Hersh Rudrawal
hrudrawa@ucsc.edu
5/4/2021

CSE13s Spring 2021
Assignment 5-Hamming Codes

PRE-LAB

1.Look up table

Convert decimal numbers to Hamming code and find the code's position in H^T

0=(0 0 0 0) no errors
1=$(0,0,0,1)_2$= (1 0 0 0) row 4
2=$(0,0,1,0)_2$= (0 1 0 0) row 5
3=$(0,0,1,1)_2$= (1 1 0 0) none
4=$(0,1,0,0)_2$=(0 0 1 0) row 6
5=(1 0 1 0)
6= (0 1 1 0)
7=(1 1 1 0)
8=(0 0 0 1)
9= (1 0 0 1)
10=(0 1 0 1)
11=(1 1 0 1)
12=(0 0 1 1)
13= (1 0 1 1)
14=(0 1 1 1)
15=(1 1 1 1)

H^T

Row	
0	0 1 1 1
1	1 0 1 1
2	1 1 0 1
3	1 1 1 0
4	1 0 0 0
5	0 1 0 0
6	0 0 1 0
7	0 0 0 1

Lookup table

num	Err pos
0	HAM_OK
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

2.

a)1110 0011_2

First convert the code to Hamming code by reversing the order- (1 1 0 0 0 1 1 1)

Next find the error syndrome - matrix multiplication of the Hamming code and H^T matrix

e=(1 1 0 0 0 1 1 1) * H^T % 2

=(0 + 1 + 0+ 0+ 0 +0 +0 +0), (1 + 0 + 0+ 0+ 0 +1 +0 +0), (1 + 1 + 0+ 0+ 0 +0 +1 +0), (1 + 1 + 0+ 0+ 0 +0 +0 +1)

=(1 2 3 3)%2

=(1 0 1 1)= 13_{10}

Once we calculate e, we can check our lookup table to see what position contains the error bit

Then we can flip the bit in that position it to correct it

Error in second element

b)1101 1000_2= (0 0 0 1 1 0 1 1)

(0 0 0 1 1 0 1 1) * H^T

=(0 + 0 + 1+ 1+ 0 +0 +0 +0) (0 + 0 + 0+ 1+ 0 +0 +0 +0) (0 + 0 + 0+ 1+ 0 +0 +1 +0) (0 + 0 + 0+ 0+ 0 +0 +0 +1)

=(2 1 2 1)%2

=(0 1 0 1)=10_{10}

10 in the lookup table means the error is Uncorrectable

DESCRIPTION

In this assignment we will be creating a program that can encode data from a file into Hamming code(8,4) and decode it as well. The program will print the following statistics- total bytes processed, uncorrected errors, corrected errors, and the error rate

TOP LEVEL DESIGN

Bit vector

This program will create a bit vector. The parameter “length” is the number of bits, and the array “vector” will contain blocks of 8 bits(byte). The number of bytes to allocate - length/9+1.

bv.c

//Inspired from eugene’s section

Struct BitVector

    uint32 length
    uint8 *vector

bv_create(length)

    Create a new BitVector by allocating memory using malloc
    If we could not allocate memory for the Bitvector then return a null pointer
    Set the Bitvector length to the length parameter
    Create the vector array using calloc(length/9+1,size of uint8)
    If vector could not be created, free the BitVector and return a null pointer

bv_delete(Bitvector)
    Free the vector array in the BitVector
    Free the BitVector
    Set pointer to Null

bv_length(Bitvector)
    Return length variable of the BitVector

bv_set_bit(Bitvector, uint i)
    Store the block/byte the bit is in- i/8
    Store position of bit in the byte- i%8
    Conduct a left shift by x(position) on an unsigned int 1
    Set the byte to- OR operation on the byte and the shifted value

bv_clr_bit(Bitvector, i)
    Find byte, position and shift 1 by position
    Invert the shifted value
    Set the byte to- AND operation on the byte and the shifted value

bv_get_bit(Bitvector, i)
    Find byte, position and shift 1 by position
    Get and store the byte in a variable- b
    Set b to- AND operation on b and the shifted value
    Right shift b by the position
    Return b

bv_xor_bit(Bitvector, i, bit)
    Find byte, position
    Shift the bit parameter by position
    Set the byte to- XOR operation on the byte and the shifted value

bv_print(Bitvector)
    Create for loop (i=0;i<length;i++)
        Print the value of bv_get_bit(Bitvector, i)

Bit Matrix

This program will create a 1d array of Bitvectors, but will treat it as a 2d array with rows and columns. We can get the location of a specific bit in the matrix by using this formula- r(row of desired bit)*n(num of rows)+c(column of desired bit)

