ECE 350 Final Project Report

Hamming Encoder/Decoder

Hsingchih Tang, ht114

April 23, 2020

1 Project Overview

This project is a SECDED hamming encoder/decoder program built in Java. The encoding function of the tool applies the generic single-error-correction (SEC) hamming encoding algorithm to encode a sequence of binary digits, with an additional overall parity bit added to the beginning of the encoded sequence (i.e. index 0) to achieve double-error-detection (DED). The decoding function is able to validate the input encoded binary sequence: if an error is detected, the corrected decoded sequence will be displayed, and a dialogue will pop up to inform user about the corrupted bit location; if double-error is detected, the program will pop up a dialogue to alert user of the error, and still display the decoded binary sequence after trimming off the parity bits.

The SECDED encoding algorithm requires adding k parity bits for m data bits, such that $2^k \geq m + k + 1$. The encoded sequence will therefore have a length of $m + k + 1$, since an additional overall parity bit will also be added. While the maximum number of data bits that 31 parity bits could accomodate is $m = 2^{31} - 32$, the length of its encoding will be $2^{31} - 32 + 31 + 1 = 2^{31}$, which exceeds the maximum array length allowed in Java by exactly one bit. Thus, the longest binary sequence that the tool could encode should be $2^{31} - 33$ (i.e. Integer.MAX_VALUE - 32), and the longest sequence that this program is expected to be able to decode is $2^{31} - 1$. Hence, I set the maximum input length as $2^{31} - 33$, and designed the program to reject new character entries once this limit is reached.

I have built a graphical user interface through which the user could enter a binary sequence and perform hamming encoding and decoding, and besides the basic error detection and correction functionalities, the program is also able to load inputs as well as to save output results from/to files in .txt format. If the input sequence contains invalid characters (anything that is not 0 or 1), the program will also reject the input and pop up an alert dialogue to the user.

Code for this project is kept in this github repository.

2 Design, Specifications and Code

The entire project consists of two modules, **view** and **coder**, respectively containing source code for the front end and the back end as well as Exception classes that I speicifically created for handling bit corruptions and invalid input formats.

2.1 view

Layout of the GUI is configured by view.fxml, which could be found under the resources directory inside this module. When user ticks the "encode" or "decode" checkbox on the GUI, corresponding functions (*setEncode()*, *setDecode()*) in the **ViewController** class will be invoked to update the Controller's status. When the OK button is pressed, *ViewController.run()* will be invoked, which then passes the input binary sequence to the **HammingCoder** class on the back end, which will process the input and return a new binary sequence as the output result. Similarly, when the "Upload file" or "Save to file" buttons are pressed for input loading and output saving, functions *invokeLoadFile()* and *invokeSaveFile()* will be triggered.

To handle bit corruptions, in the decoding mode, prior to calling the *HammingCoder.decode* function, the **ViewController** also passes input into *HammingCoder.validateCode*. This function would throw corresponding Exceptions if the input is not a binary sequence, or if any single-/double-bit error is detected. If the corruption occurs on a single bit, the corrupted bit index will also be included in the Exception's message field, so that this information could be extracted and displayed on the GUI. The input will also be corrected by flipping the corrupted bit; then the corrected version will be passed into *HammingCoder.decode* to retrieve the accurate decoding result.

2.2 coder

This module contains two classes, **BinaryCode** and **HammingCoder**.

BinaryCode

This class is a wrapper for binary sequence inputs. The class takes in either a Collection of boolean values or a single String through the constructor, and stores the 1s and 0s in its own private field as a list of boolean values. If the string passed in contains any invalid bits, the constructor would throw an **InvalidInputFormatException** to the caller, which will eventually be received by **ViewController** and trigger the alert dialogue on the GUI.

HammingCoder

- **Parity bit coverage mapping**
  This class keeps a mapping between parity bit indices (powers of two, e.g. 1, 2, 4, ...) and their corresponding index coverage in the encoding. The coverage of a parity bit at index 2^k are all the indices whose binary format has value=1 at the $(k+1)^{th}$ least significant bit (e.g.

the parity bit at index 1 covers indices 1, 3, 5, ...). The map is expanded every time the class receives a new input that requires more parity bits than already recorded by the map.

- **Encoding**
  To encode a binary sequence, the function initializes a new BinaryCode object and feeds in bits in the original input sequentially, and inserts 0s at the indices of parity bits. Then it loops through all the parity bits, looks up the map for the corresponding index coverages, and determines the parity bit values based on even parity principle. After all the ordinary parity bits (i.e. all parity bits except for the one for overall parity) have been set, it loops through the entire encoded sequence again, and determines the value for the overall parity bit. The overall parity bit is then placed at the beginning (i.e. index 0) of the sequence.

- **Decoding**
  The decoding process is much simpler – the function simply trims off all the bits at the known parity bit indices.

