CHAPTER 3

PRINCIPLES OF DATA COMPRESSION TECHNIQUES

3.1 INTRODUCTION

Compression is the art or science of representing information in a compact form. To create these compact representations by identifying and using structures that also exists in the data. Data can be characters in a text, samples of speech or images or sequence of numbers which are generated by other applications. Data compression is essential need for modern day to day life, because information are generated and used in digital form and is represented by array of data. In applications involving multimedia huge bytes of data are needed to represent information. So effective compression algorithm is applied to get the compactness for transmission in communication systems and advanced multimedia applications.

3.2 FUNDAMENTALS OF COMPRESSION TECHNIQUES

A compression technique consists of two basic components such as an encoding (Compression) and decoding (De compression) process. The encoding algorithm that takes input data (K) and generates compressed data C(K) which is less number of bits compared to input data 'K' and a decoding algorithm that reconstructs the original data 'K^1' from the compressed data C(K) are shown in Figure 3.1. Encoding and decoding of information together is called compression technique or algorithm.

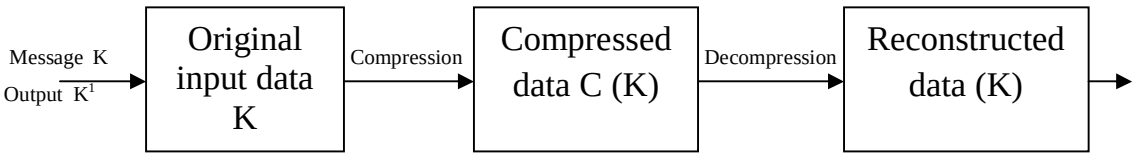

Figure 3.1 Block diagram of basic compression principle

3.2.1 Advantages of Data Compression

(i) It reduces the data storage

(ii) The user can experience rich quality signals for data representation

(iii) Enhanced Data security can also be achieved by proper encryption methods.

(iv) The rate of input – output operations in a computing device can be greatly increased due to shorter presentation of data.

(v) Data Compression also reduces the cost of backup and recovery of data in computer systems by storing the backup of large database files in compressed format.

3.2.2 Disadvantages of Data Compression

(i) Extra overhead applied in encoding and decoding process is one of the most serious drawbacks of data compression, which discourages its use in some areas.

(ii) Reliability of the records gets reduced by Data compression.

(iii) Compressed, sensitive data transmitted through a noisy communication channel is risky because the burst errors introduced by the noisy channel can destroy the transmitted data.

(iv) Disorder of data properties of a compressed data will result in compressed data different from the original data.

(v) In many hardware and system implementation, the extra complexity added by data compression can increase cost of the system reduce its efficiency.

3.3 CLASSIFICATION OF COMPRESSION TECHNIQUE

Based on the requirement of data compression it is broadly classified in to lossy compression scheme and lossless compression scheme is shown in Figure 3.2. Lossy compression algorithm can only reconstruct an approximation of the original message that is not an exact replica of input data. It is applied to images and sound where a least bit of loss in resolution is usually undetectable or acceptable level. Lossless algorithm, which can reconstruct the original message accurately, that is exact replica of original input data. It is typically used in text data, scientific data such as medical, industrial application and also in military application.

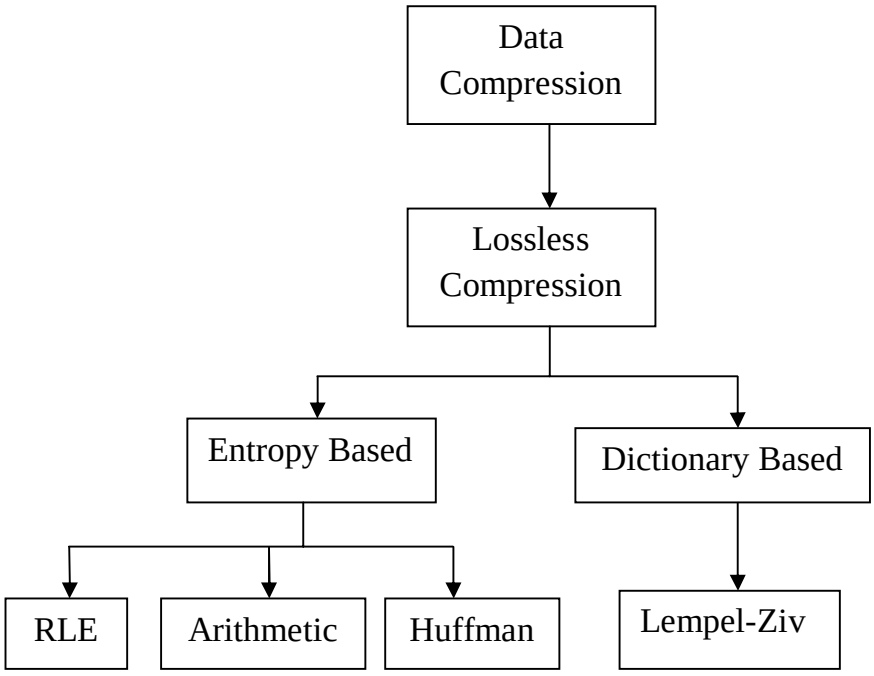

Figure 3.2 Classification of data compression

3.4 LOSSLESS COMPRESSION

As their name implies, there is no loss of information. The original data (K) can be recovered exactly from the compressed data C (K). It is generally used for applications that cannot tolerate any difference between the original and reconstructed data. Text data compression is an important area for lossless compression. It is very important that the reconstructed data is identical to the input data. Even small differences will make serious problems in some important applications like military field. In the battle field instead of receiving the message “Do not move forward” it might be reconstructed as “Do now move forward”. So it is not good to allow mismatches to appear in the compression process.

3.4.1 Entropy Based

Entropy coding can be used for different media regardless of the medium specific. The data to be compressed are seen as a sequence of digital data values and their definitions are ignored. It is lossless as the data prior to encoding is identical to the data after decoding and no information is lost. As example entropy coding can be used for compression of any type of data in a file system.

