Ruchit Patel

CSE13S, Spring 2021

Prof. Darrell Long

1 May 2021

Assignment 5: Hamming Codes

Brief Description

- In this assignment, an extended version of the Hamming(7, 4)--Hamming(8, 4)--error correction code (ECC) is implemented.
- The code itself is systematic and an even parity checking scheme is used.
- The lab has two main components: encoding and decoding.
- Arguments:
    - Encoder and Decoder:

      -h (prints the help message),

      -i (specifies the input file),

      -o (specifies output file).
    - Unique to Decoder:
        - v (prints decoder statistics to stderr)

Pre-lab Questions

1. Completed lookup table (Please see Decoding for more details)

Array Index	Bit position
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

2. Decoding the codes
   a. $1110\ 0011_2$
      - The matrix for the code would be [1 1 0 0 0 1 1 1] * H^T (mod 2) gives us **e** = [1 0 1 1] which in binary is 1101_2 and in decimal would be 13. Thus, the entry in the lookup tables (same as checking transpose rows) tells us that the first bit is incorrect (counting from zero).
      - The correct code now would be [1 0 0 0 0 1 1 1]. The data bits (first four) hence would be [1 0 0 0] or 0001_2.
   b. $1101\ 1000_2$
      - The matrix for the code would be [0 0 0 1 1 0 1 1] * H^T (mod 2) gives us **e** = [0 1 0 1] which in binary is 1010_2 and in decimal would be 10. Thus, the entry in the lookup tables (same as checking transpose rows) tells us that there is no such entry and thus more than one error occurred.
      - Hence, the code could not be corrected.

Descriptions/Implementation

Argument Parsing

- Arguments are parsed using getopt GNU utility.
- If -i argument is encountered, the file name pointed by optarg is opened. Error handling is also done there (see Error Handling for more details). The handling is similar for -o flag.
- If -v is encountered, inside the case for -v, an integer value is set (no Set since only one argument is set for later).

Pseudocode:

```
define flags
while (getopt returns valid)
    switch to the argument case if valid
        function according to the argument
    if invalid print usage message and return
```

BitVectors (BV): Representing Bits

- The BitVector data struct is an array of integers. However, each of those integer's bit has significance.
- Instead of using a whole integer to represent true (1) or false (0), each bit of the integer array is used to be space efficient.

- Structure/Members of a BitVector

a.	Length of vector array	:	Total length of vector in bits.
b.	Array of bytes	:	Array that holds bytes.

- Methods associated with a BitVector (Pseudocode)
  a. bit vector pointer bv_create (length of vector)

```
assign memory for BV structure
set the length element to $length$
assign memory for the array equal to size of integer * $\lfloor length/8 \rfloor + 1$(minimum elements
for *length* bits) [credits: from the lab documentation]
if cannot be assigned then return
else initialize each integer to zero (either memset or loop)
return pointer to newly created BV
```

b. bv_delete (pointer to BV pointer)

```
if BV is not empty/null
        free the memory for BV array
        free the memory for BV structure
        set BV pointer to null
```

c. integer bv_length (BV pointer)

```
return the length element of BV struct pointed by the pointer
```

d. bv_set_bit (BV pointer, integer index) [CREDITS: From the lab documentation]

```
locate the byte for index (index / 8)
locate the bit inside that byte (index % 8)
set the bit using bitwise operation (byte ORed with 1 left shifted by bit location)
```

e. bv_clr_bit (BV pointer, integer index)

```
locate the byte for index (index / 8)
locate the bit inside that byte (index % 8)
clear the bit using bitwise operation (byte ANDed with NOT of  1 left shifted by bit location)
```

f. integer bv_get_bit (BV pointer, integer index)

```
locate the byte for index (index / 8)
locate the bit inside that byte (index % 8)
return the bit using bitwise operation (byte ANDed with  1 left shifted by bit location)
```

g. bv_xor_bit (BV pointer, integer index, integer bit)

```
locate the byte for *index* (*index* / 8)
locate the bit inside that byte (*index* % 8)
set the bit using bitwise operation (*bit* XORed with 1 left shifted by bit
location)
```

h. bv_print (BV pointer)

```
loop over the vector array inside the bit vector length times
        each time call bv_get_bit and print accordingly
```

BitMatrix (BM)

- The BitMatrix data type is, as the name suggests, a matrix of bits.
- The matrix's data is stored in a BitVector of length row times columns of the matrix (to store each element of matrix)

