A Comparative Study of Lossless Compression Algorithm on Text Data

Amit Jain[1], Kamaljit I. Lakhtaria[2]
(Corresponding author: Amit Jain)

Department of Computer Science and Engineering[1]
Sir Padampat Singhania University, Udaipur (Raj.) India

Department of Computer Science and Engineering[2]
Sir Padampat Singhania University, Udaipur (Raj.) India
(Email: amitscjain@gmail.com)

(Received Nov. 20, 2013; revised and accepted Jan. 30, 2014)

Abstract

With increasing amount of text data being stored rapidly, efficient information retrieval and Storage in the compressed domain has become a major concern. Compression is the process of coding that will effectively reduce the total number of bits needed to represent certain information. Data compression has been one of the critical enabling technologies for the ongoing digital multimedia revolution. There are lots 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 on English text files: LZW, Huffman, Fixed-length code (FLC), and Huffman after using Fixed-length code (HFLC). All the above algorithms are evaluated and tested on different text files of different sizes. To find the best algorithm among above, comparison is made in terms of compression: Size, Ratio, Time (Speed), and Entropy. The paper is concluded by the decision showing which algorithm performs best over text data.

Keywords: Data Compression, Huffman Coding, LZW, RLE.

1 Introduction

Data compression is a technique that transforms the data from one representation to another new compressed (in bits) representation, which contains the same information but with smallest possible size [1]. The size of data is reduced by removing the excessive information. The data to be stored or transmitted at reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel for more data transmission. Similarly, compressing a file to half of its original size is equivalent to doubling the capacity of the storage medium. It may then become feasible to store the data at a higher, thus faster, level of the storage hierarchy and reduce the load on the input/output channels of the computer system.

Benefits of compression

It provides a potential cost saving associated with sending less data over switched telephone network where cost of call is usually based upon its duration.

It not only reduces storage requirements but also overall execution time.

It also reduces the probability of transmission errors since fewer bits are transferred.

It also provides a level of security against illicit monitoring [2].

Data compression can be lossless, Lossless data compression makes use of data compression algorithms that allows the exact original data to be reconstructed from the compressed data. Lossless data compression is used in many applications. For example, it is used in the popular ZIP file format and in the Unix tool gzip. Lossless compression is used when it is important that the original and the decompressed data be identical, or when no assumption can be made on whether certain deviation is uncritical. Typical examples are executable programs and source code. Some image file formats, notably PNG, use only lossless compression [3].

Another family of compression is lossy compression. A lossy data compression method is one where compressing data and then decompressing it retrieves data that may well be different from the original, but is "close enough" to be useful in some way. Lossy data compression is used frequently on the Internet and especially in streaming media and telephonic applications. These methods are typically referred to as codec in this context. Most lossy data compression formats suffer

from generation loss: repeatedly compressing and decompressing the file will cause it to progressively lose quality. This is in contrast with lossless data compression [4].

Following are some definitions that are used in this research:

Compression size

Is the size of the new file in bits after compression is complete?

Compression ratio

Is a percentage that results from dividing the compression size in bits by the original file size in bits and then multiplying the result by 100%.

$$\text{Compression Ratio} = 100 * \frac{\text{Size after Compression}}{\text{Size before Compression}}$$

Compression time

Time taken for the compression and the time taken for decompression is considered separately. Compression time is the time in millisecond that we need for each symbol or character in the original file for compression, it results from dividing the time in millisecond that is needed for compressing the whole file by the number of symbols in the original file and scales as millisecond / symbol. If the compression and decompression times of an algorithm are less or up to an acceptable level then it implies that the algorithm can be accepted with respective to the given time factor.

The paper is organized as follows: Section 1 contains a brief Introduction about Compression and its types, Section 2 presents a brief explanation about different compression techniques, Section 3 has its focus on comparing the performance of compression techniques and the final section contains the Conclusion.

2 Data Compression Techniques

Various kind of text data compression algorithms have been proposed till date, mainly those algorithms is lossless algorithm. This paper examines the performance of the Run Length Encoding Algorithm (RLE), Arithmetic Encoding Algorithm, Huffman Encoding Algorithm, Adaptive Huffman Encoding Algorithm and Shannon Fano Algorithm [5]. Performance of above listed algorithms for compressing text data is evaluated and compared.

