Comparative Analysis of Lossless Text Compression Techniques

Nathanael Jacob
Department of E & TC
VIT, Pune.
Maharashtra.India-411037.

Priyanka Somvanshi
Department of E & TC
VIT, Pune.
Maharashtra.India-411037.

Rupali Tornekar
Department of E & TC
VIT, Pune.
Maharashtra.India-411037.

ABSTRACT
Data compression is an effective means for saving storage space and channel bandwidth. There are two main types of compression lossy and lossless. This paper will deal with lossless compression techniques named Huffman, Arithmetic, LZ-78 and Golomb coding. The paper attempts to do comparative analysis in terms of their compression efficiency and speed. The test files used for this include English text files, Log files, Sorted word list and geometrically distributed data text file. The implementation results of these compression algorithms suggest the efficient algorithm to be used for a certain type of file to be compressed taking into consideration both the compression ratio and speed of operation. In terms of compression ratios, Golomb is best suited for very low frequency Text files, arithmetic for moderate and high frequency. Implementation is done using MATLAB software.

General Terms
Text Compression.

Keywords
Huffman, Arithmetic, LZ-78, Golomb, compression ratio.

1. INTRODUCTION
Compression is a process of reducing the amount of data needed for storage or transmission of a given piece of information (text, graphics, video, sound, etc.) typically by use of encoding techniques. Data compression is used in a computer system to store data in a format that occupies less space than the original form. In order to effect data compression special software packages are required to compress the data and to reopen it to its original size. Data compression is characterized as either lossy or lossless. The compression process is said to be lossless if the recovered data are assured to be identical to the source; otherwise it is said to be lossy. Lossless compression techniques are requisite for the applications involving textual data, since losing a single character can be in the worst case make the text dangerously misleading.

In general, the input stream, generated from a data source, is fed into an encoder. The encoder then codes the stream of symbols and compresses data. If the compression is effective, the resulting stream of codes will be smaller than the original symbols. The decision to output a certain code for a certain symbol is based on a model. A notion of model is useful in understanding how the encoder works. The model defines the parameters that need to be used by the compression algorithm. For example, in Huffman coding, the probability of characters is used for coding. To regenerate original data from the compressed data, decoder is used. The decoder applies the reverse algorithm of that used by the encoder. Moreover, the decoder has some prior knowledge as to how the data is being encoded. This is how the standard compression algorithm works. The performance parameter compression ratio is given by ratio of uncompressed file size to the compressed file size.

In the next section, the related work is briefed. In Section 3, the compression techniques used to manipulate compressed data are discussed. Section 4 contains a preliminary performance analysis. We offer our conclusions in Section 5.

2. RELATED WORK
Singla *et al.* [1] gives the comparative analysis between Huffman and Arithmetic, concluding arithmetic coding is superior to Huffman. As arithmetic accommodates adaptive models easily and provide separation between model and coding. In arithmetic coding there is no need to translate each symbol into an integral number of bits, but it involves the large computation on the data like multiplication and division. The disadvantage of arithmetic coding is that it runs slowly, complicated to implement and it does not produce prefix code. Arithmetic Compression [2] is more suitable for small text when compared with Huffman compression and for large text Huffman compression is suitable.

When the size of the dictionary is unlimited the LZ78 compression [3] is optimal since the text file is large. Patterns in the dictionary may not be repeating patterns, and furthermore all parents of patterns are also included in the dictionary. The dictionary will diverge if these non-repeating patterns and pattern's parents are not handled properly. Hence, pruning the dictionary after each modified, LZ78 iteration is essential to have a convergent dictionary, thus enabling easier extraction of repeating patterns [4]. Kodituwakku *et. al.* [5] compares various lossless algorithms tested on different types of files, conclude that Shannon-Fano algorithm can be considered as the most efficient algorithm among the selected ones and RLE is suited for the text file having more number of repeating runs.

In our paper we compared the lossless data compression algorithms Huffman, Arithmetic, Golomb, LZ-78 in terms of their compression ratio, time required for coding. Also we suggest the efficient algorithm to be used for a certain type of file to be compressed taking into consideration both the compression ratio and compressed file size.

