

A Comparative Study Of Text Compression Algorithms

Senthil Shanmugasundaram
Department of Computer Science,
Vidyasagar College of Arts and Science,
Udumalpet, Tamilnadu, India
E-mail : senthil_udt@rediffmail.com

Robert Lourdasamy
Computer Science & Info. System Department,
Community College in Al-Qwaiya,
Shaqua University, KSA
(Government Arts College, Coimbatore-641 018)
E-mail : robert_lourdes@yahoo.com

Abstract: Data Compression is the science and art of representing information in a compact form. For decades, Data compression has been one of the critical enabling technologies for the ongoing digital multimedia revolution. There are lot of data compression algorithms which are available to compress files of different formats. This paper provides a survey of different basic lossless data compression algorithms. Experimental results and comparisons of the lossless compression algorithms using Statistical compression techniques and Dictionary based compression techniques were performed on text data. Among the statistical coding techniques the algorithms such as Shannon-Fano Coding, Huffman coding, Adaptive Huffman coding, Run Length Encoding and Arithmetic coding are considered. Lempel Ziv scheme which is a dictionary based technique is divided into two families: those derived from LZ77 (LZ77, LZSS, LZH and LZB) and those derived from LZ78 (LZ78, LZW and LZFG). A set of interesting conclusions are derived on their basis.

I. INTRODUCTION

Data compression refers to reducing the amount of space needed to store data or reducing the amount of time needed to transmit data. The size of data is reduced by removing the excessive information. The goal of data compression is to represent a source in digital form with as few bits as possible while meeting the minimum requirement of reconstruction of the original.

Data compression can be *lossless*, only if it is possible to exactly reconstruct the original data from the compressed version. Such a lossless technique is used when the original data of a source are so important that we cannot afford to lose any details. Examples of such source data are medical images, text and images preserved for legal reason, some computer executable files, etc.

Another family of compression algorithms is called *lossy* as these algorithms irreversibly remove some parts of data and only an approximation of the original data can be reconstructed. Approximate reconstruction may be desirable since it may lead to more effective compression. However, it often requires a good balance between the visual quality and the computation complexity. Data such as multimedia images, video and audio are more easily compressed by lossy compression

techniques because of the way human visual and hearing systems work.

Lossy algorithms achieve better compression effectiveness than lossless algorithms, but lossy compression is limited to audio, images, and video, where some loss is acceptable.

The question of the better technique of the two, “lossless” or “lossy” is pointless as each has its own uses with lossless techniques better in some cases and lossy technique better in others.

There are quite a few lossless compression techniques nowadays, and most of them are based on dictionary or probability and entropy. In other words, they all try to utilize the occurrence of the same character/string in the data to achieve compression. This paper examines the performance of statistical compression techniques such as Shannon- Fano Coding, Huffman coding, Adaptive Huffman coding, Run Length Encoding and Arithmetic coding. The Dictionary based compression technique Lempel-Ziv scheme is divided into two families: those derived from LZ77 (LZ77, LZSS, LZH and LZB) and those derived from LZ78 (LZ78, LZW and LZFG).

The paper is organized as follows: Section I contains a brief Introduction about Compression and its types, Section II presents a brief explanation about Statistical compression techniques, Section III discusses about Dictionary based compression techniques, Section IV has its focus on comparing the performance of Statistical coding techniques and Lempel Ziv techniques and the final section contains the Conclusion.

II. STATISTICAL COMPRESSION TECHNIQUES

2.1 RUN LENGTH ENCODING TECHNIQUE (RLE)

One of the simplest compression techniques known as the Run-Length Encoding (RLE) is created especially for data with strings of repeated symbols (the length of the string is called a *run*). The main idea behind this is to encode repeated symbols as a pair: the length of the string and the symbol. For example, the string

'abbaaaabaabbbbaa' of length 16 bytes (characters) is represented as 7 integers plus 7 characters, which can be easily encoded on 14 bytes (as for example '1a2b5a1b2a3b2a'). 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.