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 [6]. For example, the string 'abbaaaaabaabbbaa' 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 Huffman Coding

The Huffman coding algorithm [7] is named after its inventor, David Huffman, who developed this algorithms a student in a class on information theory at MIT in1950. 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 almost same as Shannon - Fano algorithm. Both the algorithms employ a variable bit probabilistic coding method. Both the algorithms differ slightly 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 semiadaptive 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.

2.3 Shannon Fano Coding

Shannon – Fano algorithm is developed by Claude Shannon and R. M. Fano [14, 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). From the given list of symbol, develop either 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.

2.4 Arithmetic Encoding

This encoding technique developed by Jorma Rissane. It provides extremely high coding efficiency and superior Compression to the better-known Huffman algorithm. Arithmetic coding is a method to ensure lossless data compression. It is indeed a form of variable length entropy encoding. In the case of other entropy encoding techniques, the input message is separated into its component symbols and each symbol is replaced by a code word. But arithmetic coding encodes the entire message into a single number, a fraction n where (0.0_n< 1.0) [8].

The coding algorithm is symbol wise recursive; i.e., it operates upon and encodes (decodes) one data symbol per iteration or recursion. On each recursion, the algorithm successively partitions an interval of the number line between 0 and 1, and retains one of the partitions as the new interval. Thus, the algorithm successively deals with smaller intervals, and the code string, viewed as a magnitude, lies in each of the nested intervals. The data string is recovered by using magnitude comparisons on the code string to recreate how the encoder must have successively partitioned and retained each nested subinterval.

2.5 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 [9] and by Robert G. Gallager [10] and then improved by Donald Knuth [11] 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 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.

3 Methodologies

In order to test the performance of above mentioned compression algorithms e.g. the Run Length Encoding Algorithm, Shannon Fano Algorithm, Adaptive Huffman Encoding Algorithm, Huffman Encoding Algorithm and Arithmetic Encoding, the algorithm were implemented and tested with a various set of text files. Performances of the algorithm were evaluated by computing the compression ratio, compression time.

The performances of the algorithms depend on the size of the source file and the organization of different symbols and text patterns in the source file. Therefore, research work done to include text files of different types such as notepad files, source codes, e-books in pdf files, etc, and of different file sizes are used as source files. A chart is drawn in order to verify the relationship between the file sizes after compression, the compression and decompression time.

An algorithm which gives an acceptable saving percentage with minimum time period for compression and decompression is considered as the best algorithm.

4 Results/Comparison

Five lossless compression algorithms are tested on ten different types, size and contents of text files. All the text files were of different size. The first 3 text files were in normal English language. The next 2 files are computer programs, having more repeating set of words. The last 5 file are the pdf files written in normal English language.

Followings are the results for 10 different text files.

4.1 Results

Arithmetic coding algorithm result has not been considered as results were not accurate due to overflow problem. Results of all other 4 algorithms and their comparisons are given below.

According to result of Table 1, the compression ratio of RLE algorithm is very low. For the file number 1, 3 and 7, we see that the size of compressed file is larger than original file size. Among the given 4 algorithm, we can see that the size of compressed file created by Adaptive Huffman algorithm is very less in compare to other algorithm.

Table 1 – Comparison based on compressed file size

Original File			**Compressed File Size**			
S. No.	***File Name***	***File Size***	***RLE***	***Adaptive Huffman***	***Huffman Encoding***	***Shannon Fano***
1	Paper1	22,094	22,251	13,432	13,826	14,127
2	Paper2	44,355	43,800	26,913	27,357	27,585
3	Paper3	11,252	11,267	7,215	7,584	7,652
4	Prog1	15,370	13,620	8,584	8,961	9,082
5	Prog2	78,144	68,931	44,908	45,367	46,242
6	Book1	39,494	37,951	22,863	23,275	23,412
7	Book2	118,223	118,692	73,512	74,027	75,380
8	Book3	180,395	179,415	103,716	104,193	107,324
9	Book4	242,679	242,422	147,114	147,659	150,826
10	Book5	71,575	71,194	44,104	44,586	44,806

From the Table 2, its shows that compression ratio achieved by RLE algorithm is not more than 2% of original file, that is not a reasonable compression. In the Adaptive Huffman algorithm, the compression ratio of selected files is within the range of 55% to 65%. The compression ratio does not depend on file size but it depends on structure and contents of file. In the Huffman Encoding algorithm, the compression ratio range within 58% to 67%. The compression ratios for Shannon Fano approach are in the range of 59% to 64% which is slightly equivalent to the Huffman Encoding algorithm.

