Relations between error correcting codes, total perfect codes, commutative rings and their zero divisor graphs

Pantazi Andrei

March 2024

Contents

1 Introduction

In this Bachelor thesis we study the relations and similarities between the **error correcting codes (ECC)** and **total perfect codes (TPC)**. Both error correcting codes and total perfect codes are core concepts in coding and information theory. They might seem like they have nothing in common at first, but as we dwelve deeper we find that they share some interesting characteristics especially in terms of their structural properties and applications. It is important to mention that error correcting codes were invented long before the apparition of total perfect code (hamming error correction codes being the pioneer for the ECC). However, with the introduction of total perfect codes (initially as an algebraic-graph theory concept) new insight knowledge was discovered for understanding/improving coding theory. The discovery of TPC gave us a fresh perspective on coding/information theory, prompting reevaluation and enhancement of ECC designs. Let us now take a look at the similarities between ECC and TPC

- **Structural Similarities**
  Both TPC and ECC are represented with the help of mathematical/algebraic structures. TPC are subgraphs of the zero divisor graph for a certain ring, while ECC are sets of codewords, typically represented as vectors over finite fields. One could see the TPC (strictly the code), as "the set of correct words", where each vertex represents a codeword and each vertex in the code represents a correct codeword. Error correcting codes are methods to repair the errors caused by a noisy channel, and the channel, with all its words(vertices) can be represented as the zero divisor of a graph. Despite their different representation, both codes have specific structural properties that are crucial for their applications. TPC have adjacency properties within the graph, while ECC have properties related to distance and error correction capabilities.

- **Algebraic connections**
  TPC are closely related to algebbraic structures such as rings and fields(we will study especially the rings that admits TPC), as they are subsets of elements with specific algebraic properties. These properties often stem from the algebraic structure of the ring/field. Similarly ECC are rooted in algebraic structures (particularly finite fields and algebraic coding theory). The encoding/decoding processes of error correction codes involve operations such as parity checks, matrix multiplication, polynomials, modular and finite field arithmetic.

- **Optimization**
  From the optimization point of view, both TCP and ECC involve combinatorial optimization problems. TCP codes aim to maximize coverage while satisfying specific adjacency constraints.

  ECC involve optimizing properties such as minimum distance(hamming distance) which correspond to expanding the error correction capabilities

while minimizing redundancy.

- **Information transmission and storage**
  ECC are used primarly in telecomunications and data storage to ensure reliable transmission through noisy channels, noisy storage. They play an extremly important role in technologies such as digital storage systems and wireless/satellite communication.

  While TPC do not have direct applications in communication or storage, understanding their properties can lead to insights that benefit ECC. Studying the structural properties of total perfect codes helps improve the efficiency of ECC(greater error correction capabilities, lesser redundancy).

After the discovery and study of TPC, researchers saw the potential for improving the efficiency of error correction in digital storage systems. By using the insight knowledge from the TPC they developed improved versions of SR codes with enhanced error correction capabilities and reduced redundancy. These enhanced SR codes inspired by the structural properties observed in TPC gained immense importance in ensuring reliable data storage and transmission.

2 Error correcting codes

Error correcting codes **(ECC)** are techniques used in computer science and telecommunications to **detect and correct errors** that occur during data transmission or storage. These errors can be caused by various factors such as noise in communication channels, physical defects in storage media, or other sources of interference.

There are different types of error correcting codes, each with its own algorithms, properties and performance but they all serve the common purpose of ensuring data integrity by detecting and correcting errors. In this paper we will cover **Hamming error correction codes** and **Solomon-Reed error correction codes**.

- **Error Detection**: Error detecting codes are capable of detecting whether errors have occurred in the sent message. Common techniques for error detection include parity checks(we'll see that at[2.3]) and cyclic redundancy checks (CRC). This is the first step of the error correcting codes.

- **Error Correction**: Error correcting codes not only detect errors but also have the ability to correct them. These codes are designed to append redundancy symbols onto the transmitted data, allowing the receiver to identify and correct errors based on the redundant information. Examples of error correcting codes include Hamming codes, Reed-Solomon codes, and convolutional codes.

- **Redundancy**: Error correcting codes achieve error detection and correction by adding redundant bits/redundant symbols to the original data.

These redundant bits/symbols contain additional information that allows the receiver to reconstruct the original data even if errors occur during transmission or storage. A great example of this principle we'll encounter when talking about Solomon-Reed codes(see 2.2).

- **Hamming Distance**: The Hamming distance between two binary strings is the measure of their "difference"—the number of positions at which the corresponding bits are different. For example from 10001 to 11011, we have the hamming distance equal to 2, from 001 to 100 or 010 the distance is equal to 2. Error correcting codes are designed to have a certain minimum Hamming distance, which determines their ability to detect and correct errors. A higher minimum Hamming distance generally results in better error correction capabilities.
- **Encoding and Decoding**: Encoding refers to the process of transforming the message, more exactly in the context of error correcting codes, adding redundant bits to the original data, while decoding involves extracting the original data from the received encoded data, possibly correcting any errors in the process. Encoding and decoding algorithms are essential components of error correcting codes.
- **Applications**: Error correcting codes are used in various applications where data integrity is critical, such as telecommunications, digital data storage (e. g., hard drives, flash memory), satellite communication, wireless networks, and optical communication systems. All channels of communications are after all noisy channels, and so error correcting codes are a big part of our daily lives.

Question. You scratch a DVD or a CD and yet the data inside is intact. How does that happen ?

Answer. The scratching really does affect the 1's and 0's on the disk and so the data that is being read is different than the data that was initially stored. By using error correcting codes, the software is able to detect and correct the errors (unless the scratching was really really bad)

Keep in mind that any file (audio, video, text etc..) is ultimately a sequence of 1's and 0's and so it can be corrected if corrupted !

A simple but inefficient method to check for errors inside a file would be to keep 2 additional copies of that file in order to check if errors exist, and correct them.

Example 2.1. Consider the following 8bit uncorrupted file: **0 0 1 1 0 0 0 1**. And suppose that this file is corrupted just after getting the copies we need, resulting in the original file now being **0 0 1 1 0 1 0 1**.

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & \boxed{1} & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & \boxed{0} & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & \boxed{0} & 0 & 1 \end{bmatrix} \begin{matrix} \text{original corrupted file} \\ \text{copied uncorrupted file} \\ \text{copied uncorrupted file} \end{matrix}$$

We can compare all the elements from each column with each other in order to to find and correct the error

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

In this example, we can easily see that by using the copies of the file, we come to the conclusion that the element a_{05} of the matrix is corrupted, and so we can change it to 0.
However there are 2 big problems with this approach:

- **(Most important)** The redundancy elements needed for this method are far too many, occupying **2 times** as much space than the initial file. For a file of let us say 256bits, we need **an additional 512bits** just to make sure that there are no errors in the file.
- If for some reason, one of the copied sequences(redundancy sequences) required for error finding and error correcting is copied wrong then the whole algorithm is futile. Needles to say, if both copied versions are wrong at a certain bit, then the algorithm is actually breaking the file

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

a_{05} is about to be changed to 1
(to be corrupted)
because of the double error in the redundancy

In conclusion this method is an awful way to check and correct errors and should not be used under any circumstance.
Luckily for us there are many types of error correcting codes

2.1 Hamming correcting codes

Hamming codes use $\approx$ **3.5%** of the initial file, rather than **200%** as we saw in the example (2.1). Making it much more efficient than the previous method.
For a file of 256 bits, **9** are used for **redundancy**, and the rest **247** are called **message bits** and are free to carry out any message that we want.
If any bit from those 247 message bits gets flipped, just by looking at the whole file (the whole 256bit block), we are able to identify if there was an error and where precisely where it is.

Problem: if inside the message bits there are two errors, the machine will be able to say that two errors were found, but it won't know where and so it won't be able to correct them.

The basic principle of error correcting codes, is that in a vast space of all possible messages, only some subsets will be considered valid messages.

The history of Hamming correcting codes[3]
In 1940 a young man, named Richard Hamming was working for a company called Bell Labs, and some of his work involved using a very big expensive punch card computer at which he had limited access. The programs he kept putting through it kept failing because from time to time one bit would get misread. Frustrated by these reading errors he invented the **world's first error correction code**.

Example 2.2. Let us consider the file A, consisting of a **16**-bit block in the form of a 4×4 matrix, where a_{ij} represents an element of the matrix (with value 0 or 1), for $i, j \in \mathbb{N}$ such that $i, j < 4$.

$$A(\text{uncorrupted}) = \begin{bmatrix} \boxed{0} & \boxed{0} & \boxed{1} & 1 \\ \boxed{1} & 0 & 1 & 0 \\ \boxed{0} & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

We will choose 4 bits for redundancy as shown above (the elements $a_{01}, a_{02}, a_{10}, a_{20}$)

Parity check
For the parity check we reserve one special bit (the green one, a_{00} called **parity bit**) which will be responsible for tuning, and the rest are free to carry a message) The only job of this special bit is to assure us that the total number of 1's in the file is an even number. In our case, the matrix A has an odd number of 1's and so, we flip the bit to be 1.

$$A(\text{uncorrupted modified}) = \begin{bmatrix} \boxed{1} & \boxed{0} & \boxed{1} & 1 \\ \boxed{1} & 0 & 1 & 0 \\ \boxed{0} & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

but if the file would by default contain a even number of 1's, then the reserved bit would have been kept at a 0

Remark 2.3. Notice that if any bit of the message gets flipped, either from 0 to 1 or from 1 to 0, it changes the total count of 1's from being even to being odd. Therefore, if the sender sends out an even number of 1's and the receiver gets an odd number of 1's, you can be sure that some error occurred (it could be 1, 3, or more specifically $2n-1$ errors), even though you might have no idea where it occurred. On the other hand, if there had been $2n$ errors (an even number of errors), then the receiver's parity would still be even, and so the receiver cannot be fully confident that an even count means the message is error-free.

We might complain that a message which gets messed up by only 2 bit flips is pretty weak, and we would be absolutely right. However, we must keep in mind that there is no method for error detection or correction that can give us 100% confidence that the message we receive is the one that the sender intended. After

all, enough random noise could always change one valid message into another valid message. Instead, the goal is to come up with a scheme that's good enough for a certain maximum number of errors or to reduce the probability of a false positive. Parity checks on their own are quite weak, but by distilling the idea of change across a full message down to a single bit, what it gives us is a tool for more sophisticated methods.

For example, as Hamming was searching for a way to identify where an error happened, not just that it happened, his key insight was that if you apply some parity checks to certain carefully (in our case, 4 parity checks are required)selected subsets of the message, you can pin down the location of the single bit error.

In the end after doing all 4 parity checks we can detect and correct a single error and/or only detect 2 errors. For more than 2 errors, all bets are off.

Let us now see a full example of a Hamming code in action!

