Error correcting codes and their relationship with total perfect codes

Pantazi Andrei

Babes Bolyai faculty of mathematics

2024

What are error correcting codes?

Error-correcting codes are techniques used in digital communications and data storage to detect and correct errors that may occur during data transmission or storage. These codes are essential for ensuring the integrity and reliability of data, especially in noisy channels. The first error correcting code was invented in 1947, at Bell Labs, by the American mathematician Richard Hamming, who invented the **Hamming error correction code**

How does the Hamming error correction work?

Frustated by the random errors and burst errors that occured during his work at Bell Labs, Richard Hamming designed a simple, but efficient way of finding and correcting **single bits error**. He thought that by adding **redundancy** to the data, and by performing some operations called **parity checks**, he would ensure safe passage of the message through noisy channels. Let us see now how a hamming error correction works !

Hamming ECC 1

Suppose we want to send a message to our friend overseas. So many things could go wrong and corrupt the message. From some faulty hardware, all the way to a solar storm which we wouldn't even feel. In order to avoid (to some extend) corrupting the message we can incorporate the message into a Hamming ECC to ensure safe passage of the message. Suppose the message we want to send (in binary) is:

$$11001011011$$

In order for us to understand how the message is sent, we need to understand how it is composed (i.e. how the redundancy is added to our original data)

Hamming ECC 2

The message contains 11 binary symbols, and so we would use Hamming(15,11). The meaning of this notation is as follows:

- The **11** from the second parameter of the notation represent the number of data bits we send
- The **15** from the first parameter represents the number of data bits plus the number of redundancy bits (11+4)
- We start with a simple 4×4 matrix. The element a_{00} of the matrix will be called the **parity bit of the whole matrix**
- We will also have 4 redundancy bits, on the positions $a_{01}, a_{02}, a_{02}, a_{10}, a_{20}$
- The total number of bits in the matrix is $16 = 2^4$, which tell use that we have to perform 4 **parity checks**

Hamming ECC 3

We begin by filling the matrix from the top right corner (a_{03}), avoiding the spaces left for parity bits.

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

For the next step we need to define the parity groups. The idea behind the Hamming error correction code is that by performing parity checks on certain carefully selected parts of the matrix we can find and correct the error. The parity groups are as follows:

- **First group** is composed by the **second** and **fourth column** of the matrix

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

Hamming ECC 4

- **Second group** is composed by the **third** and **fourth column** of the matrix

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

Hamming ECC 5

- **Third group** is composed by the **second** and **fourth row** of the matrix

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

Hamming ECC 6

- **Fourth group** is composed of the third and fourth row of the matrix

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

Hamming ECC 7

To perform the **parity check** we simply count the number of 1's in the group. If we have an **even** number of 1's, we place **0** in the parity bit place. If we have a **odd** number of 1's, then we put **1** in the parity bit place. We start counting the number of 1's in the first group, and we have 4 bits that are equal to 1, and so we place 0 in the parity bit place.

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

We do the same for the rest of the groups

Hamming ECC 8

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

After placing the values of the redundacy bits/parity bits , we compute the parity of the whole matrix as we did for the groups. If we have a even number of 1's we place 0 at a_{00}, and if we have an odd number of 1's we place 1.

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

Hamming ECC 9

Now suppose that during transmission some random error occurs and one of the bits gets flipped, corrupting the message.

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

The first thing we need to do is check the parity bit of the matrix to see if an error occured indeed, and after checking, we count **9** 1's. But the parity bit of the matrix tells us that we should have an **even** number of 1's, and so we are confident in saying that an **odd** number of errors occured.

IMPORTANT: The parity bits tell us only if the number of 1's is odd or even, and so if an **even** number errors occured, the parity bits **would not** be able to tell us if any error occur(except when the number of errors=2 which we will see later). On the other hand if an **odd** number of error occurs, the parity bits wouldn't be able to tell us more than the fact that **at least one** error did occur.

