AN INTRODUCTION TO ALGEBRAIC CODING THEORY

Project Report Submitted To

MAHATMA GANDHI UNIVERSITY

In partial fulfillment of the requirement

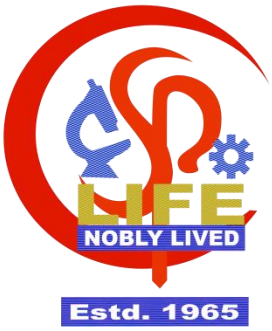
For the award of

THE MASTER DEGREE IN MATHEMATICS

BY

REMONA BASTIN

M.S.c (IV Semester) Reg No. 180011015187



DEPARTMENT OF MATHEMATICS

ST PAUL'S COLLEGE, KALAMASSERY

2018 - 2020

CERTIFICATE

This is to certify that the project entitled **"AN INTRODUCTION TO ALGEBRAIC CODING THEORY"** is a bonafide record of studies undertaken by REMONA BASTIN (Reg no. 180011015187), in partial fulfillment of the requirements for the award of M.Sc. Degree in Mathematics at Department of Mathematics, St. Paul's College, Kalamassery, during 2018–2020

Dr Savitha K S
Head of Department of Mathematics

Dr Pramada Ramachandran
Department of Mathematics

Examiner :

DECLARATION

I **Remona Bastin** hereby declare that the project entitled **"An Introduction To Algebraic Coding Theory"** submitted to department of Mathematics St. Paul's College, Kalamassery in partial requirement for the award of M.Sc Degree in Mathematics, is a work done by me under the guidance and supervision of **Dr. Pramada Ramachandran,** Department of Mathematics, St. Pauls's College , Kalamassery during 2018 – 2020.

I also declare that this project has not been submitted by me fully or partially for the award of any other degree, diploma, title or recognition earlier.

Date: **Remona Bastin**

Place :Kalamassery

ACKNOWLEDGEMENT

I express my heartfelt gratitude to my guide **Dr. Pramada Ramachandran ,** Department of Mathematics, St Paul's College, Kalamassery for providing me necessary stimulus for the preparation of this project.

I would like to acknowledge my deep sense of gratitude to **Dr. Savitha K.S**, Head of the Department of Mathematics, St Paul's College Kalamassery and all other teachers of the department and classmates for their help at all stages.

I also express my sincere gratitude to **Prof. Valentine D'Cruz**, Principal, St. Paul's College, Kalamassery for the support and inspiration rendered to me in this project report.

I also wish to express my sincere thanks to all the faculty members of Department of Mathematics for the help and encouragement to bring this project a successful one

Kalamassery **Remona Bastin**

AN INTRODUCTION TO ALGEBRAIC CODING THEORY

CONTENTS

Chapter 1

Introduction

1.1Passing Notes

Suppose you are sitting in your English Class and you are handed a note which reads ‘THIS CLPSS IS BORING’. You immediately notice the error in the note, and in addition to wondering how the author of the note ever got into college, you wonder what the actual intended message was. Very quickly you replace ‘CLPSS’ with ‘CLASS’ and the meaning becomes clear. You have just applied some coding theory in your English class by detecting and correcting an error, but how did you do it? In order to detect and correct the error so quickly, you made three assumptions:

1. That a correct word will be an English word.

2. That the correct word contains five letters.

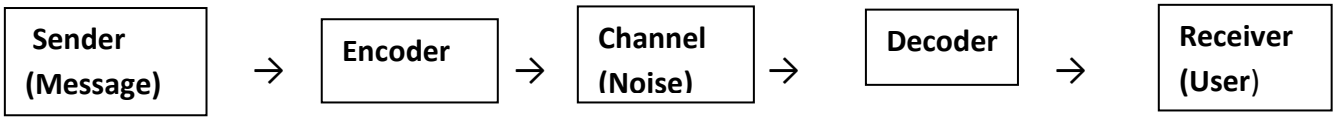
3. That it is more likely that only one letter was incorrect as opposed to two or more.

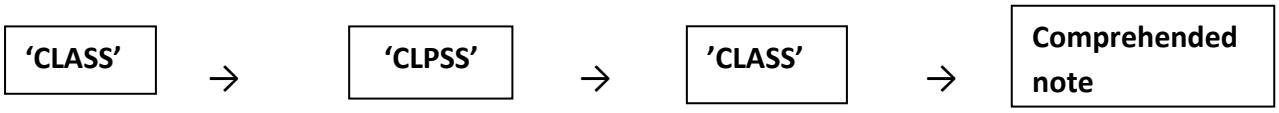
For now, we will ignore the assumption that the correct word should make sense in the context of entire message. Assumption 2 allowed us to make sense of the errant message, but it does have a weakness. In the above example ‘CLASS’ was the only English word we could make by changing one letter in ‘CLPSS’, but what if the received word was ‘CLASK’? Now by changing one letter we can make words like ‘FLASK’ and ‘CLASP’ as well as ’CLASS’. Using only our assumptions (1 and 2) we cannot justify choosing one word over

another, that is, in this case we cannot correct the error. In some cases our two assumptions may not even let us catch errors, for example if the received word was an English word, but the wrong one, say ‘GLASS’.

The above situation can be thought of as a simple example of interpreting a message with a decoding algorithm. This is an absolutely crucial process in fields like telecommunication, electrical engineering, and computing where data is sent over a noisy “channel” where it may be altered before it is received. The main aims of Coding Theory are to detect and correct transmission errors as thoroughly and rapidly, and thus as efficiently as possible. Note that Coding Theory is NOT cryptography, that is it doesn’t protect data from malicious eyes.

The typical model of this system is as follows:





We will now begin a mathematical treatment of Coding Theory in order to understand the development of this fascinating (and useful!) field of study

1.2 BASIC ASSUMPTIONS

We state some fundamental definitions and assumptions which we will apply throughout.

1.2.1 Definition

- In many cases, the information to be sent is transmitted by a sequence of *zeros* and *ones*.
- We call a 0 or 1 a *digit*.
- A *word* is a sequence of digits.
- The *length of a word* is the number of digits in the word. Thus 0110101 is a word of length seven. A word is transmitted by sending its digits, one after to other, across a binary channel.
- The term 'binary' refers to the fact that only two digits 0 and 1 are used. Each digit is transmitted mechanically, electrically, magnetically or otherwise by one of two types of easily differentiated pulses.
- A binary code is a set C of words. The code consisting of all words of length two is C= {00, 10, 01, 11}
- A block code is a code having all its word of the same length; this number is called the length of a code. We will consider only block codes. So, for us the term code will always mean a binary block code.
- The word that belong to a given code C_0, will be called code-words. We shall denote the number of code-words in a code C by $|C|$.
- *Self-complementing* binary codes are those whose members complement on themselves. For a binary code to become a self-complementing code, the following two conditions must be satisfied:

- The complement of a binary number should be obtained from that number by replacing 1’s with 0’s and 0’s with 1’s
- The sum of the binary number and its complement should be equal to decimal 9.