3.4.1.1 Run length coding

The simple and popular data compression technique is called Run length encoding (RLE) is designed especially for data with strings of repeated symbols (The length of the string is called a Run). The main idea is to encode repeated symbols as pairs i.e. the length of the string and the symbol. This is most useful on data that consists of many runs. Typical examples are graphic images like animations, icons, line drawings etc. It won’t be useful with files

that don't have many runs as it could potentially increase the file size. The below given example Table 3.1 explains in detail about Run length encoding.

The main idea used is to encode repeated symbols as pair. The string "svvvsssssaappppavsssssssspvvpps" of length is 30 bytes and represented by 12 integers and 12 characters. So finally it can be encoded in 24 bytes, The Example.2 uses special symbols i.e. only strings of a length longer or equal to some fixed number say usually 3, are replaced by special symbol and a pair. The special symbol can be selected as any one of the unused characters. This needs additional preprocessing of data to find unused characters. The string "svvvsssssaappppavsssssss" of length is 24 bytes is represented by 5 characters plus 3 special symbols and 3 pairs. This can be encoded on 14 bytes.

Table 3.1 Run length encoding (RLE) example

Method	Data to Compress	Length of the string	Compressed Data	Length of Compressed data
Example 1	"svvvsssssaappppavsssssspvvpps"	30 bytes	<1,s>,<3,v>,<5,s>,<2,a>,<4,p>, <1,a>,<1,v>,<7,s>,<1,p>,<2,v>, <2,p>,<1,s>.	24bytes
Example 2	"svvvsssssaappppavsssssss"	24 bytes	sX<3,v>,aaX<4,p>,avX <7,s>	14 bytes

3.4.1.2 Huffman Coding

The Huffman coding algorithm is named after its inventer, David Huffman. It is more successful method used for compression. The idea behind Huffman coding is to find a way to compress the storage of data using variable length codes. Huffman coding finds the optimal way to take advantage of varying character frequencies in a particular file, on average, using this coding on standard files can shrink them anywhere from 10% to

30% depending on the character distribution. The idea behind the coding is to give less frequent characters and groups of characters, long codes. Also the coding is constructed in such a way that no two constructed code are prefixes to each other. This plays a crucial role for easily deciphering the code. The Huffman algorithm is simple and can be described in terms of creating a Huffman code tree. The procedure for building this tree is,

- Start with list of free nodes.
- Each node corresponds to a particular symbol in the alphabet
- Select two free nodes with lowest weight from the list
- Create a parent node for these two nodes selected and the weight is equal to the weight of the sum of two child nodes
- Two child nodes are removed from the list and the parent node is added to the list of free nodes.
- Repeat the process until only a single tree remains as shown in Figure 3.3.

Example

A source generates 4 different symbols a_1, a_2 , a_3, a_4 with probability of {0.4 , 0.35, 0.2 , 0.05 } $a_1 = 0.4$, $a_2 = 0.35$, $a_3 = 0.2$, $a_4 = 0.05$.

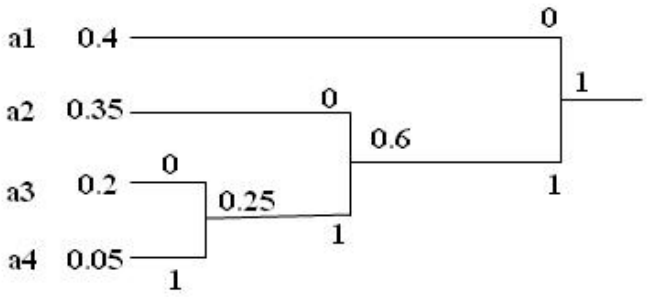

Figure 3.3 Huffman code tree

The tree can then be read backwards from right to left by assigning bits to different branches as shown in Table 3.2.

Table 3.2 Bit assignment based on frequency of the character

Symbol	Code
a_1	0
a_2	10
a_3	110
a_4	111

3.4.1.2.1 Limitations of huffman coding

(i) Huffman code will be optimal, if exact probability distribution of the Source symbols is known

(ii) Each symbol is encoded with integer number of bits

(iii) When the source statistics changes, then Huffman code is not much efficient.

(iv) Use of the multiplications in the encoding and decoding process to compute the ranges, may be prohibitive for many real time fast applications.

3.4.1.3 Arithmetic coding

One of the most powerful compression techniques is arithmetic coding. It is a form of entropy encoding used in lossless data compression Huffman coding suffers from the fact that an integral value of bits is needed to code a character. Arithmetic coding completely bypass the idea of replacing every input symbol with a codeword. Instead it replaces a stream of input symbols with single floating point number. Arithmetic coding was developed by (Elias) in the early 1960's and further developed by (Pasco),

(Rissanen) and (Langdon). The main aim of arithmetic coding is to assign an interval to each symbol. Then each interval is assigned a decimal number. The algorithm starts with an interval of 0.0 and 1.0. Based on input symbol probability the interval is subdivided in to smaller interval, which will act as new interval and is again divided into small parts. This is repeated for each and every symbol until the final interval uniquely determines the input data.

A small example will be used to illustrate the idea of arithmetic coding. An alphabet with symbols S_0, S_1,... S_n, where each symbol has a probability of occurrence of p_0, p_1, ... p_n such that $\sum p_i = 1$. From the fundamental information theory, optimal coding for Si requires - ($p_i \times \log2 (p_i)$) bits. Usually the optimal number of bits would be fractional. Similar to Huffman coding, arithmetic coding provides the ability to represent symbols with fractional bits. As $\sum p_i = 1$, where probability p i, with unique non-overlapping range of values between 0 and 1. There's no magic in this, just creating ranges on a probability line.

