EE450 - Programming Assignment 1

Junzhe Liu / 2270250947

In this assignment, we mainly dealt with Cyclic Redundancy Check algorithm at both transmitter side and the receiver side, and we also compared the performances with one-byte checksum algorithm under the circumstances when a random error occurs and flips some of the bits of the codeword sequence. In conclusion, CRC shows a rather high F-1 score (0.9767 in the given testing dataset) and performs better in detecting errors.

Contents of code

1.crc_tx

In this part, we conduct CRC algorithm on the transmitter side: that is, for a given code word D and a generator G, we'll compute the CRC sequence (the remainer R of D divided by G), and append it at the end of data sequence D to form a final transmission codeword D'=D+R.

The most important function in the program is "binary_divide", it conducts one intermediate step in the whole division and computes the remainer R. Because in the division each step the dividend will be changed to the current Remainer R appended with the next bit of the current dividend, therefore we can program this process as a loop. In

each iteration of the loop, function “binary_divide” will be called and computes the remainer R, and then R is concatenated with the next bit of D.

As a result, crc_tx will print out the final codeword (which is D’) and the CRC sequence (which is R) for each testing example.

2. crc_rx

This part CRC algorithm is conducted on the receiver side to determine if a received codeword D is correct or not. This is performed by dividing the codeword D with the generator G, and check if the remainer R is an all-zero sequence. If yes, then the codeword is accepted (but it still could have some errors), otherwise CRC algorithm will not pass this codeword.

Apparently, this program only needs some small changes in the crc_tx.cpp. The most import part is that we need to check if a string is an all-zero sequence, otherwise there exist at least an “1” at some certain position. To implement this function, I used the string.find() method to look for character “1” in the sequence, if it could not find any “1” in the sequence, then the algorithm will identify this sequence as an all-zero string.

```
size_t found = R.find("1");
if (found==std::string::npos)   // Did not find any "1" in
 the sequence
    cout<<"pass"<<endl;
```

```
else cout<<"Not pass"<<endl;
```

As a result, crc_rx will print “Not pass” for the failed codewords D, and “pass” for the passed ones.

3. crc_vs_checksum

In this proportion we test the CRC’s ability of detecting errors in sequences and compared it with 1-byte checksum algorithm. This program mainly contains 3 parts: 1.compute CRC; 2.compute 1-byte checksum; 3.compare their performances on the error sequences.

In part 1, function “binary_divide” and “CRC” are used to perform single step division, and compute the final codeword respectively. Part 2 is comprised of several functions: “Bin2Dec” computes a decimal number and returns an integer of the given binary string; “Dec2Bin” does the exact opposite thing: it returns a binary sequence of a decimal number smaller than 255; function"checksum" computes the overall checksum of a string according to 1-byte checksum algorithm; function “one_byte_checksum” computes the checksum and returns the final codeword D’. In part 3, we read in data D, generator G, and random error sequence T from the file, and perform CRC and 1-byte checksum algorithm respectively on the data D. Then we’ll need to identify if T is an all-zero sequence, if it is, then we don’t need to perform any transformation on both codewords, True Positive and False Negative will be computed in this situation; Otherwise, we’ll flip the bits of both codewords according to the random error sequence, and then check again if the error codewords passes the check, if it passes, False

Positive will increase. F-1 score can be computed given TP, FN, and FP.

4. makefile

makefile is used to compile and run the above cpp programs:

```
/* Makefile */
all: crc_tx.cpp crc_rx.cpp crc_vs_checksum.cpp
    g++ -o crc_tx.out crc_tx.cpp
    g++ -o crc_rx.out crc_rx.cpp
    g++ -o crc_vs_checksum.out crc_vs_checksum.cpp

crc_tx: crc_tx.out
    ./crc_tx.out

crc_rx: crc_rx.out
    ./crc_rx.out

crc_vs_checksum: crc_vs_checksum.out
    ./crc_vs_checksum.out

clean:
    -rm crc_tx.out crc_rx.out crc_vs_checksum.out
```

Please use “make all” or “make” first to compile the files, and run the 3 programs respectively. If you need to re-compile, use “make clean” to delete all executable files.

Idiosyncrasy

There are only one restrictions: the generator sequence G should always begin with “1”, that is to say, the leftmost bit of G must be “1”. In the program, I did not consider any situation that G begins with “0”, because in real use, the coefficient of the highest power event should not be 0. If there exist some example that G begins with 0, then the answer might not be right.

Reused Code

All the codes are written by myself. I did not use any code from other resources.