Hamming ECC 10

Now we start checking which of the parity bits of the groups is noted wrong in order to find the location of the corrupted bit, and change it .

IMPORTANT: Since we are working with bits(binary), we are interested exclusively in finding the **location of the error** rather than the magnitude, since the magnitude will allways be **1**.When the error location is found we simply "flip" the bit.

- We start checking the parity for the **first group**, which is composed of the **second** and **third column**. The parity bit for this group is $\boldsymbol{a_{01}}$

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

After computing the number of 1's in the first group, we come to the conclusion that the error did not occur in this part of the matrix.

Hamming ECC 11

- We check the parity for the **second group**, which is composed of the **third** and **fourth column**. The parity bit for this group is $\boldsymbol{a_{02}}$

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

After computing the number of 1's in the second group, we come to the conclusion that the error did occur in this part of the matrix (and so the error is in either the third or the fourth column). To keep in mind this, we temporarly modify the parity bit to be 1.

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

Hamming ECC 12

- We check the parity for the **third group**, which is composed of the **second** and **fourth row**. The parity bit for this group is $\boldsymbol{a_{10}}$

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

After computing the number of 1's in the third group, we come to the conclusion that the error did occur in this part of the matrix (and so the error is in either the second or the fourth row). To keep in mind this, we temporarly modify the parity bit to be 1.

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

Hamming ECC 13

- We check the parity for the **fourth group**, which is composed of the **third** and **fourth row**. The parity bit for this group is $\boldsymbol{a_{20}}$

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

After computing the number of 1's in the fourth group, we come to the conclusion that the error did not occur in this part of the matrix.

Hamming ECC 14

Conclusion

We now know that the error:

- **IS NOT** in the first group
- **IS** in the second group
- **IS** in the third group
- **IS NOT** in the fourth group

We start putting all of this together(error NOT IN second column, NOT IN fourth column which tells us that the error is in the third column; error NOT in third row, error NOT in fourth row). We conclude that the error can be in only one place and that place is $\boldsymbol{a_{13}}$. And so we flip the bit. We also change the parity bits back to their initial state to keep a clean workflow.

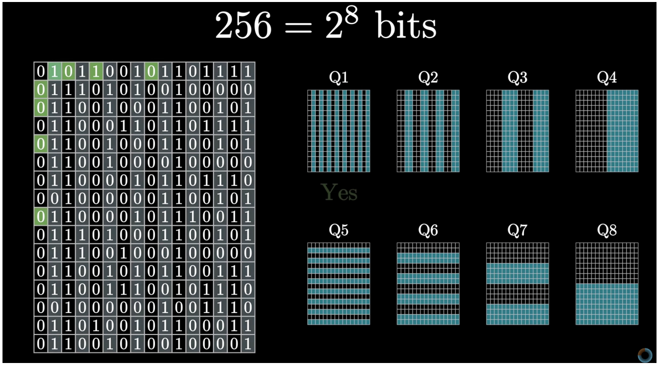
$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Hamming ECC 15

Now the receiver is able to get all the data bits, since he knows which bits are used for redundancy, and which bits are the actual data bits.

Remarks

- If one of the parity bits gets corrupted, we proceed with the same protocol and we find the error. Nothing is different.
- If two error occurs, we are able to tell that two errors occured indeed but we would not be able to find the location of these errors.(because the parity bit of the matrix tells us there are no errors, while the parity bit of a certain group/groups would tell us that there are indeed errors)
- For more than 2 errors, all bets are off, the algorithm does not work
- Hamming error correction code is a briliant way of detecting and corecting single bits error.
- Hamming error correction code can also be used for much larger data quantities/for much larger matrices



$$256 = 2^8 \text{ bits}$$

0	1	0	1	1	0	0	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0	1	1	0	1	1	1	1
0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	1
0	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
0	0	1	0	0	0	0	0	0	1	1	0	0	1	0	1
0	1	1	0	0	0	0	1	0	1	1	1	0	0	1	1
0	1	1	1	0	1	0	0	0	1	1	0	0	1	0	1
0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	1	0	1	0	1	1	0	0	1	1	1
0	1	1	0	0	1	1	1	0	0	1	0	1	1	1	0
0	0	1	0	0	0	0	0	0	1	0	0	1	1	1	0
0	1	1	0	1	0	0	1	0	1	1	0	0	0	1	1
0	1	1	0	0	1	0	1	0	0	1	0	0	0	0	1