Let A be an uncorrupted matrix, sent by Alex to Mihai. Unfortunately due to some noise the matrix that Mihai receives has one error. We use Hamming code to search and solve the error.
the matrix sent by Alex:

$$A(\text{uncorrupted}) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

the element a_{00} is the parity bit (in our case 0 because the parity of the whole matrix is 0 - it has an even number of 1's) and a_{01}, a_{02}, a_{10}, a_{20} are used for redundancy

the matrix received by Mihai:

$$A(\text{corrupted}) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

We are now ready to search for the errors. The first thing we should do, is count the number of 1's in Mihai's matrix. We count **9** bits which tells us that we have a number of n=2k+1 errors (an odd number of errors), since the parity bit received was 0. (Hamming correcting codes are absolete when it comes to more than 2 errors as previously mentioned)

we start the first parity check:

$$A(\text{corrupted, first parity group check in work}) = \begin{bmatrix} 0 & 0 & \boxed{1} & \boxed{0} \\ 1 & 0 & \boxed{1} & \boxed{1} \\ 1 & 0 & \boxed{1} & \boxed{0} \\ 1 & 1 & \boxed{1} & \boxed{0} \end{bmatrix}$$

we chose a_{02} as the parity bit of the group, and we want to have an even number of 1's in this group. We check and see that we have an odd number of 1's and so we change the parity bit to be 0.

$$A(\text{corrupted first parity group check DONE}) = \begin{bmatrix} 0 & 0 & 0 & \boxed{0} \\ 1 & 0 & \boxed{1} & \boxed{1} \\ 1 & 0 & \boxed{1} & \boxed{0} \\ 1 & 1 & \boxed{1} & \boxed{0} \end{bmatrix}$$

we are now confident that the error is in this group.
we start the second parity check:

$$A(\text{corrupted second parity group check in work}) = \begin{bmatrix} 0 & \boxed{0} & 0 & \boxed{0} \\ 1 & \boxed{0} & 1 & \boxed{1} \\ 1 & \boxed{0} & 1 & \boxed{0} \\ 1 & \boxed{1} & 1 & \boxed{0} \end{bmatrix}$$

we chose a_{01} as the parity bit of the group, and we want to have an even number of 1's in this group. We check, and see that we have an even number of 1's and so we leave the bit as is.
we start the second parity check:

$$A(\text{corrupted second parity group check DONE}) = \begin{bmatrix} 0 & \boxed{0} & 0 & \boxed{0} \\ 1 & \boxed{0} & 1 & \boxed{1} \\ 1 & \boxed{0} & 1 & \boxed{0} \\ 1 & \boxed{1} & 1 & \boxed{0} \end{bmatrix}$$

After doing these first two parity checks, we can say that the error is in the first group(first group=third column and fourth column, second group=second column and fourth column), but not in the second. And so we are confident in saying that the error is in first group - second group, we find the error's column number (the third column of the matrix)
Third parity check:

$$A(\text{corrupted third parity group check in work}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} \\ \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} \end{bmatrix}$$

we chose a_{20} as the parity bit of the group, and we want to have an even number of 1's in this group. We check, and see that we have an odd number of 1's and so flip the parity bit

$$A(\text{corrupted third parity group check DONE}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & \boxed{0} & \boxed{1} & \boxed{0} \\ \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} \end{bmatrix}$$

we know now that the error is in the first group, and in the third. By intersecting the elements of the third group with the elements of the first group that are only in the first group, and not in the second one, we come to the conclusion that the error can only be in one of two places. Either a_{22} or a_{32}. After doing the last group parity check, we will know in which of these places is the error:

$$A(\text{corrupted fourth parity group check in work}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} \\ 0 & 0 & 1 & 0 \\ \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} \end{bmatrix}$$

we chose a_{10} as the parity bit of the group, and we want to have an even number of 1's in this group. We check, and see that we have an even number of 1's and so leave the parity bit as is.

$$A(\text{corrupted fourth parity group check DONE}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} \\ 0 & 0 & 1 & 0 \\ \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} \end{bmatrix}$$

Conclusion:
The error is on the position a_{22} and so we flip the bit, to get the actual matrix, the matrix that Alex sent.

$$A(\text{corrupted, repaired}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & \boxed{0} & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

the elements $a_{00}, a_{01}, a_{02}, a_{10}, a_{20}$ are used for redundancy, and have no purpose, other than helping the receiver find and correct an error. 11 out of the 16 bits are used to send the data, and those are the bits that we are interested in. (in technical jargon, we use Hamming(15,11) for this example)

Question:What happens if the noise affect the parity bit of the whole matrix ?
Answer:We proceed exactly the same and at some point we find the error
Question:What if we have bigger matrices ?
In our case we had a matrix of 4×4, in total 16 bits, which is 2^4, and from it we get the number of parity groups we must check (in our previous case, 4). For a matrix of 16×16, 256 bits, we have 8 parity groups.

[3]

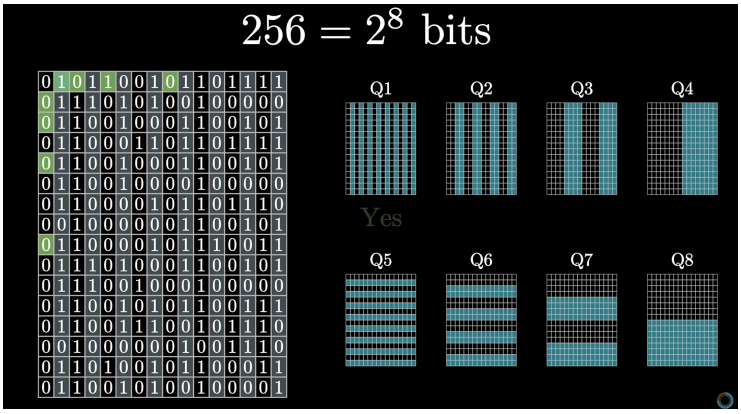


Figure 1: In this case we have 8 parity group checks

Conclusion: Hamming error correcting codes are great for detecting and correcting single bit errors. They can detect (but not correct) 2-bit errors, and do not work for more than 2 errors .

2.2 Solomon-Reed correcting codes

[7]

Reed-Solomon codes are a group of error-correcting codes that were introduced by Irving S. Reed and Gustave Solomon in 1960. They have many applications, including consumer technologies such as CDs, DVDs, Blu-ray discs, QR codes, data transmission technologies such as DSL and WiMAX, broadcast systems such as satellite communications, DVB and ATSC, and storage systems such as RAID 6.
In order for us to fully understand how a Solomon-Reed correcting code works we must review a few mathematical key concepts:

2.2.1 Modular Arithmetic

Modular arithmetic is based on the ring(field to be more precise) of integers modulo a prime number p, denoted by $\mathbb{Z}_p$.

Definition 2.4. We say that two integers a and b are congruent mod p if p | (a - b), and we use the notation it like: a $\equiv$ b (mod p).

Let us see some examples:

Example 2.5. We choose p=5 for the following examples. We can say that 1≡6, and that 11≡16.

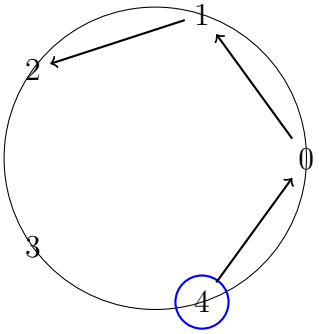
We can also perform operations, like addition, and multiplication (also substraction, the inverse of addition)

Example 2.6. $3+3 \equiv 1, 3 \cdot 3 \equiv 4, 3-4 = -1 \equiv 4$.

A simple way to understand the way modular arithmetic work, is to plot all the numbers smaller than p on a circle, and walk around it counterclockwise (from 1 to 2, from 2 to 3 and so on) when it comes to addition, and clockwise when it comes to subtraction, since 3-5=3+(-5). When it comes to multiplication, we simply see it as repetitive addition.

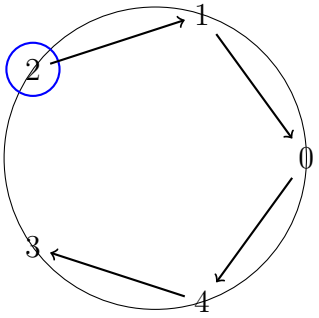
4+3≡2

here we start from the number 4, and walk around the circle counterclockwise, 3 steps



2-4=2+(-4)=-2≡3

here we start from the number 2, and walk around the circle clockwise, 4 steps



Let us consider the algebraic structure ($\mathbb{Z}_p$,+).

- Since we have **closure under addition** ($\forall a, b \in \mathbb{Z}_p$, also $a + b \in \mathbb{Z}_p$), we have that + is an operation.

  Proof. :
  We choose two arbitrary elements a, b from $\mathbb{Z}_p$. We encounter two cases :

  - If $a + b < p$, then from the definition of $\mathbb{Z}_p$, $a + b \in \mathbb{Z}_p$.
  - If $a + b \geq p$, then we use the modulo operation to bring $a + b$ back into the range of $\mathbb{Z}_p$. Since $\mathbb{Z}_p$ consists of integers modulo p, we can represent $a + b$ as $(a + b) \mod p$, which ensures that $a + b$ is an element of $\mathbb{Z}_p$. And so we proved the closure under addition

  □

- since we have **associativity** for $\forall a, b, c \in \mathbb{Z}_p$, we have that ($\mathbb{Z}_p$,+) is a **semigroup**

  Proof. : We choose three arbitrary elements a, b, c from $\mathbb{Z}_p$. In order to prove that + in associative in $\mathbb{Z}_p$, we must show that :

  $$(a+b)+c=a+(b+c)$$

  Since a, b, c are elements from $\mathbb{Z}_p$, we can write them all as a mod p, b mod p, c mod p, and so we get

  $$(a \bmod p+b \bmod p)+c \bmod p=a \bmod p+(b \bmod p+c \bmod p)$$
  and we know that a mod p+b mod p = (a+b) mod p (as previously showed) and we get:
  $$\Rightarrow (a \bmod p+b \bmod p)+c \bmod p=(a+b) \bmod p+c \bmod p$$
  $$\Rightarrow (a+b+c) \bmod p$$
  $$\Rightarrow a \bmod p+(b+c) \bmod p$$
  $$\Rightarrow a \bmod p+(b \bmod p+c \bmod p) \Rightarrow + \text{ is associative in } \mathbb{Z}_p$$

  □

- Since we have **identity element** ($\forall a \in \mathbb{Z}_p$, $\exists \hat{0}$ (an identity element) so that $a + \hat{0} = \hat{0} + a$), so ($\mathbb{Z}_p$,+) is a **monoid**

  Proof. $\forall a \in \mathbb{Z}_p \; a + \hat{0} = \hat{0} + a = a$ so there exist an identity element

  Since we have **an inverse** ($\forall a \in \mathbb{Z}_p$, $\exists$ -a (inverse) so that a+(-a)=$\hat{0}$, we get that ($\mathbb{Z}_p$,+) is a **group** □

Proof. $\forall a \in \mathbb{Z}_p \; \exists$ b $\in \mathbb{Z}_p$ (we cannot say -a in $\mathbb{Z}_p$, because there are no negative numbers here) so that a+b=p=$\hat{0}$ (p mod p = $\hat{0}$). and that b is more exactly : b=p-a □

- Since we have **commutativity** ($\forall a, b \in \mathbb{Z}_p$ a+b=b+a we get that ($\mathbb{Z}_p$,+) is a **abelian group**

  Proof. let us consider the succesor function s, s : $\mathbb{Z}_p -> \mathbb{Z}_p$
  s(n)=n+1, $\forall n \in \mathbb{Z}_p$

  (for styling reasons, all elements seen below 1, n, m are noted as 1 mod p, n mod p, m mod p, but writte as 1, n, m)
  Step 1: m+1=1+m. This we prove by induction on m. True when m=1.
  If true for m then: m+s(1)=m+1+1=s(m+1) by the definition of addition.
  s(m+1)=m+1+1=s(1+m) by the induction hypothesis.
  s(1+m)=1+1+m=1+s(m) by the definition of addition.
  Step 2: Assume that m+n=n+m. We would like to prove that m+s(n)=s(n)+m. The proof is fairly similar to that of Step 1.
  m+s(n)=s(m+n) by the definition of addition.
  s(m+n)=s(n+m) by the inductive hypothesis.
  s(n+m)=n+s(m) by the definition of addition.
  n+s(m)=s(n)+m by the lemma.
  $\Rightarrow$ + is indeed commutative on $\mathbb{Z}_p$, making ($\mathbb{Z}_p$,+) an **abelian group**

  □

Let us consider the algebraic structure ($\mathbb{Z}_p$,·).

- Since we have **closure under multiplication** ($\forall a, b \in \mathbb{Z}_p$, also $a \cdot b \in \mathbb{Z}_p$), $\Rightarrow \times$ is an operation.

