Shweta Jones
CSE 13S - Spring 2021

DESIGN - Asgn 5 - Hamming Codes

Lab Description:

This program acts as a linear error-correcting code in order to correct errors caused by punch card readers. This assignment contains 3 codes, hamming.c, bv.c, and bm.c. The generator.c and encoder.c use these files to print out an output file concerning this information. Calculating the parity bits one-by-one can append them to the message can provide one solution to creating a Hamming code for a message. Decoding this code can help to identify these arrows and recover the message when and if an error is found.

Pre-Lab Questions:

1. The general equation is:

$$(\text{code}) \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (\text{mod } 2)$$

a. **$1110\ 0011_2$**

$(11100011) \rightarrow (2234)\ \%\ 2 \rightarrow (0010) \rightarrow (0100)$

This contains an error in row 5, so I can correct this by flipping element 5.

b. **$1101\ 1000_2$**

$(11011000) \rightarrow (3232)\ \%\ 2 \rightarrow (1010) \rightarrow (0101)$

There is an error is this code, however, it is unfixable.

2. The look-up table looks like this:

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

Program Pseudocode:

decoder code:

$$(\text{code}) \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \pmod 2$$

-h : Prints out a help message
-i infile : Specify the input file path containing data
-o outfile : Specify the output file path to write the encoded data
-v : Prints statistics of the decoding process to stderr. The statistics to print are the total bytes processed, uncorrected errors, corrected errors, and the error rate. The error rate is defined as (uncorrected errors/total bytes processed), the ratio of uncorrected errors to total bytes processed

```
uint8_t msg = 0x00
While fgetc != EOF
        uint8_t packed = pack_byte(upper, lower)
        uint8_t decoded = ham_decode(m,packed, &msg)
        fputc(decoded, outfile)
bm_delete(&m)
fclose(infile)
fclose(outfile)
```

uint8_t pack_byte (uint8_t upper , uint8_t lower) - EXTRA FUNCTION

return (upper << 4) | (lower & 0xF)

encoder code:

$$(\text{code}) \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (\text{mod } 2)$$

-h : Prints out a help message
-i infile : Specify the input file path containing data
-o outfile : Specify the output file path to write the encoded data
Create generator matrix, create and set bits
Set bit using nested for loops
While fgetc != EOF
    uint8_t upper = upper_nibble(byte)
    uint8_t lower = lower_nibble(byte)
    uint8_t upperCode = ham_encode(m, upper)
    uint8_t lowerCode = ham_encode(m, lower)
    fputc(upperCode, outfile)
    fputc(lowerCode, outfile)
bm_delete(&m)
fclose(infile)
fclose(outfile)

void helpInfo(void) - EXTRA FUNCTION
    Prints help information

uint8_t lower_nibble(uint8_t val) - EXTRA FUNCTION
    return val & 0xF;

uint8_t upper_nibble(uint8_t val) - EXTRA FUNCTION
    return val >> 4

bv.c

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

```
BitVector *bv_create(uint32_t length)
        BitVector *v = (BitVector *) malloc (sizeof(BitVector));
          v->length = length;
          if (length%8 == 0)
             v->vector = (uint8_t *) calloc ((length/8), sizeof (uint8_t)
          else
             uint32_t diff = length%8
                uint32_t newlength = diff + length
                v->vector = (uint8_t *) calloc ((newlength/8), sizeof (uint8_t))
          Return v

void bv_delete(BitVector **v)
        free((*v)->vector)
         ree(*v)
         *v = NULL

uint32_t bv_length(BitVector *v)
        return v->length

void bv_set_bit(BitVector *v, uint32_t i)
        uint8_t mask = 1 << (i%8)
        v->vector[i/8] = v->vector[i/8] | mask

void bv_clr_bit(BitVector *v, uint32_t i)
        uint8_t mask = ~(1 << (i%8))
        v->vector[i/8] = v->vector[i/8] & mask

uint8_t bv_get_bit(BitVector *v, uint32_t i)
        uint8_t data = v->vector[i/8]
        return (data >> i%8) & 1

void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit)
        v->vector[i/8] = v->vector[i/8]^bit

void bv_print(BitVector *v)
        for (uint32_t i = 0; i<v->length; i++)
              printf("%d ", bv_get_bit(v, i))
        printf("\n")
```

bm.c

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

BitMatrix *bm_create(uint32_t rows, uint32_t cols)
    BitMatrix *m  = (BitMatrix *) calloc (1, sizeof(BitMatrix))
    m->rows = rows
    m->cols = cols
    m->vector = bv_create(rows * cols)
    return m

void bm_delete(BitMatrix **m)
    free((*m)->vector)
    free(*m)
    *m = NULL

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)
    uint32_t cols = bm_cols(m)
    bv_set_bit(m->vector, (r*cols)+c)

void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t c)
    bv_clr_bit(m->vector, r*(bm_cols(m))+c)

uint8_t bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c)
    uint32_t cols = bm_cols(m)
    return bv_get_bit(m->vector, (r*cols)+c)

BitMatrix *bm_from_data(uint8_t byte, uint32_t length)
    BitMatrix *m = bm_create(1, length)
    for (uint32_t i=0; i<length; i++)
        if (byte & (1<<i))
```

```
                bv_set_bit(m->vector, i)
            else
                bv_clr_bit(m->vector, i)
        return m

    uint8_t bm_to_data(BitMatrix *m)
        uint8_t val = 0
        for (uint32_t i=0; i < (bm_rows(m) * bm_cols(m)); i++)
            if (bv_get_bit(m->vector, i) == 1)
                val |= 1%2 << (7-i)
        return val

    BitMatrix *bm_multiply(BitMatrix *A, BitMatrix *B)
        BitMatrix *pdt = bm_create(A->rows, B->cols)
        for (uint32_t i=0; i<A->rows; i++)
            for (uint32_t j=0; j<B->cols; j++)
                uint8_t result  = 0
                for (uint32_t k = 0; k<B->rows; k+=1){
                    result += (bm_get_bit(A, i, k) * bm_get_bit(B, k, j))
                if (result > 0)
                    bm_set_bit(pdt, i, j)
        return pdt;

    void bm_print(BitMatrix *m)
        for (uint32_t i=0; i<bm_rows(m); i+=1)
            for (uint32_t j=0; j<bm_cols(m); j+=1)
                printf("%d ", bm_get_bit(m, i, j))
```

hamming.c

```
    uint8_t ham_encode(BitMatrix *G, uint8_t msg)
        BitMatrix *A = bm_create(1, 4)
        uint8_t mask = 1
        uint8_t bit
        for (uint32_t i=0; i<4; i++)
            bit = (msg >> i) & mask
            if (bit == 1)
                bm_set_bit(A, 0, i)
        BitMatrix *answer = bm_multiply(A, G)
        uint8_t val = bm_to_data(answer)
        return val
```

HAM_STATUS ham_decode(BitMatrix *Ht, uint8_t code, uint8_t *msg)

```
BitMatrix *A = bm_create(1, 8)
uint8_t mask = 1
uint8_t bit
for (uint32_t i=0; i<8; i++)
        bit = (msg >> i) & mask
        if (bit == 1)
                bm_set_bit(A, 0, i)
BitMatrix *answer = bm_multiply(A, G)
uint8_t val = bm_from_data(answer)
return val
```