Image Encoding with Error Correction Ability Using Hamming Code

Ng Kyle - 13520040[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520040@std.stei.itb.ac.id

***Abstract*— Data are used in many forms, from simple text to large files. These data are stored through the encoding of bits of 0's and 1's which are used to transfer data between two subjects and used as a way to store information as data on storage. These data in its storage and transfers may have some information corrupted through many factors. These corruption(s) on data will lead to wrong information retrieved. Such it is needed for a good data encoding or better storage/transfer mechanisms. A good encoding will ensure more reliable data stored/transferred, one of which is through Hamming Code. This paper will discuss ways we can encode images through Hamming Code such an image encoded will be able to self-correct itself up to a certain rate of errors to be corrected. The resulting encoded data will lead to more reliable data retrieved and preserve the information of image encoded.**

***Keywords*— Boolean, Data, Encoding, Error Correction, Hamming Code, Image.**

I. INTRODUCTION

Information is of foremost importance in current society, such misinformation may lead to critical problems. These pieces of information are stored and transferred as a series of 0's and 1's or in a general term as binary. A flip between 0 and 1 in this series of data may lead to significantly different data retrieved. Such there needs to be a form of the mechanism of how data transferred/stored needs to be stored/packed which will reduce the risk of a wrong data retrieved. The way data arrangement are stored in computer science terms is called data encoding.

Data corruption may happen in different ways, either in its transfer stages or storing stages. To resolve these, there are multiple solutions, from sending multiple copies of the same information (also called repetition) to better data transfer mechanisms. Such in 1950, Richard W. Hamming comes up with a solution which is a self-correcting code which was eventually named Hamming Code (named after Hamming). Hamming Codes are a family of linear error-correcting codes, which are codes that can self-correct errors in themselves.

Hamming Codes are classified as perfect codes which means Hamming Codes fall into a category that has the best possible rate of data stored in a block of information transferred /stored to the number redundant pieces of information transferred. Yet, Hamming Codes comes with its limitations wherein a block of Hamming Code, only ables to correct one error. This limitation may be improved by dividing the data into smaller blocks which leads to a worse rate of data stored. It can also be improved to be able to detect two-bit errors (without correcting them), which is called as Extended Hamming Code.

This paper focuses on how to implement Hamming Codes to encode images which in itself is a series of binaries. Hamming Codes implemented will be varied and compared with different Hamming Code schemes. Reducing the possibility of wrong information retrieved which in result image retrieved will have identical information wise. By which, the encoded information will be able to handle errors during image transfer as efficient and effective as possible.

II. THEORITICAL BASIS

A. Binary Code

Binary code is a sequence of data encoded in binary which is a series of 0's and 1's. Binary code is used in computers that represent data, which may be interpreted as texts, numbers, or other types of data. Binary codes are interpreted and processed by computers which the data can be used, stored, and transferred. These strings of 0's and 1's will be processed computer which also uses boolean algebra in its lowest level of computation (hardware level).

Binary codes are usually divided into parts, where a single 0 or a 1 in the string is called a bit and 8 bits form 1 byte. The form of binary code can be interpreted as states of on-offs. This is most valuable for computer hardware to computations on which uses transistors which turned on and off with certain voltage (5 Volts and 0 Volts).

B. Boolean Algebra

1. Definition

Let B a set defined with two binary operators, + and -, and a unary operator, '. 0 and 1 are two different elements in set B, it follows that for every $a, b, c \in B$ [1]:

1. Identity
   i. $a + 0 = a$
   ii. $a \,.\, 1 = a$
2. Commutative
   i. $a + b = b + a$
   ii. $a \,.\, b = b \,.\, a$
3. Distributive
   i. $a \,.\, (b + c) = (a \,.\, b) + (a \,.\, c)$

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

4. Complement
   For every $a \in B$, there exist a unique element $a' \in B$ such that :
   i. $a + a' = 1$
   ii. $a.a' = 0$

2. Operations

From the set of three base operators (+, . , ‘), we can derive other boolean operations and functions. In this scope of the topic, we will derive a particular binary operator called the XOR operator. XOR operator, we define as ‘$\oplus$ ‘ operator. For, it holds that if and only if.In all, for all possible combinations of a and b, we define :

Table i. Boolean Binary Operators

a	b	a + b	a . b	a $\oplus$ b
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

The derivation of $\oplus$ operator from . and + :

$$a \oplus b = (a + b).(a' + b') \quad (1)$$

C. Image

Images can be stored as vector (vector graphics) or as a bit-map (raster/bitmap graphics) [2]. The most basic is bitmap images which are stored as an array of pixels. Each pixel can consist of RGB values or Greyscale images. In RGB, each pixel consists of three values, which usually of 256 levels of each color. Such a pixel is an array of three values ranging from 0-255 for each Red, Green, and Blue channel (also called true color). While, in greyscale images, each pixel will only hold a value from 0-255. Thus a value can be represented as an 8-bit value (1 byte).

