Data Communications - Spring 99 - CA2

Mohammad Ali Toufighi - 810195371

In this project, we were going to implement source and channel coding for a data communication channel. Source coding was added to implement data compression, and channel coding was added for error detection and correction.

Phase One: Source Coding

For this phase, I was going to use Huffman coding to compress characters into binary digits. I needed to read the probability of occurrence of each character in English alphabet from the provided file (`freq.mat`) and create the character-to-Huffman mapping.
To read the file above, since it was in MATLAB-compatible binary format, I had to use `mat4py` library. After reading this file, I stored it as a dictionary with characters as keys, and their probability as values. (See `read_probabilities()`)
Also to create the mappings, I used a min heap (Python's `heapq` implementation) to get the minimum values as fast as possible. (See `create_huffman_mappings()`)

1.1. Huffman Encoding:

I used the above function to create the character mappings. Then, to encode strings, I simply used this mapping to find the equivalent Huffman value of each character and appended the Huffman-mapped value to the resulting binary string. (See `HuffmanEncoder::encode()`)

1.2. Huffman Decoding:

In this section I also used the aforementioned `create_huffman_mappings()`, but I reversed its key-values to have Huffman code to character mapping. Doing this was fine, as Huffman codes are unique. To decode, as these coded values are Prefix-Free, I followed the substring bit-by-bit to find a mapping, and added the corresponding character to the resulting string. (See `HuffmanDecoder::decode()`)

Phase Two: Channel Coding

For this phase, I used Convolutional Encoding to encode compressed bits, and Viterbi Decoder to decode bits received from the channel. The mapping used here is the one present in the project description.

2.1. Convolutional Encoder:

I simply implemented the State Machine. When a new bit is seen in the stream, I looked at the current state (stored in `shift_register` variable), and using the provided mapping I chose what should be sent to the channel. (See `ConvolutionalEncoder::encode()`)

2.2. Viterbi Decoder:

For me, this was the most challenging part of the project. Here are the steps I followed to do the decoding part:

- Set initial Path Metric values to zero for state `00`, and infinity for other states. (Because we always start from `00`)
- While there are any bits in the stream, select two consecutive bits from it.
- For each state, find the outgoing edges (By prepending `1` and `0` to the state index, similar to the shift register idea)
- For each outgoing edge, find the Branch Metric, and store the possible Path Metric from this edge for the destination state.
- For each state, get the minimum possible Path Metric value and store it as the new Path Metric of this state.
- Find the overall minimum of Path Metrics in this stage of time, and append the corresponding edge value to the resulting binary string.

`(See ViterbiDecoder::decode())`

Phase Three: Noise

As I was using strings to store binary representations, I slightly changed the provided `noise()` function to be compatible with strings. I also reduced the probability of noise happening to `0.01` to see how output is close to the initial string. The result was satisfying and while noise probability is not little that much, our error detection mechanism helped the pipeline to correct to the original values.

Source code is also available in my GitHub repository.