Afzal Khan
afkhan@ucsc.edu
04/19/2021

CSE 13s Spring 2021
Assignment 5: Hamming Codes
Design Document

Overview:

The objective of this lab is to be able to encode and decode sets of generated Hamming codes. Hamming codes, named after the applied mathematician Richard Wesley Hamming, is a "linear error checking code." Using bits, the process can encode and decode bit messages with the help of *parity bits*. In this program, we will be using a Hamming(8,4) code. Simply put, the bits are organized in nibbles. The least significant nibble of the byte is the message to be encoded, and the most significant nibble are the *parity* bits. There are two ways to encode the message bit vector. The first is using parity equations, and the other is using a 4 x 8 encoding matrix. Using the formula $\vec{c} = \vec{m}G$, where *c* is the generated hamming code, *m* is the message, and *G* is the encoding matrix. In order to decode a Hamming code, the equation $\vec{e} = \vec{c}H^T$ is used where *e* is the *error syndrome*, *c* is the generated Hamming code, and H^T is the encoding matrix transposed. A bit vector as well as a bit matrix ADT will be implemented in order to implement the algorithm. The code must support a set of command line arguments for both decode and encode.
Encoder:

- -h: Prints out a help message
- -i [infile]: Specify the input file path containing data to encode into Hamming codes
- -o [outfile]: Specify the output file path to write the encoded data

Decoder:

- -h: Prints out a help message
- -i [infile]: Specify the input file path containing Hamming codes to decode
- -o [outfile]: Specify the output file path to write the decoded Hamming codes to
- -v: Prints statistics of the decoding process to stderr

(*sourced from lab pdf)*

Pre-lab Questions:

1. ----------|----------

0	HAM_OK
1	4
2	5
3	HAM_ERR
4	6
5	HAM_ERR
6	HAM_ERR
7	3
8	7
9	HAM_ERR
10	HAM_ERR
11	2
12	HAM_ERR
13	1
14	0
15	HAM_ERR

2.
   a. $1110\ 0011_2$

$$\vec{e} = \vec{c}H^T = (1\ 1\ 0\ 0\ 0\ 1\ 1\ 1) \begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \pmod 2$$

$= (1\ 2\ 3\ 3) \pmod 2$

$= (1\ 0\ 1\ 1)$ row 2

$= (1\ 1\ 0\ 0\ 0\ 1\ 1\ 1) \rightarrow (1\ 0\ 0\ 0\ 0\ 1\ 1\ 1)$

$\vec{m} = 0001_2$

b. $1101\ 1000_2$

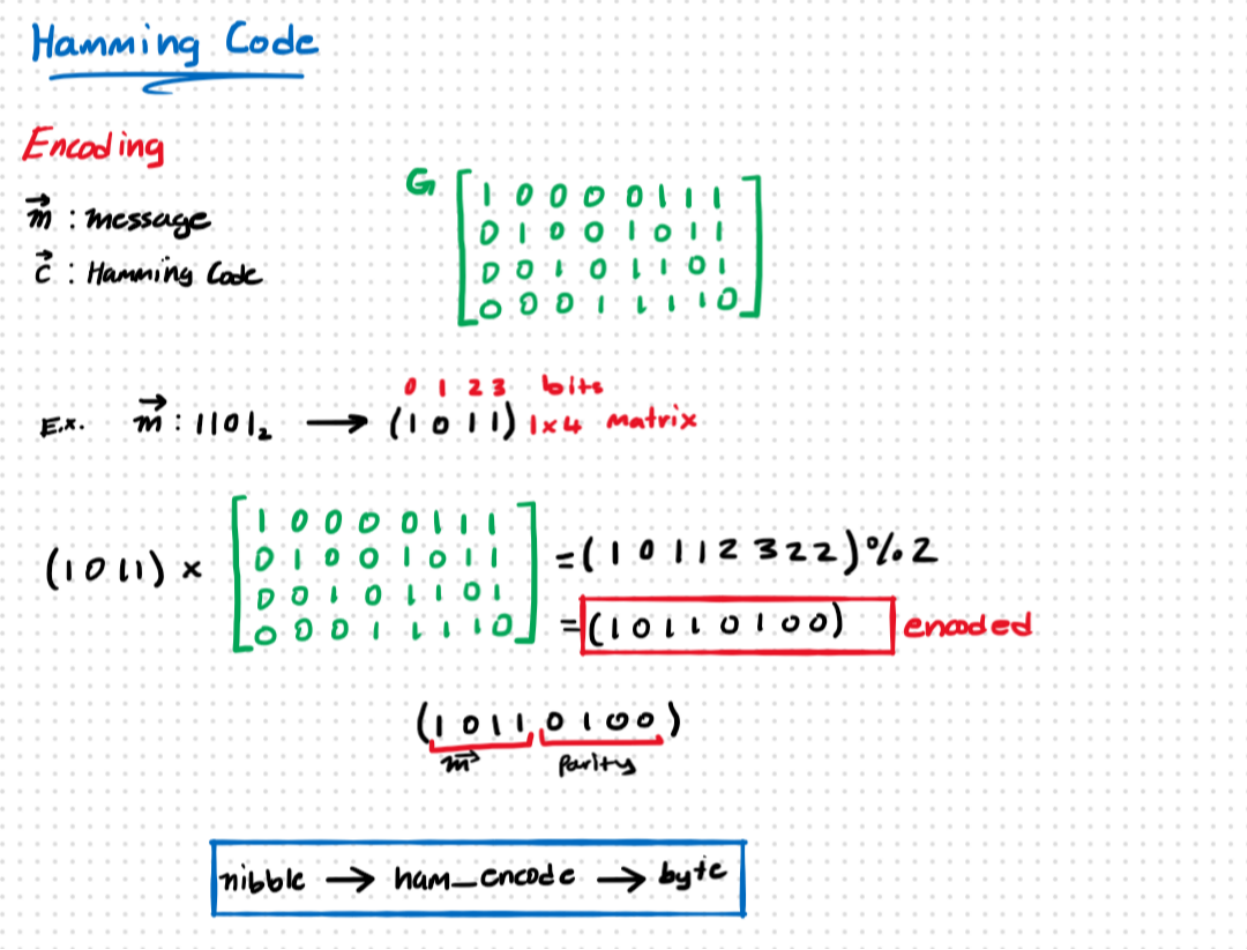
$$\vec{e} = \vec{c}H^T = (0\ 0\ 0\ 1\ 1\ 0\ 1\ 1) \begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \pmod 2$$