2.2 SHANNON FANO CODING

Shannon – Fano algorithm was simultaneously developed by Claude Shannon (Bell labs) and R.M. Fano (MIT)[3,15]. It is used to encode messages depending upon 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 frequency or probability table.
2. Sort the table according to the frequency, with the most frequently occurring symbol at the top.
3. Divide the table into two halves with the total frequency count of the upper half being as close to the total frequency count of the bottom half 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 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.

2.3 HUFFMAN CODING

The Huffman coding algorithm [6] 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 idea is to replace fixed-length codes (such as ASCII) by variable-length codes, assigning shorter codewords to the more frequently occurring symbols and thus decreasing the overall length of the data. When using variable-length codewords, it is desirable to create a (uniquely decipherable) prefix-code, avoiding the need for a separator to determine codeword 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.

2.4 ADAPTIVE HUFFMAN CODING

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 [2] and by Robert G. Gallager[5] and then improved by Donald Knuth [8] and Jefferey S. Vitter [17,18]. 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 generates

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.

2.5 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[11], Rissanen [13,14] and Langdon[9].

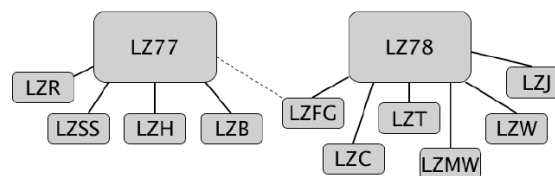
The main aim of Arithmetic coding is to assign an interval to each potential symbol. Then a decimal number is assigned to this interval. The algorithm starts with an interval of 0.0 and 1.0. After each input symbol from the alphabet is read, the interval is subdivided into a smaller interval in proportion 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.

3.1 LEMPEL ZIV ALGORITHMS

The Lempel Ziv Algorithm is an algorithm for lossless data compression. It is not a single algorithm, but a whole family of algorithms, stemming from the two algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 and 1978.



3.1.1 LZ77:

Jacob Ziv and Abraham Lempel have presented their dictionary-based scheme in 1977 for lossless data compression [20]. 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 $\langle o, l, c \rangle$, 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[7].

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[12].

The LZ77 algorithm is given below:

```
While (lookAheadBuffer not empty) {
  get a reference (position, length) to longest match;
  if (length > 0)
  {
    output (position, length, next symbol);
    shift the window length+1 positions along;
  }
  else {
    output (0, 0, first symbol in the lookahead buffer);
    shift the window 1 character along;
  }
}
```

With regard to other algorithms the time for compression and decompression is just the same. 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 Szymanski [16] removes the requirement of mandatory inclusion of the next non-matching symbol into each codeword. Their algorithm uses fixed length codewords 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[10] 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 technique 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.

3.1.2 LZ78

In 1978 Jacob Ziv and Abraham Lempel presented their dictionary based scheme [21], 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 codewords output by the algorithm consists of two elements $\langle i, c \rangle$ where 'i' is an index referring to the longest matching dictionary entry and the first non-matching symbol. In addition to outputting the codeword 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 codeword 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:

```
w := NIL;
while ( there is input ) {
  K := next symbol from input;
  if (wK exists in the dictionary) {
    w := wK;
  } else {
    output (index(w), K);
    add wK to the dictionary;
    w := NIL;
  }
}
```

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.

LZW

Terry Welch has presented his LZW (Lempel–Ziv–Welch) algorithm in 1984[19], which is based on LZ78. It basically applies the LZSS principle of not explicitly transmitting the next non-matching symbol to LZ78 algorithm. The dictionary has to be initialized with all

possible symbols from the input alphabet. It guarantees that a match will always be found. LZW would only send the index to the dictionary. The input to the encoder is accumulated in a pattern 'w' as long as 'w' is contained in the dictionary. If the addition of another letter 'K' results in a pattern 'w*K' that is not in the dictionary, then the index of 'w' is transmitted to the receiver, the pattern 'w*K' is added to the dictionary and another pattern is started with the letter 'K'. The algorithm then proceeds as follows:

```

w := NIL;
while ( there is input ) {
  K := next symbol from input;
  if (wK exists in the dictionary) {
    w := wK;
  } else {
    output (index(w));
    add wK to the dictionary;
    w := K;
  }
}

```

In 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 [4], 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. We have 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.

