Sebastian Carbonero
sicarbon@ucsc.edu
05/5/2021

CSE13s Spring 2021
Assignment 5: Hamming Codes
Design Document

In this assignment, you will be creating a program that encodes and decodes user inputs with hamming codes. This program will also be able to correct potential errors that occurred during the delivery process of a message. You will need to create a Bit vector ADT which will let the user modify bits within a byte and will be a dependency for the Bit matrix ADT. The Bit Matrix will be used by hamming code.c to encode and decode the messages.

PRE LAB QUESTIONS

Understanding of Hamming Codes

1. Complete the rest of the look-up table shown below.

Look up table:

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

2. Decode the following codes. If it contains an error, show and explain how to correct it. Remember, it is possible for a code to be uncorrectable.

(a) 1110 0011

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

Using the matrix on the right (Figure 1.1), we can matrix multiply the flipped bits of 1110 0011 which would be 1100 0111.

So Multiplying ((1 1 0 0 0 1 1 1) x Figure1.1) mod 2 would give us:

0 + 1 + 0 + 0 + 0 + 0 + 0 + 0 = 1 % 2 = 1
1 + 0 + 0 + 0 + 0 + 1 + 0 + 0 = 2 % 2 = 0
1 + 1 + 0 + 0 + 0 + 0 + 1 + 0 = 3 % 2 = 1
1 + 1 + 0 + 0 + 0 + 0 + 0 + 1 = 3 % 2 = 1

Figure 1.1

So the final answer is (1 0 1 1). But we have an error in this message, so we can fix it by looking at the table, and since 1011 is in the second row of figure 1.1, that means that the error is at the second index of (1 1 0 0 0 1 1 1), so we can flip that bit and the result would be (1 0 0 0 0 1 1 1) which would be the original message.

(b) 1101 1000

Using the matrix (Figure 1.1), we can matrix multiply the flipped bits of 1101 1000 which would be 0001 1011.

So Multiplying ((0 0 0 1 1 0 1 1) x Figure1.1) mod 2 would give us:

0 + 0 + 0 + 1 + 1 + 0 + 0 + 0 = 2 % 2 = 0
0 + 0 + 0 + 1 + 0 + 0 + 0 + 0 = 1 % 2 = 1
0 + 0 + 0 + 1 + 0 + 0 + 1 + 0 = 2 % 2 = 0
0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1 % 2 = 1

So the final answer is (0 1 0 1). We obviously have an error in this message, but it is uncorrectable since 0101 does not match any of the indexes in figure 1.1. Also 5 Represents HAM_ERR in the lookup table, so we would return HAM_ERR.

PROGRAM SPECIFICS

How the Program will work

You will create two programs that will run separately from each other. The program names will be called encode.c and decode.c

For encode.c and decode.c, when the user runs the commands:

```
$ ./encode -[hi:o:]
```

The command ./encode by itself will let the program hang and wait for the user to input any text. By default, the program itself will use **stdin** to get the information the user adds.

Similarly, you can pipe in files with stdin like `./tsp < filename or cat filename | ./tsp`

When the user uses -h, a help guide will pop out and when the user inputs - i, we will not use the default stdin, instead we will use fopen() to get the data from the user-specified file name. If the user inputs -o, the program will write the program's output into a file of user choice. Similarly, stdout will be the default if the user decides not to use -o.

To parse the arguments, we are going to use getopt.

The user inputs and outputs for encode.c and decode.c should act the same.

TOP LEVEL DESIGN

Pseudo Code and Explanations of program

You will create two programs that will have two mains, and the programs should take in user inputs like so:

In order to optimize this program, we can check if the bytes inputted by the file have been already decoded or encoded.

To do this, we are going to create an array that will store the memory based on the byte or nibble value. For the encode, we will split the byte and put the values of the encoded byte into the position of the nibble index that was encoded. For the next characters, we will check if the same values have been stored in the nibble value.

