CSC 317 Homework #1 – Due 3/6/2015

Introduction to Error Correction Code: Hamming Code

Background:

In telecommunication, Hamming codes are a family of linear error-correcting codes that generalize the Hamming(7,4)-code invented by Richard Hamming in 1950. Hamming worked at Bell Labs in the 1940s on the Bell Model V computer, an electromechanical relay-based machine with cycle times in seconds. Input was fed in on punched cards, which would invariably have read errors. During weekdays, special code would find errors and flash lights so the operators could correct the problem. During after-hours periods and on weekends, when there were no operators, the machine simply moved on to the next job. Hamming worked on weekends, and grew increasingly frustrated with having to restart his programs from scratch due to the unreliability of the card reader. Over the next few years, he worked on the problem of error-correction, developing an increasingly powerful array of algorithms. In 1950, he published what is now known as Hamming Code, which remains in use today in applications such as ECC memory. Hamming codes are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance 3.

General algorithm

The following general algorithm generates a single-error correcting (SEC) code for any number of bits.

1. Number the bits starting from 1: bit 1, 2, 3, 4, 5, etc.
2. Write the bit numbers in binary: 1, 10, 11, 100, 101, etc.
3. All bit positions that are powers of two (have only one 1 bit in the binary form of their position) are parity bits: 1, 2, 4, 8, etc. (1, 10, 100, 1000)
4. All other bit positions, with two or more 1 bits in the binary form of their position, are data bits.
5. Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.
    a. Parity bit 1 covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.
    b. Parity bit 2 covers all bit positions which have the second least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.
    c. Parity bit 3 covers all bit positions which have the third least significant bit set: bits 4–7, 12–15, 20–23, etc.
    d. Parity bit 4 covers all bit positions which have the fourth least significant bit set: bits 8–15, 24–31, 40–47, etc.
    e. In general each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

The form of the parity is irrelevant. Even parity is simpler from the perspective of theoretical mathematics, but there is no difference in practice.

Here shown visually:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded Data Bits		P_1	P_2	D_1	P_3	D_2	D_3	D_4	P_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}
Parity Bit Coverage	P_1	x		x		x		x		x		x		x		x
	P_2		x	x			x	x			x	x			x	x
	P_3				x	x	x	x					x	x	x	x
	P_4								x	x	x	x	x	x	x	x

Here are only 15 encoded bits (4 parity, 11 data) shown, but the pattern continues forever. It is not hard to see that if you have *n* parity bits, you can have 2^n-1 bits, with 2^n-*n*-1 data bits. It is now easy to get all Hamming codes:

Parity Bits	Total Bits	Data Bits	Name	Rate
2	3	1	Hamming(3,1)	1/3
3	7	4	Hamming(7,4)	4/7
4	15	11	Hamming(15,11)	11/15
5	31	26	Hamming(31,26)	26/31
…				
n	2^n-1	2^n-*n*-1	Hamming(2^n-1, 2^n-*n*-1)	(2^n-*n*-1)/(2^n-1)

Implementation:

Because Hamming codes are linear codes the codes can be computed in linear algebra terms through matrices. For the purposes of Hamming codes, two Hamming matrices needs be defined: the code generator matrix $\boldsymbol{G}$ and the parity-check matrix $\boldsymbol{H}$. In the case of the Hamming(7,4) the matrices are:

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

Rows 1,2 and 4 of $\boldsymbol{G}$ should look familiar as they map the data bits to their parity bits.

- P_1 covers D_1, D_2 and D_4
- P_2 covers D_1, D_3 and D_4
- P_3 covers D_2, D_3 and D_4

The remaining rows (3, 5, 6, 7) map the data to their position in encoded form and there is only 1 in that row so it is an identical copy. In fact, these four rows are linearly independent and form (by design) the identity matrix.
The three rows of $\boldsymbol{H}$ look familiar. These rows are used to compute the syndrome vector at the receiving end and if the syndrome vector is the null vector (all zeros), the received word is error-free; if non-zero then the value indicates which bit has been flipped.

The matrix $\boldsymbol{G} = (\boldsymbol{I}_k|-\boldsymbol{A}^T)$ is called a (canonical) generator matrix of a linear (*n*,*k*) code,
and $\boldsymbol{H} = (\boldsymbol{A}|\boldsymbol{I}_{n-k})$ is called a parity-check matrix.
This is the construction of $\boldsymbol{G}$ and $\boldsymbol{H}$ in standard (or systematic) form. Regardless of form, $\boldsymbol{G}$ and $\boldsymbol{H}$ for linear block codes must satisfy $\boldsymbol{HG^T} = \boldsymbol{0}$ (an all zeros matrix).