4.1 PRACTICAL COMPARISON OF STATISTICAL COMPRESSION TECHNIQUES

Table – I shows the comparative analysis between various Statistical compression 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 can not 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.

The overall performance in terms of average BPC of the above referred Statistical coding methods are shown in Fig 1.

Table I. Comparison of BPC for different Statistical Compression techniques

| S.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. | book1 | 768771 | 8.17 | 4.83 | 4.57 | 4.56 | 4.55 |
| 3. | book2 | 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. | obj1 | 21504 | 7.21 | 6.57 | 6.45 | 6.11 | 5.97 |
| 6. | obj2 | 246814 | 8.05 | 6.50 | 6.33 | 6.31 | 6.07 |
| 7. | paper1 | 53161 | 8.12 | 5.34 | 5.09 | 5.04 | 4.98 |
| 8. | paper2 | 82199 | 8.14 | 4.94 | 4.68 | 4.65 | 4.63 |
| 9. | progc | 39611 | 8.10 | 5.47 | 5.33 | 5.26 | 5.23 |
| 10. | progl | 71646 | 7.73 | 5.11 | 4.85 | 4.81 | 4.76 |
| 11. | progp | 49379 | 7.47 | 5.28 | 4.97 | 4.92 | 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 |

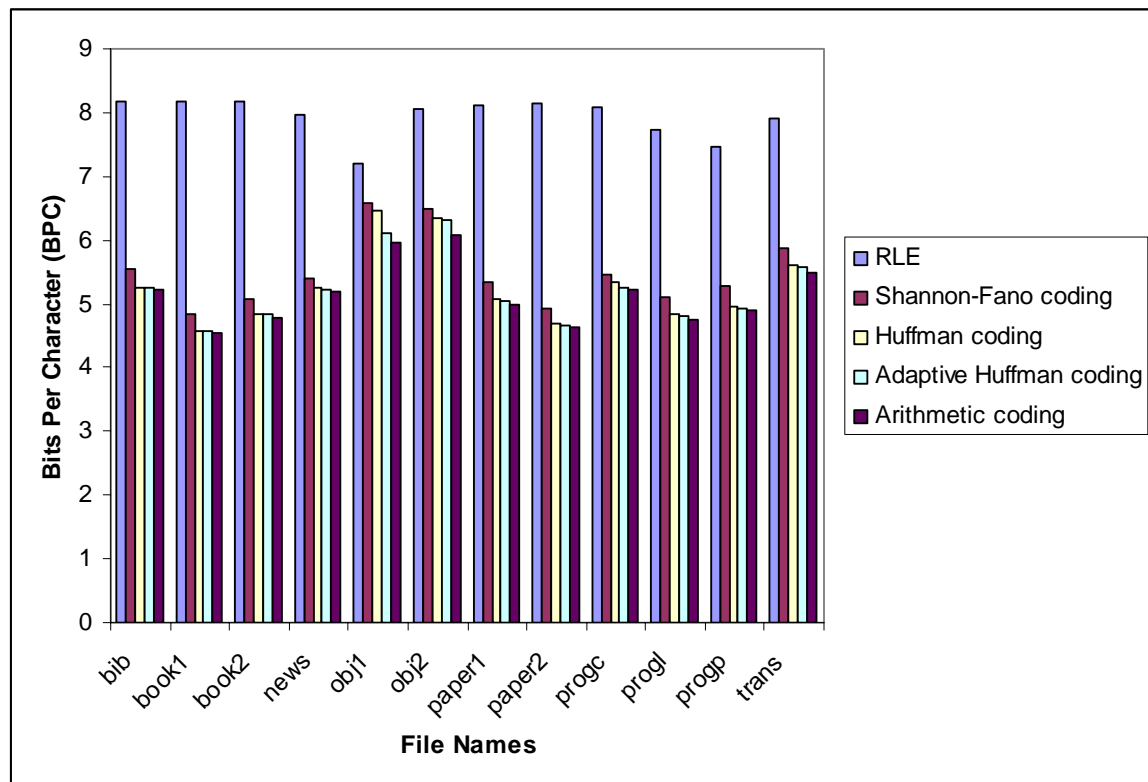


Fig 1. Chart showing Compression rates for various Statistical Compression techniques

The overall behaviour of Shannon-Fano coding, Static Huffman coding and Adaptive Huffman coding is very similar with Arithmetic coding achieving the best average compression. The reason for this is the ability of this algorithm to keep the coding and the modeler

separate. Unlike Huffman coding, no code tree needs to be transmitted to the receiver. Here, encoding is done to a group of symbols, not symbol by symbol, which leads to higher compression ratios. One more reason is its use of fractional values which leads to no code waste.

