*CS4220 Project 1*

*Hamming Code Error Detection*

Worsham, James R.

Department of Computer Science

University of Colorado

Colorado Springs, Colorado, U.S.

jworsham@uccs.edu

Rawlins, Matthew

Department of Computer Science

University of Colorado

Colorado Springs, Colorado, U.S.

mrawlin2@uccs.edu

Jessop, Thomas

Department of Computer Science

University of Colorado

Colorado Springs, Colorado, U.S.

tajessop2@gmail.com

*Abstract*—this document presents the design of a program that demonstrates the capabilities of Data Link Layer Hamming codes. The project presented and subsequent algorithms demonstrate the successful application, and prove the correctness of Hamming codes in order to handle physical discrepancies in computer networking. Hamming codes are used to ensure the correctness of data received over a network connection; if bits in the data happen to be flipped during transmission Hamming codes can account for the bits and correct them.

Keywords—networks, error correction, Hamming Codes

# Introduction

In the field of computer science one of the most important concepts is computer networking; without network technology, computers and individuals would not be able to communicate and the field would be severely limited in application. As the development of technology and communication progressed, it was noticed that data on a wire is not always provided in integrity, meaning that data sent from one end of a line may not be the same as data received on the other end. In an effort to handle this physical limitation, scientists developed mathematical and numerical methods to account for such loss potential. One such algorithm developed was that of Hamming Codes which uses a pattern of data bits and check bits to compute a modulo 2 sum parity, in order to find and correct incorrect bits in a string of data.

# Hamming Codes

Using Hamming codes is a technique that can be utilized to perform error correcting on the data link layer. The Hamming Code technique is based on another system called Hamming distance, which is the “number of bit positions in which two codewords differ”. The significance “is that if two codewords are a Hamming distance *d* apart, it will require *d* single-bit errors to convert one into the other”. [1] The algorithm for finding the Hamming distance is based on a lower limit property of the number of check bits needed to correct single errors. This method of check bits provides the foundation for the use of Hamming codes.

The utilization of Hamming codes is the error correction technique where the “bits of the codeword are numbered consecutively, starting with bit 1 at the left end, bit 2 to its immediate right, and so on. The bits that are powers of 2 are the check bits. The rest are filled up with the m data bits”. [1] Based on a parity-check bit, this technique has the ability to correct any single mistake in the data transmission, such as a dropped bit. Our code demonstrates this concept by taking any codeword, encoding and decoding it using Hamming Code, and proving that a single bit error can be corrected based on the parity and check bits.

# Solution

## The Algorithm

The algorithm we developed to solve this problem, is based off of a dynamic for-loop that iterates over powers of two. We decided that for the sake of writing good code, we did not want to hash out a hardcoded method, but rather devised a dynamic algorithm that would work for any length of data. The caveat is that while our algorithm (both for encoding and decoding) works for any length of data, we have it uniquely defined at the moment to only handle the standard 8 bit byte. We did for the simplicity of the project itself; it would have required more robust input handling beyond the scope of this project to handle dynamically sized data, but the algorithm would support it.

We observed the fact that if we take the value 1 and shift it left (<<) by any number X, it yields the following result:

1 << x = 2^x

We used this numerical property to devise a for-loop that simply iterates over every power of 2 in a set of data. This for-loop is used in both the encoding and decoding because they both require an iteration of powers of 2. Furthermore, this for-loop accounts for the read power-of-2, skip power-of-2 pattern used by Hamming code calculations (i.e. 1 on 1off, 2 on 2 off, 4 on 4 off, etc.). The for-loop works as follows:

int x;

int y;

int cnt = 0;

for (x = 1; x < DEFAULT\_SIZE; x = 1 << ++cnt) {

y = --x;

while (y < DEFAULT\_SIZE) {

/\*

\* Encoder/Decoder logic here

\*/

// Skip power-of-2 the next set

y += (x + 1);

}

}

where the DEFAULT\_SIZE was the defined value we used for the size of our codeword.

For the process of creating the codeword, we first allocate enough memory for the codeword and place the original data in the appropriate indices of the new codeword (this is done with a for-loop and power-of-2 bit shifting calculations). Then we use the dynamic, power-of-2 for-loop discussed above and calculate and pad the retrospective parities at each check bit. That completes the codeword, and it is returned to the calling system.

The process of decoding works in a similar manner; we start by using the power-of-2 for-loop and iterating in the Hamming code manner (1 on 1off, 2 on 2 off, 4 on 4 off, etc.) to calculate the retrospective parities. At any case, if a parity is found to be odd (which indicates a dropped bit), we grab the index of where we are at. Once this process is finished, we simply check which indices we found to be incorrect, add the numbers together and flip the bit at that location. Then we reconstruct the original byte by removing the check bits, and return the original byte to the caller, thusly correcting any integrity problems happening during transmission. A good note is that this algorithm works whether or not a bit was changed, due to the nature of the codeword.

## The Program

The solution we wrote takes one parameter from the command line. The argument is a path to a text file that contains the byte string in question. All that should be in the text file is a single byte string (no whitespace). For example, 00101101 would be acceptable content of the input file. Once the program starts running, it will display the byte string found within the input file and the codeword resulting from encoding the original string with Hamming codes. At this point, the program will prompt the user as to if the data hypothetically changed during transmission (to simulate dropping a bit). If the user selects yes (‘Y’), the program will prompt for what the string looks like on the other end of the transmission. Then the program will use the check bits and parity of the codeword to decode it and reconstruct the original string. In all cases where only one (or none) bit was flipped, the decoder will return the original string; thusly proving the correctness and applicability of Hamming codes.

To compile the source code into an executable, we simply used the GNU C Compiler (GCC). The command we used to compile was as follows: gcc -o ham cs4220\_main.c. This command works on Linux machines and will work on Windows machines through Cygwin (assuming the Cygwin installation supports GCC). To run the code on Linux we used ./ham input.txt, and on Windows we used ham.exe input.txt.

## Testing

This program was tested using various inputs and ensuring in all cases that the encoded string and decoded string were correct. For each set of inputs we tried, we calculated the codeword by hand and ensured it matched the one the program produced. Furthermore, we tested the decoder without flipping the bits and flipping random bits to ensure that the decoding of the parity worked. In all cases, the program produced the correct coderword and decoded data. For an example run of the program without simulating dropped bits, see Figure 1.

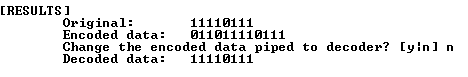


Figure 1

For an example run of the program in which a bit is flipped, see Figure 2.

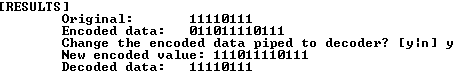


Figure 2

# Conclusion

The use and accuracy of Hamming codes was proven a viable solution by this program. This project effectively showed how Hamming codes can be used for error correction in network transmission. It is a lightweight, portable and integrity-proven technique for ensuring that bits in data are not dropped while being transported over physical network connections.

# References

1. A. S. Tanenbaum and D.J. Wetherall, Computer Networks, 5th ed. , Boston, Mass.: Prentice Hall, 2011.