Assignment 2 - LZ77 Algorithm and analysis

Otho Komatsu, ID - 170020142, ENE/CIC

***Abstract*—This report describes a simple C++ project approach implementation of LZ77 compression algorithm, and assess it performance on sample files. Upon this results, an analysis over the implementation is made to explain its perfomance behaviour, and then compared to others commercial algorithms.**

***Index Terms*—compression, entropy , lz77, variable-length compression, lossless compression**

I. Introduction

Dictionary coders are a a class of lossless data compression algorithms which is mainly based on storing a list(**a dictionary**) of frequently occurring symbols patterns. Then, the coding process is based on references over these patterns. A particular algorithm that is studied in this report is the LZ77 approach, which essentially deals the dictionary as parts of the sequences already encoded. Basically the process of matching is computed over a *sliding window*, which contains two buffers: the *search buffer* and *lookahead buffer*. When the match is computed, the algorithm then send a triple to the pattern encoded: $< o, l, c >$, where o stands for offset; l, length; and c to the codeword that follows the pattern matched. If no pattern is found, we send only the first character one the lookahead buffer.

The following description is a particular approach over this general description above, pointing only its peculiarities and disctints implementations choices.

A. The algorithm

LZ77 algorithm developed is simply separated in two parts: its **Encoder** and **Decoder**.

Before the encoding process, the file to be compressed is read and then stored in a content buffer. From this buffer, the encoder starts to read and make its triples. For each character read from the content buffer, 3 major process occurs: match pattern at search buffer, update on search buffer tree, and triple making.

In the match pattern step, the algorithm seeks within the search buffer tree the node with the largest match pattern that exists with the look ahead buffer. With the best match, the offset and length are computed. Its important to mention that using due to the C++ STL library intern operations, when a match tie occurs in the tree search, the algorithm selects the least recently node inserted, that is, with the larger offset.

In the updating step, a pattern sequence from the content buffer is stored, with its size equals to look ahead size and its sequence starting from the read character. This pattern is then inserted on a binary search tree that represents the search buffer. Its size is determined by the search buffer size.

After all this computing on the current look ahead buffer read, the offset and length computed, as well as the symbol that follows the match pattern, are stored as triples, and then all this process repeats, until all characters from the buffer is read.

