Verification Codes for Deletions

Michael Mitzenmacher

Abstract

We demonstrate new deletion codes based on the verification code paradigm introduced in [5]. Verification codes for deletions (VCDs) are low-density parity-check codes that handle packet-level deletions, where in this context a packet is simply a collection of bits. VCDs have polynomial time encoding and decoding algorithms and can be analyzed asymptotically using standard LDPC techniques. VCDs are also robust against transpositions, random insertions, and random substitution errors.

1 Introduction

In a deletion channel, symbols are sent over a channel, but the receiver may obtain only a subset of the symbols sent. For convenience, we will think of the symbols as being sent in packets, with one symbol per packet. In the case where packets (or the symbols themselves) are designed to include sequence numbers, a deletion channel is equivalent to an erasure channel, as the receiver in this case knows which packets are missing. An interesting question is what is achievable without sequence numbers. Although channels that can insert, delete, and substitute symbols are practically interesting in the context of synchronization, codes for deletion channels have historically received less attention than error-correcting or erasure codes.

Here, we demonstrate new deletion codes based on low-density parity-check (LDPC) codes using the verification code framework, introduced in [5]. Verification codes are designed to work with data organized as packets. For example, packets could be thousands of bits, as is the case with Internet packets, or a packet could simply represent a thirty-two bit integer. Deletions are assumed to occur at the packet level; that is, an entire packet disappears. Verification codes for deletions (VCDs) have polynomial time encoding and decoding algorithms and can be analyzed asymptotically using standard LDPC techniques. VCDs are also robust against transpositions, random insertions, and random substitution errors. For lack of space we save a detailed analysis of these issues for the full paper and focus here on analyzing deletions.

As suggested above, packet-level deletions can be handled using sequence numbers and an erasure code. This is very effective, particularly if packets are large; hence our results may only be of theoretical interest. However, for small packets (e.g., 64 bits), sequence numbers may be expensive in terms of bits required, making VCDs a viable alternative. Also, VCDs robustness to random substitution and insertion errors may offer further advantages over erasure codes with sequence numbers.

After briefly describing previous work, we present verification codes for the case of where the data sent is random and each packet is deleted independently with probability $p$. We then describe generalizations of this approach to situations where data are not random, or random insertions and substitutions can occur.

*Harvard University, Division of Engineering and Applied Sciences, 33 Oxford St., Cambridge, MA 02138. mihai@eecs.harvard.edu Supported in part by an Alfred P. Sloan Research Fellowship and NSF grants CCR-9983832, CCR-0118701, and CCR-0121154.
1.1 Previous Work

Deletion codes, although not as well studied as erasure and error-correcting codes, have a long history; see [13, 3, 2, 12] for details and references. More recently, the case of detecting a single deletion is covered in a recent survey of Sloane [13]. Diggavi and Grosgeiuser consider variations of Shannon’s theorem to prove bounds on the capacity of a channel that deletes bits independently with probability \( p \) [3, 4]. Davey and MacKay design codes for channels that can insert, delete, or substitute bits, using various approaches to determine the probability of the value for each sent bit and applying a low-density parity-check code on the end results [2].

The verification code framework utilizes low-density parity-check codes in the setting where packet values lead to a false verification with suitably small probability [5]. Earlier ideas similar to verification codes were suggested by Metzner [1, 9].

2 Framework for Low-Density Parity-Check Codes

We briefly summarize the now standard framework for low-density parity-check (LDPC) codes, following [8]. LDPC codes are easily represented by bipartite graphs. The variable nodes correspond to symbols in the codeword. The check nodes correspond to constraints on the adjacent variable nodes. We assume henceforth that there are \( n \) variable nodes and \( m \) check nodes. The design rate \( R \) is given by \( R = \frac{n-m}{n} \). In the case of general alphabets with \( q \) symbols, we assume that the symbols are interpreted as numbers modulo \( q \), and that the constraints represent constraints on the sum of the variable nodes modulo \( q \). In the case where the alphabet consists of the \( q = 2^b \) strings of \( b \) bits, the constraints are generally taken to be parity check constraints. That is, the sum operation is a bitwise exclusive-or instead of a modular addition.