So, from the table, we can derive the decision that RLE has lowest compression ratio and Adaptive Huffman has best compression ratio, although the compression ratio achieved by Adaptive Huffman is relatively same as achieved by Huffman Encoding and Shannon Fano, the difference is not more than 2%.

Table 2 – Comparison based on compression ratio

Original File			Compression Ratio			
S. No.	***File Name***	***File Size***	***RLE***	***Adaptive Huffman***	***Huffman Encoding***	***Shannon Fano***
1	Paper1	22,094	100.7106	60.7947	62.5780	63.9404
2	Paper2	44,355	98.7487	60.6763	61.6773	62.1914
3	Paper3	11,252	100.1333	64.1219	67.4013	68.0056
4	Prog1	15,370	88.6141	55.8490	58.3018	59.0891
5	Prog2	78,144	88.2102	57.4682	58.0556	59.1753
6	Book1	39,494	96.09307	57.8898	58.9330	59.2798
7	Book2	118,223	100.3967	62.1807	62.6164	63.7608
8	Book3	180,395	99.4567	57.4938	57.7582	59.4938
9	Book4	242,679	99.89409	60.6208	60.8453	62.1504
10	Book5	71,575	99.4676	61.6192	62.2927	62.6000

From the Table 3, it shows that the compression time of RLE algorithm is relatively low but for the Adaptive Huffman algorithm, the compression time is relatively high. The Compression time of Huffman Encoding algorithm and Shannon Fano algorithm is relatively low in compare to Adaptive Huffman algorithm but higher than RLE algorithm.

Table 3 – Comparison based on compression time

Original File			Compression Time (ms)			
S. No.	***File Name***	***File Size***	***RLE***	***Adaptive Huffman***	***Huffman Encoding***	***Shannon Fano***
1	Paper1	22,094	359	80141	16141	14219
2	Paper2	44,355	687	223875	54719	55078
3	Paper3	11,252	469	30922	3766	3766
4	Prog1	15,370	94	41141	5906	6078
5	Prog2	78,144	1234	406938	156844	162609
6	Book1	39,494	141	81856	13044	12638
7	Book2	118,223	344	526070	134281	153869
8	Book3	180,395	2766	611908	368720	310686
9	Book4	242,679	2953	1222523	655514	549523
10	Book5	71,575	344	231406	42046	42997

From the Table 4, it shows that the decompression time of RLE algorithm is relatively low but for the Adaptive Huffman algorithm, the decompression time is relatively high. The decompression time of Huffman Encoding algorithm and Shannon Fano algorithm is relatively low in compare to Adaptive Huffman algorithm but higher than RLE algorithm.

Table 4 – Comparison based on decompression ratio

Original File			Decompression Time (ms)			
S. No.	***File Name***	***File Size***	***RLE***	***Adaptive Huffman***	***Huffman Encoding***	***Shannon Fano***
1	Paper1	22,094	2672	734469	16574	19623
2	Paper2	44,355	2663	1473297	20606	69016
3	Paper3	11,252	2844	297625	6750	8031
4	Prog1	15,370	2500	406266	9703	9547
5	Prog2	78,144	17359	2611891	224125	229625
6	Book1	39,494	2312	1554182	12638	12022
7	Book2	118,223	1469	1271041	99086	114187
8	Book3	180,395	2250	1554182	288232	255933
9	Book4	242,679	1828	2761631	470521	441153
10	Book5	71,575	1532	633117	34293	32869

4.2 Comparison of Result

In order to compare the performance of selected algorithm, the compressed file size, compression ratio, compression and decompression time are compared. Figure 1 shows the compression file size of selected 10 files for the entire algorithms.

The sizes of compressed files are compared with original file size and result is shown in Figure 1. The figure shows that saving percentage of RLE algorithm is very less. The compressed files of all other 3 algorithms are relatively similar. The *compressed file size increased according to original file size that indicates the saving percentage of algorithm depends on the redundancy of file.*

