15:848: Practical Information and Coding Theory for Computer Systems
Fall 2018

Lecture 2

September 5 2019

Lecturer: Rashmi Vinayak *Scribe: Zhili Feng, Carlos Martin*

2.1 General model

Our model is the following: We encode a **message** m into a **codeword** c and send it through a noisy channel. At the other end of the channel, we receive a (possibly different) codeword c' which is then decoded into either a **message or error**.

Since the channel is noisy, it might change the codeword during transmission. A changed field in the codeword vector (such as a flipped bit) is called an **error**. A missing field in the codeword vector (such as a lost byte) is called an **erasure**.

The decoder may be able to **detect** some errors. It may also be able to **correct** some errors and/or erasures.

2.2 Block codes

In these codes, each message and codeword is of **fixed size**. More precisely, $k = |m|$ denotes the size of each message m and $n = |c|$ denotes the size of each codeword c.

We use Σ to denote the codeword alphabet. For example, in the case of binary, we have $\Sigma = \{0, 1\}$. $q = |\Sigma|$ denotes the size of this alphabet. Σ^n is the set of all strings of length n on this alphabet. $C \subseteq \Sigma^n$ denotes the set of all codewords.

Let $\Delta(x, y)$ denote the number of positions where x and y differ. This is the "distance" between x and y. For example, $\Delta(100, 111) = 2$ since these strings differ at 2 positions. For a given set of codewords C, let

$$d = \min\{\Delta(x, y) : x, y \in C, x \neq y\}$$

denote the minimum distance between any two **distinct** codewords in C (two keywords that are equal are obviously at distance zero to each other).

A code that operates under these parameters is described as an $(n, k, d)_q$ code.

2.3 Role of minimum distance

Theorem 2.1. *A code with minimum distance d can:*

1. *Detect any $(d - 1)$ errors*

2. *Recover any* $(d-1)$ *erasures*

3. *Correct any* $\lfloor \frac{d-1}{2} \rfloor$ *errors*

Proof:

Case 1: Since every codeword is at distance at least d to any other codeword, you cannot corrupt a codeword into another codeword by changing $d-1$ or fewer positions. Thus as long as the number of errors (changed positions) is less than d, you will end up at a string that does not correspond to a valid codeword, which can be detected.

Case 2: You can recover from $(d-1)$ erasures (missing positions) by finding the codeword that matches the given string at the non-erased positions. For example, suppose $d = 3$ and consider the string

$$0xx111$$

which has 2 erasures. Suppose 010111 is a valid codeword. Since it matches at the non-erased positions, it **must** be the correct codeword, because any other codeword would differ by at least 3 positions and thus couldn't possibly match all the non-erased positions for the above string.

This emphasizes an important difference between errors and erasures: you **know** at which positions erasures occur, but you **do *not* know** at which positions errors occur.

Case 3: To correct an error, the string that results from applying the error to a codeword must remain closer to that codeword than to any other codeword.

Thus if the distance between the original codeword and any other codeword is at least d, the maximum number of errors we can have is $\lfloor \frac{d-1}{2} \rfloor$, i.e. just **under halfway** to the other codeword. If we go more than halfway, we could end up closer to that other codeword. □

In other words:

- For s**-bit error detection** we need $d \geq s+1$ (just over the maximum distance we can move away from the original codeword, so that we can't end up at another codeword).
- For s**-bit error correction** we need $d \geq 2s+1$ (just over twice the maximum distance we can move away from the original codeword, so that we can't end up being matched to a different codeword that is closer).
- To correct a erasures and b errors we need $d \geq a + 2b + 1$.

2.4 Large-scale distributed storage systems

Since servers can become unavailable, we need to store data in a **redundant fashion**. The traditional approach for this is **replication**. That is, we store multiple copies of data. An example is 3-replication. Suppose we have the following "blocks" of data:

```
abcd
```

With 3-replication we end up with 3 replicas:

```
abcd
abcd
abcd
```

However, this approach is too expensive for large-scale data. We need a better alternative: **sophisticated codes**. For example, suppose we want to store two data blocks `a` and `b` and tolerate any 2 failures. With 3-replication, we need to store 6 blocks:

```
block 1    a
block 2    a
block 3    a
block 4    b
block 5    b
block 6    b
```

We can tolerate any 2 failures, such as

```
block 1    ~~a~~
block 2    ~~a~~
block 3    a
block 4    b
block 5    b
block 6    b
```

The **storage overhead** is 3×, i.e. we needed 6 blocks to safely store 2 blocks of data. Now consider the following **erasure code**:

```
block 1    a
block 2    b
block 3    a + b
block 4    a + 2b
```