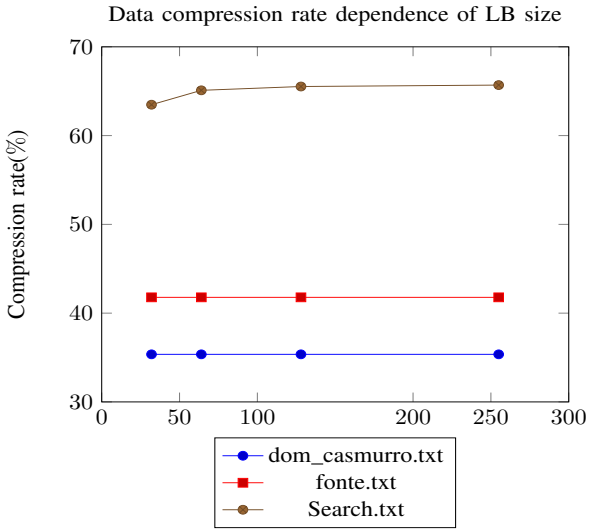

Fig. 1. Data compression rate x LB size

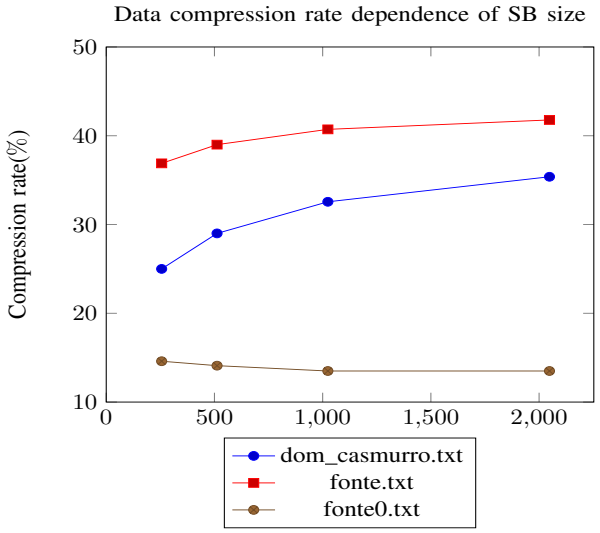

Fig. 2. Data compression rate x SB size

At this point, we have a list of all triples that represents the original file content. As a project design choice from the author, the offset and length were encoded using the Huffman algorithm, and the third member from the triple sent as a byte(1B) character. Finally, the compressed file is written with the header and the content. The header contains:

TABLE I
TRIPLE SIGNAL ENTROPY

Files	Offset			Length			Codeword		
	Entropy	**Average**	**Difference**	**Entropy**	**Average**	**Difference**	**Entropy**	**Average**	**Difference**
dom_casmurro.txt	10.9528	10.9694	1.5%	2.71117	2.75412	1.559%	5.0613	5.08389	4.42%
fonte.txt	10.993	11.0004	0.06%	2.04709	2.12848	3.82%	3.18012	3.23783	1.78%
fonte0.txt	6.8343	6.89404	0.86%	3.40474	3.43709	0.94%	2.27559	2.28477	0.4%
fonte1.txt	10.9939	11.0004	0.06%	2.17638	2.25205	3.36%	3.92439	3.95587	0.795%
TEncEntropy.txt	9.79425	9.81501	0.21%	4.0421	4.07909	0.906%	5.7328	5.76363	0.534%
TEncSearch.txt	10.69	10.7178	0.543%	4.24599	4.28231	0.848%	5.80999	5.84151	0.539%

TABLE II
COMPRESSION ALGORITHMS RATES COMPARISONS

Files	LZ77	Huffman	7zip
	ξ	ξ	ξ
dom_casmurro.txt	35.39%	41.40%	66.35%
fonte.txt	40.72%	63.13%	60.73%
fonte0.txt	13.5%	77.78%	53%
fonte1.txt	35.16%	58.42%	56.64%
TEncEntropy.txt	41.24%	32.33%	78.87%
TEncSearch.txt	65.7%	34.99%	87.39%

***where:** ξ compression rate(around the compressed file with header)

the meta values for the offset encoding followed by the meta values of the length encoding. For each one, the format is the same: number of codes(2B), followed by a list of symbol(2B), encode size, and corresponding code. Then the rest of the compressed file is filled up with the encoded triple values.

The second part is the decoder. It first reads the compressed file and stores its content in a buffer. After this, the buffer is read according to the described pattern, parsing the header and constructing the offset and length symbol table. With the offset and length's code and symbol corresponding table, the triples are read and decoded. With the original triple, the decompressed file is reconstructed just replicating the match form the triple. This is repeated until all triples is read, in other words, all content is recovered from the decompression. Then, we have finally the original file.

II. SIMULATION RESULTS

To assess the LZ77 algorithm implementation performance, a set of sample files were used as test, measuring the following values on analysis: signal entropy and compression rate, varying the value of search buffer(SB) size and look ahead buffer(LB) size; and to its best performance value of buffer and look ahead buffers size.

First, we'll study the algorithm behaviour varying the parameters SB and LB size. The first analysis is relate the compression rate to the variation of the SB, as we can observe in the figure 1. At this analysis, the search buffer size varies with the look ahead size fixed at 255. Here, we see that look ahead buffer size doesn't improve significantly the algorithm's compression rate.

Similarly, the second analysis observes the relation between the compression rate and LB size as shown in the figure 2. In this plot, the SB is fixed at 2048. Differently, the Search Buffer impacts significantly in the algorithm performance, raising 10% on the compression rate.

With the above comparison, we can obtain the best SB and LB size choice. With these values, we will use them as reference and fixed values to determine the signal entropy and compression rate values.

The signal entropy analysis in the case of LZ77 algorithm, considering that we don't encode the values but send match patterns in triples whose values is independent from each other, can be interpreted as three separated analysis of each value from the triple. I.e, the analysis can be made as separated signals of offset, length and codeword values. In the case of the codeword, the average value is the theoretical Huffman code performance applied to this field. Thereby, with these values we obtain the values summarised at table I. With these results, its clear that the Huffman code is a proper model to these two values, although the codeword is sent as byte.

