Codes for Deletion and Insertion Channels with Segmented Errors

Zhenming Liu Michael Mitzenmacher

Abstract

We consider deletion channels and insertion channels under an additional segmentation assumption: the input consists of disjoint segments of b consecutive bits, with at most one error per segment. Under this assumption, we demonstrate simple and computationally efficient deterministic encoding and decoding schemes that achieve a high provable rate even under worst-case errors. We also consider more complex schemes that experimentally achieve higher rates under random error.

1 Introduction

Channels that allow deletions and insertions are remarkably challenging. For example, the capacity of the binary i.i.d. deletion channel, where n bits are sent and each bit is deleted with probability d, remains unknown, despite substantial recent progress [4, 5]. Even the case where n bits are sent and just one bit is deleted provides many interesting open problems [7]. While some attempts have been made to design coding schemes for such channels, the work has generally not led to provable performance guarantees and still seems far from optimal.

In this paper, we consider deletion and insertion channels under an additional *segmentation assumption* about the location of the errors. Specifically, we assume that the input is naturally grouped in consecutive segments of b consecutive bits, and there is at most one error in each segment. For example, if our segments consist of eight bits, and at most one deletion occurs per segment, on the input

$$0001011100101111,$$

which consists of two segments, it would be possible that the fourth and eleventh bits were deleted, so that the received sequence would be

$$00001110001111,$$

but *not* that last two bits were deleted, leaving

$$00010111001011.$$

We emphasize that the segments are implicit, and that no segment markers appear in the received sequence. Hence our problem *does not* immediately reduce to the simpler problem of fixing one deletion in a given b-bit block. Our goal is to develop efficient codes in this setting.

This additional assumption appears quite natural for many practical settings. Consider the case of disk drives, a commonly given example for synchronizations errors. Deletions may occur because of a timing mismatch between the device reading the data and the data layout. In such situations, there might naturally be a minimal gap between deletions, as the drift caused by the timing error may require several reading several additional bits before the timing error yields a further deletion. Our model encompasses the case where there is such a minimal gap, although it can also allow nearby deletions that cross a segment boundary. Our model would therefore also include settings such as when data is naturally written out in segments (e.g., bytes) by a writer that might erroneously delete a bit per segment, because of timing or other issues, and the reader must deal with the resulting bit sequence.

Another compelling motivation for considering channels with segmentation is the existing theoretical challenges in handling random or worst-case insertions and deletions. Considering channels with additional assumptions may yield insight into the more general problem.

We find that the segmentation assumption greatly simplifies the problem of dealing with insertions or deletions. Our primary result demonstrates a deterministic coding scheme inspired by the idea of prefix coding in compression. Our coding scheme allows for left-to-right decoding of a message, as long as a small amount of lookahead (corresponding to the next segment) is available. The scheme has provable performance guarantees under the segmentation assumption, even with adversarially chosen errors. As an example, with segments of eight bits (one byte), allowing up to one adversarial deletion per segment this scheme provides a code with a rate of 44.8%. The same result holds if we instead allow up to one adversarial insertion per segment. Our coding scheme is computationally simple and quite amenable to use in hardware. We believe the resulting transmission rates prove sufficiently high to be useful in practical settings.

We also consider extensions of our approach to give schemes that provide larger transmission rates under random errors, again with the assumption of at most one error per segment. The idea is to allow some ambiguity in decoding in order to increase the rate, and then incorporate check bits and checksums to resolve the ambiguities. Here our results are experimental, but as an example, again with segments of length one byte, we can achieve rates above 54% with very low error rates. Such schemes, however, also take additional computation time over our simpler schemes.

While our results are generally incomparable with previous results because of our additional assumptions, we note that previous experimental approaches to channels with insertions and deletions generally allowed much fewer errors with non-trivial block error rates [2, 3, 6]. Codes of rate 50% handling only deletions or insertions at a rate of 2 to 6 percent are typical. We believe the performance as well as the simplicity of our schemes represents a significant advance over previous work.

2 A Deterministic Approach

We first present a deterministic decoding scheme. We work over the binary channel, although our techniques can extend to larger alphabets. For ease of exposition we first consider deletions, and then consider insertions.

2.1 The Communication Model

Formally, our channel transmits binary streams of fixed length n, where n is known to the sender and receiver. We write the input as $X = x_1 x_2 \ldots x_n$. We use the notation $X(j,k)$ to refer to the substring $x_j x_{j+1} \ldots x_k$, and similarly for other bit sequences. For the *segmented deletion channel*, the received sequence $Y = y_1 y_2 \ldots y_m$ is obtained by deletion a number of bits from the input sequence, under the following condition: at most one bit from each set of bits $X(bi+1, b(i+1))$ can be deleted by the channel for $i = 0, \ldots, n/b - 1$. (For convenience we assume that b divides n evenly.) We use $s_i = X(bi+1, b(i+1))$ to refer to the bits constituting the ith segment in X, but we also abuse notation and use s_i to refer to the corresponding received bits in Y where the meaning is clear. We say the ith segment s_i starts at position y_ℓ if the first undeleted bit of the ith segment occurs at position y_ℓ. We emphasize that our scheme functions for *any* set of deletions satisfying the properties of the segmented deletion channel.