$= (2\ 1\ 2\ 1) \pmod 2$

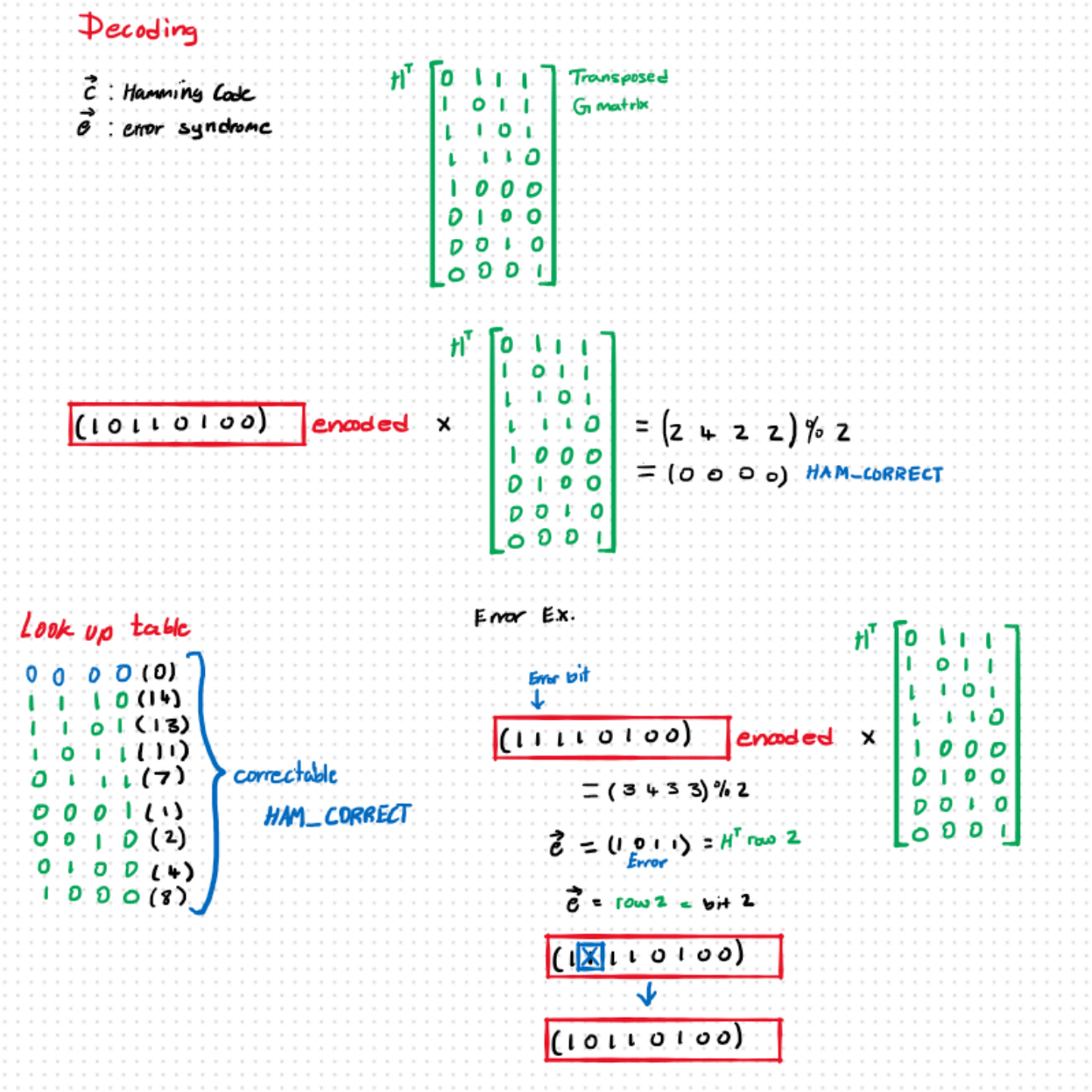
$= (0\ 1\ 0\ 1)$

= HAM_ERR

Approach:



The process involved in encoding the *msg* is quite simple. Essentially, the function *ham_code* is going to accept a nibble or 4 bits. The four bits are then converted using the vector ADT. The bits are organized from left to right with the LSB to the left and MSB to the right. The vector is then used in the bit matrix ADT to perform a matrix multiplication with the encoding matrix. The result of this operation is an encoded 1x8 bit matrix. The first four bits are the encoded *msg* and the last four are the parity bits.



The decoding process is very similar to that of encoding. When decoding, the encoded message is taken as a 1x8 matrix and multiplied with the transpose of the encoding matrix, a 8x4 matrix. The result is a 1x4 matrix. Using a look up table, the resulted bit vector, *e*, is compared against the decoding matrix. Each row on the decoding matrix corresponds to a bit on the original encoded vector. If *e* exists on one of the rows, then the corresponding bit is incorrect and must be flipped. If *e* is does not exist, then the error is uncorrectable. When *e* is (0 0 0 0), it indicates that there is no error, and the first four bits of the encoded bits is the original message.

Pseudocode:

The following will cover a lot of the pseudo code for the different ADTs and algorithms in this program.

BitMatrix

```
structure fot BitMatrix:
    rows
    columns
    BitVector v

BitMatrix bm_create(rows, cols):
    memory allocated for matrix
    m.rows = rows
    m.cols = cols
    m.v = bit vector create

bm_rows:
    return rows

bm_cols:
    return cols;

bm_set_bit(r, c):
    position = current row x # total columns + current column