An image is stored as a series of bits, which takes into account the number of pixels, the colour scale, and the level of the image. Such an image with pixels and a level of 256 for each channel :

Table ii. Image Memory Usage

RGB (24 bit) image	$w \times h \times 8 \times 3$
Greyscale image	$w \times h \times 8$

These values are calculated without taking into account compressions, such as the data of the image being stored as a raw bitmap.

D. Linear Codes

A linear codes (or called linear error correcting code) are codes which has a structure with linear transformation (matrix and vector transformations) and ables to detect and correct error in itself through said transformations. Error detection differs from error correction. Error detection implies that the code able to detect error in itself, but it doesn’t have the ability to correct said errors. Error correction implies that a code able to detect an error in it and correct said error in itself. Linear codes have their limitations that are its error detection and error correction ability. These limitations can be found from its Hamming Distance (d_H), which is the difference between two codewords (string of bits) of the same length in the same index/position. Minimum Hamming distance can be calculated as :

$$d_{min} = \min_{c_i \neq c_j} d_H(c_i, c_j)$$

For a codeword of length of three we can visualize Hamming Distance as points of a cube, where each point represent it’s coordinate (codeword).

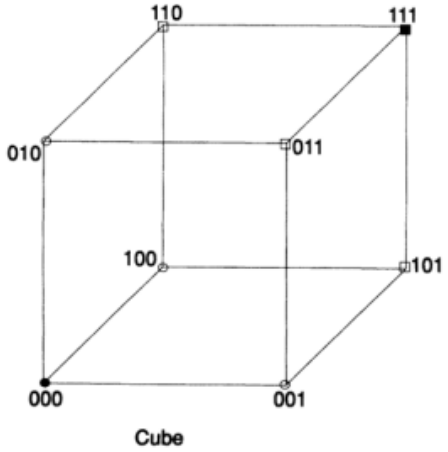

Figure 1. Cube Representing Codeword
Hamming – The Art of Doing Science and Engineering

Minimum Hamming distance will determine the limitations of linear code.

Table iii. Hamming Distance and Limitation

Hamming Distance	Limitation
1	Unique Decoding
2	Single Error Detection
3	Single Error Correction
4	1 Error Correction and 2 Error Detection
5	Double Error Correction
$2k + 1$	k Error Correction
$2k + 2$	k Error Correction and $k + 1$ Error Detection

D. Hamming Codes

Hamming Codes is a binary linear code with a hamming distance of three which implies it has the ability to correct a single error in its codeword. It has a property of Hamming Distance of 3 which will be shown. Hamming codes can be defined and interpreted in 2 ways, which are in its pure algorithm of binary operations and through linear transformations, hence Hamming Code is a linear code.

Hamming code as its pure algorithm form can be seen as for every message we will encode, we will create a codeword which includes parity/check bits in it. Hamming code with a codeword of length n bits and message k bits will be defined as Hamming Code of Hamming). Such that we have bits as parity/check bits in the codeword which will be bits in the position of power of two. The value of n for standard Hamming Code valid for $n =$

$2^m - 1$, for integer $m \geq 2$. Or formally n is defined :

$$n = \{ 2^m - 1 \mid m \in N, m \geq 2 \} \quad (2)$$

This means, a codeword for a standard Hamming Code will have the length of one less power of two (a Hamming Code with n other than defined above called as a General Haming Code). This comes from the logic where we will view every bit in codeword as its binary representation of said position. Trivial Hamming Code is represented as in Fig.1, where the valid codeword is 000 and 111, while other combination string of bits 0 and 1 will means it has an error of 1 bit differs from one of the valid codes and can be corrected two one of valid codes. All Hamming Code has a hamming distance of 3, for a codeword of length 3, we can visualize it in 3-Dimension cube form as in figure 1. For a codeword of length n, can be represented as n-Dimension cube, where every bit in the code represents coordinate in said dimension. Some of the first values of n (codeword length), k (number of message bits), m (number of parity bits) and its corresponding Hamming Code category :

Table iv. Hamming Code n values

Hamming Code (n,k)	n	k	m
Hamming(3,1)	3	1	2
Hamming(7,4)	7	4	3
Hamming(15, 11)	15	11	4
Hamming(31, 26)	31	26	5

As we can see, the more message bits we encode for a block of Hamming Code, the less space of codeword used as check/parity bits. The efficiency of a the bits transmitted can be calculated as

$$Efficiency = \frac{k}{n} = \frac{2^m - 1 - m}{2^m - 1} = 1 - \frac{m}{2^m - 1} \quad (3)$$

We can see for larger value of m (larger block of codeword), results to more efficient encoding as Efficiency approaches 1.