Figure: In this case we have 8 parity group checks

Reed-Solomon ECC

Reed-Solomon error correction codes are a class error-correcting codes widely used in digital communications and data storage systems. Developed by Irving S. Reed and Gustave Solomon in 1960, these codes are particularly effective at detecting and correcting multiple random symbol errors in data transmission. Reed-Solomon codes operate over a finite field, allowing them to correct bursts of errors by adding redundant information to the original data. This redundancy enables the accurate reconstruction of the original data even when portions of it are corrupted or lost, making these codes essential in technologies like CDs, DVDs, QR codes, and satellite communications.

How does Reed-Solomon codes work?

Suppose we want to send to our friend overseas a message consisting of 4 integers, the message being 2521. We assign to each data packet(a data packet in this context would be a integer/digit) a label, representing the index of the value. We then create a polynomial using Lagrange interpolation with the property that p(0)=1,p(1)=5,p(2)=2,p(3)=1(the polynomial at index i gives us the value of the i'th packet). We then decide the amount of packets we can afford to lose during transmission(we choose k=2), and using the previously mentioned polynomial we append the value of the polynomial at the indexes of the newly added redundancy packets to the initial data that we want to transmit resulting in a message of length 4+k=4+2=6. This way if some of the packets gets lost(at most k packets can be lost for the algorithm to perform) during transmission the receiver is able to retrieve the lost packets by plugging the indexes of the lost packets into the polynomial. Let us now see a full example of a Reed-Solomon code in action !

RS ECC 1

- We want to send a message consisting of 4 integers "2521".
- In order to make our life easier we will see the message packets as points. Point 1: (0,2) Point 2:(1,5) Point 3:(2,2) and so on ...

Since we have 4 integers we will have 4 Lagrange polynomials, and so we start searching for them.

$$l_1 = -\frac{1}{6}(x-1)(x-2)(x-3)$$

This is the first lagrange polynomial. It needs to equal 1 when x=0, and 0 for the other values of the messsage. It is easy to notice that when plugging the indexes of the data packets we get 1,0,0,0. We do the same for the rest of the packets to find all the lagrange polynomials which we need for constructing the polynomial representing the message.

RS ECC 2

$$\begin{aligned} l_1 &= -\frac{1}{6}(x-1)(x-2)(x-3) \\ l_2 &= \frac{1}{2}(x)(x-2)(x-3) \\ l_3 &= -\frac{1}{2}(x)(x-1)(x-3) \\ l_4 &= \frac{1}{6}(x)(x-1)(x-2) \end{aligned}$$

Reed-Solomon Error Correction (RS ECC)

Now that we have all the Lagrange polynomials, we are ready to find the polynomial corresponding to the message. To do this, we simply multiply each Lagrange polynomial by its corresponding data packet value (the y-value of the point where the Lagrange polynomial equals 1). Therefore, we obtain:

$$\begin{aligned}
P(x) &= 2 \cdot l_1(x) + 5 \cdot l_2(x) + 2 \cdot l_3(x) + 1 \cdot l_4(x) \\
&= 2 \cdot \left(-\frac{1}{6}(x-1)(x-2)(x-3)\right) + 5 \cdot \left(\frac{1}{2}(x)(x-2)(x-3)\right) \\
&\quad + 2 \cdot \left(-\frac{1}{2}(x)(x-1)(x-3)\right) + 1 \cdot \left(\frac{1}{6}(x)(x-1)(x-2)\right) \\
&\vdots \\
P(x) &= \frac{4}{3}x^3 - 7x^2 + \frac{26}{3}x + 2
\end{aligned}$$

bal bla

$P(x) = \frac{4}{3}x^3 - 7x^2 + \frac{26}{3}x + 2$. This is the polynomial corresponding to our message. It is easy to see that when plugging the values of the indexes we find the value of the data packets, namely:
P(0)=2
P(1)=5
P(2)=2
P(3)=1
Now we need to complete the values for the redundancy packets that we've added and so we get P(4)=10,P(5)=37.The messsage that we want to send now becomes

$$\begin{bmatrix} \boxed{2} & \boxed{5} & \boxed{2} & \boxed{1} & \boxed{10} & \boxed{37} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

