x) = x \circ \mathsf{enc}(\mathsf{sk}(x)).$$

To decode this, we simply note that the last $(|\mathsf{enc} \circ \mathsf{sk}| - t)$ bits of $\mathsf{enc}_{\mathsf{sk}}(x)$ is definitely produced by deleting t bits from $\mathsf{enc}(\mathsf{sk}(x))$ (i.e. in the ball $\mathcal{B}_{\mathsf{del}}(\mathsf{enc}(\mathsf{sk}(x)))$).Thus, by definition we can recover the sketch $\mathsf{sk}(x)$, and together with the corrupted version of x (the first $|x| - t$ bits of $\mathsf{enc}_{\mathsf{sk}}(x)$), we can recover x.

Thus, we get the first fact:

Fact 1

If there exists a valid sketch of length $|\mathsf{sk}|$, there exists a valid (systematic) encoding of length $n + O(|\mathsf{sk}|)$.

We may also want to get the correct constant on the leading order term, in which case we should consider instead

$$\mathsf{enc}'_{\mathsf{sk}}(x) = (x, \mathsf{sk}(x), \mathsf{enc}(\mathsf{sk}(\mathsf{sk}(x))))$$

which is decoded by iterating the previous protocol, so instead we have

Fact 2

If there exists a valid sketch of length $|\mathsf{sk}|$, then there exists a valid (systematic) encoding of length $n + |\mathsf{sk}| + O(|\mathsf{sk} \circ \mathsf{sk}|)$. In particular, if $|\mathsf{sk} \circ \mathsf{sk}| = o_t(|\mathsf{sk}|)$, then the length of the encoder is $n + (1 + o_t(1))|\mathsf{sk}|$.

This implies that under fairly reasonable assumptions, we can always produce an encoder which only requires additional length on the same order as the sketch. For this reason, instead of solving the deletion coding problem, we may opt to solve document exchange problem.

1.3 Preliminaries from Algebraic Error Correcting Codes

We recall the construction of a standard Reed-Solomon code. Given a prime power $q > n$, we can construct a $(n, k, d)_q$ code by associating the symbol set with the finite field $\mathbb{F}_q$, encoding the message $x = (x_1, ..., x_k)$ as a degree $k-1$ polynomial $f_x(t) = x_k t^{k-1} + x_{k-1} t^{k-2} + ... + x_1$, then our encoding is simply

$$x \mapsto (f_x(\alpha_1), f_x(\alpha_2), ..., f_x(\alpha_n))$$

for a choice of evaluation points $\alpha_1, ..., \alpha_n \in \mathbb{F}_q$. The resulting code is minimum-distance separable and thus has distance $d = n - k + 1$. If we want to correct t errors, we require at least $d = 2t + 1$.

We can also turn this into a systematic code as follows: instead of encoding x as a polynomial in the above way, we can use Lagrange interpolation to pick a degree $k-1$ polynomial g_x satisfying $g(\alpha_i) = x_i$, then using the same encoding with g_x instead of f_x. The number of redundancies required to correct t errors is thus at most $n - k = d - 1 = 2t$. Formally,

Fact 3 (c.f. [4])

Let $t < n$ and q be a power of 2 such that $n + 2t \leq q \leq O(n)$. Then, there exists a map $\mathsf{RS} : \mathbb{F}_q^n \to \mathbb{F}_q^{2t}$, computable in $O_t(n(\log n)^4)$ time, such that $\{(x, RS(x)) : x \in \mathbb{F}_q^n\}$ are error-correcting codes that can correct t errors in $O_t(n(\log n)^4)$ time.

Informally, for a block length larger than $\log(n + 2t)$, it takes $O(t)$ redundancies to correct for t block errors.

1.4 Notation

We use the following conventions for notation:

1. Unless otherwise specified, n denotes the length of the original message and t denotes the number of deletions.
2. $\log = \log_2$.
3. $O_t(f(n))$ denotes any quantity that is $O(f(n))$ when t is fixed. $\Omega_t(f(n)), o_t(f(n)), \omega_t(f(n)), \Theta_t(f(n))$. This is in contrast to $O(f(n, t))$, which we reserve for the function is bounded by $C \cdot f(n, t)$ for some absolute constant C whenever one of t or n are sufficiently large.
4. For a string x, we will use $|x|$ to denote the length. For a function f whose outputs are strings of fixed length, $|f|$ will denote that length.
5. For strings x, y, $x \circ y$ denotes the concatenation of x and y. For functions f, g, $f \circ g$ denotes the function $(f \circ g)(x) = f(g(x))$.

1.5 Overview of discussion

In this paper, we would like to focus on the deletion coding problem for a constant number, t, of deletions.

In subsection 1.2, we saw that it was equivalent to consider the document exchange problem with t deletions, which we will do instead for the next three sections.

In section 2, we discuss the mixed-string sketch, which gives the order optimal growth rate of $O_t(\log n)$ (c.f. the Hamming bound). We start with the simplest idea to fix alignment: explicit buffers that cannot be entirely removed during deletion. However, this at best requires at least $O(\sqrt{n})$ extra length for encoding, so we need a better idea to get the optimal rate.

In section 3, we discuss the IMS Protocol [6]. The fundamental observation behind this protocol is that if our message is broken up into chunks, then at most t of those chunks can be affected by t deletions, and the remaining chunks appear along contiguous substrings in the corrupted

message. The protocol has multiple layers of increasing "resolution", and makes use of Reed-Solomon codes. Even though this has a worse sketch length of $O(t \log^2 n)$ (compared to the mixed strings approach), this actually forms the basis of a more advanced code in [3], where both ideas are combined to produce a sketch of length $O(t \log n)$. However, we will not go into detail there due to its complexity, and we refer readers to [3].

Finally, in section 4, we describe syndrome compression ([10], which leverages number-theoretic properties to compress any valid sketch into a potentially shorter one. Following the approach of the authors in [10], we show that compressing the mixed string sketch gives a sketch whose length has the best known leading order of $4t \log n$. This is within twice of the nonconstructive GV construction, which is really good. Beyond [10], we suggest that it is possible to start from any sketch at all, and directly iteratively compress it into a leading-order-optimal sketch.