- Structure/Members of a BitMatrix

a. Number of Rows	:	Total number of rows for the matrix
b. Number of Columns	:	Total number of cols for the matrix
c. A BitVector	:	A BitVector to store all elements

- Methods associated with a BitMatrix (Pseudocode)
    a. BM pointer bm_create (rows, columns)

```
assign memory for BM structure
create a BitVector of length row times columns by calling its constructor
if cannot be assigned then return
return pointer to newly created BM
```

b. bm_delete (pointer to BM pointer)

```
if BM is not empty/null
        delete the BV using bv_delete
        free the memory for BM structure
        set BM pointer to null
```

c. integer bm_rows (BM pointer)

return the *rows* element of BM struct pointed by the pointer

d. integer bm_rows (BM pointer)

return the *columns* element of BM struct pointed by the pointer

e. bm_set_bit (BM pointer, row, column) [CREDITS: From the lab document]

set the bit using bv_set_bit where index is given by row value * total columns + column value

f. bm_clr_bit (BM pointer, row, column)

clear the bit using bv_clr_bit where index is given by row value * total columns + column value

g. integer bm_get_bit (BM pointer, row, column)

get the bit using bv_get_bit where index is given by row value * total columns + column value

return the bit

h. BM pointer bm_from_data (integer byte, integer length)

```
create a BitMatrix using bm_create where rows = 1 and columns = length (max 8)
loop for length - 1 times
    each time get the nth bit from the byte using bitwise operations (1 ANDed with byte
    << current iterator value)
    set the bit [if bit at n of byte = 1] using bm_set_bit where row = 1 and column = n
    (loop counter)
```

i. integer/byte bm_to_data (BM pointer)

```
create a temporary value to hold the final result
loop starting at length of BM's BitVector
    each time if the bit at iterator -1 is 1 make the (iterator - 1)th bit of temp = 1 (bit op)
```

j. BM pointer bm_multiply (BM pointer A, BM pointer B)

```
check input matrix constraints (A should be in form mxn and B should be in the form nxp)
create a mxp matrix (let's call it C)
perform matrix multiplication:
    loop over mxp  times: [iterator i]
        loop over n times: [iterator j]
            get A's bit at [row, col] = [total/p, j]  (row would always be total/p)
            get B's bit at [row, col] = [j, total%p] (col would always be total%p)
            multiply (logical AND) the above bits
            add (XOR) the result with c's BitVector where index = i
```

k. bm_print (BM pointer)

```
for each row in the matrix:
    for each column in the matrix:
        call bm_get_bit where row = row, column = column
        print the returned bit
    print newline
```

Hamming Code

- The Hamming code used for this lab--Hamming(8, 4)--is a systematic and linear ECC.
- For example, a nibble of data (4 bytes) would be encoded where 4 bits are parity bits and 4 bits are data bits. Example of an encoded byte:

P_3	P_2	P_1	P_0	D_3	D_2	D_1	D_0

where P_x are parity bits and D_x are data bits.
- Here,

  P_0 is a parity bit to check even parity for D_0, D_1, and D_3,

  P_1 is a parity bit to check even parity for D_0, D_2, and D_3,

  P_2 is a parity bit to check even parity for D_1, D_2, and D_3,

  and P_3 serves as a parity check for the whole byte.

Encoding

- Since the parity check is even, a XOR or (mod 2) operation is used.
- For this lab, to create/calculate the parity bits, a generator matrix is used.
- Since, the input is a nibble (1x4 matrix) with 4 data bits and the output is a byte (1x8 matrix) with 4 additional parity bits, the generator matrix G is a 4x8 matrix.
- Thus, the generator matrix such that [D_3 D_2 D_1 D_0] * G mod 2 = [D_3 D_2 D_1 D_0 P_3 P_2 P_1 P_0] is given by the matrix G = (CREDITS: calculations are from the lab documentation)

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{matrix}$$

  NOTE: The order of bits is reversed in binary. Thus, $D_3D_2D_1D_0$ for the matrix's input would be $D_0D_1D_2D_3$ in binary. The same is true for the parity bits and the resulting matrix.