For example, suppose an alphabet 'a', 'b', 'c', 'd', and 'e' with probabilities of occurrence of 30%, 15%, 25%, 10%, and 20%. Then choose the following range assignments to each symbol based on its probability:

Table 3.3 Arithmetic coding model sample symbol ranges

Symbol	Probability	Cumulative Probability	Range
a	30 %	0.30	[0.00 , 0.30)
b	15 %	0.45	[0.30 , 0.45)
c	25 %	0.70	[0.45 , 0.70)
d	10 %	0.80	[0.70 , 0.80)
e	20 %	1.00	[0.80 , 1.00)

By assigning each symbol its own unique probability range, a single symbol can be encoded by its range. Using this approach to encode a string as a series of probability ranges, that doesn't compress, additional symbols may be encoded by restricting the current probability range by the range of a new symbol being encoded.

Lower bound = 0

Upper bound = 1

current range = upper bound - lower bound

upper bound = lower bound + (current range × upper bound of new symbol)

lower bound = lower bound + (current range × lower bound of new symbol)

Example:

Encode the string "ace" using the probability ranges from Table 3.3 Start with lower and upper probability bounds of 0 and 1.

Encode 'a'

current range = 1- 0 = 1

upper bound = 0 + (1 × 0.3) = 0.3

lower bound = 0 + (1 × 0.0) = 0.0

Encode 'c'

current range = 0.3 - 0.0 = 0.3

upper bound = 0.0 + (0.3 × 0.70) = 0.210

lower bound = 0.0 + (0.3 × 0.45) = 0.135

Encode 'e'

current range = 0.210 - 0.135 = 0.075

upper bound = 0.135 + (0.075 × 1.00) = 0.210

lower bound = 0.135 + (0.075 × 0.80) = 0.195

The string "ace" ay be encoded by any value within the probability range [0.195, 0.210).

The decoding process must start with an encoded value which is represented as string. The encoded value usually lays within the lower and upper probability range bounds of the string. As the encoding process restricts the ranges (without shifting), the initial value falls within the first encoded symbol range. The symbols successively encoded would be identified by removing the scaling applied by the known symbol. To achieve this, subtract the lower probability range and multiply it by the symbol size range. Based on the discussion above, decoding a value may be performed by the following steps.

Encoded value = encoded input

current range = upper bound of new symbol - lower bound of new symbol

encoded value = (encoded value - lower bound of new symbol) ÷ current range

Example:

Using the probability ranges from Table 3.3 decode the three character string encoded as 0.02

Decode first symbol

0.20 is within [0.00, 0.30)

0.20 encodes ‘a’

Remove effects of ‘a’ from encode value

Current range = 0.30 – 0.00 = 0.30

Encoded value = (0.20 – 0.0) / 0.30 = 0.67

Decode second symbol

0.67 is within [0.45, 0.70)

0.67 encodes ‘c’

Remove effects of ‘c’ from encode value

Current range = 0.70 – 0.45 = 0.25

Encoded value = (0.67 – 0.45) / 0.25 = 0.88

Decode the third symbol

0.88 is within [0.80, 1.00)

0.88 encodes ‘e’

The encoded string is “ace”.

3.4.1.3.1 Limitations of arithmetic coding

(i) The encoded value is not unique because any value with in the final range can be considered as the encoded message .It is desirable to have a unique binary code for the encoded message

(ii) The encoding algorithm does not transmit anything until encoding of the entire message has been completed. As a result, the decoding algorithm can not start until it has received the complete encoded data. The above two limitations can be overcome by using the binary arithmetic coding.

(iii) The precision required to represent the intervals grows with length of the message.

(iv) Use of the multiplications in the encoding and decoding process in order to compute the ranges in every step, may be prohibitive for many real time fast applications.

3.4.2 Dictionary Based

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. Repetitions are replaced by (shorter) references to a "dictionary" containing the original.

3.4.2.1 Lempel ziv algorithms

The Lempel Ziv Algorithm is an algorithm for lossless data compression. It is not a single algorithm, but stems from algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 and 1978 is shown in Figure 3.4.

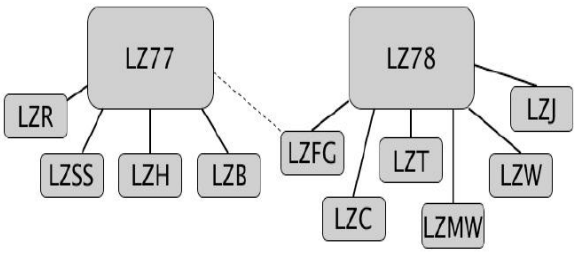

Figure 3.4 Classification of lempel ziv family

- **LZ77:**

Jacob Ziv and Abraham Lempel have presented their dictionary-based scheme in 1977 for lossless data compression (Jacob Ziv et al 1977). 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 would be repeated. If any repetition occurs, it will be encoded as a pointer to an earlier occurrence and the pointer would be accompanied by the number of characters to be matched. It is simple adaptive algorithm which requires no prior knowledge of the source and seems to require no assumptions about the characteristics of source. Also the dictionary is simply a portion of the previously encoded

sequence. The input sequence would be examined by the encoder 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 coded. 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 the match. If there is no match, output does not contain any pointers. In the reference output of LZ77, is 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. No match means, 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 constants. Also the LZ77 compression performance mainly depends on these values. Normally the offset is coded on 12–16 bits, and limited from 0 to 2^{16} symbols. Hence there is no need to remember more than 65535 last seen symbols in the sliding window. The match length is usually encoded on a byte, which provides maximum length, say 255.

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 quick. The decoding is much faster than encoding and it is one of the important features of this process. Most of the LZ77 compression time is used to search for the longest match, whereas 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, like LZSS, LZH and LZB. LZSS (Storer et al 1982) removes the requirement of mandatory inclusion of the next non-matching symbol into each codeword. 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. Coding is performed in two phases. The first same as LZSS, and the second uses statistics measured in the first to code pointers and explicit characters using Huffman coding.

