Assignment 5 Design Document

Sean Siddens

CSE13s

Spring 2021

Overview

This assignment includes two programs: `encode,` which generates (8, 4)-systematic Hamming codes from inputted data, and `decode,` which decodes Hamming codes while detecting and correcting errors.

Bit Vectors

The process of encoding and decoding relies on representing our data as bit vectors, and then applying vector matrix multiplication. A bit vector has two fields, `length,` which is the length of the bit vector in bits, and `vector,` which is an array of bytes to store the bits in.

We use various functions in order to manipulate individual bits within the bit vector. In order to set a bit, we use bitwise OR on the bit we want:

```
bv_set_bit(v, i):
      v.vector[i / 8] |= 1 << (i % 8)
```

`i / 8` gets us the index of the byte we want, and `i % 8` will get the offset of the bit within the byte.

Clearing a bit utilizes the same idea but instead does a bitwise AND:

```
bv_clr_bit(v, i):
      v.vector[i / 8 ] &= ~(1 << (i % 8))
```

After shifting over to the bit we want, doing a NOT flips the bit to 0, and all other bits to 1, meaning doing an AND will have no effect on the other bits, and will only clear the bit we want.

We also need a function to XOR an individual bit within the bit vector with another bit:

```
bv_xor_bit(v, i, bit):
      v.vector[i / 8] ^= bit << (i % 8)
```

In order to get a bit from the bit vector, we shift the bit we want over to the LSB, and AND it with 1.

```
bv_get_bit(v, i):
      Return 1 & (v.vector[i / 8] >> i % 8)
```

Bit Matrices

Encoding and decoding requires vector matrix multiplication, so we use a bit matrix wrapper around bit vectors in order to facilitate matrix multiplication.

The bit matrix has 3 fields: `rows`, which are the number of rows of the matrix, `cols`, which are the number of columns of the matrix, and `vector`, which is a bit vector to store all entries of the matrix.

Although a matrix is two dimensional, we store it flattened within a single dimensional bit vector. Accessing the entry of the matrix at the r^{th} row and the c^{th} column is simply: `r * cols + c`, where `cols` is the number of columns in the matrix. In order to get, set, and clear a bit within the bit matrix, we simply call the bit vector implementations of these functions, passing the index computed with the above formula.

Another function we use is one which converts a provided byte of data into a 1 by `length` bit matrix, where length is some number less than or equal to 8. In order to set values of the bit matrix to the same values as the data we provide, we use bitwise XOR. Values in the bit matrix are guaranteed to be initialized to 0 when its newly created. 0 XOR 0 will be zero, and 0 XOR 1 will be one, so XORing the bits of the bit matrix with the bits of the byte we provide will ensure they both match.

```
bm_from_data(byte, length):
      m = bm_create(1, length)
      For (i = 0; i < length; i++):
            bv_xor_bit(m.vector, i, (byte >> i) & 1)
      Return m
```

We also implement a function to convert the first 8 bits of a bit matrix into a byte of data. If the number of columns of the bit matrix are less we set the first `cols` number of bits in the byte to match the bit matrix. We OR each bit in the byte with the corresponding bit in the bit matrix to set them to the same value.

```
bm_to_data(m):
      byte = 0
      If (m.cols < 8):
            length = m.cols
      Else:
            length = 8

      For (i = 0; i < length; i++):
            byte |= (bm_get_bit(m, 0, i) << i)

      Return byte
```

As mentioned above, matrix multiplication is used for encoding and decoding, so we implement a function for multiplying two bit matrices together. If we are computing the product of AB, where A and B are bit matrices, the product is only defined if the columns of A are equivalent to the rows of B. If A is an m by n matrix, and B is an n by k matrix, the product matrix will have dimensions m by k.

If we wish to compute the (r, c)-entry of the resulting product matrix, we compute the dot product of the r^{th} row of A and the c^{th} column of B. Instead of using scalar addition and multiplication to compute the dot product, we use bitwise XOR and AND, since these operations correspond to addition and multiplication mod 2.

```
bm_multiply(A, B):
      assert(A.cols == B.rows)
      M = bm_create(A.rows, B.cols)

      // Compute each entry of product matrix
      For (r = 0; r < M.rows; r++):
            For (c = 0; c < M.cols; c++):
                  entry = 0
                  // Compute dot product
                  For (i = 0; i < A.cols; i++):
                        entry ^= (bm_get_bit(A, r, i) & bm_get_bit(B, i,
c))

                  bv_xor_bit(M.vector, r * M.cols + c, entry)

       Return M
```

Encoding and Decoding

Our (8, 4) systematic Hamming code is 8 bits long, and encodes 4 bits of data. We represent the data we wish to encode as a 1 x 4 bit matrix, and this is multiplied by a generator matrix G in order to create the Hamming code for that message. The `ham_encode` function takes in the generator matrix and the 4 bit message and returns the 8 bit hamming code:

```
ham_encode(G, msg):
      M = bm_from_data(msg, 4)
      C = bm_multiply(M, G)
      code = bm_to_data(C)

      Delete M and C
      Return code
```