As example, for Hamming(15,11) we can encode a message bits of 01011000111 as codeword $TWX0Y101Z1000111$, where $WXYZ$ parity bits with corresponding values of 1110 and T can be 0/1 which is redundant (not used as message bit nor parity bit).

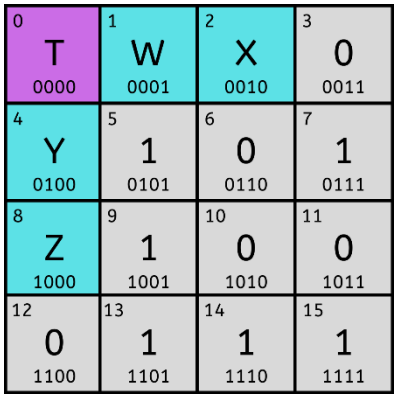

Figure 2(a). Position of Message bits in Hamming(15,11)

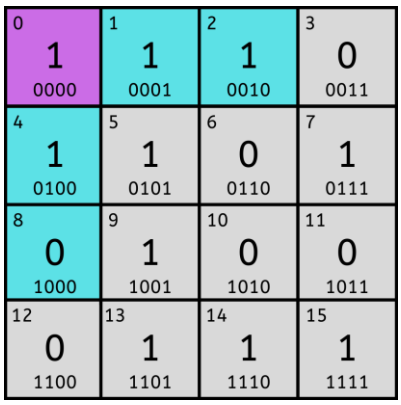

Figure 2(b). Complete Hamming(15,11)

Every parity bits will check for all bits with position in binary corresponding to its bit 1 position and will the value of the parity bit will be set to 0/1 such that the number (sums) of all bits value 1 with those position will be even. Here are the list of parity bits position and its corresponding bits to check in Hamming(15,11).

Table v. Parity bit positions Hamming(15,11)

Parity Bit Position	Binary Positions	Bits positions (decimal)
0001	_ _ _ 1	1, 3, 5, 7, 9, 11, 13, 15
0010	_ _ 1 _	2, 3, 6, 7, 10, 11, 14, 15
0100	_ 1 _ _	4, 5, 6, 7, 12, 13, 14, 15
1000	1 _ _ _	8, 9, 10, 11, 12, 13, 14, 15

Parity bit positions always placed in positions with bit representation of exactly one bit 1 (position of power of 2 in decimal). Each parity bit will record the number of 1s in bits it checks in forms of rows or columns. Thus this property of parity checks in a way sets up a binary search for any error in any positions through checking the number of 1s in the parity blocks and the status of the parity bit in the block. Another property is, for every parity block, there will only exist 1 parity bit position. This holds true for every Hamming Code since every parity block is defined by the position of 1s.

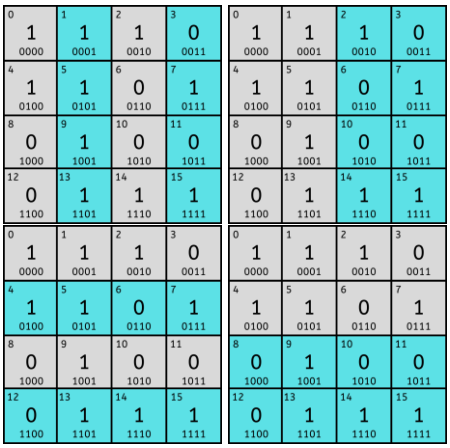

Figure 3. Parity check blocks in Hamming(15,11)

From those properties, to check a codeword, we can do xor to all binary representation positions with value of 1. The binary representations result of those will results to 0 if the codeword

is valid (no flipped bits) or will results the position of the flipped bit for a bit error. For more than 1 error, the resulting xor operations will results to a single position which may or may not be one of the flipped bits, means we can't fix nor determine more than 1 error, such the nature of a linear code with Hamming Distance of 3.

