Tuesday, May 4, 2021

Assignment 5: Hamming Codes

Design Doc

- For this assignment we will be writing Hamming Code encoders and decoders to identify and recover errors in messages transferred in noisy conditions.
- The purpose of this assignment is to learn how Hamming Code works to error check and correct messages in noisy channels. We will also learn how matrix multiplication is used in c code.

Design Process/Current questions

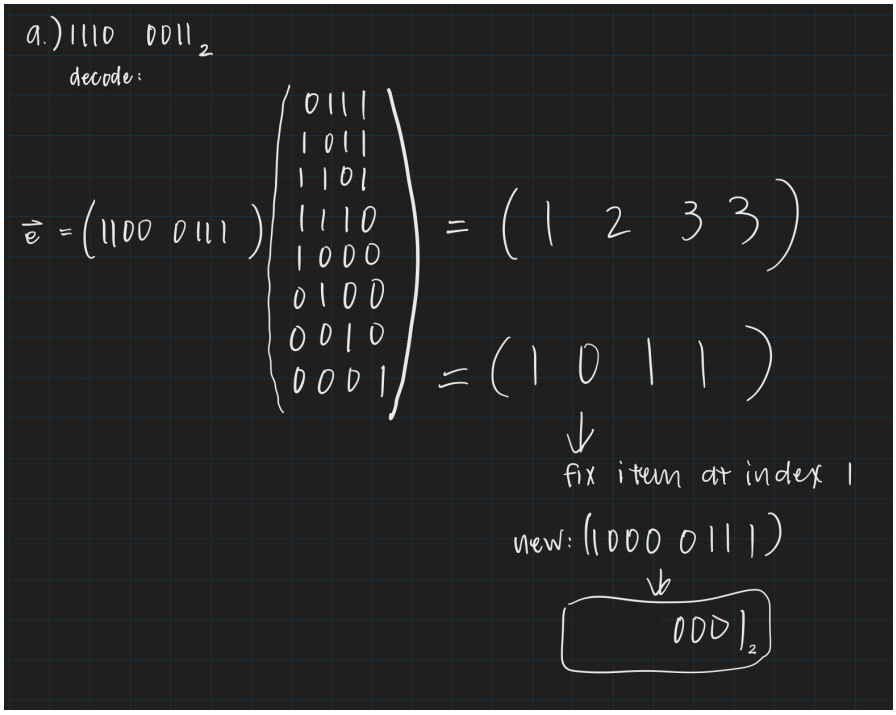
- I took linear algebra last fall, so I am fairly familiar with how hamming codes work. We did a group project on it and I did everything :(.
- The sections helped a lot on laying out how the matrix would be used for this assignment. It was quite confusing since we were treating a 1d array as a matrix.
- None of the logic in my initial design was wrong, but I did add more specific details about my code because I wasn't sure on that at the beginning of the assignment.
- The order a bit vector was supposed to be in (msb to lsb vs lsb to msb) was quite confusing. Ifound it easier to just forget how a matrix worked and just focus on the specific indices that are to be multiplied together.
- There were many many errors in my code in valgrind, most were impossible to find since my code worked fine. After lots of experimenting, I fixed it with a few simple if statements in my get_bit functions.

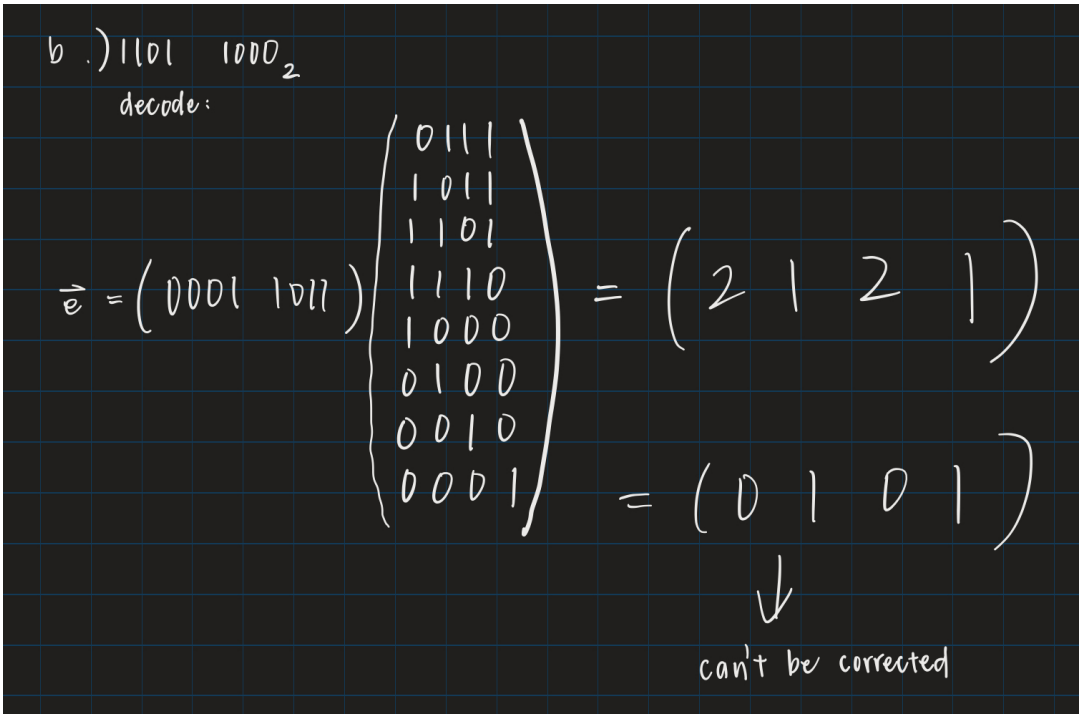
Prelab:

1. Lookup Table

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

2.


a.) 1110 0011₂
decode:
ē = (1100 0111)
0111
1011
1101
1110
1000
0100
0010
0001
= (1 2 3 3)
= (1 0 1 1)
fix item at index 1
new: (1000 0111)
0001₂


b.) 1101 1000₂
decode:
ē = (0001 1011)
0111
1011
1101
1110
1000
0100
0010
0001
= (2 1 2 1)
= (0 1 0 1)
can't be corrected

Pseudocode:

Encoder:

Will take in a file, take every 4 bits and encode it using matrix multiplication and the encoder matrix:

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

(Taken from design doc)

Every 4 bits will be multiplied to G for a 8 bit encoded version. Each character in the infile is 1 byte so every character will be split into upper 4 bits and lower 4 bits for the process. Then each generated encoded bit vector will be printed to the outfile.

Encoder:
main:
    take program argument with getopt, and set appropriate infile, outfile
    create generator matrix G
    for every character read in file:
        split into lower and upper 4 bits
        encode lower nibble
        encode upper nibble
        write both the outfile
    close file, free memory
HAM_Encoder:
    multiply message to matrix M, return the resulting bit vector

Decoder:

Will take in a file, take every 8 bits and decode it using the decoding matrix, and recover any error bits if possible using the decoding matrix:

$$H = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(Taken from design doc)

Every 16-bit read will be the upper and lower nibble of a character. Check for errors using matrix. Multiply the 8 bit vector to H. If the resulting bit vector is 0, then no errors. If the resulting 4 bit vector matches one of the vectors in matrix H, then that bit index in the encoded read bit is wrong, so flip. If the resulting bit vector is not 0, and not one in matrix H, then the error is non-reversible.

Decoder:

main:

- take program argument with getopt, and set appropriate infile, outfile
- create parity matrix H
- for every 2 characters read in file:
  - for lower and upper, check for error using parity matrix:
  - track status from returned ^^
  - put upper and lower of original character back together
  - write character to file output.
- print statistics to stderr
- close file, free memory

HAM_Decode:

- lookup table define
- multiply bit vector to matrix G to get error bit vector status
- if status in lookup is HAM_OK, then no corrections needed, all is good
- if status in lookup is HAM_ERR then no corrections can be made. Can't fix.
- if status in lookup is an index greater than/equal to 0, fix bit at that index.
- set pointer to fixed bit vector, return status

ADTs:

The two ADTs we will be using are for bit vectors and bit matrices.

Bit vector:

(Most of these functions were given or explained to us by Sahiti in her section)

Bit vector is an ADT that will help us perform bit operations on bit vectors. Has items: length (length of vector) and *vector, which are the items inside the bit vector

Functions:

- bv_create (length): allocate space for struct, set length to length and create space for an array of bytes initialized to 0.
- bv_delete: deletes everything inside struct, set pointer to NULL after all memory is freed
- bv_length: returns length of bit vector
- bv_set_bit (i): make bit i 1
- bv_clr_bit (I): make bit i 0.
- bv_get_bit(i): gets bit at i
- bv_xor_bit(i, bit): XORS the ith bit in bit vector with *bit*
- Bv print: prints the whole vector

Bit Matrix:

(Most of these functions were given or explained to us by Sahiti in her section)

Bit Matrix uses bit vector functions to create matrices and perform operations on a matrix. Has items: rows, cols, *vector (the matrix, made using bit vector)

Functions:

- bm_create (rows, cols): allocate space for struct, set rows and cols, create space for matrix, initialized to 0.
- bm_delete: deletes everything inside struct, will call bv_delete, set pointer to NULL after all memory is freed

- bm_rows: returns number of rows in matrix
- bm_cols: returns number of columns in matrix
- bm_set_bit (r, c): make item at row r column c 1
- bm_clr_bit (r, c): make item at row r column c 0
- bm_get_bit(r, c): gets bit at row r column c
- bm_from_data(byte, length): create a new matrix, with row = 1, column = length, copy over first *length* number of bits into new matrix
- bm_to_data: takes the first 8 bytes of a matrix and returns it as an 8 bit integer.
- bm_multiply(matrix A, matrix B): reform matrix multiplication on matrix A and matrix B and mod 2. Create a new bit matrix to hold the result and return it.
- bm_print: prints the whole matrix.