The program encode will read in a byte at a time from input, and encode each nibble of the byte using this function, writing the Hamming codes to output. If an output and an input file is provided, the output file must have the same permissions as the input file

The generator matrix G is a 4 by 8 matrix whose values should be initialized as follows:

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

Pseudocode for encode:

```
main(args):
      Input = stdin
      Output = stdout
      Process command line arguments:
            If arg == 'h':
                  Print help message
                  Exit program
            If arg == 'i':
                  Input = filename provided
            If arg == 'o':
                  Output = filename provided

      If input is not stdin and output is not stdout:
            Output.permissions = input.permissions

      G = create_generator_matrix()
      While ((input_byte = get_byte(input)) != EOF):
            Msg1 = lower_nibble(input_byte)
            Msg2 = upper_nibble(input_byte)

            Code1 = ham_encode(G, msg1)
            Code2 = ham_encode(G, msg2)

            put_byte(Code1, outfile)
            put_byte(Code2, outfile)
```

To decode a Hamming code, we multiply it by the transpose parity checking matrix H^T:

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

The resulting 1x4 matrix is called the error syndrome, and will allow us to check for errors and correct them if possible. If the error syndrome is the zero vector, then we know no error has occurred. If the error syndrome matches the n^{th} row of H^T, then the error occurred in the n^{th} bit of the Hamming code, and we can flip that bit in order to fix it. If the error syndrome does not match any of the rows of the matrix, then the error cannot be corrected.

We use a lookup table, where the index into the table is the error syndrome, and the value is the bit we have to flip. If we successfully correct the error, the ham_decode function returns HAM_CORRECT. However, if the error syndrome does not match any of the rows, the lookup value will be HAM_ERR, which is what we'll return. If the error syndrome is the zero vector, then there's nothing to correct, and we return HAM_OK.

Pre-lab 1:

Here's how the lookup table is defined:

Index	Value
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	HAAM_ERR
11	2
12	HAM_ERR
13	1
14	0
15	HAM_ERR

Pre-lab 2:

Here’s two examples of how to decode a Hamming code:

a) $1110\ 0011_2$

   c = (11000111)

   e = cH^T = (1011)

   Since (1011) matches the 2nd row of the parity checking matrix, then the error occurred in the second bit of the Hamming code, and that is the bit we flip.

   m = (1000)

b) $1101\ 1000_2$

   c = (00011011)

   e = cH^T = (0101)

   (0101) does not match any row, so we cannot correct it.

As mentioned above, `ham_decode` returns a status depending on whether it corrects an error, has no error to correct, or cannot correct the error. We pass the decoded message back through the pointer `msg` in the lower nibble.

```
ham_decode(Ht, code, msg):
      c = bm_from_data(code, 8)

      e = bm_multiply(c, Ht)

      Lookup_val = ham_lookup[bm_to_data(e)]
      If Lookup_val is HAM_OK or HAM_ERR:
            msg = lower_nibble(bm_to_data(c))

            Delete c and e
            Return Lookup_val
      Else:
            If bm_get_bit(c, 0, lookup_val) is 1:
                  bm_clr_bit(c, 0, lookup_val)
            Else:
                  bm_set_bit(c, 0, lookup_val)

            msg = lower_nibble(bm_to_data(c))
            Delete c and e
            Return HAM_CORRECT
```

The `decode` program reads in two bytes from input at a time, and decodes each byte using `ham_decode`, then repacks those decoded nibbles into a byte, which is then written back to output.

If the verbose option is provided, then we also print statistics, including how many bytes we processed, how many uncorrected errors there were, how many corrected errors there were, and the rate of uncorrected errors overall. Also, just like in the encoding program, if we provide an input and an output file, the output file should have the same permissions as the input.

Pseudocode for `decode`:

```
main(args):
      input = stdin
      output = stdout
      verbose = false

      Process command line args:
            If args == 'h':
                  Print help message
                  Exit program
            If args == 'v':
                  verbose = true
            If args == 'i':
                  input = input filename provided
            If args == 'o':
                  Output = output filename provided

      If input is not stdin and output is not stdout:
            input.permissions = output.permissions

      Ht = create_parity_checker_matrix()

      Total_bytes, Uncorrected_errs, Corrected_errs = 0

      While true:
            If (code_byte1 = get_byte(input)) == EOF:
                  Break
            If (code_byte2 = get_byte(input)) == EOF:
                  Break
            Total_bytes += 2

            Msg_lower = 0
            Status = ham_decode(Ht, code_byte1, msg_lower)
```

```
        If status is HAM_ERR:
            Uncorrected_errs += 1
        Else if status is HAM_CORRECT:
            Corrected_errs += 1

        Msg_upper = 0
        Status = ham_decode(Ht, code_byte2, msg_upper)
        If status is HAM_ERR:
            Uncorrected_errs += 1
        Else if status is HAM_CORRECT:
            Corrected_errs += 1

        put_byte(pack_byte(msg_upper, msg_lower), outfile)

    If verbose is true:
        print_statistics(stderr)
```