Tu-An Nguyen
tunhnguy@ucsc.edu
02/07/2021

CSE 13S Spring 2020
Assignment 4: Hamming Codes
Design Document

Pre-lab

Questions

1. Calculate Hamming codes for $0000_2 - 1111_2$ using the generator matrix. Show your work.
2. Decode the following codes. If it contains an error, explain how you can correct it; however, some errors cannot be corrected.
    a. $1110\ 0011_2$
    b. $1101\ 1000_2$
3. Complete the rest of the look-up table shown below.

0	0
1	5
...	...
15	HAM_ERR

My work

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

A x G when A = (0000)

C[0][0] = 0 % 2 = 0

C[0][1] = 0 % 2 = 0

⋮

C[0][7] = 0 % 2 = 0

∴ C = (0000 0000)

A x G when A = (0001)

C[0][0] = 0 % 2 = 0

C[0][1] = 0 % 2 = 0

C[0][2] = 0 % 2 = 0

C[0][3] = 1 % 2 = 1

C[0][4] = 1 % 2 = 1

C[0][5] = 1 % 2 = 1

C[0][6] = 1 % 2 = 1

C[0][7] = 0 % 2 = 0

∴ C = (0001 1110)

A x G when A = (0010)

C[0] = 0 % 2 = 0

C[1] = 0 % 2 = 0

C[2] = 1 % 2 = 1

C[3] = 0 % 2 = 0

C[4] = 1 % 2 = 1

C[5] = 1 % 2 = 1

C[6] = 0 % 2 = 0

C[7] = 1 % 2 = 1

∴ C = (0010 1101)

A x G when A = (0011)

C[0] = 0 % 2 = 0

C[1] = 0 % 2 = 0

C[2] = 1 % 2 = 1

C[3] = 1 % 2 = 1

C[4] = 2 % 2 = 0

C[5] = 2 % 2 = 0

C[6] = 1 % 2 = 1

C[7] = 1 % 2 = 1

∴ C = (0011 0011)

A x G when A = (0100)

C[0] = 0 % 2 = 0

C[1] = 1 % 2 = 1

C[2] = 0 % 2 = 0

C[3] = 0 % 2 = 0

C[4] = 1 % 2 = 1

C[5] = 0 % 2 = 0

C[6] = 1 % 2 = 1

C[7] = 1 % 2 = 1

∴ C = (0100 1011)

A x G when A = (0101)

C[0] = 0 % 2 = 0

C[1] = 1 % 2 = 1

C[2] = 0 % 2 = 0

C[3] = 1 % 2 = 1

C[4] = 2 % 2 = 0

C[5] = 1 % 2 = 1

C[6] = 2 % 2 = 0

C[7] = 1 % 2 = 1

∴ C = (0101 0101)

A×G when A = (0110)

C[0] = 0 % 2 = 0

C[1] = 1 % 2 = 1

C[2] = 1 % 2 = 1

C[3] = 2 % 2 = 0

C[4] = 2 % 2 = 0

C[5] = 1 % 2 = 1

C[6] = 1 % 2 = 1

C[7] = 2 % 2 = 0

$\boxed{C = (0110\ 0110)}$

A×G when A = (0111)

C[0] = 0 % 2 = 0

C[1] = 1 % 2 = 1

C[2] = 1 % 2 = 1

C[3] = 1 % 2 = 1

C[4] = 3 % 2 = 1

C[5] = 2 % 2 = 0

C[6] = 2 % 2 = 0

C[7] = 2 % 2 = 0

$\boxed{C = (0111\ 1000)}$

A×G when A = (1000)

C[0] = 1 % 2 = 1

C[1] = 0 % 2 = 0

C[2] = 0 % 2 = 0

C[3] = 0 % 2 = 0

C[4] = 0 % 2 = 0

C[5] = 1 % 2 = 1

C[6] = 1 % 2 = 1

C[7] = 1 % 2 = 1

$\boxed{C = (1000\ 0111)}$

A×G when A = (1001)

C[0] = 1 % 2 = 1

C[1] = 0 % 2 = 0

C[2] = 0 % 2 = 0

C[3] = 1 % 2 = 1

C[4] = 1 % 2 = 1

C[5] = 2 % 2 = 0

C[6] = 2 % 2 = 0

C[7] = 1 % 2 = 1

$\boxed{C = (1001\ 1001)}$

A×G when A = (1010)