2 Mixed-String Code

This section will review the construction of [2] which yields the first explicit code with redundancy better than $O_t(\sqrt{n})$ for $t \geq 2$-deletions, which was previously the best known bound by [5]. The main result is summarized by the following theorem:

Theorem 4 (c.f. [2])
Fix an integer $t \geq 2$. Given a message $s \in \{0,1\}^n$, there exists an encoding $\hat{s} \in \{0,1\}^n$ and a sketch $\mathsf{sk}(s)$ of size $O(t^2 \log t \log n)$ such that given $\hat{s}' \in \mathcal{B}_{\mathsf{del}}(\hat{s})$ and $\mathsf{sk}(s)$, s can be decoded. The encoding of s into $\hat{s}$ takes $O_t(n)$ time while computing $\mathsf{sk}(s)$ and decoding s both take $O_t(n(\log n)^4)$ time.

2.1 Buffer-based Approaches and the Curse of Explicit Buffers

To motivate the ensuing approach, we first understand the inefficiencies in the buffer-based framework of [5] which leads to $\Omega_t(\sqrt{n})$ redundancies. The main idea is to solve the alignment issue by inserting a buffer b between blocks such that corrupted blocks can still be segregated after t-deletions. The exact form of b and mechanism for splitting up the blocks are not important for our argument but one can imagine using a sufficiently long run of 0's (with some technical details) and then define a splitting point whenever a certain number of consecutive 0's is encountered.

Concretely, suppose a message $s \in \{0,1\}^n$ is divided into u blocks $x_1, ..., x_u$ of size $\frac{n}{u}$ and we define a "buffed"-up version of s as

$$\mathsf{Buff}(s) = x_1 \circ b \circ x_2 \circ b \circ ... \circ b \circ x_u$$

which will be our codeword. After passing $\mathsf{Buff}(s)$ through our deletion channel, we obtain a corrupted codeword

$$\widehat{\mathsf{Buff}}(s) = x'_1 \circ b'_1 \circ x'_2 \circ b'_2 .... \circ b'_{u-1} \circ x'_u.$$

By the definition of b being a valid buffer, we can recover $\hat{s} = (x'_1, x'_2, ..., x'_u)$. Comparing this to our original string $s = (x_1, x_2, ..., x_u)$, we see that a deletion error in s has been transformed into an error in the block x_i (a deletion in s occurring in block x_i would be reflected by a shorter x'_i)! More formally, we denote $i : \{0,1\}^{\frac{n}{u}} \to \mathbb{F}_q$ to be an inclusion of $\{0,1\}^{\frac{n}{u}}$ into the field $\mathbb{F}_q$ where $\log q$ is at least $\lceil \frac{n}{u} \rceil$. As an abuse of notation, $i(x')$ where $|x'| < \frac{n}{u}$ refers to padding $\hat{x}$ with leading 0's until it has length $\frac{n}{u}$ before applying i. Then, a deletion error in s corresponds to an error in one of the coordinates of $(i(x_1), ..., i(x_u)) \in \mathbb{F}_q^u$, since the received word is essentially $(i(x'_1), ..., i(x'_u)) \in \mathbb{F}_q^u$.

Given this transformation from deletions to errors, we can now apply a standard systematic code for t-errors over the field $\mathbb{F}_q$ (e.g. the RS code in Fact 3). The sketch of s in this case is then $\mathsf{sk}(s) = \text{Redundant}(i(x_1), ..., i(x_u)) \in \mathbb{F}_q^{\Omega(t)}$, which is the $\Omega(t)$ redundant $\mathbb{F}_q$ elements required to correct t errors in $\{i(x_i)\}$. Note that given $\mathsf{sk}(s)$, we can use these redundant $\mathbb{F}_q$ elements to recover $(i(x_1), ..., i(x_u))$ from $(i(x'_1), ..., i(x'_u))$, and hence recover $s = x_1 \circ ... \circ x_u$.

Finally, a t-deletion resistant encoding of s is then $(\mathsf{Buff}(s), \mathsf{enc}(\mathsf{sk}(s)))$ where enc is any t-deletion resistant encoding (e.g. a weaker one found by brute force search, since we are now only encoding

the small sketch). This is valid since $\mathsf{sk}(s)$ is recoverable from $\mathsf{enc}(\mathsf{sk}(s))$ by definition of enc being t-deletion resistant, and because we have already argued that we can recover s from $\widehat{Buff}(s)$ and $\mathsf{sk}(s)$. Note that we work with encodings instead of sketches in this section only, due to the additional need to augment the message with buffers.

Counting the total number of redundant bits, we have

$$\begin{aligned}
\#\text{Redundancies} &= u|b| + |\mathsf{enc}(\mathsf{sk}(s))| \\
&= \Omega(u + |\mathsf{sk}(s)|) && (|\mathsf{sk}(s)| \leq |E(\mathsf{sk}(s))|) \\
&= \Omega(u + t \log q) \\
&= \Omega\left(u + \frac{tn}{u}\right) && (q \geq 2^{\frac{n}{u}}) \\
&= \Omega(\sqrt{tn}) && (u + \tfrac{tn}{u} \geq \sqrt{tn} \text{ by the AM-GM inequality})
\end{aligned}$$

which is asymptotically worse than the GV bound. As seen from the above, introducing an explicit buffer b coerces a trade-off between the number of delimiters (scaling linearly in u) and the number of error correction bits (scaling linearly in the block size and hence inversely in u). Thus, an explicit buffer scheme is inefficient even if $|b| = O(1)$ and enc does not add any redundant bits!

To summarize, we have 1) split a message into blocks and insert an *explicit buffer* b between them to rectify the alignment issue and transform deletions into errors and 2) construct a sketch for t *block errors*. The key insights of mixed string encoding is to 1) leverage *implicit buffers* to circumvent the $\Omega(\sqrt{tn})$ curse of explicit buffers and 2) in a concatenation code fashion, correct t-errors in the *sketches of the blocks* $\{sk_0(x_i)\}$ where sk_0 is a weak sketch instead of the blocks themselves.