We try to send it, and on the way to the receiver some error occurs, and the message that is received is:

Frame Title

$$\begin{bmatrix} \boxed{2} & \boxed{5} & \boxed{1} & \boxed{37} \\ 0 & 1 & 3 & 5 \end{bmatrix}$$

Now using the same thinking as before the receiver creates a polynomial representing the message that he received. The beautiful part of the Reed-Solomon erro correction is that this polynomial is going to be the same as the polynomial that was created by the sender, and so by plugging the missing indexes into the polynomial, it is easy to find the missing values. The polynomial that passes through n points $(n > 3)$ is **unique**.

Frame Title

Proof
Suppose that the polynomial P(x) is not unique, meaning that there exists another polynomial Q(x)(different from P(x)) so that for each x P(x)=Q(x). Say we have n points (x_1,y_1),(x_2,y_2),...(x_n,y_n). Well that simply means that we have P(x_i)=Q(x_i)=y_i. Now suppose that we have a third polynomial R(x)=P(x)-Q(x). This polynomial is not 0, since P(x) and Q(x) are different. It easy easy to notice that the degree of both Q(x) and P(x) is less or equal to n-1, which means that the **degree of R(x) should also be less or equal to n-1**. But we know that for every i (where i from 0 to n) R(x_i)=P(x_i)-Q(x_i), and P(x_i)=Q(x_i). Well that means R(x_i)=0 for all i between 1 and n. And since we have exactly n points we arrive to the conclusion that **the degree of R(x) is n**, which of course is a **contradiction** since we have previously showed that it's degree should be less than n-1.

Frame Title

We start searching for the lagrange polynomials, corresponding to each of the points. The points of the receiver as as follows:

$$(0,2),(1,5),(3,1),(5,37)$$

Giving us the Lagrange polynomials:
$l_1 = -\frac{1}{15}(x-1)(x-3)(x-5)$
$l_2 = \frac{1}{8}(x)(x-3)(x-5)$
$l_3 = -\frac{1}{12}(x)(x-1)(x-5)$
$l_4 = \frac{1}{40}(x)(x-1)(x-3)$
Now we use this Lagrange polynomials to find the polynomial for the message:
$\mathrm{P(x)}=2\cdot \mathrm{l}_1+5\cdot \mathrm{l}_2+1\cdot \mathrm{l}_3+37\cdot \mathrm{l}_4$
...
$P(x) = \frac{4}{3}x^3 - 7x^2 + \frac{26}{3}x + 2$

Frame Title

Now the receiver simply plugs in the indexes that were missing to find the missing values of the message.P(2)=3,P(4)=10.

Important

Reed Solomon error correcting codes can be used over a **finite field**. The sender can choose a prime number p, that is bigger than every value in the data packets, and by doing so, using modular arithmetic we can create and find the corresponding polynomial more efficient. The greatest element in the data packets was 5, and so we can choose p=7. We can look at the polynomial mod 7:

$$\begin{aligned} P(x) &= \frac{4}{3}x^3 - 7x^2 + \frac{26}{3}x + 2 \quad \text{mod } 7 \\ &= \frac{4}{3}x^3 + \frac{26}{3}x + 2 \end{aligned}$$

$$P(0) = 2$$

$$P(1) = 5$$

$$P(2) = 2$$

$$P(3) = 1$$

BCH codes

BCH (Bose–Chaudhuri–Hocquenghem) codes are a class of error-correcting codes that are widely used in digital communication and storage systems to detect and correct errors. Developed by Bose and Ray-Chaudhuri in 1960, and independently by Hocquenghem in 1959, these codes are particularly powerful because they can be designed to correct multiple random errors in a block of data. A BCH code has two parameters BCH(n,m), where m represents the number of actual data bits, and m represents the number of data bits + the number of redundancy bits added. An interesting fact about this type of ECC is that we first need to decide how many errors we should correct, by choosing n and m.