  Proof. We choose two arbitrary elements a, b from $\mathbb{Z}_p$. We encounter two cases :

  - If $a \cdot b < p$, then from the definition of $\mathbb{Z}_p$, $a \cdot b \in \mathbb{Z}_p$.
  - If $a \cdot b \geq p$, then we use the modulo operation to bring $a + b$ back into the range of $\mathbb{Z}_p$. Since $\mathbb{Z}_p$ consists of integers modulo p, we can represent $a \cdot b$ as $(a \cdot b) \mod p$, which ensures that $a \cdot b$ is an element of $\mathbb{Z}_p$. And so we proved the closure under addition

  □

- since we have **associativity** for $\forall a, b, c \in \mathbb{Z}_p$, we have that ($\mathbb{Z}_p$,·) is a **semigroup**

Proof. We choose three arbitrary elements a, b, c from $\mathbb{Z}_p$. In order to prove that + in associative in $\mathbb{Z}_p$, we must show that :

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Since a, b, c are elements from $\mathbb{Z}_p$, we can write them all as a mod p, b mod p, c mod p, and so we get

(a mod p·b mod p)· c mod p=a mod p·(b mod p· c mod p)
and we know that a mod p ·b mod p = (a·b) mod p (as previously showed) and we get:
⇒ (a mod p·b mod p)·c mod p=(a·b) mod p·c mod p
⇒ (a·b·c) mod p
⇒ 2 a mod p·(b·c) mod p
⇒ a mod p·(b mod p·c mod p) ⇒ · is associative in $\mathbb{Z}_p$

□

- Since we have **identity element** ($\forall a \in \mathbb{Z}_p$, $\exists$ 1 (an identity element) so that a·1=1·a=a), we get that ($\mathbb{Z}_p$,·) is a **monoid**

Proof. $\forall a \in \mathbb{Z}_p$, a·1=1·a=a ⇒ there exist an identity element

□

- Since we do not necessarily have **an inverse** ($\forall a \in \mathbb{Z}_p$, $\exists$ a^{-1} (inverse) so that a·a^{-1}=1, ⇒ ($\mathbb{Z}_p$,·) is **NOT a group**

Proof. We can easily find an element in $\mathbb{Z}_p$ that does not have an inverse, and that is **0**. We cannot find an element a $\in \mathbb{Z}_p$ so that 0·a=1 □

- Since we have **commutativity** ($\forall a, b \in \mathbb{Z}_p$, a·b=b·a ⇒ ($\mathbb{Z}_p$,·) is a **abelian monoid**

Proof. let us consider the succesor function s, s : $\mathbb{Z}_p -> \mathbb{Z}_p$

S(n)=n+1, $\forall n \in \mathbb{Z}_p$ (for styling reasons, all elements from down below a, b,1 are noted as a mod p, b mod p, 1 mod p, but wrote as a, b,1)

Claim: S(a)×b=a×b+b
Base Case: We induct on b
Let b=0

See that S(a)·0=0=a·0+0 by zero property of multiplication and additive identity.
Inductive Step: Suppose S(a)·b=a·b+b
We must show that: S(a)·S(b)=a·S(b)+S(b)

S(a)·S(b)=S(a)·b+S(a)
a·b+b+S(a)=a·b+S(b+a)
a·b+S(a+b)=a·b+a+S(b)
a·S(b)+S(b)

$\Rightarrow \cdot$ is indeed a commutative operation on $\mathbb{Z}_p$

□

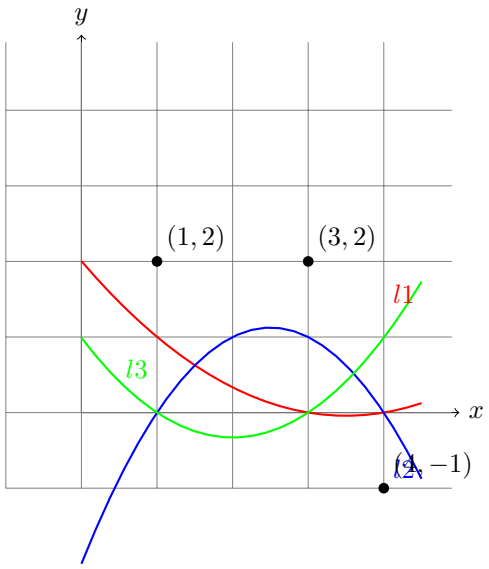
Conclusion:By keeping in mind that p was chosen arbitrarly, we conclude that these are universals proofs, and more than that, we conclude that the algebraic structure ($\mathbb{Z}_p$,+,·) is a **commutative unitary ring**when p is not a prime number, and it is a field when p is prime.

2.2.2 Lagrange Interpolation

Question: Given a list of coordinates x and y, how can we come up with a polynomial that goes through all of these coordinates ?
Answer:We use Lagrange interpolation. By using this formula, we get a polynomial of degree $n-1$ where n is the number of points. Let us consider three coordinates :(1,2), (3,2), (4,-1). What we are trying to find is a degree two polynomial, a quadratic that goes through all of these points.
Step 1:
We consider 3 different polynomials (called Lagrange polynomials) l1, l2, l3, that equal **1** at one x-coordinate, and **0** at every other x-coordinate.

$$l1 = \frac{1}{6}(x-4)(x-3)$$
$$l2 = -\frac{1}{2}(x-1)(x-4)$$
$$l3 = \frac{1}{3}(x-1)(x-3)$$

It is easy to observe that for l1 we have l1(1)=1, l1(3)=0, l1(4)=0. & $alsothat$: $l2(1) = 0, l2(3) = 1, l2(4) = 0, l3(1) = 0, l3(3) = 0, l3(4) = 1$.

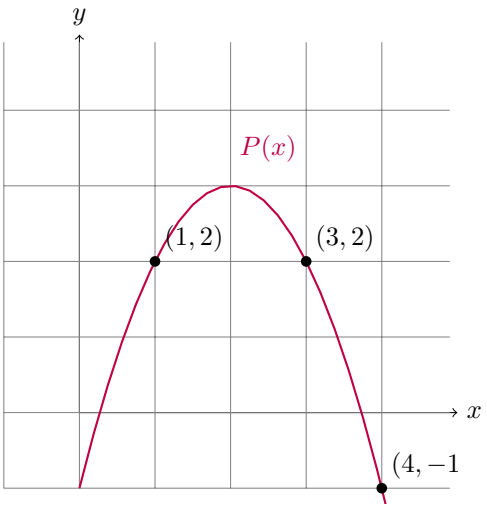


Step 2

We multiply each of the Langrange polynomials, with the y-coordinate of the point at which that equals 1, and we sum them all up in order to obtain a **unique** polynomial that goes through all those points.

$$\begin{gathered}\mathbf{2}\text{l1}+\mathbf{2}\text{l2-}\mathbf{1}\text{l3=P(x)} \\ 2\cdot\tfrac{1}{6}\text{(x-3)(x-4)}+2\cdot\text{-}\tfrac{1}{2}\text{(x-1)(x-4)-1}\cdot\tfrac{1}{3}(x-1)(x-3)=P(x) \\ \dots \\ -x^2+4\cdot\text{x-1=P(x)}\end{gathered}$$

(where P(x) represents the polynomial that goes through all the points)



Remark 2.7. An extremely important aspect of this polynomial (P(x))is that it is **unique**.

Proof. Consider n distinct points (x_1, y_1),(x_2, y_2),... (x_n, y_n). Let us assume now that P(x) is **not** unique. That means that there exists a different polynomial of degree $n-1$, Q(x), so that P(x_i)=Q(x_i)=y_1. Let us now consider the polynomial R(x)=P(x)-Q(x). Since we assumed that P(x) and Q(x) are distinct, R(x) cannot be 0. Also, since we know that both P(x) and Q(x) are of degree $n-1$, we observe that R(x) is of degree at most $n-1$, and that is:

$$\text{degree(R(x))} \leq n-1$$

And that means that R(x) has at most n-1 roots. But we know that $\forall i \in \mathbb{N}$, R(x_i)=P(x_i)-Q(x_i), and that is 0 (since P(x_i)-Q(x_i)=0). So R(x_i)=0 for all and given that we have n different points, we came to the conclusion that the degree of R(x) is n, which of course is a **contradiction**. □

Remark 2.8. The key fact that brings modular arithmetic and Lagrange interpolation together is that we can actually perform Lagrange interpolation using modular arithmetic. Given that our setup for Lagrange interpolation only requires addition, substraction, multiplication and divison, and since we know that they follow the same rules as regular arithmetic, we can actually use lagrange interpolation with modular arithmetic. Keep in mind thou calculations for polynomials are completely different. Let us consider a polynomial $f(x) = (x^2 + 2) mod 5$. The typical visual for this polynomial is some quadratic, but we want to think of the polynomials as a function so we plot this polynomial. We can consider the set of all inputs, and the set of all outputs. Given that the set of inputs and outputs are finite we can actually plot this entire

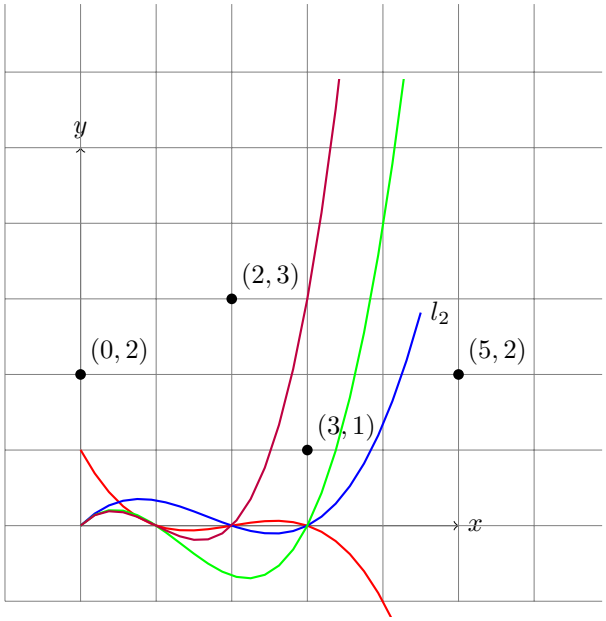
function over this field and although this looks entirely different to the function over real numbers, we've defined a whole new number system.

How is this useful ? Let us put this all together and talk about Reed-Solomon codes. This is a beautiful application of polynomials and modular arithmetic.

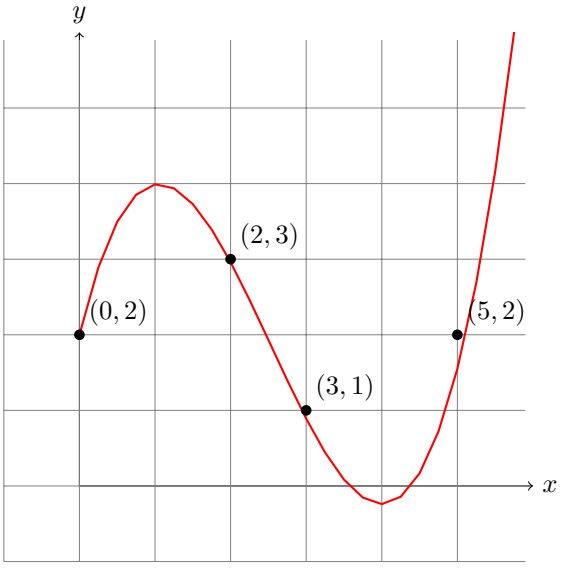
Example 2.9. Consider the message M, sent from the computer A to computer B. The message M is composed of 4 packets (n=4), each of them containing a number.
Step 1:
We start by setting a number k=2 which represents the number of packets which we can afford to lose(for k=2, we are allowed to lose at most 2 packets). K is practically the redundancy we talked about when disscusing Hamming codes. After choosing k=2, the number of packets from the message is now 6 (n=6).
Step 2:
We assign a label to each packet from the message, that label being the index at which the packet is found in the message. By doing so, if for some reason we lose some arbitrary packets, we know which packet was lost. We start by choosing a large prime number p, such that p is greater than than all the numbers contained in each of the packets. Now assume that all the arithmetic we do, is mod this prime number p. We then use Lagrange interpolation mod p to construct a polynomial f(x) so that f(0)=first packet, f(1)=second packet and so on, plugging the values given by the polynomial, for the redundancy packets as well. So, a short recap : A sends to B, the message M, with n+k packets. The first n packets of the message is the message itself, and the last k, are used for redundancy, to ensure that the code is correct (and are filled in, with the the values resulted by plugging the label value into the polynomial). The values and sent to B (with/without the interference of some noise), and B plugs the missing labels in order to find the value of the lost packets. It is easy to notice that the polynomial passing through those points is of degree at most n+k-1.
Step 3:
Now lets assume that the packages with index 1, and index 4 are lost during transmission, due to noise. Since the polynomial constructed by the first 4 packets is unique, if we were to use Lagrange interpolation on the packets received by B, we must end up with the same polynomial (since we proved that is unique), and finally we can plug the index values (label names) in the polynomial in order to find the missing packages.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ \boxed{2} & \boxed{4} & \boxed{3} & \boxed{1} & \boxed{2} & \boxed{2} \end{bmatrix} \quad \begin{matrix} \text{labels/index} \\ \text{packets} \end{matrix} \; <\!\!- \text{Message sent by A}$$

$$\begin{bmatrix} 0 & 2 & 3 & 5 \\ \boxed{2} & \boxed{3} & \boxed{1} & \boxed{2} \end{bmatrix} \quad \begin{matrix} \text{labels/index} \\ \text{packets} \end{matrix} \; <\!\!- \text{Message received by B}$$



We use the algorithm explained before(create the Lagrange polynomials and sum them up using the y-coordinate, and we create the polynomial passing through all those points). By using the resulted polynomial, we can find all the missing packages by plugging the label of the missing package into the polynomial.



