Princeton Univ. F'20 COS 521: Advanced Algorithm Design

Lecture 23: Coding Theory and Error Correcting Codes

Lecturer: *Sahil Singla* Last updated: November 19, 2020

Computer and information systems are prone to data loss—lost packets, crashed or corrupted hard drives, noisy transmissions, etc.—and it is important to prevent actual loss of important information when this happens. Today's lecture concerns *error correcting codes*, a stepping point to many other ideas, including a big research area (usually based in EE departments) called *information theory*. This area started with a landmark paper by Claude Shannon in 1948, whose key insight was that data transmission is possible despite noise and errors if the data is *encoded* in some redundant way and moreover, that we can quantify exactly how much redundancy is required for a given amount of noise.

1 Basic Setup

Our basic setup for the lecture will be as follows. We have some bit vector $b \in \{0,1\}^n$ that we want to store in a potentially corruptible storage device or that we want to transmit over a noisy channel. When we read the transmitted bits, instead of reading $b_1, \ldots, b_n$, we see corrupted bits $\tilde{b}_1, \ldots, \tilde{b}_n$ where for some set of bits $\mathcal{I} \subset \{1, \ldots, n\}$, $\tilde{b}_i \neq b_i$ for $i \in \mathcal{I}$.

Our goal is to encode our bit string b into some larger string $E(b) \in \{0,1\}^m$ $(m > n)$ which adds redundancy to the string. The hope is that, even if some bits of this longer string are corrupted, we'll be able to either:

1. **Detect** that $E(b)$ is corrupted, at which point we might ask whoever we're communicating with to resend b. Detection is most helpful where errors are rare.

2. **Correct** the corruptions in $E(b)$ and recover b from the corrected string. This is equivalent to not only detecting corruptions in $E(b)$, but being able to know at which indices they occurred (i.e. to find $\mathcal{I}$). Correction is a more reasonable goal when errors are frequent or, as is the case in data storage, when it's not possible to ask for a retransmission.

Example 1 (Redundancy through repetition)**.** *The simplest way to introduce* redundancy *is to simply repeat each bit, say k times. As was discussed in class, we can do this in a few ways. For example, we could set $E(b) = [b_1, \ldots, b_n, b_1, \ldots, b_n, \ldots, b_1, \ldots, b_n]$ or we could set $E(b) = [b_1, \ldots, b_1, b_2, \ldots, b_2, \ldots, b_n, \ldots, b_n]$. The first is more robust to "runs" of continuous errors, which are common in practice. In either case, by producing an encoded message of length kn, we produce a string from which we can detect if there was an error as long as there are $< k$ errors, not matter where the errors appear. If there are $\geq k$ errors, all copies of b_1 could be flipped and we won't be able to tell. We can correct an error as long as there are $< k/2$ errors (take the majority bit for each position.)*

Example 2 (Parity checks)**.** *Another simple method to add redundancy is* checksums. *Let $\oplus$ denote the XOR function: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$.*

Suppose we transmit 3 *bits* $b = [b_1, b_2, b_3]$ *as* $E(b) = [b_1, b_2, b_3, b_1 \oplus b_2 \oplus b_3]$. *The last bit is the* parity *of the first three: i.e. it's equal to one if there were an odd number of bits in* b, *or zero if there were an even number. If one of the bits (or the parity bit itself) gets flipped, the parity will be incorrect. However, if two bits get corrupted, the parity becomes correct again! Thus this method can detect when a single bit has been corrupted.*

More complicated checksums also allow for error correction. Consider storing 7 *bits:* $b_1, b_2, b_3, b_1 \oplus b_2, b_1 \oplus b_3, b_2 \oplus b_3, b_1 \oplus b_2 \oplus b_3$. *It is easily checked that, if up to three bits are flipped, we can detect that there was an error. If one bit is flipped, we can actually find where it is an correct it. Try to convince yourself of how to do this.*

2 Code Distance and the Hadamard Code

The last example given above is called the Hadamard code, which is obtained by computing the parity of all possible 2^n subsets of bits in b (really $2^n - 1$ since the parity of the empty set is always zero).

In particular, We define $E(b) : \{0,1\}^n \to \{0,1\}^{2^n}$ as follows. For $j \in 1, \ldots, 2^n$ let $S_j \subseteq 1, \ldots, n$ be the set of all bits equal to one in the length n binary representation of j. We set:

$$[E(b)]_j = \bigoplus_{i \in S_j} b_i$$

For $n = 2$, there are 2^2 potential values for the vector $b \in \{0,1\}^2$. Below we write $E(b)$ as the rows of a matrix with 2^2 columns, each corresponding to a possible value of b:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

The rows of this matrix are called the "codewords" in our code: a codeword is any bit vector that equals $E(b)$ for some input b. For $n = 3$, our codewords are the rows in

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Do you recognize this matrix? If we replaced the 0s with -1s, this is the same Hadamard matrix we saw in the Lecture on Fast Johnson Lindenstrauss transforms. If you recall from that lecture, the ± 1 version of this matrix was orthogonal. This implies that any row in the matrix, i.e. any possible code word $E(b)$, has exactly $2^n/2$ bits in common with any other row.