2.2 Mixed Strings and Implicit Buffers

A natural approach of constructing implicit buffers is to exploit repeated occurrences of substring within a codeword itself. Once again, it is beneficial for the implicit buffer to recur frequently since the error correction redundancies scales inversely with the number of blocks u.

If a substring p in a codeword is "preserved" by deletion, it can be used as an implicit buffer. Intuitively, the potential of a substring p in becoming a buffer can be damaged by deletion in two ways. A rightful instance of p in the original code word could be destroyed due to bits in that instance being deleted (e.g. $0110 \to 010$ destroys 11) or a fictitious p could be created due to deletion (e.g. $0110 \to 00$ creates 00). These adversarial cases are precisely precluded by the following two conditions for p to be preserved.

Definition 4

Given $s \in \{0,1\}^n$, let $s' \in \mathcal{B}_{\mathsf{del}}(s)$ be obtained from deleting the bits at positions $1 \leq i_1 < .... < i_t \leq n$ in s. For any substring p of s, s' is **p-preserving** with respect to s if

1. No substring of p in s contains a bit in positions $i_1, ..., i_t$.
2. s and s' have an equal number of occurrences of p as substrings.

If a received word s' is p-preserving with respect to s, we can use p as a buffer to again transform deletions into block errors. However, the main limitations are that 1) p is not known to be preserved a priori and 2) there is no guarantee that p recurs frequently enough in s for the subsequent error-correction sketch to be efficient (i.e. whether p will be an efficient implicit buffer). As such, we hedge our bets by defining codewords to fall within a class of "buffer-rich" strings where all $p \in \{0,1\}^m$ are efficient buffer candidates since they occur in every length d substring of s.

Definition 5

A string $s \in \{0,1\}^n$ is called **t-mixed** every length d substring of s contains all $p \in \{0,1\}^m$ as sub-substrings. We denote the set of t-mixed strings of length n by $\mathcal{M}_n$. The parameters d, m as functions of n, t are:

$$d = \lfloor 20000t(\log t)^2 \log n \rfloor$$

$$m = \lceil \log t + \log\log(t+1) + 5 \rceil$$

where m is chosen to satisfy $2^m > 2t(2m-1)$.

The hope is then that s' is p-preserving with respect to $s \in \mathcal{M}_n$ for most $p \in \{0,1\}^m$ and hence taking some form of a "majority vote" leads to a valid decoding algorithm once we have a decoding algorithm for the limiting case where some p is preserved. Indeed, we shall show that the total number of p's affected by a single deletion is at most $2m-1$ so such a majority scheme is feasible since we have chosen m such that the total number of p's (2^m) is larger than twice the total number of affected p's ($t(2m-1)$).

2.3 p-Preserving Sketch

As a preliminary step, we will construct a sketch for a mixed string s while assuming that a known $p \in \{0,1\}^m$ will be preserved by deletion.

Our approach will parallel that in Section 2.1, with $b \to p$ and $Buff(s) \to s$. Since there is no longer a need to introduce explicit buffers, we can now consider a sketch of s rather an encoding. As a crude first step, suppose we follow Section 2.1 with the appropriate substitutions and tried to define our sketch of s as $\mathsf{sk}(s) = RS(i(x_1), ..., i(x_u)) \in \mathbb{F}_q^{2t}$ given in Fact 3. A slight technicality here is that the map i does not make sense since $x_1, ..., x_u$ are now not necessarily the same length. However, note that $s = x_1 \circ p \circ x_2 \circ p ... \circ p \circ x_u$ and since p occurs in every length d substring of s, $|x_i| \leq d - m < d$. Hence, an intuitive idea is to define $i(x_i)$ to be the i applied to x_i padded with leading 0's to length d. However, we then encounter a problem in recovering x_i from $i(x_i)$ during decoding since we do not know the number of padded zeros. The correct map is then $i'(x_i) = (i(x_i), |x_i|)$ where $i(x_i)$ refers to the intuitive definition, since the second component now informs us the number of zeros to strip off after inverting i. With this clarification, our actual sketch is $\mathsf{sk}(s) = \mathsf{RS}(i'(x_1), ..., i'(x_u)) \in \mathbb{F}_q^{2t}$ where $\log q = \max(|i(x_i)| + \log d, \log(n+2t)) = O(d)$ (the max is necessary due to the inequality $q \geq u + 2t$ in Fact 3). Then, our sketch has length $|\mathsf{sk}(s)| = 2t \log q = 2t \cdot O(d) = O(t^2(\log t)^2 \log n)$, where the dependence on t is suboptimal.

The next insight of [2] is that given the correspondences between blocks $x_i \leftrightarrow x_i'$ after fixing the alignment issue, we have u smaller deletion problems of size $|x_i| \leq d$. Thus, instead of directly finding x_i, we just have to find $\mathsf{sk}_0(x_i)$ where sk_0 is any other sketch (plus some technical details, again due to $|x_i|$'s being different). If we have some weak sketch sk_0 with domain $\{0,1\}^d$, it suffices to construct $\mathsf{sk}(s)$ such that we can recover $\{(\mathsf{sk}_0(x_i), |x_i|)\}$ from $\{x_i'\}, \mathsf{sk}(s)$ after which we can recover x_i from x_i', $|x_i|$, $\mathsf{sk}_0(x_i)$[1], and thus also recover $s = x_1 \circ p \circ x_2 \circ p ... \circ p \circ x_u$. Note that here $\mathsf{sk}_0(x_i)$ again means sk_0 applied to x_i padded with $d - |x_i|$ leading zeros (so we again need the trick of encoding $|x_i|$). Now, t deletions in s are reflected as t errors in $\{(\mathsf{sk}_0(x_i), |x_i|)\}$, which can be guarded by $\mathsf{RS}(\{(\mathsf{sk}_0(x_i), |x_i|)\}) \in \mathbb{F}_q^{2t}$ where $\log q = \max((|\mathsf{sk}_0(x_i)| + \log d), \log(n+2t))$. If sk_0 is little-o in its input length such that $|\mathsf{sk}(x_i)| = o(d)$, the second term in the max dominates asymptotically so we only need $2t\log(n+2t)$ redundant bits.