In Matrix representation, for a general hamming code, we need two matrices, which are Generator Matrix and Parity Matrix. We define Generator Matrix as

$$G = [I_{k \times k} \mid S] \quad (4)$$

where I is an Identity matrix and S is matrix with k rows, where every rows represents all possible m-bit string with more than one 1 bits (at least 2 bits of 1), which in term called strings with weight (the number of 1 bits) at least 2.

For every Generator Matrix G we have a parity matrix H which defined as

$$H = [S^T \mid I_{k \times k}] \quad (5)$$

Such for a message string $\vec{m}$, we encode to a codeword $\vec{c}$ with a form of Hamming Code encoding by the formula:

$$\vec{c} = \vec{m}G \quad (6)$$

and to check codeword $\vec{c}$ to a resulting vector $\vec{s}$ with formula:

$$\vec{s} = H\vec{c}^{\,T} \quad (7)$$

where $\vec{s}$ called syndrome vector. For a no error codeword, will results to $\vec{0}$ vector of $\vec{s}$, and for other value of $\vec{s}$ will give a given column vector in H, which the corresponding column index gives the flipped bit index in $\vec{c}$ for a 1 bit error. More than 1 error will give a combination of two column vector in H, which won't be detected by Hamming Code encoding. Linear transformation defined with boolean operations XOR and AND. We may check, for every column vector in H can only be achieved from combination of 3 other vector columns in H, thus Minimum Hamming Distance of a Hamming Code is three. Note : In this paper, we will refer $\vec{m}$ as our message bits vector and m as the number of parity bits.

III. Hamming Code Implementation on Image

A. Hamming Codes Implementation Non-Matrix

Standard Hamming Codes in implementation needs us to divide our data, such that stream of bits will be partitioned to chunks of message bit vectors. The size of block determined by the Hamming Code used. For Hamming(n, k), the data will be partitioned to blocks of k bits. Thus, we need to consider the number of bits on said data stream to be transmitted such that we can encode all said data to a same encoding of Hamming Code. We want for a bit stream data of length L to be divided to blocks of bits of size k, such that $L \bmod k = 0$. For the two ways of implementation of Hamming Code, we can analyze by it's complexity.

Hamming code for Hamming(n, k) without matrix requires us to divide data to message string of length k, after which we will pad the to corresponding size $n + 1$. Thus giving the length of codeword to be length of power of two. After which, we assign the 0^{th} position to as 0/1 (redundant bit) and all position of power of two as parity bit. After which, we assign the values of each parity bit corresponding to each parity block (1 for odd number of 1s or 0 for even number of 1s), thus for every parity block there are even number of 1s. To check a codeword, we will xor all the positions with value 1, with resulting bit representation as the flipped bit (error) for a single error codeword. We can generalize for all meesage of length arbitrary k since we won't be restricted to the columns nor the rows of said vector. In all, we can write out the algorithm for encoding Standard Hamming Code as:

1. Determine the Hamming(n,k).
2. Set 0/1 for the 0^{th} position bit (optional).
3. Set bits in message bit vector to non power of two positions in codeword consecutively.
4. Set every parity bit according to number 1s in each parity block as:
   - 1 for odd number of 1s in parity block
   - 0 for even number of 1s in parity block

After which, the codeword is said a well-prepared block which is set for transfer.

Algorithm to check a codeword and correcting the error bit, which will be done when receiving a codeword, can be listed as:

1. Set an initial value of 0.
2. Enumerate codeword from 1 (or 0) to $2^m - 1$ as the position of said bits.
3. Iterate all bits in codeword, for all bits in codeword, if the value is 1, then xor the position to current value.
4. Resulting value indicates the position of flipped bit in the codeword.
5. Complement the value in said position from step 4.
6. Extract all values not in position of power of two.

Here is a code in python on Encoding Standard Hamming Code shown below:

```
def encodeHamming1(msg,n,k) :
    m = n - k
    encoded = []
    r = 0 #rth redundant bit

    #set up codeword loop
    for i in range(1, n+1):
        if i == 2 ** r :
            encoded.append(0)
            r += 1
        else :
            encoded.append(int(msg[i-r-1]))

    #set paritybit values
    rval = 0
    for i in range(m) :
        rval = 2 ** i
        count1 = 0
        for j in range(1,n+1) :
            if ((j & (1 << i)) & rval == rval and encoded[j-1] ==1):
```

```
            count1 += 1
        if (count1 % 2 == 1) :
            encoded[rval-1] = 1
    return encoded
```

