Assignment 1 - C++ Huffman Coding implementation and analysis about its properties

Otho Komatsu, ID - 170020142, ENE/CIC

***Abstract*—This report aims to describes a particular simple implementation approach of the Huffman Coding in C++, do an brief analysis of its implementation, in contrast with its theoretical approach, and compare its performance with commercial algorithms, as `.zip`, `tar.xz`, and `.7z`.**

***Index Terms*—compression, entropy , huffman, variable-length compression, lossless compression**

I. Introduction

Huffman coding is one important algorithm that explores the relation between the frequency of a symbol on a alphabet and its code length. A interesting property is its steps that follows a restriction rule: the two *symbols* with the least frequency has the same length *codeword*, differing in its last bit.

A. The algorithm

The Huffman Coding algorithm developed consists in 3 basic parts: `IO`, `Huffman Coding`, `Huffman Decoding`.

First, the algorithm reads a input file byte by byte, converts the byte `bitstream` into a string, and stores it inside a map as a key, associated with a value that express the byte pattern counting. Thereby, the probability table is computed, that stores the byte pattern associated with its probability in the input `bitstream` sequence.

In the second stage, the Huffman encoding takes the algorithm steps. First, it sorts all read byte symbols from the `bitstream` according to its probability, and extracts the two symbols with the least probability and groups it inside a pair, and stores it into a vector, μ. These two symbols becomes an nth α for each iteration, becoming a intermediary symbol with its probability equals the sum of the two grouped symbol's probability. We call the α symbol as father node and its grouped source symbols as children nodes from a tree. The α generated symbol is inserted inside a vector γ. This process repeats until only two symbols is left, where one assumes value "1" and the other "0". With the knowledge that each of these two lasts is the last fathers generated, *i.e.*, the algorithm go through all the tree from the root to its leafs, iterating each parent node in γ from the most recently generated to the oldest and its respective children nodes in μ. Each iteration then concatenates the fathers code to each children, that represents the symbols generated from the code(original alphabet symbols or α). As each children is also a father, all symbols gets its own symbols according to its father node(α symbol). With the code generated, each code and its respective symbol is stored inside a map. With the Huffman code, the original file is read byte by byte again and its content is converted to its corresponding code, written in a buffer with the encoded content. After this process, the compressed file is created. First, a header is added. The first 2 bytes is the number of symbols from the alphabet, then for each symbol the following pattern is written $< \pi, size, \lambda >$, with π as 1 byte symbol, $size$ the corresponding code size, and λ the code to the symbol. Following, the encoding content is inserted.

Finally at the third stage, the Huffman decoder gets reads the compressed file following the pattern described above. All the content is stored inside a map. Then, the compressed file content is read, matching the pattern for the code stored inside the map. For each match, the corresponding original symbol is written in the decompressed content buffer. After all the decoding, the decompressed content, which is the original content file, is written in a file, recovering the original file compressed.

II. Simulation Results

Many test files were used to measure the performance of the Huffman coding algorithm implementation from this project by studying their compression rate. These include, common text files from literature books(`dom_casmurro.txt`), code files(TEncEntropy and TEncSeach), a random sequence of characters(fonte, fonte0 and fonte1), and a image(lena.bmp)

To measure the Huffman coding efficiency, entropy value were computed to the test file's input alphabet extracted from them, as well as the resulting encoding average size length, as depicted at table I. The difference is well highlighted with the relative difference vallue, computed as the fraction between the difference obtained from the entropy and average size, and the signal entropy theoretically expected.

As we can observe in table I, the signal entropy and average size length are very close, with a minimum difference on test the tests of 0.265% and maximum of 6.52%. Actually, this difference respects the theoretical Huffman performance [1]:

$$H(S) \leq R \leq H(S) + 1$$

In order to understand the Huffman encoding performance compared to other kinds of compression algorithms, these sample test files were compressed by another 3 different commercial algorithms, used routinely to pack files and projects: 7zip, Gzip, Zip(based on Deflate algorithm).