For the decode, we are going to store the decoded values into the index of the byte value. Once we decode the byte, we will have a nibble of value. With this, we can store this nibble value onto the index of the byte value to make the process of decoding faster.

The main function for decode.c and encode.c should look something similar to this:

```
int main(int argc, char *argv[]):
FILE infile = stdin, outfile = stdout
Int opt = 0

while ((opt = getopt(argc, argv, [hvui:o:] aka OPTIONS)) != -1):
      switch (opt): // check for user inputs
            case h then print help guide
            case i Open the file and check for file open failure
            case o Close the file and check for file open failure
            case ? then print out help guide

Return the status of the fgetc inputs
```

Matrix G **(Figure 1.2)** for the encode matrix will look like this:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

For the **encode.c** file input function, we are going to use fgetc and to get each byte in the file to encode. Also with each byte we are going to need to split the byte into two and turn it into two nibbles. Then send each nibble to the ham_encode function.

```
read_file(infile, outfile) {
      Here create a hamming 8x4 matrix with contents of matrix G
      figure 1.2

      // Create a while loop for user inputs
      while(not end of file) {
        user_input = fgetc(infile)



        // Here split the user input into nibbles
        code1 = encode(8x4 matrix, lsb_nibble)
        code1 = encode(8x4 matrix, msb_nibble);
        fputc(code1, outfile);
        fputc(code2, outfile);
      }
}
```

To get the upper nibble use something like this:

```
upper_nibble = upper_nibble >> 4
```

This will clear the lower half of the nibble to only receive the top nibble.

To get the lower nibble use something like this:

```
lower_nibble = lower_nibble & 0xF
```

Matrix H **(Figure 1.3)** for the decode matrix will look like this:

$$H = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

For the **encode.c** file input function, we are going to use fgetc and to get two bytes each from the infile. For each byte pair read, decode the hamming (8,4) with ham_decode to recover the original message, and pack the two nibbles together into a byte

```
read_file_decode (infile, outfile) {
      Here create the transpose matrix of the parity check matrix H^t using
      bm_create()

      // Create a while loop for user inputs
      while(not end of file) {
        // Get two bytes
        byte1 = fgetc(infile)
        byte2 = fgetc(infile)



        // Here split the user input into nibbles

        // check if byte has been decoded already
        if (decoded1 = check_mem(byte1), decoded1 == 0):
              // if byte has not been decoded, call the decode function
              decoded1 = decode(H^t Matrix, lsb_byte_code)
        if (decoded2 = check_mem(byte2), byte == 0):
              decoded2 = decode(H^t Matrix, msb_byte_code);

        // pack byte
        packbyte(decoded1, decoded2)
        fputc(packed_up_byte, outfile);
      }
}
```

To pack up the byte, it should look something like this:

```
lower_nibble = lower_nibble & 0xF
```

HAMMING CODES

The backbone of the program

Encode Function:
In order to create hamming codes, we are going to need to use matrix multiplication to encode and decode the messages.

For the encode function, we should receive a message with only the lower nibble of the byte. With each message, we are going to create a matrix of 4 rows and 1 column (for the nibble message). To do this, we are going to use **bm_from_data**.

Then, we are going to multiply matrix nibble with matrix G to get the result of 6 rows and 1 column. To format the output, we are going to use **bm_to_data**. Finally, delete the matrices that were created to prevent memory leaks.

And finally return the encoded byte.

The function should look something similar to this:

```
ham_encode(Matrix G, uint8_t msg):
     nibble_matrix  = create a 4x1 matrix with bm_from_data
     result_multiply = multiply the nibble_matrix and Matrix G with
bm_multiply
     result = format the bit matrix multiply to only the first element of
the matrix
     bm_delete(&nibble_matrix)
     bm_delete(&result_multiply)
```

Decode Function:

To decode the hamming message, we are going to need to receive 1 byte at a time and transform the byte into a nibble (which would be either the lower or upper nibble of the original message).

We are first going to matrix multiply the encoded message to decode the message and see if the message has any errors. To check, we will loop through the lookup table and see if the message matches any of the values in the indexes.