Thus, by using modular arithmetic and polynomials, we can ensure safe passage for a message between two machines. Now let us see how all these concepts

work together with algebraic structures(rings) and graphs.

3 Rings

Let us now remember what a ring is. An algebraic structure(R, +, ·) , where R is a set along with 2 operations (in our case +,·), and more exactly, where **(R,+)** is an **abelian group**, **(R,·)** is a **semigroup** and the distributivity of + with respect to · holds.

3.1 Groups

By an operation on a set R we understand a map

$$\mathrm{f}: \mathrm{R} \times \mathrm{R} \to \mathrm{R}$$

Definition 3.1. Let · be an operation on A. We say that:

- · is **associative** if: $(a_1 \cdot a_2) \cdot a_3 = a_1 \cdot (a_2 \cdot a_3), \forall a_1, a_2, a_3 \in A$;
- · is **commutative** if $a_1 \cdot a_2 = a_2 \cdot a_1, \forall a_1, a_2 \in A$.
- e ∈ A is **an identity** element for · if a · e = e · a = a, ∀ a ∈ A.
- a^{-1} is **an inverse** for a, in the context of · if a·a^{-1}=e where e represents the identity element of ·

When using the multiplicative or additive notation, an identity element e is usually denoted by 1 or 0, respectively.

Definition 3.2. An algebraic structure $(A, \cdot)$, where · is an operation, is called a **groupoid**. If · is associative, then it is a **semigroup**. If it has an identity element, then it is a **monoid**. If $\forall a \in A$, there is an inverse for a and $(A, \cdot)$ is a monoid, then it is a **group**. If it is commutative, then it is a commutative groupoid, commutative semigroup, commutative monoid, and so on.

Definition 3.3. A group $(G, \cdot)$ is called a **cyclic group** if there exists some element g (called a **generator element**) with the following property:

$$\forall x \in G, \text{ with } x = g \cdot n = \underbrace{g + g + \cdots + g}_{n \text{ times}} \text{ for some integer } n.$$

Example 3.4. $\mathbb{Z}_4$ is a cyclic group, consisting of the elements $\{\hat{0}, \hat{1}, \hat{2}, \hat{3}\}$. The generator elements of $\mathbb{Z}_4$ are as follows:

- $\hat{0}$ is **not** a generator element:

$$\begin{aligned} 0 \cdot \hat{0} &\mod 4 = 0 \\ 1 \cdot \hat{0} &\mod 4 = 0 \\ 2 \cdot \hat{0} &\mod 4 = 0 \\ 3 \cdot \hat{0} &\mod 4 = 0 \end{aligned}$$

- $\hat{1}$ is a generator element:

$$\begin{aligned} 0 \cdot \hat{1} &\mod 4 = 0 \\ 1 \cdot \hat{1} &\mod 4 = 1 \\ 2 \cdot \hat{1} &\mod 4 = 2 \\ 3 \cdot \hat{1} &\mod 4 = 3 \end{aligned}$$

- $\hat{2}$ is **not** a generator element:

$$\begin{aligned} 0 \cdot \hat{2} &\mod 4 = 0 \\ 1 \cdot \hat{2} &\mod 4 = 2 \\ 2 \cdot \hat{2} &\mod 4 = 0 \\ 3 \cdot \hat{2} &\mod 4 = 2 \end{aligned}$$

- $\hat{3}$ is a generator element:

$$\begin{aligned} 0 \cdot \hat{3} &\mod 4 = 0 \\ 1 \cdot \hat{3} &\mod 4 = 3 \\ 2 \cdot \hat{3} &\mod 4 = 2 \\ 3 \cdot \hat{3} &\mod 4 = 1 \end{aligned}$$

$\{\hat{1}, \hat{3}\}$ are the generator elements of $\mathbb{Z}_4$.
Other examples of cyclic groups are $\mathbb{Z}_n$ (where $\mathbb{Z}$ is modulo n for some integer n), $n\mathbb{Z}$ (multiples of n), and the group of primitive polynomials modulo a prime polynomial, as we will see later in 3.4.1.

In other words, if all the non-zero elements of the group can be obtained by repeatedly adding an element g, then the group is a **cyclic group**, and g is called a **generator element** of G.

Remark 3.5. The generator elements of a cyclic group Zn are the elements that satisfy the following property:

$$\text{x such that gcd(x, n)=1}$$

Proof. Consider $\langle g \rangle$ to be the subgroup generated by g in $\mathbb{Z}_n$.

- Suppose we take $g \in \mathbb{Z}_n$ and $\gcd(g, n) = 1$.
- By definition, $\langle g \rangle = \{gk \mod n \mid k \in \mathbb{N}\}$.
- Since $\gcd(g, n) = 1$, there exists a multiplicative inverse for g, denoted as g^{-1}, such that $gg^{-1} \equiv 1 \mod n$.
- To show g generates $\mathbb{Z}_n$, we need to show $gk \mod n$ covers all elements in $\mathbb{Z}_n$.
- Let $x \in \mathbb{Z}_n$. Since $\gcd(g, n) = 1$, (then by **Bezout's identity**) there exist integers k and y such that:

  $$gk + ny = x$$

  Taking this equation modulo n:

  $$gk \equiv x \mod n$$

- Thus, $x \in \langle g \rangle$, and since x was choosen arbitrary, then we conclude that g is a generator of $\mathbb{Z}_n$.

□

3.2 Ideals

[6] In the following subsection, we consider all rings to be commutative.

Ideals are for rings what are normal subgroups for groups. When we think of normal groups, we may think about cosets, factor groups and kernels of homomorphisms. These concepts carry over to ideals in rings.

We will start studying what an ideal is by recalling the definition of a normal subgroup.

Definition 3.6.

We start with a group G. A normal subgroup is a subgroup N of G that divides G into cosets. All the cosets together form a group of their own, called quotient group. But this does not happen with every subgroup of G, it only happens when

$$g * N * g^{-1} = N \forall g \in G$$

If this is true, then N is a normal subgroup of G, and we denote it as follows :

$$N \triangleleft G$$

This should not be mistaken with the subgroup sign, noted like this:

$$N \leq G$$

Remark 3.7. it G abelian, then so every subgroup is also a normal subgroup. (if we take into consideration commutativity $g * N * g^{-1} = N \forall g \in G$, we can write $g * g^{-1} * N = N$ which is always correct no matter what N, or g we chose, except 0 , the same works for +)

Example 3.8. If we chose the group $(\mathbb{Z}, +)$, we can choose subgroups as follows $0\mathbb{Z}$(trivial normal subgroup), $1{\cdot}\mathbb{Z}$(trivial normal subgroup), $2\mathbb{Z}$, $3\mathbb{Z}$.... Let us now look more closely at the multiples of 6 ($6\mathbb{Z}$). This normal subgroup partitions the integers into 6 cosets, but only the normal subgroup is a group. The other 5 cosets are simply sets. If we write the 6 cosets like this, then the factor group is a finite group with 6 elements.

$$Cosets : \{\hat{0}, \hat{1}, \hat{2}, \hat{3}, \hat{4}, \hat{5}\}$$

This is the abstract algebra way of talking about the integers mod 6 (modular arithmetics as explained before). So a normal subgroup partitions a group G into cosets, and if we treat the cosets as elements, then they also form a group which we call "a factor group". The factor group is a completeley separate group from N and G. Normal subgroups and factor groups are used to decompose groups into simpler parts. Much like the integers can be factored into primes, and molecules can be broken apart into atoms, finite groups can be reduced to simple groups. We'll take these ideas and generalize them to rings.

Proposition 3.9. *Suppose we have a ring R, and we want some subset "I" to generate cosets that will act like a ring. We will use I, as notation for Ideal. What properties must I have in order to be an ideal ?. For starters, since R is an abelian group under addition, we definitely want I to be a normal subgroup of R under addition, and because R is abelian every subgroup is normal. This gives us our first requirement : I must be an additive subgroup of R. Since I is a subgroup, we can cover R in its cosets(there may be a finite/infinite number of cosets). We start by picking two arbitrary cosets x+I and y+I. Because I is a normal subgroup of R, if we add these two cosets together, we get the coset (x+y)+I.*

$$(x + I) + (y + I) = (x + y) + I$$

But R is a ring, and so, we want to be able to multiply two cosets. If we multiply these two cosets, we would hope to get xy+I

$$(x + I) * (y + I) \stackrel{?}{=} xy + I$$

For this to be true, then if we pick any element in the first coset, and multiply it by any element in the second coset, we should get and element in the third coset, and if we expand..

$$(x + i_1) * (y + i_2) \stackrel{?}{=} xy + i_3$$

$$xy + i_1y + xi_2 + i_1i_2 \stackrel{?}{=} xy + i_3$$

$$i_1y + xi_2 + i_1i_2 \stackrel{?}{=} i_3$$

in other words, no matter which two elements we pick from the cosets, the expression on right hand side should always be an element of I. However it is unclear why this should be true. There's a small trick we can use to find out when this does happen.

Trick 1: we we want to prove that:

$$i_1y + xi_2 + i_1i_2 \in I$$

Pick elements from cosets

$$i_1 + x \in x + I$$

$$y \in y + I$$

the element y is in the coset y+I because it is equal to y+0. Now we want the product of these elements to be in the coset xy+I.

$$\text{Question: when is } y * (x + i_1) \in xy + I \text{ ?}$$

$$(x + i_1) * y = xy + i_2$$

$$xy + i_1y = xy + i_2$$

$$i_1y = i_2$$

And since we choose y randomly from R, and i_1 from I, we conclude that i_1y is an element from I, no matter what y and i we choose.

We can add this to our list of required properties for an ideal I. We'll drop the index when adding it to our list of requirements

Trick 2 Now we pick the element x from the coset x+I, and then we pick any element from y+i_1, from the second coset. Just like before the product of these two elements should be in the coset xy+I.

$$\text{Question: when is } x * (y + i_1) \in xy + I \text{ ?}$$

$$x * (y + i_1) = xy + i_2$$

$$xy + xi_1 = xy + i_2$$

$$xi_1 = i_2$$

And since we choosed elements randomly from the coset (i_1, i_2), and from the ring(x, y), we conclude that no matter what element we choose from the ring,

x*i_1, should be in I.

If we return to our earlier work, we wanted to show that this expression was an element of the ideal (I). we have just shown for coset multiplication to work, the first and the second terms must be in I. if we substract the first two terms from both sides we get this: on the right hand side all three elements are in I, and since I is an abelian subgroup, the difference is also in I. Additionally this shows that the product of any two elements in I is also in I.
That is, I is closed under multiplication, so we are actually done building our definition of an ideal I. If these requirements are met, then the cosets of R can be added and multiplied together. We call I an ideal, and if we treat the cosets as new elements, they form a ring that is called factor ring.
Conclusion:
An ideal of a ring R is a subgroup of R with two additional properties:

$$I \leq R$$
$$\forall r \in R, x \in I$$
$$r * x \in I$$
$$x * r \in I$$

These additional properties are a generalization of the concept of multiples. For example, if you multiply any integer with a multiple of 5, you get another multiple of 5. With ideals, if you multiply an element in the ring by an element in the ideal, you get another element in the ideal.
Quick Notes:the ideal is a normal subgroup that is closed under multiplication

3.3 Local Rings

Local rings have some very interesting properties when it comes to admiting total perfect codes, see 5.2 and 5.10, so we have to define what a local ring is, in order to understand the next chapter.

Definition 3.10. A ring R is called **local**, if it has a **unique maximal ideal**.

Definition 3.11. We say that an ideal I of a ring R is a **maximal ideal** if for any ideal J of R with I⊆J, we have that either $J = I$ or $J = R$. In other words the only ideal that contains the maximal ideal is the whole ring, or trivially, that specific ideal.