The Offset value, on the other hand, have a high entropy value, independent of the chosen encoding to this value. It is, the implemented algorithm presents a defect approach that provokes these bad performance on the offset computation to the matches.

For the compression rate analysis, we will have the compression rate for each sample file and compare with 2 other algorithms: Huffman code(from the 1st assignment), and 7zip, a commercial coder. The results a summarised at table II. The expected result from this algorithm performance is that its compression rate is between Huffman code and Deflate algorithm(7zip). As we can see, the results pattern doesn't meet the theoretical ratio, a result that indicates the program's flawed implementation. The only results that presents a good performance are the codes files TEncEntropy.txt and TEncSearch.txt, which syntax and files content structure have a strict and simple pattern, taking advantage of the dictionary approach from the algorithm.

III. CONCLUSION

At the first results, we can infer that the look ahead buffer presents an improve according to an increase on its size until a specific value, between 32 and 64 as we can see in the case of the file Search.txt. Following that value, the compression performance doesn't change. What indicates that large patterns doesn't happens frequently, but a certain large amount of medium partterns. Futhermore, a 1 byte value to the lookahead buffer size, that limitates the length value transmited on the

triple(0 to 255), is sufficient to present a performance sufficient and that doesn't promotes a unnecessary overhead.

In contrast, the Search Buffer have a great impact on the compression performance. Its rate jump from 25 to 30% to the dom_casmurro file, and 36 to 41% on the fonte.txt. Its expected behaviour, since the greater the Search buffer more probability to find bigger matches to the look ahead sequence.

Now, the entropy values is the main concern to the algorithm performance. As mentioned, the codeword are sent as a 1 byte symbol. Comparing with its entropy, which values are mostly inferior to 1 byte, we can assume that in most cases there's a proper encoder to compress the symbol representation. What we refer to proper is, in the case, the encoder algorithms whose design model is the best fit to the symbols occurence distribution. That is, depending on the symbols histogram, a appropriate algorithm is applied to its curve.

Thereby, an possible improvement would be a algorithm's heuristic that deduce the best encoder model that fits to the occurence distribution of the symbols. With this approach, we could have a approximately gain of 3 to 6 bits per symbol as we can see in the codeword's entropy. However, the coder would need to take into account the header overhead that could prejudice the bits saving on the coder choice.

Another problem detected is the bad offset entropy value. We can remember that in the algorithm tie case approach, due to the STL library tree data structure's behaviour; the chosen match is the one that is the most distant from the compared sequence, that is, that which presents the largest value of offset. We take in example the following sequence:

$$cabracadabrarrarrad$$

In this sequence, instead of sending:

$$[< 0,0,c >< 0,0,a >< 0,0,b >< 0,0,r >$$
$$< 3,1,c >< 2,1,d >< 7,4,r >< 3,5,d >]$$

The above implementation sends:

$$[< 0,0,c >< 0,0,a >< 0,0,b >< 0,0,r >$$
$$< 3,1,c >< 5,1,d >< 7,4,r >< 3,5,d >]$$

We can notice the $< 5,1,d >$ triple, giving preference to the most distant pattern. This kind of behaviour is a bad practice, because this kind of choice forces the offset values to vary significantly, that produces an spread histogram do the offset values(a not good behaviour to the triple's values). With this gap, an possible improvement to the algorithm would be a fix on this issue, refining the match seek design giving preference to the nearest matches at ties cases, and in some cases, choose shortest matches but nearer over largest match but distant.

Lastly, the encoder performance compared to the Huffman's code(implemented in the 1st assignment) and the commercial algorithm Deflate 7zip, doesn't present a good and expected results. Taking into account the above problems, and the possible unnecessary overhead from the compressed file header(deriving from the Huffman code), these is the most probably causes for the bad algorithm performance, although the good results in the codes files. Therefore, those improvements are issues that worth to be explored to have a more compatible results to the theoretical model.

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.