- In *weighted codes*, each digit is assigned a specific weight according to its position. ... Examples:8421,2421,7421 are all weighted codes.
- *Non-weighted* codes: The non-weighted codes are not positionally weighted.In other words codes that are not assigned with any weight to each digit position.
- A *linear code* is a type of block code used in error detection and correction in transmitting bits or data on a communications channel. A linear code of length n transmits blocks containing n bits (symbols).

We also need to make certain basic assumptions about the channel. These assumptions will necessarily shape the theory that we formulate.

- The first assumption is that a code-word of length n consisting of 0's and 1's is received as a word of length n consisting of 0's and 1's, although not necessarily the same as the word that was sent.
- The second is that there is no difficulty identifying the beginning of the first word transmitted. Thus, if we are using code-words of length 3 and receive 011011001 we know that the words received are in order, 011, 011, 001 . This assumption means, again using length 3, that the channel cannot deliver 01101 to the receiver, because a digit has been lost here.
- The final assumption is that the noise is scattered randomly as opposed to being in clumps called bursts. That is, the probability of any one digit being affected in transmission is the same as that of any other digit and is not influenced by errors made in

neighbouring digits. This is not a very realistic assumption for many types of noise such as lightning or scratches on compact discs. We shall eventually consider this type of noise.

- In a perfect, or noiseless, channel, the digit sent, 0 or 1, is always the digit received. If all channels were perfect, there would be no need for coding theory. But fortunately (or unfortunately, perhaps) no channel is perfect; every channel is noisy. Some channels are less noisy or more reliable, than others.
- A binary channel is symmetric if 0 and 1 are transmitted with equal accuracy; that is the probability of receiving the correct digit is independent of which digit,0 or 1, is being transmitted

1.3. CORRECTING AND DETECTING ERROR PATTERNS

We consider now the possibilities of correcting and detecting errors. In this section we intend to develop an intuitive understanding of the concepts involved in correcting and detecting errors, while a formal approach is adopted in later sections.

Suppose a word is received that is not a codeword. Clearly some error has occurred during the transmission process, so we have detected that an error (perhaps several errors) has occurred. If however a codeword is received, then perhaps no errors occurred during transmission, so we cannot detect any error.

The concept of correcting an error is more involved. As in the introduction when we were inclined to correct 'gub' to 'gun' rather than to 'rat', we appeal to intuition to suggest that any received word should be corrected to a codeword that requires a few changes as possible. (In a later

section we show that the probability that such a codeword was sent is at least as great as the probability that any other codeword was sent). To consolidate these ideas, we shall discuss some particular codes. Notice that our assumption that no digits are lost or created in transmission precludes decoding 'gub' to 'fire truck'.

Example 1.3.1

Let C_1 = {00, 01, 10, 11} Every received word is a codeword and so C_1 cannot detect any errors. Also C_1 corrects no errors since every received word requires no changes to become codeword.

Example 1.3.2

Modify C_1 by repeating each codeword three times. The new code is

$$C_2 = \{000000, 010101, 101010, 111111\}.$$

This is an example of a repetition code. Suppose that 110101 is received. Since this is not a codeword we can detect that at least one error has occurred. The codeword 010101 can be formed by changing one digit, but all other codewords are formed by changing more than one digit. Therefore we expect that 010101 was the most likely codeword transmitted, so we correct 110101 to 010101. (A codeword that can be formed from a word w with the least number of digits being changed is called a closest codeword; this idea is formalized later.) In fact if any of the codewords, c element of C_2, is transmitted and one error occurs during transmission, then the unique closest codeword to the received word is c; so any single error results in a word that we correct to the codeword that was transmitted.

1.4 . BCD Codes

During the earliest period of development of this subject, the binary-coded decimal (BCD) codes (or systems, as they were called then) were popular, and some of them have been used even in modern times These codes were based on the premise that, in addition to the number systems with base b, there are other special number systems that are hybrid in nature and are useful in computation, as computer inputs and outputs are mostly in decimal notation.

❖ Four-Bit BCD Codes.

These codes are defined as follows.

a) **8421 code**. A number system in base b requires a set of b distinct symbols for each digit. In computing the decimal (b = 10) and the binary (b = 2) number systems we need a representation or coding of the decimal digits in terms of binary symbols (called bits). This requires at least four bits, and any 10 out of the 16 possible permutations of these four bits represent the decimal digits. A systematic arrangement of these 10 combinations is given in Table (a), where d denotes the decimal digit. In the BCD code, the weights of the positions are the same as in the binary number system, so that each decimal digit is assigned a combination of bits, which is the same as the number represented by the four components regarded as the base 2 number. This particular code is also called direct binary coding. The nomenclature 8421 follows

from the weights assigned by the leftmost 1 in the successive bits in this representation.

The 8421 code uses four bits to represent each decimal digit. For example, the number 697 is represented by the 12-bit number 0110 1001 0111, which has 3 four-bit decades. Although this number contains only 0s and 1s, it is not a true binary number because it does not follow the rules for the binary number system. In fact, by base conversion rules we have $(697)_{10} = (1010111001)_2$. Thus, it is obvious that arithmetic operations with the 8421 code or any other BCD code would be very involved. However, as we shall soon see, it is quite easy for a computer program to convert to true binary, perform the required computations, and reconvert to the BCD code.

A digital computer can be regarded as an assembly of two-state devices as it computes with 0's and 1's of the binary system. On the other hand, we are accustomed to decimal numbers. Therefore, it is desirable to build a decimal computing system with two-state devices. This necessity has been responsible for the development of codes to encode decimal digits with binary bits. A minimum of four bits are needed.

The following features are desirable in the choice of a code:

(i) Ease in performing arithmetical operations;

(ii) Economy in storage space;

(iii) Economy in gating operations, error detection, and error correction;

(iv) Simplicity.

b) **Excess-3 code**. This code represents a decimal number d in terms of the binary equivalent of the number d + 3. It is a self-complementing but not

a weighted code, and since it does follow the same number sequence as binary, it can be used with ease in arithmetical operations.

c) **2421 code**. This code is a self-complementing weighted code, commonly used in bit counting systems. Other weighted codes are: 5421 code, 5311 code, and 7421 code, which are presented in Table (a).

Table (a) : BCD Code

D	8421	Excess -3	2421	5421	5311	7421
0	0000	0011	0000	0000	0000	0000
1	0001	0100	0001	0001	0001	0111
2	0010	0101	0010	0010	0011	0110
3	0011	0110	0011	0011	0100	0101
4	0100	0111	0100	0100	0101	0100
5	0101	1000	1011	1000	1000	1010
6	0110	1001	1100	1001	1001	1001
7	0111	1010	1101	1010	1010	1000
8	1000	1011	1110	1011	1011	1111
9	1001	1100	1111	1100	1100	1110

Chapter -2

Digital Arithmetic

In this chapter we describe constructive procedures in the form of error detecting, correcting, and decoding codes that are used for encoding messages being transmitted over noisy channels. The goal of such codes is to decode messages with no error rate or the least error rate. Most of these codes involve certain basic iterative procedures for simple error-correcting codes, which are described in detail in the following chapters. During the past half century, coding theory has shown phenomenal growth, with applications in areas such as communication systems, storage technology, compact disc players, and global positioning systems. Before we enter into these developments, we must review some basic digital logic and related rules that are useful for the development of the subject.