Example 3.12. $(p\mathbb{Z}, +, *)$ is a maximal ideal, for every prime number p.

Proof. Consider the algebraic structure (13$\mathbb{Z}$,+,*) (multiples of 13). We first check if it is an ideal of $\mathbb{Z}$ (we use 3.2). In order to do that we must check first whether (13$\mathbb{Z}$,+) is a subgroup of $\mathbb{Z}$. We have closure under addition with respect to the group part(we choose two elements from (13$\mathbb{Z}$,+) a, b and so we know that a=13x, and b=13y, and so a+b=13x+13y=13(x+y) which is

indeed an element from (13$\mathbb{Z}$,+)), and we check if (13$\mathbb{Z}$,+) is a group. We have associativity, we have identity element(0), and we have inverses for all element in 13$\mathbb{Z}$. Therefore (13$\mathbb{Z}$,+) is a subgroup of ($\mathbb{Z}$,+). Now we must show that $\forall r \in (\mathrm{Z}, +, *)and\forall$i in$(13\mathrm{Z}, +, *)$ r*i is still an element from (13$\mathbb{Z}$,+,*). We can easily check this by choosing a random r, and a random i. We can write i of the form i=13*x, and we get that r*i=r*13*x, which is the same with (r*x)*13 and therefore is in the ring (13$\mathbb{Z}$,+,*) and so p$\mathbb{Z}$ is a maximal ideal for $\mathbb{Z}$. □

Definition 3.13. Let R be a local ring, and I be its unique maximal ideal. The algebraic structure R/I is called the **residue field** of R.

Definition 3.14. An element x from the ring R is called a **unit** if is invertible. This means that there exists $x^{-1} \in R$ such that $x \cdot x^{-1} = 1$.

Lemma 3.15. *Let R be a ring and $I \triangleleft R$ an ideal. If I contains a unit $x \in U(R)$, then $I = R$.*

Proof. $x \in I, x^{-1} \in R, \mathrm{so} x x^{-1} = 1 \in I$
But $\forall r \in R$we have that$r \cdot 1 = r \in I$, so we conclude that I=R

□

Proposition 3.16. *Let R be a ring and $\mathfrak{m}$ a non-trivial ideal in R. If every non-unit $x \in R \setminus U(R)$ is an element of $\mathfrak{m}$, then $\mathfrak{m}$ is the unique maximal ideal of R, so R is local.*

Proof. Let I be a non-trivial ideal of R. Due to Lemma 3.15, all the elements of I are non-units, hence $I \subseteq \mathfrak{m}$, since all the non-units are inside $\mathfrak{m}$. Hence, $\mathfrak{m}$ is a unique maximal ideal, which shows that R is a local ring. □

Proposition 3.17. *Let R be a ring and $\mathfrak{m}$ a maximal ideal. If for any $m \in \mathfrak{m}$ we have that $1 + m$ is a unit in R, then R is local.*

Proof. We choose an arbitrary element $x \in R \setminus \mathfrak{m}$. Therefore the ideal generated by x and $\mathfrak{m}$ is R since $\mathfrak{m}$ is maximal, and $\mathfrak{m} \subseteq (x, \mathfrak{m})$. This means that $(x, \mathfrak{m}) = (1)$, therefore there exists a $y \in R$ and $m \in \mathfrak{m}$ such that $xy + m = 1$. Thus $xy = 1 - m$ and hence xy is a unit in R. Thus there exists a $z \in A$ such that $(xy)z = 1$ and $x(yz) = 1$ (since multiplication is associative in a ring), therefore x is a unit in R and so by 3.16, $\mathfrak{m}$ is a unique maximal ideal, thereby making R a local ring. □

3.4 Fields

Definition 3.18. We discussed earlier about rings and their properties. Consider a ring (R,+,·) with some additional properties such as:

- **Identity element** in raport with the multiplicative operation. It is usually denoted with e.
- **Inverses** in raport with the multiplicative operation ·. More exactly all the elements from the ring (except 0) have a unique inverse.

This type of algebraic structure is called a **field**. One can see it as two groups bound together by distributivity. In other words, (R,+,·) is a field if and only if (R,+) is a commutative group, and (R^*,·) is a commutative group aswell and for every a, b, c ∈ R a(b+c)=ab+ac.

Example 3.19.
- $(\mathbb{Q}, +, \cdot)$: The field of rational numbers
- $(\mathbb{R}, +, \cdot)$: The field of real numbers
- $(\mathbb{C}, +, \cdot)$: The field of complex numbers
- $(\mathbb{Z}p, +, \cdot)$: The integers mod a prime number p., Z3 for example
- $(\mathbb{Z}np, +, \cdot)$: The integers mod a non-prime number np, is **NOT** a field because there exists elements with have multiple inverses, or without a inverse at all.

3.4.1 Galois Fields

A Galois Field, named after Evariste Galois is also known as a finite field (a field in which exists a finite number of elements). It is particularly useful in translating computer data since they are represented in binary forms, and all files are ultimately a sequence of bits (0 and 1's). Representing data as a vector in a Galois Field allows mathematical operations to work data easy and efficiently. As a consequnce its main usage is in the cryptography/data handling.

Definition 3.20. A **Galois Field** is a particular type of finite field.

Properties of Galois Fields

- A Galois Field is denoted by GF(q) has exactly q elements $q = p^n$, for some prime number p, and an integer n. "p" is called the **characteristic** of the Galois Field.
- It is a field (GF(q),+,·) and so we also have all the field properties, namely;
  - **closure**(for + and ·)
  - **distributivity** (of · with regards to +)
  - **asociativity**(for + and ·)
  - **identity element**(for + and ·)
  - every element has a **unique inverse** (for + and ·)
  - **commutativity** (for + and ·)
- in GF(q), $(GF^*(q), \cdot)$ forms a cyclic group of order q-1. This implies that there exists a generator g such that every non-zero element can be written as a power of g.

$$\forall x \in (\mathrm{GF}(q)^*, \cdot), \exists k \in \mathbb{Z} \text{ such that } x = g^k \tag{3.1}$$

Example 3.21. Consider GF(4), which is equivalent to GF(p^n), p=2, n=2. If we consider $GF^*(4)$, we have the elements 1,2,3, and we can easily check and see that 3 is the generator element for this cyclic group: $3^1 mod 4 = 3$, $3^2 mod 4 = 1$, $3^3 mod 4 = 3$, $3^4 mod 4 = 3$

- For any prime power q, there is a unique(up to isomorphism) finite field with q elements.
- Elements of GF(q), (q=p^n) can be represented as **polynomials** of degree less than n, with the coefficients in GF(p).

Example 3.22. Consider GF(8), which is equivalent to GF(p^n), p=2, n=3. Since n=3, we have 3 position bits in binary, and the following polynomials.

Number	Binary	Polynomial
0	000	0
1	001	1
2	010	x
3	011	$x+1$
4	100	x^2
5	101	x^2+1
6	110	x^2+x
7	111	x^2+x+1
8	**001**	**1**
...	...	...

Construction of Galois Fields

Proposition 3.23. *When constructing a Galois Field, GF(p^n) for n 1,* ***irreducible polynomials*** *play a crucial role, as we'll see in the next examples.*

Definition 3.24. An irreducible polynomial over a field K is a non-constant polynomial that cannot be factored into the product of two non-constant polynomials over K.

Example 3.25. Irreducible polynomials over GF(2)

- x
- x+1 These are trivially irreducible since they cannot be factored further.
- x^2+x+1
- x^3+x+1

Remark 3.26. Galois fields when used together with irreducible polynomials form a great environment for error correcting codes (for encoding and decoding, as we will see in the next chapter)

4 BCH codes

Bose–Chaudhuri–Hocquenghem (BCH) codes are a class of cyclic error-correcting codes widely used in digital communication and data storage. Developed in the 1960s, BCH codes are known for their ability to correct multiple random errors within data blocks. They offer flexibility in the design of codes, allowing for different error-correction capabilities by adjusting parameters like block length and error-correction capacity. BCH codes are particularly valued for their efficient encoding and decoding algorithms, making them a fundamental tool in error correction, especially in systems requiring high reliability.

What is different about BCH codes?
In most error correction codes we construct a code and find out its minimum distance to estimate its error correcting capabilities. When working with BCH codes, we go the other way around, more exactly, we decide at the start the number of errors we want to correct, and then we construct the generator polynomial.

4.1 Construction of a BCH error correcting codes

[1] In order to create a BCH code, we need the following:

– A primitive polynomial p (an irreducible polynomial needed for creating the field over which the codewords will be created)

Definition 4.1. A **primitive element** of $GF(p^n)$ where p is a prime number and n is an integer ($n \geq 1$), is an element α such that every element from the field (except 0, of course) can be expressed as a power of α.

Example 4.2. Consider $GF(5)$. Since 5 is a prime number, we are working in a field, and modulo arithmetic will work as expected. Consider the element 2.

$$\begin{aligned} 2^0 &\equiv 1 \pmod 5 = 1 \\ 2^1 &\equiv 2 \pmod 5 = 2 \\ 2^2 &\equiv 4 \pmod 5 = 4 \\ 2^3 &\equiv 8 \pmod 5 = 3 \end{aligned}$$

Hence, 2 is a primitive element of $GF(5)$.

Definition 4.3. A **primitive polynomial** P(x) over GF(q) isa prime polynomial over GF(q) with the property that in the extension field constructed modulo p(x), the field element represented by x is a primitive element.

* An extension field (modulo over the primitive polynomial P) over which we'll define the codewords.

Remark 4.4. Primitive polynomials of every degree exists over every galois field.

Remark 4.5. A primitive polynomial can be used to construct an extension field.

Example 4.6. We can construct GF(8) using the primitive polynomial:

$$p(x) = x^3 + x + 1$$

Consider the primitive element of GF(8) be α. Thus we can represent all the elements of GF(8) by the powers of a evaluated modulo p(x).

Powers of α	**Polynomial Representation**
α^0	1
α^1	z
α^2	z^2
α^3	$z+1$
α^4	z^2+z
α^5	z^2+z+1
α^6	z^2+1
α^7	1
...	...

* A minimal polynomial (which is determined by the extension field and primitive polynomial)
* A generator polynomial, used to encode and decode the message/codeword.

Proposition 4.7. *Factorization of* $x^{q-1} - 1$.
Let $a_1, a_2, \ldots, a_{q-1}$ *denote the non-zero field elements of GF(q). Then:*

$$x^{q-1} - 1 = (x - a_1)(x - a_2) \cdots (x - a_{q-1})$$

Proof. The set of non-zero elements of GF(q) forms a finite group under multiplication. Let a be any non-zero element of the field. That means a can be represented as the power of the primitive element α, i. e., $a = \alpha^r$ for some integer r.
Therefore:

$$a^{q-1} = (\alpha^r)^{q-1} = (\alpha^{q-1})^r = 1^r = 1 \quad \text{because} \quad \alpha^{q-1} = 1, \text{since} q^{-1} = ord(GF(q)^*)$$

Hence, a is a zero of $x^{q-1} - 1$. This is true for any non-zero element a. So every one of the q-1 elements of GF(q) is a root of $x^{q-1} - 1$. And since we have deg($x^{q-1} - 1$=q-1) we have:

$$x^{q-1} - 1 = (x - a_1)(x - a_2) \cdots (x - a_{q-1}).$$

□

Example 4.8. Consider the field GF(5). The non-zero elements of the field are 1, 2, 3, and 4. Therefore, we can write:

$$x^4 - 1 = (x-1)(x-2)(x-3)(x-4).$$

Proposition 4.9. *Obtaining generator polynomials.*
We know that in order to find generator polynomials for cyclic codes of block length n, we first have to factorize $x^n - 1$. Thus, $x^n - 1$ can be written as a product of its p prime factors:

$$x^n - 1 = f_1(x)f_2(x)\cdots f_p(x).$$

Any combination of these factors can be multiplied together to find a generator polynomial $g(x)$. If the prime factors of $x^n - 1$ are distinct, then there are $2^p - 2$ different non-trivial cyclic codes of block length n. There are two trivial cases that are being disregarded:

$$g(x) = 1 \quad \textit{and} \quad g(x) = x^n - 1.$$

Not all of the possible cyclic codes are "good codes" in terms of their minimum distance. We now evolve a strategy for finding good codes with desirable distance.