The last two blocks are called "parity blocks". If we delete any two blocks, we can still recover `a` and `b`. For example, suppose the first two blocks are deleted. Then we can recover `a` and `b` as follows:

$$\begin{aligned} \texttt{b} &= (\texttt{a + 2b}) - (\texttt{a + b}) \\ &= \texttt{block 4} - \texttt{block 3} \\ \texttt{a} &= 2(\texttt{a + b}) - (\texttt{a + 2b}) \\ &= 2(\texttt{block 3}) - \texttt{block 4} \end{aligned}$$

In this case, we only needed 4 blocks to safely store 2 blocks of data, so the storage overhead is only 2×. Thus, to attain the desired fault tolerance, we need much less storage. This works with more blocks of data. Example with $n = 14$ and $k = 10$:

```
a b c d e f g h i j
```

is stored as

$$\underbrace{\texttt{a b c d e f g h i j}}_{\text{10 data blocks}}\underbrace{\texttt{P1 P2 P3 P4}}_{\text{4 parity blocks}}$$

Almost all large-scale storage systems today—at companies like Facebook, Google, Amazon, and Microsoft—employ erasure codes.

2.5 Error-correcting multibit messages

Hamming codes are named after Richard Hamming (1915-1998), a pioneer in error-correcting codes and computing in general. These codes are of the form $(2^r - 1, 2^r - r - 1, 3)_2$ for any $r > 1$, e.g.

$$(3, 1, 3)_2, (7, 4, 3)_2, (15, 11, 3)_2, (31, 26, 3)_2, \ldots$$

which correspond to $r = n - k = 2, 3, 4, 5, \ldots$ "parity bits". Since the minimum distance $d = 3$, they can detect $d - 1 = 2$-bit errors or correct $\lfloor \frac{d-1}{2} \rfloor = 1$-bit errors. The high-level idea of Hamming codes is to "localize" the error in a string, i.e. narrow down its position.

2.5.1 Encoding

Consider the case where $r = n - k = 4$, i.e. $(n, k, d)_q = (15, 11, 3)_2$. We encode a message

$$m_{15}\ m_{14}\ m_{13}\ m_{12}\ m_{11}\ m_{10}\ m_9\ m_7\ m_6\ m_5\ m_3$$

of length 11 as follows:

$$m_{15}\ m_{14}\ m_{13}\ m_{12}\ m_{11}\ m_{10}\ m_9\ \underline{p_8}\ m_7\ m_6\ m_5\ \underline{p_4}\ m_3\ \underline{p_2}\ \underline{p_1}\ \underline{p_0}$$

where the non-message bits (i.e. parity bits) are underlined. To denote a number in binary, we write a subscript 2 after it. For example, $1101_2 = 2^3 + 2^2 + 2^0 = 13$. The parity bit $p_8 = p_{1000_2}$ is assigned the following value:

$$p_8 = m_{15} \oplus m_{14} \oplus m_{13} \oplus m_{12} \oplus m_{11} \oplus m_{10} \oplus m_9$$
$$p_{1000_2} = m_{1111_2} \oplus m_{1110_2} \oplus m_{1101_2} \oplus m_{1100_2} \oplus m_{1011_2} \oplus m_{1010_2} \oplus m_{1001_2}$$

Notice that all the terms on the right-hand side, together with p_{1000_2} itself (so 8 in total), are *precisely* those bits whose position in the codeword, when expressed in binary, has a 1 in the 4th bit (i.e. those bits whose position is $1xxx_2$). Similarly for $p_4 = p_{0100_2}$

$$p_4 = m_{15} \oplus m_{14} \oplus m_{13} \oplus m_{12} \oplus m_7 \oplus m_6 \oplus m_5$$
$$p_{0100_2} = m_{1111_2} \oplus m_{1110_2} \oplus m_{1101_2} \oplus m_{1100_2} \oplus m_{0111_2} \oplus m_{0110} \oplus m_{0101_2}$$

These are all the bits in the positions of the form $x1xx_2$. Similarly for $p_2 = p_{0010_2}$

$$p_2 = m_{15} \oplus m_{14} \oplus m_{11} \oplus m_{10} \oplus m_7 \oplus m_6 \oplus m_3$$
$$p_{0010_2} = m_{1111_2} \oplus m_{1110_2} \oplus m_{1011} \oplus m_{1010_2} \oplus m_{0111_2} \oplus m_{0110_2} \oplus m_{0011_2}$$