This code has a time complexity of $O(n \times m)$, where n represents the length of codeword and m represents the number of parity bits in the codeword. As seen in tge code, we won't need to add 0^{th} bit since it won't effect our calculations as we will see in code for error correcting Hamming Code in python shown below:

```
def Hamming1(encoded):
    error = 0
    idx = 1
    for i in encoded :
        if (i==1) :
            error = error ^ idx
        idx += 1
    if (error > 0 and error < idx) :
        encoded[error-1] = int(not encoded[error-1])
    r = 0
    bits = []
    for i in range(1,len(encoded)+1):
        if (i == 2 ** r) :
            r += 1
        else :
            bits.append(encoded[i-1])
    return bits
```

Time complexity to check and correct received encoded Hamming Code is $O(n)$, where n represents the length of list encoded. Which means, it will take rather longer to encode a chunk of bit vector than to receive and correct error of a codeword in Hamming Code.

This algorithm is the most intuitive, since it acts as how Hamming Code in essence. It xor all bit representation of positions with value 1. Since we have prepared the codeword such for every parity block, we will always have even number of 1s, which means when we xor the bits, we will eventually ends up to 0 value, since every 1 bit has a pair and counted once (since every parity check are unique). But, this algorithm has its own cons. As we find, we won't need to track 0^{th} position value, as it is redundant and doesn't contain our message bit information, where as when we xor 0^{th} position (which in binary representation will be m number of 0s), it will not effect our calculation. Thus, as mentioned before, 0^{th} position can be completely discarded or set as abritary value of 0/1.

Con of above algorithm are it doesn't have the capability to encode message with length that doesn't follow definition in (2). We can expand our Hamming Code out of its standard forms, which in term we need to find the least parity bits we need to encode the message bit, yet it is sufficient to parity check all bits in the message. Formally we need to find m known length of string k :

$$m \mid k + m + 1 \leq 2^m < 2^{m+1} - 1 \qquad (8)$$

Yet, this will results to a higher complexity of encodin since we will need to find such m that follows (y). Snippet of expanded algorithm for encoding Hamming Code which is put in the beginning of encoding (replacing line 1 in function encodeHamming1) shown below :

```
def encodeHamming2(msg) :
    k = len(msg)
    m = 1
    while (2**m < k + m + 1):
        m += 1
    n = k + m
    …
```

For a uniform and repeated use of the same Hamming(n,k), it is most beneficial to determine $n, k,$ and m values beforehand rather than adding another redundant loop as above.

B. Hamming Codes Implementation Matrix Form

In formal matrix representation of Haming Code, we need to generate matrix G and H to encode and check message and codeword by applying linear transformation to message $\vec{m}$ and codeword c. For a Hamming Code with form Hamming(n, k), we can create a Parity Matrix H such that:

1. Parity Matrix H will be formed from every possible combination of 0s and 1s (excluding binary representation for 0).
2. Matrix H will have m rows and $2^m - 1$ rows.
3. Every column vector is a binary representation for every number 1 through $2^m - 1$.

As example, for Hamming(7,4), we will have parity matrix H with the form

$$H(7,4) = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Where column 1 through 7 are column vector for its coressponding column. This matrix representation is analog with its representation as in Fig. 2 for $H(15,11)$ which has a parity matrix in the form

$$H(15,11) = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

In general, we want H to be in systematic form such that in the form as in (z) by moving all columns with one 1s to the right of the matrix. Such for $H(7,4)$ in its systematic form will be

$$H(7,4) = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

To form our generating matrix G, we transpose S in H and concatenate Identity matrix $(I_{k \times k})$ to the left of said S^T.

Generating matrix for Hamming(7,4) is

$$G(7,4) = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Such we can encode a message $\vec{m}$ by (6) and check encoded codeword by (7). With the form of systematic G and H, for Hamming(n, k) and a message with length k, we will get codeword length $n = k + m$ such that first k bits are the message bits and consecutive m bits as parity bits. For Hamming(7,4) we have message bit of the form $abcd$ and codeword $abcdxyz$ where as xyz are the parity bits.