Encoding:

To get the encoded message that is transmitted, we take the 4 data bits, assembled as a vector $\boldsymbol{p}$, and pre-multiply by $\boldsymbol{G}$, modulo 2. The original 4 data bits are converted to 7 bits with 3 parity bits added ensuring even parity. The following $\boldsymbol{p}$ will be used in this example:

$$\boldsymbol{p} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Suppose we want to transmit this data ($\boldsymbol{p}$), to get the message vector x we simply take the product of $\boldsymbol{G}$ and $\boldsymbol{p}$ modulo 2.

$$\boldsymbol{x} = \boldsymbol{Gp} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Parity Check:

If no error occurs during transmission, then the received message $\boldsymbol{r}$ is identical to the transmitted message $\boldsymbol{x}$: $\boldsymbol{r} = \boldsymbol{x}$. The receiver multiplies $\boldsymbol{H}$ and $\boldsymbol{r}$ (modulo 2) to obtain the *syndrome* vector $\boldsymbol{z}$, this vector will tell if an error occurred or not. If an error occurred, $\boldsymbol{z}$ will tell which message bit was wrong.

$$\boldsymbol{z} = \boldsymbol{Hr} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Error Correction:

Since $\boldsymbol{z}$ (the syndrome) is the null vector, no error has occurred. We base this observation on the fact that that when the data vector is multiplied by $\boldsymbol{G}$, a change of basis occurs into a vector subspace that is the kernel of $\boldsymbol{H}$. As long as nothing happens during transmission, $\boldsymbol{r}$ will remain in the kernel of $\boldsymbol{H}$ and the multiplication will yield the null vector. Otherwise, supposing a single bit error has occurred, we can write $\boldsymbol{r} = \boldsymbol{x} + \boldsymbol{e_i}$ modulo 2, where $\boldsymbol{e_i}$ is the *i-th* unit vector, that is, a zero vector with a 1 in the *i*th position counting from 1.

$$\boldsymbol{e_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

If we have a single bit error in the i^{th}, we multiply this vector ($\boldsymbol{r} = \boldsymbol{x} + \boldsymbol{e_i}$) with $\boldsymbol{H}$

$$\boldsymbol{Hr} = \boldsymbol{H}(\boldsymbol{x} + \boldsymbol{e_i}) = \boldsymbol{Hx} + \boldsymbol{H}e_i$$

Since $\boldsymbol{x}$ transmitted error free data the product of **H** and **x** is zero.

$$\boldsymbol{Hx} + \boldsymbol{He}_i = \boldsymbol{0} + \boldsymbol{H}e_i = \boldsymbol{He}_i$$

The product of $\boldsymbol{H}$ and the i^{th} standard basis vector will pick up the i^{th} column in $\boldsymbol{H}$, and we now know what column of $\boldsymbol{H}$ the error occurred.

$$r = x + e_5 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Now

$$\boldsymbol{z} = \boldsymbol{Hr} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

We have an error in the fifth column of $\boldsymbol{H}$. Another example

$$r = x + e_3 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

$$\boldsymbol{z} = \boldsymbol{Hr} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

If the syndrome indicate we have an error in the i^{th} entry all we have to do to correct it is to flip it (negate it).

Decoding:

Once we have deemed the received vector error free (it passed parity check or we corrected it), we will need to decode the message back to the original data. First we need a decoding matrix.

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

The columns in $\boldsymbol{R}$ should look familiar as there are all zeros in the columns that are parity bits and the remaining columns create the identity matrix.

The received data $\boldsymbol{p_r}$ is equal to $\boldsymbol{Rr}$ (using the example from above):

$$\boldsymbol{p_r} = \boldsymbol{Rr} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Assignment:
Write a program that can create, parity-check, error correct and decode a message using Hamming(7,4) or Hamming(15,11). The three Hamming matrices ($\boldsymbol{G}$, $\boldsymbol{H}$ and $\boldsymbol{R}$) for Hamming(7,4) is provided in the background. You will have to create the Hamming matrices for the Hamming(15,11) (I am more than confident you can do that). The program will first ask the user what Hamming code to be used (or if the user wants to exit). The next question should be if user wants to create, parity check (with error correction), or decode a message. The user will then provide either the data or the message, the program will calculate either the message or the data. All messages and data will be entered as a vector of binary digits. Example, four bit data `1011`, is a seven. Fifteen bit message, will have eleven data bits and four parity bits and look something like `100100111011110`, is 18910 (don't know if the message has the correct parity check or not).

