Kyven Wang
kwang126@ucsc.edu
9 May 2021

CSE13S Spring 2021
Assignment 5: Hamming Codes
Design Document

Description:

This assignment is made up of two programs: a Hamming code encoder and decoder operating with the Hamming(8, 4) systematic code.The generator and parity-checker matrices are represented with bit matrices which are made of underlying bit vectors. The input is read from stdin by default, or a file if specified. The output is also written to stdout by default, or a file if specified. The decoder can output statistics.

Pre-lab:

1. If e = 0, no error and don't need to look at look-up table. If one and only one of the digits differs from the rest, then it is a fixable error and appears on the parity-checker matrix, with its row index corresponding with the index of the bit that was erroneously flipped. Otherwise, there have been more than 1 errors or there have been no errors.

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

2.

1110 0011

$$e = [0+1+1+0+0+0+0+0,\ 1+0+1+0+0+0+0+0,\ 1+1+0+0+0+0+1+0,\ 1+1+1+0+0+0+0+1]$$

$$e = [0\ 0\ 1\ 0]$$

$$= 0100_2\ (4_{10})$$

The error syndrome has one digit that differs from the rest, indicating a singular error. By looking at the corresponding index in the above look-up table, we can see that there is an error at the fifth index, so the digit at the fifth index must be flipped in order to fix it.

1101 1000

$$e = [0+1+0+1+1+0+0+0,\ 1+0+0+1+0+0+0+0,\ 1+1+0+1+0+0+0+0,\ 1+1+0+0+0+0+0+0]$$

$$e = [1\ 0\ 1\ 0]$$

$$= 0101_2\ (5_{10})$$

The error syndrome has more than 1 digit that differs from the rest, which indicates more than 1 error. This is confirmed by looking at the above look-up table, where the corresponding index points to a HAM_ERR. Therefore, this error is unfixable.

Sources:

- asgn5.pdf (Professor Darrell Long)

Given Code: (taken from asgn5.pdf unless specified otherwise)

Bit Vector ADT:

```
struct BitVector {
    uint32_t length; // Length in bits.
    uint8_t *vector; // Array of bytes.
};
```

Bit Matrix ADT:

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

Enumerated Status Codes:

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

Using fstat() and fchmod():

```
// Opening files to read from and write to.
FILE *infile = fopen("filename", "rb");
FILE *outfile = fopen("filename", "wb");

// Getting and setting file permissions
struct stat statbuf;
fstat(fileno(infile), &statbuf);
fchmod(fileno(outfile), statbuf.st_mode);
```

Helper functions:

```
// Returns the lower nibble of val
uint8_t lower_nibble(uint8_t val) {
        return val & 0xF;
}

// Returns the upper nibble of val
uint8_t upper_nibble(uint8_t val) {
        return val >> 4;
}

// Packs two nibbles into a byte
uint8_t pack_byte(uint8_t upper, uint8_t lower) {
        return (upper << 4) | (lower & 0xF);
}
```

Files: (taken from asgn5.pdf)

- **encode.c** *Contains implementation of the Hamming Code encoder*
- **decode.c** *Contains implementation of the Hamming Code decoder*
- **error.c** *Provided in resources repo*
- **entropy.c** *Provided in resources repo*
- **bv.h** *Contains bit vector ADT interface*
- **bv.c** *Contains implementation of the bit vector ADT*
- **bm.h** *Contains bit matrix ADT interface*
- **bm.c** *Contains implementation of the bit matrix ADT*
- **hamming.h** *Contains interface of the Hamming Code module*
- **hamming.c** *Contains implementation of the Hamming Code module*
- **Makefile** *allows the grader to type make to compile the program. Typing make builds program and ./sorting alone as well as flags runs program.*

  - CFLAGS=-Wall -Wextra -Werror -Wpedantic included.

  - CC=clang specified.

  - make clean removes all files that are compiler generated.

  - make builds the encoder, decoder, error-injection program, and entropy-measure program, as does make all.

