CENG 325 Program #1 – Due 9/25/2020

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 $\boldsymbol{H}\boldsymbol{G}^{\boldsymbol{T}} = \boldsymbol{0}$ (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.

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

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 Hamming code. The three Hamming matrices (***G***, ***H*** and ***R***) for Hamming(7,4) is provided in the background. You will have to create the appropriate Hamming matrices to receive any of the higher grades (I am more than confident you can do that).

Code structure:

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

Notes and hints:

- You are required to write this in Python. Python has some great numeric libraries, so I recommend you look at `NUMPY`, as it would be useful in this aspect. A couple of others that could be helpful are: `RANDOM` and `MATH`.
- You are given all the matrices for the Hamming(7,4), so that should ease the process of creating the structure of the program.
- The `DOT`-function in `NUMPY` handles matrix-vector multiplication, and the `REMAINDER`-function handles modulo 2, so it is a good idea to look up those function.

Submission contents:

- Documentation - all of the following should be in: `Hamming.pdf`
  - Description of the program.
  - Description of the algorithms and libraries used.
  - Description of functions and program structure.
  - Description of the testing and verification process.
  - Description of what you have submitted and how to run it!
  - A clear statement that tells me what level I will be grading at!
  - Format: PDF (if you are not used to write LaTex, or use Overleaf, you can write in any word processor and export as PDF if the option is available, or convert to PDF)
- Main program `Hamming.py`.
- Submit your code and the pdf in the Dropbox on D2L.

Required elements :

For a D (63 – 73%):
Just implement what is in the text (***G***, ***H***, and ***R***), and use [1,0,1,1] as ***p*** to create ***x***. Use ***x*** as ***r*** to do the first parity check. Next use [0,1,1,0,1,1,1] as ***r***, do the parity check. You do not need to implement the error vectors, and do the correction.

Example output:
```
Message [1 0 1 1]
Send Vector [0 1 1 0 0 1 1]
Received Message [0 1 1 0 0 1 1]
Parity Check [0 0 0]
```

For a C (73 – 83%)

- Use the three Hamming matrices (***G***, ***H***, and ***R***) that are in the text.
- Randomly create a 4-bit array to use as ***p***, and create ***x***.
- Apply an error to a random bit in ***x*** (allow 0 to mean no error, and 1 to be the zero-bit etc).
- Do a parity check on ***x*** (i.e. create ***z***).
- If needed do an error correction on the correct bit (note that the ***z*** is in little endian bit order).
- Decode the corrected message.

Example output.
```
Message          : [1 1 0 0]
Send Vector      : [0 1 1 1 1 0 0]
Received Message : [0 1 1 1 1 1 0]
Parity Check     : [0 1 1]
Corrected Message: [0 1 1 1 1 0 0]
Decoded Message  : [1 1 0 0]
>>> main()
Message          : [0 1 0 0]
Send Vector      : [1 0 0 1 1 0 0]
Received Message : [0 0 0 1 1 0 0]
Parity Check     : [1 0 0]
Corrected Message: [1 0 0 1 1 0 0]
Decoded Message  : [0 1 0 0]
```

For a B (83 – 93%)
You will now allow the user to decide if they will use Hamming(7,4) or Hamming(15,11). So you will need two versions of (***G***, ***H***, and ***R***), all six can of course be hardcoded into the code.

- Prompt the user to tell what mode to use:
  `H1511` for Hamming(15,11) and let anything else default to Hamming(7,4)
- Create a random 4-bit or 11-bit (depending on mode) array as ***p***.
- Hamming encode ***p*** (i.e. calculate ***x)***.
- Create a receive vector ***r*** with a random error (allow for the possibility for no error).
- Do a parity check on ***x*** (i.e. create ***z)***.
- If needed do an error correction on the correct bit (note that the ***z*** is in little endian bit order).
- Decode the corrected message.

Example output.

```
>>> main()
Enter mode:H1511
Message          : [1 1 1 1 0 1 0 1 0 1 0]
Send Vector      : [1 1 1 1 1 1 1 1 0 1 0 1 0 1 0]
Received Message : [1 1 1 1 1 1 0 1 0 1 0 1 0 1 0]
Parity Check     : [1 1 1 0]
Corrected Message: [1 1 1 1 1 1 1 1 0 1 0 1 0 1 0]
Decoded Message  : [1 1 1 1 0 1 0 1 0 1 0]
>>> main()
Enter mode:H74
Message          : [0 0 0 1]
Send Vector      : [1 1 0 1 0 0 1]
Received Message : [0 1 0 1 0 0 1]
Parity Check     : [1 0 0]
Corrected Message: [1 1 0 1 0 0 1]
Decoded Message  : [0 0 0 1]
```

For an A (93 – 100%)

This is the full Monty! Now we will allow the user to enter the number of data bits to use. This can be any number greater than 1. We will then automatically create the three Hamming matrices (***G***, ***H***, and ***R***), of the correct size depending on the amount of data-bits the user decides to use.

- Prompt the user for the number of data-bits (***n***)
- Calculate the appropriate amount of parity bits.
- Create ***G***, ***H***, and ***R*** of the appropriate size.
- Create a random ***n***-bit array ***p***.
- Hamming encode ***p***.
- Create a receive vector ***r*** with a random error (allow for the possibility for no error).
- Calculate the syndrome.
- If needed do an error correction to the correct bit.
- Decode the corrected message.

Example output

```
>>> main()
Enter number of databits: 8
Message          : [0 0 1 1 1 0 1 0]
Send Vector      : [1 1 0 0 0 1 1 0 1 0 1 0]
Received Message : [1 1 0 0 0 1 1 0 1 0 0 0]
Parity Check     : [1 1 0 1]
Corrected Message: [1 1 0 0 0 1 1 0 1 0 1 0]
Decoded Message  : [0 0 1 1 1 0 1 0]
>>> main()
Enter number of databits: 16
Message          : [0 1 0 1 0 1 1 0 1 1 0 0 0 0 1 1]
Send Vector      : [1 0 0 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 0 1 1]
Received Message : [1 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0 0 0 0 1 1]
Parity Check     : [1 0 0 1 0]
Corrected Message: [1 0 0 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 0 1 1]
Decoded Message  : [0 1 0 1 0 1 1 0 1 1 0 0 0 0 1 1]
```

```
>>> main()
Enter number of databits: 31
Message          : [0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 1 1 0 1 0 0 1 0 0 0 1 0 1 1 1]
Send Vector      : [1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 0 0 0 0 1 0 1 1 1]
Received Message : [1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 0 0 0 0 1 0 1 1 0]
Parity Check     : [1 0 1 0 0 1]
Corrected Message: [1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 0 0 0 0 1 0 1 1 1]
Decoded Message  : [0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 1 1 0 1 0 0 1 0 0 0 1 0 1 1 1]
```

Grading approach:

- Copy the code from the submission to my computer.
- Run the code according to your document
- If the code runs correctly according to the level you ask to be graded at
  you will receive the lowest grade for that level.
- If the code do not execute, or provide the wrong output, you will receive zero points for the code part.
- The remaining 7 to 10 points will be rewarded for your program write-up.