3. IMPLEMENTATION

3.1 Huffman Coding
Huffman Coding was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952. paper "A Method for the Construction of Minimum-Redundancy Codes" [1]. Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of

the source symbol. Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code that expresses the most common source symbols using shorter strings of bits than are used for less common source symbols.

Algorithm

For Encoding

1. Read Text File.
2. Get the probabilities of every character in the file.
3. Sort the characters in descending order according to their probabilities
4. Generate the binary tree from left to right until you have just one symbol left.
5. Read the tree from right to left assigning different bits to different branches.
6. Store the final Huffman dictionary.
7. Encode every character in the file by referring to the dictionary.
8. Transmit the encoded code along with the dictionary

For Decoding

1. Read the encoded code bitwise.
2. Look for the code in the dictionary
3. If there is a match gets, its corresponding symbol else read the next bit and repeat Steps 3 & 4
4. Write the decoded text in a file

3.2 Arithmetic Coding

In lossless compression the problem is to decompose a data set into a sequence of events, then to encode the events using as few bits as possible. The basic idea of arithmetic is to assign short codeword to more probable events and longer codeword to less probable events. Data can be compressed whenever some events are more likely than others. Arithmetic coding provides an effective mechanism for removing redundancy in the encoding of data. The older and better-known Huffman codes [2] are optimal only among instantaneous codes, that is, those in which the encoding of one event can be decoded before encoding has begun for the next event. In arithmetic coding an interval is assigned to each symbol. Starting with the interval [0, 1), each interval is divided in several subintervals, which sizes are proportional to the current probability of the corresponding symbols of the alphabet. The subinterval from the coded symbol is then taken as the interval for the next symbol. The output is the interval of the last symbol. Implementations write bits of this interval sequence as soon as they are certain. Arithmetic codes assign one codeword to each possible data set and the codeword length approximately equals to -log2p(s) where p(s) is the probability of the source sequence s. The shorter codes correspond to larger subintervals and thus more probable input data sets. In practice, the subinterval is refined incrementally using the probabilities of the individual events and is encoded with bits being output as soon as they are known. Arithmetic coding is different from other coding methods for which we know the exact relationship between the coded symbols and the actual bits that are written to a file. It encodes one data symbol at a time, and assigns to each symbol a real-valued number of bits. For highly skewed probability arithmetic coding is efficient coding method.

Algorithm

The algorithm for encoding a file using arithmetic coding works conceptually as follows:

1. Calculate the probability of each symbol.
2. Calculate its cumulative probability Pc.
3. Begin with a current interval [L, H) initialized to

[0, 1)

4. For each event in the file, we perform two steps.

a. We subdivide the current interval into subintervals, one for each possible event. The size of a event's subinterval is proportional to the estimated probability that the event will be the next event in the file, according to the model of the input.

b. We select the subinterval corresponding to the event that actually occurs next, making it the new current interval.

Lower interval L = L+ Pc* (H-L)*L;

Higher interval H = L+ Pc*(H-L)*H;

5. Assign a unique identical tag for the message.

For decoding, the tag value is taken; the new tag value is calculated using formula:

$$Tag = (Tag - L(s)) / Pc(s)$$

The corresponding lower interval and cumulative probability of symbol s is taken to decode the tag value. The process continues till the end of text file.

As the length of the source sequence increases, the length of the subinterval specified by the sequence decreases, and more bits are required to precisely identify the subinterval. Implementation of arithmetic coding uses a scaling strategy a rounding strategy. In Scaling, it magnifies each subinterval prior to partitioning so that its length is always close to 1. The length of the current interval is doubled so that it reflects only the unknown part of the final interval, while in rounding it uses a fixed bit length (b-bit) (finite precision) arithmetic to measure the subinterval and perform the partitioning. Consider the following example:

- If new subinterval is not entirely within one of the intervals [0, 1/2), [1/4, 3/4) or [1/2, 1) we stop iterating and return.

- If the new subinterval lies entirely within [0,1/2), we output 0 and any 1s left over from previous symbols; then we double the size of the interval [0,1/2) and expand toward the right.