  - make encode builds just the encoder

  - make decode builds just the decoder

  - make error builds just the error-injection program

  - make entropy builds just the entropy-measure program

- *make format formats all source code, including the header files.*

- **README.md** *in Markdown. Describes how to use the program and Makefile.*
- **DESIGN.pdf** *design document*
- **WRITEUP.pdf** *a reflection containing graphs showing the amount of entropy of data before and after encoding, as well as an analysis of these graphs*

Top Level Pseudocode:

encode.c:

```
DEFINE OPTIONS: 'h' (print help msg), 'i' (allow user-specified infile), 'o' (allow
                user-specified outfile)

int main(int argc, char **argv) {
    while (getopt(argc, argv, OPTIONS) is parsing args) {
        switch (parsed arg) {
        case 'h':
            print help msg;
            end program;
            break;

        case 'i':
            set the infile to the user-given file path;
            break;

        case 'o':
            set the outfile to the user-given file path;
            break;

        default:
            break;
        {
    }

    generator matrix
    BitMatrix G = bm_create(4 rows, 8 columns);
    iterate through rows in each column so that there is a clearer logic for which bits
    to set
    for (column = 0; column < bm_cols(G); column += 1) {
        for (row = 0; row < bm_rows(G); row += 1) {

            first half columns, the bit at the coordinate where row = column is
            the only bit not set. second half columns, the bit at the coordinate
            where row = column is the only bit that is set.
            if (we are at a column within the first half of G’s columns) {
                if (row equals column) {
                    bm_set_bit(G, row, column);
                {
            } else {
                if (row doesn’t equal (column - 4)) {
                    bm_set_bit(G, row, column);
                }
            }
        }
    }

    while (there are still bytes to read from infile) {
        b = next byte in file;
        Hamming(8, 4) code = ham_encode(G, b);
        write Hamming(8, 4) code to outfile;
    }

    close infile;
    close outfile;
    delete BitMatrix;
    exit program;
}
```

decode.c:

```
DEFINE OPTIONS: ‘h’ (print help msg), ‘i’ (allow user-specified infile), ‘o’ (allow
                user-specified outfile), ‘v’ (print stats of decoding process to srderr)

int main(int argc, char **argv) {

    boolean that determines if program outputs statistics about decoding or not
    boolean stats = false;

    while (getopt(argc, argv, OPTIONS) is parsing args) {
        switch (parsed arg) {
        case ‘h’:
            print help msg;
            end program;
            break;

        case ‘i’:
            set the infile to the user-given file path;
            break;

        case ‘o’:
            set the outfile to the user-given file path;
            break;

        case ‘v’:
            set stats to true;
            break;

        default:
            break;
        {
    }

    parity-checker matrix
    BitMatrix H = bm_create(8 rows, 4 columns);

    same logic as initializing the generator matrix, but with rows and columns
    switched
    for (row = 0; row < bm_rows(H); row += 1) {
        for (column = 0; column < bm_cols(H); column += 1) {
```

```
        *first half columns, the bit at the coordinate where row = column is
        the only bit not set. second half columns, the bit at the coordinate
        where row = column is the only bit that is set.*
        if (we are at a row within the first half of H’s rows) {
            if (row doesn’t equal column) {
                bm_set_bit(G, row, column);
            {
        } else {
            if (column equals (row - 4)) {
                bm_set_bit(G, row, column);
            }
        }
    }
}

*counters for if the user wants statistics printed:
counter for amount of bytes processed*
int byte_counter = 0;

*counter for uncorrected errors;*
int err_counter = 0;

*counter for corrected errors;*
int corr_counter = 0;

while (there are still bytes to read from infile) {
    lower byte = first byte read;
    upper byte = second byte read;

    decode = ham_decode(lower byte);
    if (stats is true) {
        byte_counter += 1;

        if (decode = HAM_ERR) {
            err_counter += 1;
        } else if (decode = HAM_CORRECT) {
            corr_counter += 1;
        }
    }
```

```
        decode = ham_decode(upper byte);
        if (stats is true) {
            byte_counter += 1;

            if (decode = HAM_ERR) {
                err_counter += 1;
            } else if (decode = HAM_CORRECT) {
                corr_counter += 1;
            }
        }


        original += upper byte;
        shift original left 4 times;
        original += lower byte;
        write original to outfile;
    }

    if (stats is true) {
        error rate: uncorrected errors / bytes processed
        int err_rate = err_counter / byte_counter;

        print all the gathered statistics;
    }

    close infile;
    close outfile;
    delete BitMatrix;
    exit program;
}
```