In parity checking a codeword with (7) we get s such that s our corresponding bit location where the error is. In formal form for $\vec{e}$ as our error vector which is xored to our codeword $\vec{c}$

$$\vec{s} = H(\vec{c} \oplus \vec{e})^{\mathrm{T}}$$
$$\vec{s} = H\vec{c}^{\,T} \oplus H\vec{e}^{\,T} \quad (10)$$

by definition, $H\vec{c}^{\,T} = \vec{0}$, such

$$\vec{s} = H\vec{e}^{\,T} \quad (11)$$

where $\vec{s}$ as our syndrome vector which is a column vector in H which we can xor the corresponding bit with position value of $\vec{s}$ in H for bit in $\vec{c}$. To note, matrix multiplication here defined with boolean operations (XOR and AND). In result, our columns of S^T in G acts as our parity bits equation such Hamming(n, k) we have $n - k = m$ equations to set our parity bits. The identity matrix preserves our message bit to be encoded in our resulting codeword. The equation for our setting parity bits by matrix G and encoding our codeword $\vec{c}$ for message $\vec{m}$ defined as

$$\vec{c}_i = (\vec{m}_1 \,.\, G_{1i}) \oplus (\vec{m}_2 \,.\, G_{2i}) \oplus \ldots \oplus (\vec{m}_j \,.\, G_{ji}) \quad (12)$$

and for every row in parity matrix H represents equation for every m parity check bits in our syndrome $\vec{s}$.

To generate both matrices H and G, we can simultaneously use a function which takes n and k and will return to matrices H and G, in python code shown below:

```
def Generators (n, k):
    m = n-k
    G = [[0 for j in range(n)] for i in range(k)]
    H = [[0 for j in range(n)] for i in range(m)]
    for i in range (k):
        G[i][i] = 1
    for i in range (m):
        H[i][k+i] = 1
    r = 0
    for i in range (1,n+1):
        if i == 2**r :
            r += 1
        else :
            bin = format(i,'0'+str(m)+ 'b')
            for j in range(1,m+1):
                G[i-r-1][k+j-1] = int(bin[-j])
                H[j-1][i-r-1] = int(bin[-j])
    return G, H
```

After which we can encode our message as Hamming Code encoding make use given G by doing boolean matrix mutliplication with operations XOR ($\oplus$) replacing addition and AND (.) replacing multiplication. Such for given matrux H and G we can encode and decode our codeword. Here are the codes to encode and check codeword using H and G matrix.

```
def encodeHammingM(msg,G):
    k = len(G)
    n = len(G[0])
    codeword = [0 for i in range(n)]

    #Matrix Xor Multiplication
    for i in range(n):
        for j in range(k):
            codeword[i] = codeword[i] ^ (int(msg[j]) & G[j][i])
    return codeword

def decodeHammingM(codeword, H):
    n = len(codeword)
    m = len(H)
    syndrome = [0 for i in range(m)]

    #Matrix Xor multiplication
    for i in range(m):
        for j in range(n):
            syndrome[i] = syndrome[i] ^ (H[i][j] &
int(codeword[j]) )

    #Check any syndrome columns (error correction)
    for column in range(n):
        allsame = True
        for row in range(m):
            if (H[row][column] != syndrome[row]) :
                allsame = False
        if allsame :
            codeword[column] = int(not codeword[column])
            break
    return codeword
```

Our encoding and decoding by matrix representation has overall time complexity of $O(n \times k)$ and $O(m \times n)$, which compared to non-matrix approach it seems that matrix representation is worse in implementation. Another con for matrix representation is that it requires us to compute G and H matrix such we can perform our linear transformations. Matrix representation of Hamming Code can be expanded to its general form through finding the number of parity bits m and create matrices G and M for length of $n = k + m$ accordingly.

C. Hamming Codes Implementation on Image

Image is a matrix of values, where each element in matrix we call pixel which values represents the colour of said pixel. As we discussed before, there are multiple types of image colours. In most cases, for grey-scale image, each pixel consists of a value ranging from 0-255. While in colour image, each pixel

consists of three values with range 0-255 each representing the colour RGB (Red-Green-Blue). We can inspect that each values both coloured and greyscale has the same range and can be represented in binary form of 8 bit pattern. These values can be represented as unsigned 8 bit integers.

Considering these, to encode our image, we need to partition our datas such accordingly. We can encode all the image in a single chunk of code. Yet, as Hamming Code limitation, this chunk of code will only able to correctly corrects one bit error. Thus, we need to partition our image to smaller chunks. If we use our Standard Hamming Code, the most suitable would be Hamming(7,4) since we can send 4 bits (half of the data for each value of unsigned integer). But, this come to a cost since we only have efficiency of $\frac{4}{7}$ which is not great. We may need to consider larger chunk of codes. We can extend our Hamming Code such that it will encode every pixel in our image into a single chunk of code and transmit it, such for image with size $w \times h$, the transmitted image can correct itself for a single error in every chunk of code totalling of $w \times h$ errors.