This is a very useful fact. It implies that, even if just under *half* of the bits in a transmitted codeword are corrupted, we can detect that there was an error. Why? Because it's not possible that our transmitted codeword looks like another valid codeword if there are $< 2^n/2$ errors. Moreover, we can correct the codeword as long as there are $< 2^n/4$ errors: we just set it to equal the unique closest codeword in our code. So, while it expands the length of our message exponentially, the Hadamard code is very robust to errors.

In general, thinking about the distance between codewords is the easiest way to think about the robustness of a code. This distance dictates our ability to detect and correct errors.

Definition 1 (Minimum distance)**.** *The minimum distance Δ of a code with encoding function $E : \{0,1\}^n \to \{0,1\}^m$ is defined:*

$$\Delta = \min_{x,y \in \{0,1\}^n, x \neq y} \|x - y\|_0.$$

Here $\|\cdot\|_0$ is the hamming distance – i.e. the number of non-zero entries in a vector.

It should be clear that the following is true:

Claim 1. *If a code has distance Δ then it is possible to:*

- *Detect up to k errors in $E(b)$ for any b as long as $k < \Delta$.*
- *Correct up to k errors in $E(b)$ for any b as long as $k < \Delta/2$.*

3 Good codes

A central question in coding is: for a given input length n and code length $m > n$, how large of a distance Δ can be achieved by the code? This is a very well studied problem, but surprisingly not totally resolved.

Here's a famous and easy to prove limitation on constructing codes with a given distance.

Theorem 2 (Singleton Bound)**.** *A code with input size n, code length m, and distance Δ must have length $m > n + \Delta$.*

Proof. Let $c_1, \ldots, c_q \in \{0,1\}^m$ be the codewords of our code. Consider removing the last $\Delta - 1$ bits from $c_1, \ldots, c_q$ to create new bit vectors $\tilde{c}_1, \ldots, \tilde{c}_q \in \{0,1\}^{m-\Delta+1}$. If our code has distance Δ, $\|c_i - c_j\|_0 \geq \Delta$ for all i, j, thus $\tilde{c}_i \neq \tilde{c}_j$ for all i, j. So every vector $\tilde{c}_1, \ldots, \tilde{c}_q$ is unique.

But there are at most $2^{m-\Delta+1}$ unique bit vectors of length $m - \Delta + 1$, meaning that $q \leq 2^{m-\Delta+1}$. If our input has length n, we need at least 2^n codewords, so we conclude that $n \leq m - \Delta + 1$. □

This result makes some intuitive sense – somehow if we want to tolerate Δ lost bits, we better compensate by *adding* Δ bits of information to our code. We won't be able to construct codes that match the singleton bound – i.e. achieve distance Δ with $m = n+\Delta+1$, but we will get pretty close.

Here's a famous positive result.

Theorem 3 (Gilbert-Varshamov bound)**.** *For any input size n and distance $\Delta = pm$ for $p \in [0, 1]$, there exists a code with length $m = \frac{n}{1-H(p)}$.*

Here $H(p)$ is the famous binary entropy function appearing in Figure 1 (which is related to the notion of entropy used in the 2nd law of thermodynamics is closely related.) $H(p)$ is defined:

$$H(p) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$$

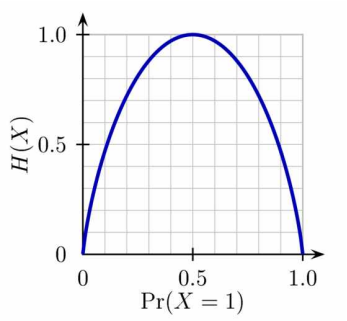

Figure 1: The graph of $H(X)$ as a function of X.

The entropy function appears repeatedly in coding and information theory applications

Proof sketch of Theorem 3. To prove the GV bound, we can explicitly construct a code with the required parameters: i.e. for n length, we give a code which has length $\frac{n}{1-H(p)}$ and distance pm.

This can be done in a greedy way. Consider the following procedure:

1. Initialize $S \leftarrow \{0,1\}^m$, and $C \leftarrow \emptyset$. C will eventually hold all the codewords in our code.
2. While S is not empty:
   (a) Chose any $z \in S$ arbitrarily. Remove z from S and add it to C: $C \leftarrow C \cup \{z\}$.
   (b) Remove any element z' in S with $\|z - z'\|_0 < pm$.