Indeed, there exists a weak sketch sk_0 that is little-o in its input length and is constructed using a brute force search.

Fact 4 (c.f. [2])

There exists a sketch $\mathsf{sk}_0 : \{0,1\}^n \to \{0,1\}^R$ for $R \approx \frac{2tn \log\log n}{\log n}$ encodable and decodable in $O_t(n^2(\log n)^{2t})$.

[1] We can assume that d, t are in our sketch, since they require $o(\log n)$ bits. Pad x_i' with leading zeros or truncate it from the front until it has length $d - t$, which then differs from x_i padded with leading zeros by t-deletions and hence the latter can be recovered using $\mathsf{sk}_0(x_i)$. Stripping $d - |x_i|$ zeros from the latter then returns x_i.

Using this form of sk_0, we obtain the following theorem, where the bottlenecks for the encoding and decoding times are due to the RS code.

Theorem 5 (c.f. [2])

For n large enough, for any $p \in \{0,1\}^m$, there exists a p-preserving sketch $\mathsf{sk}_p : \{0,1\}^n \to \{0,1\}^{2t\log(n+2t)}$ that is encodable and decodable in $O_t(n(\log n)^4)$ time. That is given $\mathsf{sk}(s)$ and s' that is p-preserving with respect to $s \in \{0,1\}^n$, one can recover s.

The explicit expression for sk_p, where $s = x_1 \circ p \circ x_2 \circ p ... \circ p \circ x_u$, is

$$\mathsf{sk}_p(s) = \text{Redundant}_{RS}(\{\mathsf{sk}_0(x_i), |x_i|\})$$

where sk_0 is given in Fact 4.

2.4 Mixed String Sketch

Having constructed a p-preserving sketch, we now turn to a sketch for mixed strings. Firstly, we have the following lemma which yields a $t(2m-1)$ upper bound on the number of invalidated buffer candidates $p \in \{0,1\}^m$ due to t deletions.

Lemma 1

Given $s \in \{0,1\}^n$ and $s' \in \mathcal{B}_{\mathsf{del}}(s)$, s' is not p-preserving with respect to s for at most $t(2m-1)$ substrings $p \in \{0,1\}^m$.

Proof. Referring to the following figure, a single deletion destroys at most m length m patterns and creates at most $m-1$ patterns. Hence, t deletions do not preserve at most $t(2m-1)$ such $p \in \{0,1\}^m$.

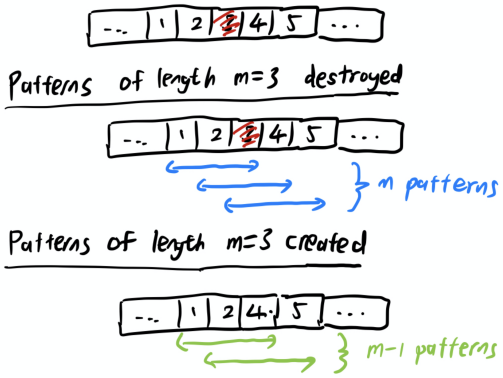

Figure 1: Patterns destroyed and created by a single deletion for $m = 3$.

□

Equipped with this lemma, we have the following sketch for mixed strings.

Theorem 6

A sketch for $s \in \mathcal{M}_n$ is $\mathsf{sk}_{mix}(s) = \circ_{p\in\{0,1\}^m} \mathsf{sk}_p(s)$, which has length $O\left(t^2 \log t \log n\right)$ and is encodable and decodable $O_t\left(n(\log n)^4\right)$ time.

Proof. Firstly, note that $m = O(\log t + \log\log t)$ is independent of n so $|\mathsf{sk}(s)| = 2^m \cdot |\mathsf{sk}_p(s)| = t\log t \cdot |\mathsf{sk}_p(s)| = O\left(t^2 \log t \log n\right)$. The encoding involves computing $\mathsf{sk}_p(s)$ for all $p \in \{0,1\}^m$

which does not change asymptotic dependencies in n. To decode, one can attempt to use the p-preserving decoder for all $p \in \{0,1\}^m$ in $O_t(n(\log n)^4)$ time and then take the majority vote for each bit in s in $O_t(n)$ time (for cases where p is not preserved, the decoder can return an arbitrary output). This majority scheme succeeds since Lemma 1 upper bounds the number of p's that are not preserved by $t(2m-1)$ but we have chosen m in Definition 5 such that the number of possible p's is $2^m > 2 \cdot t(2m-1)$. □

2.5 Encoding into Mixed Strings

Given our sketch for mixed strings, the last step is to encode messages into mixed strings in a reversible manner. Firstly, note that to ensure that a string $\hat{s}$ is in $\mathcal{M}_n$, it suffices to check that after dividing $\hat{s}$ into contiguous blocks of size $\frac{d}{2}$, each block contains all possible $p \in \{0,1\}^m$—then, any length d substring of $\hat{s}$ contains one such length $\frac{d}{2}$ block and hence contains all $p \in \{0,1\}^m$.

The main idea towards achieving this goal is to derandomize the following probabilistic method. Given a message $s \in \{0,1\}^n$, we can split s into $\lfloor \frac{n}{L} \rfloor$ blocks s_i of length $L < \frac{d}{2}$ and possibly one block of length $< L$ (this last block does not matter for our goal of constructing a mixed string). Now, suppose we choose a random **template** $T(s) \in \{0,1\}^L$ and define $\tilde{T}(s)$ to be $T(s)$ concatenated $\lceil \frac{n}{L} \rceil$ times and truncated to length n. Then, if we define $\hat{s} = s$ XOR $\tilde{T}(s)$, it is quite plausible that for large enough L in terms of m, there is a non-zero probability of every length L block $\hat{s}_i$ of $\hat{s}$ containing all $p \in \{0,1\}^m$. Here's a (non-rigorous) heuristic argument for how large L should be, based on a coupon collector argument. We can define $E_{i,p}$ to be the event that $\hat{s}_i$ contains p. Note that we just need $\cap_{i,p} E_{i,p}$ to ensure that $\hat{s} \in \mathcal{M}_n$. Pretend that we progressively reveal each independent length m block of $T(s)$ which, when XORed with the corresponding section in $\hat{s}_i$ to produce p, collects (i,p) (i.e. achieves event $E_{i,p}$). There are $\lfloor \frac{n}{L} \rfloor \cdot 2^m$ such (i,p) pairs to "collect" so a coupon collector argument naively yields $O\left(n2^m \log(n2^m)\right)$ turns of drawing "coupons". However, each revelation costs length m and draws $\frac{n}{L}$ coupons "simultaneously" from the $\frac{n}{L}$ blocks of length L, so the total length of L has to be on the order of $\frac{mL}{n} \cdot O\left(\frac{n2^m}{L} \log\left(\frac{n2^m}{L}\right)\right) = O\left(m2^m \log(n2^m)\right)$, which explains our choice of $d > 2L$ in Definition 5 given our previous choice of m.

