ECE 350 Final Project Report

Duc Tran

April 2020

1 Project Specification

My project was designed to be a Hamming encoder and decoder done in Java. In addition to that, the I planned on creating a UI interface where users can input a message, then encode it and get the encoded result. Then the users can also input a string of binary and decode the binary can print out the decoded result.

The Encoder class was designed to take in a message string and encodes the message character by character. The class has two methods to take in and encode two types of message strings; one is just a regular message of letters and numbers, while the other method is designed for a message of binary characters. The hamming encoder takes each character and converts it into its 8-bit ASCII representation. Then it applies a SECDED hamming code to the 8-bit character, so the codeword has 13 bits: 5 parity bits, and 8 message bits.

The Decoder class was designed to take in a binary string and decode the binary 13-bits at a time. Besides decoding the binary, whenever my code detects a single error, or error that it can't classify, it records the position of the character that flagged the error. This was mostly done for testing, but I also thought that it would be a useful feature to demonstrate that my code works.

The UI was implemented in Java using the JavaFX package. The main application window is divided into two halfs. The left half is devoted to the encoder portion, it has two input fields to take in the two types of messages that my Encoder class can handle, and a results box to get display the encoded message. The right half is for the decoder functionality and has a one input field and one results box.

2 Challenges

I had two main challenges when completing my project: handling more than two flipped bits, and getting JavaFX to run. I'll address the latter issue briefly, since it is not that relevant to 350 topics. I had issues with my IDE recognizing the JavaFX .jar files, but after modifying some configuration files I finally got it to work.

The more important issue was handling more than two flipped bits in the decoding proccess. Since my hamming code was a SECDED code, I know that it can only correctly fix one error and only correctly detect two flipped bits, but if more than two flipped bits occurred, that would cause my Java code to throw an error. The error was an index out of bound error since I used an array to store the binary bits. In the Decoder class, I use the checking bits to determine the position of any flipped bits, but if more then two bits were flipped, the checking bits would produce a number larger then twelve, which would cause an index out of bounds error because I used a size thirteen array to store the binary bits when I decode them. Once I realized that, checking bits were producing numbers larger then twelve, I check for that case and made the program stop decoding those thirteen bits, and moved onto the next thirteen bits.

3 Testing

The first part of the project I finished was the Encoder class, so I tested that first. I started testing by hand encoding several characters like 'a', 'q', and '@'. Then I fed the same characters to my encoder and compared the output to my hand calculated codewords. After I confirmed that the single character encoding worked, I tested out small words by the same hand calculation and comparison method.

Next, I had to test the Decoder class. First, I wanted to see if the decoder code correctly translate messages that had no errors. Since I knew that my encoder was working correctly, I used it to encode several words and sentences, and then fed the encoded messages to my decoder. I then checked if any characters were changed or if any errors were incorrectly detected. After that test, I encoded thirteen characters, and then flipped a different bit for each character. I then inputted those thirteen characters into the decoder and checked if the result was correctly fixed or not. Once all of the characters passed, I repeated the process but flipping two bits of each character and putting them as input

to the decoder. For this test though, I was only checking to see if a double error was detected. The last part of the decoder I wanted to test was what would happen if an encoded message with more then three errors was inputted, and as stated above in the challenges section, this caused my program to crash. My goal for testing this input case was just to make sure my program wasn't crashing, not how my decoder was handling the bits, so after the fix mentioned above, I moved onto the testing the UI.

The UI was simple to test since I only had to make sure that it was correctly taking in my inputs and correctly displaying the result of the encoding and decoding. Since I could visually see the result of my testing, I just had to make sure that the correct strings were appearing in the right fields and labels as I input messages.

4 Code Analysis

My Encoder class has two methods that start the encoding process. One method takes in a binary string and then calls the encoding method. The second method takes in a String and then converts each character of the string into their 8-bit ASCII representation, and then calls the encoding method. The encoding method takes its input and splits it into groups of eight binary values. Then for each eight binary values, the method converts the binary value into Boolean values, and then create the parity bits using Java's built-in XOR operator. Then the method places the parity bits in the correct spots. Finally, the method groups each encoded character into one long binary string.

The Decoder class has one main method which takes in a binary string and splits the string into groups of thirteen binary values. Similarly to the Encoder class, the method then converts the binary values to Boolean values and the checking bits are calculated. Then if the overall even checking bit is flagged, I use the rest of the checking bits to calculate the position of the error and fix it. If there is a contradiction detected, I mark the contradiction and then move onto the next thirteen bits.