At each step of this iterative procedure, we add a single codeword to our code. We want to show that the number of steps is large. To do so, we simply bound how many potential codewords z' we also remove from S at each step. This is equal to the number of remaining bit strings in S within distance pm from z, which is only smaller than the total number of strings in $\{0,1\}^m$ within distance pm from z. This value can be bounded by:

$$\binom{m}{0} + \binom{m}{1} + \cdots \binom{m}{pm-1},$$

which is at most $2^{H(p)m}$. We gave a sketch of why this was the case in class by analyzing the last term. See https://people.cs.umass.edu/~arya/courses/690T/lecture4.pdf for a

more detailed proof, or check Wikipedia: https://en.wikipedia.org/wiki/Binomial_coefficient#Bounds_and_asymptotic_formulas.

If we remove at most $2^{H(p)m}$ elements from S in each step, it must be that our procedure runs for at least $2^m/2^{H(p)m} = 2^{1-H(p)m}$, meaning we generate $2^{1-H(p)m}$ codewords with distance pm. This proves Theorem (3), as we can assign a unique code word to each element in $\{0,1\}^n$ as long as $n < 1 - H(p)m$. □

This proof might remind you of the greedy ϵ-net construction we discussed in Lecture 12. The construction is not efficient, nor can it be decoded efficiently, but there are constructions nearly matching the bound which can be.

Via a very similar proof, it's also possible to prove the following limit on codes, which is tighter than the singleton bound:

Theorem 4 (Hamming bound, aka the "sphere packing" or "volume" bound)**.** *A code with input size n and distance $\Delta = p \cdot m$ must have code length $m \geq \frac{n}{1-H(p/2)}$.*

This bound is compared to Theorem 2 and Theorem 3 in Figure 2. The white region in the plot represents a gap in our knowledge about binary codes. On one hand, it's not known how to construct binary codes that beat Gilbert-Varshamov bound. On the other, we can't prove a lower bound that matches this bound (although improvemenst on the Hamming bound do exist). Many conjecture that the GV bound is tight.

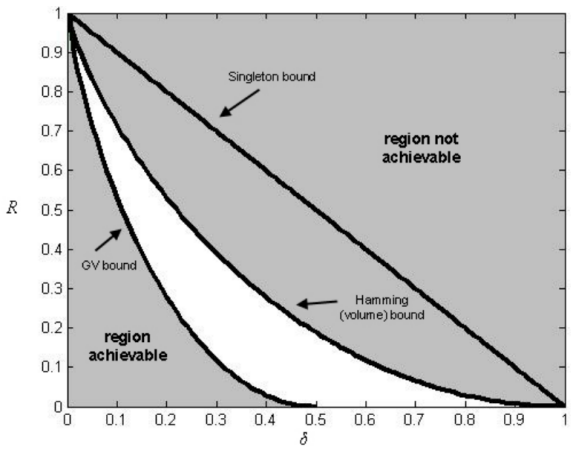

Figure 2: The plot above is taken from https://courses.cs.washington.edu/courses/cse533/06au/lecnotes/lecture5.pdf. δ corresponds to our Δ and $R = n/m$.

3.1 Aside on Shannon vs. Hamming

So far we have focused on codes from a *worst case* perspective. By seeking codes with a certain minimum distance, we can design error correction schemes which are robust to a certain number of errors in worst case locations. This approach was proposed and studied by Richard Hamming, who shared an office with Claude Shannon at Bell Labs.

In contrast, Shannon's famous theory studies *random* errors – i.e. we assume that each bit will be flipped independently with probability p. While closely related, there are subtle

differences between the worst-case and random error models. For example, under Shannon's model, it's known that a bound of $n/m \leq 1 - H(p)$ (i.e. comparable to the GV bound) is tight for error rate p (which corresponds to $p \cdot m$ errors in expectation).

4 Finite fields and polynomials

For the rest of the lecture we shift gears from theoretical bounds and analyze a popular practical code that allows for efficient encoding and decoding. It is based on finite field operations, which we review briefly.

In our case, *finite field* will refer to Z_q, the integers modulo a prime q. Recall that one can define $+, \times, \div$ over these numbers, and that $x \times y = 0$ iff at least one of x, y is 0. A degree d polynomial $p(x)$ has the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d.$$

It can be seen as a function that maps $x \in Z_q$ to $p(x)$.

Lemma 5 (Polynomial Interpolation)**.** *For any set of $n+1$ pairs $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ where the x_i's are distinct elements of Z_q, there is a unique degree n polynomial $g(x)$ satisfying $g(x_i) = y_i$ for each i.*

Proof. Let $a_0, a_1, \ldots, a_n$ be the coefficients of the desired polynomial. Then the constraint $g(x_i) = y_i$ corresponds to the following linear system.