How do BCH ECC works?

- We choose a integer t, representing the errors we intent to correct.
- We choose a suitable enviroment (Galois Field, primitive polynomial, primitive element) keeping in mind the number of errors we want to correct and the shape of the message.
- We transform the message into binary.
- We add redundancy if needed. (the length of the actual message needs to be equal to the second parameter of BCH(n,m), with m. If it is not, we simply append 0's to the left of message transformed in binary)
- We find a **generator polynomial**.
- We multiply the polynomial corresponding to the binary message with the generator polynomial, to encode the message.
- We search for the error location vector/polynomial (we will work only with GF(2^p) and so we are interested in the locations of the errors and not in the magnitude of these errors since we know they're all 1 if they exist and 0 if not)

Frame Title

- We compute **syndromes**. These should be 0, if no error occured
- Using these syndromes and some matrix computation we find the error location vector
- We add the error location vector to our received message, to correct it.
- We divide the modified received message with the polynomial generator to find the actual message.

BCH ECC 1

Suppose we wanna send the digit "9" to our friend overseas. First we need to look for a suitable GF to encode the message. To find such a GF we need to take into consideration the number of errors we wanna correct (t), and the length of the actual message (9 in binary is 1001). As mentioned before a BCH(n,m) transforms by multiplying with the generator polynomial a binary number of m bits into one of n bits. We choose t=2 (the number of errors we intend to correct). The number of bits in our initial message is 4. Keeping in mind the number of bits in the message and the number of errors we intend to correct, we choose to work with the extension field GF(16), where $16=2^4$. Since we have 2^4, we choose the primitive polynomial to be $p(x) = x^4 + x + 1$. We choose the primitive element/generator element of the field to be α, and we compute the table of GF(16) elements, where each element of the extension field is the power of α mod the primitive polynomial.

Frame Title

Power of α	**Elements of GF(16)**
0	1
1	z
2	z^2
3	z^3
4	$z + 1$
5	$z^2 + z$
6	$z^3 + z^2$
7	$z^3 + z + 1$
8	$z^2 + 1$
9	$z^3 + z$
10	$z^2 + z + 1$
11	$z^3 + z^2 + z$
12	$z^3 + z^2 + z + 1$
13	$z^3 + z^2 + 1$
14	$z^3 + 1$
15	1

Frame Title

Instead of writing "9" in its binary form which is "1001", we can now see it in it's polynomial form, namely $\alpha^3 + 1$. Now we need to find the generator polynomial in order for us to encode the message. To find this generator polynomial, we need to find the **minimal polynomial** for each of the elements of GF(16). We find this minimal polynomials by using the **conjugates** of each power of alpha. The set of elements that have the same minimal polynomial are called conjugates.

Definition

If $f(x)$ is the minimal polynomial of a, then it is also the minimal polynomial for the elements in the set $\{a, a^q, a^{q^2}, \ldots, a^{q^{r-1}}\}$, where r is the smallest integer such that $a^{q^r} = a$. The set $\{a, a^q, a^{q^2}, \ldots, a^{q^{r-1}}\}$ is called the set of conjugates. The elements in the set of conjugates are all the zeros of $f(x)$. Hence, the minimal polynomial of a can be written as:

$$f(x) = (x - a)(x - a^q)(x - a^{q^2}) \cdots (x - a^{q^{r-1}}).$$

Frame Title

Knowing this, we find that the set of conjugates along with their minimal polynomial are as follows:

$\{1, 2, 4, 8\} \Rightarrow f_1(x) = (x - z)(x - z^2)(x - z - 1)(x - z^2 - 1)$

$\{3, 6, 9, 12\}$

$\Rightarrow f_2(x) = (x - z^3)(x - z^3 - z^2)(x - z^3 - z)(x - z^3 - z^2 - z - 1)$

$\{5, 10\} \Rightarrow f_3(x) = (x - z^2 - z)(x - z^2 - z - 1)$

$\{7, 11, 13, 14\}$

$\Rightarrow f_4(x) = (x-z^3-z-1)(x-z^3-z^2-z)(x-z^3-z^2-1)(x-z^3-1)$

Frame Title

Now we can complete the table consisting the powers of α, and elements of GF(16), with their corresponding minimal polynomial

Power of α	**Elements of GF(16)**	**Minimal polynomials**
0	1	x+1
1	z	$x^4 + x + 1$
2	z^2	$x^4 + x + 1$
3	z^3	$x^4 + x^3 + x^2 + x + 1$
4	$z + 1$	$x^4 + x + 1$
5	$z^2 + z$	$x^2 + x + 1$
6	$z^3 + z^2$	$x^4 + x^3 + x^2 + x + 1$
7	$z^3 + z + 1$	$x^4 + x^3 + 1$
8	$z^2 + 1$	$x^4 + x + 1$
9	$z^3 + z$	$x^4 + x^3 + x^2 + x + 1$
10	$z^2 + z + 1$	$x^2 + x + 1$
11	$z^3 + z^2 + z$	$x^4 + x^3 + 1$
12	$z^3 + z^2 + z + 1$	$x^4 + x^3 + x^2 + x + 1$
13	$z^3 + z^2 + 1$	$x^4 + x^3 + 1$
14	$z^3 + 1$	$x^4 + x^3 + 1$

Frame Title

Now we are able to find the generator polynomial, because we know that the generator polynomial is :

$$g(x) = LCM(f_1(x), f_2(x)...f_{2t}(x))$$

- Where f_i(x) represents the minimal polynomial of α^i.
- Where t represents the number of errors we intend to correct.
- Where LCM represents the lowest common divisor.

Knowing this we find the generator polynomial as follows:

$$\begin{aligned} g(x) &= \text{LCM}(x^4 + x + 1, x^4 + x + 1, x^4 + x^3 + x^2 + x + 1, x^4 + x + 1) \\ &= (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1) \\ &= x^8 + x^7 + x^6 + x^4 + 1 \end{aligned}$$

Frame Title

So far we used GF(2^4), and as we saw we have a total number of **15** elements in the GF. Since the degree of the generator polynomial is **8** we need to use a **BCH(15,7)** to encode and decode the message polynomial. Knowing this, we append to left side of the message (which was 1001), three 0 bits, transforming or message from 1001 to 0001001. Now in order to encode the message we simple multiply the message polynomial, with the generator polynomial.

$$\begin{aligned}
binary("9") &= 1001 \\
binary("9")forBCH(15,7) &= 0001001 = \alpha^3 + \alpha^0 = \alpha^3 + 1 \\
encoded("9") &= (\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1)(\alpha^3 + 1) \\
encoded("9") &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^6 + \alpha^4 + \alpha^3 + 1 \\
binary(encoded("9")) &= 000111101011001
\end{aligned}$$

Frame Title

Now we have the polynomial/binary form of the message. We send it, and due to some noise in the channel the receiver gets a corrupted message

$$corrupted(binary(encoded("9"))) = 000\textcolor{red}{0}1110\textcolor{red}{0}011001$$

It is easy to understand that the received message r(x) is equal to the encoded codeword c(x) plus a vector error e(x).

$$\begin{aligned} received("9") &= encoded("9") + corruption("9") \\ r(x) &= c(x) + e(x) \\ r(x) &= g(x)(\alpha^3 + 1) + e(x) \end{aligned}$$

If no error occured, the the error vector/polynomial is equal to 0. As previously explained, since we are working with GF(2) (which means we have binary coefficients), we are exclusively interested in the **location** error rather than the magnitude, which we know that if it exists, then it is equal to 1.