The case where $b = n$, so that there is just one segment and hence just one deletion, has been considered extensively [7]. Of particular interest is the class of Varshamov-Tenengolts codes, or VT codes [9]. The VT code $VT_a(n)$ consists of all binary vectors $x_1 x_2 \ldots x_n$ satisfying

$$\sum_{i=1}^{n} i x_i \equiv a (\mathrm{mod} n + 1).$$

With a VT code, any single deletion can be corrected without error. The codes $VT_0(n)$ are in fact optimal codes for n up to 9; see [7, 8] for more details.

2.2 Encoding and Decoding for Deletions

In order to explain the reasoning behind the choices made for our encoding and decoding schemes, we walk through step by step showing how the properties we require arise naturally by first principle considerations.

In our encoding scheme, each segment will consist of one of a set of a b-bit codewords C. We refer to C as a code, even though strictly speaking the code for this channel consists of a concatenation of segments with each coming from C. We use the same set C for every segment, although this is not a requirement of our approach. For $u \in C$, let $D_1(u)$ be the set of all $(b-1)$-bit strings that can be obtained by deleting one bit from u. We refer to $D_1(u)$ as the set of first order descendants of u, or just the descendants of u where the meaning is clear. This follows the notation used in [7]. We also use $D_1(C) = \cup_{u \in C} D_1(u)$.

The code C is said to be 1-deletion correcting if $D_1(u) \cap D_1(v) = \emptyset$ for all $u, v \in C$ with $u \neq v$. As mentioned previously, such codes are treated extensively in [7]. It is natural that we will want our code C to have this property.

To see why, we start to explain our decoding process. Our decoder will work from left-to-right, decoding one segment at a time. Decoding a segment will only require access to the next $2b - 1$ bits in sequence. Consider what might happen as we start from the left on the received sequence Y. The first $b - 1$ bits reveal the value of the first segment; indeed, in general, when C is 1-deletion correcting, if k is the starting position of a segment, then by examining bits $Y(k, k + b - 1)$, we can determine the codeword associated with the segment. But there may be some ambiguity as to whether a bit was deleted from the segment or not, so the decoder cannot determine whether to extract the first $b-1$ or first b bits. For example, if the segments are eight bits, and the first two segments are the strings 00000000 and 00001110, then if the received sequence began with 00000000001110, it would be a mistake to extract 8 bits for the first segment. (As 10 of the first 12 zeroes remain, we can see that one 0 was deleted from each segment.) Doing so would actually remove a bit from the subsequent segment. In general, we may not be sure whether the next segment starts at y_{b-1} or y_b. If we did not control this ambiguity, it could increase as we continue decoding; the third segment could conceivably start at y_{2b-2}, y_{2b-1}, or y_{2b}, and so on.

We therefore arrange our code so that this cannot happen. At each step, there will potentially remain some ambiguity; we maintain the invariant the next segment may start at one of at most two positions, y_k or y_{k+1}. Our code is arranged so that this slight ambiguity in the starting position of a segment does not lead to any ambiguity in the decoding of the segment. The ambiguity in position naturally resolves itself at the end of the received sequence.

Because our decoder works in this fashion, it is clear that we only need to consider how the decoder works *locally.* That is, given (Y, i, k) where Y is the received string, i is the segment to be decoded, k is starting position such that the ith segment must start in position k or $k + 1$, we wish to decode the ith segment and determine an appropriate new position k' such that $(i + 1)$st segment starts at k' or $k' + 1$. We can then iterate through Y to recover X. (It should be clear in what follows that at some points in our algorithm we may have no ambiguity, so that we know the ith segment must start in some position k. The algorithm could be optimized for such situations. We do not consider such optimizations here, as they do not affect our analysis.)

Suppose that we have segment s_i starting at position k. There are two cases to consider.

- Case 1: There is no deletion s_i. In this case, the segment ends at y_{k+b-1}, and $Y(k + b, k + 2b - 2)$ is in $D_1(s_{i+1})$.
- Case 2: There is exactly one deletion in s_i. In this case, the segment ends at y_{k+b-2}, and $Y(k + b - 1, k + 2b - 3)$ is in $D_1(s_{i+1})$.

Optimistically, we might hope that by restricting our codebook we can determine which case holds at each point, in which case we can decode segment by segment with no ambiguity. The following provides an equivalent way of viewing this restriction. For a string x of length

$k > 1$, let prefix(x) be the first $k-1$ bits of x, and similarly define suffix(x) be the last $k-1$ bits of x. For a set S of strings let prefix$(S) = \cup_{x \in S}$prefix(x) and define suffix(S) similarly. Then for our code C we can require that for all $u, v \in C$ with $u \neq v$,

$$\text{prefix}(D_1(u)) \cap \text{suffix}(D_1(v)) = \emptyset.$$

In Case 1, we have $Y(k+b, k+2b-3) \in \text{prefix}(D_1(C))$, and in Case 2, we have $Y(k+b, k+2b-3) \in \text{suffix}(D_1(C))$.

