ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

A Survey on Different Compression Techniques Algorithm for Data Compression

[I]Hardik Jani, [II]Jeegar Trivedi
[I]C. U. Shah University, India
[II]S. P. University, India

Abstract
Compression is useful because it helps us to reduce the resources usage, such as data storage space or transmission capacity. Data Compression is the technique of representing information in a compacted form. The actual aim of data compression is to be reduced redundancy in stored or communicated data, as well as increasing effectively data density. The data compression has important tool for the areas of file storage and distributed systems. To desirable Storage space on disks is expensively so a file which occupies less disk space is "cheapest" than an uncompressed files. The main purpose of data compression is asymptotically optimum data storage for all resources. The field data compression algorithm can be divided into different ways: lossless data compression and optimum lossy data compression as well as storage areas.
Basically there are so many Compression methods available, which have a long list. In this paper, reviews of different basic lossless data and lossy compression algorithms are considered. On the basis of these techniques researcher have tried to purpose a bit reduction algorithm used for compression of data which is based on number theory system and file differential technique. The statistical coding techniques the algorithms such as Shannon-Fano Coding, Huffman coding, Adaptive Huffman coding, Run Length Encoding and Arithmetic coding are considered.

I. Introduction

Data compression, source coding, or bit-rate reduction involves encrypted information using fewer bits than the original representation. Lossless Compression reduces bits by identifying and eliminating statistical redundancy. Compression can be either lossy or lossless. Lossless compression reduces bits by identifying and eliminating statistical redundancy No information is lost in lossless compression. Lossy compression reduces bits by identifying marginally important information and removing it. The process of reducing the size of a data file is popularly referred to as data compression, although its formal name is source coding (coding done at the source of the data, before it is stored transmitted). No information is lost in lossless compression. Compression is useful because it helps reduce resources usage, such as data storage space or transmission capacity. Because compressed data must be decompressed to use, this extra processing imposes computational or other costs through decompression, this situation is far from being a free lunch. Data compression is subject to a space-time complexity trade-off. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed, and the option to decompress the video in full before watching it may be inconvenient or require additional storage. The design of data compression schemes involve trade-offs among various factors, including the degree of compression, the amount of distortion introduced (e.g., when using lossy data compression), and the computational resources required to compress and uncompressed the data. New alternatives to traditional systems, which sample at full resolution then compress, provide efficient resource usage based on principles of compressed sensing. Compressed sensing techniques circumvent the need for data compression by sampling off a cleverly selected basis. Lossy Compression reduces bits by identifying marginally important information and removing it. Therefore, according to demands of such a situation, development of a new Compression Algorithm for Optimum Data Storage is the need of the hour and hence our research project intends to develop the same [17].

II. Statistical Compression Techniques

1. Run Length Encoding Technique (RLE)

Run Length Encoding Technique is the simplest compression techniques it is created especially for data with strings of repeated symbols (the length of the string is called a run). This algorithm consists of replacing large sequences of repeating data with only one item of this data followed by a counter showing how many times this item is repeated. The main idea behind this is to encode repeated symbols as a pair: the length of the string and the symbol. For example, 'aaaaaaaaaabbbbbbaxxx', this string's length is **20** and as researcher can see there are lots of repetitions. Using the run-length algorithm, Researcher replaces any run with shorter string followed by a counter. 'A10b6a1x3', the length of this string is **12**, which is approximately **60%** of the initial length. Obviously this is not the optimal way to compress the given string. For instance we don't need to use the digit "1" when the character is repeated only once.
The biggest problem with RLE is that in the worst case the size of output data can be two times more than the size of input data. To eliminate this problem, each pair (the lengths and the strings separately) can be later encoded with an algorithm like Huffman coding [17].

