ECE F344: Information Theory and Coding

Assignment 1 Report

Pramit Pal
2023AAPS0765P

1 Introduction

The main goal of this assignment was to implement Huffman coding (binary and non-binary), Shannon-Type coding, Shannon coding (Shannon-Fano-Elias), and Shannon-Fano coding, and compare how efficient they are.

I completed all the requirements and took it a step further by building a working file compression tool (`huffzip`). It can optionally use LZ77 sliding-window compression before applying Huffman encoding. The tool also has a verbose mode that compares the performance of all these coding schemes against the source's theoretical Shannon entropy. For all methods, the program calculates symbol probabilities directly from how often they appear in the file, treating every byte (including spaces and formatting) as a unique symbol.

2 Methodology

I wrote all the coding schemes in C++. The probability of each symbol x_i is just $P(x_i) = n_i/N$, where n_i is how many times the symbol shows up and N is the total number of symbols. All average lengths and entropies are measured in bits.

2.1 Huffman Coding

Huffman coding builds a set of prefix-free codes to keep the expected codeword length as short as possible.

1. Start with a bunch of leaf nodes, each representing a symbol and its probability.
2. Pick the D nodes with the lowest probabilities.
3. Combine them into a new parent node whose probability is the sum of its children, and put it back into the mix.
4. Repeat this until there's only one root node left. By walking down the tree, we get the codewords $(0, 1, \ldots, D-1)$ for each symbol.

This implementation can handle binary ($D = 2$), ternary ($D = 3$), and quaternary ($D = 4$) trees by changing the node arity. If the tree doesn't perfectly fill out (i.e., $(N-1) \bmod (D-1) \neq 0$), it automatically adds dummy symbols with zero probability to make it work.

2.2 Shannon Coding (Shannon-Fano-Elias)

1. Sort all symbols from highest to lowest probability.
2. Calculate the cumulative probability $F_i = \sum_{k=0}^{i-1} P(x_k)$.
3. Set the codeword length for symbol i to $l_i = \lceil -\log_2 P(x_i) \rceil$.
4. Make the codeword out of the first l_i bits of the fractional binary representation of F_i.

2.3 Shannon-Fano Coding

This uses the classic top-down splitting approach:

1. Sort the symbols by probability, from highest to lowest.
2. Split them into two groups (left and right) so that the total probability of each group is as close as possible.
3. Add a `0` to the codes for the left group and a `1` to the right group.
4. Repeat this splitting process for each group until every symbol is on its own.

2.4 LZ77 Extension (Optional)

Before coding, a sliding-window algorithm scans the data to find repeating patterns. It replaces these repeats with `(length, distance)` pairs pointing to the previous occurrence. The final output mixes these pairs with raw data for things it hasn't seen before.

3 Empirical Results

I got the following numbers by running the `huffzip.exe` program on some test files. It calculates the entropy H, average codeword lengths L_{avg}, and efficiency ($\eta = H/L_{\text{avg}}$) on the fly.

3.1 Fixed-Distribution Source: `aaaabcdefg`

This 10-byte test file matches the 7-symbol alphabet from the assignment ($P('a') = 0.4, P_{\text{other}} = 0.1$). I ran it with: `.\huffzip --huffman -v test_alpha.txt out.tmp`.

```
------------------------------------------------------
  Source statistics
------------------------------------------------------
  Symbols (unique / total)  : 7 / 10
  Shannon entropy           : 2.5219 bits/symbol
------------------------------------------------------
  Coding scheme comparison (source bytes)
------------------------------------------------------
  Scheme                  Avg len  Efficiency
------------------------------------------------------
  Shannon                  3.2000    78.8103%
  Shannon-Fano             2.7000    93.4047%
  Huffman (binary)         2.6000    96.9972%
  Huffman (ternary)        1.6000    99.4475%  (base-3 symbols)
  Huffman (quaternary)     1.4000    90.0689%  (base-4 symbols)
------------------------------------------------------
  Actual compression
------------------------------------------------------
  Mode                      : Huffman-only
  Avg token code length     : 2.6000 bits
  Compressed size           : 1174 bytes
  Uncompressed size         : 10 bytes
  Compression ratio         : 117.4000
------------------------------------------------------
```

3.2 Application Source Evaluation: `main.cpp`

Here I tested the compressor on its own source file by running `.\huffzip -v src\main.cpp` (with LZ77 turned on).

```
-----------------------------------------------------
  Source statistics
-----------------------------------------------------
  Symbols (unique / total)  : 89 / 11408
  Shannon entropy           : 4.4921 bits/symbol
-----------------------------------------------------
  Coding scheme comparison (source bytes)
-----------------------------------------------------
  Scheme                 Avg len  Efficiency
-----------------------------------------------------
  Shannon                 5.0122    89.6231%
  Shannon-Fano            4.5571    98.5739%
  Huffman (binary)        4.5335    99.0866%
  Huffman (ternary)       2.8628    98.9999%  (base-3 symbols)
  Huffman (quaternary)    2.3154    97.0046%  (base-4 symbols)
-----------------------------------------------------
  Actual compression
-----------------------------------------------------
  Mode                      : LZ77 + Huffman
  Avg token code length     : 5.9297 bits
  Compressed size           : 5625 bytes
  Uncompressed size         : 11408 bytes
  Compression ratio         : 0.4931
-----------------------------------------------------
```

4 Analysis and Conclusions

Entropy Limits. The C++ file had an entropy of around 4.49 bits/symbol, which makes sense for text code. The theoretical max for 89 symbols would be 6.47 bits if they were all equally likely. For the small 10-character file, the entropy was 2.52 bits, which matches the math perfectly for those probabilities ($0.4/0.1 \times 6$).

Binary Huffman vs. Shannon-Fano. Testing on the main source file gave Binary Huffman a 99.09% efficiency, beating Shannon-Fano's 98.57%. This proves that building the tree from the bottom up (Huffman) is better than splitting it from the top down (Shannon-Fano). The difference is even clearer on the small test dataset (97.0% vs. 93.4%).

Shannon Coding Limitations. Standard Shannon coding maxed out at about 89.62% efficiency. This isn't surprising, since picking codeword lengths based just on probabilities—without balancing an actual tree—usually wastes a fraction of a bit per symbol.

N-ary Huffman Variations. The base-3 versions did really well, hitting 99.4% and 99.0% efficiency on the two files. However, the base-4 (quaternary) version dropped to 90.0% on the small test file. This happens because we had to pad the tree with dummy filler symbols just to make the base-4 branches work, which stretches out the codes.

Note on N-ary lengths: At first glance, it might seem impossible that the ternary (1.6) and quaternary (1.4) average lengths for the test file are lower than the Shannon entropy of 2.52 bits. However, this is because these averages are measured in *trits* (base-3 digits) and *quats* (base-4 digits), not bits. To convert them directly to bits for a fair comparison, we use the scaling factors: 1 trit $= \log_2(3) \approx 1.585$ bits and 1 quat $= \log_2(4) = 2$ bits.

LZ77 Compression. The C++ test gave a compression ratio of 0.49, showing that we really need things like LZ77 for good real-world compression. Adding LZ77 bumped the average code length to 5.92 bits (up from 4.53). But because it replaced so many repeating strings, there were way fewer total symbols to encode, slicing the final file size in half. On the flip side, compressing the tiny 10-byte file bloated it up to a 117.4 ratio. This just shows that saving the Huffman tree and file headers adds a fixed overhead that kills compression on tiny files.