LZB used by Mohammad (2009) an elaborate scheme for encoding the references and lengths with varying sizes. Regardless of the length of the phrase, every LZSS pointer is of same size. In practice a better compression is achieved by having different sized pointers, as some phrase lengths are much more likely to occur. LZB uses different technique which codes 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.

- **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 Jacob Ziv (1978) 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 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. 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 (Welch 1984), 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’.

Encoding Algorithm

1. Initialize the dictionary to contain all blocks of length one (W= {a,b}).
2. Search for the longest block **W** which has appeared in the dictionary.
3. Encode **W** by its index in the dictionary shown in Table 3.4.

4. Add **W** followed by the first symbol of the next block to the dictionary.

5. Go to Step 2.

The following example illustrates how the encoding is performed.

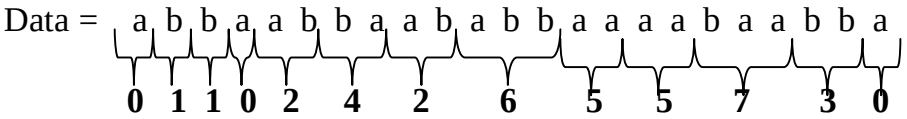

Table 3.4 Dictionary for lembel ziv algorithm

Dictionary			
Index	**Entry**	**Index**	**Entry**
0	a	7	baa
1	b	8	aba
2	ab	9	abba
3	bb	10	aaa
4	ba	11	aab
5	aa	12	baab
6	abb	13	bba

3.5 PROPOSED METHOD

Data Compression is always useful for encoding information using fewer bits than the original representation. There are many issues the size of information would affect data transmission as well as can affect the cost too. Here the idea of simple byte lossless compression algorithm reduces the standard 8 bit coding technique to some specific 5 bit coding technique. This method will reduce the size of a character considerably when the character is lengthy and the compression ratio is not affected by the content of the character.

3.5.1 Simple Byte Compression(SBC) Algorithm

Assume random sample with 8 Characters long, each character in the random sample need 8 bits to represent it. So 64 bits are needed to represent the total random sample. But using simple byte compression algorithm 5 bits are enough to represent a character instead of 8 bits. By using simple byte compression algorithm 40 bits are enough to represent the same 8 characters instead of 64 bits. The following flow diagram Figure 3.5 will explain how the compression is achieved effectively.

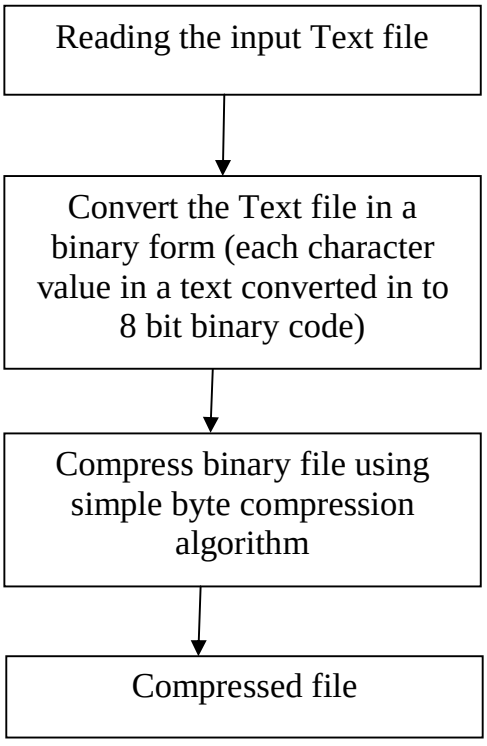

Figure 3.5 Flow diagrams for simple byte compression algorithm

Step 1: Read the input Text

Step2: Assign values to the respective input text characters from the lookup Table 3.5 shown below.

Table 3.5 Lookup table for text characters

Character	Value	Character	Value	Character	Value	Character	Value
a	1	o	15	C	29	Q	43
b	2	p	16	D	30	R	44
c	3	q	17	E	31	S	45
d	4	r	18	F	32	T	46
e	5	s	19	G	33	U	47
f	6	t	20	H	34	V	48
g	7	u	21	I	35	W	49
h	8	v	22	J	36	X	50
i	9	w	23	K	37	Y	51
j	10	x	24	L	38	Z	52
k	11	y	25	M	39	“ “	0
l	12	z	26	N	40	“ , “	53
m	13	A	27	O	41	“ . “	54
n	14	B	28	P	42	-	-

Step3: Convert the assigned values into 8 bit binary form. So this needs 8 bytes for storing 8 characters.

K_E=/k_0,k_1,k_2,k_3,k_4,k_5,k_6,k_7/$k_8$$k_9$,$k_{10}$,$k_{11}$,$k_{12}$,$k_{13}$,$k_{14}$,$k_{15}$/$k_{16}$,$k_{17}$,$k_{18}$,$k_{19}$,$k_{20}$,$k_{21}$,$k_{22}$,$k_{23}$

/ k_{24}, k_{25}, k_{26}-- k_n

Where n = 64 bit

k = Input data

K_E = Encoded input

Step 4: In each byte remove three bits, starting from the most significant bits and retain the lower five bits.

K_E = /k_3,k_4,k_5,k_6,k_7/ k_{11},k_{12},k_{13},k_{14},k_{15}/ k_{19},k_{20},k_{21},k_{22},k_{23}/ k_{27},k_{28},k_{29},k_{30},k_{31} /

k_{35}, k_{36}, k_{37}, k_{38}, k_{39} / ----k_{64} /

Step 5: Then rearrange and append the lower five bits of each character to form an array of 8 bits.

K_C=/k_3,k_4,k_5,k_6,k_7,k_{11},k_{12},k_{13}/k_{14},k_{15},k_{19},k_{20},k_{21},k_{22},k_{23},k_{27}/k_{28},k_{29},k_{30},k_{31},k_{35},k_{36}, k_{37}, k_{38} / k_{39}------- / ----- k_{64} /

K_C - Compressed data

Finally the 8 bytes of information is reduced to 5 bytes of information.