2. Shannon Fano Coding

Shannon–Fano coding, named after Claude Elwood Shannon and Robert Fano, is a technique for constructing a preface code based on a set of symbols and their probabilities. It allots less number of bits for highly probable messages and more number of bits for rarely occurring messages. The algorithm is as follows:

1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts table.
2. Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the left and the least common at the right.
3. Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible.
4. Assign the upper half of the list a binary digit '0' and the lower half a '1'.

5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

Generally, Shannon-Fano coding does not guarantee that an optimal code is generated. Shannon – Fano algorithm is more efficient when the probabilities are closer to inverses of powers of 2 [17].

3. Huffman Coding

The Huffman coding algorithm [2] is named after its inventor, David Huffman, who developed this algorithm as a student in a class on information theory at MIT in 1950. It is a more successful method used for text compression. Huffman's scheme uses a table of frequency of occurrence for each symbol (or character) in the input. This table may be derived from the input itself or from data which is representative of the input. When using variable-length code words, it is desirable to create a (uniquely decipherable) prefix-code, avoiding the need for a separator to determine code word boundaries. Huffman coding creates such a code. Huffman algorithm is not very different from Shannon - Fano algorithm. Both the algorithms employ a variable bit probabilistic coding method. The two algorithms significantly differ in the manner in which the binary tree is built. Huffman uses bottom-up approach and Shannon - Fano uses Top-down approach.

The Huffman algorithm is simple and can be described in terms of creating a Huffman code tree. The procedure for building this tree is:

1. Start with a list of free nodes, where each node corresponds to a symbol in the alphabet.
2. Select two free nodes with the lowest weight from the list.
3. Create a parent node for these two nodes selected and the weight is equal to the weight of the sum of two child nodes.
4. Remove the two child nodes from the list and the parent node is added to the list of free nodes.
5. Repeat the process starting from step-2 until only a single tree remains.

After building the Huffman tree, the algorithm creates a prefix code for each symbol from the alphabet simply by traversing the binary tree from the root to the node, which corresponds to the symbol. It assigns 0 for a left branch and 1 for a right branch.

The algorithm presented above is called as a semi adaptive or semi-static Huffman coding as it requires knowledge of frequencies for each symbol from alphabet. Along with the compressed output, the Huffman tree with the Huffman codes for symbols or just the frequencies of symbols which are used to create the Huffman tree must be stored. This information is needed during the decoding process and it is placed in the header of the compressed file.

4. Adaptive Huffman Coding

Adaptive Huffman coding calculates the frequencies dynamically based on recent actual frequencies in the source string. Adaptive Huffman coding which is also called dynamic Huffman coding is an adaptive coding technique based on Huffman coding building the code as the symbols are being transmitted that allows one-pass encoding and adaptation to changing conditions in data. The benefits of one-pass procedure is that the source can be encoded in real time, though it becomes more sensitive to transmission errors, since just a single loss ruins the whole code. The basic Huffman algorithm suffers from the drawback that to generate Huffman codes it requires the probability distribution of the input set which is often not available. Moreover it is not suitable to cases when probabilities of the input symbols are changing. The Adaptive Huffman coding technique was developed based on Huffman coding first by Newton Faller [1] and by Robert G. Gallager [4] and then improved by Donald Knuth [5] and Jefferey S. Vitter [12,13]. In this method, a different approach known as sibling property is followed to build a Huffman tree. Here, both sender and receiver maintain dynamically changing Huffman code trees whose leaves represent characters seen so far. Initially the tree contains only the 0-node, a special node representing messages that have yet to be seen. Here, the Huffman tree includes a counter for each symbol and the counter is updated every time when a corresponding input symbol is coded. Huffman tree under construction is still a Huffman tree if it is ensured by checking whether the sibling property is retained. If the sibling property is violated, the tree has to be restructured to ensure this property. Usually this algorithm codes that are more effective than static Huffman coding. Storing Huffman tree along with the Huffman codes for symbols with the Huffman tree is not needed here. It is superior to Static Huffman coding in two aspects: It requires only one pass through the input and it adds little or no overhead to the output. But this algorithm has to rebuild the entire Huffman tree after encoding each Symbol which becomes slower than the static Huffman coding.