Definition 4.10. A block length n of the form n=$q^m - 1$ is called a **primitive block length** for a code over GF(q). A cyclic code over GF(q) of primitive block length is called a **primitive cyclic code**.

Definition 4.11. The field GF(q^m) is an extension field of GF(q). The primitive block length is n=q^m-1. We continue by considering the folowing factorization over the field GF(q).

$$x^n - 1 = x^{q^m} - 1 = f_1(x)f_2(x)...f_p(x)$$

This factorization will also be valid over the extension field GF(q^m) because the addition and multiplication tables of the subfield forms a part of the tables of the extension field. We also know that g(x) divides $x^n - 1$. Hence g(x) must be the product of some of these polynomials f_i(x). Every non zero element of GF(q^m) is a zero of $x^{q^m} - 1$, and hence, it is possible to factor $x^{q^m} - 1$.

$$x^{q^m} - 1 = \prod_{i=1}^{p}(x - a_i)$$

where we know that a_i ranges over all the non zero elements of GF(q^m). This implies that each of the polynomials f_1(x) can be represented in GF(q^m) as a product of some of the linear terms, and each a_i is a zero of exactly one of the f_i(x). This specific f_i(x) is called the **minimal polynomial of** a_i.

Example 4.12. Consider the subfield GF(2), and its extension field GF(8). It is easy to notice here that $q = 2$ and $m = 3$. Consider the factorization:

$$x^{q^m} - 1 = (x-1)(x^3 + x + 1)(x^3 + x^2 + 1)$$

Next, we consider the elements of the extension field GF(8):

$$0, 1, z, z+1, z^2, z^2+1, z^2+z, z^2+z+1$$

And therefore we can write:

$$\begin{aligned} x^7 - 1 &= (x-1)(x-z)(x-z-1)(x-z^2)(x-z^2-1)(x-z^2-z)(x-z^2-z-1) \\ &= ... = (x-z-1)(x-z^2-1)(x-z^2-z-1) \end{aligned}$$

It can be seen that over GF(8) we have the following:

$$\begin{aligned} x^3 + x + 1 &= (x-z)(x-z^2)(x-z^2-z) \\ x^3 + x^2 + 1 &= (x-z-1)(x-z^2-1)(x-z^2-z-1) \\ x^7 - 1 &= (x-1)(x^3+x+1)(x^3+x^2+1) \end{aligned}$$

Minimal polynomial	a_i **in GF**(8)	**Powers of** α
$x-1$	1	0
x^3+x+1	z, z^2, z^2+z	1, 2, 4
x^3+x^2+1	$z+1, z^2+1, z^2+z+1$	3, 6, 5

Table 1: Minimal polynomials and their roots in GF(8)

The elements in the third column represent the zeros of the minimal polynomials from the first column.

Definition 4.13. Two elements from GF(q^m) that share the same minimal polynomial over GF(q) are called **conjugates** with respect to GF(q).

Example 4.14. The elements $\alpha^1, \alpha^2, \alpha^4$ are conjugates with respect to GF(2). They share the same minimal polynomial : $f_2(\mathrm{x}) = x^3 + x + 1$

Definition 4.15. 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:

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

Example 4.16. Consider GF(256) as an extension field of GF(2). Let α be the primitive element of GF(256), which makes the set of conjugates $\{\alpha, \alpha^2, \alpha^4, \alpha^8, \alpha^{16}, \alpha^{32}, \alpha^{64}, \alpha^{128}\}$.
As explained before at 4.11, the minimal polynomial of α is:

$$f(x) = (x-\alpha)(x-\alpha^2)(x-\alpha^4)(x-\alpha^8)(x-\alpha^{16})(x-\alpha^{32})(x-\alpha^{64})(x-\alpha^{128}).$$

The right-hand side of the equation when multiplied would only contain coefficients from GF(2). Similarly, the minimal polynomial of α^3 would be:

$$f(x) = (x-\alpha^3)(x-\alpha^6)(x-\alpha^{12})(x-\alpha^{24})(x-\alpha^{48})(x-\alpha^{96})(x-\alpha^{192})(x-\alpha^{129}).$$

Definition 4.17. BCH codes defined over GF(q) with block length $q^m - 1$ are called **primitive BCH codes**

Proposition 4.18. *We know that g(x) is a factor of $x^n - 1$, and therefore the generator polynomial of a cyclic code can be written in the form :*

$$g(x) = LCM[f_1(x)f_2(x)...f_p(x)]$$

where $f_i(x)$ are the minimal polynomials of the zeros of g(x). Each minimal polynomial corresponds to a zero of g(x) in an extension field.

Proposition 4.19. *Let c(x) be a codeword polynomial, and e(x) be the error polynomial. The received polynomial can be written as:*

$$v(x) = c(x) + e(x)$$

where the polynomial coefficients are in GF(q).

Let us consider now the extension field GF(q^m). Let g_1, g_2... g_p be those elements of GF(q^m) which are the zeros of g(x) ($g(g_i)$=0, for all i=1... p). Since c(x)=a(x)g(x) for some polynomial a(x), we also have $c(g_i)$=0 for i=1... p, thus $v(g_i)=c(g_i)+e(g_i)=e(g_i)$ for i=1... p.

Proposition 4.20. *For a block length n we have:*

$$v(y_i) = \sum_{j=0}^{n-1} e_j (y_i)^j$$

for $i = 1, \ldots, p$. Thus, we have a set of p equations that involve the components of the error pattern only. If it is possible to solve this set of equations for e_j, the error pattern can be precisely determined. Whether this set of equations can be solved depends on the value of p, the number of zeros of g(x). In order to solve for the error pattern, we must choose the set of p equations properly. If we have to design for a t-error correcting cyclic code, our choice should be such that the set of equations can solve for at most t non-zero e_j.

Proposition 4.21. *Let us define now the syndromes S_i=e(g_i) for i=1... p. We want to choose g_1, g_2,... g_p in such a matter that t errors can be computed from S_1, S_2... Sp. If α is a primitive element then the set of g_i which allow the correction of t errors is $a^1, a^2, ...a^2t$. As a consequence we have a simple mechanism of determining the generator polynomial of a BCH code that can correct t errors*

Proposition 4.22. ***Steps for finding g(x) for BCH codes***
For a primitive blocklength $n=q^m$-1, we choose a prime polynomial, of degree m and construct GF(q^m). We then proceed by finding fi(x) the minimal polynomial of a^i for i=1...2t, the generator polynomial for the t error correcting code is simply:

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

Codes designed in this manner can correct at least t errors. In many cases the codes will be able to correct more than t errors. For this reason :

$$d = 2t + 1$$

is called the designed distance of the code and the minimum distance $d \geq 2t + 1$.
The generator polynomial has a degree equal to n-k. Note that once we fix n and t we can determine the generator polynomial for the BCH code

Example 4.23. Consider the primitive polynomial:

$$p(z) = z^4 + z + 1$$

We shall use this to construct the extension field GF(16). We choose α=z to be the **primitive element**, and we represent the elements of GF(16) as powers of α. The minimal polynomial is computed keeping in mind the properties we have discussed at 4.11 and 4.13

$\{1, 2, 4, 8\} \Rightarrow f_1(x) = (x - z)(x - z^2)(x - z^{-1})(x - z^{-2})$
$\{3, 6, 9, 12\} \Rightarrow f_2(x) = (x - z^3)(x - z^3 - z^2)(x - z^3 - z)(x - z^3 - z^2 - z - 1)$
$\{5, 10\} \Rightarrow f_3(x) = (x - z^2 - z)(x - z^2 - z - 1)$
$\{7, 11, 13, 14\} \Rightarrow f_4(x) = (x - z^3 - z - 1)(x - z^3 - z^2 - z)(x - z^3 - z^2 - 1)(x - z^3 - 1)$

4.2 Solving a BCH code problem

[2]

Power of α	**Elements of GF(16)**	**Minimal polynomials**
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
15	1	$x+1$

Example 4.24. Let us try now to encode, correct and decode a message using BCH codes. Suppose I want to send to a friend the message "8991", and i want to be able to correct at most 2 errors(t=2). First we decide the Galois Field over which we want to work (the coeficients of the polynomial are dictated by the field we choose). Depending of the number of errors we want to correct and on the complexity of the message we want to send, we choose a extension field. We can easily transform a digit into binary, and so we can choose to work with the GF(2), and since the biggest digit is 9(which in binary would be 1001, it needs 4 bits), we choose to work with the extension field GF(16)=GF(2^4). So far we have the following:

$$\begin{aligned} \text{Maximum number of errors we intend to correct: } & t = 2 \\ \text{Prime number for the Galois field: } & p = 2 \\ \text{Power of the extension field: } & n = 4 \\ \text{Degree of the generator polynomial: } & \deg(g) = ? \end{aligned}$$

We start by choosing a primitive element α, and a primitive polynomial with degree equal to the power of the extension field (4), and we choose z^4+z+1. We then compute the table with the powers of α, and the elements of GF(16)

We now need to find the minimal polynomial for each of the powers of α, and in order to do that we group together the conjugates sets as we saw at 4.13. Since n=4, we'll have 4 conjugate sets as follows:

Power of α	**Elements of GF(16)**
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

$\{\alpha^1, \alpha^2, \alpha^4, \alpha^8\}$ the first conjugate set

$\{\alpha^3, \alpha^6, \alpha^9, \alpha^{12}\}$ the second conjugate set

$\{\alpha^5, \alpha^{10}\}$ the third conjugate set

$\{\alpha^7, \alpha^{11}, \alpha^{13}, \alpha^{14}\}$ the fourth conjugate set

We continue by using the factorization method to find the minimal polynomial for each conjugate set:

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

$\{3,6,9,12\} \quad \Rightarrow \quad f_2(x) = (x-z^3)(x-z^3-z^2)(x-z^3-z)(x-z^3-z^2-z-1)$

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

$\{7,11,13,14\} \quad \Rightarrow \quad f_4(x) = (x-z^3-z-1)(x-z^3-z^2-z)(x-z^3-z^2-1)(x-z^3-1)$

We carefully compute each of the functions (in GF(2)), and by doing so, we find the minimal polynomial for each of the conjugate sets. We complete the GF(16) table as follows:

Now, since we have all the minimal polynomials for each conjugate set, we are ready to find the generator polynomial for our BCH(15, ?) code.(we wrote ? because the second parameter of BCH is 15-deg(g(x)), and so we need to find it)

Power of α	**Elements of GF(16)**	**Minimal Polynomials**
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	$z^3 + 1$	$x^4 + x^3 + 1$
15	1	$x + 1$

Table 2: Minimal Polynomials for each conjugate set

$$g(x) = \text{LCM}(f_1(x), f_2(x), \ldots, f_{2t}(x))$$

where $t = 2$

$$g(x) = \text{LCM}(f_1(x), f_2(x), f_3(x), f_4(x))$$

$$g(x) = \text{LCM}(x^4 + x + 1, x^4 + x + 1, x^4 + x^3 + x^2 + x + 1, x^4 + x + 1)$$

$$g(x) = (x^4 + x + 1) \cdot (x^4 + x^3 + x^2 + x + 1)$$

$$g(x) = x^8 + x^7 + x^6 + x^4 + 1$$

We have now found the generator polynomial, and so we know that we will use a BCH(15,7) code since the degree of the generator polynomial is 8, and the first parameter of the BCH is 15. 15-8=7. BCH(15,7) takes a codeword of 7 bits, and it adds 8 redundancy bits to it while encoding it. We go back now to our message and transform it into binary to find its polynomial form and encode it with the generator polynomial. The algorithm is the same for each character, we will work only with the first character to show how it works We start with the message "9887". For each character, we perform the following steps:

1. **Transform the Character to Binary:**

- For the character "9":

$$9 \Rightarrow 1001$$

- Add the required redundancy (7 bits total for each codeword):

$$1001 \Rightarrow 0001001$$

2. **Polynomial Representation:**
The polynomial representation of the character "9" is given by:

$$\text{code}(\text{"9"}) = \alpha^3 + \alpha^0$$

3. **Encode the Polynomial Using the Generator Polynomial:**
Using the generator polynomial $g(x) = \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1$, we encode the polynomial of the character:

$$\begin{aligned}\text{encoded}(\text{"9"}) &= \text{code}(\text{"9"}) \cdot g(x) \\ &= (\alpha^3 + \alpha^0)(\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1) \\ &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha^0\end{aligned}$$

4. **Binary Representation of Encoded Polynomial:**
In binary form, the encoded value for "9" is:

$$\text{encoded}(\text{"9"}) = 000111101011001$$

5. **Error Occurrence:**
Suppose that during transmission, some random errors occur. The received corrupted codeword might be:

$$\text{encodedCorrupt}(\text{"9"}) = 000\mathbf{0}1110\mathbf{0}011001$$

So our friend receives a wrong message. Keep in mind that the received message polynomial is of the form:

$$v(x) = c(x) + e(x)$$

We are now interested in finding the **locations** and **magnitude** of the errors. But since we are working over GF(2), all the magnitudes 1, and so we are only interested in the locations of the errors. To find the error location polynomial we start computing the syndromes. t=2 and so we have to compute in total 4 syndromes If the message received is correct/uncorrupted those syndromes should be 0. We perform the following calculations to find the values of $v(x)$ for different powers:

$v(x) = S_1 = 1 + \alpha^3 + \alpha^4 + \alpha^8 + \alpha^9 + \alpha^{10}$

We compute the coefficients with respect to GF(2) and the powers with respect to GF(16).