Example:

Let's assume suppose that a text contains 8 characters like 'birthday'

From the Look up Table 3.5 the values of the characters in the given example are;

b – 2, i – 9, r – 18, t – 20, h – 8, d – 4, a – 1, y – 25.

Convert decimal value in to 8 bit binary form and all are arranged as follows,

K_E = /00000010/00001001/00010010/00010100/00001000/ 00000100/00000001/00011001/

K_E = 64 bits / 8 bytes

Then chop 3 bits from left side and extract 5 least significant bits as follows

K_E = /00010/01001/10010/10100/01000/ 00100/00001/11001/

Then rearrange these bits in an array of bytes as follows;

K_C = /00010010/01100101/01000100/0 0010000/00111001/

K_C = 40 bits / 5 bytes

3.5.2 Simple Byte Decompression (SBD) Algorithm

When an array of compressed data is given, each 8 bit in the array of received data is regrouped into 5 bits of binary information. These 5 bits of binary information is changed to decimal value. To the obtained decimal values characters are assigned from the Look up Table 3.5. Thus the compressed information received was reconstructed properly without any loss. The following flow diagram Figure 3.6 will explain how the decompression is achieved successfully.

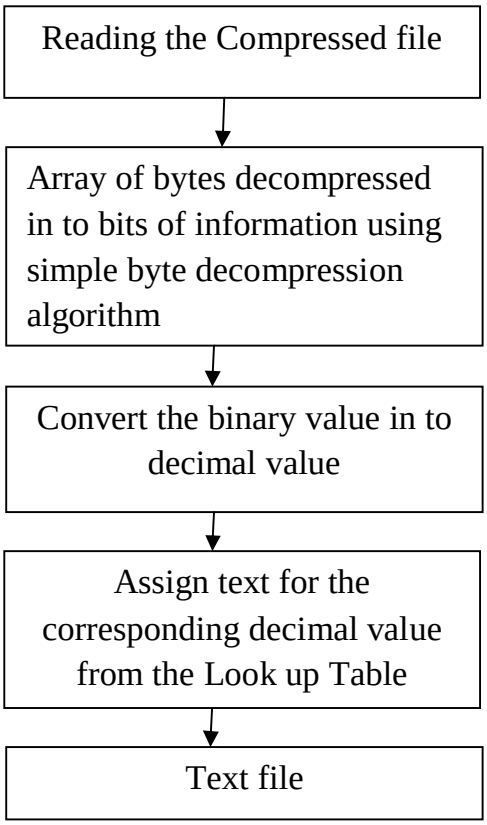

Figure 3.6 Flow diagrams for simple byte decompression algorithm

Step1: Received 5 byte of compressed information.

K_C=/k_3,k_4,k_5,k_6,k_7,k_{11},k_{12},k_{13}/k_{14},k_{15},k_{19},k_{20},k_{21},k_{22},k_{23},k_{27}/k_{28},k_{29},k_{30},k_{31},k_{35},k_{36}, k_{37}, k_{38} / k_{39}------- / ----- k_{64} /

Step2: Each 8 bit array can be rearranged to a set of 5 bits

K_C=/k_3,k_4,k_5,k_6,k_7,k_{11},k_{12},k_{13}/k_{14},k_{15},k_{19},k_{20},k_{21},k_{22},k_{23},k_{27}/
k_{28},k_{29},k_{30},k_{31},k_{35},k_{36},k_{37},k_{38} / k_{39}-------/-- --- k_{64} /

K_D = /k_3,k_4,k_5,k_6,k_7 / k_{11},k_{12},k_{13},k_{14},k_{15} / k_{19},k_{20},k_{21},k_{22},k_{23} / k_{27},k_{28},k_{29},k_{30},k_{31} /

k_{35}, k_{36}, k_{37}, k_{38}, k_{39} / ----k_{64} /

Step 3: The rearranged 5 bit binary value is converted to a corresponding decimal Value

Step 4: Based on the decimal value a character will be assigned from the lookup Table 3.5. Finally the original text is obtained without any loss as shown in Figure 3.7.

Example:

The received compressed data contains 5 byte of information

K_C = /00010010/01100101/01000100/00010000/00111001/

Then split into set of 5 bits starting from left

K_D = /00010 010/01 10010 1/0100 0100/0 00100 00/001 11001/

These sets converted in to decimal value.

00010 – 2

01001 – 9

10010 – 18

10100 – 20

01000 – 8

00100 – 4

00001 – 1

11001 – 25

With help of Look up Table shown in Table 3.5 character value is assigned to the corresponding decimal value extracted from the 5 bit binary information.

2 – b, 9 – i, 18 – r, 20 – r, 8 – h, 4 – d, 1 – a, 25 – y.

Finally ‘birthday’ is decoded correctly.