We formalize and de-randomize this intuition below:

Theorem 7 (slightly improved from [2])

There exists $\mu : \{0,1\}^n \to \mathcal{M}_n \times \{0,1\}^L$ with $L = \lceil m2^m(\log(n2^m)+1) \rceil < \frac{d}{2}$ such that $\mu(s) = (\hat{s}, T(s))$ where $T(s)$ is a template and $\hat{s} = s$ XOR $\tilde{T}(s)$.

μ is computable in $O(t \log t \cdot n) = O_t(n)$ time. Furthermore, note that given $T(s)$ and $\hat{s}$, we can recover $s = \hat{s}$ XOR $\tilde{T}(s)$ in $O(n)$ time.

Proof. Once again, we try to collect the coupons "(i, p)" where initially we have $n_0 = \lfloor \frac{n}{L} \rfloor 2^m$ such coupons to collect. Now, break $T(s)$ into blocks of length m and let n_i be the number of coupons left to collect after we have chosen the ith block of length m. Note that given n_{i-1}, choosing the ith block of length m in $T(s)$ uniformly randomly would lead to $E[n_i] = (1-2^{-m})\, n_{i-1}$. Thus, by performing a brute-force search over all $\{0,1\}^m$, we can find a length m string to set as the ith block in $T(s)$ such that $n_i \le (1-2^{-m})\, n_{i-1}$. Thus, after choosing all $\lfloor \frac{L}{m} \rfloor$ blocks in $T(s)$,

$$b_{\lfloor \frac{L}{m} \rfloor} \le \left(1-2^{-m}\right)^{\lfloor \frac{L}{m} \rfloor} \left\lfloor \frac{n}{L} \right\rfloor 2^m \le e^{-\lfloor \frac{L}{m} \rfloor 2^{-m}} n2^m < 1$$

by our choice of L, which implies that all coupons (i,p) have been collected. Since $\hat{s} = s$ XOR $\tilde{T}(s)$, when divided into length $L < \frac{d}{2}$, has each length L block containing all $p \in \{0,1\}^m$, $\hat{s} \in \mathcal{M}_n$. Since there are $\lfloor \frac{L}{m} \rfloor$ steps, for which we search over all $a \in \{0,1\}^m$ and the XOR computation and estimation of the reduction in n_i (using $\frac{n}{L}$ hashsets) for each a take $O\left(\frac{nm}{L}\right)$ and $O\left(\frac{n}{L}\right)$ time respectively, the overall time complexity is $O\left(\frac{L}{m} \cdot 2^m \cdot \frac{nm}{L}\right) = O(n2^m) = O(t \log t \cdot n)$. □

Finally, we are ready to prove the main theorem of this section.

Proof of Theorem 4. Given μ in Theorem 7, let $\mu(s) = (\hat{s}, T(s))$ where $\hat{s} \in \mathcal{M}_n$ is used in Theorem 4. Then, define $\mathsf{sk}(s) = (\mathsf{sk}_{mix}(\hat{s}), T(s))$ where $|\mathsf{sk}_{mix}(\hat{s})| = O(t^2 \log t \log n)$ and $|T(s)| = O(d) =$

$O(t(\log t)^2 \log n)$ so $|\mathsf{sk}(s)| = O(t^2 \log t \log n)$. Given $\hat{s}' \in \mathcal{B}_{\mathsf{del}}(\hat{s})$ and $\mathsf{sk}(s)$, one can first decode $\hat{s}$ in $O_t(n(\log n)^4)$ time by Theorem 6 and then recover $s = \hat{s}$ XOR $\tilde{T}(s)$ in $O(n)$ time. □

To conclude, we have seen in this section how to construct a sketch of size $O(t^2 \log t \log n)$, which has the optimal dependence on n as seen from the Hamming bound. However, the dependence on t still trails the GV construction by a factor $O(t \log t)$.

3 The IMS protocol

We follow the exposition in [3] about the protocol first introduced in [6]. We show that:

Theorem 8

There exists a sketch of length $O(t \log^2 n)$ for document exchange that can be efficiently encoded and decoded.

3.1 IMS Protocol with a single layer

We first outline a simpler version of the construction that gives a sketch of length $O(\sqrt{tn \log n})$. The first observation is that if we split the original document x into k blocks of length n/k and apply t deletions, then at least $k - t$ blocks are not affected, and in particular these blocks must appear contiguously in the corrupted document y.

How might we detect an uncorrupted block? Alice does not have the bandwidth to send most of the blocks, but this can be overcome with hashing. Suppose Alice sends over the hash of each block. Bob performs the following algorithm[2]:

- Bob checks through all contiguous substrings in y of length n/k (termed "windows" of y), and sees if the hash of any window matches Alice's hash. We term such windows **active**.
- Bob (simultaneously) removes any active window that overlaps with any other active window.
- For each remaining active window, Bob assigns it to the leftmost unassigned hash which matches this window.

We claim that with a hash of length $O(\log n)$ and a seed of length $O(\log n)$, Bob's protocol ensures that he successfully assigns the correct window for all but $4t$ blocks. We defer the discussion of correctness to Subsections 3.3 and 3.4.