2.1 Number Systems

In addition to the decimal number system, we will discuss binary, ternary, octal, duodecimal, and hexadecimal systems.

2.1.1 Decimal Numbers.

This system, also known as the base-10 system, uses ten symbols (units) 0 through 9 and positional notation to represent real numbers in a systematic manner. The decimal (from Latin decimus, meaning ‘tenth’) system is also known as denary from Latin denarius which means the ‘unit of ten’. The real numbers are created from the units by assigning different weights to the

position of the symbol relative to the left or right of the decimal point, following this simple rule: Each position has a value that is ten times the value of the position to the right. This means that each positional weight is a multiple of ten and is expressible as an integral power of ten.

The positional scheme can be expressed as follows:

$$10^{p} \ldots\ldots.10^{3}\,10^{2}10^{1}\,10^{0}\,.\,10^{-1}\,10^{-2}\,10^{-3}\ldots\ldots\ldots\ldots10^{-q}$$

↑

Decimal point

Figure 2.1.1 Positional scheme of the decimal number system

TABLE 2.1.1 Different Number Systems

Decimal	Binary	Octal	Duodecimal	Hexadecimal
0	0	0	0	0
1	01	1	1	1
2	10	2	2	2
3	11	3	3	3
4	100	4	4	4
5	101	5	5	5
6	110	6	6	6
7	111	7	7	7
8	1000	10	8	8
9	1001	11	9	9
10	1010	12	A	A

11	1011	13	B	B
12	1100	14	10	C
13	1101	15	11	D
14	1110	16	12	E
15	1111	17	13	F

Example 2.1.1.

The binary number 101111011010 is converted to octal as follows:

Binary	101	111	011	010
	↓	↓	↓	↓
Octal	5	7	3	2

Hence, $(101111011010)_2 = (5732)_8$. Note that the leading zeros are added to the remaining leftmost one or two digits without affecting the binary number in order to complete the leftmost 3-bit binary digit. The above process can easily be reversed. If an octal number is given, say (1534)8, then the equivalent

Binary number is found as follows:

Octal	1	5	3	4
	↓	↓	↓	↓
Binary	001	101	011	100

Hence, $(1534)_8 = (001101011100)_2 = (1101011100)_2$, after discarding the two leading zeros.

2.2 Boolean and Bitwise Operations

The distinction between Boolean logical and bitwise operations is important. This section is devoted to these two topics, which play a significant role in the construction of different codes.

2.2.1 Boolean Logical Operations.

The truth tables for classical logic with only two values, 'true' and 'false', usually written T and F, or 1 and 0 in the case of the binary alphabet A = {0,1}, are given in Table 2.2.1 for most commonly used operators AND, OR, XOR, XNOR, IF-THEN, AND THEN-IF. The operator NOT is defined by NOT 0=1, and NOT 1=0. The others are:

Table 2.2.1. Boolean Logical Operators

P	Q	AND	OR	XOR	XNOR	IF-THEN	THEN-IF
0	0	0	0	0	1	1	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	0	1
1	1	1	1	0	1	1	1

2.2.2 Bitwise Operations.

A bitwise operation is carried out by operators like NOT, AND, OR, AND XOR, which operate on binary numbers or one or two bit patterns at the level of their individual bits. These bitwise operators are defined as follows.

NOT (¬). This operator, also known as the complement, is a unary operation that performs a logical negation at each bit. Thus, digits that were 0 become 1, and conversely.

For example,

NOT 0110 = 1001. In certain programming languages, such as C or C++, the bitwise NOT is denoted by ∼ (tilde). Caution is needed not to confuse this bitwise operator with the corresponding logical operator '!' (exclamation point), which treats the entire value as a single Boolean, i.e., it changes a true value to false, and conversely. Remember that the 'logical NOT' is not a bitwise operation.

AND (& or ∧). This bitwise operation takes two binary representations of equal length and operates on each pair of corresponding bits. In each pair, if the first bit is 1 and the second bit is 1, then the result is 1; otherwise the result is 0. This operator, as in the C programming languages, is denoted by '&' (ampersand), and must not be confused with the Boolean 'logical AND' which is denoted by '&&' (two ampersands). An example is: 0101 & 0011 = 0001. The arithmetic operation '+' and bitwise operation '& ' are given side-by-side in Table 2.2.2. In general, the expressions x + y and x & y will denote the arithmetic and bitwise addition of x and y, respectively.

OR (|). This operation takes two binary representations of equal length and produces another one of the same length by matching the corresponding bit, i.e., by matching the first of each, the second of each, and so on, and performing the logical inclusive OR operation on each pair of corresponding bits. Thus, if in each pair the first bit is 1 or the second bit is 1 or both, then the result is 1; otherwise it is 0. Thus, for example, 0101 OR 0011 = 0111. In C

programming languages the bitwise OR is denoted by | (pipe), and it must not be confused with the logical OR which is denoted by ∨ (from Latin vel) or by || (two pipes)

XOR (⊕). This bitwise operator, known as the bitwise exclusive-or, takes two bit patterns of equal length and performs the logical XOR operation on each pair of the corresponding bits. If two bits are different, then the result is 1; but if they are the same, then the result is 0.

Thus, for example, 0101 ⊕ 0011 = 0110.

In general, if x,y,z are any items, then

(i) $x \oplus x = 0$,

(ii) $x \oplus 0 = x$,

(iii) $x \oplus y = y \oplus x$, and

(iv) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

In C programming languages, the bitwise XOR is denoted by ⊕.

Table 2.2.2 Arithmetic and Bitwise Operations

Bitwise Operations						Arithmetic and Bitwise Operations					
P	Q	AND	OR	XOR	‖	p	q	+	AND	OR	XOR
0	0	0	0	0	‖	0	0	0	0	0	0
0	1	0	0	1	‖	0	1	1	0	0	1
1	0	0	0	1	‖	1	0	1	0	0	1
1	1	1	1	0	‖	1	1	10	1	1	0

The bitwise XOR operation is the same as addition mod 2. The XOR function has the following properties, which hold for any bit values (or strings) a,b, and c:

Property 1.

$a \oplus a = 0$;

$a \oplus 0 = a$; $a \oplus 1 = \sim a$, where $\sim$ is bit complement;

$a \oplus b = b \oplus a$;

$a \oplus (b \oplus c) = (a \oplus b) \oplus c$;

$a \oplus a \oplus a = a$,

and if $a \oplus b = c$, then $c \oplus b = a$ and $a \oplus a = b$.

Property 2.