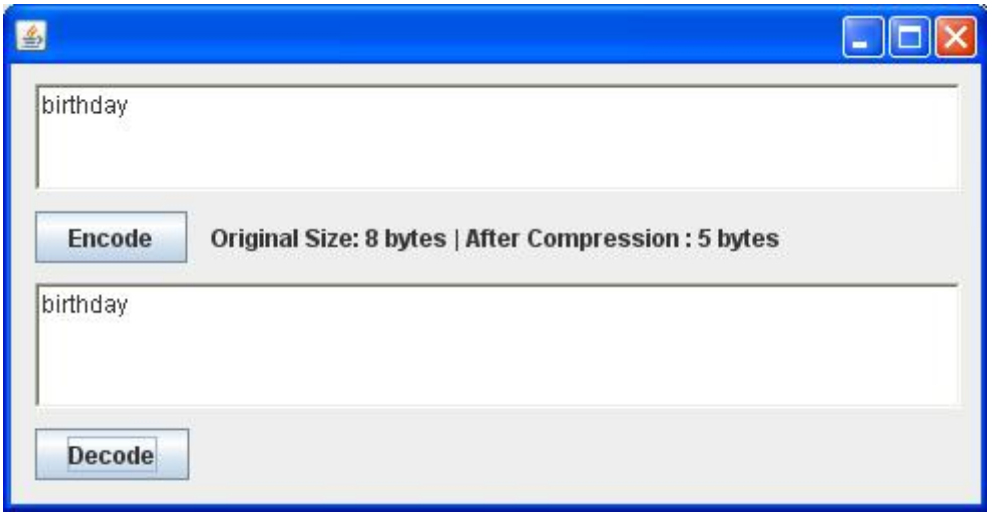

Figure 3.7 Output of text data compression

3.6 RESULTS AND DISCUSSIONS

In this section the performance of four different Lossless compression techniques like Run length encoding (RLE), Huffman coding, Arithmetic coding and Simple byte compression (SBC) algorithms are analyzed and tested with eight text files. Each is different file sizes and different contents. The input file sizes are 256 bytes, 240 bytes, 592 bytes,, 640 bytes, 816 bytes, 232 bytes, 192 bytes, 1232 bytes. To evaluate the compression effectiveness various compression parameters which are analyzed and listed below from Table 3.6 to 3.11 and Figure 3.6 to 3.12 for eight different text files.

3.6.1 Bits per Character (BPC)

It is defined as the difference between number of bits used in the compressed text and number of characters used in the original (input) text

$$\text{BPC} = \frac{\text{Number of bits used in the compressed text}}{\text{Number of characters used in original text}}$$

Table 3.6 Analysis of BPC for different compression techniques

File Name	Original Size	RLE	Huffman	Arithmetic	SBC
		BPC	BPC	BPC	BPC
Text 1	256	7.00	5.60	6.31	5.00
Text 2	240	6.40	5.93	5.65	5.03
Text 3	640	6.8	5.24	6.19	5.00
Text 4	816	15.56	6.23	5.83	5.01
Text 5	232	6.06	5.76	5.49	4.96
Text 6	192	4.66	6.45	5.97	5.00
Text 7	592	6.05	5.82	5.36	5.08
Text 8	1232	13.29	6.07	6.79	5.01
Average BPC		**8.227**	**5.887**	**5.948**	**5.011**

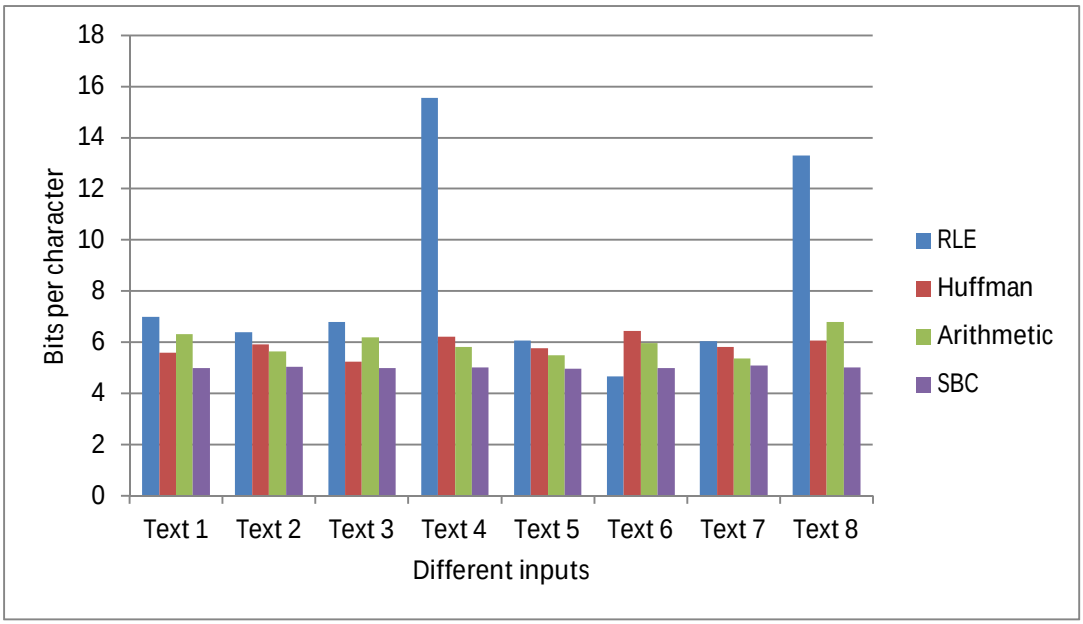

Figure 3.8 Analysis of BPC for different compression techniques

In this section the overall performance is analyzed in terms of average bits per character (BPC) for four different lossless compression techniques. All text files contain English language of different sizes. Here dictionary compression technique is not considered here, because for large file size, larger dictionary is needed for compression and decompression process, also lots of computer resources required to process and overflow problem is also occurred. From Table 3.6 and Figure 3.8 except Text file 6 all text files produced larger BPC for RLE technique. Mostly the RLE algorithm generates compressed file is larger than the original input file. This happens if the number of runs is less in the text file. RLE is performing well when the amount of run is larger in the text files. The average BPC obtained is 8.227 which are higher compared to Huffman, arithmetic and SBC. The average BPC for Huffman coding is 5.887 which is greater than SBC and lesser than RLE and arithmetic. The average BPC for arithmetic coding is 5.948 which is lesser than RLE and larger than Huffman and SBC. Due to underflow problem the result given by arithmetic is not accurate. The average BPC for SBC is 5.011 which are less compared to all lossless compression techniques

considered here. It is a best suitable method for larger and smaller files. There is no loss of characters during compression and decompression process, less computational complexity.

3.6.2 Analysis of Compression Ratio and Compression Factor

The following compression parameters are analyzed for various compression techniques like RLE, Huffman, Arithmetic and SBC using eight different text data and the corresponding performances listed here from Table 3.7 to Table 3.11 and Figure 3.9 to Figure 3.11.