Thus, given k hashes of length $O(\log n)$ each, Alice can send an additional $6t$ blocks (e.g. with a RS code) to correct for the $4t$ potentially incorrectly (pre-hashed) blocks. This gives a combined length of $O(k \log n + tn/k)$, and optimizing over k we get the promised $O(\sqrt{tn \log n})$

3.2 The full, multi-layer protocol

To improve on the above, we see that the bottle neck comes from sending $O(t)$ additional blocks of length n/k each.

As such, the next natural step would be to reduce the size of the chunks we are using since we know that in general at most t of the chunks will end up corrupted. To do this, we can subdivide each chunk into two smaller chunks, this gives us a maximum of $2t$ subchunks that we are not sure regarding the contents of (from the up to t chunks that we don't find a matching window for). We can then hash all of the subchunks, and then using something like an RS code, send an additional $2t$ hashes to correct for the up to $2t$ bad subchunks. Now we have all the hashes for the subchunks, we can repeat the sliding window method. Since there are at most t errors, there will be again at most t subchunks that we are unable to match via the sliding window method. We can then repeat this process until the chunk size is smaller than the size of the hash. At this stage we can use the identity function as our "hash" instead and then via RS codes, correct it to get our original message.

[2] This algorithm was never fully specified in [3].

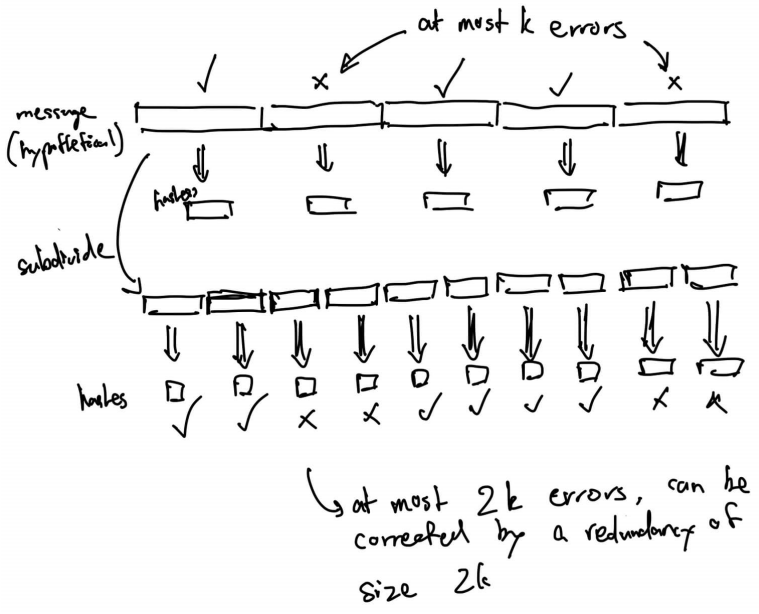

Figure 2: Subdivision as well as bound on number of errors in hashes

Let us now consider the size of the final sketch. If we set k to be $O(t)$, then we will have $O(t)$ groups initially with $O(t)$ errors. With a hash size of $O(\log n)$, our initial set of hashes that we send over has size $O(t \log n)$. Next, we note that since our initial chunks had size $\frac{n}{t}$, we have $O(\log \frac{n}{t})$ levels. Since for each level we have to send a residual of $O(t)$ 'hashes' (they're not really hashes, but they have the same size as a hash) to assist with correcting, each with size $O(\log n)$, the total size of these residuals is $O(t \log n \log(\frac{n}{t}))$. Hence, the total size of the sketch is $O(t \log n \log(\frac{n}{t})) + O(t \log n) = O(t \log n \log(\frac{n}{t}))$ which is significantly better (smaller) than $O(\sqrt{tn \log n})$ for large n. Additionally, $O(t \log n \log(\frac{n}{t})) = O(t \log^2 n)$ so we have proven the theorem at the beginning of this section as desired.

3.3 Correctness and required attributes of hash

We claim that the IMS protocol will work with $O(t)$ redundancies on each layer in conjunction with a hash that ensures no hash collisions among the values present among the windows of the original message, for a particular window size. This is in contrast to the claim in [3] that the hash must be collision-resistant across possible windows of y, which forces the hash to be of size at least $O(t \log n)$.

First, recall our protocol, we perform a sliding window over our received message, denote a window as active if the hash of its contents correspond to one of the received/derived hash for that layer. Next, we disregard all active windows that have an overlap with at least one other active window. Finally, for the remaining active windows we sequentially match them to their corresponding hash, going from left to right. If there is a window with no remaining hash to match to, we disregard it. Similarly, if there is a hash that has no window to match it, we let the all-zeros block match to it as a placeholder. We claim (and shall later prove) that at most $4t$ of the hashes have the wrong pre-image (i.e. the contents of the block matched to the hash is not the same as the block from the original message). As a side note, in actual fact at most $3t$ of the hashes have the wrong pre-image, but the proof for $4t$ is easier to explain, and within a constant factor these numbers do not matter.

First, denote a good window as a window from the original message that maps to some provided/derived hash value. Note that by our definition of the hash function as being collision free, among windows in our original message, all good windows correspond to actual chunk values at some level of our message. A bad window is defined as any window value that arises from some number of deletions that hashes to some provided/derived hash value. Evidently, all active windows by definition are either good or bad. Observe that in our first step where we disregard overlapping

active windows, one error can only cause at most three good blocks to be disregarded. To see why this is the case, consider in the absence of other errors the possible locations of bad windows created by the deletion. Letting us have a window size of w, we note that the possible range of bad windows is between w before to w after the location of the deletion. Noting that good windows are packed adjacent to each other, we note that this range covers at most (actually exactly) three good windows. Hence, overlaps caused by bad windows will cause us to disregard at most three good windows, per error (see Figure 3).