It seems that we have chosen our code so that we can distinguish Case 1 and Case 2, but this is not quite the case. The problem is the bits $Y_{k+b,k+2b-3}$ can indeed be in both prefix$(D_1(C))$ and suffix$(D_1(C))$; they simply cannot be in prefix$(D_1(u))$ and suffix$(D_1(v))$ for some $u \neq v$ in our code. There is nothing, however, that prevents these bits from being in both prefix$(D_1(u))$ and suffix$(D_1(u))$ for some $u \in C$. Moreover, this specific ambiguity seems unavoidable; for any $u \in C$, if we delete the first and last bit, we obtain a subsequence that is both in prefix$(D_1(u))$ and suffix$(D_1(u))$.

Notice, though, that under this restriction, the bits $Y(k+b, k+2b-3)$ do determine the segment s_{i+1}; that is, there is not ambiguity in what the next segment is, just where it starts and begins. By restricting our codewords slightly further, we can guarantee that this ambiguity does not increase from step to step. We prove this now. In the statement of the theorem, x^* is standard regular expression notation, corresponding to 0 or more copies of the string x.

Theorem 2.1. *Consider the segmented deletion channel with segment length b. Let C be a subset of $\{0,1\}^b$ with the following properties:*

- *for any $u, v \in \mathcal{C}$, with $u \neq v$, $D_1(u) \cap D_1(v) = \emptyset$;*
- *for any $u, v \in \mathcal{C}$, with $u \neq v$, prefix$\big(D_1(u)\big) \bigcap$ suffix$\big(D_1(v)\big) = \emptyset$;*
- *any string of the form $a^*(ba)^*$ or $a^*(ba)^*b$, where $a, b \in \{0,1\}$, is not in C.*

Then, using C as the code for each segment, there exists a linear time decoding scheme for the segmented deletion channel that looks ahead only $O(b)$ bits to decode each block.

Proof. We follow the outline of our discussion. We decode segment by segment, with the invariant that when decoding the ith segment, we know it starts either at position k or position $k+1$ in Y. The possible ending positions of the ith segment are y_{k+b-2}, y_{k+b-1}, or y_{k+b}. We must eliminate either the first or third possibility to maintain our invariant, and we must recover the ith segment.

We consider two cases. The simple case is when only one of $Y(k, k+b-2)$ and $Y(k+1, k+b-1)$ is in $D_1(C)$. For example, if $Y(k+1, k+b-1) \notin D_1(C)$, then the ith segment cannot start at y_{k+1} and must start at y_k. In this case we can determine s_i from $Y(k, k+b-2)$ and the next segment starts either at y_{k+b-1} or y_{k+b}. The argument is similar if $Y(k, k+b-2) \notin D_1(C)$.

Now suppose instead that both of $Y(k, k+b-2)$ and $Y(k+1, k+b-1)$ are in $D_1(C)$. Then $Y(k+1, k+b-2) \in \text{suffix}(D_1(C))$ and $Y(k+1, k+b-2) \in \text{prefix}(D_1(C))$. By assumption, we cannot have two distinct codewords $u, v \in C$ with $Y(k+1, k+b-2) \in \text{suffix}(D_1(u))$ and $Y(k+1, k+b-2) \in \text{prefix}(D_1(v))$, so the bits $Y(k+1, k+b-2)$ determine the segment s_i.

We now show using our final assumption on the codewords that the next segment starts either at y_{k+b-1} or y_{k+b} (but not y_{k+b+1}). Assume the next segment starts at y_{k+b+1}. Then s_i must be the subsequence $Y(k+1, k+b)$. Further, as $Y(k, k+b-2) \in D_1(s_i)$, we have that there exists j with $k-1 \leq j \leq k+b-2$ and a bit z such that

$$y_k y_{k+1} \ldots y_j z y_{j+1} \ldots y_{k+b-2} = y_{k+1} \ldots y_{k+b}. \tag{1}$$

(When $j = k-1$, the left hand side is $z y_k y_{k+1} \ldots y_{k+b-2}$.) Comparing bit by bit, we have

$$\begin{array}{rcl} y_k & = & y_{k+1} \\ y_{k+1} & = & y_{k+2} \\ .. & .. & .. \\ y_j & = & y_{j+1} \\ z & = & y_{j+2} \\ y_{j+1} & = & y_{j+3} \\ .. & .. & .. \\ y_{k+b-2} & = & y_{k+b}. \end{array}$$

But then s_i is of the form $a^*(ba)^*$ or $a^*(ba)^*b$, contradicting our assumption. $\square$

The restriction on C to exclude certain strings is an unfortunate byproduct of our approach. We emphasize, however, that of the 2^b possible codewords, only $O(b)$ of them are initially excluded. Hence we would expect that the this restriction would not dramatically reduce the possible size of the code.