Required elements:

- Typing `Hamming` will invoke your program.
- (85 pts) Implement the commands:
  - `Hamming(7,4)`
    - (30/85 pts) Uses Hamming(7,4) for encoding, parity-check and decoding
    - Usage: `H74`
      - `encode` will take a four bit data stream and create and display a seven bit message.
        - Usage: `encode 1011`
      - `parity` will take a seven bit message and check the parity if correct it will display a three bit zero vector. If incorrect it will display the three bit syndrome followed by the corrected seven bit message,
        - Usage: `parity 0100011`
      - `Decode` will take a seven bit message (assumed to have been parity checked) and create and display the four bit data.
        - Usage: `decode 0110011`
  - `Hamming(15,11)`
    - (50/85 pts) Uses Hamming(15,11) for encoding, parity-check and decoding
    - Usage: `H1511`
      - `encode` will take a eleven bit data stream and create and display a fifteen bit message.
      - `parity` will take a fifteen bit message and check the parity if correct it will display a four bit zero vector. If incorrect it will display the four bit syndrome followed by the corrected fifteen bit message,
      - `Decode` will take a fifteen bit message (assumed to have been parity checked) and create and display the eleven bit data.

    - `exit`
        - (5/85 pts) Exit the program
        - program termination
- (15 pts) Handle errors gracefully.
- Incorrect file or program names will not be accepted.

Solution method:

- Start the documentation and figure out how to generate the PDF: 2/2/15
- You will do the work in a subdirectory named `prog1`
    - `tar -czf prog1.tgz  prog1`
    - Note: **no prog1.tar.gz, no Prog1.tgz, no program1.tgz, no prog1.zip**,
- You will need to produce a Makefile to build the executable (sample below): 2/2/15
- Create the `main` function as an infinite loop and display a prompt. 2/4/15
- Handle an `<enter>` without an input. 2/6/15
- Modify the event loop code to recognize the exit command. 2/6/15
- Add the `H74` command and the `encode` to the `H74` 2/9/15
- Add `decode` to the `H74` 2/11/15
- Add the `parity` to the `H74` 2/13/15
- Work on the `Hamming(15,11)` implementation 2/16/15
- Have the `H1511` section done 2/27/15
- Add the error handling. 3/02/15

Testing:

- Try to break the code.
- Type at least the following and hit enter:
    - `<space>`
    - `<tab>`
    - random chars
    - no chars [blank line]
    - `<cmd> abc` (should not die even if it is expecting a binary string)

Code structure:

- Clean, modular, well documented code is required.
- Functions must be used to handle the user commands (a single large block is not acceptable).

Notes and hints:

- You are not required to write this in C. C++ has some great string handling abilities and so would be useful in this aspect. However, this code is only a few pages of C and is not difficult in plain C.
- You are given all the matrices for the Hamming(7,4), so that should ease the process of creating the structure of the program.
- There are libraries for linear algebra you are welcome to use them, but as you are only handling matrix-vector multiplication, it might be easier to write the functions yourself.
- Check your code with incorrect strings or just with an empty line (plain enter).

Submission contents:

- Documentation - all of the following should be in: `prog1.pdf`
  - Description of the program.
  - Description of the algorithms and libraries used.
  - Description of functions and program structure.
  - How to compile and use the program.
  - Description of the testing and verification process.
  - Description of what you have submitted: `Makefile`, external functions, main, etc.
  - Format: PDF (write in any word processor and export as PDF if the option is available, or convert to PDF)
- Main program `Hamming.c`.
- Any required external functions `*.c`.
- Any include files you use (other than the standard ones).
- The `Makefile` to build the program `Hamming`.
- Tar the directory and then gzip using the correct filename.
  [And I do know that there are better compression routines than gzip.]

Grading approach:

- copy from the submit directory to my grading directory
- `tar -xzf prog1.tgz`
- `cd prog1`
- `make`
- `Hamming`
- enter commands and compare output
- assign a grade
- `cd ..`
- `rm -rf prog1`

Example output:

```
>>./Hamming
. . .
encode 1011
x = <0, 1, 1, 0, 0, 1, 1>
. . .
parity 0100011
z = <0, 0, 0>
. . .
parity 0100111
z = <1, 0, 1>
x = <0, 1, 1, 0, 0, 1, 1>
. . .
decode 0110011
r = <1, 0, 1, 1>
```

Submission method:

You will use the submit page to submit your program. The files need to be tarred and gzipped prior to submission. Please empty the directory prior to tar/zip. [You will tar up the directory containing the files: `tar -czf prog1.tgz prog1`]

Sample Makefile to build "crash" (your version will be slightly different):

```
#---------------------------------------------------------------------

# Use the GNU C/C++ compiler:
CC = gcc
CPP = g++

# COMPILER OPTIONS:

CFLAGS = -c

#OBJECT FILES
OBJS = crash.o foo.o


crash: crash.o foo.o
	${CC} -lm ${OBJS} -o crash
crash.o: crash.c
foo.o: foo.c


#---------------------------------------------------------------------
```

Note: indents are tabs - NOT SPACES. Tab is a command (yes, horrible design to have whitespace commands).