Frame Title

In order to find the error location polynomial, we need to compute the **syndromes** (with respect to $GF(16)$), which are used to determine the coefficients of this polynomial. A **syndrome** S_i is the received polynomial evaluated at α^i. To find and correct the errors, we need to compute $2t$ syndromes.

$$\begin{aligned} S_1 &= r(\alpha) \\ &= 1 + \alpha^3 + \alpha^4 + \alpha^8 + \alpha^9 + \alpha^{10} \\ &\dots \\ &= \alpha \end{aligned}$$

$$\begin{aligned} S_2 &= r(\alpha^2) \\ &= 1 + \alpha^6 + \alpha^8 + \alpha^{16} + \alpha^{18} + \alpha^{20} \\ &\dots \\ &= \alpha^2 \end{aligned}$$

Frame Title

$$\begin{aligned}
S_3 &= r(\alpha^3) \\
&= 1 + \alpha^9 + \alpha^{12} + \alpha^{24} + \alpha^{27} + \alpha^{30} \\
&\dots \\
&= 0
\end{aligned}$$

$$\begin{aligned}
S_4 &= r(\alpha^4) \\
&= 1 + \alpha^{12} + \alpha^{16} + \alpha^{32} + \alpha^{36} + \alpha^{40} \\
&\dots \\
&= \alpha^4
\end{aligned}$$

Frame Title

Now we have the syndromes, and we are ready to search for the location of the error using the following formula (where a_i represents the coeffiecient of the α^i in the error location polynomial)

$$\begin{bmatrix} S_1 & S_2 & \ldots & S_{t-1} \\ S_2 & S_3 & \ldots & S_t \\ \vdots & \vdots & \ldots & \vdots \\ S_{t-1} & S_t & \ldots & S_{2t} \end{bmatrix} \begin{bmatrix} a_t \\ a_{t-1} \\ \vdots \\ a_1 \end{bmatrix} = \begin{bmatrix} S_{t+1} \\ S_{t+2} \\ \vdots \\ S_{2t} \end{bmatrix}$$

Frame Title

Now applying this to our case we get the following matrix computation:

$$\begin{bmatrix} \alpha & \alpha^2 \\ \alpha^2 & 0 \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \end{bmatrix} = \begin{bmatrix} 0 \\ \alpha^4 \end{bmatrix}$$

We compute this computation and we find $a_2 = \alpha^2$, $a_1 = \alpha$. We will use a_1, a_2 to compute the error locator polynomial, which has the form:

$$\lambda(x) = a_2x^2 + a_1x + 1$$

To find the error locations, we need to check $\lambda(\alpha^{-i})$ for i=0,1,2,...15. The error indexes corespond to thos values of i where $\lambda(\alpha^{-i}) = 0$. Keep in mind that since we're in GF(16) , α^{-3} is basically $\alpha^{15-2} = \alpha^{13}$.

Frame Title

$$\begin{aligned}\Lambda(1) &= \alpha^2(1)^2 + \alpha(1) + 1\\ &= \alpha^2 + \alpha + 1\\ &\Rightarrow \text{No error at position 0}\end{aligned}$$

$$\begin{aligned}\Lambda(\alpha^{-1}) &= \alpha^2(\alpha^{-1})^2 + \alpha(\alpha^{-1}) + 1\\ &= \alpha^2 \cdot \alpha^{-2} + \alpha \cdot \alpha^{-1} + 1\\ &= 1\\ &\Rightarrow \text{No error at position 1}\end{aligned}$$

$$\begin{aligned}\Lambda(\alpha^{-2}) &= \alpha^2(\alpha^{-2})^2 + \alpha(\alpha^{-2}) + 1\\ &= \alpha^2 \cdot \alpha^{-4} + \alpha \cdot \alpha^{-2} + 1\\ &= \alpha^2 + 1\\ &\Rightarrow \text{No error at position 2}\end{aligned}$$