- If the new subinterval lies entirely within [1/2, 1) we output 1 and 0s left over from previous symbols; then we double the size of the interval [1/2, 1) expanding toward the left.

- If the new subinterval lies entirely within [1/4, 3/4) we keep track of this fact for future output; then we double the size of the interval [1/4, 3/4) expanding in both directions away from the input.

3.3 LZ-78

One year after publishing LZ77 Jacob Ziv and Abraham Lempel had introduced another compression method [3]. Accordingly this procedure is called LZ78. LZ78 is based on a dictionary that will be created dynamically at runtime. Both the encoding and the decoding process use the same rules to

ensure that an identical dictionary is available. This dictionary contains any sequence already used to build the former contents. The compressed data have the general form:

1. Index (I) addressing an entry of the dictionary
2. First deviating symbol (S)

In contrast to LZ77 no combination of address and sequence length is used. Instead only the index to the dictionary is stored. The mechanism to add the first deviating symbol remains from LZ77.

Also instead of having carte blanche access to all the symbol strings in the preceding text, a dictionary of strings is built a single character at a time. The first time the string "Mark" is seen, for example, the string "Ma" is added to the dictionary. The next time, "Mar" is added. If "Mark" is seen again, it is added to the dictionary. This incremental procedure works very well at isolating frequently used strings and adding them to the table.

An important theoretical property of LZ78 is that when the input text is generated by a stationary, ergodic source, compression is asymptotically optimal as the size of the input increases. That is, LZ78 will code an indefinitely long string in the minimum size dictated by the entropy of the source. Very few compression methods enjoy this property. A source is ergodic if any sequence it produces becomes entirely representative of the source as its length grows longer and longer. Since this is a fairly mild assumption, it would appear that LZ78 is the solution to the text compression problem. The optimality, however, occurs as the size of the input tends to infinity, and most texts are considerably shorter than this! It relies on the size of the explicit character being significantly less than the size of the phrase code. Since the former is about 8 bits, it will still be consuming 20% of the output when 240 phrases have been constructed. Even if a continuous input were available, we would run out of memory long before compression became optimal.

The real issue is how fast LZ78 converges toward this limit. In practice, convergence is relatively slow, and performance is comparable to that of LZ77. The reason why LZ techniques enjoy so much popularity in practice is not because they are asymptotically optimal but for a much more prosaic reason-some variants lend themselves to highly efficient implementation.

Algorithm

For Encoding

1. Initially the dictionary and P are empty.
2. Read the input character string one by one.
3. Check whether the input character is present in the dictionary or not.
4. If it is not present in the dictionary then mark it as new index entry.
5. For each character of the input stream, the dictionary is searched for a match.
6. If a match is found, then last matching index is set to the index of the matching entry, and nothing is output.
7. If a match is not found, then a new dictionary entry is created.
8. Once the dictionary is full, no more entries are added. When the end of the input stream is reached, the algorithm outputs last matching index.

For Decoding

1. Initially dictionary is empty
2. Read the codeword consisting index (I) and symbol (S)
3. Output the original input character string using these codewords; simultaneously the dictionary is also updated.

The process goes on till the end of encoded codewords.

3.4 Golomb Coding

Golomb coding is a lossless data compression method using a family of data compression codes invented by Solomon W. Golomb in the 1960s. Golomb code gives an optimal prefix code where alphabets are geometrically distributed. This makes Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values. Golomb code is run-length based coding method. In machine code files, facsimile Data, video signals the Run Length Encoding (RLE) is used. Golomb coding is a practical and powerful implementation of RLE of binary streams. In RLE instead of sending long runs of '0's or '1's, it sends only how many are in the run. We propose two methods for Golomb coding; one is symbol encoding using absolute value and another using index integer value. In symbol encoding using absolute value the symbols are encoded in terms of ASCII values while in index method index integer values are assigned to the symbols.