If the value matches an index with value HAM_ERR, we will just return a HAM_ERR and set the msg to the error of the lower nibble.

If HAM_OK, then we will return HAM_OK status and the message.

If the result matches the value of the index in the lookup table, we are going to need to flip the bit of the encoded message to fix the error.

The function should look like this:

```
ham_decode(Matrix H^t, uint8_t code, uint8_t *msg):
     code_matrix  = create a 8x1 matrix with code using bm_from_data
     result_multiply = multiply the code_matrix and Matrix H^t with
bm_multiply
     result = format the bit matrix multiply to only the first element of
the matrix
    // since we now have a nibble, loop through the lookup table
     if result == HAM_OK:
         *m = lower half of the byte
          return lookup[i]
     if result == HAM_ERR:
         *m = lower half of byte
          Return lookup[i]
     If result == lookup[i]:
         // flip bit of index of place i of the byte
         // then fix the message and set message pointer to m
     bm_delete(&result_multiply)
     bm_delete(&byte_matrix)
```

ADT PSEUDOCODE

The following pseudocode will show how the ADTs will work.

Bit Vectors

In order for our bit matrix to work, we are going to need to implement a bit vector which would hold an array of bytes.

To get the position of each byte or index, we are going to need to divide index/8 to get the floor of the index. So with this information, we can do bit wise operations to **get a certain bit** we want, **set a bit**, and **clear a bit**

To set a certain bit, we are going to want to:

```
bit = bit | (1 << (i % 8));
```

This lets us mask the certain bit we want by shifting the masked bit left to the ith bit we want.

To get a certain bit, we are going to want to:

```
(bit >> (i % 8) & 0x1)
```

This lets us shift the bit we want to the lsb which ANDing it with 1 would clear the rest of the bits to the left.
To clear a bit, we are going to perform:

```
byte & ~(0x1 << (i % 8))
```

This lets us mask the bit we want with the opposite of its value, then ANDing it would clear the certain bit.

Bit Matrix
For our hamming codes to work, we are going to need to create a bit matrix which would use the Bit vector ADT to perform matrix multiply, bit matrix to data, and bit matrix from data.

To get a certain bit in the bit matrix we are going to use this formula for the index.

```
index row * matrix columns + index column
```

Just like the example in prelab question 2, we are going to need to use that same process to multiply matrices.

So, if bitmatrix A columns = to bitmatrix B columns,
we are going to create a bit matrix of A rows x b columns as the result.

To perform matrix multiplication in code, it should look something similar to this:

```
for i in a->rows:
      for j in b->columns:
            product = 0
            for k in a->columns:
                  product += getbit(A, i, k) * getbit(B, k, j)
                  if (product % 2 == 1):
                        setbit(resultmatrix, i, j)
```

Then for **bm_from_data**, we are going to need to create a bit matrix of either 1x4 or 1x8 (for a byte or a nibble). So we are going to want to loop through the length of the input then set each of the bits from the input.

It should look something similar to this:

```
bm = bm_create(length, 1)
for i in length:
      b = (byte >> i) & 1 // get bit to check
      if  (b == 1):
            setbit(bm, i, 0)
return bm
```

Then for **bm_to_data**, we are going to need get the data from the bit matrix and turn it into a byte of uint8_t

It should look like this:

```
byte = 0
for i in range(8):
      byte |= getbit(matrix result, i) << i
return byte
```

DESIGN PROCESS

Throughout this lab, I modified my design many times

- At the beginning of this lab, I did not use stdin, so I had to create a new function for stdin for std in to be the default
- At first, I did not flip the byte for the decoding which took a while to figure out, but once I did, everything else worked.
- I optimized the program by storing the encode and decoding values into an array for the programs to check if the values have already been decoded or encoded.

Overall, I learned a lot from this project. It taught me how to deal with multiple moving parts for the program to work together, showing me how important it is for your programs to be reliable.