Given these restrictions, finding a valid C for a given segment size b corresponds naturally to an independent set problem, similar to those for 1-bit deletion codes [7]. We take the underlying graph where there is a vertex for each possible codeword, and two codewords are connected by an edge if they cannot simultaneously be in the code according to our restrictions. A valid code corresponds to an independent set on this graph, and we therefore seek a maximum independent set. For small b this can be done by exhaustive calculation, and for larger b heuristic techniques can be used to find large codes. In general, proving optimality for such independent set problems can be difficult; related results appear in [1, 8].

We have exhaustively checked to find optimal codes for $b = 8$ and 9, shown in Figures 1 and 2. When $b = 8$, so that segments are bytes, the (unique) optimal code contains 12 codewords, corresponding to a rate of slightly more than 44.8%. It is worth noting that even if segment markers were given at the receiving end, and an optimal 1-deletion correcting code is used per segment, the maximal such code has only 30 codewords [7], corresponding to a rate of slightly more than 61.3%. Our rate of 44.8% is over over 73% of this benchmark. For $b = 9$ we found 28 different codes consisting of 20 codewords. Hence for $b = 9$ the rate is over 48%; comparing to the 52 codewords for an optimal 1-deletion correcting code for one segment, our codes achieves over 75% of this rate. We conjecture that the rates for

00100100
00101011
01110000
01110011
01111100
01111111
10000000
10000011
10001100
10001111
11010100
11011011

Figure 1: An optimal code for $b = 8$ with our deterministic scheme.

000011100
000011111
000100011
000100100
010111011
010111100
011001111
011010011
011010100
011111111
100000000
100101011
100101100
100110000
101000011
101000100
111011011
111011100
111100000
111100011

Figure 2: An optimal code for $b = 9$ with our deterministic scheme.

optimal codes satisfying the conditions of Theorem 2.1 increase with b. We would also like for the ratio between the size of these codes and the optimal 1-deletion correcting codes to increase with b, and for both these ratios to converge to 1, but these conjectures may be too optimistic.

The inherent limitations of exhaustive search prevents us from finding optimal codes for larger values of b. Indeed, [8] reports on the difficulties of finding independent sets for similar graphs arising from coding problems. Nevertheless, we find that using simple randomized greedy heuristics yields codes with good rates. For example, when $b = 16$, so segments are two bytes, we have found a code with 740 codewords, giving a rate of approximately 59.57%, by using a simple greedy strategy: repeatedly choose a remaining element of minimal degree, and delete the element and all of its neighbors from the graph.

Our decoding algorithm is particularly amenable to hardware implementation. One possible implementation (in pseudocode) is given as procedure LOCAL-DECODE in Figure 3. Each membership check could be performed by a lookup table, as could the DECODE operation, which decodes sequences to obtain a segment value. While the rates grow larger as b increases, the computational problem of finding a code grows, as do the corresponding size of the lookup tables.

LOCAL-DECODE(Y : string, k, i : integers)
if $Y(k, k+b-2) \in D_1(C)$ and $Y(k+1, k+b-1) \notin D_1(C)$
    then $s_i \leftarrow \text{DECODE}\big(Y(k, k+b-2)\big)$ and $k' \leftarrow k+b-1$
        return (s_i, k');
if $Y(k, k+b-2) \notin D_1(C)$ and $Y(k+1, k+b-1) \in D_1(C)$
    then $s_i \leftarrow \text{DECODE}\big(Y(k+1, k+b-1)\big)$ and $k' \leftarrow k+b$
        return (s_i, k');
$\triangleright$ the final case implies $Y(k, k+b-2) \in D_1(C)$ and $Y(k+1, k+b-1) \in D_1(C)$
$s_i \leftarrow \text{DECODE}\big(Y(k, k+b-2)\big)$ and $k' = k+b-1$.
return (s_i, k').

Figure 3: A decoding algorithm based on local decoding.

For larger values of b, the lookup tables can be avoided, at the cost of more computation and perhaps some loss of rate. Specifically, the class of VT codes provide an example of 1-deletion correcting codes with a simple decoding algorithms [7]. If one restricts oneself to a code that is a subset of a VT code meeting the required conditions, then one can use the decoding mechanism for VT codes in place of lookup operations. Subsets of VT codes have the further advantage that they are smaller than the entire set of possible codewords, making the search for appropriate maximal independent sets that yield codes easier. On the other hand, restricting oneself to subsets of VT codes will generally reduce the rate.

2.3 Encoding and Decoding for Insertions

Our approach works entirely similarly for the *segmented insertion channel*. In this model, the channel transmits a binary stream of fixed length n, given by $X = x_1x_2 \ldots x_n$. The received sequence $Y = y_1y_2 \ldots y_m$ is obtained by inserting a number of bits into the input sequence, under the following condition: at most one bit is added in each segment of bits $X(bi+1, b(i+1))$ for $i = 0, \ldots, n/b - 1$. The bit can be inserted before or after any bit in the sequence. (Note that under this model we can have two bits inserted in a row, but only on either side of a segment boundary.)