These are all the bits in the positions of the form $xx1x_2$. Similarly for $p_1 = p_{0001_2}$

$$p_1 = m_{15} \oplus m_{13} \oplus m_{11} \oplus m_9 \oplus m_7 \oplus m_5 \oplus m_3$$
$$p_{0001_2} = m_{1111_2} \oplus m_{1101_2} \oplus m_{1011_2} \oplus m_{1001_2} \oplus m_{0111_2} \oplus m_{0101_2} \oplus m_{0011_2}$$

These are all the bits in the positions of the form $xxx1_2$. Thus

$p_8 =$ localizes the error to the positions 1xxx or 0xxx
$p_4 =$ localizes the error to the positions x1xx or x0xx
$p_2 =$ localizes the error to the positions xx1x or xx0x
$p_1 =$ localizes the error to the positions xxx1 or xxx0

In addition, we can define a parity check bit p_0 as following:

$$p_0 = m_{15} \oplus m_{14} \oplus m_{13} \oplus m_{12} \oplus m_{11} \oplus m_{10} \oplus m_9 \oplus m_7 \oplus m_6 \oplus m_5 \oplus m_3 \oplus p_1 \oplus p_2 \oplus p_4 \oplus p_8$$

We don't need p_0 so our codewords are really of length $n = 15$ rather than 16.

2.5.2 Decoding

After transmission, we generate

$$b_8 = p_8 \oplus m_{15} \oplus m_{14} \oplus m_{13} \oplus m_{12} \oplus m_{11} \oplus m_{10} \oplus m_9$$
$$b_4 = p_4 \oplus m_{15} \oplus m_{14} \oplus m_{13} \oplus m_{12} \oplus m_7 \oplus m_6 \oplus m_5$$
$$b_2 = p_2 \oplus m_{15} \oplus m_{14} \oplus m_{11} \oplus m_{10} \oplus m_7 \oplus m_6 \oplus m_3$$
$$b_1 = p_1 \oplus m_{15} \oplus m_{13} \oplus m_{11} \oplus m_9 \oplus m_7 \oplus m_5 \oplus m_3$$

Since $b_i = p_i \oplus p_i^*$ where p_i^* is the original value of p_i, b_i tells us if there is an error somewhere in the 1xxx positions of the codeword. For example, if $b_8 = 0$ then $p_8 = p_8^*$ and there cannot be a 1-bit error (flip) anywhere in the 1xxx positions (including the parity bit itself). If $b_8 = 1$ then $p_8 \neq p_8^*$ and there must have been a flip somewhere in these positions.

Repeating this reasoning for b_4, b_2, and b_1 tells us the following: If the b_i are all zero, there are no errors. Otherwise, $b_8b_4b_2b_1$ gives us the exact location of the error. For example, $b_8b_4b_2b_1 = 0100$ would tell us that the bit at position $0100_2 = 4$, i.e. p_4, is wrong. $b_8b_4b_2b_1 = 1100$ would tell us that the bit at position $1100_2 = 12$, i.e. m_{12}, is wrong.

2.5.3 Extended Hamming code

Originally, we discarded the bit at position 0000_2 (i.e. p_0) because we didn't make use of it. However, we can *extend* our definition of the Hamming code to include this additional bit which functions as an additional parity check on the rest of the codeword. This increases the minimum distance d from 3 to 4, yielding a $(2^r, 2^r - r - 1, 4)_2$ code. Thus it can either detect $d - 1 = 3$-bit errors, or detect 2-bit errors and correct $\lfloor \frac{d-1}{2} \rfloor = 1$-bit errors. In other words, it can have the ability to detect 3-bit errors, or have the ability to correct 1-bit errors (and detect 2).

2.6 A lower bound on parity bits: Hamming bound

Question: How long must codewords be to ensure that we can put the valid codewords (i.e. those which correspond to messages) at a minimum distance of $d = 3$ from each other? Each of the 2^k codewords eliminates n neighbors plus itself, i.e. each of the 2^k codewords needs a neighborhood of $1 + n$ nodes that is disjoint from (does not overlap with) the neighborhood of any other codeword. Therefore, since there are 2^n strings in total, we need to have

$$\begin{aligned} 2^n &\geq 2^k(1+n) \\ n &\geq k + \log_2(1+n) \\ n &\geq k + \lceil \log_2(1+n) \rceil \end{aligned}$$

In the above Hamming code, we have $n \geq 11 + \lceil \log_2(15+1) \rceil = 15$, i.e. n must be at least 15. But n in fact *is* equal to 15. Thus the Hamming code matches this lower bound exactly. Codes that match this lower bound exactly are called **perfect codes**.