5. Arithmetic Coding

Huffman and Shannon-Fano coding techniques suffer from the fact that an integral value of bits is needed to code a character. Arithmetic coding completely bypasses the idea of replacing every input symbol with a codeword. Instead it replaces a stream of input symbols with a single floating point number as output. The basic concept of arithmetic coding was developed by Elias in the early 1960's and further developed largely by Pasco[7], Rissanen [9,10] and Langdon[3]. The main aim of Arithmetic coding is to assign an interval to each potential symbol. Then a decimal number is assigned to this interval. The algorithm starts with an interval of 0.0 and 1.0. After each input symbol from the alphabet is read, the interval is subdivided into a smaller interval in proportion to the input symbol's probability. This subinterval then becomes the new interval and is divided into parts according to probability of symbols from the input alphabet. This is repeated for each and every input symbol. And, at the end, any floating point number from the final interval uniquely determines the input data.

III. Dictionary Based Compression Techniques

Arithmetic algorithms as well as Huffman algorithms are based on a statistical model, namely an alphabet and the probability distribution of a source. Dictionary coding techniques rely upon the observation that there are correlations between parts of data (recurring patterns). The basic idea is to replace those repetitions by (shorter) references to a "dictionary" containing the original.

A. Lempel ZIV Algorithms

LZW compression has its roots in the work of Jacob Ziv and Abraham Lempel. In 1977, they published a paper on "sliding-window" compression, and followed it with another paper in 1978 on "dictionary" based compression. These algorithms were named LZ77 and LZ78, respectively. Then in 1984, Terry Welch made a modification to LZ78

Which become very popular and was dubbed LZW (guess why).

ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

This is exactly the approach that LZW compression takes. It starts with a "dictionary" of all the single character. It then starts to expand dictionary as information gets sent through. Pretty soon, redundant string will be coded as a single bit, and compression has occurred.

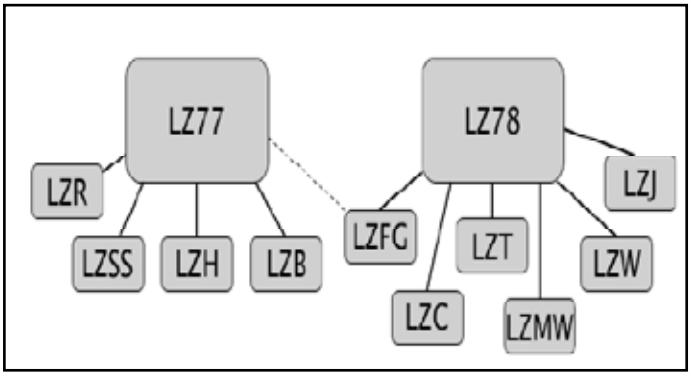

Fig. 1 : Lempel Ziv Algorithm Family]

This figure shows how the two original Lempel Ziv algorithms, LZ77 and LZ78, work and also presents and compares several of the algorithms that have been derived from the original Lempel Ziv algorithms.