As before, under our encoding scheme, each segment will consist of one of a fixed set of a b-bit codewords C. Paralleling our previous notation, let $I_1(u)$ be the set of all $(b+1)$-bit strings that can be obtained by inserting one bit into u, and $I_1(C) = \cup_{u \in C} I_1(u)$. The code C is 1-insertion correcting if $I_1(u) \cap I_1(v) = \emptyset$ for all $u, v \in C$ with $u \neq v$. Note that the VT codes are 1-insertion correcting as well as 1-deletion correcting.

We first show the corresponding version of Theorem 2.1 modified for insertion channels. We then prove something more subtle: our resulting codes for segmented insertion channels and segmented deletion channels are entirely the same.

Theorem 2.2. *Consider the segmented insertion channel with segment length b. Let C be a subset of $\{0,1\}^b$ with the following properties:*

- *for any $u, v \in \mathcal{C}$, with $u \neq v$, $I_1(u) \cap I_1(v) = \emptyset$;*
- *for any $u, v \in \mathcal{C}$, with $u \neq v$, prefix$\big(I_1(u)\big) \bigcap$ suffix$\big(I_1(v)\big) = \emptyset$;*
- *any string of the form $a^*(ba)^*$ or $a^*(ba)^*b$, where $a, b \in \{0,1\}$, is not in C.*

Then, using C as the code for each segment, there exists a linear time decoding scheme for the segmented insertion channel that looks ahead only $O(b)$ bits to decode each block.

Proof. The proof follows the same pattern as Theorem 2.1. We decode segment by segment, with the invariant that when decoding the ith segment, we know it starts either at position k or position $k+1$ in Y. The possible ending positions of the ith segment are y_{k+b-1}, y_{k+b}, or y_{k+b+1}. We must eliminate either the first or third possibility to maintain our invariant, and we must recover the ith segment.

As before, the simple case is when only one of $Y(k, k+b)$ and $Y(k+1, k+b+1)$ is in $I_1(C)$. In this case we can determine s_i and the two possible starting points of the next segment.

If instead both $Y(k, k+b)$ and $Y(k+1, k+b+1)$ are in $I_1(C)$, then $Y(k+1, k+b) \in$ suffix$(I_1(C))$ and $Y(k+1, k+b) \in$ prefix$(I_1(C))$. These bits determine the segment s_i. Our additional assumption on the codewords of C will suffice to bound the ambiguity at the next step.

Assume the next segment starts at y_{k+b}. Then s_i must be the subsequence $Y(k, k+b-1)$. Further, as $Y(k+1, k+b+1) \in I_1(s_i)$, we have that there exists j with $k+1 \leq j \leq k+b+1$ such that

$$y_{k+1}y_{k+2} \cdots y_{j-1}y_{j+1} \cdots y_{k+n+1} = y_ky_{k+1} \cdots y_{k+b-1}. \tag{2}$$

(When $j = k-1$, the left hand side is $zy_ky_{k+1}...y_{k+n-2}$.) Comparing bit by bit, we have

$$\begin{aligned} y_k &= y_{k+1} \\ y_{k+1} &= y_{k+2} \\ .. &\quad .. \quad .. \\ y_{j-2} &= y_{j-1} \\ y_{j-1} &= y_{j+1} \\ y_j &= y_{j+2} \\ .. &\quad .. \quad .. \\ y_{k+b-1} &= y_{k+b+1}. \end{aligned}$$

But then s_i is of the form $a^*(ba)^*$ or $a^*(ba)^*b$, contradicting our assumption. $\square$

Theorem 2.2 shows that we can solve a similar independent set problem to find codes for the segmented insertion channel. In fact, however, the codes obtained under Theorem 2.1 and Theorem 2.2 are actually the same. To demonstrate this requires the following straightforward lemma:

Lemma 2.3. *For $u \neq v$,*

$$D_1(u) \cap D_1(v) = \emptyset \leftrightarrow I_1(u) \cap I_1(v) = \emptyset, \tag{3}$$

and

$$\mathit{prefix}\big(D_1(u)\big) \bigcap \mathit{suffix}\big(D_1(v)\big) = \emptyset \leftrightarrow \mathit{prefix}\big(I_1(u)\big) \bigcap \mathit{suffix}\big(I_1(v)\big) = \emptyset. \tag{4}$$

Note that, from this lemma, we have that the conditions of Theorem 2.1 and Theorem 2.2 are in fact equivalent, and hence a code derived by Theorem 2.1 for the segmented deletion channel would also be suitable for the segmented insertion channel (and vice versa).