- **Corruption Detection**
  To identify corruption(s) in an input, two helper functions, *detectSingleError* and *validateOverallParity*, are implemented. *validateOverallParity* returns a boolean value to indicate whether the overall parity test on bit 0 has passed. And *detectSingleError* examines every ordinary parity bit and identifies the type and location of corruption. It firstly initializes a list of integers from 0 to the maximum index in the input, and a boolean flag to signal any test failure, and then loops through all the ordinary parity bits. If a parity bit passes the test, all indices covered by this bit will be removed from the list; if any test fails, the flag will be set to true, and the list will be shrinked to the intersection of itself and the coverage of this failed parity bit. By design of Hamming Coding, when only one bit is corrupted, there must be exactly one index left in the list at the end of the loop, which will be the corrupted location; when more than one bits are corrupted, no element will be left in the list. *detectSingleError* returns the corrupted index when the single error bit could be identified, returns -2 when the failure flag is raised while no element is left in the list (i.e. at least two bits have been corrupted), and returns -1 if all parity tests have passed (i.e. no corruption).

  Corruption situations are then determined by the following logic:

  - *detectSingleError* detects a single-bit corruption + *validateOverallParity* fails
    The bit at the index returned by *detectSingleError* is corrupted . **HammingCoder** will throw a **SingleBitErrorException** and include the corrupted index in the exception message.
  - *detectSingleError* detects no corruption + *validateOverallParity* fails
    The bit at index 0 is corrupted. A **SingleBitErrorException** will be thrown, and "0" will be included in the exception message.

- *detectSingleError* detects a single-bit corruption + *validateOverallParity* succeeds
  More than one bits are corrupted. A **DoubleBitErrorException** will be thrown.
- *detectSingleError* detects multiple-bit corruption
  More than one bits are corrupted, regardless of the result returned by *validateOverall-Parity*. A **DoubleBitErrorException** will be thrown.
- *detectSingleError* detects no corruption + *validateOverallParity* succeeds
  No bit is corrupted.

3 Challenges

The biggest challenge I've encountered was to translate the corruption detection logic into the program. I am aware that there exists a matrix approach for constructing Hamming codes, which would be easier to implement but might work better in a different programming language. As I am choosing Java in order to better structure the project and make the GUI development easier, I decided to not implement the matrix algorithm.
In order to overcome the challenge, I've used the **BinaryCode** class as a wrapper to replace the digits with boolean values, which would make the inputs easier to process. The encoding, decoding and validation functions in **HammingCoder** are broken down into smaller substeps executed in helper methods to make the code more structured and easier to understand. In addition, I've also taken advantage of the **Exception** classes to communicate about corruptions and specific bit indices between the back end and the front end.

4 Testing

In order to ensure that the program could consistently output correct results, I have developed a few JUnit tests for **HammingCoder** and **BinaryCode**. In each test, I iterate for 10,000 times, generate random inputs in every iteration, and verify that the expected outputs are returned by the method under testing.
For encoding and decoding, a random binary sequence is generated in each iteration and passed through the *encode* and *decode* functions. The test then asserts that the original sequence could be recovered.
For single-bit corruption detection, a random binary sequence is again generated and encoded in each iteration. One randomly chosen bit in the encoding gets flipped, and the corrupted encoding is passed into *validateCode*. The test then asserts that a **SingleBitErrorException** has been thrown, and that the correct index value is contained in the exception message. Double-bit corruption detection is tested simlarly, except that two randomly chosen bits are flipped before calling *validateCode*, and the test then asserts that a **DoubleBitErrorException** has been thrown.

5 Additional Features

In my original project proposal, I set the graphical user interface as an extension goal, and was also suggested by my TA to implement the file loading/saving feature as another extension, and so far I have succeesfully accomplished both. Given extra time, I might continue to explore the matrix approach and implement a different algorithm for Hamming Coding, probably in a different programming language such as Python or MATLAB.

6 Images

Figure 1: GUI

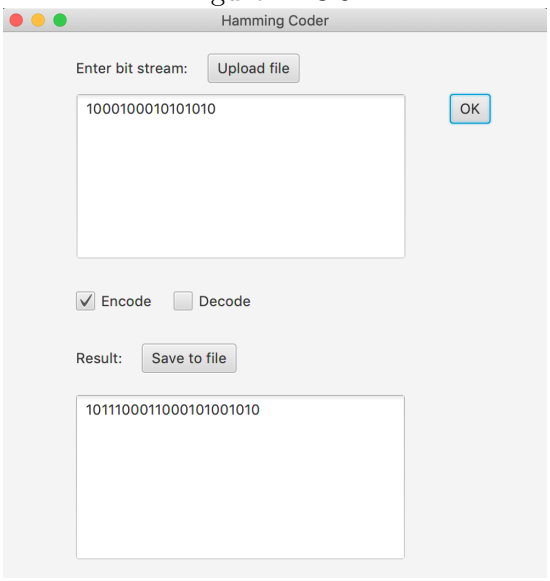

Figure 2: Corruption Detection