

An Introduction to Coding Theory ¹

Dustin Boswell

May 31, 2001

¹based on “Algorithmic issues in coding theory” by Madhu Sudan, an invited paper to the *Conference on Foundations of Software Technology and Theoretical Computer Science*, Kharagpur, India, 1997

1 The Problem

Suppose you have some message to send to a receiving party, but along the way a portion of the message will be altered in random places. What can you do to ensure that the other end will get the exact message? It is easy to see that naive attempts such as duplicating the message or otherwise encoding such redundancy will not work since the “noise” of transmission occurs randomly, and in proportion to the length of the message. The same issue arises in data storage, when a portion of the information is expected to ‘decay’ or otherwise be lost. Coding theory (and ‘error-correcting codes’) is the field of solving such problems.

2 Notation

Suppose we have some message m which consists of k letters $m_1m_2\dots m_k$ each from a fixed alphabet Σ . Before sending the message over the noisy channel, we first encode the message as $C(m)$, where C is some encoding function we have chosen. The encoded message $C(m)$ is of length $n \geq k$, (but still on the same alphabet). After transmission we get the received message R , still of length n . Suppose that sending the message resulted in up to t of the n letters being altered. From R (and knowing the coding scheme C), we would like to be able to determine, with certainty, which message m must have been sent.

We refer to an encoding $C(m)$ as a *codeword*. Since typically $n > k$, the set of codewords is a strict subset of the possible strings of length n . We refer to the set of codewords as \mathbf{C} . Our goal is to come up with some \mathbf{C} such that any two codewords $C(m_1), C(m_2) \in \mathbf{C}$ are not “close” to each other. That way, given R and the number of errors t , we can deduce exactly which codeword $C(m)$ was sent. And from this, we can use a decoding function to get m .

3 Hamming Distance

The typical measure of how “close” two (equal-length) strings are, is their Hamming Distance $\Delta(m_1, m_2)$, which is simply the number of places where they differ. This is obviously a metric. $\Delta(m_1, m_1) = 0$. $\Delta(m_1, m_2) = \Delta(m_2, m_1)$. And it is easy to convince yourself that $\Delta(m_1, m_2) + \Delta(m_2, m_3) \leq \Delta(m_1, m_3)$. (Given a series of alterations from m_1 to m_2 , and from m_2 to

m_3 , this is in turn a way to change m_1 to m_3 , and does so in $\Delta(m_1, m_2) + \Delta(m_2, m_3)$ steps.)

We use the Hamming Distance to measure the distance d of a code C . Define

$$d(\mathbf{C}) = \min_{c_1 \neq c_2 \in \mathbf{C}} \Delta(c_1, c_2) \quad (1)$$

which tells us that it takes at least d alterations to get from one codeword to another.

If we are using a coding system with distance d , how many errors t can an encoded message withstand and still be uniquely decoded? Well, if we are given a received word R , which has had at most t errors, then the sent codeword $C(m)$ must be within a Hamming Distance of t from R . And ambiguity occurs when there are two such code words both a distance of t or less. But since the Hamming Distance is a metric, we use the triangle inequality to know that the distance from one codeword to the received word, plus the distance from the received word to any other codeword, must be less than or equal to the distance between the codewords. And that distance is bounded by $d(\mathbf{C})$. Thus there can be roughly $d/2$ errors until decoding becomes ambiguous. More precisely, in the case where $t \leq \lfloor (d-1)/2 \rfloor$, we can determine which $C(m)$ was sent. Once again, if there were two codewords $C(m_1), C(m_2)$ which were both a distance t ($= \lfloor (d-1)/2 \rfloor$) or less from R , this would imply $\Delta(C(m_1), C(m_2)) < d$ which contradicts the distance of C being d in the first place.

4 Example: Dustin's Simple Code

00 – 01000
01 – 10100
10 – 00011
11 – 11111

As shown, this code takes binary messages of length 2, and encodes them to length 5. We claim that this code has a distance of 3. For example to get from the first codeword to the second requires flipping the first three bits. And it is true that for any other pair, it also requires changing at least 3 bits.

Now if a code has a distance of d , then that means it should be able to correct $t = \lfloor (d-1)/2 \rfloor$ errors. And for our simple code, $t = 1$. So it is the case that if we send any of those codewords across and change at most one

of the bits, the receiver will know for a fact which codeword it came from, and hence what the original message was.

5 $[n, k, d]_q$ notation:

In general we refer to a coding scheme as a $[n, k, d]_q$ code. Here k is the size of our original message, n is the size the message gets inflated to before sending it across, d is the distance of the code, and q is the size of the alphabet we're dealing with ($|\Sigma| = q$). Our 'simple code' was a $[5, 2, 3]_2$ code.