Frame Title

$$\vdots$$

$$\begin{aligned}
\Lambda(\alpha^{-6}) &= \alpha^2(\alpha^{-6})^2 + \alpha(\alpha^{-6}) + 1 \\
&= \alpha^2 \cdot \alpha^{-12} + \alpha \cdot \alpha^{-6} + 1 \\
&= 0 \\
&\Rightarrow \text{Error at position 6}
\end{aligned}$$

$$\vdots$$

$$\begin{aligned}
\Lambda(\alpha^{-11}) &= \alpha^2(\alpha^{-11})^2 + \alpha(\alpha^{-11}) + 1 \\
&= \alpha^2 \cdot \alpha^{-22} + \alpha \cdot \alpha^{-11} + 1 \\
&= 0 \\
&\Rightarrow \text{Error at position 11}
\end{aligned}$$

Frame Title

Now since we've identified the error locations, we simply flip the bits at those specific locations to get the actual message. But the message that we get after doing so, is the encoded messsage (the polynomial message, multyplied with the generator polynomial), so we have to simply divide the message with the generator polynomial, using basic polynomial division. We will then get a 7 bit message, the message that was suppose to be sent.

Conclusion

BCH codes are extremly efficient at correcting burst and random errors, they are used along with a finite field and a primitive polynomial, and are quite complex in terms of computations and memory.

What are total perfect codes?

Total perfect codes, in the context of algebra are usually defined over a ring, and more exactly, over the zero divisor graph of the ring. Most of the error correction codes(BCH codes, Hamming codes) are devised over field, but we know that mathematically speaking a field is actually a ring with some extra features (inverses with respect to multiplication). Total perfect codes are important because they represent an ideal in coding theory where the code efficiently covers the entire space of possible received messages, ensuring optimal error correction and detection.

Frame Title

Let R be a ring, which realizes zero-divisor-graph G(R), with the vertex set Z*(R). G(R) is said to admit a total perfect code if $\exists$ some set

$$C(R) \subseteq Z^*(R)$$

(a set of vertices from the vertices set of the ring) such that

$$|Ann(x) \cap C(R)| = 1, \forall x \in Z^*(R)$$

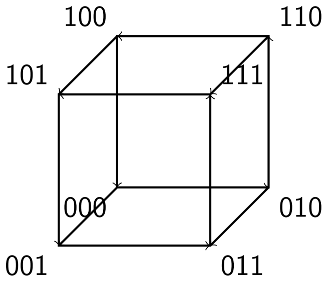
(no matter what vertex we choose from the set of vertices of the graph, inside the code, we have exactly one annihilator for that vertex)

Frame Title

where Ann(x)=ann(x)$\{0, x\}$, denotes the open neighborhood of x in G(R), and ann(x) $= \{y \in \mathbb{R} \,|\, xy = 0\}$, denotes the annihilator of an element x in the ring R. We conclude that if R realizes G(R) as its zero-divisor graph, the a code C(R) is a total perfect code in G(R) if and only if the subgraph created by C(R) is a matching in G(R), and the set Ann(x) $\| x \in$ C(R) is a partition of the set Z*(R).

Frame Title

It is interesting to notice that total perfect codes may exist in other forms, not only as sets of vertices from zero divisor graphs of rings. They can also be observed in Hamming graphs, and other graphs representing error correction codes. Consider the Hamming graph(3,2) (where 3 represents the number of bits, and 2 is the size of the alphabet since we are working with bits)



Frame Title

It is easy to notice that the sets of vertices(000,111),(100,011),(010,101),(001,110) are all total perfect codes in the graph. As a remark, when working with HammingGraph(3,2) these are the only total perfect codes.

Frame Title

Conclusion

Error correcting codes are creative tools used to encode, detect and correct errors. The efficiency of ECC increased over time, while the redundancy decreased. Total perfect codes are the next step in the evolution of ECC, and more important in the whole domain of data storage and data transmission.