4.2 PRACTICAL COMPARISON OF LEMPEL ZIV ALGORITHMS

This section deals with comparing the performance of Lempel-Ziv algorithms. LZ algorithms considered here are divided into two categories: those derived from LZ77 and those derived from LZ78. The BPC measurements are referred from [1]. Table – II shows the comparison of various algorithms that are derived from LZ77 (LZ77, LZSS, LZH and LZB). Table – III shows the comparative analysis of algorithms that are derived from LZ78 (LZ78, LZW and LZFG). The BPC values that are referred from [1] are based on the following parameters.

The main parameter for LZ77 family is the size of the window on the text. Compression is best if the

window is as big as possible but not bigger than the text, in general. Nevertheless, larger windows yield diminishing returns. A window as small as 8000 characters will perform much better, and give a result nearly as good as the ones derived from the larger windows. Another parameter which limits the number of characters is needed for some algorithms belonging to LZ family. Generally a limit of around 16 may work well. For LZ77, LZSS and LZB the storage (characters in window) were assumed to be of 8 KB and for LZH it was assumed as 16 KB.

Regarding LZ78 family, most of the algorithm requires one parameter to denote the maximum number of phrases stored. For the above mentioned LZ78 schemes, except LZ78 a limit of 4096 phrases was used.

Fig 2 shows a comparison of the compression rates for the different LZ77 variants.

Table II. Comparison of BPC for the different LZ77 variants

| <i>S.No</i> | <i>File Names</i> | <i>File Size</i> | <i>LZ77</i> | <i>LZSS</i> | <i>LZH</i> | <i>LZB</i> |
|-------------|--------------------|------------------|-------------|-------------|-------------|-------------|
| | | | <i>BPC</i> | <i>BPC</i> | <i>BPC</i> | <i>BPC</i> |
| 1. | bib | 111261 | 3.75 | 3.35 | 3.24 | 3.17 |
| 2. | book1 | 768771 | 4.57 | 4.08 | 3.73 | 3.86 |
| 3. | book2 | 610856 | 3.93 | 3.41 | 3.34 | 3.28 |
| 4. | news | 377109 | 4.37 | 3.79 | 3.84 | 3.55 |
| 5. | obj1 | 21504 | 5.41 | 4.57 | 4.58 | 4.26 |
| 6. | obj2 | 246814 | 3.81 | 3.3 | 3.19 | 3.14 |
| 7. | paper1 | 53161 | 3.94 | 3.38 | 3.38 | 3.22 |
| 8. | paper2 | 82199 | 4.1 | 3.58 | 3.57 | 3.43 |
| 9. | progc | 39611 | 3.84 | 3.24 | 3.25 | 3.08 |
| 10. | progl | 71646 | 2.9 | 2.37 | 2.2 | 2.11 |
| 11. | progp | 49379 | 2.93 | 2.36 | 2.17 | 2.08 |
| 12. | trans | 93695 | 2.98 | 2.44 | 2.12 | 2.12 |
| | Average BPC | | 3.88 | 3.32 | 3.22 | 3.11 |

The output of Table – II reveals that the Bits Per Character is significant and most of the files have been compressed to a little less than half of the original size. Of LZ77 family, the performance of LZB is significant compared to LZ77, LZSS and LZH. The average BPC

which is significant as, shown in Table – II, which is 3.11.

Amongst the performance of the LZ77 family, LZB outperforms LZH. This is because, LZH generates an optimal Huffman code for pointers whereas LZB uses a fixed code.