B. LZ77

Jacob Ziv and Abraham Lempel have presented their dictionary-based scheme in 1977 for lossless data compression [15]. Today this technique is much remembered by the name of the authors and the year of implementation of the same. LZ77 exploits the fact that words and phrases within a text file are likely to be repeated. When there is repetition, they can be encoded as a pointer to an earlier occurrence, with the pointer accompanied by the number of characters to be matched. It is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source. In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window which consists of two parts: a search buffer that contains a portion of the recently encoded sequence and a look ahead buffer that contains the next portion of the sequence to be encoded. The algorithm searches the sliding window for the longest match with the beginning of the look-ahead buffer and outputs a reference (a pointer) to that match. It is possible that there is no match at all, so the output cannot contain just pointers. In LZ77 the reference is always output as a triple <o,l,c>, where 'o' is an offset to the match, 'l' is length of the match, and 'c' is the next symbol after the match. If there is no match, the algorithm outputs a null-pointer (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer. The values of an offset to a match and length must be limited to some maximum constants. Moreover the compression performance of LZ77 mainly depends on these values. Usually the offset is encoded on 12–16 bits, so it is limited from 0 to 65535 symbols. So, there is no need to remember more than 65535 last seen symbols in the sliding window. The match length is usually encoded on 8 bits, which gives maximum match length equal to 255[8]. In LZ77 encoding process one reference (a triple) is transmitted for several input symbols and hence it is very fast. The decoding is much faster than the encoding in this process and it is one of the important features of this process. In LZ77, most of the LZ77 compression time is, however, used in searching for the longest match, whereas the LZ77 algorithm decompression is quick as each reference is simply replaced with the string, which it points to. There are lots of ways that LZ77 scheme can be made more efficient and many of the improvements deal with the efficient encoding with the triples. There are several variations on LZ77 scheme, the best known are LZSS, LZH and LZB. LZSS which was published by Storer and Szymanksi [11] removes the requirement of mandatory inclusion of the next non-matching symbol into each code word. Their algorithm uses fixed length code words consisting of offset and length to denote references. They propose to include an extra bit (a bit flag) at each coding step to indicate whether the output code represents a pair (a pointer and a match length) or a single symbol. LZH is the scheme that combines the Ziv – Lempel and Huffman techniques. Here coding is performed in two passes. The first is essentially same as LZSS, while the second uses statistics measured in the first to code pointers and explicit characters using Huffman coding. LZB was published by Mohammad Banikazemi[6] uses an elaborate scheme for encoding the references and lengths with varying sizes. Regardless of the length of the phrase it represents, every LZSS pointer is of the same size. In practice a better compression is achieved by having different sized pointers as some phrase lengths are much more likely to occur than others. LZB is a 1technique that uses a different coding for both components of the pointer. LZB achieves a better compression than LZSS and has the added virtue of being less sensitive to the choice of parameters.

C. LZ78

In 1978 Jacob Ziv and Abraham Lempel presented their dictionary based scheme [16], which is known as LZ78. It is a dictionary based compression algorithm that maintains an explicit dictionary. This dictionary has to be built both at the encoding and decoding side and they must follow the same rules to ensure that they use an identical dictionary. The code words output by the algorithm consists of two elements <i,c> where 'i' is an index referring to the longest matching dictionary entry and the first non-matching symbol. In addition to outputting the code word for storage / transmission the algorithm also adds the index and symbol pair to the dictionary. When a symbol that is not yet found in the dictionary, the code word has the index value 0 and it is added to the dictionary as well. The algorithm gradually builds up a dictionary with this method. The algorithm for LZ78 is given below: LZ78 algorithm has the ability to capture patterns and hold them indefinitely but it also has a serious drawback. The dictionary keeps growing forever without bound. There are various methods to limit dictionary size, the easiest being to stop adding entries and continue like a static dictionary coder or to throw the dictionary away and start from scratch after a certain number of entries has been reached. The encoding done by LZ78 is fast, compared to LZ77, and that is the main advantage of dictionary based compression. The important property of LZ77 that the LZ78 algorithm preserves is the decoding is faster than the encoding. The decompression in LZ78 is faster compared to the process of compression.

D. LZW

Terry Welch has presented his LZW (Lempel–Ziv– Welch) algorithm in 1984[14], which are based on LZ78. It basically applies the LZSS principle of not explicitly transmitting the next non-matching symbol to LZ78 algorithm. The original proposal of LZW, the pointer size is chosen to be 12 bits, allowing for up to

4096 dictionary entries. Once the limit is reached, the dictionary becomes static. LZFG which was developed by Fiala and Greene [18], gives fast encoding and decoding and good compression without undue storage requirements. This algorithm uses the original dictionary building technique as LZ78 does but the only difference is that it stores the elements in a trie data structure. Here, the encoded characters are placed in a window (as in LZ77) to remove the oldest phrases from the dictionary.