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \ldots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \ldots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \ldots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Figure 3: Linear system corresponding to polynomial interpolation; matrix on left side is *Vandermonde*.

This system has a unique solution iff the matrix on the left is invertible, i.e., has nonzero determinant. This is nothing but the famous *Vandermonde* matrix, whose determinant is $\prod_{i \leq n} \prod_{j \neq i} (x_i - x_j)$. This is nonzero since the x_i's are distinct. Thus the system has a solution. Actually the solution has a nice description via the Lagrange interpolation formula:

$$g(x) = \sum_{i=0}^{n} y_i \prod_{j \neq i} \frac{(x - x_j)}{x_i - x_j}.$$

□

Corollary 6. *If a degree d has more than d roots (i.e., points where it takes zero value) then it is the zero polynomial.*

5 Reed Solomon codes and their decoding

The Reed Solomon code from 1960 is ubiquitous, having been used in a host of settings including data transmission by NASA vehicles and the storage standard for music CDs.

It is simple and inspired by Lemma 5. The idea is to break up a message into chunks of $\lfloor \log q \rfloor$ bits, where each chunk is interpreted as an element of the field Z_q. If the message has $(d+1)\lfloor \log q \rfloor$ bits then it can be interpreted as coefficients of a degree d polynomial $p(x)$. The encoding consists of evaluating this polynomial at n points $u_1, u_2, \ldots, u_n \in Z_q$ and defining the encoding to be $p(u_1), p(u_2), \ldots, p(u_n)$.

Suppose the channel corrupts k of these values, where $n - k \geq d + 1$. Let $v_1, v_2, \ldots, v_n$ denote the received values. If we knew which values are uncorrupted, the decoder could use polynomial interpolation to recover p. Trouble is, the decoder has no idea which received value has been corrupted. We show how to recover p if $k < \frac{n-d}{2} - 1$.

Lemma 7. *There exists a nonzero degree k polynomial $e(x)$ and a polynomial $c(x)$ of degree at most $d + k$ such that*

$$c(u_i) = e(u_i)v_i \qquad \text{for } i = 1, 2, \ldots, n. \tag{1}$$

Proof. Let $I \subseteq \{1, 2, \ldots, n\}$, with $|I| = k$ be the subset of indices i such that v_i has been corrupted. Then (1) is satisfied by $e(x) = \prod_{i \in I}(x - u_i)$ and $c(x) = e(x)p(x)$ since $e(u_i) = 0$ for each $i \in I$ and nonzero outside I. □

The polynomial e in the previous proof is called the *error locator polynomial*. Now note that if we let the coefficients of c, e be unknowns, then (1) is a system of n equations in $d+2k+2$ unknowns. This system is *overdetermined* since the number of constraints exceeds the number of variables. But Lemma 7 guarantees this system is feasible, and thus can be solved in polynomial time by Gaussian elimination.

We will need the notion of a polynomial *dividing* another. For instance $x^2 + 2$ divides $x^3 + x^2 + 2x + 2$ since $x^3 + x^2 + 2x + 2 = (x^2 + 2)(x + 1)$. The algorithm to divide one polynomial by another is the obvious analog of integer division.

Lemma 8. *If $n > d+2k+1$ then any solution $c(x), e(x)$ to the system of Lemma 7 satisfies (i) $e(x)$ divides $c(x)$ as a polynomial (ii) $c(x)/e(x)$ is $p(x)$.*

Proof. The polynomial $c(x) - e(x)p(x)$ has a root at u_i whenever v_i is uncorrupted since $p(u_i) = v_i$. Thus this polynomial, which has degree $d + k$, has $n - k$ roots. Thus if $n - k > d + k + 1$ this polynomial is identically 0. □

6 Code concatenation

Technically speaking, the Reed-Solomon code only works if the error rate of the channel is less than $1/\log_2 q$, since otherwise the channel could corrupt one bit in *every* value of the polynomial.

To allow error rate $\Omega(1)$ one uses *code concatenation*. This means that we encode each value of p —which is a string of $t = \lceil \log_2 q \rceil$ bits—with another code that maps t bits to $O(t)$ bits and has minimum distance $\Omega(t)$. Wait a minute: you might say. If we had such a code all along then why go to the trouble of defining the Reed-Solomon code?

The reason is that we do have such a code by the greedy construction discussed above: but since we are only applying it on strings of size t it can be encoded and decoded in

$\exp(t)$ time, which is only q. Thus if q is polynomial in the message size, we still get encoding/decoding in polynomial time.

This technique is called code concatenation.