As a consequence of Property 1, given $(a \oplus b)$ and a, the value of the bit b is determined by $a \oplus b \oplus a = b$. Similarly, given $(a \oplus b)$ and b, the value of a is determined by $b \oplus a \oplus b = a$. These results extend to finitely many bits, say a,b,c,d, where given $(a \oplus b \oplus c \oplus d)$ and any 3 of the values, the missing value can be determined. In general, for the n bits $a_1, a_2, \ldots, a_n$, given $a_1 \oplus a_2 \oplus \cdots \oplus a_n$ and any $(n - 1)$ of the values, the missing value can be easily determined.

Property 3. A string **s** of bits is called a symbol. A very useful formula is $\mathbf{s} \oplus \mathbf{s} = 0$ for any symbol s.

1.2.3 Applications.

Some applications involving the above bitwise operations are as follows:

The bitwise AND operator is sometimes used to perform a bit mask operation, which is used either to isolate part of a string of bits or to determine whether a particular bit is 1 or 0. For example, let the given bit pattern be 0011; then, to determine if the third bit is 1, a bitwise AND operation is performed on this bit pattern and another bit pattern containing 1 in the third bit. Thus, 0011 and 0010 = 0010. Since the result is non-zero, the third bit in the given bit pattern is definitely 1. The name 'bit masking' is analogous to use masking tape to mask or cover the parts that should not be changed.

The bitwise AND operator can be combined with the bitwise NOT to clear bits. Thus, consider the bit pattern 0110. In order to clear the second bit, i.e., to set it to 0, we apply the bitwise NOT to a arbitrary bit pattern that has 1 as the second bit, followed by the bitwise AND to the given bit pattern and the result of the bitwise NOT operation. Thus, [NOT 0100] AND 0110 = 1011 AND 0110 = 0010.

The bitwise OR is sometimes applied in situations where a set of bits is used as flags. The bits in a given binary number may each represent a distinct Boolean variable. By applying the bitwise OR to this number, together with a bit pattern containing 1, will yield a new number with that set of bits. As an example, given the binary number 0010, which can be regarded as a set of four flags, where the first, second, and fourth flags are not set (i.e., they each have value 0) while the third flag is the set (i.e., it has value 1), the first flag in this given binary number can be set by applying the bitwise OR to another value with first flag set, say 1000. Thus, 0010 OR 1000 = 1010. This technique is used

to conserve memory in programs that deal with a large number of Boolean values.

The bitwise XOR operation is used in assembly language programs as a short-cut to set the value of a register to zero, since operating XOR on a value against itself always yields zero. In many architectures this operation requires fewer CPU clock cycles than the sequence of operations that are needed to load a zero value and save it to the registers. The bitwise xor is also used to toggle flags in a set of bits. For example, given a bit pattern 0010, the first and the third bits may be toggled simultaneously by a bitwise XOR with another bit pattern with 1 in the first and the third bits, say 1010. Thus, 0010 $\oplus$ 1010 = 1000.

2.3 Residues, Residue Classes, and Congruences

For each pair of integers n and b, b > 0, there exists a pair of integers q and r such that $n = bq + r, 0 \leq r < b$.

The quantity r is called the residue of n modulo b and is denoted (in Gaussian notation) by $b|n$. For example $5|15=0$, $5|17=2$. Further if $n \geq 0$ then $b|n=r$ is the remainder, and q is the quotient when n is divided by b. The quantities q and r are unique (proof of uniqueness follows from the division algorithm)

Consider the class in which a comparison is made of the remainders when each of the two integers n and m are divided by b. If the remainders are the same then $b|(n-m)$ and we say that the two numbers n and m have the same residue modulo b, so that n and m differ by an integral multiple of b. In this case we say that n and m are congruent modulo b and write $n \equiv m \pmod{b}$.

The symbol ≡ is an equivalence relation (with respect to a set); that is, it is a relation R between the elements of a set such that if α and β are arbitrary elements, then either α stands in a relation R to β (written as α R β), or it does not.

Moreover, R has the following properties:

(i) α R α (reflexive);

(ii) if α R β, then β R α (symmetric); and

(iii) if α R β and β R ϒ, then α R ϒ (transitivity)

The equality between numbers is an equivalent relation for either α = β, or α ≠β; α = α; if α = β, then β = α; and if α = β and β = ϒ, then α = ϒ. Other examples are congruency of triangles, similarity of triangles, parallelism of lines, children having the same mother, or books by the same author. The congruency n ≡ m (mod b) possesses the above three properties. In fact, we have

Theorem 2.3.1.

Congruence modulo a fixed number b is an equivalence relation.

Proof. There are three cases

(i) b|(n-n) so that n≡n (mod b)

(ii) If b|(n-m), then b(m-n); thus if n≡m (mod b), then m≡n (mod b)

(iii) If b(n-m) and b(m-l), then n-m≡kb, m-l ≡jb where k and j are integers. Thus, n − l = (k + j)b, i.e., if n ≡ m (mod b) and m ≡ l (mod b), then n ≡ l (mod b).

Lemma 2.3.2:

If a | bc and (a,b)=1 then a | c

Proof.

If (a,b) = 1, then there exist integers x and y such that ax+by = 1. Multiply both sides of the equality by c. Then acx+bcy = c, and a divides both ac and bc. Hence a divides c.

Theorem 2.3.3.

The following results are true:

(i) If m ≡ n (mod b) and u ≡ v mod (b), then the following congruencies hold:

(a) m + u ≡ n + v (mod b),

(b) mu ≡ nv (mod b),

(c) k m ≡ k n (mod b) for every integer k;

(ii) if k m ≡ k n (mod b) and (k,b) = d, then m ≡ n (mod$\frac{b}{d}$), where (k,n) = d means d is the g.c.d. of k and b.

(iii) If f(x) is a polynomial with integral coefficients and m ≡ n (mod b), then f(m)≡ f(n) (mod b).

Chapter -3

Hamming Codes

3.1 Error Correcting Codes

Error detection and error correction are integral parts of many high-reliability and high-performance computer and transmission/storage devices. In data storage systems, memory caches are used to improve system reliability. The cache is generally placed inside the controller between the host interface and the disk array. Any reliable cache memory design must include errorcorrection code (ECC) functions to safeguard the loss of data. Similarly, ECC is an important design aspect of many communication applications, such as satellite receivers. The significance of ECC lies in performance and cost efficiency by correcting any error and avoiding repeated retransmission of data. When a message or data is transmitted through a channel, the data received depends on the properties of the resulting errors, which may be caused by the characteristics of the channel and the system.

There are three major categories of errors that are encountered:

1. Random errors. These are bit errors that are independent of one another; they are generally caused by the noise in the channel. They are simply isolated erroneous bits in a message or data, caused by thermal (voltage) noise in communication channels.

2. Burst errors. These are bit errors that occur sequentially in time and groups. Sometimes defects in the digital storage media cause these kinds of errors. They are difficult to correct by some codes, although block codes can handle this kind of errors effectively.

3. Impulse errors. These are large blocks of data that are full of errors; they are typically caused by lightning strikes or major system failures. Impulse errors generally cause catastrophic failures in a communication system; they are so severe that they are not even recognized or detected by forward error correction.