In designing error correcting codes there's this tradeoff between d and n . We want the ratio d/n (called the 'distance rate') to be high, so it can withstand a high error rate, but we also want the ratio k/n ('message rate') to be high, which is to say we don't want to have to blow up our message of length k to an exorbitant size n before sending it.

Our goal is to have our code system be asymptotically good, which just means that as n goes to infinity, the ratios k/n and d/n don't go to 0. Those are often hard to find, however, so sometimes we settle for "weakly good" codes, which have a constant distance rate, but a polynomial message rate (more on this later).

Notice that this notion of being asymptotically good doesn't apply to our 'simple code' because it is designed for a fixed k and a fixed n . Typically, a coding system will work for any size message. However, it's not immediately clear why we even care that a code works for arbitrary size messages, since we can (naively) just break it up into smaller messages.

The main reason is that for a noisy channel we are not guaranteed that the errors are evenly distributed. Say for example, we have a noisy channel with transmission error of $p_e = 1/5 = .2$. And say we have a message that is 10-bits long. Now we decide to use our 'simple code' and break up the 10 bits into 5, 2-bit chunks. Notice that the total encoded message will be 25 bits long, and will have an expected 5 bits that are changed (due to error). If those 5 bits of error occur in the first 5 bits, for example, then the 2 bits encoded in that 5 bit codeword is completely lost.

The point is that breaking up a large message into smaller chunks and then encoding those chunks does not preserve the distance of the original code. That is, $c * [n, k, d]_q \neq [cn, ck, cd]_q$. If we had a $[25, 10, 15]_2$ code to begin with, then that code would have been able to withstand the 1/5 error rate, no matter where the errors occurred.

6 Practical Issues to Keep in Mind

6.1 Encoding

Given a message, how to we get its corresponding code word? Obviously, it's just a mapping from strings of length k to a subset of strings of length n . So we could easily just make a table like we did for the 'simple code'. But there are 2^k possible messages, and each has a corresponding n size codeword. So doing a table lookup (although it's $O(n)$ time), would take $O(n2^k)$ space. One goal, therefore is to have an efficient encoding mechanism.

6.2 Decoding

The naive method to decode would be to just go through all the code words and measure its Hamming Distance from the received word R , and taking the one whose distance is less than t from it. This however requires exponential time. Thus, we hope to find efficient decoding mechanisms (which will turn out to be a difficult problem).

7 Linear Codes

A set of codewords \mathbf{C} compose a *linear code* if

$$\forall a \in \Sigma, x, y \in \mathbf{C} \quad x + y \in \mathbf{C}, a \cdot x \in \mathbf{C}$$

That is, if we think of codewords as vectors in n dimensions, the set of codewords form a vector space. Since $n > k$, codewords are sparse in this n dimensional space. That is, $\mathbf{C} \subset \Sigma^n$. In fact, we should be able to represent any codeword as a linear combination of the basis vectors for that subspace \mathbf{C} . This key insight allows us to use the concept of a *generator matrix*. If we construct such a matrix G , whose columns are the basis vectors for the codewords, then multiplying G by some "coefficient column vector" m , we will get some codeword. And so we can just define our message m to be this column vector. Each message, when pre-multiplied by G will result in a codeword $C(m)$ which we take as our encoding of m .

It is a fact that for every linear code, there is a generator matrix G of size $n \times k$ such that $C(m) = G \cdot m \forall m$. This is a good thing, because it means that encoding a message only takes $O(nk)$ time. Also, it is true that for every such code, there is a parity check matrix H , that is of size $(n - k)$ by n . This matrix allows you to check if a received message R of length n

is a codeword or not. This is done by checking if $H \cdot R = 0$. If it is, then it is a codeword, otherwise not.

In addition to efficient encoding, generator matrices allow for a compact representation of the coding system. That is, when referring to a linear code, rather than specifying all the codewords, we can just refer to the generator matrix (which is size $O(nkq)$ compared to the exhaustive table of size $O(nq^k)$).

8 Hadamard Code

The Hadamard Code is a non-linear code over a binary alphabet, whose codewords are the set of rows from a Hadamard Matrix.