By this experiment, 2 measures were computed and gathered for each algorithm listed: ν, the compressed file size in kiloBytes, and ξ, the compression rate given by the following equation:

TABLE I
HUFFMAN COMPRESSION ALGORITHM PERFORMANCE

	Signal Entropy	Average size length	Relative difference
dom_casmurro.txt	4.64295	4.68055	0.81%
fonte.txt	2.89398	2.94954	1.9%
fonte0.txt	1.58349	1.668	5.34%
fonte1.txt	3.30326	3.32634	0.70%
lena.bmp	7.50972	7.52963	0.265%
TEncEntropy.txt	5.27191	5.30785	0.681%
TEncSearch.txt	5.15808	5.19171	6.52%

TABLE II
COMPRESSION ALGORITHMS RATES COMPARISONS:

Files	**Original file size**	**Huffman**			**7zip**		**gzip**		**ZIP**	
		λ	ν	ξ	ν	ξ	ν	ξ	ν	ξ
dom_casmurro.txt	389.6 kB	227.9	228.3	41.40%	131.1	66.35%	153.6	60.57%	153.8	60.52%
fonte.txt	1 MB	368.6	368.7	63.13%	392.7	60.73%	428.3	57.17%	428.5	57.15%
fonte0.txt	1 kB	0.208	0.222	77.78%	0.470	53%	0.322	67.80%	0.463	53.70%
fonte1.txt	1 MB	415.8	415.8	58.42%	433.6	56.64%	489.7	51.03%	489.8	51.02%
lena.bmp	263.3 kB	247.8	248.6	5.57%	182.5	30.70%	227.6	13.56%	227.7	13.52%
TEncEntropy.txt	19.4 kB	12.9	13.1	32.33%	4.1	78.87%	4.2	78.35%	4.4	77.32%
TEncSearch.txt	253.0 kB	164.2	164.5	34.99%	31.9	87.39%	38.6	84.74%	38.7	84.70%

***where:** λ is the compressed file size without header(bits), meanwhile ν is the size with header(Kilobytes), ξ compression rate(around the compressed file with header)

$$\xi = 1 - \frac{l}{L}$$

Where l is the compressed file size(including the header) in bytes, and L the original file size also given in bytes.

Note that we only have access to the compressed file size without header from the Huffman code, as this information is easily obtained from the project variables data. In contrast, once the other algorithms used were from a already made program, this information is not well available to observe, hence not included in the table II.

By analysing these results, it is possible to note that the Huffman performance is very variable according to the test file content pattern. In some tests, Huffman is comparable to the commercial algorithms, as in the case of `fonte.txt`, `fonte1.txt`. As well, a very bellow perform in `dom_casmurro.txt`, lena.bmp, `TEncEntropy.txt`, and `TEncSearch.txt`. And finally, a surpassing result only in `fonte0.txt` file.

III. CONCLUSION

By the results, the Huffman coding presents a very close average size length to the entropy, although the compression performance is not the best possible to reach. Furthermore, the values gathered from the experiment shows the lower and upper bound that the Huffman Coding could reach

By the comparison showed in II and analysed above, its notable that the Huffman Code presents a good performance when compress a random input alphabet and limited quantity of symbols. But in contrast, we can see a poor performance in files where the input symbols contemplates a low range of characters, and the probability is extremely unbalanced between the characters. It's most common in files that have a syntax and own semantic, for example a text or a code file. Or there's some pattern that the file describes about something. That's why the files dom_casmurro, TEncEntropy.txt, TEnctSearch.txt, which fits i the first group mentioned; and lena.bmp, in the second group, presents a bad general performance, according to its compression rates.

If we follow the examples of the commercial algorithms showed in the comparison, most of them deals with the dependency of grouped characters and explores it to represent redundancies in the file with less bits, what the tradicional Huffman Coding algorithm implemented not explores.

In this sense, a interest optimization that could happen in the current Huffman coding implementation is explores the possibility of grouping symbols in fixed size blocks, and uses it as an symbol from the alphabet. This allows the algorithm to deal with consecutive symbols that shares a dependency on each other, as explored by the commercial algorithms. Thus, redundancy can be expressed with a more compressed way.

An another interest optimization would be adapt the Huffman coding algorithm tree building mechanism, so that it would generate a minimum variance Huffman coding, and constructs a canonical tree, reducing the header overhead by sending only one number(the size of the code, sending the symbols following its lexicographical order) instead of variable size tuples of three values.

ACKNOWLEDGMENT

The `Bitstream` read and write operations to read and write and manipulate the simulation test files comes from a imported library made by the discipline's professor Eduardo Peixoto. The author would like to thank for the library code provided.

References

[1] K. Sayood, Introduction to Data Compression. Elsevier - Morgan Kaufmann, 2012.