The length of chunk code to transmitted can be choosen accordingly, with larger chunk code will be encoded more efficiently, it comes with less ability as of how much error it may able to correct. To note, the error can be handled is only 1 bit for every chunk of code, such even if we have the ability to handle $w \times h$ errors in total, it still limits us with the ability of correcting one bit error for every block.

Case for Greyscale coloured image, we may want to partition it such that every chunk of code will contain a 1 byte (8 bits) information which we encode by Hamming(12,8). The value of m need will be 4 by checking our requirement of parity check bits in (d). Below is the code for reading an image, encoding the image as Hamming(12,8), and decoding with error correction the 'received' image.

```
import cv2
import numpy as np
from hamming1 import encodeHamming1
from hamming1 import Hamming1

def bitstouint(bits):
    uint = 0
    for bit in bits:
        uint = (uint <<1) | bit
    return uint

img = cv2.imread("Lena.png",0)
height = img.shape[0]
width = img.shape[1]
codewords = [[0 for j in range(width)] for i in range(height)]
received = [[0 for j in range(width)] for i in range(height)]

#encoding every pixel as Hamming(12,8)
for i in range(height):
    for j in range(width):
        codewords[i][j] =
encodeHamming1(format(img[i][j],'08b'), 12,8)

#read and correct every pixel
for i in range(height):
    for j in range(width):
        received[i][j] = bitstouint(Hamming1(codewords[i][j]))
received = np.array(received,dtype=np.uint8)
cv2.imwrite("Lena1.jpg", received)
```

For the test image Lena.png :


Figure 4(a) Lena.jpg


Figure 4(b) Lena1.jpg

Fig. 4(a) is the initial image, and Fig 4(b) on the right is the resulting image after encoding and reconstruction without any error along the proccess. Simulating random single bit error for every pixel in Lena.jpg and save it as Error1.jpg we get an image of :


Figure 5(a) Error1.jpg


Figure 5(b) Lena2.jpg

Fig. 5(a) is the image which we introduce a single bit error for every pixel. Fig. 5(b) is the resulting image after error correction of Hamming Code. We can see the results of Fig. 5(b) is identical with the initial image in Fig. 4(a) as also the case for Fig. 4(b). We can modify our code such that we will able to encode more than 1 pixel in a chunk of code, but it gives us a problem which it won't be able to handle error in every pixel.

We will simulate to create a multiple bit error for every pixel in the image in the process of 'transferring'.


Figure 6(a) Error2.jpg


Figure 6(b) Lena3.jpg

Resulting image after error introducement shown in Fig. 6(a) with resulting image after error correction in Fig. 6(b). The resulting image got more distorted from the error introduced

image, this shows the inablity of Hamming Codes as a code with Hamming Distance of three to correct error more than 1 bit. The resulting error may not detected Hamming Code completely or a false correction happened, thus creating more distorted result. Here we have a difference between Standard Hamming Codes where length of codeword of the form $2^n - 1$ and Expanded Hamming Code. Standard Hamming Code will always detect an error in its codeword (resulting XOR operations will not exceed the index of the codeword), while Expanded form in some cases have index which is not stored (larger than the size of the codeword), such it will pass the error.

IV. Hamming Code Improvements

In practice, error can occur in consecutive in transmission, called as burst error. Normal implementation of Hamming Code is not effective for such cases. To handle this case, we will need to interlace of blocks of encoded message. Such, when we decode our message these burst of errrors will be "redistributed".

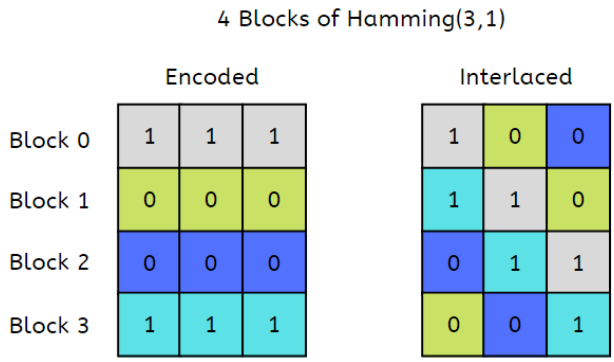

Figure 7. Interlacing of 4 Block Hamming(3,1)