A Hadamard Matrix is an n by n matrix whose entries are $+1$ and -1 so that $MM^T = nI_n$. Equivalently, it turns out that if you take any two rows or columns of the matrix then they have a Hamming Distance of exactly $n/2$. This would imply that the set of codewords comprising the rows of the matrix has a distance of $n/2$. Let us prove this important fact:

Consider the matrix $D = MM^T = nI_n$ in the definition. Notice the element D_{ij} corresponds to the dot product of the i^{th} row of M with the j^{th} column of M^T . But the j^{th} column of M^T is just the j^{th} row of M . Thus D_{ij} is the dot product of the i^{th} and j^{th} rows of M . Now consider the case when $i \neq j$ (two distinct codewords). Since $D = nI_n$, D_{ij} must be 0. In taking the dot product of two rows, multiplying corresponding elements results in either $+1$ or -1 for a given location in the row. That is, since each element is either $+1$ or -1 in the first place, the product is $+1$ if the elements (from the two rows) agree, and -1 if they don't. Let a be the number of places that the two rows agree, and let d be the number of places they disagree. The dot product D_{ij} is then $(+1)a + (-1)d$, which we said is 0. So $a = d$. And since there are n elements in a row, then $a + d = n$. Thus we have $a = d = n/2$, which is to say they agree (disagree) in exactly $n/2$ places.

Obviously, n must be even if a Hadamard matrix is to exist, although it is not sure whether every even n has a corresponding Hadamard Matrix ¹. Typically, we let n be a power of 2, so $N = 2^L$. A given message of length L , over the binary alphabet, can take 2^L possible values. And so we just map each possible message to a row of the corresponding n by n Hadamard Matrix. This results in a $[n, L, n/2]_2$ code, where $n = 2^L$. To encode a binary message m , we can just takes its numeric value (e.g. $m = 101$ has a

¹Simple construction mechanisms are known for when n is a power of 2.

numeric value of 5) as a pointer to which row of the Hadamard Matrix to use as the corresponding codeword.

Notice this code has a distance rate of $d/n = \frac{n/2}{n} = 1/2$ which is constant (a good thing). However, the message rate ($k/n = L/n = L/2^L$) goes to zero as n becomes large (and does so exponentially).

9 Reed Solomon Codes

The Reed Solomon Code is a wonderful linear code which makes use of the number theory behind finite fields. Whereas normally, our alphabet size ($q = |\Sigma|$) is fixed ahead of time (often binary), now we require that it be at least the size of our encoded message ($q \geq n$). This odd restriction will become clear in a moment.

We first define x_0, \dots, x_{n-1} to be some fixed enumeration of n of the elements of Σ . We typically pick some *generator*² g , and let $x_i = g^i$. The point is that $x_0 \neq x_1 \neq \dots \neq x_{n-1}$, (which could only make sense if $q \geq n$).

Now, given a message $m = m_1 \dots m_k$, we build a $k - 1$ degree polynomial $M(x) = \sum_{j=0}^{k-1} m_j(x)^j$. Don't confuse the x that is the variable of the polynomial, with the enumeration x_0, \dots, x_{n-1} . At this point, they have nothing do to with each other. $M(x)$ is simply a polynomial in x , and the enumeration is simply a set of numbers.

Given our message m and the corresponding polynomial $M(x)$ that we just built, we now go and encode m . (Notice that $M(x)$ was by no means an encoding of m , just simply a tool that we will now use to encode m .) To do so, we evaluate the polynomial $M(x)$ at n different points. Which n points? The n points we chose as our enumeration earlier. More explicitly, we let $C(m) = c_0 \dots c_{n-1}$, where $c_i = M(x_i)$.

Now there are q^k possible messages m (since each letter can take q values, and there are k letters). Also, there are q^k many $k - 1$ degree polynomials (since there are k coefficients, and each coefficient can take q values). And the way we've defined the construction of polynomials from messages results in each message getting its own polynomial. That is, there is a one to one mapping. Now polynomials form a vector space (adding them together results in a polynomial of the same degree, and multiplying a polynomial by a constant and results in a same degree polynomial). This means that the Reed Solomon Code is a *linear* code, and more importantly, that it has a generator matrix. Indeed it does, and in fact the matrix representation of

²Recall that a generator g (every finite field has one) can be used to obtain all the elements from a field by simply taking increasing powers of g .

encoding perhaps makes it more clear what is going on when you encode a message (see the figure).

What is the distance of a Reed Solomon Code? If we have two messages $m1 \neq m2$, what is the closest (in terms of Hamming Distance) that their encodings $C(m1)$ and $C(m2)$ could be? Let $M1(x)$ and $M2(x)$ be the corresponding polynomials for messages $m1$ and $m2$.