bm.c

```
Struct BitMatrix
    Uint32 rows
    Uint32 cols
    Bitvector *vector
*bm_create(rows,cols)
    Use malloc to allocate memory for BitMatrix
    If malloc failed, return null pointer
    Set the rows and cols variable in the BitMatrix to the parameters
    Use bv_create to create *vector of length(rows*cols)
    Return pointer to Bitmatrix

bm_delete(BitMatrix)
    Free *vector
    Free Bitmatrix
    Set pointer to Null
bm_rows(BitMatrix)
    Return var rows in BitMatrix
bm_cols(BitMatrix)
    Return var cols in BitMatrix
bm_set_bit(BitMatrix, rows, cols)
    Location of bit= rows*num of rows+cols
    Use bv_set_bit function with parameters(vector,location)
bm_clr_bit(Bitmatrix, rows, cols)
    Find location
    Use bv_clr_bit()
bm_get_bit(BitMatrix, rows, cols)
    Find location
    Use bv_get_bit() and return the bit

//Convert byte to bit matrix
*bm_from_data(byte,length)
    Create a new bit matrix rows=1, cols=length,
    If length>8 return Null
    Create for loop(i=0;i<length;i++)
        Conduct left shift by i on uint 1
        Get and store the byte in a variable- b
        Set b to- AND operation on b and the shifted value
        Right shift b by i
        Call bm_clr_bit with rows=0,cols=i
```

```
        If b equals 1 then us bm_set_bit with same parameters
    Return pointer to M, return Null pointer if M could not be created

bm_to_data(BitMatrix)
    Create variable “x” to store data
    Use for loops to iterate through the BitMatrix’s rows- i and cols- j
        Get the bit at (i,j)
        Shift bit to appropriate position and add value to x
    Return x

*bm_multiply(BitMatrix A, BitMatrix B)
    Create a new matrix “C” where rows equals the rows in A, and cols equals the cols in B
    Create uint8 x
    Created nested for loop to iterate through C
    For i=0;i< C rows
        For j=0;j< C cols
            Set x to 0
            For k=0;k < A cols
                Using bm_get_bit
                Get the bit in A at(i,k), and the bit in B at(k,j)
                add to x the value of the AND operation on the bit in A and B
            Mod x by 2
            If x is 1, set the bit at position (i,j) in C
    Return pointer to C

bm_print(BitMatrix)
    Create nested loop to iterate through the Bitmatrix
        For i<rows
            For j < BitMatrix cols
                Print bm_get_bit(BitMatrix,i,j)
            Print new line
```

Hamming
The implementation of the Hamming code module. Contains function to encode and decode Hamming code.
hamming.c
Typedef enum HAM_STATUS //provided on lab document
    HAM_OK=-3
    HAM_OK=-2
    HAM_OK=-1

Create an array that matches the lookup table in the prelab

ham_encode(BitMatrix G, msg)
    //G is the generator matrix
    Create a matrix “M” using bm_from_data(msg, 4)
    Multiply the new Bitmatrix M by Bitmatrix G, store in a BitMatrix C
    Convert C to data and return its value

ham_decode(BitMatrix Ht, code, msg)
    //Ht is the transpose of the parity-checker matrix
    Create a bit matrix “c” out of the code, row= 1, cols=8
    Multiply the c by Ht and store the result in a BitMatrix “e”
    Convert e to data
    Then use the lookup table(in prelab) to fix any fixable errors in c and return the appropriate enum value.

encode.c
Our encoder program will read a file, and convert each byte in the file to Hamming code. This program takes the following options: -h help, -i infile(path of file containing data) and -o outfile(file where to write the Hamming Code). By default the infile and outfile are stdin and stdout. To get the option we can use getopt and a switch statement.

The implementation of the following functions are provided on the lab document*
*lower_nibble(val){
    Return value & 0xF
}
*upper_nibble(val){
    Return val > 4
}

main(argc,argv)
    Set FILE infile to stdin
    Set FILE outfile to stdout
    Create int int opt, set to 0

While getopt does not return -1
Create Switch opt
Case -h
Print help message
Case -i
Set infile to new file
Case -o
Set outfile to new file
Default
Return error msg

To encode the data we need to create the generator matrix, then we need to read the data. Since we are using Hamming(8,4) code, we break up the data into nibbles using the provided functions. Then we use ham_encode to encode each nibble.
Open infile
Open outfile
Create generator matrix G
While there are still bytes to read from file
Get and store the data in uint8
Get the lower nibble of the data using lower_nibble() function
Get the upper nibble of the data upper_nibble() function
Encode lower nibble using ham_encode(G,lower nibble)
Encode upper nibble using ham_decode(G,upper nibble)
Use fputc to print both codes to outfile
Close files
Delete generator matrix

decoder.c
This program will read Hamming code and decode it returning the msg.This program takes the following options: -h help, -i infile(path of file containing the code), -o outfile(file where print the decode message) and -v (prints the statistics to stderr). By default the infile and outfile are stdin and stdout. To get the option we can use getopt and a switch statement.

The implementation of the following functions are provided on the lab document*
*pack_byte()

main(argc,argv)
Set FILE infile to stdin
Set FILE outfile to stdout
Create a bool stats, set to false //check if we need to print out stats
Create 4 variables to store statistics - byte processed, corrected errors, uncorrected errors, error rate
Create int int opt, set to 0

While getopt does not return -1
    Create Switch opt
    Case -h
        Print help message
    Case -i
        Set infile to new file
    Case -o
        Set outfile to new file
    Case -v
        Set stats to true
    Default
        Return error msg

Open infile
Open outfile
Create the H^T matrix using bm_create
While there are still bytes to read from file
    We need to get 2 bytes at a time- 1 for lower nibble and the other form upper nibble.
    Decode the first code
    Decode the second code
    Update statistic variables
    Pack the 2 nibbles using pack_byte()
    Use fputc to print decode msg to outfile

print statistics if required
Close files
Delete H^T