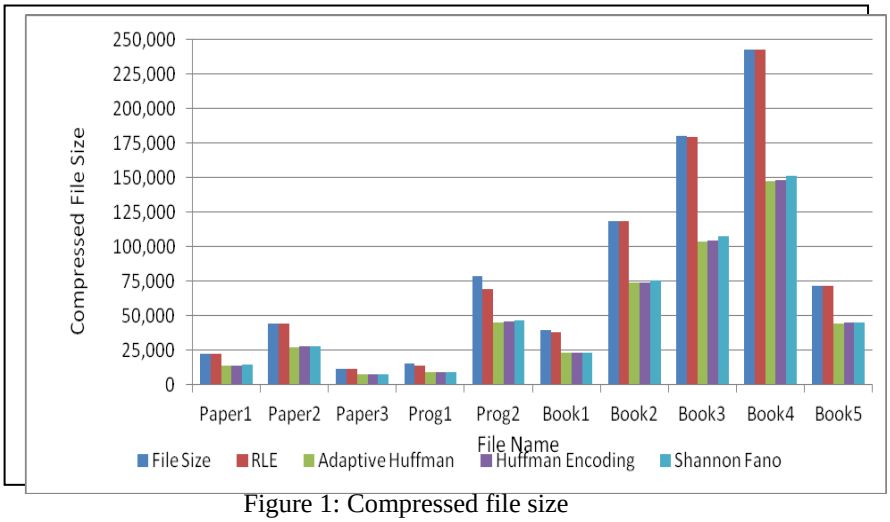

Figure 1: Compressed file size

Figure 2 shows the comparison of compression time of all 4 algorithms. Compression time increase with the increase of file size. For RLE algorithm the compression time does not depends on the size of file, it remain almost constant. Compression time for RLE is very low but for Adaptive Huffman algorithm, the compression time is very high.

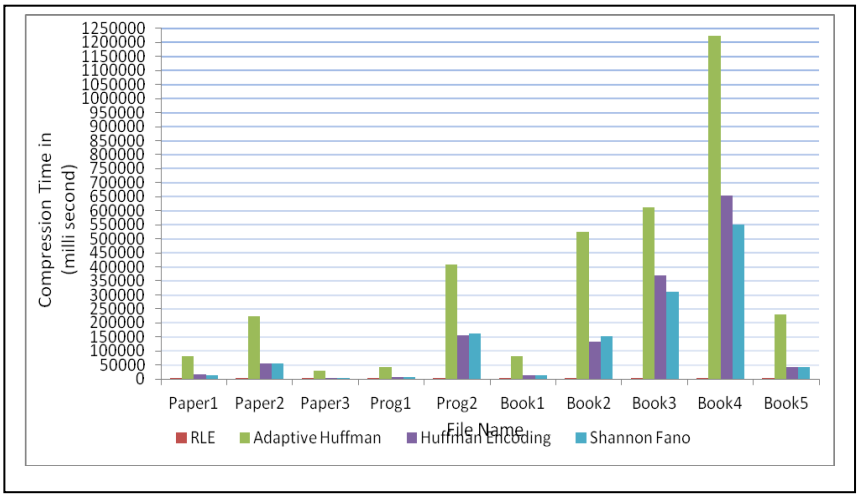

Figure 2: Compression time

Figure 3 shows the decompression time of all the algorithm. The decompression time of RLE algorithm is almost negligible and almost same for all the files of different size. the decompression time of Huffman Encoding and Shannon Fano is relatively same but for Adaptive Huffman algorithm, the decompression time is very high.

5 Conclusions

We have taken statistical compression techniques for our study to examine the performance of compression algorithm over English text data. This text data are available in the form of different kind of text file which contain different text patterns. By considering the compression time, decompression time and compression ratio of all the algorithms we have drawn the graph and table. From the above comparison and graph, it can be derived that the Huffman Encoding can be considered as the most efficient algorithm among selection ones.

We also note that; the contents of file (i.e. the number of different character or symbols and the frequency for each symbol) are effective factor on the performance of the data compression techniques. So, we suggest to make another test for the four techniques that we study but on the other sample tested files that contain different number of symbols.

Data compression stills an important topic for research these days, and has many applications and useful needed. So, we suggest continuing searching in this field and trying to combine two techniques in order to get best one.

