Pre-lab

$$G = \begin{pmatrix} 1000 & 0111 \\ 0100 & 1011 \\ 0010 & 1101 \\ 0001 & 1110 \end{pmatrix} \qquad \boxed{\begin{array}{c} \vec{c} = \vec{m} \cdot G \\ \uparrow \\ \text{from } 0000_2 \rightarrow 1111_2 \end{array}}$$

1. $\overset{0000_2}{\underline{\vec{c}(0000)}} = (0\,0000000)$ $\quad \overset{1000_2}{\vec{c}(0001)} = (00011110)$ $\quad \overset{0100_2}{\vec{c}(0010)} = (00\,1011\,01)$

$\overset{1100_2}{\vec{c}(0011)} = (00110011)$ $\quad \overset{0010_2}{\vec{c}(0100)} = (01001011)$ $\quad \overset{1010_2}{\vec{c}(0101)} = (01010101)$

$\overset{0110_2}{\vec{c}(0110)} = (01100110)$ $\quad \overset{1110_2}{\vec{c}(0111)} = (011\cancel{1}100\,0)$ $\quad \overset{0001_2}{\vec{c}(1000)} = (10000111)$

$\overset{1001_2}{\vec{c}(1001)} = (10011001)$ $\quad \overset{0101_2}{\vec{c}(1010)} = (10101010)$ $\quad \overset{1101_2}{\vec{c}(1011)} = (10110100)$

$\overset{0011_2}{\vec{c}(1100)} = (11001100)$ $\quad \overset{1011_2}{\vec{c}(1101)} = (11010010)$ $\quad \overset{0111_2}{\vec{c}(1110)} = (1110\,0001)$

$\overset{1111_2}{\vec{c}(1111)} = (11111111)$

<u>pseudocode:</u> m[rows][cols] G[rows][cols]

```
for ( int i=0 ; i < m_rows ; i++){
    for( int j=0 ; j< G_cols ; j++){
        for (int k =0 ; k < m_cols; k++){
            c[i][j]  += m[i][k] x G[k][i];
        }
    }
}
```

2. $\vec{e} = \vec{c} \cdot H^T$

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

a) $11100011_2 \Rightarrow \vec{c} = (1\underset{\uparrow}{1}000111) \longrightarrow$ then $\vec{e} = (1\underline{0}11)$. This contains an error which can be matched up with $H^T[1]$.

need to be flipped, and ~~the~~ original message becomes 0001_2

So it can be corrected by flipping over the ~~[illegible]~~ 1st bit. Return HAM_ERR_OK

b) $11011000_2 \Rightarrow \vec{c} = (00011011) \longrightarrow \vec{e} = (0101)$. This contains an error and cannot be fixed for there're more than one bits are flipped.

Return HAM_ERR.

Decoding pseudocode: $\vec{e}[0][cols]$ $H^T[rows][cols]$

```
int find;
for( int i=0; i<H^T_rows; i++){
    for( int j=0; j<H^T_cols; j++){
        if ( e[0][j] != H^T[~~rows~~ i][j]){
            find = -1;
            break the loop }
        find = i;  ← the ith bit need to be flipped
    }
}
if(find == -1) { return HAM-ERR }
else{ check the ith bit's value (0 or 1)
    then use bm_set-bit( ) /bm_clr_bit( ) }
```

3.

0	0
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

$$H^T = \begin{array}{c|c} & \text{index:} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} \end{array}$$

Wendi Tan
wtan37@ucsc.edu
2/4/2021

CSE13s Winter 2021
Assignment 4: Hamming Codes
Design Document

Hamming codes are a family of linear error-correcting codes. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors. By contrast, the simple parity code cannot correct errors, and can detect only an odd number of bits in error. Hamming codes are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance of three.
(cite through https://en.wikipedia.org/wiki/Hamming_code)

In this lab, I wrote two programs in C, one for encoding hamming codes, and the other for decoding them. More specifically, the user is asked to enter any input and output files by typing -i or -o, followed by streams (*stdin* and *stdout* are default files). The encoding program would read bites through the given message in the input file and generate its corresponding hamming code, printed in the output file. Oppositely, the decoding program would decode hamming codes, identify errors, correct certain errors, and generate the original message.

The entire project possesses seven program files including `bm.c, bm.h, decoder.c, error.c, generator.c, hamming.c, hamming.h`. Generator and decoder are two programs we are going to execute, and both of them include `bm.h` and `hamming.h`, which include called functions.

Top-Level:

1. Main functions in bm.c:

This bm.c file contains functions which can be called when building bit matrices including bm_create(), bm_delete(), bm_rows(), bm_cols(), bm_set_bit(), bm_clr_bit(), bm_get_bit(), bm_print(). (BitMatrix is an APT created inside bm.c)

*BitMatrix *bm_create(uint32_t rows, uint32_t, cols):* create a bit matrix.

*void bm_delete(BitMatrix *m):* free row-pointers, free matrix[][] pointer, and finally free the BitMatrix pointer.

*uint32t_t bm_rows(BitMatrix *m):* return *m's rows.

*uint8t_t bm_cols(BitMatrix *m):* return *m's columns.