A family of codes can be determined by assigning degree distributions to the variable and check nodes. In regular LDPC codes all variable nodes have the same degree, and all check nodes have the same degree. More flexibility can be gained by using irregular codes, where the degrees of each set of nodes can vary. The idea of using irregular codes was introduced in [7, 8]. We associate with each degree distribution a vector. Let \( (\lambda_2, \ldots, \lambda_{d_v}) \) be the vector such that the fraction of edges connected to variable nodes of degree \( i \) is \( \lambda_i \). (We assume a minimum degree of two throughout.) Here \( d_v \) is the maximum degree of a variable node. Similarly, let \( (\rho_2, \ldots, \rho_{d_c}) \) be such that the fraction of edges connected to check nodes of degree \( i \) is \( \rho_i \), and \( d_c \) is the maximum degree of a check node. Based on these degree sequences, we define the polynomials \( \lambda(x) := \sum_{i=2}^{d_v} \lambda_i x^{i-1} \) and \( \rho(x) := \sum_{i=2}^{d_c} \rho_i x^{i-1} \), which prove useful in subsequent analysis. The \( \lambda_i \) and \( \rho_i \) variables must satisfy a constraint so that the number of edges is the same on both sides of the bipartite graph. This constraint is easily specified in terms of the design rate by the equation \( R = 1 - \int_0^1 \frac{\rho(x)dx}{\int_0^1 \lambda(x)dx} \). Once degrees have been chosen for each node (so that the total degree of the check nodes and the variable nodes are equal), a specific random code can be chosen by mapping the edge connections of the variable nodes to the edge connections of the check nodes, as described for example in [7, 8].

Without loss of generality we assume henceforth that the constraints are such that the sum of the symbols associated with the variable nodes adjacent to the \( i \)th check node is \( i \). This is a 

\[ \text{The actual rate} \ R \ \text{tends to be slightly higher than the design rate} \ R \ \text{in practice, because the check nodes are not necessarily all linearly independent. This causes at most a vanishingly small difference as} \ n \ \text{gets large, so we ignore this distinction henceforth.} \]
deviation from the standard LDPC approach, where the constraints are generally that the sums are 0; this deviation will prove useful subsequently. For this setting, the linear time encoding schemes of [11] still apply.

We consider message-passing algorithms below. To determine the asymptotic performance of such codes, it suffices to consider the case where the neighborhood of each node is a tree for some number of levels. That is, there are no cycles in the neighborhood around each node. The martingale arguments relating the idealized tree model and actual graphs was first applied to coding in [6] and is now standard; see for example [6, 7, 8, 10].

3 Random errors and random deletions

To present the ideas behind verification codes for deletions, we consider the simplest possible case. We assume that the data being sent is random. That is, a subset of the variable nodes carry symbols chosen independently and uniformly at random from the alphabet of size \( q \); the rest of the symbols are chosen so as to satisfy the constraints. (See [11] for more details.) Also each packet is deleted with constant probability \( p \), independent of other packets.

We now describe an algorithm PlaceNodes. For convenience, we describe PlaceNodes for the case of regular graphs, where the degrees of variable and check nodes are \( d_v \) and \( d_c \), respectively; it generalizes to irregular graphs trivially. Throughout the algorithm PlaceNodes, each check node is in one of two possible states: unverified and verified. When a node is unverified, the algorithm has not yet fixed the final value for that node. Hence the decoding algorithm begins with all nodes being unverified. When a node is verified, it is given a fixed value for the remainder of the algorithm. Hence the algorithm PlaceNodes should, with high probability, never assign a verified node an incorrect value.

Suppose that \( a \) out of \( n \) node values arrive. The decoding algorithm begins by considering the exclusive-or of all \( \binom{n}{d_c} \) combinations of \( d_c \) nodes that arrived. The decoding algorithm then applies the following rules in any order as much as possible:

1. If there exists a combination of \( d_c \) variable nodes whose sum equals \( i \), then associate these nodes with the neighbors of the \( i \)th check node. That is, label all of the neighbors of the \( i \)th check node as verified, and assign the value of the \( j \)th neighbor of this node to be the \( j \)th variable node in order of arrival in the associated combination.