IV. Experimental Results

In this section we focus our attention to compare the performance of various Statistical compression techniques (Run Length Encoding, Shannon-Fano coding, Huffman coding, Adaptive Huffman coding and Arithmetic coding), LZ77 family algorithms (LZ77, LZSS, LZH and LZB) and LZ78 family algorithms (LZ78, LZW and LZFG). Research works done to evaluate the efficiency of any compression algorithm are carried out having two important parameters. One is the amount of compression achieved and the other is the time used by the encoding and decoding algorithms. Researchers has tested several times the practical performance of the above mentioned techniques on files of Canterbury corpus and have found out the results of various Statistical coding techniques and Lempel –Ziv techniques selected for this study. Also, the comparative functioning and the compression ratio are presented in the tables given below.

Table I. Compression of BPC for different Statistical Compression Techniques

Sr. No.	File Names	File Size	RLE	Shannon Fano Coding	Huffman Coding	Adaptive Huffman Coding	Arithmetic Coding
			BPC	BPC	BPC	BPC	BPC
1	bib	111261	8.16	5.56	5.26	5.24	5.23
2	Book 1	768771	8.17	4.83	4.57	4.56	4.55
3	Book 2	610856	8.16	5.08	4.83	4.83	4.78
4	News	377109	7.98	5.41	5.24	5.23	5.19
5	Obj 1	21504	7.21	6.57	6.45	6.11	5.97
6	Nbj 2	246814	8.05	6.50	6.33	6.31	6.07
7	Paper 1	53161	8.12	5.34	5.09	5.04	4.98
8	Paper 2	82199	8.14	4.94	4.68	4.65	4.63
9	Prog C	39611	8.10	5.47	5.33	5.26	5.23
10	Prog 1	71646	7.73	5.11	4.85	4.81	4.76
11	Prog P	49379	7.47	5.28	4.97	492	4.89
12	Trans	93695	7.90	5.88	5.61	5.58	5.49
	Average BPC		7.93	5.50	5.27	5.21	5.15

1. Practical Comparison of Statistical Compression Techniques

Table – I shows the comparative analysis between various Statistical compressions techniques discussed above. As per the results shown in Table – I, for Run Length Encoding, for most of the files tested, this algorithm generates compressed files larger than the original files. This is due to the fewer amount of runs in the source file. For the other files, the compression rate is less. The average BPC obtained by this algorithm is 7.93. So, it is inferred that this algorithm can reduce on an average of about 4% of the original file. This cannot be considered as a significant improvement. BPC and amount of compression achieved for Shannon-Fano algorithm is presented in Table-I. The compression ratio for Shannon-Fano algorithm is in the range of 0.60 to 0.82 and the average BPC is 5.50. Compression ratio for Huffman coding algorithm falls in the range of 0.57 to 0.81. The compression ratio obtained by this algorithm is better compared to Shannon-Fano algorithm and the average Bits per character is 5.27. The amount of compression achieved by applying Adaptive Huffman coding is shown in Table – I. The adaptive version of Huffman coding builds a statistical model of the text being compressed as the file is read. From Table – I it can be seen that, it differs a little from the Shannon-Fano coding algorithm and Static Huffman coding algorithm in the compression ratio achieved and the range is between 0.57 and 0.79. On an average the number of bits needed to code a character is 5.21. Previous attempts in this line of research make it clear that compression and decompression times are relatively high for this algorithm because the dynamic tree used in this algorithm has to be modified for each and every character in the source file. Arithmetic coding has been shown to compress files down to the theoretical limits as described by Information theory. Indeed, this algorithm proved to be one of the best performers among these methods based on compression ratio. It is clear that the amount of compression achieved by Arithmetic coding lies within the range of 0.57 to 0.76 and the average bits per character is 5.15.