In general, all simple error correction codes are not sufficiently efficient to detect and correct burst and impulse errors, and they fail to reconstruct the message in the case of catastrophic errors. In the current state of advancements in this field, the Reed-Solomon codes were designed specifically to correct random and burst errors, and detect the presence of catastrophic errors by examining the message. If the number of errors per data is small, these errors can be totally corrected using the Reed-Solomon codes. These codes are therefore very useful in system design since they flag the unrecoverable message at the decoder. We will first discuss simple error detection and error correction codes, and slowly build up the analysis and description to finally reach the state-of-the-art aspects of modern coding theory.

3.1.1 Binary Linear Hamming Codes.

These codes were discovered by R. W. Hamming and M. J. G. Golay. In particular, the Hamming code refers to the (7,4) code introduced by Hamming in 1950 to provide a single error correction and double error-

detection (SEC-DED) code for errors introduced on a noisy communication channel. It was used to reduce the computer resonance and time that was wasted when the message was corrupted without the receiver realizing it and lending to the failure of communication. In general, all binary Hamming codes of a given length are equivalent. The dimension of a binary linear (n,k) code of length n = 2k −1 is n − k, where k is the number of data bits in the code, and its distance is d = 3, thus making it an exactly single-error correcting code.

3.2 Hamming (7,4) Code

This code has 4 data bits and 3 parity bits, hence the name. The parity bits are denoted by 2^r, r = 0,1,2, i.e., the bit numbers 1,2,4. Thus, using the exponent form, the three parity bits, denoted by p_1,p_2,p_4, are added to every four data bits of message, denoted by d_1,d_2,d_3,d_4, forming a codeword $c = \{p_1,p_2,d_1,p_4,d_2,d_3,d_4\}$ that is used to detect all single-bit and two bit errors and correct only a single-bit error. The algorithms for encoding and decoding are explained below. Although limited in its application, this code has been very effective in situations where excessive errors do not occur randomly in a transmission medium, that is, the Hamming distance between the transmitted and received words is at most 1, which can be corrected by this code.

3.2.1 Encoding and Decoding.

The encoding part of the algorithm is described in Table 3.2.1 (where y=yes and n=no).

Table 3.2.1 Encoding of Hamming (7,4) Code

Bit Location	1	2	3	4	5	6	7
Codeword c	c_1	c_2	c_3	c_4	c_5	c_6	c_7
Encoded Bit	p_1	p_2	d_1	p_4	d_2	d_3	d_4
p_1	y	n	y	n	y	n	y
p_2	n	y	y	n	n	y	y
p_4	n	n	n	y	y	y	y

The Venn diagram for Table 3.2.1, shown in Figure 3.2.1, is a geometrical representation of Table 3.2.1. It shows that the parity bit p_1 covers the data bits 1,3,5,7; the parity bit p_2 covers the data bits 2,3,6,7; and the parity bit p_4 covers the data bits 4,5,6,7. All these bits correspond to the entry 'y' in the above table. The Venn diagram is a visual means of establishing a relation between the parity bits and codeword. For smaller values of parity bits, it works fine, but as their number increases, the diagram becomes complicated and eventually becomes unintelligible even for $m = n - k > 4$.

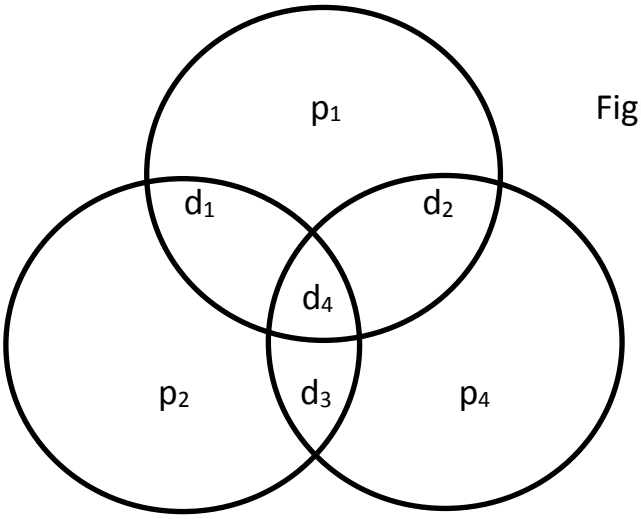


Figure 3.2.1 Venn Diagram for 4 data bits and 3 parity bits

Representing 'y' by 1 and 'n' by 0 in Table 4.2.1, the parity-check matrix H and the code-generator matrix G are defined, respectively, as

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

where H is a 3 × 7 matrix and G a 4 × 7 matrix. The entries in H represent the above table, or Figure 3.2.1, while the entries in G correspond to the following scheme: The first, second, and fourth rows in G represent the 'y' from p_1,p_2 and p_4, each under d_1,d_2,d_3,d_4, respectively, while the third, fifth, sixth, and seventh rows represent the identity matrix, since they are linearly independent, and inserted into G in that manner as part of the algorithm. Note that $HG^T = 0 \bmod 2$

3.2.2 Computation of Parity Bits.

The information from Table 3.2.1, or equivalently from the above Venn diagram, is used to compute the parity bits p_1,p_2,p_4, given the data bits d_1,d_2,d_3,d_4, by the formula

$p_1 = \neg (d_1 \oplus d_2 \oplus d_4)$,

$p_2 = \neg (d_1 \oplus d_3 \oplus d_4)$,

$p_4 = \neg (d_2 \oplus d_3 \oplus d_4)$.

Then the codeword is $c = \{p_1,p_2,d_1,p_4,d_2,d_3,d_4\}$. This is also one of the methods for encoding and decoding of Hamming (7,4) code, as shown in Example 3.2.2.

Example 3.2.1.

Given the data bits [1010], the three parity bits by formula (3.2.2) are

$p_1 = \neg (1 \oplus 0 \oplus 0) = 0$, $p_2 = \neg (1 \oplus 1 \oplus 0) = 1$, $p_3 = \neg (0 \oplus 1 \oplus 0)=0$. The codeword is c = [0110010].

Example 3.2.2.

Using the data from Example 3.2.1, the transmitted codeword is [0110010]. Suppose that the received word through a noisy channel is w = [0 1 0 0 0 1 0], where an error has occurred in the third bit (shown in boldface). The receiver uses formula (3.2.2) to check again the above parity bits $p_4\ p_2\ p_1 = 010$ as follows: $p_1 = \neg (0 \oplus 0 \oplus 0) = 1$, $p_2 = \neg (0 \oplus 1 \oplus 0) = 0$, $p_4 = \neg (0 \oplus 1 \oplus 0) = 0$. Note that two of these computed parity bits, $p_1\ p_2$, do not match with the original parity bits. Next, the bit in error is computed by the syndrome $010 \oplus 001 = (011)_2 = (3)_{10}$. Hence, the third bit is in error, which is corrected by flipping it or negating its value, and thus, the single-bit error is corrected.