An equivalent way to state the question is to ask how many of the elements of $C(m1) - C(m2)$ are non-zero? (Remember that we would like $C(m1)$ and $C(m2)$ to be as different as possible, since this means the code has a large distance.) Notice that although there are two different polynomials for $m1$ and $m2$, both of those polynomials get evaluated at the same n points (the enumerated points we set ahead of time). That is, if we let $C(p) = c_0 \cdots c_{n-1} = C(m1) - C(m2)$, we see that each c_i is simply $M1(x_i) - M2(x_i)$. Once again, at each element of the difference vector between the two codewords $(C(p))_i = C(m1)_i - C(m2)_i$, it is simply the difference between two polynomials evaluated at the same point $(C(p))_i = M1(x_i) - M2(x_i)$. (See the figure for a more visual representation.) The difference between two polynomials is also a polynomial. If the preceding was a bit unclear the thing to remember is this: $C(m1) - C(m2)$ (the difference between two codewords, which we are currently interested in) is actually just another polynomial $P = M1 - M2$, evaluated at n different points. Now, we wanted to know how many of these elements are non-zero. By the Fundamental Theorem of Algebra, we know that there are at most $k - 1$ roots of a degree $k - 1$ polynomial. And since P is a such a polynomial, we conclude that at most $k - 1$ of the elements of $C(m1) - C(m2)$ are 0. This implies that at least $n - k + 1$ of the elements of $C(m1)$ and $C(m2)$ are different.

The Reed Solomon Code is thus a $[n, k, n - k + 1]_q$ code, for $q \geq n \geq k$. The beauty of this code is that we can pick n and k to be most anything in accordance to our needs. In other codes, such as the Hadamard Code, the relation of k to n is fixed. Nevertheless, if we chose $k = n/2$ for example, we see that the Reed Solomon Code has both a constant message rate, and a constant distance rate (this is a great thing). However, it has the awkward restriction that the alphabet grow with the message size. Practically this is not such an issue. For n up to roughly 4 billion, we can accomplish this with a 32 bit alphabet (a 32 bit word is typical for a computer, and so $q = 2^{32}$). Theoretically speaking, however we would like codes to be defined for a fixed alphabet. That is, the fact that Reed Solomon Codes are 'asymptotically good' is in a sense cheating. However, there are ways to somewhat circumvent this.

10 Concatenation

Idea: Can we somehow combine two codes to take advantage of the strengths of both? Yes!

Suppose we have a code $\mathbf{C}_1 = [n_1, k_1, d_1]_{q_1}$, which takes k_1 letters, each from Σ_1 (where $|\Sigma_1| = q_1$), and outputs n_1 letters, also on Σ_1 . And now we wish to combine this with another code $\mathbf{C}_2 = [n_2, k_2, d_2]_{q_2}$. We can do this so long as the (yet unclear) restriction $q_1 = q_2^{k_2}$ is met.

Encoding proceeds as follows: given some message m which has k_1 letters, each from Σ_1 , we first encode the message with $C_1()$ as normal. Each of the letters in $C(m)$ is also on Σ_1 . But since $q_1 = q_2^{k_2}$, we can think of each letter from $C(m)$, as a *message* which has k_2 letters on Σ_2 . We then encode each of those messages (there are n_1 messages), with $C_2()$. Since each message is encoded to a length of n_2 , we now have n_1 messages of length n_2 , all on Σ_2 . We simply concatenate those n_1 messages to get one $n_1 n_2$ length message over Σ_2 .

This concatenation of codes has taken a length k_1 message over Σ_1 and output-ed a length $n_1 n_2$ message over Σ_2 . We *could* have thought of this as taking a $k_1 k_2$ message over Σ_2 . To see this, remember $q_1 = q_2^{k_2}$, so each letter on Σ_1 can be thought of as a k_2 long message over Σ_2 . Although implementation-ally, this is not the case, we *could* have thought of it this way, and algebraically, we use this convention so that the concatenation $C_1 \cdot C_2 = [n_1 n_2, k_1 k_2, d']_{q_2}$ for some d' .

Now we will show that $d' \geq d_1 d_2$. Recall that our first step was to use C_1 to encode the letters of m . Since d_1 is the distance of C_1 , then any two different messages $m_1 \neq m_2$ will have at least d_1 different letters in the encodings $C(m_1), C(m_2)$. Also recall that the letters from the encoding $C(m)$ is then taken as a *message* for C_2 . Thus, the next step of encoding the n_1 messages will have started off with at least d_1 different messages (encoding m_1 vs. m_2). And each of those different messages are encoded with C_2 which has a distance of d_2 , which means that at least d_2 of the letters will be different of the different messages after the second step of encoding. That is, at least $d_1 d_2$ letters will be different. (This is an algebraically obtuse thing to explain, so the reader is urged to look over the diagram and work things through.)