Proof. Let $u = u_1u_2\dots u_n$ and $v = v_1v_2\dots v_n$. For (3), we have that if $D_1(u) \cap D_1(v) \neq \emptyset$, then there exist positions i, j with $i \neq j$ such that

$$u_1u_2\dots u_{i-1}u_{i+1}\dots u_n = v_1v_2\dots v_{j-1}v_{j+1}\dots v_n.$$

Without loss of generality let $i < j$. It follows that

$$u_1u_2\dots u_{i-1}u_iu_{i+1}\dots u_jv_ju_{j+1}\dots u_n = v_1v_2\dots v_{i-1}u_iv_i\dots v_{j-1}v_jv_{j+1}\dots v_n,$$

and hence $I_1(u) \cap I_1(v) \neq \emptyset$. The argument is entirely similar in the other direction. For (4), if $\text{prefix}(D_1(u)) \cap \text{suffix}(D_1(v)) \neq \emptyset$, then then there exist positions i, j such that

$$u_1u_2\dots u_{i-1}u_{i+1}\dots u_{n-1} = v_2\dots v_{j-1}v_{j+1}\dots v_n.$$

If $i \neq j$ again it follows (assuming $i < j$) that

$$u_1\dots u_{i-1}u_iu_{i+1}\dots u_jv_ju_{j+1}\dots u_{n-1} = v_2v_3\dots v_{i-1}u_iv_i\dots v_{j-1}v_jv_{j+1}\dots v_n.$$

The case where $i = j$ follows similarly, as does the other direction of the equivalence. $\square$

We emphasize that the fact that 1-deletion codes are also 1-insertion codes, and more generally that codes that correct up to k deletions also correct up to k insertions, appears to well-known in this area; however, this alone does not imply Lemma 2.3.

3 Higher rates via more complex parsing

Our work to this point demonstrates that the segmentation assumption is useful for deriving deterministic, worst-case decoding schemes. In this section we expand upon this idea by considering a generalization of our deterministic approach to a randomized approach designed to obtain higher rates under the segmentation assumption. For simplicity, we describe only the case of deletions, although similar ideas would hold for insertions as well. Also, we emphasize that there are no provable performance guarantees for these codes, and we suspect they could be optimized further. This effort should therefore be seen as a proof of concept that allowing further controlled ambiguity may lead to better practical codes.

As before, we denote the input by X and the output by Y. As performance depends on channel behavior, we parametrize by having each segment lose a bit with probability p; the bit lost in each segment is chosen independently and uniformly at random. Also, similar to our previous scheme, our encoding uses a fixed 1-deletion correcting code C for each segment. Under the segmentation assumption, there is a natural recursive way of expressing whether a received string Y can be successfully decoded into a message. Let decode(k,t) be True if there is some decoding of the first k bits into t segments consistent with the segmentation assumption, and False otherwise. Naturally decode$(0,0)$ is True and decode$(i,0)$ is False if $i \neq 0$. Then

$$\text{decode}(k,t) = \text{ OR } \begin{cases} (\text{decode}(k-b,t-1) \text{ AND } (Y(k-b+1,k) \in C)) \\ (\text{decode}(k-b+1,t-1) \text{ AND } (Y(k-b+2,k) \in D_1(C))). \end{cases}$$

This recursion naturally suggests a decoding algorithm: find all parsings of the received string that decodes into n/b segments consistent with the recursion above. Such recursions naturally arise in dynamic programming algorithms, and our decoding algorithm uses a dynamic programming framework. The problem is that there may be many consistent parsings that lead to multiple valid decodings; we have ambiguities to cope with when both cases of the OR occur. Indeed, our deterministic scheme can be seen in this framework: we choose C so that there is no ambiguity in terms of the codeword for each segment, and the ambiguity in position is limited so strongly as to allow left-to-right decoding.

To achieve higher rates than the deterministic scheme, we allow more ambiguities to occur by using a larger 1-deletion correcting code C, and then expend more computational effort to remove them subsequently by using check bits. At a high level, an underlying open question is how well one can trade off increasing rate and increasing computation with such an approach.

3.1 Encoding

In the encoding phase, we start with a message string M of length $|M|$, which is converted into the input string X of length n via a sequence of transformations. First, one or more checksums is added. A standard one byte checksum was obtained by taking the exclusive-or of the bytes of M (padded as necessary). We also in some experiments used a second checksum by taking the exclusive-or of M when broken into 11-bit blocks. The checksums invalidate many of the parsings found when decoding. This gives us a new string M_1.

Second, check bits are added. We have two types of parity check bits: global and local. Each global parity check bit is the exclusive-or of s bits of M_1 chosen randomly (with replacement). The purpose of these check bits is primarily to remove any remaining incorrect parsings when decoding. There are g global check bits concatenated to the end of M_1.

There are also l local parity check bits. Each local check bit is determined by a position w; the local check bit is placed between the wth and $(w+1)$st bits of M_1, and it too is exclusive-or of s bits of M_1 chosen randomly (with replacement), with the restriction that each of these s bits is one of the first w bits of M_1. (If two local parity check bits have the same position w, they are simply placed sequentially between the wth and $(w+1)$st bits of M_1.) Thus a local check bit depends only on the prefix of bits of M_1 before it. As we describe below, local check bits allow us to reduce the number of potentially valid parsings of M_1 as we parse the received string, improving the computational complexity of the decoding.

