Specification

Assignment 5. Hamming codes

Files needed:

encode.c: this is where our encoding is taking place. One of 2 main files

decode.c: this is where we decoding is taking place. The other main file

error.c: an unmodified file given to us which causes randomly generated errors. Some errors are unfixable though based on the matrices G and matrix H^T.

entropy.c: this will allow us to get information based on what we have implemented which will be a tool used for out writeup this week

bv.h: header file for the bit vector we have to implement

bv.c: bit vector is how we are going to be constructing the bit matrix, by take a floor divided by 8 + 1 of a uint8_t to diminish the size.

bm.h: header file for the bit matrix

bm.c: big cse 12 flashbacks, but we will be constructing a bitmap based on how many bit vectors are constructed which will allow us to find a particular bit to be xored or what not. Will provide more details after my pseudocode.

hamming.h: header file for the hamming codes

hamming.c: this is where all the magic happens for the week. The two main functions will be held here

Makefile: a make file to easily run the program by prompting `make` into the Linux terminal on our Ubuntu virtual machine.

README.md: describes how the program works and runs in addition to how to use the makefile

WRITEUP.pdf: As we had in assignments 2 and 3 we have yet again a writeup where we have to show some level of thought on the problem and explain how and why we tested our program the way we did. Lucky for us this week we have a supplemental entropy.c file which will help us in testing.

Program Description: Hamming codes is an Error correction algorithm invented by Richard Hamming. Because errors can be caused thanks to "noise" as described in the assignment document, we need to try to see if we can detect an error that might have occurred and see whether its fixable or not. In our case our errors will only be fixable at specific set of numbers, and numbers can range from 0x0-0xF if we are talking in terms of hex or simply 0-15 in binary. We can fix up to 8 of these errors thanks to the Matrices G and H which we will discuss in a bit, but we are essentially just correcting potential errors that may occur. According to the assignment document this is a common thing with reading dvds, cds, etc because scratches on the disc could be problematic but these error correction algorithms can try to fix these buffers if it is possible.

encode.c

```
Helper functions
  Lower nibble(){
Gets lower nibble
}
```

```
  Upper nibble(){
Gets upper nibble
}
main()
optarg
   Switch h
     Print helper message
   Switch i
     Reads to specified binary infile
   Switch o
     Writes to specified binary outfile

Construct G matrix
   While (file != EOF){
     Lower = lower_nibble()
     Upper = upper_nibble()
     ham_encode(lower)
     Put ham encode in outfile
     ham_encode(upper)
     Put ham encode in outfile
}
```

This module is one of the main files needed to run the program. In this file we have our typical optarg which will allow us to have a helper message, read a specified binary infile and outfile. We also give file permissions to the infile so the outfile can attribute these permissions. Then we construct the g matrix which is a constant. Then we begin reading the file with fgetc which only reads a byte at a time but in our case we need to split the it up into two separate nibbles because ham_encode takes a nibble and generates a byte. Once the hamming codes are generated we put them into stdout or a specified outfile.

Decode.c

```
Helper function
  Pack_byte(){
  Which packs a byte
}
main()
optarg
   Switch h
     Print helper message
   Switch i
     Reads to specified binary infile
   Switch o
     Writes to specified binary outfile

Construct Ht matrix
   While (file != EOF){
     If counter % 2 = 1
```

```
      Msg1 Ham_decode
      Continue
     If counter % 2 = 0
      Msg2 Ham_decode
     If counter % 2 = 0
      Packbyte(msg1,msg2)
      fputc(out)
```

Decoding uses the code provided from the encoder which was only supplied the message to see if there was an error at a specific bit. The way we can know where the error occurred is based on the matrix multiplication where we will receive a 1x4 matrix. When we compare whats on the 1x4 matrix with whats on the transpose matrix we can see if at a specific bit there was an error that occurred and if it is a value on the lookup table then it will correct it. If it isnt on the transpose matrix it will be a Ham_Err. The table will be provided at the bottom of the DESIGN doc and be present in the hamming.c file