V. Conclusion

Researcher has taken up Statistical compression techniques and Lempel Ziv algorithms for our study to examine the performance in compression. In the Statistical compression techniques, Arithmetic coding technique outperforms the rest with an improvement of 1.15% over Adaptive Huffman coding, 2.28% over Huffman coding, 6.36% over Shannon-Fano coding and 35.06% over Run Length Encoding technique. LZB outperforms LZ77, LZSS and LZH to show a marked compression, which is 19.85% improvement over LZ77, 6.33% improvement over LZSS and 3.42% improvement over LZH, amongst the LZ77 family. LZFG shows a significant result in the average BPC compared to LZ78 and LZW. From the result it is evident that LZFG has outperformed the other two with an improvement of 32.16% over LZ78 and 41.02% over LZW.

References

[1] *Faller N., "An adaptive system for data compression", In Record of the 7th Asilornar Conference on Circuits, Systems and Computers, pages 593-597, Piscataway, NJ, 1973. IEEE Press.*

[2] *Huffman D.A., "A method for the construction of minimum redundancy codes", Proceedings of the Institute of Radio Engineers, 40 (9), pp. 1098–1101, September 1952.*

[3] *Langdon G.G., "An introduction to arithmetic coding", IBM Journal of Research and Development, 28(2), pp. 135–149, March 1984.*

[4] *Gallager R.G., "Variations on a theme by Huffman", IEEE Transactions on Information Theory, IT-24(6):668-674, November 1978.*

[5] *Knuth D.E., "Dynamic Huffman coding", Journal of Algorithms,6(2):163-180, June 1985.*

[6] *Mohammad Banikazemi, "LZB: Data Compression with Bounded References", Proceedings of the 2009 Data Compression Conference, IEEE Computer Society, 2009.*

[7] *Pasco.R., "Source coding algorithms for fast data compression", Ph.D thesis, Department of Electrical Engineering, Stanford University, 1976.*

[8] *Przemyslaw Skibinski, "Reversible Data transforms that improve effectiveness of universal lossless data compression", Ph.D thesis, Department of Mathematics and Computer Science, University of Wroclaw, 2006.*

ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

[9] Rissanen J., "Generalised Kraft inequality and arithmetic coding", IBM Journal of Research and Development, 20, pp. 198–203,1976.

[10] Rissanen J and Langdon G.G., "Arithmetic coding", IBM Journal of Research and Development, 23 (2), pp. 149–162, 1979.

[11] Storer J and Szymanski T.G., "Data compression via textual substitution", Journal of the ACM 29, pp. 928–951, 1982.

[12] Vitter J.S., "Design and analysis of dynamic Huffman codes", Journal of the A CM, 34(4):825-845, October 1987.

[13] Vitter J.S., "Dynamic Huffman coding", ACM Transactions on Mathematical Software, 15(2):158-167, June 1989.

[14] Welch T.A., "A technique for high-performance data compression", IEEE Computer, 17, pp. 8–19, 1984.

[15] Ziv. J and Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory 23 (3), pp. 337–342, May 1977.

[16] Ziv. J and Lempel A., "Compression of Individual Sequences via Variable-Rate Coding", IEEE Transactions on Information Theory 24 (5), pp. 530–536, September 1978.

[17] Senthil Shanmugasundaram, Robert Lourdusamy, "A Comparative Study Of Text Compression Algorithms", "International Journal of Wisdom Based Computing" , December 2011.

[18] Fiala E.R., and D.H. Greene, "Data Compression with finite windows", Communications of the ACM 32(4):490-505, 1989.

Author's Profile

I Hardik B. Jani. received my Bachelors degree in Commerce with Information Technology in 2007 from Saurashtra University and Masters degree in Master of Science in Information Technology & Computer Application in 2010 from Saurashtra University. I served as a full-time Lecturer at Faculty of Computer Science Department in C. U. Shah University. My research interest includes Compression Algorithms, Database, security and operating system.