Fig 2 shows a comparison of the compression rates for the different LZ77 variants

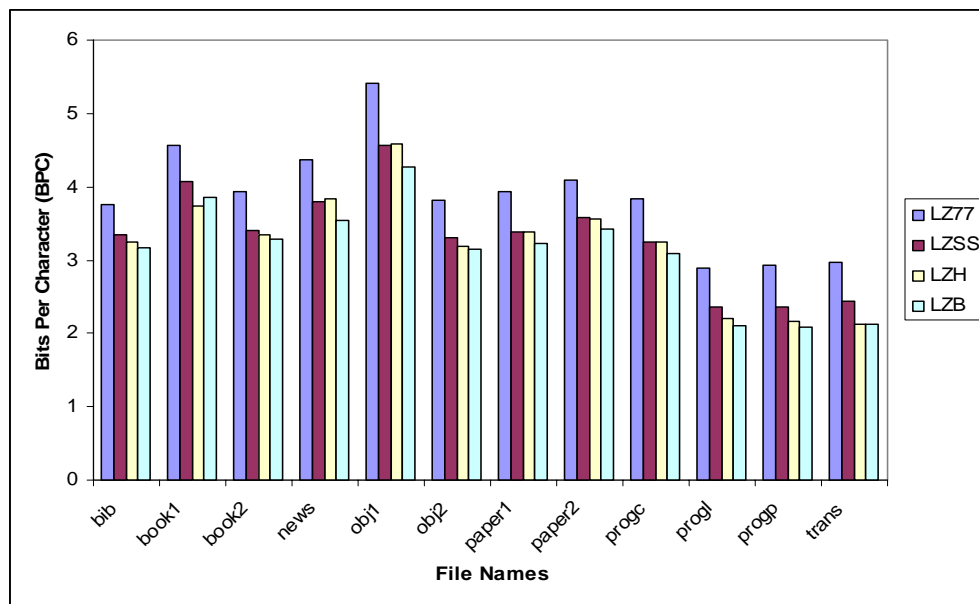


Fig 2. Chart showing Compression rates for the LZ77 family

Table III. Comparison of BPC for the different LZ78 variants

| S.No | File Names | File Size | LZ78 | LZW | LZFG |
|------|--------------------|-----------|-------------|-------------|-------------|
| | | | BPC | BPC | BPC |
| 1. | Bib | 111261 | 3.95 | 3.84 | 2.9 |
| 2. | book1 | 768771 | 3.92 | 4.03 | 3.62 |
| 3. | book2 | 610856 | 3.81 | 4.52 | 3.05 |
| 4. | news | 377109 | 4.33 | 4.92 | 3.44 |
| 5. | obj1 | 21504 | 5.58 | 6.3 | 4.03 |
| 6. | obj2 | 246814 | 4.68 | 9.81 | 2.96 |
| 7. | paper1 | 53161 | 4.5 | 4.58 | 3.03 |
| 8. | paper2 | 82199 | 4.24 | 4.02 | 3.16 |
| 9. | progc | 39611 | 4.6 | 4.88 | 2.89 |
| 10. | progl | 71646 | 3.77 | 3.89 | 1.97 |
| 11. | progp | 49379 | 3.84 | 3.73 | 1.9 |
| 12. | trans | 93695 | 3.92 | 4.24 | 1.76 |
| | Average BPC | | 4.26 | 4.90 | 2.89 |

We have tried to infer from Table – III the compression performance of LZ78 family. Most of the ASCII files are compressed to just less than half of the original size and within each file the amount of compression is consistent. The LZW method, having no boundary, accepts phrases and so the compression

expands the file ‘obj2’ by 25%, which is considered as a weakness of this approach. Also from Table – III it is obvious that the performance of LZFG is the best amongst these methods, giving an average BPC of 2.89 which is really significant. Amongst LZ78 family, LZFG’s performance is the best because the scheme that

it uses is carefully selected codes to represent pointers which is like the best scheme in the LZ77 family. Fig 3

represents a comparison of the compression rates for the LZ78 family.