$S_1 = \alpha \neq 0$

$v(x^2) = S_2 = 1 + \alpha^6 + \alpha^8 + \alpha^{16} + \alpha^{18} + \alpha^{20}$

We compute the coefficients with respect to GF(2) and the powers with respect to GF(16).

$S_2 = \alpha^2 \neq 0$

$v(x^3) = S_3 = 1 + \alpha^9 + \alpha^{12} + \alpha^{24} + \alpha^{27} + \alpha^{30}$

We compute the coefficients with respect to GF(2) and the powers with respect to GF(16).

$S_3 = 0$

$v(x^4) = S_4 = 1 + \alpha^{12} + \alpha^{16} + \alpha^{32} + \alpha^{36} + \alpha^{40}$

We compute the coefficients with respect to GF(2) and the powers with respect to GF(16).

$S_4 = \alpha + 1 = \alpha^4$

Now by using the syndromes we've just calculated we try to find the error location polynomial using the formula explained in the last chapter.

$$\begin{pmatrix} S_1 & S_2 \\ S_2 & S_3 \end{pmatrix} \begin{pmatrix} b \\ a \end{pmatrix} = \begin{pmatrix} S_3 \\ S_4 \end{pmatrix}$$

$$\begin{pmatrix} \alpha & \alpha^2 \\ \alpha^2 & 0 \end{pmatrix} \begin{pmatrix} b \\ a \end{pmatrix} = \begin{pmatrix} 0 \\ \alpha^4 \end{pmatrix}$$

We compute the matrix ecuation by mulplying with the inverse of the matrix to find the second term of the multiplication. After the computations we find that :

$$b = \alpha^2, a = \alpha$$

We will use a_1, a_2 to compute the error locator polynomial, which has the form:

$$\lambda(x) = a_2 x^2 + a_1 x + 1$$

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 $\alpha^{15-2} = \alpha^{13}$.

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

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

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

$$\vdots$$

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

$$\vdots$$

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

So the errors are at position 6 and 11. Since we are working over GF(2) we are not interested in the magnitude of these errors (as exaplined before) because all the errors(if they exist) are equal to 1. Now we need to divide the corrected polynomial by the generator polynomial to find the message. In order to do that we can use a method **called ciclyc redundancy check**(if we are working with the binary form of the polynomial), or we can do it by using simple

polynomial division. We'll go ahead and use the second version for it is much easier to compute in this context.

Encoded polynomial for "9" is:

$$\begin{aligned}\text{encoded("9")} &= 000111101011001 \\ &= (001001) \cdot (000000111010001)\end{aligned}$$

Thus, the original vector polynomial for "9" is:

$$\frac{000111101011001}{000000111010001}$$

This should be computed using the **cyclic redundancy check**.

The encoded polynomial for "9" is:

$$\begin{aligned}\text{encoded("9")} &= (\alpha^3 + \alpha^0) \cdot (\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1) \\ &= \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha^0\end{aligned}$$

Thus, the original vector polynomial for "9" is:

$$\frac{\alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha^0}{\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1}$$

This should be computed using **simple polynomial division** with respect to GF(2)

Divide $\alpha^{11}+\alpha^{10}+\alpha^9+\alpha^8+\alpha^6+\alpha^4+\alpha^3+1$ by $\alpha^8+\alpha^7+\alpha^6+\alpha^4+1$:

$$\begin{array}{r|l}
 & \boxed{\alpha^3 + 1} \\
\hline
\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1 & \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^6 + \alpha^4 + \alpha^3 + 1 \\
 & \underline{\alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^7 + \alpha^3} \\
 & \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1 \\
 & \underline{\alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + 1} \\
 & 0
\end{array}$$

And since we know we are working with BCH(15,7), we know that the element that the receiver must receive must be 7 bits long and so instead of the binary vector polynomial of α^3+1 (which is **1001**), we must have **0001001**.

Remark 4.25. It is interesting to notice that when we are working with modular arithmetic(**exclusively** modulo 2) **addition**, **substraction** and **XOR** operation are the same thing/they have the same impact.

Proof.

$+$	0	1
0	0	1
1	1	0

−	0	1
0	0	1
1	1	0

XOR	0	1
0	0	1
1	1	0

□

Quick Recap
We encoded a message4.24, corrupted it4.24, corrected it4.24, and decoded it again**??**, ensuring safe passage for the data between the two machines.

5 Total perfect codes in graphs realized by commutative rings

[5]

5.1 Graphs realized by commutative rings

[4]

Proposition 5.1. *Consider R to be a unitary commutative ring (we denote with 1 the identity element for multiplication), and consider Z(R) be the set of zero divisors.*

Definition 5.2. Consider x a non-zero element of a ring R. If $\exists a \neq 0 \in R$ such that x· a=$\hat{0}$ then x is called a **zero divisors** of the ring R.

Definition 5.3. A ring without any zero divisors is called an **integral domain**.

*We associate a graph G(R) to the ring R with vertices $Z^*(R) = Z(R) - 0$ (the zero divisors of the ring without 0). $x \neq y, x, y \in Z(R)$ x and y are adjacent in G(R) if and only if xy=yx=$\hat{0}$. We say that x is the **annihilator** of y, and y is the annihilator of x. As a consequence, G(R) is empty if and only if R is an integral domain. The adjacency relation between x and y is always symmetric and reflexive, but it is generaly not transitive. For it to be transitive we would need a complete graph.*

Definition 5.4. Consider a, b, c as vertices and let $\sim$ be a relation on the set $A = \{a, b, c\}$.

* We say that $\sim$ is **reflexive** if $\forall a \in A$, $a \sim a$.

* Let $a \sim b$. We say that $\sim$ is **symmetric** if $b \sim a$. In other words, if $x \sim y$, then $y \sim x$.
* Let $a \sim b \sim c$. We say that $\sim$ is **transitive** if $a \sim c$. In other words, if $x \sim y \sim z$, then $x \sim z$.

A relation that is reflexive, symmetric, and transitive is called an **equivalence relation**.

Proof. Let R be a commutative ring with unity which is not an integral domain. Suppose that G(R) is **NOT** a complete graph, and suppose the relation of adjacency is transitive. Then $\forall x \in Z(R)$, x is adjacent with all the element within Z(R), which tells us that the graph is complete, which is a **contradiction** with our assumption. □

5.2 Total perfect codes in graphs

We start this section by talking about total perfect codes in graphs, which are realized as zero-divisor graphs. Let R be a ring, which realizes the zero-divisor-graph G(R), with the vertex set Z*(R). G(R) is said to admit a total perfect code if ∃ some set

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

(a set of vertices 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) 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 ∈ C(R) is a partition of the set Z*(R). As a remark, any total perfect code in G(R) contains and even number of vertices.
Let us dive now in some examples:

Example 5.5. Consider a graph G_1 as shown in figure 1, with the vertex set $V(G_1) = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$, and edge set $E(G_1) = \{(x_1, x_2), (x_1, x_3), (x_1, x_4), (x_1, x_6), (x_4, x_5), (x_4, x_7), (x_5, x_6), (x_6, x_7)\}$. For a ring $R = \mathbb{Z}_{12}$, we get the following $Z^*(R) = \{\hat{2}, \hat{3}, \hat{4}, \hat{6}, \hat{8}, \hat{9}, \hat{10}\}$ (this is the set of zero divisors of R, excluding the $\hat{0}$ element). And so, $|V(G_1)| = |Z^*(R)|$ (the number of vertices in the graph is equal

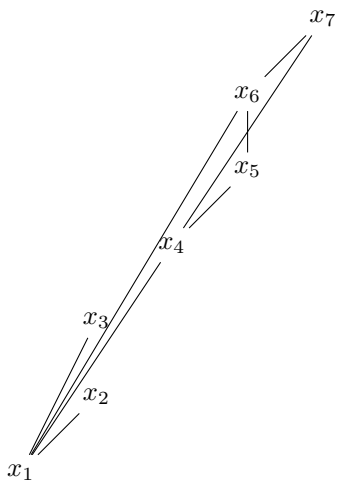


Figure 2: Graph G_1

to the number of non-zero zero divisors from R).
$\text{ann}(\hat{6}) = \{\hat{0}, \hat{2}, \hat{4}, \hat{6}, \hat{8}, \hat{10}\}$, and $\text{Ann}(\hat{6}) = \{\hat{2}, \hat{4}, \hat{8}, \hat{10}\}$.
$\text{Ann}(\hat{2}) = \{\hat{6}\}$, $\text{Ann}(\hat{3}) = \{\hat{4}, \hat{8}\}$, $\text{Ann}(\hat{4}) = \{\hat{3}, \hat{6}, \hat{8}\}$, $\text{Ann}(\hat{8}) = \{\hat{3}, \hat{6}\}$, $\text{Ann}(\hat{9}) = \{\hat{4}, \hat{8}, \hat{10}\}$, and finally $\text{Ann}(\hat{10}) = \{\hat{6}\}$. It is clear from figure 1, from the adjacencies in G_1 that $\text{Ann}(\hat{6}) = N(x_1)$, $\text{Ann}(\hat{2}) = N(x_2)$, $\text{Ann}(\hat{3}) = N(x_5)$, $\text{Ann}(\hat{4}) = N(x_4)$, $\text{Ann}(\hat{8}) = N(x_6)$, $\text{Ann}(\hat{9}) = N(x_7)$, $\text{Ann}(\hat{10}) = N(x_3)$. From all these relations, we conclude that $\mathbb{Z}_{12}$ realizes G_1 as its zero-divisor graph. We can find a subset $C(R) = \{\hat{6}, \hat{4}\}$, so that $\forall x \in Z^*(R), |\text{Ann}(x) \cap C(R)| = 1$. Therefore, $G(R)$ admits a total perfect code.