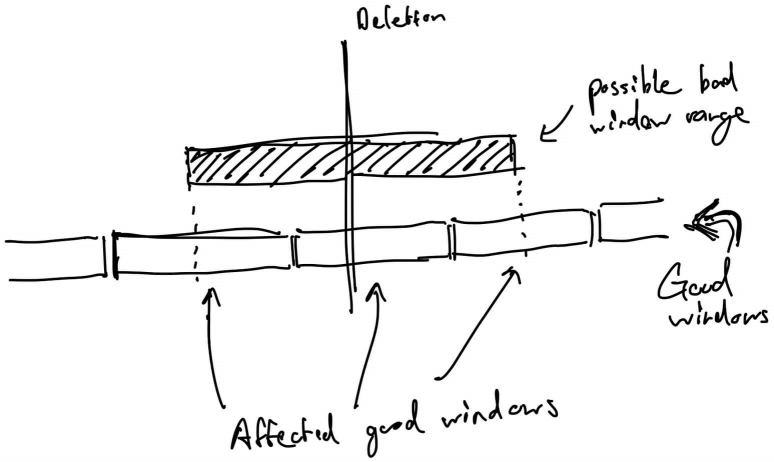

Figure 3: Interaction between bad windows and good windows

Next, note that after disregarding, we have at most t bad windows remaining. This is because each error can only contribute at most one bad window now. Else, if it contributed more than one bad window, these windows would overlap, and we would have disregarded them.

Finally, note that the t bad windows cause at most t wrong pre-images. Additionally, the disregarding of $3t$ good windows causes at most $3t$ wrong pre-images. When we move to a lower layer, these wrong pre-images and split and hashed to give $8t$ possibly wrong hashes. However, this number of errors is $O(t)$ and hence by our result on RS codes, can be corrected by sending $O(t)$ redundancies, and our proof is complete.

3.4 Generating a hash function

We conclude from the last subsection that it is sufficient for Alice to find and encode a hash function $h : \{0,1\}^{n/k} \to \{0,1\}^{|h|}$ such that h does not produce hash collisions on different windows[3] of h.

If $|h| < \log n - O(1)$ then by the pigeonhole principle, we are forced to have at least one pair of collisions between $O(n)$ distinct windows.

On the other hand, heuristically, if we selected h uniformly at random, then the pair $(h(x), h(y))$ is uniformly random for distinct x, y and thus the probability of a clash is $1/2^{|h|}$. By a union bound, the probability of a clash between any pair is at most $n^2/2^{|h|}$, and for $|h| \geq 3 \log n$ this goes to 0 as $n \to \infty$.

The main issue with this heuristic is that Alice has no way to share such a hash with Bob within the bandwidth limits, so we need some way to seed h with a relatively small number of bits such that $(h(x), h(y))$ is still uniformly random. Specifically, how many random bits do we need to seed h such that the bits of $\{h(x)\}_{x \in \{0,1\}}$ are (approximately) $2|h|$-independent?

[3]It should be noted here that we only require non-collision between distinct windows, since we cannot hash the same window content to different values. This does not matter in our context, because we are only trying to recover the content of the window and not the position of the window.

Theorem 9 (c.f. [1])
We can generate N bits such that every K bits has a distribution of TV distance at most ϵ away from the uniform distribution using at most

$$O\left(\log\log N + K + \log\frac{1}{\epsilon}\right) \text{ bits.}$$

In our setting, we set $N = |h| \cdot 2^{|h|}$ (total number of bits of h), $K = 2|h|$ (total number of bits in two hash values), then our hash requires $O(|h| + \log 1/\epsilon)$ bits, and the probability of $h(x)$ and $h(y)$ clashing is at most $1/2^{|h|} + \epsilon$, so we can set $\epsilon = 1/n^3$, $|h| = 3\log n$ to get the desired conclusion of a length $O(\log n)$ hash seeded by $O(\log n)$ bits.

Algorithmically, on Alice's end, she can check by brute force all possible seeds ($2^{O(\log n)}$ of them) in polynomial time to find one without hash collisions between windows of x.

3.5 Extensions

In [3], the authors consider a more sophisticated version of the IMS protocol with the following changes:

- They consider ε-self matching hash functions and and ε-synchronization hash functions.
- They use implicit buffers (Section 2) to break up the blocks instead of simply k blocks the same size (in the paper, these are called p-split points).
- Instead of halving the block size on each level, they split it into more parts.

The conclusion is that the authors of [3] were able to achieve $O(t \log n)$ with these improvements.

4 Syndrome Compression

Syndrome Compression, coined in [10], is effectively a procedure to bootstrap sketches and encoders. We show how this works in the document exchange setting.

Theorem 10 (adapted from [10])
Let sk be a sketch for length n documents corrupted by t deletions. Then for some absolute constant $C, D > 0$, there exists a sketch sk' for the same parameters whose length is

$$|\mathsf{sk}'| \le 4t\log n + \frac{C|\mathsf{sk}|}{\log|\mathsf{sk}|} + D.$$

The key idea a sketch must only distinguish between strings x and z only if t deletions can be applied on both to make them the same. Thus, we can pick a suitable a to "mod out by" so that the sketch still distinguishes them. (Note that the choice of a depends on x, so we should really write a_x, but we are suppressing this for readability.)

The "compressed" sketch function is, in fact, $\mathsf{sk}'(x) = (a, \mathsf{sk}(x) \mod a)$ for an apt choice of a (where we interpret $\mathsf{sk}(x)$ as an integer between 0 and $2^{|\mathsf{sk}|} - 1$ inclusive). We would like to avoid any collision between x, y if their edit distance is not more than $2t$, so we need $a \nmid f(x) - f(z)$ for all z in the edit ball of x.

Here we need a number theoretic fact:

Theorem 11 (c.f. [8])
There exists a constant $C > 0$ such that for sufficiently large n, the number of positive divisors of n is less than

$$2^{\frac{C\log n}{\log\log n}}.$$

Now we are ready to prove the syndrome compression bound.

