Kyven Wang

kwang126@ucsc.edu

9 May 2021

CSE13S Spring 2021

Assignment 5: Hamming Codes

Writeup

In this assignment, we had to create a Hamming(8, 4) encoder and decoder utilizing bit vectors and bit matrices. While working on this, I gained a lot of insight into: bit vectors and matrices, encoding and decoding data, look-up tables, memoization, and bitwise operations. When we were introduced to this assignment, the big question was: How can we make changes to individual bits when the data we are working with is byte-addressable? The answer to this question was bit vectors. We make our own data type whose main component is an array of bytes. Given an amount of bits, allocate space for however many bytes are needed to hold these bits. Then we make our own functions that enable us to make changes to and get information about the individual bits within these bytes, essentially making our own “bit-addressable” data type. In order to implement this, we had to make extensive use of bitwise operations, such as shifting, AND, OR, and XOR. These operations not only allow us to do things to individual bits, but they also allow us to do math operations in mod 2. Shifting can be used to obtain a desired bit, AND can be used to mask out undesired bits and also as multiplication mod 2, OR can be used to set bits, and XOR can be used to flip bits and also as addition mod 2. With these tools, making our own “bit-addressable” data type was not so difficult.

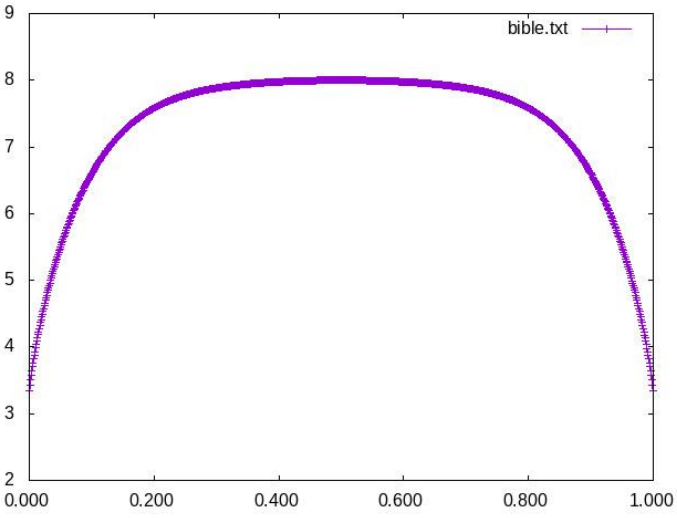
Once we had the solution to the problem of byte-addressable data, we needed to solve another problem, which was the issue of encoding and decoding data using Hamming(8, 4)

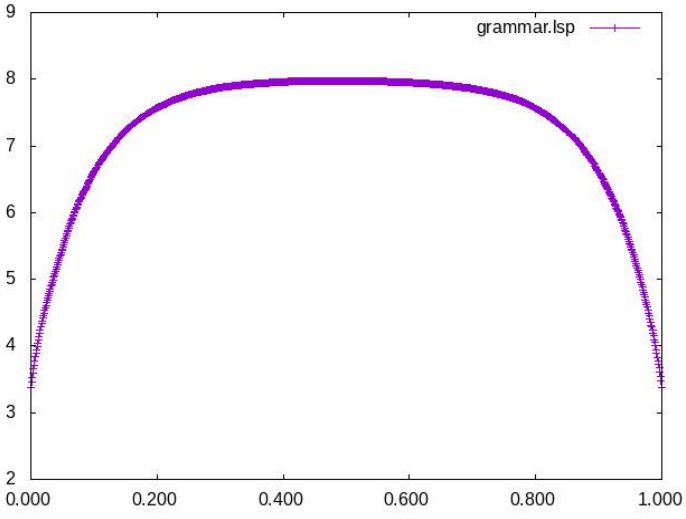
systematic code. In order to do this, we needed to use matrix multiplication, multiplying nibbles of data by a generator matrix to encode, and multiplying bytes of encoded data by a parity-checker matrix to decode. The first step to figuring this out was creating a bit matrix data type utilizing the bit vectors we had just created. Rather than making a 2D array with each element representing a bit, we instead used a singular 1D bit vector whose size was equal to the amount of elements in the matrix. Since we know the amount of rows and columns in the matrix, we are able to divide this single bit vector into pieces, with each piece representing a new row. This seemed a little strange at first, but it really isn't so strange when you realize that 2D arrays are essentially the same thing: a 1D array of arrays. Once I had this figured out, the matrix operations themselves were relatively easy to figure out, especially considering my taking a linear algebra course this quarter.

After these two ADTs came the actual encoding and decoding part of the assignment: starting with creating the functions to actually encode and decode given data. This is where I had my first experience with a look-up table, which is really just an array stored in memory. Plugging the error syndrome into this look-up table to get the corresponding error is much faster and less expensive than comparing the error syndrome to each row in the parity-checker matrix every time we decode. Similarly, I used memoization for the first time in my encoding program. Rather than calculating parity bits by multiplying with the generator matrix every single time, we can do a calculation once and then save the result into an array at the index corresponding with the data bits, so next time we have to perform that same calculation we don't actually have to do it again.

When I finished the programs, it was time to do some testing on the entropy of our encoder using an error-injection program and entropy measuring program provided to us. I started by first getting the entropy of the files before they were encoded, and then did many tests

on the encoded files through a range of 0.000 - 1.000 error rate. Throughout my tests, I noticed one big detail: the graphs for entropy vs. error rate all looked pretty much the same no matter the length of the file. Another detail is that the entropy of the files before they were encoded was larger than the entropy after they were encoded with either a very low or very high error rate. Below are two graphs that represent this. The x-axis is the rate at which errors are injected into the encoded file, and the y-axis is the resulting entropy. The first graph is the entropy of bible.txt, which contains roughly 30000 lines (~4,000,000 bytes). Its entropy before being encoded is 4.342751. The second graph is the entropy of grammar.lsp, which contains roughly 90 lines (~3,700 bytes). Its entropy before being encoded is 4.632268. Despite the large difference in file size, we can see from these two graphs that the entropy is roughly the same across all tests. A couple of small differences are: grammar.lsp's graph seems to start levelling off a bit more sharply than bible.txt, and bible.txt's entropy peaks at 7.999976 while grammar.lsp peaks at 7.977114, slightly lower than bible.txt.





Since the entropy between large files and small files remains pretty much the same, we have to draw our conclusions about entropy through the general shape of these graphs rather than the differences between them. Entropy, from what I've gathered, is essentially how "predictable" a given message is. Using this definition, we can draw the conclusion that an encoded message with little to no errors and an encoded message with a very high error rate is more predictable than the original text before being encoded. Therefore, with little to no errors, we get a low entropy because the encoded message is still readable and therefore the information is easily derived from it. On the other hand, with a higher error rate, the entropy is a lot higher as the message becomes more and more corrupted, making it less predictable. One odd thing about these graphs, however, is that after an error rate of 0.5, the entropy begins decreasing as the error rate gets higher and higher. The rate at which it decreases is about equal to the rate at which it initially increased when the error rate was rising from 0 - 0.5. The reason why I think this happens is that, as the errors become more and more common, the text in a way becomes more and more predictable. It is difficult for me to put this into a solid explanation, but I think the edge cases are a good way of demonstrating this idea. Notice how with 0 errors, the entropy is about equal to if the error rate is 100%. These two cases are equally predictable, since with a 0% error rate we know that none of the characters will be unpredictably changed. On the other hand, with a 100% error rate we know that all the characters will be erroneously changed, so in both of these cases the information that follows any given part of the text is 100% predictable.