bv.c:

```
equation to locate a byte in a bit vector
DEFINE BYTE(i): (i / 8)

equation to locate a bit in a bit vector.
DEFINE BIT(i): (i % 8)

struct BitVector {
    uint32_t length;
    uint8_t *vector;
};

BitVector *bv_create(uint32_t length) {
    BitVector v = allocate memory for a BitVector;
    v.vector = allocate memory for an array of (size / 8) + 1 length;

    if (sufficient memory was allocated for v and v.vector) {
        v.length = length;

        initialize every element to 0
        for (int i = 0; i < v.vector's length; i += 1) {
            v.vector[i] = 0;
        }
    }

    return v;
}

void bv_delete(BitVector **v) {
    if (v and v.vector exist) {
        free v.vector;
        free v;
    }

    return;
}

uint32_t bv_length(BitVector *v) {
    return v.length;
}
```

```
void bv_set_bit(BitVector *v, uint32_t i) {
    need to use bitwise operations to actually get the desired bit, same for all
    functions below where we modify a specific bit
    start with 1 (00000001) and shift it left until it is at the same index the bit we want
    to set is at. then do a bitwise OR and that bit has been set.
    set = 1 shifted left BIT(i) times;
    v.vector[BYTE(i)] OR= set;
    return;
}

void bv_clr_bit(BitVector *v, uint32_t i) {
    start with 1 (00000001) and shift left until at index we want to clear. then NOT it
    so that all the digits are 1 except the one at the index we want to clear. then do a
    bitwise AND on that and the specified byte to clear the bit.
    clr = 1 shifted left BIT(i) times;
    bitwise NOT on clr;
    v.vector[BYTE(i)] AND= clr;
    return;
}

uint8_t bv_get_bit(BitVector *v, uint32_t i) {
    shift 1 left until at index of bit we want. then simply AND with the byte to mask the
    other bits. now that we have the desired bit, shift back to the beginning.
    bit = 1 shifted left BIT(i) times;
    bit = v.vector[BYTE(i)] AND bit;
    bit = right shift BIT(i) times;
    return bit;
}

void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit) {

    specified bit = specified bit XOR’d with the given bit
    bit = left shift BIT(i) times;
    v.vector[BYTE(i)] XOR= bit;
    return;
}

void bv_print(BitVector *v) {
    for (int i = 0; i < v.vector’s length; i += 1) {
        get rid of all bits in front of it and then shift it all the way down
        print v.vector[BYTE(i)] shifted left BIT(i) times and then right 7 times;
    }

    return;
}
```

bm.c:

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

BitMatrix *bm_create(uint32_t rows, uint32_t cols) {
    BitMatrix m = allocate memory for a BitMatrix;

    if (sufficient memory was allocated for m) {
        m.rows = rows;
        m.cols = cols;

        create a BitVector of length rows * cols since total number of elements in a
        matrix = rows * cols
        bv_create(rows * cols);
    }

    return m;
}

void bm_delete(BitMatrix **m) {
    if (m and m.vector exist) {
        bv_delete(m.vector);
        free m;
    }
```

```
        return;
}

uint32_t bm_rows(BitMatrix *m) {
        return m.rows;
}

uint32_t bm_cols(BitMatrix *m) {
        return m.cols;
}

void bm_set_bit(BitMatrix *m, uint32_t r, uint32_t c) {
        m.vector[r][r * m.cols + c] = 1;
        return;
}

void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t c) {
        m.vector[r][r * m.cols + c] = 0;
        return;
}

uint8_t bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c) {
        return m.vector[r][r * m.cols + c];
}