We emphasize that the random choices of positions and bits is considered part of the code and is shared information between the sender and receiver. After the global and local check bits are inserted, giving an intermediate string M_2, we map blocks of M_2 into blocks of X, where each block of X consists of several segments using codewords from the 1-deletion correcting code C. That is, we find k_1 and k_2 such that 2^{k_1} is slightly smaller than $|C|^{k_2}$ and take blocks of k_1 bits of M_2 to obtain k_2 segments of X determined by some appropriate fixed mapping.

3.2 Decoding

We describe a multi-pass decoder that takes the received string $Y = y_1 y_2 \dots y_m$ and reconstructs the original message.

In the first pass, the decoder determines the valid parsings of the received string, using a dynamic programming framework. From this, we can design a second pass that finds all possible valid codewords for the ith segment, for every i. For convenience, we say that each segment has a bucket of possible strings. We note that it is certainly possible that some $c_i \in C$ corresponds to the ith segment under multiple parsings, and hence this bucketing of possible segment strings is helpful in the next step of the decoding. Also, to be clear, for a codeword c_i to be valid for the ith segment, there must be some parsing of Y into n/b segments for which the ith segment is c_i. This approach is accomplished via for example standard trellis constructions and we therefore do not describe our specific implementation in detail.

In the final pass, we scan the buckets from left to right. The decoder maintains a set S_i of potentially valid prefixes of i segments of the intermediate string M_2. (Initially, S_0 contains the empty string.) If there is just one string in the ith bucket, then S_i is obtained from S_{i-1} by concatenating that string with every element of S_{i-1}. If there is more than one string in the ith bucket, then one could simply take all possible concatenations of strings in S_{i-1} with strings in the ith bucket. However, we have found the following approach more efficient.

When there is more than one string in the ith bucket, we find the next bucket j that has just one string. We then find all valid decodings from segment i to segment j, and compute S_j through the appropriate concatenation of strings with S_{i-1}. Normally, j is not much larger than i; in our experience when $b = 8$, $j - i$ is usually less than 6. Decoding blocks of buckets in this way significantly decreases the size of the prefix set S_j over simply sequentially concatenating all possibilities when there are multiple strings in consecutive buckets.

Because the set S_i may expand quickly with i as we read through the buckets from left to right, we use local parity check bits to cut down the size of S_i as we go. Whenever we reach a segment with a local check bit, we can throw out from S_i any string that does not appropriately match the parity check. Strictly speaking, we could avoid local check bits and only use global check bits. In practice, however, we have found that this dramatically slows the decoding because of the speed with which the S_i grow. Analyzing this tradeoff is an open question.

Finally, after all the buckets are read, we can use the global parity check bits and the checksums to reduce the size of the final set $S_{n/b}$. The decoder is successful if the size of this set is reduced to 1. In this case, we have successfully found M_2 without any ambiguity, from which point we can obtain the message M.

3.3 Experimental Results

We present experimental results demonstrating that this approach of allowing greater ambiguity but requiring more complex parsing of the received sequence can yield higher rates in practice. The gains are moderate but far from trivial. In return, the approach requires substantially more computation.

Our experiments were run with 1600 bits consisting of 8 bit segments being sent through the channel. We used one eight-bit checksum or an eight-bit and elevent-bit checksum. We also used 40 local check bits and varied the number of global check bits in order to demonstrate how they affect the probability of successful decoding. Each check bit had degree 97, a somewhat high number but one we found worked well in experiments. (There is nothing special about 97; similar numbers in that range would yield similar performance.) We note that we found that not using local check bits led to extremely large and highly variable numbers of parsings, making decoding far too expensive in terms of computation time. Increasing the length of the message would therefore naturally require proportionally additional local check bits to keep the number of parsings manageable throughout the decoding process. The remaining bits were message bits, the number chosen to yield 1600 bit packets. Our codes utilized 25 codewords, and blocks of 37 bits from the message (and check bits) were mapped to 64 bits (8 codewords), as 2^{37} is less than 25^8. A more efficient encoding here would slightly improve the rate.

Recall that in this setting successful decoding results in a single possible input; unsuccessful decoding results in a list of multiple possible inputs, and in this sense can be seen as a list decoding for the received sequence. In this setting, the list decoding gives all possible concatenations of codewords consistent with the received sequence and the segmentation assumption. Technically, if the decoding process became too slow, because at some intermediate point of the decoding there were too many potentially valid prefixes, the process should terminate with an unsuccessful decoding that returns no codeword. With our choices of parameters, we avoided this in our experiments.