Example 5.6. Consider a graph G_2 as shown in figure 3 with vertex set:
$V(G_2) = \{y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}\}$ and edge set :
$E(G_2) = \{(y_1, y_2), (y_1, y_3), (y_1, y_4), (y_1, y_5), (y_1, y_6), (y_1, y_7), (y_1, y_8), (y_8, y_9), (y_8, y_6), (y_8, y_7), (y_8, y_{10}), (y_8, y_{11}), (y_6, y_{11}), (y_7, y_{11})\}$. For a ring $\mathbb{Z}_2 \times \mathbb{Z}_8$, we have $Z^*(\mathbb{Z}_2 \times \mathbb{Z}_8) = \{(\overline{0}, \overline{1}), (\overline{0}, \overline{2}), (\overline{0}, \overline{3}), (\overline{0}, \overline{4}), (\overline{0}, \overline{5}), (\overline{0}, \overline{6}), (\overline{0}, \overline{7}), (\overline{1}, \overline{0}), (\overline{1}, \overline{2}), (\overline{1}, \overline{4}), (\overline{1}, \overline{6})\}$. Therefore, $|V(G_2)| = |Z^*(\mathbb{Z}_2 \times \mathbb{Z}_8)|$ and $\text{Ann}(\overline{0}, \overline{6}) = N(y_1)$, $\text{Ann}(\overline{0}, \overline{1}) = N(y_2)$, $\text{Ann}(\overline{0}, \overline{3}) = N(y_3)$, $\text{Ann}(\overline{0}, \overline{5}) = N(y_4)$, $\text{Ann}(\overline{0}, \overline{7}) = N(y_5)$, $\text{Ann}(\overline{0}, \overline{2}) = N(y_6)$, $\text{Ann}(\overline{0}, \overline{6}) = N(y_7)$, $\text{Ann}(\overline{0}, \overline{4}) = N(y_8)$, $\text{Ann}(\overline{1}, \overline{2}) = N(y_8)$, $\text{Ann}(\overline{1}, \overline{6}) = N(y_{10})$, $\text{Ann}(\overline{1}, \overline{4}) = N(y_{11})$. From all these relations we conclude that $\text{Z}_2 \times \mathbb{Z}_8 \textit{realizes} \text{G}_2$ as its zero divisor graph. It is easy to verify from $\text{G}(\mathbb{Z}_2 \times \mathbb{Z}_8)$ that there is no subset $C(\mathbb{Z}_2 \times \mathbb{Z}_8) \subseteq Z^*(\mathbb{Z}_2 \times \mathbb{Z}_8)$ such that for all $x \in Z^*(\mathbb{Z}_2 \times \mathbb{Z}_8)$, $|\text{Ann}(x) \cap C(\mathbb{Z}_2 \times \mathbb{Z}_8)| = 1$, and so, the graph does not admit total perfect codes.

In the first example we see that the graph realized by a ring Z_{12}

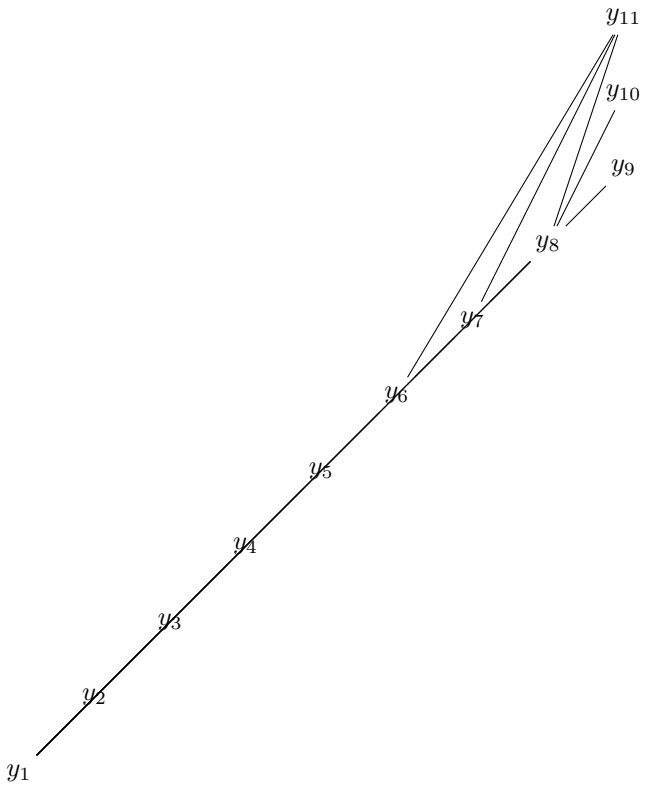


Figure 3: Graph G_2

admits a total perfect code, while in the second example, the graph realized by a ring $Z_2 \times Z_8$ does not admit a total perfect code. It is interesting to characterize rings of which the realized zero-divizor graphs admit total perfect codes.

In the section below we start to characterize rings by their zero divisor graph, and their properties, studying most importantly which rings(which zero divisor graphs of those rings) actually admit total perfect codes

Definition 5.7. A ring R is called *reduced* if it contains no nilpotents elements. A element x from R, is called nilpotent if it satisfies the following property:

$$x^n = 0 \text{ for some } n \in N$$

Example 5.8. $3^2 = 0, in Z_9$, 3 is a nilpotent element of order 2.
Z_5 is a reduced ring

Proposition: Consider R a finite commutative local ring with unity. G(R) admits a total perfect code if and only if G(R) has at least one vertex of degree 1.
Proof:Say G(R) admits a total perfect code, denoted as C(R). R is a **local** ring and so, by [4] there $\exists x \in Z*(R)$, adjacent to all the vertices contained in G(R). This means that x ∈ C(R) (no matter what x we choose). If the zero divisor graph has no degree 1 vertices, then that means that every vertex of G(R) has degree **more than 1**. Explicitely, the vertex y ∈ C(R) different from x, is adjacent to another vertex z ∈ $Z^*(R)$. This means that yy=0 for some z ≠ x. We also have xz=0 since x adjacent to all vertices of the graph. Therefore we have $|Ann(x) \cap C(R)| \forall x \in Z^*(R)$.

Suppose G(R) has at least one vertex of degree 1 (let it be a). Clearly a is adjacent to x, since x is adjacent to all vertices of G(R)(except x itself), but a is not adjacent to any vertex of G(R). The vertices x and a cover all the vertices from G(R), and so, we conclude that there exists a subset C(R) of Z*(R) so that $|Ann(x) \cap C(R)| = 1$ for all x in Z*(R)
If the graph resulted by eliminating the vertex a, along with all the edges associated with the vertex a, then, a is called a **cut-vertex**

Corollary 5.9. *Consider R a ring, any ring which is **not** a integral domain. If in G(R) we have a vertex x, so that x^2=0, then, for all other vertices(a, b, c.. etc), we have x adjacent with xa, xb, xc, and more precise : xa, xb, xc.. ⊆ ann(x).*

Proof. if x^2=0, then, precisely no matter what other vertex y we choose , we have x(xy)=0, because we have xxy=x^2y=0·y=0. Anal-

ogously for all non-nilpotent items we have that if x^n=0, xx..x(n times)y=0, since x^n=0. □

Proposition 5.10. *Consider R to be a finite local ring. Then G(R) admits a total perfect code if and only if we are in one of the following situations:*

1. *There is some $x \in R$ such that $|ann(x)| = 2$, while $|Z(R)| \geq 3$;*
2. *R is isomorphic to $\mathbb{Z}_9$ or $\mathbb{Z}_3[X]/(X^2)$.*

Proof. : Suppose $G(R)$ admits a total perfect code. Then $G(R)$ has one degree vertices as shown before. Let x be a vertex of degree 1 inside $G(R)$. Then either $G(R)$ has cut-vertices, or $G(R)$ has only two vertices, implying that R is isomorphic with $\mathbb{Z}_9$ or $\mathbb{Z}_3[X]/(X^2)$. Consider y a vertex adjacent to x. It is clear that y is a cut vertex of $G(R)$, since if we eliminate the vertex from the graph, being the only vertex that is adjacent to x, then x remains alone, leaving the graph unconnected. We have that $ann(x) = \{0, y\}$ or $ann(x) = \{0, y, x\}$. If $ann(x) = \{0, y, x\}$, then $x^2 = 0$. Keeping in mind that $x(x + y) = 0$, we have that $x + y \in ann(x)$. Since the only vertex adjacent to x is y, $x + y$ must be 0 (or x, but since y is not 0 is not viable). That means $x = -y$, however since x is of degree 1, the entire graph must consist of only the vertices x and y. This is a contradiction, since a graph must have at least 3 vertices to have a cut-vertex.

□

Proposition 5.11. *Consider R to be a unitary commutative ring and say R realizes G(R) on $|Z^*(R)| \geq 2$ vertices as its zero divisors graph. Then G(R) admits a total perfect code if and only if $|Z^*(R)| = 2$*

Proof. For $|Z^*(R)| = 2$, G(R) is a simple path which admits a "trivially" perfect code (a code made up of only one vertex). If we consider the case when $|Z^*(R)|2$ then we have $|Ann(x) \cap C(R)|1$, no matter what C(R), or x we choose, since the graph is complete. Thus if $|Z^*(R)| \geq 3$, then G(R) does not admit a total perfect code. □

Proposition 5.12. *Consider R a unitary commutative ring and let G(R) be a bipartite graph representing the zero divisors of R. Then G(R) admits a total perfect code if and only if the graph is complete.*

Proof. Let G(T) be the complete bipartite graph realized by R of order n+m where n, m ∈ N, with bipartition of $Z^(R)$ as $Z_1^*(R) = x_1, x_2...x_n$ and $Z_2^*(R) = y_1, y_2...y_m$. For any two vertices $x_i \in Z_1^*(R)$ and $y_j \in Z_2^*(R)$, with $1 \leq i \leq n$ and $1 \leq j \leq m$, with the code C(R)=x_i, y_j. We see that we can check that for all a $\in Z_1^*(R)$ and b $\in Z_2^*(R)$, we have $|Ann(a) \cap C(R)| = 1$ and $|Ann(b) \cap C(R)| = 1$. Therefore C(R) is a total perfect code for G(R). Suppose that a

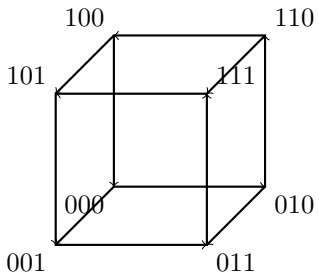
bipartite graph G(R) realized by R with the partitions sets of $Z^*(R)$ as $Z_1^*(R)$ and $Z_2^*(R)$, admits a total perfect code, C(R). It is easy to understand why C(R) must contain a vertex from $Z_1^*(R)$ and another one from $Z_2^*(R)$, otherwise we would have $x_p x_q = 0$, for some $x_p, x_q \in Z_1^*(R)$, or $y_s y_r = 0$ for some $x_s, x_r \in Z_2^*(R)$. Which is impossible, since we are talking about adjancies inside a partite set. If G(R) is not complete, then there exists some vertex x $\in Z^*(R)$ in some partite set which is not adjacent to all vertices of other partite set. If x $\in Z_1^*$, then x=x_i, for some $x_i \in Z_1^*$. By our assumption, x_i is not adjacent to all vertices $y_j \in Z_2^*$, which implies $|Ann(x_i) \cap C(R)| = \emptyset$, a contradiction. Similarly if x=y_j for some $y_j \in Z_2^*(R)$, then $|Ann(y_j) \cap C(R)| = \emptyset$, which once again contradicts our initial assumption, that C(R) is a total perfect code for G(R). And so we conclude that if G(R) a bipartite graph that admits a total perfect code, then G(R) must be complete. □

6 Total perfect codes in Hamming graphs

It is interesting to notice that total perfect codes may also appear also in the graphs realized by error correcting codes, since many error correction codes are defined over finite fields, and since we know that a field is (mathematically speaking) nothing more than a ring with some extra properties.

Example 6.1. Consider the **Hamming graph(3,2)**.

* 3 represents the number of "bits" (in our case, but it does not necesarily need to be bits, it can be in any base, but we have bits since the number of symbols in the alphabet is 2).
* 2 is the number of symbols in the alphabet (0 and 1, we are interested in working with bits/ working in binary)



Take for example the pairs(000,111), (010,101), (100,011), (001,110). It is easy to check that these pairs form a total perfect code. As a remark, those are the only total perfect codes in the Hamming graph(3,2)

7 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.

References

[1] IIT Delhi July 2018. Introduction to bch codes: Generator polynomials. *https://www.youtube.com/watch?v=16aggpH4Meg*, 2018.

[2] IIT Delhi July 2018. Multiple error correcting bch codes, decoding of bch codes. *https://www.youtube.com/watch?v=UPvJ2J2qGRkt=574s*, 2018.

[3] 3Blue1Brown. But what are hamming codes? the origin of error correction. *https://www.youtube.com/watch?v=X8jsijhllIAt=519s*, 2021.

[4] David F. Anderson and Philip S. Livingston. The zero-divisor graph of a commutative ring. 1998.

[5] Rameez Raja. Total perfect codes in graphs realized by commutative rings. 2021.

[6] Socratica. Ideals in ring theory (abstract algebra). *https://www.youtube.com/watch?v=F0wA0xLZSQ8t=176s*, 2020.

[7] vcubingx. What are reed-solomon codes? how computers recover lost data. *https://www.youtube.com/watch?v=1pQJkt7-R4Qt=792s*, 2022.