C[0] = 1 % 2 = 1

C[1] = 0 % 2 = 0

C[2] = 1 % 2 = 1

C[3] = 0 % 2 = 0

C[4] = 1 % 2 = 1

C[5] = 2 % 2 = 0

C[6] = 1 % 2 = 1

C[7] = 2 % 2 = 0

$\boxed{C = (1010\ 1010)}$

A×G when A = (1011)

C[0] = 1 % 2 = 1

C[1] = 0 % 2 = 0

C[2] = 1 % 2 = 1

C[3] = 1 % 2 = 1

C[4] = 2 % 2 = 0

C[5] = 3 % 2 = 1

C[6] = 2 % 2 = 0

C[7] = 2 % 2 = 0

$\boxed{C = (1011\ 0100)}$

A x G when A = (1100)

C[0] = 1 % 2 = 1

C[1] = 1 % 2 = 1

C[2] = 0 % 2 = 0

C[3] = 0 % 2 = 0

C[4] = 1 % 2 = 1

C[5] = 1 % 2 = 1

C[6] = 2 % 2 = 0

C[7] = 2 % 2 = 0

$\boxed{C = (1100\ 1100)}$

A x G when A = (1101)

C[0] = 1 % 2 = 1

C[1] = 1 % 2 = 1

C[2] = 0 % 2 = 0

C[3] = 1 % 2 = 1

C[4] = 2 % 2 = 0

C[5] = 2 % 2 = 0

C[6] = 3 % 2 = 1

C[7] = 2 % 2 = 0

$\boxed{C = (1101\ 0010)}$

A x G when A = (1110)

C[0] = 1 % 2 = 1

C[1] = 1 % 2 = 1

C[2] = 1 % 2 = 1

C[3] = 0 % 2 = 0

C[4] = 2 % 2 = 0

C[5] = 2 % 2 = 0

C[6] = 2 % 2 = 0

C[7] = 3 % 2 = 1

$\boxed{C = (1110\ 0001)}$

A x G when A = (1111)

C[0] = 1 % 2 = 1

C[1] = 1 % 2 = 1

C[2] = 1 % 2 = 1

C[3] = 1 % 2 = 1

C[4] = 3 % 2 = 1

C[5] = 3 % 2 = 1

C[6] = 3 % 2 = 1

C[7] = 3 % 2 = 1

$\boxed{C = (1111\ 1111)}$

2. $H^T = \begin{pmatrix} 0111 \\ 1011 \\ 1101 \\ 1110 \\ 1000 \\ 0100 \\ 0010 \\ 0001 \end{pmatrix}$

a) $A \times H^T$ where $A = (1100\ 0111)$

D[0]: 1 %2 = 1

D[1]: 2 %2 = 0

D[2]: 3 %2 = 1

D[3]: 3 %2 = 1

$\boxed{D' = (1011)}$ The second bit was flipped

Flipping the second bit of A gives us

$A = (1000\ 0111)$ or $1110\ 0001_2$ as the original message

b) $A \times H$ where $A = (0001\ 1011)$

D[0] = 2 %2 = 0

D[1] = 1 %2 = 1

D[2] = 2 %2 = 0

D[3] = 1 %2 = 1

$\boxed{D = (0101)}$ There is an uncorrectable error

0101 is not identifiable in the H^T matrix

3.

