Qintong Wu
8421493946
qintongw@usc.edu

Huffman Compression

EE565: Project

2018-04-23

1 Abstract

In this project, the Huffman encoding algorithm is implemented to compress the given text. Correspondingly, the decoding algorithm can decode the compressed file according to the generated mapping file in the encoding step. The average code length is 4.65 and The compression ratio is about 58%.

2 Algorithm

The core part is to generate a dictionary of codewords for probability distribution, then assign associated symbols to the codewords. To minimize the average code length, the shortest codeword should be linked to highest probability and vice versa.

The following describes the dictionary generating process.

```
Data: given text
Result: compressed file named compressed.txt and codewords mapping file CodewordsMap.mat
initialization: count the probability for each symbol, then filter out the zero probability symbol;
while not yet finishing loop all symbols do
    1. sort probabilities in ascending order
    2. pick up the first two then sum up as new probabilty
    3. also pick up the first two symbols then combine together
    4. store the new probability and symbols
end
while not yet finishing loop all symbols do
    1. get the last symbol group which contains the all symbols that not finishing assigning codewords
    2. append either '0' or '1' to different symbols under the same group
end
re-sort the codewords using initialization information to minimize the average codeword length;
generate map object that contains codeword-to-symbol mapping;
```

Algorithm 1: Dictionary generating process

After mapping obtained, the encoding and decoding processes can be done quickly.

During encoding process, the index of each character in sorted symbol vector is found then used to locate the right codewords generated before. Codeword will be appended to output binary string sequentially. Once encoding process done, the whole binary string are encoded in **uint16** format.

For decoding process, we could easily decode the binary string due to the instantaneous code feature. That is, binary character is sequentially read in and checked whether the current string is in the map or not; if so, the corresponding symbol will be appended to the result string and clear the current string, or keep getting more binary character in otherwise.

3 Miscellaneous and Discussion

During the experiment, I found that compression ratio is heavily depend on the saving file format. For example, if raw binary string are printed in the file, then the file size will be around 2 megabytes, which is not acceptable for it's even bigger! The reason why this happens is that raw string is stored as **char** format, and this solution is not efficient at all because each binary character takes 8bits space. Thus, I wrote two helper function **hexer** and **dehexer** to save raw string in **uint16** format and get them back to raw string after reading the compressed file. By doing so, the file size is scaled down dramatically.

One more thing to notice, if **uint8**, aka **uchar**, is used, the compressed file size is about the same as the original file size. This fact indicates that the compression ratio can be further improved by squeezing more bits together. Once the best average code length is achieved, the compression ratio is depend on the file format.

4 How to run it

A demo file *main.m* is given. The procedure is also given below:

1. specify the original file path
2. use *encoder* function to compute the compressed data **comp** and mapping **map**, average codeword length can also be obtained but not necessary for the decoding process
3. save the compressed data in **uint16** format and dump the mapping file **map**
4. load the compressed data in **uint16** format and the mapping file **map**
5. use *decoder* function to decode the compressed data with the mapping, the output will be the original text
6. dump the text, maybe compared to the original text

It takes 25 seconds to run in my computer.

5 Conclusion

The Huffman encoder and decoder are implemented in this project and it achieves the best average codeword length (compared to the built-in Huffman function).

The file list is given below:

- get_probs.m — for initialization
- dict.m — to generate the dictionary
- encoder.m — encoding the given text
- hexer.m — further encoding compressed text in uint16 format
- dehexer.m — reverse the uint16 to the binary string
- decoder.m — decoding the binary string
- demo.m — it's just a demo
- cmd_output.txt — command line output generated by running the demo