2.6.1 For more errors

To correct 2 errors, each of the 2^k codewords must eliminate itself, its neighbors, and its neighbors' neighbors, i.e.

$$1 + n + \binom{n}{2} = \binom{n}{0} + \binom{n}{1} + \binom{n}{2}$$

nodes. That is, all strings obtained by starting at a codeword and flipping 0, 1, or 2 bits must belong to that codeword. More generally, to correct s errors each codeword must "take ownership" over a neighborhood of

$$1 + n + \binom{n}{2} + \cdots + \binom{n}{s} = \sum_{i=0}^{s} \binom{n}{i}$$

nodes, so we need

$$n \geq k + \log_2 \sum_{i=0}^{s} \binom{n}{i}$$

Side note: These lower bounds assume arbitrary placement of bit errors. In practice, errors are likely to have patterns, such as being evenly spaced or clustered. If we assume such **regular errors**, we might be able to do better. We will come back to this later when we talk about Reed-Solomon codes. This is a big reason why Reed-Solomon codes are used much more than Hamming-codes.

2.7 Linear codes

One way to perform encoding, i.e. map messages $m \in \Sigma^k$ to codewords $c \in C \subseteq \Sigma^n$, is to use a giant lookup table, i.e. an array whose ith entry is the codeword corresponding to the ith message. For example,

m	c
000	0000
001	0011
010	0101
011	0110
...	...

However, this encoding is highly inefficient, because we need to store 2^k entries of n bits each. Instead, we can impose some additional structure on our code so that we don't have to use a giant lookup table. A kind of structure used in practical codes is **linearity**.

In mathematics, a field is a type of algebraic structure that contains addition, subtraction, multiplication, and division like those of rational and real numbers. If Σ is a field, then Σ^n forms a vector space.

A set of codewords $C \subseteq \Sigma^n$ (of size 2^k) is called a **linear code** if it is a linear subspace of Σ^n (of dimension k). This means that there exists a set of k independent vectors $v_i \in \Sigma^n$ for $1 \leq i \leq k$ that span the subspace, i.e. such that every codeword c can be expressed as

$$c = a_1 v_1 + a_2 v_2 + \ldots a_k v_k$$

for some $a_i \in \Sigma$. Such a linearly independent spanning set v is called a **basis**.

2.7.1 Properties of linear codes

1. The linear combination of two codewords is a codeword. This follows from the definition of a linear subspace (i.e. a subspace that is closed under linear combinations of its elements). In particular, **zero** (the codeword of all zeros) is always a codeword.

2. The minimum distance d is equal to the least weight nonzero codeword, where the **Hamming weight** $w(x)$ of a codeword x is the number of positions i where $x_i \neq 0$.

Proof:

$$\begin{aligned} d &= \min_{\substack{x,y\in C \\ x\neq y}} \Delta(x,y) \\ &= \min_{\substack{x,y\in C \\ x\neq y}} w(x+y) && \text{[relation between } \Delta \text{ and } w\text{]} \\ &= \min_{\substack{z\in C \\ z\neq 0}} w(z) && \text{[closure under } +\text{]} \end{aligned}$$

□

3. Every linear code has two matrices associated with it:

   (a) A $k \times n$ **generator matrix** G such that

   $$C = \{xG \mid x \in \Sigma^k\}$$

   G can be constructed by stacking the spanning vectors.

   (b) An $(n-k) \times n$ matrix H such that $C = \{y \in \Sigma^n \mid Hy^\mathsf{T} = 0\}$. That is, C is the null space of H. For a received word $y \in \Sigma^n$, Hy^T is called the syndrome. The syndrome is 0 if and only if the received word is a valid codeword. If the syndrome is not 0, it might contain information that could help us recover the original uncorrupted codeword.

Advantages of linear codes include the following:

- Encoding is efficient (vector-matrix multiple)
- Error detection is efficient (vector-matrix multiply)
- The syndrome Hy^T contains error information

Decoding: In general, we might have a q^{n-k} sized table for decoding (one for each possible syndrome). If $n-k$ is reasonably large, we want other approaches.

The following are basis vectors for the $(7,4,3)_2$ Hamming code:

	m_7	m_6	m_5	p_4	m_3	p_2	p_1
$v_1 =$	1	0	0	1	0	1	1
$v_2 =$	0	1	0	1	0	1	0
$v_3 =$	0	0	1	1	0	0	1
$v_4 =$	0	0	0	0	1	1	1

The least Hamming weight among nonzero codewords is 3, so $d = 3$.