- Methods associated with the encoder (Pseudocode)
  a. integer/byte ham_encode (BitMatrix pointer G, byte message)

```
extract the lower four bytes of message (message & 0xF) [Credits: From the lab doc]
convert the extracted bits to a matrix using bm_from_data where byte is message and length =
4 (a nibble)
multiply the returned array with generator matrix G using bm_multiply
return the result of bm_to_data where matrix is the resultant matrix of the multiplication
```

Decoding

- The integrity of the transmitted message (1x8 matrix) is calculated by decoding it to an error syndrome (1x4 matrix) by using a 8x4 matrix.
- This 8x4 matrix turns out to be the transpose H^T of the Generator matrix. (why?)
- After matrix multiplication, the error syndrome is either zero if no errors occurred, is the same as one of the rows of H^T if one error occurred, or would have different vectors if more than one error occurred during transmission.

- Methods & Data Structures associated with the decoder (Pseudocode)

a. STATUS ham_decode (BitMatrix pointer Ht, byte code, byte message pointer)

```
call bm_from_data to store into matrix where byte is the *message* and length = 8 (a byte)
multiply the transpose array Ht with the byte code array
call bm_to_data on the resultant matrix to get the error syndrome
check the error code in the lookup table
    if it is zero then
        call bm_to_data to get the message byte
        extract the last four (data) bits using nibble >> 4 (Credits: From the lab doc)
        store the nibble in *message pointer*
        return no error status
    else lookup the row index in the lookup table which corresponds to error code
        if it corresponds to a row in the matrix then
            correct/flip the $row^{th}$ bit with of code
            perform the same steps 1-4 when error is zero
            return corrected status
        if it is not zero and does not correspond to a row in the matrix then
            return more than one error status
```

b. Lookup Table

   i. Since the error syndrome (1x4 matrix) can have up to 2^4 - 1 values, a lookup table--an array--of 16 elements (zero is included) is created to avoid comparing the error syndrome with each row of the transpose matrix.

   ii. Each row of the transpose matrix corresponds to an index of the array. That index entry is set to 1 to indicate that the transpose has the error syndrome row and the bit could be flipped. All the other bits are set to HAM_ERR to indicate that no such row is present and more than one error occurred.

   iii. The transpose matrix is defined as follows: (Credits: Calculations are from the lab document)

Transpose	In Binary	In Decimal	Index
0 1 1 1	1 1 1 0	14	0
1 0 1 1	1 1 0 1	13	1
1 1 0 1	1 0 1 1	11	2
1 1 1 0	0 1 1 1	7	3
1 0 0 0	0 0 0 1	1	4
0 1 0 0	0 0 1 0	2	5
0 0 1 0	0 1 0 0	4	6
0 0 0 1	1 0 0 0	8	7

iv. Thus the lookup table array will look like as follows: (-2 = HAM_ERR)

e_2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bit	0	4	5	-2	6	-2	-2	3	7	-2	-2	2	-2	1	0	-2

NOTE: 0 is uninitialized since it means no error.

v. Hence, depending on the value of the lookup table index, we know which bit to flip if possible.

File I/O

- The input file, specified by -i argument, is opened in read binary mode. The default input file is stdin.
- After opening the file, the permissions of the input file are looked up using fstat and stored in a statbuf structure.
- Those permissions are also set for the output file using chmod.

Error Handling

- An error is returned if:
    - Input or Output file cannot be opened.
    - Permissions to file could not be set.
    - No input is read (cannot setup bit matrix).
    - Generator or the transpose matrix could not be allocated.

High-level Encoder Program Flow (error checking is done throughout the program)

```
Set default files to stdin and stdout
Parse arguments and open files
Change output file permissions
Create the 4x8 generator matrix (using a function that uses for loops)
While end of file is not reached:
    read one byte from the input file
    extract lower and upper halves of the byte
    encode both of them using ham_encode
    put both bytes (lower followed by upper byte) to output file
Free memory and close files
```

High-level Decoder Program Flow (error checking is done throughout the program)

```
Set default files to stdin and stdout
Parse arguments and open files
Change output file permissions
Create the 8x4 transpose matrix (using a function that uses for loops)
While end of file is not reached:
    read two bytes from the input file (lower followed by upper byte)
    decode both of them individually
    update statistics (actual values are stored in an array whose values can be referred by return status * -1 (to make it positive)
    pack lower and upper halves of the upper and lower decoded bytes
    put the packed byte to output file
Print statistics (If verbose option is passed)
    Look up the values in the array mentioned before
    note: the values for the indexes are stored in an enum which is used to access it
Free memory and close files
```