    bitvector set bit(position)

bm_clr_bit(r, c):
    position = current row x # total columns + current column
    bitvector clr bit(position)

bm_get_bit(r, c):
    position = current row x # total columns + current column
    bitvector get bit(position)
```

```
bm_from_data(byte, length):
    assert that length is either 4 or 8
    x = create a bit matrix
    for i in (0, length):
        if (byte bitwise and with (1 offset by i)) > 0:
            set bit at row 0 col i to 1 in matrix x
    return matrix x

bm_to_data:
    varaible data
    for i in (0,8):
        if (bit at row 0 col i) == 1:
            data = data bitwise or (1 offset by i)

    return data;

bm_multiply(A, B):
    x = create bit matrix using rows of A and cols of B

    product = varible for product of two matrix postions
    for i on rows of matrix:
        for j in cols of matrix x:
            for k in cols of matrix B:
                product = bit at A(i,k) bitwise and B(k, j)
                position = current row of x # total columns + current column
                add the product to matrix x at position
    return matrix x

bm_print:
    for row in matrix:
        for cols in matrix:
            print bit at (row, col)
        print new line
```

BitVector

```
BitVector bv_create(length):
    v = allocate memory for struct
    v.length = length
    v.vector = allocate memory for vector

bv_length(v):
    return v.length

bv_set_bit(i):
    v.vector(byte location) bitwise or with (1 offset to i)

bv_clr_bit(i):
    v.vector(byte location) bitwise and with complement(1 offset to i)

bv_get_bit(i):
    return v.vector(byte location) shifted right (1 offset to i)

bv_xor_bit(i, bit):
    return v.vector(byte location) XOR (bit offset to i)

bv_print(v):
    iterate through length and get bits
```