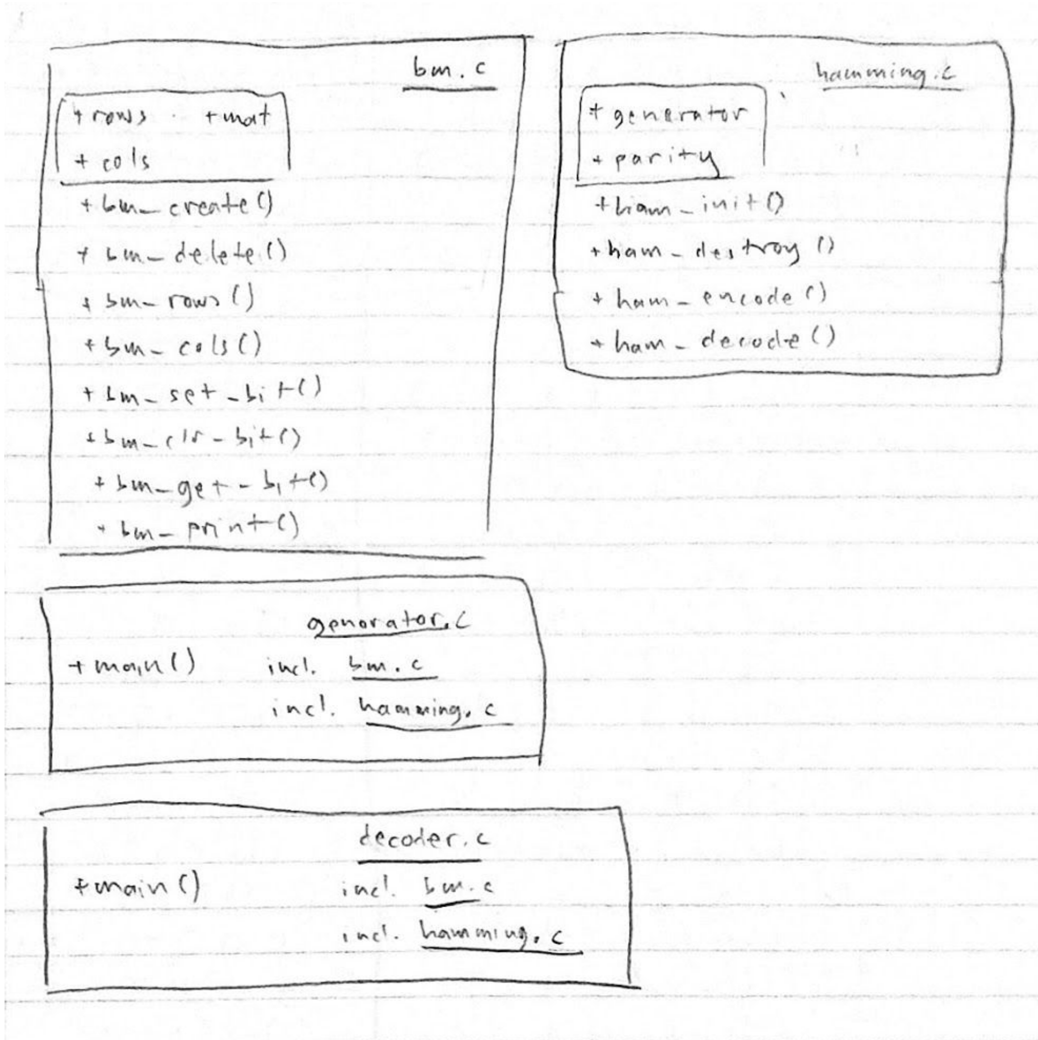
0	0	8	8
1	5	9	HAM_ERR
2	6	10	HAM_ERR
3	HAM_ERR	11	3
4	7	12	HAM_ERR
5	HAM_ERR	13	2
6	HAM_ERR	14	1
7	4	15	HAM_ERR

Purpose

The purpose of this assignment is to implement a Hamming code library that uses the Hamming(8,4) code. There are two programs for this assignment: one for generating Hamming codes and another for decoding them. The decoder will print statistics such as total bytes processed, uncorrected errors, corrected errors, and the error rate. There will be a program already provided that injects errors (noise) into the Hamming codes.

Structure/Layout

Here is a simplified sketch of the layout of the program:



Here are short descriptions of each program:

- `bm.c`: The implementation of the Bit Matrix ADT.
- `hamming.c`: The implementation of the Hamming Code module.
- `generator.c`: The implementation of the Hamming Code generator.
- `decoder.c`: The implementation of the Hamming code decoder.