Proof of Theorem 3. Recall that from the proof of Theorem 1, the edit ball has size

$$|\mathcal{B}_{\mathsf{edit}}(x)| \le 2n^{2t}.$$

Now, we do a quick count: for each z, $|f(x)-f(z)| \le 2^{|\mathsf{sk}|}$, so it has at most $2^{C\cdot\frac{|\mathsf{sk}|}{\log|\mathsf{sk}|}}$ divisors. Thus over all $z \in \mathcal{B}_t(x)$, there are at most $2^{C\cdot\frac{|\mathsf{sk}|}{\log|\mathsf{sk}|}} \cdot (2n^{2t})$ factors in total, which leaves the smallest possible a to be at most $2^{C\cdot\frac{|\mathsf{sk}|}{\log|\mathsf{sk}|}} \cdot (2n^{2t}) + 1$. Indeed,

$$\log\left(2^{C\cdot\frac{|\mathsf{sk}|}{\log|\mathsf{sk}|}} \cdot (2n^{2t}) + 1\right) \le 2t\log n + \frac{C\cdot|\mathsf{sk}|}{\log|\mathsf{sk}|} + 1 + o(1)$$

so transmitting both a and $\mathsf{sk}(x) \mod a$ will take twice that amount. □

In [10], the authors examine this only for the specific choice of a mixed string sketch (with some minor modifications). Plugging in $|\mathsf{sk}| = O(t^2 \log t \log n)$, we obtain that the compressed sketch satisfies

$$|\mathsf{sk}'| \le 4t\log n + O\left(\frac{t^2\log t\log n}{\log\log n}\right) = (4 + o_n(1))t\log n.$$

This means that we can obtain (asymptotically) an explicit code up to within twice of the GV bound, which is amazing.

4.1 Iterated Syndrome Compression

Here we consider the possibility of starting from a trivial sketch (e.g. the repetition code) and performing iterated syndrome compression to obtain a sketch of leading order $4t\log n$

Define sk_0 to be the repetition code: that is, sk_0 repeats each symbol in the input $(t+1)$ times (i.e. $|\mathsf{sk}_0| \le n(t+1)$). Now we define

$$\mathsf{sk}_{i+1} := \text{syndrome compressed version of } \mathsf{sk}_i$$

Hence, we can bound the length by defining a sequence satisfying the recurrence $L_0 = n(t+1)$ and

$$L_{i+1} = \frac{C\cdot L_i}{\log L_i} + (4t\log n + D)$$

Note that as a function of L_{i+1} as a function of L_1 is monotone decreasing, so assuming $L_0 \ge L_1$, we get that the entire sequence is monotone non-increasing. Fix $k, M > 0$. Then either $L_k \le 2^m$, or $L_0 \ge L_1 \ge ... \ge L_k > 2^M$, so

$$\begin{aligned}
L_k &\le (4t\log n + D) + \tfrac{C}{M}\cdot L_{k-1} \\
&\le ... \\
&\le (4t\log n + D)(1 + \tfrac{C}{M} + ... + (\tfrac{C}{M})^{k-1}) + (\tfrac{C}{M})^k L_0 \\
&\le (4t\log n + D)\left(1 - \frac{C}{M}\right)^{-1} + \left(\frac{C}{M}\right)^k L_0
\end{aligned}$$

Hence,

$$L_k \le \min\left\{(4t\log n + D)\left(1 - \frac{C}{M}\right)^{-1} + \left(\frac{C}{M}\right)^k L_0, 2^M\right\}$$

Previously we assumed that $L_0 = O_t(N)$, so we may pick $M = \omega_t(1) = o_t(\log\log n)$ and $k = \Omega_t(\log n)$ to get

$$L_k \le \min\{(4t\log n)(1 + o_t(1)) + o_t(1), (\log n)^{o_t(1)}\} = (4 + o_t(1))t\log n$$

which says that $|\mathsf{sk}_k|$ has the desired leading order term.

5 Discussion

All in all, the results give a satisfactory solution to efficient deletion coding in the regime of a constant number of errors. We were able to asymptotically get to within a factor of 2 of the best non-constructive construction, and a factor of 4 within the Hamming bound.

That said, we found these two issues to be prevalent within the papers that we read and reviewed:

- More often than not, there was unclear dependencies on t, so the results could not be safely applied in scenarios where t was not constant (e.g. in the $t = \Theta(\log n)$ regime). It would be interesting to rigorously extend these method to beyond the constant error regime and see how the conclusions evolve.
- Despite being efficient algorithms for constant time, the run-time is usually not polynomial in t, and this is due to the presence of some type of brute-force search over the deletion or edit ball. Can we design algorithms to encode and decode in time $\mathsf{poly}(n, t)$?

References

[1] N. Alon, O. Goldreich, J. Hastad, and R. Peralta. Simple construction of almost k-wise independent random variables. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 544–553 vol.2, 1990. doi: 10.1109/FSCS.1990.89575.

[2] J. Brakensiek, V. Guruswami, and S. 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.

[3] K. Cheng, Z. Jin, X. Li, and K. 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] S. Gao. *A New Algorithm for Decoding Reed-Solomon Codes*, pages 55–68. Springer US, Boston, MA, 2003. ISBN 978-1-4757-3789-9. doi: 10.1007/978-1-4757-3789-9_5. URL https://doi.org/10.1007/978-1-4757-3789-9_5.

[5] V. Guruswami and C. Wang. Deletion codes in the high-noise and high-rate regimes. *IEEE Transactions on Information Theory*, 63(4):1961–1970, 2017. doi: 10.1109/TIT.2017.2659765.

[6] U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1665–1676 vol. 3, 2005. doi: 10.1109/INFCOM.2005.1498448.

[7] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10(8), 1966. URL https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

[8] J.-L. Nicolas and G. Robin. Highly composite numbers by srinivasa ramanujan. *The Ramanujan Journal*, 1(2):119–153, 1997. doi: 10.1023/A:1009764017495. URL https://doi.org/10.1023/A:1009764017495.

[9] L. Schulman and D. Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory*, 45(7):2552–2557, 1999. doi: 10.1109/18.796406.

[10] J. Sima, R. Gabrys, and J. Bruck. Syndrome compression for optimal redundancy codes. In *IEEE International Symposium on Information Theory, ISIT 2020, Los Angeles, CA, USA, June 21-26, 2020*, pages 751–756. IEEE, 2020. doi: 10.1109/ISIT44484.2020.9174009. URL https://doi.org/10.1109/ISIT44484.2020.9174009.