Bv.c

```
struct  BitVector {
      uint32_t  length; //  Length  in bits.
      uint8_t *vector; // Array of bytes.
};
BitVector *bv_create(uint32_t length)
 Create ADT by allocating memory
 Need to calloc the vector bc they need to all be zeros if not replac
}

Void bv_delete(BitVector **v){
  Delete all bit vectors
}

Uint32_t bv_length(bitVector *v){
Return length
}

Void bv_set_bit(BitVector *v, uint32_t i)
 Remember how we calloced. We set specific bit to 1's
}

void_bv_clr_bit(BitVector *v, uint32_t i){
 Clears if there is a 1 in the ith bit in bit vector
}

Void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit){
 Xors using a modulus 2 of a specified bit to check for parity
}

Void bv_print(Bitvector *v){
```

```
 Will print bit vector. Debugging purposes
}
```

The bit vector in this assignment will help us construct the bit matrix and allow us to access certain indexes from a specific row and column so it can be encoded and decoded. As this module is used to construct the basis of everything. As in most ADT we have the create and delete function which is standard process by now, and then we have the length which returns the length of the bit vector which when making the bit vectors will have to be divided by 8 and + 1 if there is a remainder. They gave us an equation that looked like this n/ 8 + 1 `uint8_t` s. We will also need to set a bit at specific locations, clear a bit at specific locations, etc. Lastly we need to use xor as part of the program to find the parity bits.

Bm.c

```
struct  BitMatrix {
      uint32_t  rows;
      uint32_t  cols;
      BitVector *vector;
};

BitMatrix *bm_create(uint32_t rows, uint32_t cols){
 Same gist. We create everything per usual. Calloc the vector
}
void bm_delete(BitMatrix **m){
Delete per usual
}
uint32_t bm_rows(BitMatrix *m){
Return rows
}
uint32_t bm_cols(BitMatrix *m){
Return columns
}
void bm_set_bit(BitMatrix *m, uint32_t r, uint32_t c){
r * n + c will let you set a bit at a specific part in the bit matrix
}
void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t c){
Does opposite of bm_set_bit
}
uint8_t bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c){
Gets access to the bit at a specific spot
}
uint8_t bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c){
}
uint8_t bm_to_data(BitMatrix *m){
}
BitMatrix *bm_multiply(BitMatrix *A, BitMatrix *B){
```

```
Performing bit multiplication with A = test bit B= G and H^T matrices
}
void bm_print(BitMatrix *m){
Debug print statement
}
```

Bit matrix as previously mentioned is a matrix of all the bit vectors we made in the previous ADT. In this one we will create and delete per usual, we will set, clear, get bit per usual, and then we have bit matrix multiplication that performs matrix multiplication so we can get a code for encoding and decode a 4 bit error message for decoding.

Hamming.c

```
typedef  enum  HAM_STATUS {
      HAM_OK       =  -3,   // No  error  detected.
      HAM_ERR      = -2,   //  Uncorrectable.
      HAM_CORRECT = -1    //  Detected  error  and  corrected.
} HAM_STATUS;

Lookup = {lookup table}

uint8_t ham_encode(BitMatrix *G, uint8_t msg)
 From data
 Bm_multiply
 To data
HAM_STATUS ham_decode(BitMatrix *Ht, uint8_t code, uint8_t *msg)
 From data
 Bm_multiply
 To data
 Compare if to data is a ham ok, ham err, ham, correct
  If ham ok
    msg=nibble
    Return ham ok
  If ham err
    Return ham err
  If ham correct
    If msg[index] = 1
      Clr bit
    If msg[index] = 0
      Set bit
  Msg = nibble
  Return ham correct
```

This file will call the hamming codes. The encode and decode will take place here. In encode we simply call three functions and that's all there is to it. The multiplication takes place here. Then we have decode which does the same thing but instead has to see if the error syndrome matches any of the values from the lookup table. If it does then you know where to fix the error but if you don't then you just return ham_err. Refer to the prelab questions for a better understanding of the lookup table.

Pre lab Questions:

1. Look up table:

0 |HAM_OK
1 |4
2 |5
3 | HAMM_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

2a) 1110 0011.

After performing matrix multiplication with the H^T matrix we get (4,3,2,2) and if me % 2 we get (0,1,0,0). So there is an error at the 6th bit so that will be corrected

2b) 1101 1000

After performing matrix multiplication with the H^T matrix we get (2,3,2,3) and if me %2 we get (0,1,0,1). This results in a HAM_ERR so the bit is unfixable