3.6.2.1 Compression Ratio(C_R)

It is defined as the ratio of the number of bytes in the uncompressed representation to the number of bytes in the compressed representation.

C_R = Size of uncompressed data / Size of compressed data

C_R > 1 indicate Compression and C_R < 1 indicate Expansion

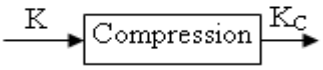

Figure 3.9 Compression ratio

K – Input data (Uncompressed data)

K_C – Compressed data

Compression ratio (C_R) = Bits in K / Bits in K_c

3.6.2.2 Compression factor (C_F)

It is reciprocal of compression ratio. It is defined as the ratio between sizes of compressed data to the size of uncompressed data. The value gives how much compression has been achieved

Table 3.7 Performance evaluation of huffman coding

File Name	Original Size	Compression Ratio(C_R)	Compression Factor(C_F)	BPC
Text 1	256	1.49	0.67114	5.60
Text 2	240	1.52	0.65789	5.93
Text 3	640	1.31	0.76335	5.24
Text 4	816	1.437	0.69589	6.23
Text 5	232	1.16	0.86206	5.76
Text 6	192	1.26	0.79365	6.45
Text 7	592	1.621	0.61690	5.82
Text 8	1232	1.57	0.63694	6.07

Table 3.8 Performance evaluation arithmetic coding

File Name	Original Size	Compression Ratio(C_R)	Compression Factor(C_F)	BPC
Text 1	256	1.42	0.70422	6.31
Text 2	240	1.582	0.63211	5.65
Text 3	640	1.601	0.62460	6.19
Text 4	816	1.292	0.77399	5.83
Text 5	232	1.32	0.7575	5.49
Text 6	192	1.17	0.85470	5.97
Text 7	592	1.12	0.89285	5.36
Text 8	1232	1.36	0.73529	6.79

Table 3.9 Performance evaluation of run length encoding

File Name	Original Size	Compression Ratio(C_R)	Compression Factor(C_F)	BPC
Text 1	256	1.683	0.59417	5.00
Text 2	240	1.572	0.63613	5.03
Text 3	640	1.6	0. 62500	5.00
Text 4	816	1.59	0.62745	5.01
Text 5	232	1.783	0.56085	4.96
Text 6	192	1.674	0.59737	5.00
Text 7	592	1.598	0.62741	5.08
Text 8	1232	1.654	0.60459	5.01

Table 3.10 Performance evaluation simple byte compression

File Name	Original Size	Compression Ratio(C_R)	Compression Factor(C_F)	BPC
Text 1	256	1.33	0.75187	7.01
Text 2	240	1.25	0.80000	6.40
Text 3	640	1.17	0.85470	6.8
Text 4	816	0.5	2	15.56
Text 5	232	1.31	0.75862	6.06
Text 6	192	1.71	0.58333	4.66
Text 7	592	1.32	0.75757	6.05
Text 8	1232	0.60	1.66	13.29