Algorithm

1. Represent each subsequence of identical symbols by a pair (L,a) where L is the length of the subsequence, and 'a' is the recurring symbol in the subsequence. For e.g.: 'aaabbbbaaaa' is coded as (3,a) (4,b) (4,a)
2. The encoding of symbol is done by any method using absolute value or index integer value. Calculate the number of zero's. From that fix the parameter as M= ⌈-1/log2⌉ P to an integer value.
3. For N, the number to be encoded, find
   i. Quotient = q = ⌊N/M⌋
   ii. Remainder = r = N modulo M
4. Generate Codeword as in format:
   <Quotient Code> <Remainder Code>,
   i. Quotient Code (in unary coding)
      a. Write a q-length string of 1 bits
      b. Write a 0 bit
   ii. Remainder Code
      a. If M is power of 2, code remainder as binary format. So log2M bits are needed.
      b. If M is not a power of 2, set b= ⌈log2 M⌉
         If r < (2b – M) code r as plain binary using b-1 bits.
         If r ≥ (2b – M) code the number (r + 2b- M) in plain binary representation using b bits.

4. Results

For the comparative analysis of data compression algorithms different types of test text files are taken which includes English text files, Log files, Sorted word list and geometrically distributed data text file, the sample files are shown in Fig.1. The algorithms are executed on Intel(R)

Core(TM)2 Duo CPU T6600 @ 2.20GHz, 2.20Ghz with RAM: 4.00GB (3.50 GB usable), Windows 7 (64bit).

For the Test1 file consisting English text the arithmetic and Huffman gives compression ratio of 1.738, 1.721 respectively which is good one, while LZ-78, Golomb does not perform well. For the small variations in the text files the Golomb fails to compress the file so its compression ratio is less than 1 for all test files except Test4. The time required for encoding is very large for arithmetic i.e. speed of encoding is very slow, as it has great computational complexity. If the file size is large the dictionary based LZ-78 fails to compress much more compared to Huffman and arithmetic. The test files are shown in Fig. 1 and result in

RESULT TABLE I

File Name	Encoding Time(sec)	Decoding Time(sec)	Original Size(Byte	Compressed	Compression
A. Huffman Coding					
Test1	5.19364304	57.8890950	18036	10475.2502	1.72177
Test2	0.18720108	6.91170801	1953	1291.375	1.51234
Test3	0.15442160	0.64738037	231	115.12500	2.00651
Test4	0.16503020	0.1004763	116	19.875	5.83647
B. Arithmetic Coding					
Test1	269.595014	57.4358269	18036	10376.875	1.73809
Test2	6.64236644	1.87559584	1953	1282.625	1.52265
Test3	0.3081655	0.09852928	231	115.375	2.00216
Test4	0.058989	0.048297	116	14.87	7.79831
C. LZ-78 Coding					
Test1	18.4466144	1.32978234	18036	12375.8797	1.45735
Test2	0.53048965	0.16914965	1953	1477.25005	1.32205
Test3	0.03739823	0.03395197	231	163.999744	1.40853
Test4	0.010867	0.014461	116	38	3.05263
D. Golomb (Absolute) Coding					
Test1	607.488288	702.6754	18036	24768	0.7282
Test2	6.976569	11.627427	1953	2608.3	0.7488
Test3	0.820101	0.944019	231	358.5	0.6444
Test4	1.630964	0.182628	116	18.25	6.3562
E. Golomb (Index) Coding					
Test1	415.698433	786.4765	18036	19746	0.9134
Test2	5.544032	9.544212	1953	2119.1	0.9216
Test3	1.135808	1.630463	231	209	1.1053
Test4	0.293351	0.150408	116	11.625	9.9785

5. CONCLUSION

On the basis of results we conclude that Huffman coding is optimal when the probability of each input symbol is a negative power of two. Huffman fails to compress the data in large amount when the symbols have skewed probability and long runs as compared to Arithmetic and RLE based Golomb coding respectively. For the text containing highly skewed probability symbols, the arithmetic coding performs very well, though the computational complexity is very high and its speed is less compared to Huffman.