This interlacing will give the ability to correct burst of errors, since on transmission, for given transmission block of data, a burst of errors will be single error for multiple blocks of encoded data. We will first encode our image as usual, after which rather than we straightly transfer our blocks of data, we first interlace our blocks of data. When we receive our data, we also need to deinterlace our data before decoding our data. This implementation still have problem if the error have a periodicity of the blocks of data we interlace. As example, in Fig. 7, we find if the error have a periodicity of 4, we have multiple error for a single block of encoded data. Yet, this will add much more time for interlacing and deinterlacing of our data that may takes us the same proccessing or longer time compared to our encoding time. Code for interlace and deinterlace given below :

```
def interlace(encodedMat):
    codelen = len(encodedMat[0][0])
    width = len(encodedMat[0])
    height = len(encodedMat)
    blocks = width*height
    totalbits = blocks * codelen
    interlaced = [[[0 for k in range(codelen)] for j in range(width)] for i in range(height)]
    n = 0
    for i in range(height):
        for j in range(width):
            for k in range(codelen):
                row = n // (codelen * width)
                col = (n - row *(codelen * width)) // codelen
                bits = n % codelen
                interlaced[row][col][bits] = encodedMat[i][j][k]
                n = n + blocks
                if n >= totalbits :
                    n = (n%totalbits) + 1
    return interlaced
def deinterlace(trfMat):
    codelen = len(trfMat[0][0])
    width = len(trfMat[0])
    height = len(trfMat)
    blocks = width*height
    totalbits = blocks * codelen
    result = [[[] for j in range(width)] for i in range(height)]
    n= 0
    for i in range(height):
        for j in range(width):
            for k in range(codelen):
                row = (n // width) % height
                col = n % width
                result[row][col].append(trfMat[i][j][k])
                n+=1
    return result
```

Using interlace and deinterlace add two extra steps from encode-transfer-decode to encode-interlace-transfer-deinterlace-decode. By interlacing, we will in general have better image for any pattern of error on transferring since it can handle burst errors. Simulating 1 in 8 error chance for every bit on transferring (uniform probability/chance) for both with and without interlace with same pattern of error, resulting decoded image:

Figure 8(a) Non-Interlaced

Figure 8(b) Interlaced

Resulting image shows a better image from interlacing in Fig. 8(b) compared to without interlacing in Fig. 8(a).

Another room for improvement in Hamming Code is by using the 0th bit as our total parity bit. Thus, the algorithm will be able to detect for a double bit error without the ability to correcting it while still having the ability to error a single bit error. This algorithm is called as SECDED (Single Error Correction, Double Error Detection). By extending our Hamming Code, we need to transfer an extra 0th bit, which in effect our total codeword will have a total length of 2^m. In general, we can

extend our Hamming(n,k) codes into Extended Hamming(n+1,k). This in implementation can help by giving an error status for a double bit error (which can't be corrected) and to request retransfering of said data with the downside of more redundant data to be transferred.

V. Conclusion

Hamming Code in its implementation can be used as encoding of any binary code other than image. Hamming Code encoding for image is best use to ensure data transffered while using as little space used for redundant/parity check bits as possible. Hamming Code also can be "expanded" and "extended" from it's base forms such it can encode variety length of message and able two detect two bit errors. To make use of our hamming code, we need to determine how many message bits to be encoded for every block of data. The more blocks we use, the more single bit errors that our algorithm can corrects, with the drawback of less efficiency for every block and overall data. Interlacing also provides better encoding such it will handle burst errors up to certain limitations with the con of extra steps it takes to encode-transfer-decode sequence.

VII. Acknowledgment

I wish to show my appreciation Mr. Rinaldi Munir as this assignment encourages me to do deeper research into other parts in discrete mathematics and to practice on writing research papers. I wish to extend my special thanks to Mr. Grant Sanderson from channel 3blue1brown as his video to introduce me to the field of Error Correcting Codes and its elegance in implementation which is the reason for me to choose this topic as my research.

References

[1] Hamming, R. W. The Art of Doing Science and Engineering: Learning to Learn. Australia: Gordon and Breach, 1997.

[2] Andrew.cmu.edu. 2021. [online] Available: https://www.andrew.cmu.edu/user/nbier/15110/lectures/lec15a_sound_video.pdf [Accessed 05-Dec-2021].

[3] Math.mit.edu, 2021. [Online]. Available: https://math.mit.edu/~goemans/18310S15/Hamming-code-notes.pdf. [Accessed: 05- Dec- 2021].

[4] Math.ryerson.ca, 2021. [Online]. Available: https://math.ryerson.ca/~danziger/professor/MTH108/Handouts/codes.pdf. [Accessed: 06- Dec- 2021].

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2020

Ng Kyle 13520040