Pseudocode

```
generator.c

int lower_nibble(int val):
    return val & 0xF

int upper_nibble(int val):
    return val >> 4

int main(int argc, char **argv):
    while there are options:
        set options

    ham_init()

    while not EOF:
        int byte = fgetc()

        int lonibble = lower_nibble(byte)
        int hinibble = upper_nibble(byte)

        int locode, hicode;

        ham_encode(lonibble, &locode)
        ham_encode(hinibble, &hicode)

        output locode and hicode w/ fputc()

    ham_destroy()
    close input and output files w/ fclose()

    return 0
```

```
decoder.c

int pack_byte(int upper, int lower):
    return (upper << 4) | (lower & 0xF)

int main(void):
```

```
    while there are options:
        set options

    ham_init()

    while not EOF:
        int locode = fgetc()
        int hicode = fgetc()

        int lonibble, hinibble;

        han_decode(locode, &lonibble)
        han_decode(hicode, &hinibble)

        count bytes, errors, and corrections

        int byte = pack_byte(hinibble, lonibble)

        fputc(byte)

    output total number of bytes, errors, and corrections to stderr

    ham_destroy()
    close input and output files w/ fclose()

    return 0
```

```
bm.c

struct BitMat:
    int rows
    int cols
    int **mat

int bytes (int bits):
    if bits % 8 == 0:
        return bits / 8
    return bits / 8 + 1

BitMat *bm_create (int rows, int cols):
    allocate memory for BitMat mat
    set mat.rows = rows and mat.cols = cols
```

```
    allocate memory for rows in mat
    for r in [0, rows):
        allocate bytes(cols) amount of memory for columns in mat[r]

void bm_delete (BitMat **m):
    free allocated memory in m
    free allocated memory for m

int bm_rows (BitMat *m):
    return mat.rows

int bm_cols (BitMat *m):
    return mat.cols

int byte_col (int c):
    int ans = bytes(c)

    // Account for zero indexing.
    if ans == 0:
        ans -= 1

   return ans

void bm_set_bit (BitMat *m, int r, int c):
    int index = c % 8
    int mask = 1 << index
    m.mat[r][byte_col(c)] &= | mask

void bm_clr_bit (BitMat *m, int r, int c):
    int index = c % 8
    int mask = ~(1 << index)
    m.mat[r][byte_col(c)] &= mask

int bm_get_bit (BitMat *m, int r, int c):
    int index = c % 8
    int mask = 1 << index

    int result = m.mat[r][byte_col(c)] & mask
    result = result >> index

    return result

void bm_print (BitMat *m):
```

```
    for r in [0, bm_rows(m)):
        for c in [0, bm_cols(m)):
            if(bm_get_bit(m, r, c) == 0):
                print 0
            else:
                print 1
        print newline
```

```
hamming.c

static BitMat *generator = null
static BitMat *parity = null

ham_rc ham_init(void):
    generator = bm_create(4, 8)

    for r in [0, bm_rows(generator)):
        bm_set_bit(generator, r, r)

    for r in [0, bm_rows(generator)):
        for c in [4, bm_cols(generator)):
            if c == 4 + r:
                continue
            bm_set_bit(generator, r, c)

    parity = bm_create(8, 4)

    for r in [0, bm_rows(parity) / 2):
        for c in [0, bm_cols(parity)):
            if c == r:
                continue
            bm_set_bit(parity, r, c)

    int c = 0;
    for r in [bm_rows(parity) / 2, bm_rows(parity)):
        bm_set_bit(parity, r, c)
        c += 1

    if failed to create generator and parity BitMats:
        return HAM_ERR
    return HAM_OK
```

```
void ham_destroy(void):
    bm_delete(generator)
    bm_delete(parity)

void i_to_bv (BitMat *m, int bits):
    for c in [0, bm_cols(m)):
        int index = bm_cols(m) - 1 - c
        int bit = bits >> index // shifts integer right until desired bit
is in the LSB

        bit &= 1 // masks bit except the LSB

        if bit == 1:
            bm_set_bit(m, 0, c)

void ham_vxm(BitMat *a, BitMat *b, BitMat *c):
    for i in [0, bm_cols(b)):
        int byte = bm_get_bit(c, 0, i)

        for j in [0, bm_rows(b)):
            byte += bm_get_bit(a, 0, j) * bm_get_bit(b, j, i)

        byte %= 2

        if byte == 1:
            bm_set_bit(c, 0, i)
        else:
            bm_clr_bit(c, 0, i)

int bv_binary (BitMat *v):
    int ans = 0

    for c in reverse of [0, bm_cols(v)):
        ans <<= 1

        // if bit at r, c is 1, set 1 at LSB of ans
        if bm_get_bit(v, 0, c) == 1:
            ans |= 1

    return ans

ham_rc ham_encode(int data, int *code):
```

```
    if data or code pointers are invalid:
        return HAM_ERR

    BitMat d = bm_create(1, 4)
    BitMat c = bm_create(1, 8)

    i_to_bv(d, data) // turns data to a bit vector d

    // performs multiplication of bit vector d and bit matrix generator and
stores result in bit vector c
    ham_vxm(d, generator, c)

    // reverses bit vector c and sets it to code
    code = bv_binary(c)

    // frees allocated memory for d an c
    bm_delete(d)
    bm_delete(c)

    if value in code is invalid:
        return HAM_ERR
    return HAM_OK

ham_rc ham_decode(int code, int *data):
    if code or data pointers are invalid:
        return HAM_ERR

    // lookup table for error checking
    int[] lookup = {0, 5, 6, -1, 7, -1, -1, 4, 8, -1, -1, 3, -1, 2, 1, -1}

    BitMat c = bm_create(1, 8)
    BitMat d = bm_create(1, 4)

    i_to_bv(c, code)) // turns code to a bit vector c

    // performs multiplication of bit vector c and bit matrix parity and
stores result in bit vector d
    ham_vxm(c, parity, d)

    // reverses bit vector d and sets it to data
    data = bv_binary(d)

    if lookup[data] == 0:
```

```
        // frees allocated memory for d an c
        bm_delete(d)
        bm_delete(c)

        return HAM_OK
    else if lookup[data] == -1
        // frees allocated memory for d an c
        bm_delete(d)
        bm_delete(c)

        return HAM_ERR

    int flipped_pos = lookup[data] - 1 // position of flipped bit

    if bm_get_bit(c, 0, flipped_pos) == 1:
        bm_clr_bit(c, 0, flipped_pos)
    else:
        bm_set_bit(c, 0, flipped_pos)

    // performs multiplication of corrected bit vector c and bit matrix
parity and stores result (original message) in bit vector d
    ham_vxm(c, parity, d)

    // reverses bit vector d and sets it to data
    data = bv_binary(d)

    // frees allocated memory for d an c
    bm_delete(d)
    bm_delete(c)

    return HAM_ERR_OK
```