LZ-78 gives good compression ratio for highly correlated data. For the large file size, larger dictionary is required causing higher efforts for addressing and administration both at runtime. Also as the file size increases the compression efficiency of the algorithm increases because of increase in size of dictionary. The compression is optimal when the text file is large as there are more chances to replace identified words by using a small index number. Golomb coding is RLE based coding method. It gives its best performance when the data in text is highly geometrically distributed. In Golomb coding when the runs are encoded using the index-integer method, it gives better performance than the absolute valued. Using the index method the compression ratio and speed is increased as compared to absolute valued method as it requires more number of bits to encode the symbol in ASCII value. The disadvantage of index method is that at decoder side the index valued dictionary must be provided. Golomb coding is highly dependent on the geometric distribution of data.

6. REFERENCES

[1] Vikas Singla, Rakesh Singla and Sandeep Gupta, "Data compression modelling: Huffman and Arithmetic", International Journal of The Computer, the Internet and Management, Vol. 16 No.3, Page(s):64- 68.Sept-Dec, 2008

[2] O.Srinivasa Rao, Prof.S.Pallam Setty, "Comparative Study of Arithmetic and Huffman Compression Techniques for Enhancing Security and Effective Bandwidth Utilization in the Context of ECC for Text", International Journal of Computer Applications, Vol. 29 No.6, Page(s):44-60, September 2011.

[3] Ahmed S. Musa, Ayman Al-Dmour, Mansour I. Irshid, "An Efficient Text Compression Technique Based on Using Bitwise Lempel-Ziv Algorithm", Australian Journal of Basic and Applied Sciences, 4(12),ISSN 1991-8178, INSInet Publication, Page(s):6564-6569, 2010.

[4] Hsuan-Huei Shih, Shrikanth S. Narayanan and C.-C. Jay Kuo; "A Dictionary Approach To Repetitive Pattern Finding In Music", IEEE International Conference on Multimedia and Expo (ICME 2001) , Page(s): 397- 400.

[5] S.R. Kodituwakku, U. S.AMARASINGHE, "Comparison of lossless data compression algorithms for text data", Indian Journal of Computer Science and Engineering, ISSN : 0976-5166, Vol 1 No 4 416-425, 2010.

[6] Mamta Sharma, "Compression Using Huffman Coding", IJCSNS International Journal of Computer Science and Network Security, Vol.10 No.5, Page(s):133-141, May 2010.

[7] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the IRE, Vol. 40, No. 9, Page(s): 1098-1101, September 1952.

[8] Jacob Ziv, Abraham Lempel, "Compression of Individual Sequences via Variable-Rate Coding", IEEE Transactions on Information Theory, Vol.24, No.5, Page(s):530-536, September 1978.

[9] Forrest Elliott and Manfred Huber, Learning Macros with an Enhanced LZ78 Algorithm, Technical Report CSE, The University of Texas at Arlington, 2005.

[10] David Salomon, Data Compression: The Complete Reference, Third edition, Pgs 185-188. Springer, 2004.

ENGLISH TEXT (Test1)

Mrs. Rachel Lynde lived just where the Avonlea main road dipped down into a little hollow, fringed with alders and ladies' eardrops and traversed by a brook that had its source away back in the woods of the

old Cuthbert place; it was reputed to be an intricate, headlong brook in its earlier course through those woods, with dark secrets of pool and cascade; but by the time it reached Lynde's Hollow it was a quiet,

LOG FILE (Test2)

216.154.140.251-[03/Feb/2003:04:21:02+0100"GET thumbn/cf104_r.jpg HTTP/1.1" 200 4559 "http://www.fighter-planes.com/data5060.htm" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 95)"

216.154.140.251 - - [03/Feb/2003:04:21:07 +0100] "GET /full/mig17.jpg HTTP/1.1" 200 11756 "http://www.fighter-

SORTED WORD LIST (Test3)

galvanomagnetism
galvanometer
galvanometers
galvanometric

GEOMETRICALLY DISTRIBUTED (Test4)

wwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwbbbbbbbbbbbbwwwwwwwwwwwwwwww
wwbbbbwwwbbbbbbbbbbbbbbbbbbwwwwwww
wwwwbbbbbbbwwwwwwwww

Figure1. Input Text files