Example 3.2.3

Construct (7,4) Hamming Code for the message [1000]. Conider even parity

Here 7 →indicates number of total bits in hamming code

Therefor it requires about 7-4=3 Parity bits.

During transmission it has 7 bit position

ie. Bit Position 1 2 3 4 5 6 7

Encoded Bit p_1 p_2 d_1 p_4 d_2 d_3 d_4

p_1 p_2 1 p_4 0 0 0

To define,

p_1 check → 1 , 3 , 5 , 7 = 100 = 1

p_2 check → 2, 3, 6, 7 =100 = 1

p_4 check → 4, 5, 6, 7 =000 = 0

Therefor the Ham ming Code = 1110000.

3.3(a)Syndrome and Error Detection

Let $v = (v_0, v_1, \ldots\ldots\ldots\ldots, v_{n-1})$ be a codeword from a binary (n,k) linear block code with generator matrix G and parity check matrix H.

Assume v is transmitted over a BSC, then binary received sequence.

$$r = (r_0, r_1, \ldots\ldots\ldots\ldots, r_{n-1}) = v + e$$

$$= (v_0, v_1, \ldots\ldots\ldots\ldots, v_{n-1}) + (e_0, e_1, \ldots\ldots\ldots\ldots, e_{n-1})$$

$$= (v_0 + e_0 + v_1 + e_1, \ldots\ldots\ldots\ldots\ldots\ldots, v_{n-1} + e_{n-1})$$

Where the binary vector $e = (e_0, e_1, \ldots\ldots\ldots\ldots\ldots\ldots, e_{n-1})$ is the error pattern.

The "1's" in e represent transmission errors ie,

$$r_i = \begin{cases} 1 & \text{if } r_i \neq v_i \\ 0 & \text{if } r = v_i \end{cases}$$

and e_i= 1 indicates that the i^{th} position in r has an error.

After receiving r the decoder must determines if r contains errors (error detection). And locate the errors in r (error correction).

Error detection is achived by computing the (n-k) tuple

$$s = (s_0, s_1, \ldots\ldots\ldots\ldots, s_{n-k-1}) = rH^T \quad (\text{Syndrome})$$

r is a codeword if and only if $s = rH^T = 0$

If $s \neq 0$, r is not a codeword and transmission errors have been detected.

If s = 0, r is a codeword and no errors are detected. If r is a codeword other than the actual transmitted codeword then an undetected error occurs. This happens whenever the error pattern e is a non–zero codeword.

The syndrome s computed from the received vector r actually depends only on the error pattern e and not on the transmitted code word v.

$$s = r \cdot H^T = (v + e) H^T = v \cdot H^T + e \cdot H^T \quad \text{Since } v \cdot H^T = 0$$

$$s = e \cdot H^T$$

Example 3.3.1

Consider a (7,4) linear code with parity – check matrix

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

Let r = (0 1 0 0 0 0 1) The syndrome of r is

$s = (s_0, s_1, s_2) = r. H^T$

$$= (0\ 1\ 0\ 0\ 0\ 0\ 1) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = (1\ 1\ 1) \neq 0$$

3.3(b) Syndrome and Error Correction

The syndrome s computed from the received vector r actually depends only on the error pattern e, and not on the transmitted code word v.

$$s = r.\ H^T = (\ v + e\ )H^T \ \text{(since } vH^T = 0)$$

For error pattern $e = (e_0, e_1, \ldots\ldots\ldots\ldots, e_{n-1})$ and H given by

$H = [I_{n-k} : p^T]$

$$= \left[\begin{array}{ccc|cccc} 1 & 0 & 0\ldots\ldots\ldots 0 & p_{0,0} & p_{1,0} & \ldots\ldots\ldots\ldots & p_{k-1,0} \\ 0 & 1 & 0\ldots\ldots\ldots 0 & p_{0,1} & p_{1,1} & \ldots\ldots\ldots\ldots & p_{k-1,1} \\ . & . & \ldots\ldots\ldots\ldots & & & & \\ . & . & \ldots\ldots\ldots\ldots & & & & \\ 0 & 0 & 0\ \ldots\ldots\ldots\ 1 & p_{0,n-k-1} & p_{1,n-k-1} & \ldots\ldots\ldots\ldots & p_{k-1,n-k-1} \end{array}\right]$$

The syndrome equation can be written as

$$s_j = e_j + e_{n-k}p_{0j} + e_{n-k+1}p_{1j} + \ldots\ldots\ldots\ldots\ldots\ldots + e_{n-1}p_{k-1j} \quad 0 \le j \le n-k$$

This is a set of n-k equations in n unknowns $e_0, e_1, \ldots\ldots, e_{n-1}$.

The decoder must solve of these equations for the estimated error pattern e

Estimated codeword is

$$v = r + e$$

There are 2^k possible solutions to the syndrome equations and only one solutions represents the true error pattern.

To minimize the probability of a decoding error. The most possible error pattern that satisfies the above equation is chosen as the true error vector

Recall for BSC the maximum likelihood decoder choose v as the codeword v that minimizes Hamming weight of the error pattern e.

Example 3.3.2

Let

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

Suppose v =(1 0 0 1 0 1 1) is transmitted and r = (1 0 0 1 0 0 1) is received . To Find the syndrome

SL No:	Bit in Error	Bit in error vetor (e) non zero bit show error	Syndrome Vector
1	1st	1 0 0 0 0 0 0	1 0 0
2	2nd	0 1 0 0 0 0 0	0 1 0
3	3rd	0 0 1 0 0 0 0	0 0 1

4	4th	0 0 0 1 0 0 0	1 1 0
5	5th	0 0 0 0 1 0 0	0 1 1
6	6th	0 0 0 0 0 1 0	1 1 1
7	7th	0 0 0 0 0 0 1	1 0 1

$s = e\,H^T$

$$s = (1\ 0\ 0\ 0\ 0\ 0\ 0) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

= [(1+0+0+0+0+0+0) (0+0+0+0+0+0+0) (0+0+0+0+0+0+0)] = 1 0 0 [Using XOR]

Repeat the same procedure using the other 6 error bit and thus we get the above syndrome error.

The syndrome of r $\rightarrow$ $s = v\,H^T$

$$s = (1\ 0\ 0\ 1\ 0\ 0\ 1) \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

= (1 1 1) $\neq$ 0 which implies there is an error

At syndrome vector (1 1 1) the non zero bit show error as (0 0 0 0 0 1 0)

v = e + r

= (0 0 0 0 0 1 0) + (1 0 0 1 0 0 1)

= (1 0 0 1 0 1 1) that gives the original code v

Chapter – 4

LDPC Codes

4.1 Introduction

Low-density parity-check (LDPC) codes were invented in the 1960s by Gallager [1962]. They were forgotten for 30 years until they were rediscovered by MacKay and Neal [1996], and have now become a major area of research and application. These codes are also known as Gallager codes. They are decoded iteratively and have become successful in recovering the original codewords transmitted over noisy communication channels, now a major area of research and applications.