2. If all but one of the neighbors of the \( i \)th check node is verified, the remaining neighbor becomes verified, with its final value being set so that the sum of all neighbors of the check node equals \( i \).

In this decoding process the check nodes play two roles. First, they may verify that the appropriate position of a collection of nodes is correct, according to first rule. This verification rule applies because of the following fact: if the sum of the values of a collection of \( d_c \) nodes is \( i \), then with high probability these nodes must be the neighbors of check node \( i \). Concretely, we have the following lemma:

**Lemma 1** The probability that any set of \( d_c \) variable nodes not associated with the \( i \)th check node sum to \( i \) is at most \( 1/q \). Hence the probability of a false verification at any point in the decoding process is at most \( \binom{n}{d_c} / q \).

**Proof:** Consider any set of \( d_c \) variable nodes containing at least one variable node not associated with \( i \)th check node. If the \( d_c \) nodes are associated with some other check node, then their sum is
not \( i \) by definition. Otherwise, there is one variable node \( z \) that is linearly independent from the other \( d_c - 1 \) variable nodes. That is, viewing the variable nodes initially assigned random values (corresponding to the message) as variables and the other variable nodes as linear combination of these variables, \( z \) is linearly independent of the \( d_c - 1 \) other nodes of the set. For these \( d_c \) nodes to sum to \( i \), given the value for the other \( d_c - 1 \) variable nodes, \( z \) must take on the precise value that makes the sum 0. The probability of this is \( 1/q \). The second part of the lemma then follows by a trivial union bound. \( \square \)

To see the value of the above lemma, consider the case where symbols consist of \( b \) bits, so \( q = 2^b \). Thus the probability of a failure from a false verification is exponentially small in \( b \) at each step. Hence, if \( d_c \) is constant, the packet size \( b \) needs to be only \( \Omega(\log n) \) bits in order that the probability of failure due to a false verification be polynomially small in \( n \). In practice \( b = 64 \) bits will generally suffice.

The other role of a check node is to correct a neighboring variable node that was received incorrectly, according to Rule 2. A check node can correct a neighbor after all other neighbors have been verified and therefore are known to have the correct value with high probability.

3.1 A message-passing decoding algorithm

The algorithm above is easily modified to a message-passing algorithm, which allows us to apply standard analysis techniques. The goal of our analysis is to determine the asymptotic threshold fraction of deletions tolerable under our decoding process as \( n \) grows large, denoted by \( p^* \). Consider an edge \((v, c)\) between a variable node \( v \) and a check node \( c \), and the associated tree describing the neighborhood of \( v \). (Recall that we assume that the neighborhood of \( v \) is accurately described by a tree for some fixed number of levels.) The tree is rooted at \( v \), and the tree branches out from the check nodes of \( v \) excluding \( c \), as shown in Figure 1.

The decoding process occurs in rounds, with each round having two stages. In the first stage, each variable node passes to each neighboring check node in parallel its current state and, if verified, a value. In the second stage, if possible \( c' \neq c \) that neighbors \( v \) verifies \( v \) and provides it an appropriate value if possible; if this occurs, \( v \) changes its value and state appropriately. (For convenience in the analysis, we think of each variable node as passing on to the check node \( c \) the current value excluding any information obtained directly from \( c \). This avoids the problem of a circular flow of information, where messages from \( c \) later affect messages sent to \( c \).)

To analyze this process, let \( a_j \) be the probability that in round \( j \), \( v \) is unverified, although \( v \) successfully arrived. Similarly let \( b_j \) be the probability that in round \( j \), \( v \) is unverified and the variable for \( v \) was initially deleted. Hence \( 1 - a_j - b_j \) is the probability that in round \( j \), \( v \) can confirm to \( c \) that it has been verified via another check node. We ignore the possibility in
the analysis that a false verification occurs, since as we have already argued, for a sufficiently large alphabet this occurs with negligible probability. (Technically we can simply condition on a false verification not happening.) Initially $a_0$ is simply $1 - p$ and $b_0 = p$. If $a_j + b_j$ tends to 0, then our decoding algorithm will be successful, since then the probability that a node remains unverified falls to 0.