Hamming

```
var total bytes counter = 0
var uncorrected errors = 0
var corected errors = 0

ham_encode(G, msg):
    x = create matrix from msg data of length 4
    y = resulting matrix after multiply matrix x with encoding matrix G
    msg = matrix y back to data
    delete matrix x
    delete matrix y
    return msg
```

```
ham_encode(Ht, code, msg):
    total bytes counter++
    lookup_table[16] = create an array look up table with al possible values and
results
    x = create matrix from code data of size 8
    y = resulting matrix from multiplying matrix with decoding matrix Ht
    temp = matrix y back to data
    check = lookup_table[temp]
    if value of check = no errors:
        msg = lower nibble of code
        return no errors enum
    else if value of check = fixable errors:
        corrected errors++
        msg = lower nibble of corrected code
        return error fixed enum
    else:
        uncorrected errors++
        return error enum
```

Encode

```
lower_nibble:
    function to get lower nibble of byte

upper_nibble:
    fucntion to get upper nibble og byte

pack:
    function to combine two nibbles

main function:
    set infile and outfile to defaul locations
    bool var help = false to determine when to print help message
    var choice = command line option
```

```
while (get options from command line != end of line)
    switch
        case 1:
            set bool help flag = true
        case 2:
            set infile to specified infile
            if (unable to open):
                print(unable to open)
        case 3:
            set outfile to specified location
            if (unable to open):
                print(unable to open)
        default:
            prints help message

set file permissions to match

if help flag = true:
    print help message

create G encoding matrix

encoding look up table[16];
for i in encoding look up table:
    encoding look up table at i endex = hamming encode of i

while byte = chars in infile != end of file:
    lower = lower nibble of byte
    upper = upper nibble of byte

    lower byte = encoding look up table[lower]
    upper byte = encoding look up table[upper]

    put lower byte in outfile
    put upper byte in outfile

close outfile
close infile

delete G encoding matrix

return 0
```

Decode

```
lower_nibble:
    function to get lower nibble of byte

upper_nibble:
    fucntion to get upper nibble og byte

pack:
    function to combine two nibbles

main function:
    set infile and outfile to defaul locations
    bool var help = false to determine when to print help message
    bool var stats = false to determine when to print file stats
    var choice = command line option

    while (get options from command line != end of line)
        switch
            case 1:
                set bool help flag = true
            case 2:
                set bool stat flag = true
            case 3:
                set infile to specified infile
                if (unable to open):
                    print(unable to open)
            case 4:
                set outfile to specified location
                if (unable to open):
                    print(unable to open)
            default:
                prints help message

    set file permissions to match

    if help flag = true:
        print help message

    create Ht encoding matrix
```

```
    while byte = chars in infile != end of file:
        lower nibble = ham decode(byte)
        byte = next byte in file
        upper nibble = ham decode(byte)
        pack both lower and upper nibbles together to construct original char
        put char in outfile

    if stats = true:
        print out stats about errors and fixes

    close outfile
    close infile

    delete Ht decoding matrix

    return 0
```

The *BitVector* ADT along with the *BitMatrix* ADT are strongly intertwined with each other. Essentially, the bitvector structure creates a matrix vector of the binary values of a char. These values are organized in order from LSB to MSB as opposed to MSB to LSB in binary. This is to help with the matrix arithmetic. The function implemented withing the bitvector ADT are to help manipulate individual bits. Using bitwise operators such as *or*, *and*, *xor*, the functions are able to retrieve the values of bits in specific locations of the byte. The logical shift left and right are strongly responsible for being able to offset bits and retrieve their values.

The bitmatrix ADT uses this functionality of vectors to implement a matrix of any row by column size. The functionality of the bitmatrix structure allows for accessing of specific bit on the matrix, as well as setting them. One of the most important functionalities of the bitmatrix is its ability to take a char and convert it into a matrix as well as going backwards. This helps with reading of the file chars and quickly converting them into code that must be encoded.

Together, the two ADTs make up the entirety of the hamming logic. The encode and decode function simply take the code handed to them, covert to a matrix, multiply by their respective encode or decode matrix, and covert the result into a char. Although this a very superficial explanation, the ADTs simplify the process greatly.

Optimization:

The greatest optimization in this code comes from the encoding look up table. Being a form of memorization, the encode file essentially takes all the possible 4 bit values and throws them into an array as indexes. Then each binary value is assigned it corresponding hamming encode. This way in large files, the algorithm does not have to run functions like bitmatrix multiply, numerous times. This is especially helpful sine it has a time complexity of $O(n^3)$.

Outcomes:

From this lab I learned:

- How to create a BitVector and BitMatrix ADT
- How to represent binary values
- Hamming(8, 4)
- Parity bits
- How to manipulate and change binary values using bitwise operators
- Learned the significance of file permissions
- Learned how to extract data from a file in terms of bytes
- Packing and unpacking nibbles and bytes