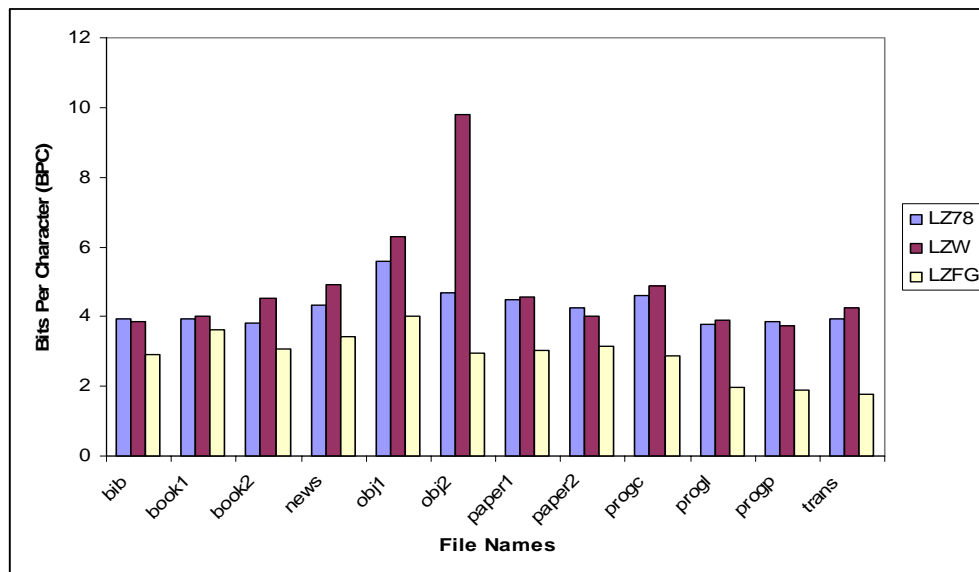


Fig 3. Chart showing compression rates for the LZ78 family

V. CONCLUSION

We have 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] Bell T.C, Cleary J.G, and Witten I.H., "Text Compression", Prentice Hall, Upper Saddle River, NJ, 1990.
- [2] Faller N., "An adaptive system for data compression", *In Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, pages 593-597, Piscataway, NJ, 1973. IEEE Press.
- [3] Fano R.M., "The Transmission of Information", Technical Report No. 65, Research Laboratory of Electronics, M.I.T., Cambridge, Mass.; 1949.
- [4] Fiala E.R., and D.H. Greene, "Data Compression with finite windows", *Communications of the ACM* 32(4):490-505, 1989.
- [5] Gallager R.G., "Variations on a theme by Huffman", *IEEE Transactions on Information Theory*, IT-24(6):668-674, November 1978.
- [6] 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.
- [7] Khalid Sayood, "Introduction to Data Compression", 2nd Edition, San Francisco, CA, Morgan Kaufmann, 2000.
- [8] Knuth D.E., "Dynamic Huffman coding", *Journal of Algorithms*, 6(2):163-180, June 1985.
- [9] Langdon G.G., "An introduction to arithmetic coding", *IBM Journal of Research and Development*, 28(2), pp. 135-149, March 1984.
- [10] Mohammad Banikazemi, "LZB: Data Compression with Bounded References", *Proceedings of the 2009 Data Compression Conference*, IEEE Computer Society, 2009.
- [11] Pasco.R., "Source coding algorithms for fast data compression", Ph.D thesis, Department of Electrical Engineering, Stanford University, 1976.
- [12] 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.
- [13] Rissanen J., "Generalised Kraft inequality and arithmetic coding", *IBM Journal of Research and Development*, 20, pp. 198-203, 1976.
- [14] Rissanen J and Langdon G.G., "Arithmetic coding", *IBM Journal of Research and Development*, 23 (2), pp. 149-162, 1979.
- [15] Shannon C.E., "A mathematical theory of communication," *Bell Sys. Tech. Jour.*, vol. 27, pp. 398-403; July, 1948.
- [16] Storer J and Szymanski T.G., "Data compression via textual substitution", *Journal of the ACM* 29, pp. 928-951, 1982.
- [17] Vitter J.S., "Design and analysis of dynamic Huffman codes", *Journal of the ACM*, 34(4):825-845, October 1987.
- [18] Vitter J.S., "Dynamic Huffman coding", *ACM Transactions on Mathematical Software*, 15(2):158-167, June 1989.
- [19] Welch T.A., "A technique for high-performance data compression", *IEEE Computer*, 17, pp. 8-19, 1984.
- [20] Ziv. J and Lempel A., "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory* 23 (3), pp. 337-342, May 1977.
- [21] 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.