The Launch class is the class that runs the UI. It has its own instance of the Encoder class and Decoder class so that it can feed the user input to the those classes as needed.

5 Future Improvements

The main thing I would want to work on would be improving the layout of the UI. The UI is not fancy or aesthetically pleasing, just functional. In addition to that, I would want to improve on giving feedback for the result of the decoding output. Right now it tells which which character had a single bit error, but it does not tell you which bit was flipped, which might be useful for testing. One last improvement that I could do would be expanding beyond SECDED, which would involve adding more parity bits, though that would raise the overhead in space usage.

6 Images

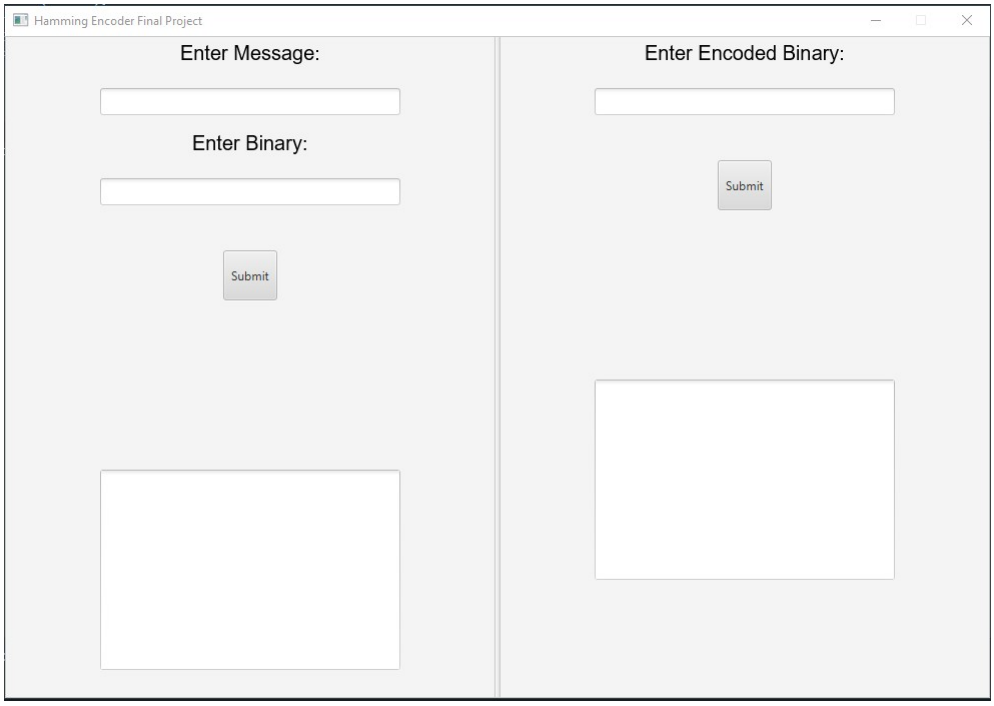

Figure 1: UI of the Hamming Encoder Project. Left side is the Encoder side, right side is the Decoder side.

```java
public String decodeMessage(String binary){
    singleErrorsLoc.clear();
    doubErrorLoc.clear();
    StringBuilder message = new StringBuilder();
    for (int i = 0; i < binary.length(); i+=13){
        String charBin = decodeCharacter(binary.substring(i, i+13).toCharArray(), i/13);
        message.append(charBin);
    }
    String singleErr = "Number of Single Errors Detected: " + singleErrorsLoc.size();
    String errLoc = "Single Errors Detected at the Following Characters: " + singleErrorsLoc.toString();
    String doubErr = "Number of Multiple Errors Detected: " + doubErrorLoc.size();
    String errLocation = "Multiple Errors Detected at the Following Characters: " + doubErrorLoc.toString();
    String finalMessage = message.toString() + "\n" + singleErr + "\n" + errLoc + "\n" + doubErr + "\n" + errLocation;
    return finalMessage;
}
```

Figure 2: Main decoder method.

```java
private String encodeMessage(String message) {
    StringBuilder encodedMessage = new StringBuilder();
    for (int i = 0; i < message.length(); i+=8){
        int endInd = (i+8>message.length()) ? message.length() : i+8;
        String binaryChar = message.substring(i, endInd);
        String encodedChar = hammingCode(binaryChar);
        encodedMessage.append(encodedChar);
    }
    return encodedMessage.toString();
}

public String encodeString(String message){
    String binaryMessage = convertToBinary(message);
    return encodeMessage(binaryMessage);
}

public String encodeBinary(String binary){
    return encodeMessage(binary);
}
```

Figure 3: Main encoder methods.