In our experiments, we varied the probability each segment deleted a uniformly chosen bit from $p = 0.1$ to $p = 0.9$, varied the number of global check bits from 0 to 20, and varied whether we used one checksum (8 bits) or two (8 and 11 bits). (We did not attempt to optimize the number of local or global check bits as p varied; rather, we tried to find a single scheme effective acoss the entire range of p.) Our experimental results are presented in Figures 4 and Figures 5. Each table entry is the result of 1000 trials. Our results demonstrate that higher rates (above 54%) are possible with low error rates using this technique. Indeed, when using two checksums, we saw no errors using these parameters.

Anecdotally, in our experiments we found that both success rates and efficiency are best when the probability a segment is in error is small, because the smaller segment error rate generally leads to a smaller number of possible parsings. We tested this further by taking 1000 trials with $p = 0.1$ and $p = 0.2$, using only five local check bits, zero global check bits, and one checksum. We obtained no errors under these settings, with small increases in the computation required. While we expect further experimentation and optimization could slightly improve these results, overall these experiments amply demonstrate that non-trivial rate gains over our deterministic approach are certainly possible by allowing increased ambiguity.

global bits	rate	segment error rate								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
0	54.8%	100%	100%	99.9%	100%	99.9%	99.7%	99.9%	100%	99.8%
10	54.2%	100%	100%	100%	100%	100%	100%	100%	100%	100%
20	53.5%	100%	100%	100%	100%	100%	100%	100%	100%	100%
average ($\max S_i$)		1.6	5.3	45.4	156.1	336.9	561.6	736.0	1001.6	1507.9

Figure 4: Performance chart for sample codes when the number of local check bits is 40 and there is only one 8 bit checksum. Each row (except the last) indicates experiments with a fixed number of global check bits. All entries (except the last row) indicate the success rate over 1000 trials. The last row gives the average of $\max S_i$ over all of the trials at that segment error rate.

global bits	rate	segment error rate								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
0	54.1%	100%	100%	100%	100%	100%	100%	100%	100%	100%
10	53.5%	100%	100%	100%	100%	100%	100%	100%	100%	100%
20	52.9%	100%	100%	100%	100%	100%	100%	100%	100%	100%
average ($\max S_i$)		1.6	5.6	34.6	146.8	338.3	550.1	686.8	852.8	700.5

Figure 5: Performance chart for sample codes when the number of local check bits is 40 and there is one 8-bit checksum and one 11-bit checksum. Each row (except the last) indicates experiments with a fixed number of global check bits. All entries (except the last row) indicate the success rate over 1000 trials. The last row gives the average of $\max S_i$ over all of the trials at that segment error rate.

4 Conclusion

We have introduced the segmented deletion channel and the segmented insertion channel, new variations of insertion/deletion models motivated by timing considerations. We have demonstrated that one can develop codebooks that allow for greedy left-to-right decoding for these segmented channels, based on controlling the inherent ambiguity in these channels. We have shown that such codes can achieve relatively high rates even under adversarial errors satisfying the segmentation condition. Our approach is sufficiently general that it should be applicable to similar channels.

We have further considered what we think is the natural extension of this approach to achieve higher rates under less severe, non-adversarial conditions. Namely, we allow more ambiguity in the decoding process, using dynamic programming and local check bits to control the number of possible parsings and global check bits to constrain the result to a single possible decoding. Our initial work is promising but leaves open questions in both how to analyze and design such schemes. Alternatively, there may be other means of controlling ambiguity that allow for efficient decoding in the same spirit as our approach.

References

[1] S. Butenko, P. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. Finding maximum independent sets in graphs arising from coding theory. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pp. 542-546, 2002.

[2] J. Chen, M. Mitzenmacher, C. Ng, and N. Varnica. Concatenated codes for deletion channels. In *Proceedings of the IEEE International Symposium on Information Theory*, p. 218, 2003.

[3] M.C. Davey and D.J.C. MacKay. Reliable Communication over Channels with Insertions, Deletions, and Substitutions. *IEEE Transactions on Information Theory*, volume 47, number 2, pp. 687-698, 2001.

[4] E. Drinea and M. Mitzenmacher. On Lower Bounds for the Capacity of Deletion Channels. *IEEE Transactions on Information Theory*, volume 52, number 10, pp. 4648-4657, 2006.

[5] M. Mitzenmacher and E. Drinea. A Simple Lower Bound for the Capacity of the Deletion Channel. *IEEE Transactions on Information Theory*, volume 52, number 10, pp. 4657-4660, 2006.

[6] E. Ratzer. Marker codes for channels with insertions and deletions. *Annals of Telecommunications*, 60:1-2, p. 29-44, January-February 2005.

[7] N. Sloane. On Single-Deletion-Correcting Codes. Arxiv preprint math.CO/0207197, 2002.

[8] N. Sloane. Challenge Problems: Independent Sets in Graphs. At http://www.research.att.com/∼njas/doc/graphs.html.

[9] R. R. Varshamov and G. M. Tenengolts. Codes which correct single asymmetric errors (in Russian), *Automatika i Telemekhanika*, 26:2, pp. 288-292, 1965. English translation in *Automation and Remote Control*, 26:2, pp. 286-290, 1965.