

Rahul Vaidun
rvaidun@ucsc.edu
May 8 2021

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

I. Description

This assignment is for encoding and decoding hamming codes. To easily do this the encoders and decoders are using Bit Vectors and Bit Matrices.

II. Prelab

1.	0	HAM CORRECT
	1	4
	2	5
	3	ERR
	4	6
	5	ERR
	6	ERR
	7	3
	8	7
	9	ERR
	10	ERR
	11	2
	12	ERR
	13	1
	14	0
	15	ERR

2.

a. $1110\ 0011_2$

The error syndrome is $\tilde{x}H^T = 1011 \rightarrow 1101_2 \rightarrow 13_{10}$
table[13] = 1 Refer to lookup table. Be

A value of 1 means the first byte needs to be flipped.

b. $1101\ 1000_2$

error syndrome = $(0, 1, 0, 1) \rightarrow 1010_2 \rightarrow 10_{10}$
table[10] = ERR

This means the error is uncorrectable

III. Pseudocode

In order to encode and decode the hamming code we will need to implement Bit Vectors and Bit Matrices. The Pseudocode implementation can be seen below

A. BitVector

```
class BitVector:
    def __init__(self, length):
        self.length = length
        self.vector = [0] * length

    def bv_length(self):
        return self.length

    def bv_set_bit(self, i):
        bytupos = i // 8
        bitpos = i % 8
        self.vector[bytupos] = ((1 << bitpos) |
self.vector[bytupos])

    def bv_clr_bit(self, i):
        bytupos = i // 8
        bitpos = i % 8
        self.vector[bytupos] = (self.vector[bytupos] & ~(1 <<
(bitpos)))

    def bv_get_bit(self, i):
        bytupos = i // 8
        bitpos = i % 8
        return (self.vector[bytupos] >> bitpos) & 1

    def bv_xor_bit(self, i, bit):
        bytupos = i // 8
        bitpos = i % 8
```

```

        b = self.bv_get_bit(i)
        # clear bit and then set it to either 0 or 1
        self.vector[bytepos] = (self.vector[bytepos] & (
            ~(1 << bitpos))) | ((b ^ bit) << bitpos)

def bv_print(self):
    print(self.vector)

```

B. BitMatrix

```

class BitMatrix():
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.BitVector = BitVector(rows * cols)

    def bm_rows(self):
        return self.rows

    def bm_cols(self):
        return self.cols

    def bm_set_bit(self, r, c):
        self.BitVector.bv_set_bit(r * self.cols + c)

    def bm_clr_bit(self, r, c):
        self.BitVector.bv_clr_bit(r * self.cols + c)

    def bm_get_bit(self, r, c):
        self.BitVector.bv_get_bit(r * self.cols + c)

    def bm_from_data(self, byte, length):
        bm = BitMatrix(1, length)

```

```

        for b, i in enumerate(bits(byte)): # iterate thorough all
the bits

            if b == 1:

                bm.bm_set_bit(1, i)

        return bm

def bm_to_data(self, length):
    x = 0
    for i in range(length):
        if self.BitVector.bv_get_bit(i) == 1:
            x |= 1 << i
        else:
            x &= ~(1 << i)
    return x

def bm_multiply(self, B):
    bm = BitMatrix(self.rows, B.cols)
    for k in range(self.cols):
        for i, j in range(self.rows), range(B.cols):
            A = self.bm_get_bit(i, k)
            B = self.bm_get_bit(k, j)
            res = A ^ B
            if res == 1:
                bm.bm_set_bit(i, j)

def bm_print(self):
    print(self.BitVector)

```

C. Hamming module

The Hamming encode is very simple. Just take the Generator matrix that is passed and multiply with the code using `bm_from_data`. Then return the result of the multiplication through a matrix with `bm_to_data`

```
ham_encode(BitMatrix G, uint8_t msg) {  
    BitMatrix m = bm_from_data(msg,4) only length of 4 since the msg is a nibble  
    BitMatrix code = bm_multiply(m,G)  
    return bm_to_data(code);  
}
```

Ham Decode is a little harder but still relatively simple. Construct a lookup table with the errors and bytes to flip. This makes it easy to figure out if there is a byte that needs to be flipped

```
ham_decode(BitMatrix ht, uint8_t code, uint8_t msg) {  
    Table = [CORRECT,4,5,ERR,6,ERR,ERR,3,7,ERR,ERR,1,0,ERR]  
    bm = bm_from_data(code,8)  
    Error_syndrome = bm_multiply(bm, Ht)  
    Es = bm_to_data(error_syndrome)  
    If (es == 0) {  
        Return ham_ok and set msg  
    }  
    Else if table[es] equals ham_err {  
        Return ham_err  
    }  
    If bm_get_bit(bm, 0, table[es]) then clear the bit. Else set the bit  
    Msg = corrected code  
    Return ham_correct  
}
```

D. Encode

The Encoder will read nibbles from a file and encode byte by byte. Pseudocode is shown below.

```
main() {  
    In_fp = stdin  
    Out_fp = stdout  
    Struct stat statbuf  
    encode_table[16]  
    Parse command line arguments for input file and output file  
    Change file permissions with fstat and fchmod  
    BitMatrix *G = bm_create(4,8)  
    Set the bits in this bit matrix to match the Generator matrix for 8,4 hamming code  
}
```

```

        For (i = 0; i < 16; i++) {
            encode_table[i] = ham_encode(G,i)
        }
    While (fgetc is not End of File) {
        fputc(encode_table[lower_nibble(byte)],out_fp)
        fputc(encode_table[upper_nibble(byte)],out_fp)
    }

```

E. Decode

Decode functions similarly to encode but lookup table is constructed differently

```
define STATS_INDEX 4
```

```
Define STATS_TOTAL 0
```

```
Define STATS_CORRECTED 3
```

```
Define STATS_ERR 2
```

```

main() {
    Verbose = false
    Int Inc, unc (Lower nibble code and upper nibble code)
    uint8_t lnm unlm (Lower/Upper nibble message)
    Ham_status hs;
    Uint8_t decode_table[256]
    Uint8_t status_table[256]
    Uint32_t stats[stats_index] = 0 0 0 0
    In_fp = stdin
    Out_fp = stdout
    Struct stat statbuf
    Parse command line arguments for input file and output file
    Change file permissions with fstat and fchmod
    BitMatrix *G = bm_create(8,4)
    Set the bits in this bit matrix to match the Generator matrix for 8,4 hamming code
    For (i = 0; i < 256; i++) {
        Hs = ham_decode(Ht,i,&lnm)
        Status_table[i] = hs
        Decode_table[i] = lnm
    }
    Lnm = 0
    While (fgetc for 2 bytes is not End of File) {
        stats[STATS_TOTAL] += 2
        If status_table[first_byte] is HAM_CORRECT or HAM_OK {
            Lnm = Decode_table[first_byte]
        }
        Stats[status_table[first_byte] + STATS_INDEX]++;
    }
}

```

```
        Repeat for the second byte
        fputc(pack_byte(second_byte,first_byte), out_fp);
    }
    If (verbose){
        Print the stats
    }
}
```