BitMatrix *bm_from_data(uint8_t byte, uint32_t length) {
        BitMatrix m = allocate memory for BitMatrix;
        m.vector = allocate memory for array of size [1][length];

        if (sufficient memory was allocated for m and m.vector) {
                m.rows = 1;
                m.cols = length;








                for (int i = 0; i < length; i += 1) {
```

```
                *use the shifting to mask out the bits we don't want and leave us
                with the bit we want to put in the corresponding index. ex: the 5th
                bit corresponds to index 3, so shift left by 3 to mask out the 3 bits
                on its left and then shift right 7 so that it is the only bit left.*
                m.vector[0][i] = (byte shifted left by i) shifted right by 7;
            }
        }

        return m;
}

uint8_t bm_to_data(BitMatrix *m) {
        uint8_t data = 0;

        for (int i = 0; i < 8; i += 1) {

            *start from MSB and simply shift data left by 1 to make room for the next
            less significant bit*
            data = m.vector[0][i];
            shift data left by 1;
        }

        return data;
}

BitMatrix *bm_multiply(BitMatrix *A, BitMatrix *B) {
        BitMatrix C = allocate memory for BitMatrix;
        C.vector = allocate memory for array size [A.rows][B.cols];

        if (sufficient memory was allocated for C and C.vector) {

            *matrix multiplication: as we go down the columns in matrix A, go down the
            rows in matrix B and multiply the corresponding numbers. at the end of
            this operation, all of these numbers added up is the first index of the
            product matrix. move right 1 column in matrix B and repeat the process.
            after all of matrix B's columns have been iterated through, we now have
            the first row of the product matrix. if matrix A has multiple rows, we move
            to the next row in matrix A and repeat the entire process again, resulting in
            the corresponding row in the product matrix. XOR is equivalent to addition*
```

```
                *mod 2, AND is equivalent to multiplication mod 2.*
                for (a_rows = 0; a_rows < A.rows; a_rows += 1) {
                        for (b_cols = 0; b_cols < B.cols; b_cols += 1) {
                                for (a_cols = 0; a_cols < A.cols; a_cols += 1) {
                                        C.vector[a_rows][b_cols] XOR=
                                A.vector[a_rows][a_cols] AND B.vector[a_cols][b_cols];
                                }
                        }
                }
        }

        return C;
}

void bm_print(BitMatrix *m) {
        for (int rows = 0; rows < m.rows; rows += 1) {
                for (int cols = 0; cols < m.cols; cols += 1) {
                        print m.vector[rows][cols];
                }

                print a newline;
        }

        return;
}
```

hamming.c:

```
uint8_t ham_encode(BitMatrix *G, uint8_t msg) {
        BitMatrix c = bm_multiply(msg, G);
        return bm_to_data(c);
}

HAM_STATUS ham_decode(BitMatrix *Ht, uint8_t code, uint8_t *msg) {
        BitMatrix code_vector = bm_from_data(code, 8);
        BitMatrix c = bm_multiply(code_vector, Ht);
        uint8_t data = bm_to_data(c);
        if (data is 0) {
                *msg = data;
                return HAM_OK;
        } else if (lookup table[data] is a HAM_ERR) {
                *msg = data;
                return HAM_ERR;
        } else if (lookup table[data] is a number) {
                flip corresponding digit;
                *msg = data;
                return HAM_CORRECT;
        }
}
```

Changes:

- removed freeing of every element in a BitVector's vector
- changed freeing of BitMatrix vector so that it is freeing all the arrays instead of freeing every individual element
- corrected look-up table in pre-lab question 1 and answer in first part of pre-lab question 2
- changed how the size of the vector is determined in bv_create
- clarified on how we will actually find a specific bit in a bit vector
- corrected logic on how to set, clear, get, xor, and print a bit
- clarified on what kind of operations to do in the matrix multiplication and removed unnecessary mod 2 of entire matrix B and C
- removed HAM_STATUS ADT from top of hamming.c
- fixed generator and parity-checker matrix initialization logic
- made a BitMatrix vector a single BitVector rather than an array of BitVectors
- removed freeing of every element in a BitMatrix vector
- corrected look-up table in pre-lab question 1 again and answer in first part of pre-lab question 2 again