Design Process

1. I read the Assignment 4 specifications (`asgn4.pdf`) to get a good idea of the subject matter and my task. I also referred to the supplemental readings outlined at the end of the specifications.
2. Before starting on the lab, I did the prelab.
3. I then laid out the program in a high level diagram before writing my pseudocode.
4. With help from the `asgn4.pdf` and recorded sections, I wrote my pseudocode.
    a. I made helper functions to improve readability of `ham_encode()` and `ham_decode()`

i. I first made `i_to_bv()` in `hamming.c` that converts an `int` to a `BitMat` structure that only has one row (a.k.a. a bit vector).

ii. I also made another helper function in `bm.c` called `bm_get_reversed_row()` that returns the whole reversed byte located at a specified row.

iii. After realizing that I used the same instructions to perform matrix multiplication, I made it into a helper function in `hamming.c` called `ham_vxm()` which stores the result of multiplying a bit vector and bit matrix into another bit vector.

5. I started setting up the `Makefile`, `README.md`, `*.c`, and `*.h` files, making sure everything compiles fine before starting to program.
6. After everything compiled fine, I began writing my `generator.c` and `decoder.c` programs.
    a. At this point, my error handling is preliminary.
    b. I had some issues with my `Makefile`, but was able to fix it by compiling `bm.c` and `hamming.c` when running `gen` and `dec`.
7. Afterwards, I began implementing `bm.c`.
    a. First, I implemented `bm_print()` for debugging purposes.
    b. I then realized my implementation of `bm_print()` used `bm_rows()`, `bm_cols()`, and `bm_get_bit()`, so I implemented those, too.
    c. Then I began implementing `bm_create()` and `bm_delete()`. I created a helper function `bytes()` to calculate the number of bytes to allocate for each row of a `BitMat`'s `mat`.
        i. I tested `bm_delete()` by simply creating and deleting a `BitMatrix` in `gen.c` and running `valgrind`.
    d. I realized my implementation of `bm_get_bit()` didn't take into account when the bit of interest lies outside of the first byte in a `BitMat`'s row. I fixed this by calculating the position of the byte that contains the bit of interest. With this, I can mask the correct byte.
        i. I later made this a helper function called `byte_col()` which will help me implement `bm_set_bit()` and `bm_clr_bit()`.
    e. After I was able to print `BitMat`s of varying sizes, I began implementing `bm_set_bit()` and `bm_clr_bit()` functions.
8. Then, I started implementing `hamming.c`.
    a. I started with `ham_init()` and finished with little problems.
    b. I then implemented `ham_destroy()` and tested it in `gen.c` using `valgrind`.
    c. Next, I tackled `ham_encode()` and implemented the aforementioned helper functions.
        i. The first error I encountered was in `bm_clr_bit()` where I used a '`!`' operator to invert the mask instead of the '~' operator.
        ii. The next error I faced had to do with my helper function, `i_to_bv()`, where I failed to reverse the binary representation when inputting the bits into the bit vector.

   iii. I then tackled a runtime error: I was returning `HAM_ERR` when `data` and `code` was `0` when only the `code` pointer needed to be checked.
   iv. Also, my `code` pointer was not pointing to a new value at the end of the function. This was because I was changing where the pointer was pointing instead of changing the value.
d. Finally, with most of my bugs ironed out, I started implementing `ham_decode()` with more confidence.
   i. My first implementation was changing `data` to the index of my lookup table instead of the original message.
   ii. In the end, I couldn't find out why my program couldn't detect the errors.