As a specific example of why concatenation is a useful thing, consider concatenating Reed Solomon Codes with Hadamard Codes.

Recall that a Reed Solomon Code is a $[n, k, n - k + 1]_q$ code in general. But say we let $n = q = 2^m$ for some m . And let $k = 0.4n$. So we have a $[n, 0.4n, .6n + 1]_{2^m}$ code.

Recall that a Hadamard Code is a $[2^L, L, 2^{L-1}]_2$ code. Now if we correspond $L = m$, we get a $[2^m, m, 2^{m-1}]_2 = [n, \lg(n), n/2]_2$ code.

Combining the two results in a $[n^2, 0.4n \lg(n), 0.3n^2 + .5n]_2$ code. This code has a constant distance rate ($\lim_{n \rightarrow \infty} d/n = .3$). And although it does not have a constant message rate, (the limit still goes to 0), it is a “polynomial” message rate, which is much better than the exponential message rate of the Hadamard Code by itself. And in addition, the code is now defined over a binary alphabet, which removes the odd restriction imposed by Reed Solomon Codes of having a growing alphabet.

11 Complexity Issues

11.1 Maximum Likelihood Decoding

Given a generator matrix G , and a received word R , we would like to know which codeword is the closest to R in terms of Hamming Distance. This is the codeword which is most likely to have been sent before the errors occurred, and in decoding, would be taken as such. This simply stated question is actually quite difficult. In fact, it was proven to be NP-Hard for general input G and R .

This casts a dark shadow for coding theorists. But there are two possible ways to get around it. One way is to hope that there exists a decoding algorithm for specific linear codes (such as Reed Solomon Codes). That is, although we cannot guarantee an efficient decoding algorithm for any linear code there is, it might be the case that such efficient algorithms exists for some of them. And in fact it is the case. Nevertheless, we would still like to be able to say something about decoding in general. And so our other attempt is to limit the distance away from R in which we search for a codeword.

11.2 Bounded Distance Decoding

Suppose that in addition to G and R we are also given a restriction t , and that our task now is to find *all* codewords within a distance t of R . Maybe this modified problem will be simpler (i.e. not NP-Hard) than Maximum Likelihood Decoding. Unfortunately, it is inherently just as difficult as the later. The reason is that there is no restriction on t . For example, what if the t that is specified is greater than d , the distance of the code? In this case, finding all the codewords within a distance t of R is actually *more* work

than the Maximum Likelihood problem. Thus it seems sensible to place a restriction on t .

11.3 Fractional Distance Decoding

Suppose that we are given G , R , and some $t = \epsilon d$ (for $0 < \epsilon < 1$). And now (like above) we are asked to find all the codewords with t distance of R . Is this problem any easier? The problem with it is that we don't know d ahead of time. That is, for a generator matrix G , it is not obvious what the distance of the underlying code is. And so we need to determine d before we can attempt this problem.

11.4 Minimum Distance of a Code

Suppose we are given G , and d and are asked "is the minimum distance of the code specified by $G \geq d$?" This decision problem was recently proved to be Co-NP Complete. But don't let that get you down! Maybe we don't need to know what d is exactly. Maybe if we know what d is within some tolerable range of error, then we can proceed with a Fractional Distance Decoding algorithm (hoping one exists), by adjusting ϵ to withstand that margin of error.

11.5 Approximate Distance of a Code

Given G , we would like to know some estimate d' such that $d' = d \pm \alpha(n)d$. If had an efficient algorithm to do this, and an efficient algorithm to solve the Fractional Distance Decoding algorithm, then we would have a reasonable method for decoding arbitrary linear codes. This is currently an open problem.

12 Conclusion

Although polynomial-time decoding algorithms for general linear codes doesn't look promising, much work has been done on specific codes. For example, the Reed Solomon Codes have poly-time decoding algorithms (which are somewhat complicated). In addition, there are a set of "expander codes" created by Sipser and Spielman, which have encoding and (bounded distance) decoding algorithms that take *linear* time.

Coding theory is used in many applications. Hadamard Codes have been used by the Mariner space probe sending information back to Earth. Reed

Solomon Codes (among a *vast* number of other uses) are the standard for encoding CD's and DVD's. For example, you can scratch over a 1000 bits in a row on a CD, and it will still read it perfectly! Much of the success of digital storage and communication is due to coding systems like Reed Solomon Codes.