The evolution of the process from round to round, assuming that the neighborhood of $v$ is given by a tree, is given by:

$$a_{j+1} = a_0 \lambda (1 - \rho (1 - b_j)), \quad (1)$$
$$b_{j+1} = b_0 \lambda (1 - \rho (1 - a_j - b_j)). \quad (2)$$

We explain the derivation of equation (2). A node $v$ that was deleted and remains unverified in the $(j + 1)$st round was initially deleted with probability $b_0$. Also, it cannot be the case that there is some check node $c'$ other than $c$ neighboring $v$ that has all of its children verified after $j$ rounds, or else $v$ could be corrected and verified for the $(j + 1)$st round. Now each $c'$ has $k - 1$ children below it with probability $\rho_k$, and each child is verified after $j$ rounds with probability $1 - a_j - b_j$. The probability that $v$ has not been corrected due to a specific check node $c'$ by round $j$ is therefore

$$\sum_i \rho_i (1 - a_j - b_j)^{i-1} = \rho (1 - a_j - b_j).$$

As $v$ has $k - 1$ other neighboring check nodes besides $c$ with probability $\lambda_k$, the probability that $v$ remains unverified in round $j + 1$ is

$$\sum_i \lambda_i (1 - \rho (1 - a_j - b_j))^{i-1} = \lambda (1 - \rho (1 - a_j - b_j)).$$

This yields equation (2); equation (1) is derived by similar considerations.

It should be noted at this point that the analysis above is entirely similar to the original analysis for verification codes for errors. A deletion corresponds to an error; an undeleted symbol that has not yet been placed corresponds to an erroneous symbol that has not yet been corrected. Based on this correspondence, we can conclude following the argument from [5],

**Theorem 1** For any rate $R$, with $0 < R < 1$, and a given $\epsilon > 0$, there exists a family of verification deletion codes of rate $R$ that correct a $1 - R - \frac{\sqrt{R - 3R^2}}{2} - \epsilon$ fraction of errors with high probability.

The theorem simply applies the irregular graph parameters for good erasure codes given in [7]. Such codes are not necessarily optimal here; better codes can be found through non-linear optimization. An interesting open question is to find a family of better parameters that improve the above theorem.

Note that the decoding can be done in polynomial time, assuming that $d_c$ is a constant. Initially, $O(n^{d_c})$ work must be done to consider all relevant combinations of packets. After that, whenever the status of a variable node is changed the corresponding check node must see whether it can send further information on to other packets. The entire decoding time is $O(n^{d_c})$. For Theorem 1, however, requires irregular coding schemes with very large constant maximum degree $d_c = O(\ln \frac{1}{\epsilon})$.

As a perhaps more appealing example, applying equations (1) and (2) to a 3-6 regular LDPC code yields a VCD code of rate 1/2 that runs in $O(n^6)$ time and can handle over 13% of the symbols being deleted.
4 Improvements and Variations

The running time of VCD codes in the case of random deletions as above can be improved by not considering all $O(n^{d_c})$ combinations of $d_c$ symbols. Each received packet can be associated with its expected position in the sent stream of packets, and it will lie within $O(\sqrt{n \log n})$ packets of its expected position with all but polynomially small probability. Taking advantage of this can reduce the decoding time to $O(n^{d_c/2} \log(n))$.

The code as described works only if the data is random. Non-random data can be made to appear random if the sender and receiver share a source of (pseudo)-randomness; simply XOR the data with an agreed (pseudo)-random string.

The codes above are also robust against small amounts of packet-reordering (transpositions), as long as packets associated with every check node are received in the correct order; variations to handle arbitrary re-ordering also exist and will be described in the full paper. Similarly, the codes are also robust against random substitution errors or insertions of random packet data, as such errors or insertions maintain the property that a false verification does not occur with high probability.

References