Any linear block code that can be defined by its parity-check matrix .If this matrix is sparse i.e. it contains only a small number of 1's per row or column then the code is called a low density parity check code.

4.2 Representations for LDPC codes

Basically there are two different possibilities to represent LDPC codes. Like all linear block codes they can be described via matrices. The second possibility is a graphical representation.

4.2.1 Matrix Representation

Lets look at an example for a low-density parity-check matrix first. The matrix defined in equation (1) is a parity check matrix with dimension n×m for a (8,4) code.

We can now define two numbers describing these matrix. w_r for the number of 1's in each row and w_c for the columns. For a matrix to be called low-density the two conditions $w_c << n$ and $w_r << m$ must be satisfied. In order to do this, the parity check matrix should usually be very large, so the example matrix can't be really called low-density

$$H = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

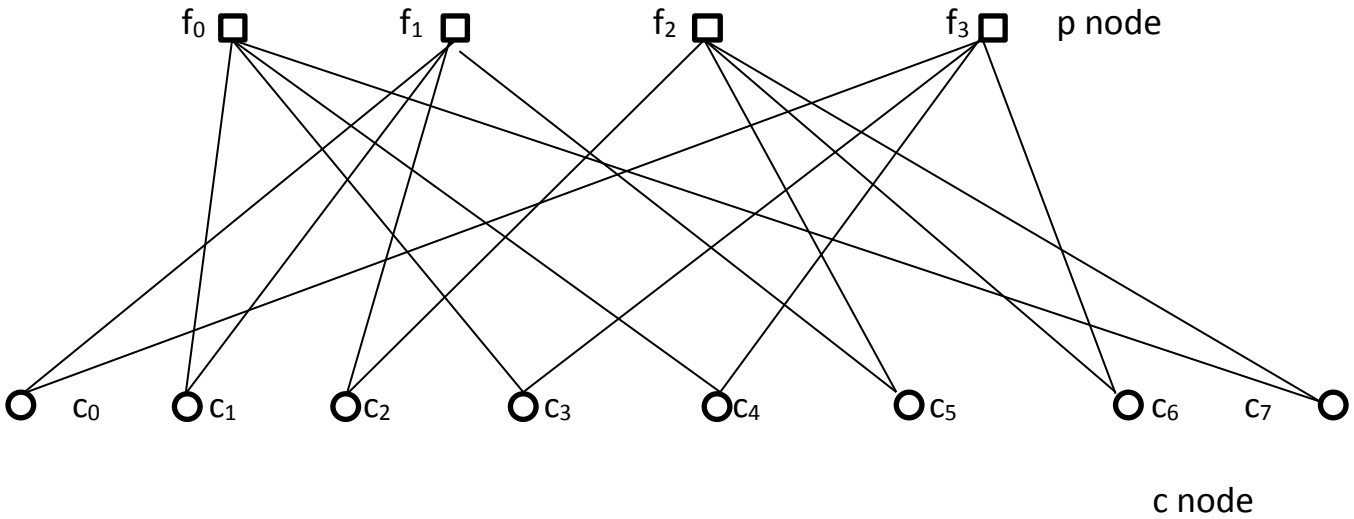


Figure 4.2.1: Tanner graph corresponding to the parity check matrix in equation (1). The marked path $c_2 \rightarrow f_1 \rightarrow c_5 \rightarrow f_2 \rightarrow c_2$ is an example for a short cycle. Those should usually be avoided since they are bad for decoding performance.

4.2.2 Graphical Representation

Tanner introduced an effective graphical representation for LDPC codes. Not

only provide these graphs a complete representation of the code, they also help to describe the decoding algorithm as explained later on in this tutorial.

Tanner graphs are bipartite graphs. That means that the nodes of the graph are separated into two distinctive sets and edges are only connecting nodes of two different types. The two types of nodes in a Tanner graph are called variable nodes (p-nodes) and check nodes (c-nodes)

Figure 4.1.1 is an example for such a Tanner graph and represents the same code as the matrix in 1. The creation of such a graph is rather straight forward. It consists of m check nodes (the number of parity bits) and n variable nodes (the number of bits in a codeword). Check node f_i is connected to variable node c_j if the element h_{ij} of H is a 1.

4.3 Tanner Graphs

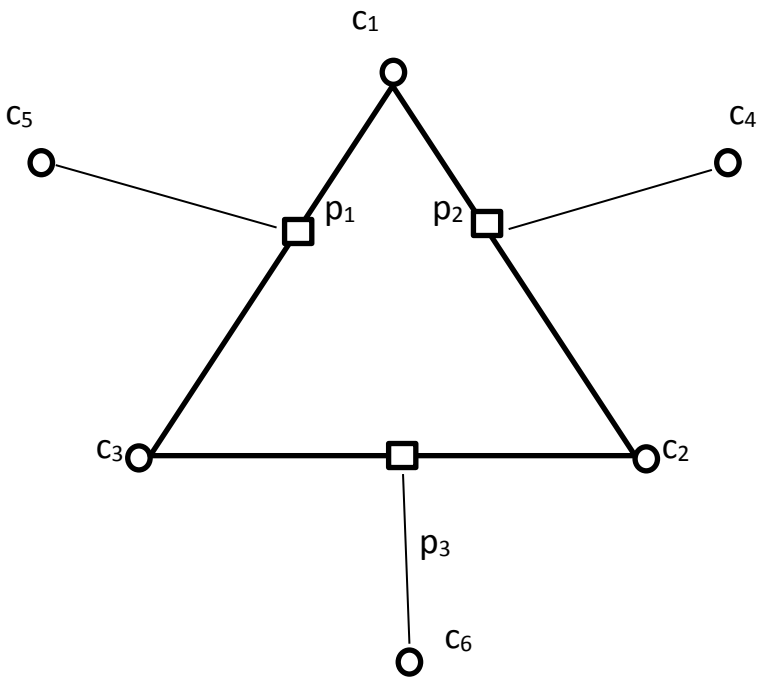
A Tanner graph is a pictorial representation for the parity-check constraints . In Tanner graphs each square represents a paritycheck bit and each circle connected to a square represents a bit that participates in that parity check. Thus, the nodes of the graph are separated into two distinct sets: c-nodes (bottom nodes, circles) and p-nodes (top nodes, squares). For an (n,k) code, the c-nodes c_i, i = 1,... ,n, and the p-nodes p_j, j = 1,... ,m, m = n − k, represent the message bits (symbols) and the parity bits, respectively, for a transmitted word c_i (or received word denoted by w_i).