*void bm_set_bit(BitMatrix *m, uint32_t r, uint32_t, c):* set m[r][c] as 1.

```
Pseudocode:
byte = m[r][c/8]
c = c % 8
mask = 1 left shift by c
result = byte | mask
return result
```

*void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t, c):* set m[r][c] as 0.

```
Pseudocode:
byte = m[r][c/8]
c = c % 8
mask = not(1 left shift by c)
result = byte & mask
return result
```

*void bm_get_bit(BitMatrix *m, uint32_t r, uint32_t, c):* get the bit in m[r][c].

```
Pseudocode:
byte = m[r][c/8]
c = c % 8
mask = 1 left shift by c
result = (byte & mask)right shift by c
return result
```

*void bm_print(BitMatrix *m):* a debugging function, printing the bit matrix m.

```
Pseudocode:
for(i = 0, i < m_rows){
      for(j = 0, j < m_cols){
            print (bm_get_bit(m[i][j]))
      }
      print new line
}
```

2. **main functions in hamming.c: ham_init(), ham_encode() & ham_decode()**

ham_init():

The ham_init() initiates two bit-matrix, generator and parity, allowing usage in generator and decoder programs. I wrote it by calling the bm_create() function to generate two bit-matrices. And then set bit for each place generator and parity contain a “1”.

```
Pseudocode:

Create bit-matrices generator and parity;
set_bit (generator, 0, 0), set_bit (generator, 1, 1), set_bit
(generator, 2, 2) … ;
set_bit (parity, 0, 1), set_bit (parity, 0, 2), set_bit (parity, 0,
3) … ;
check whether they are successfully created, and return HAM_ERR or
HAM_OK.
```

*ham_encode(uint8_t data, uint8_t *code):*

This encoding function requires an 8-bit-unsigned “data” (but entered as a nibble) and a pointer called “code.” “Data” is the integer format of the vector *m*, the original message, and “*code” is “data’s” hamming code generated though the equation *c* = *m* * *generator*. Since I need all the parameters in the equation to be bit matrices, the first thing I did is to convert “data” into a bit matrix:

```
Pseudocode:

Variables: data, data_vector(1*4 matrix), mask, bit

for (int i = 0, i < rows of data_vector) {   //loop for 4 times since
data_vector has 4 columns
      mask = 1 << i
      bit = mask & data
      bit = bit >> i
      if bit == 1, set data_vector[0][i] as 1 otherwise 0
}
```

Next step is to calculate hamming code for “data” (original message):

```
Variables: data_vector(1*4 matrix), generator(4*8 matrix), c(hamming
code, 1*8 matrix)

for (int i = 0, i < rows of data_vector){
      for(int j = 0; j < columns of generator){
            for (int k = 0; k < rows of generator){
                  c[i][j] += data_vector[i][k] * generator[k][j]
            }
            If c[i][j] % 2 == 1, set c[i][j] as 1 otherwise 0
      }
}
```

Now we get the humming code matrix *c*. Since the value of pointer points to uint8_t, we need convert bit matric *c* to an 8-bit-unsigned integer:

```
Variables: c(1*8 matrix), code_integer, mask, bit

code_integer = 0
for (int i = 0, i < 8){
      mask = c[0][i]
      bit = mask << i
      code_integer = code_integer | bit
}
*code points to code_integer
```

Thus, combining three parts we can get the whole encoding code.

*ham_decode(uint8_t code, uint8_t *data):*

In decoding function, I separated it into two parts: get bit matrix e (error syndrome), and correct it to form the original message if possible. Since parts of the program show similar as ones in ham_encode(), I wrote three functions instead of repetitive code, which are *int_to_mat*(*//convert uint8_t to bit matrix*), *mat_to_int*(*//convert uint8_t to bit matrix*), and *mat_mult*(*//matrices multiplication*).

Part 1: calculating *e* using *e = c * parity* (H^T), *c* is the bit matrix of "uint8_t code". Similar as for encoding code, I converted the integer "code" into a bit matrix *c*, and then get the bit matrix *e*.

Part 2: correcting the error: 1). Build up a look-up array and convert *e* to an integer "error" to find whether the error can be corrected in look_up[error]; 2). If the error can be corrected, flip the corresponding bit, and get the original message.

```
Pseudocode:

Variables: look_up[16] = {0, 4, 5, -1, 6, -1, -1, 3, 7, -1, -1, 2, -1, 1,
0, -1},error(uint8_t), e (bit matrix for error), error_vector(1*4),
code_vector(1*8), data, code

      create data_vector and code_vector;
      convert code to code_vector;
      multiply code_vector by $H^T$ to get error_vector;
      convert error_vector to uint8_t error;
      and check the corresponding lookup[error] to determine which bit to
      flip, if error>15 or error<0, cannot correct the error;
      then flip the bit by either set the bit or clear the bit.
```

3. **Program in generator.c:**

The generator.c is the encoding program. It contains one main function and two of the functions provided in the instruction: lower_nibble() and upper_nibble().

Structure:

- Use getopt() to parse command-line options and open the infile and outfile. →
- Initialize G and H^T. →
- Read byte from infile, dividing it up into upper nibble and lower nibble with lower_nibble() and upper_nibble(). →
- Print generated hamming code into outfile one by one. →
- Repeat above two steps. →
- Destroy matrices and close files.

4. **Program in decoder.c:**

The decoder.c is the decoding program. It contains one main function and the function provided in the instruction, pack_byte().

Structure:

- Use getopt() to parse command-line options and open the infile and outfile. →
- Initialize G and H^T, setup variables uncorrected_error, corrected_error, read_bytes, and error_rate as 0. →
- Read two bytes from infile →

- Print generated original message into outfile as one byte (use pack_byte() to combine upper and lower nibbles). →
- Repeat above two steps, and count uncorrected_error, corrected_error, read_bytes in side the loop →
- Destroy matrices and close files. →
- Calculate error_rate with error_rate = uncorrected_error/ read_bytes →
- Print out uncorrected_error, corrected_error, read_bytes, and error_rate.

This lab took me quite a long time, but some errors still exist for I don't have any time to fix them because of the midterms and loads of homework. However, I think the basic structure of generator and decoder is working.