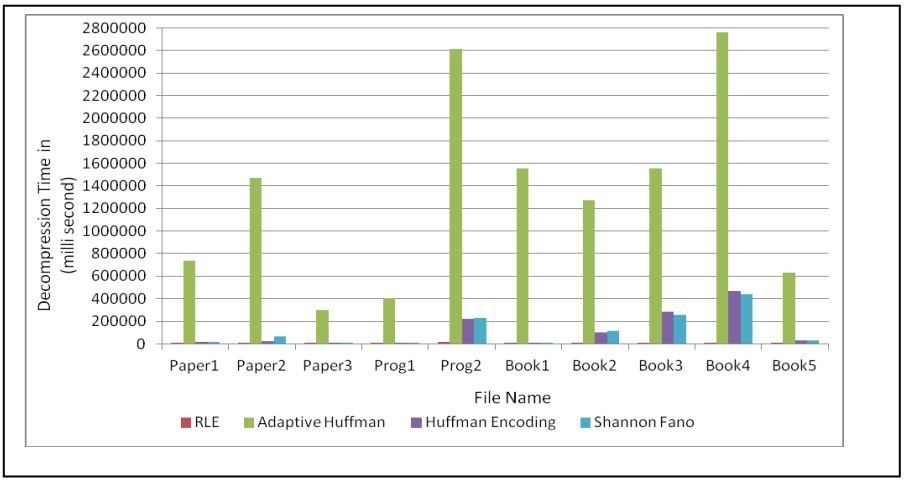

Figure 3: Decompression time

References

[1] I. M. Pu, Fundamental Data Compression, Elsevier, Britain, 2006.

[2] Data Compression: Advantages and Disadvantages: http://www.esrf.eu/computing/Forum/imgCIF/PAPER/advantages_disadvantages.html, last accessed on Feb. 2013.

[3] Lossless Compression: http://en.wikipedia.org/wiki/Lossless_compression, last access on Feb. 2013.

[4] Lossy Compression: http://en.wikipedia.org/wiki/Lossy_compression, last access on Feb. 2013.

[5] W. Kesheng, J. Otoo and S. Arie, "Optimizing bitmap indices with efficient compression", ACM Trans. Database Systems, vol. 31, pp. 1-38, 2006.

[6] E. Blelloch, Introduction to Data Compression, Computer Science Department, Carnegie Mellon University, 2002.

[7] D. A. Huffman, "A method for the construction of minimumredundancy codes", Proceedings of the Institute of RadioEngineers, vol. 40, no. 9, pp. 1098–1101, 1952.

[8] A. S. E. Campos, Basic arithmetic coding by Arturo Campos Website, Available from: http://www.arturocampos.com/ac_arithmetic.html. (Accessed 02 February 2009)

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

[10] R. G. Gallager, "Variations on a theme by Huffman", IEEE Transactions on Information Theory, vol. IT-24, no. 6, pp. 668-674, Nov. 1978.

[11] D. E. Knuth, "Dynamic Huffman coding", Journal of Algorithms, vol. 6, no. 2, pp. 163-180, June 1985.

[12] J. S. Vitter, "Design and analysis of dynamic Huffman codes", Journal of the ACM, vol. 34, no. 4, pp. 825-845, October 1987.

[13] J. S. Vitter, "Dynamic Huffman coding", ACM Transactions on Mathematical Software, vol. 15, no. 2, pp. 158-167, June 1989.

[14] R. M. Fano, "The Transmission of Information", Technical Report No. 65, Research Laboratory of Electronics, M.I.T., Cambridge, Mass.; 1949.

[15] K. Lakhtaria, "Protecting computer network with encryption technique: A Study." Ubiquitous Computing and Multimedia Applications. Springer Berlin Heidelberg, pp. 381-390, 2011.

[16] C. E. Shannon, "A mathematical theory of communication," Bell Sys. Tech. Jour., vol. 27, pp. 398-403, July 1948.

Amit Jain is working in CSE Department, Sir Padampat Singhania University, Udaipur, India. He is having 17 years of teaching experience. He has taught to post-graduate and graduate students of engineering. He is pursuing Ph.D. in Computer Science, in the area of Information Security. He has presented 3 papers in International Journal, 5 papers in International Conference and 8 papers in National Conference.

Dr. Kamaljit I Lakhtaria is working in CSE Department, Sir Padampat Singhaniya University, India. He obtained Ph. D.

in Computer Science; area of Research is "Next Generation Networking Service Prototyping & Modeling". He holds an edge in Next Generation Network, Web Services, MANET, Web 2.0, Distributed Computing. His inquisitiveness has made him present 18 Papers in International Conferences, 28 Paper in International Journals. He is author of 8 Reference Books. He is member of Life time member ISTE, IAENG. He holds the post of Editor, Associate Editor in many International Research Journal. He is reviewer in IEEE WSN, Inderscience and Elsevier Journals.