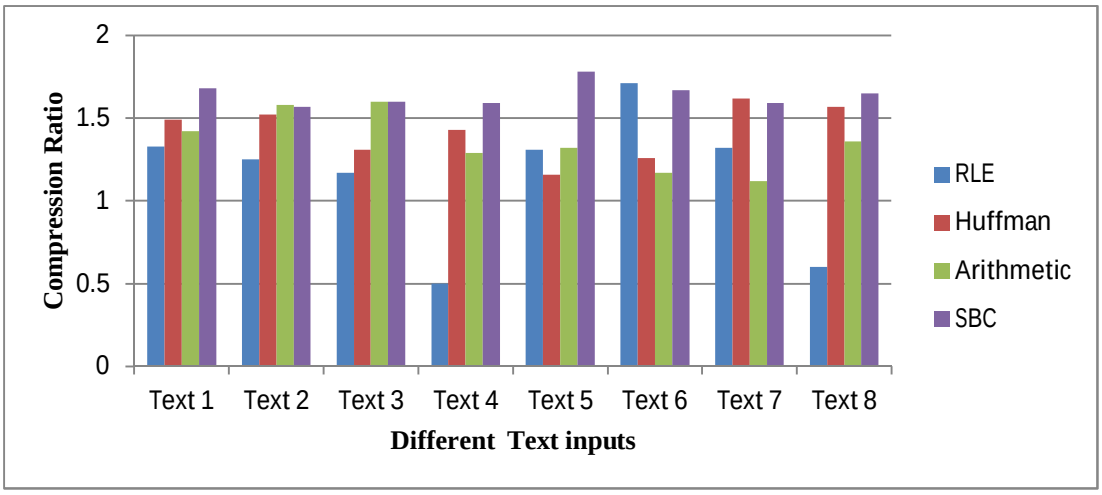

Figure 3.10 Compression ratios of different compression techniques

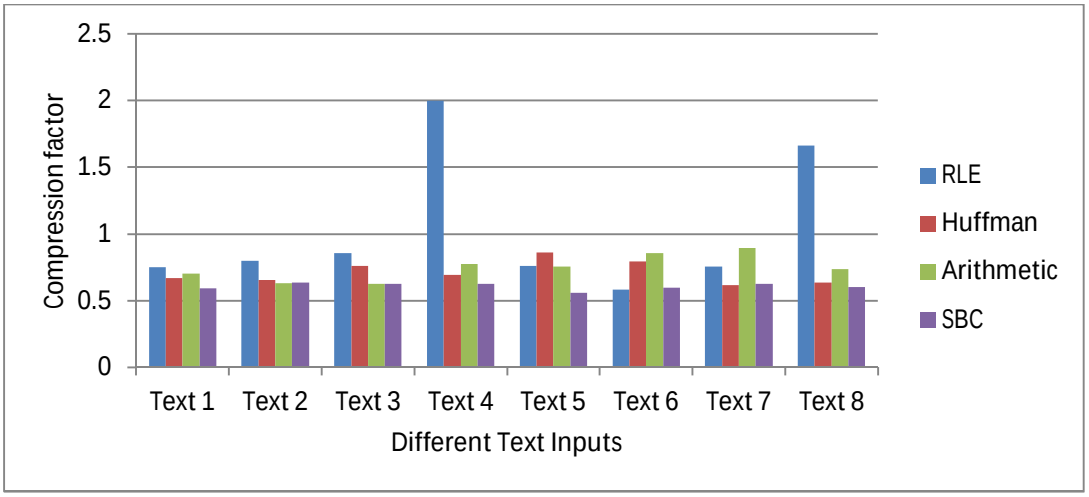

Figure 3.11 Compression factor analyses for different compression techniques

According to the result the Table 3.7 shows Huffman technique, the compression ratio is Between 1.16 to 1.621 and compression factor is between 0.61 to 0.86. The compression ratio is less compared to SBC and compression factor is higher than SBC also Huffman fails to compress the data in large amount when the symbols have skewed probability and long runs as compared to RLE and SBC coding respectively. Table 3.8 shows arithmetic technique, the compression ratio is between 1.12 to 1.601 and compression factor is between 0.62 to 0.89. 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. Table 3.9 shows RLE technique, the compression ratio between 0.5 to 1.71 and compression factor is between 2 to 0.85. The compression ratio is greater than 1 the compression is good less than 1 compression is poor, instead of compress the text file it expanded. In RLE the text file 4 & 8 are not compressed it expanded because compression ratio is less than 1. Table 3.10 shows SBC technique, the compression ratio is between 1.57 to 1.68 and compression factor is between 0.56 to 0.63 which is good compared to all other techniques. In SBC all text files the compression ratio is greater than 1.5 and the compression factor is also good compared to other methods listed here.

3.6.3 Saving Percentage (S_P)

It is also a reasonable measure of compression performance. The value gives how much percentage of space occupied by the output data of its original size.

$$S_P = \frac{\text{Size before compression} - \text{Size after compression}}{\text{Size before compression}} \%$$

Table 3.11 Saving percentage of different compression techniques

File Name	Original Size	Saving %			
		RLE	Huffman	Arithmetic	SBC
Text 1	256	24.813	32.89	29.578	40.58
Text 2	240	20.00	34.21	36.79	36.38
Text 3	640	14.53	23.665	37.54	37.5
Text 4	816	-100	30.44	22.601	37.25
Text 5	232	24.13	13.79	24.25	43.91
Text 6	192	41.67	20.63	14.53	40.26
Text 7	592	24.24	38.31	10.71	37.25
Text 8	1232	-66	36.31	26.47	39.54

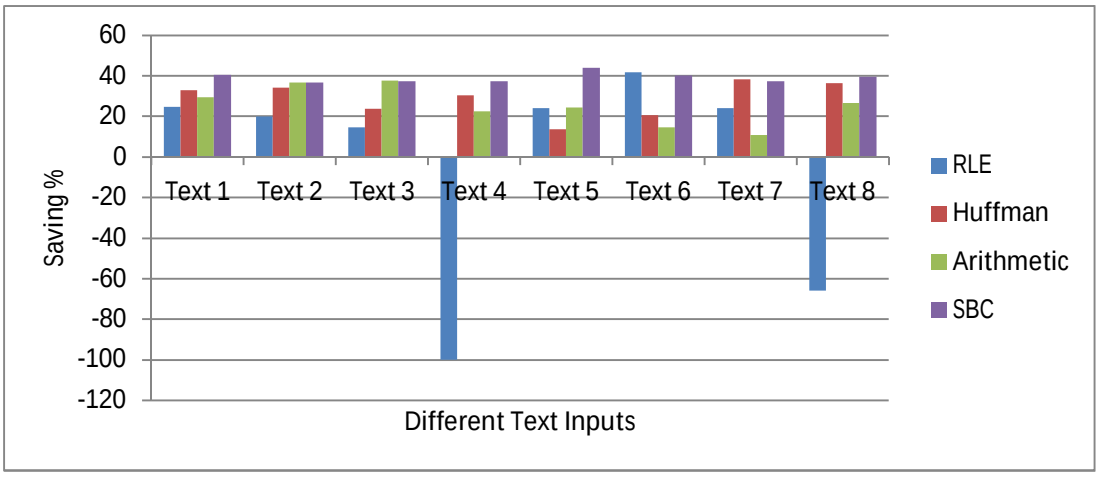

Figure 3.12 Analysis of saving percentages for different compression techniques

The saving percentages of various lossless compression techniques are analyzed and the corresponding results are obtained in the Table 3.11 and Figure 3.12. In RLE the saving percentage is from -100% to 41.67% which is very low saving percentage compared to other methods. In Huffman and arithmetic the saving percentage is almost same from 13.79% to 38.31% and 14.53% to 37.54% which are very low compared to SBC. In simple byte compression technique the saving percentage is from 36.38% to 43.91%. In SBC all text files the saving percentage is greater than 36 % so SBC is suitable for all type text data with less complexity.

3.7 CONCLUSION

In this chapter several existing lossless compression techniques are analyzed and compared with the simple byte compression (SBC) technique and the performance can be seen from Table 3.6 to 3.11 and Figure 3.6 to 3.12. Due to underflow problem in arithmetic coding the original files cannot be generated after decompression process. So the arithmetic algorithm is not considered for large applications. The RLE technique is also not considered for large and secured applications because it cannot produce good results if the runs are not long. The compression ratio and saving percentage

are low in Huffman coding compared to SBC method. By considering the compression parameters like BPC, compression ratio, compression factor and saving percentages of all existing lossless algorithms simple byte compression is considered as the most efficient algorithm. The output values of this algorithm are at an acceptable range and it shows better result for any kind of text files.