(4.3.1) A Tanner graph represents a linear code C if there exists a parity-check matrix H for C associated with the Tanner graph. All variable c-nodes connected to a particular p-node must sum, mod 2, to zero, which is same as

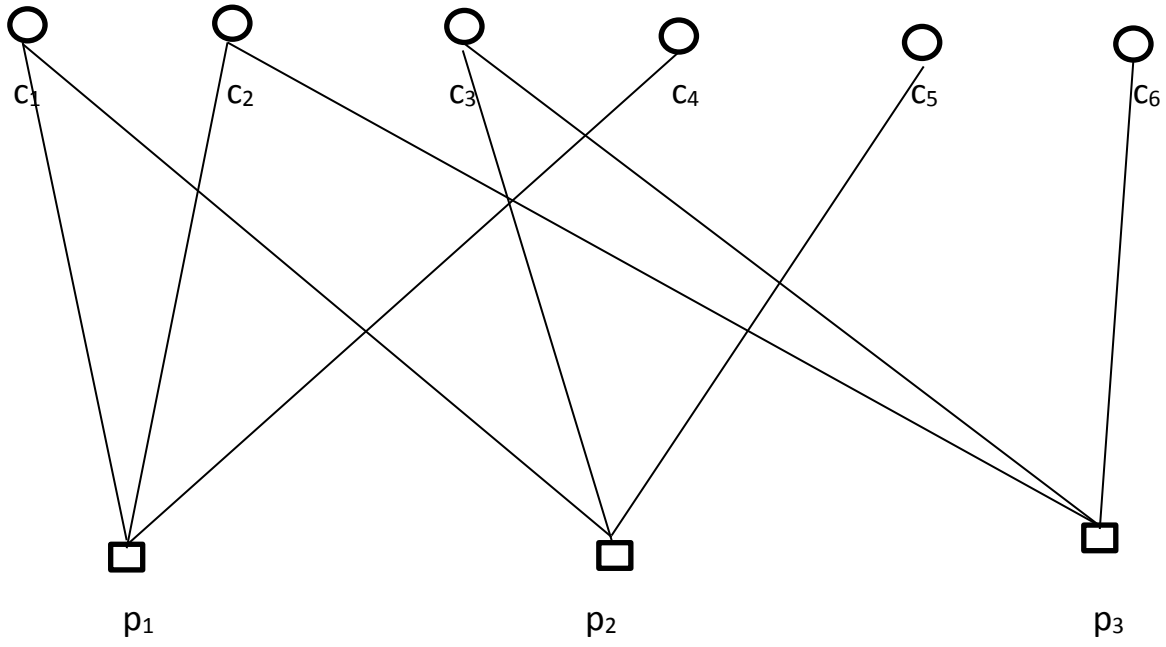
XOR-ing them to zero. This provides the constraints. Tanner graphs are bipartite graphs, which means that nodes of the same type cannot be connected, i.e., a c-node cannot be connected to another c-node; the same rule applies to the p-nodes. For example, in Figure 4.3.1(a) or (b) the c-nodes for a (6,3) code are denoted by $c_1,\ldots,c_6$, and the parity-check bits denoted by p_1,p_2,p_3 are the p-nodes. Tanner graphs can also be presented in the vertical form in which the c-nodes are the left nodes and p-nodes the right nodes, and c_i and p_j start at the top and move downward (see §17.5). Sometimes a general notation is used in which the c-nodes and p-nodes are denoted as x-nodes x_i and y-nodes y_j, respectively.

The constraints for the example in Figure 4.3.1 are presented in two ways, both implying the same constraints equations, but the representation (b) is generally easier in the case when the number of c-nodes and p-nodes is large.

In this example the first parity-check bit p_1 forces the sum of the bits c_1,c_2 and c_4 to be even, the second parity-check bit p_2 forces the sum of the code bits c_1,c_3 and c_5 to be even, and the third parity-check bit p_3 forces the sum of the code bits c_2,c_3 and c_6 to be even. Since the only even binary number is 0, these constraints can be written as $c_1 \oplus c_2 \oplus c_4 = 0$, $c_1 \oplus c_3 \oplus c_5 = 0$, $c_2 \oplus c_3 \oplus c_6 = 0$. The only 8 codewords that satisfy these three parity-check constraints are {000000, 001011, 010101, 011110, 100110, 101101, 110011, 111000}. In this code (Figure 4.3.1) the first three bits are the data bits and the last three bits are then uniquely determined from the constraints; for example, if the data bits are 010, then the codeword to be transmitted is 010101 as determined from the above list of 8 codewords. The parity-check matrix H so obtained must be transformed first into the form $[-P^T \mid I_{n-k}]$; then the generator matrix G is obtained by transforming it into the form $[I_k \mid P]$.



(a)



(b)

Figure 4.1.1 Tanner graph for the (6,3) code.

The graph in Figure 4.3.1 is said to be regular since there are the same number of constraints at each parity-check bit, as seen by the same number of lines connecting each square (p-node) to circles (c-nodes). If this number is not the same at each p-node, then the graph is said to be irregular, The graph of Figure 4.3.1 (a) or (b) is equivalent to the parity-check matrix

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

4.4Decoding LDPC codes

The algorithm used to decode LDPC codes was discovered independently several times and as a matter of fact comes under different names. The most common ones are the belief propagation algorithm , the message passing algorithm and the sum-product algorithm. In order to explain this algorithm, a very simple variant which works with hard decision, will be introduced first. Later on the algorithm will be extended to work with soft decision which generally leads to better decoding results. Only binary symmetric channels will be considered.

4.4.1 Hard-Decision Decoding.

.The decoding scheme runs through the following steps:

Step 1. All c-nodes c_i send a message to their p-nodes p_j that a c-node c_i has only the information that the corresponding received i-th bit of the codeword is w_i; that is, for example, node c_1 sends a message containing w_1 (which is 1)

to p_2 and p_4, node c_2 sends a message containing w_2 (which is 1) to p_1 and p_2, and so on.

Step 2. Every parity node p_j computes a response to every connected c-node. The response message contains the bit that p_j believes to be correct for the connected c-node ci assuming that the other c-nodes connected to p_j are correct. For our example, since every p-node p_j is connected to 4 c-nodes, a p-node p_j looks at the received message from 3 c-nodes and computes the bit that the fourth c-node must be such that all the parity-check constraints are satisfied.

Step 3. The c-nodes receive the message from the p-nodes and use this additional information to decide if the original message received is correct. A simple way to decide this is the majority vote. For our example it means that each c-node has three sources of information concerning its bit: the original bit received and two suggestions from the p-nodes.

Step 4. Go to Step 2.

Bibliography

- Algebraic and Stochastic Coding Theory, Dave. K . Kythe, Prem .K. Kythe, CRC Press Taylor & Francis Group.
- Coding Theory The Essential, D.G. Hoffman, D.A. Linder, K.T. Phelps, C.A. Rodger, J.R. Wall ,Auburn University, Auburn Alabama.
- Algebraic Coding Theory –Michael Toymill ,The University Of Puget Sound, pdf
- I.N. Herstein, Topic in Algebra, Wiley Eastern Ltd, New Delhi, 1975.
- Hungerford, Algebra, Springer
- M. Artin, Algebra, Prentice- Hall of India, 1991
- N.Jacobson, Basic Algebra Vol. I, Hindustan Publishing Corporation
- P.B. Bhattacharya, S.K. Jain, S.R. Nagapaul, Basic Abstract Algebra 2nd edition, Cambridge University Press, Indian Edition, 1997.
- World Wide Web.