iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 2 **Issue:** X **Month of publication:** October 2014

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology(IJRASET)

Data Compression Algorithms and Comparison between Arithmetic and Huffman Encoding

Chaynika Kapoor[1], Apoorva Adlakha[2], Abhishek Jain[3]
[1,2,3]*Department of Computer Science Engineering, DCE Gurgaon*

Abstract: Data compression is a process which reduces the size of data, by removing the excessive information. Shorter data size is suitable because it simply reduces the cost. The aim of data compression is a method to reduce redundancy in stored or communicated data, thus increasing effective data density. Data compression is an essential application in the area of file storage and distributed system because in distributed system data have to send from and to all system. So to increase speed and performance efficiency data compression is used. There are number of different data compression methodologies, which are used to compress different data formats like text, video, audio, image files. There are two forms of data compression "lossy" and "lossless", in lossless data compression, the integrity of data is preserved. The compression techniques also include the most useful techniques which are the HUFFMAN ENCODING and the RUN-LENGTH ENCODING

1.INTRODUCTION

Data compression is a set of steps for packing data into a smaller space, while allowing for the original data to be seen again. Compression is a two-way process: a compression algorithm can be used to make a data package smaller, but it can also be run the other way, to decompress the package into its original form. Data compression is useful in computing to save disk space, or to reduce the bandwidth used when sending data (e.g., over the internet).

Lossless compression packs data in such a way that the compressed package can be decompressed, and the data can be pulled out exactly the same as it went in. This is very important for computer programs and archives, since even a very small change in a computer program will make it unusable.

This type of compression works by reducing how much waste space is in a piece of data. For example, if you receive a data package which contains "AAAAABBBB", you could compress that into "5A4B", which has the same meaning but takes up less space. This type of compression is called "run-length encoding", because you define how long the "run" of a character is. In the above example, there are two runs: a run of 5 A's, and another of 4 B's.

The problem with run-length encoding is that it only works on long pieces of the same value of data. If you receive a package with "ABBAABAAB" inside, that can be compressed into "1A2B2A1B2A1B"; but that's longer than the original! In this case, there's another method that can be used: checking how often a particular value comes up in the whole data package. This is often called frequency compression.

The most common kind of frequency compression is called Huffman coding, after the scientist who came up with the idea. The basic plan is to give each distinct value in a piece of data a code: values that crop up all the time get shorter codes, and values that only show up once or twice get longer codes.

For some types of data, lossy compression can go a lot further; this is most often the case with media files, like music and images. Lossy compression throws out some of the data so that there's less to store. Beyond a certain level of fine-grain detail, or past a particularly high tone, people do not notice if the information is missing. As a result, it can simply be removed from the data.

II. HUFFMAN ENCODING

The Huffman encoding algorithm is an optimal compression algorithm where only the frequency of individual letters are used to compress the data. The idea behind this algorithm is that if you have some letters that are more frequent than others, it makes sense to use fewer bits to encode those letters than to encode the less frequent letters.

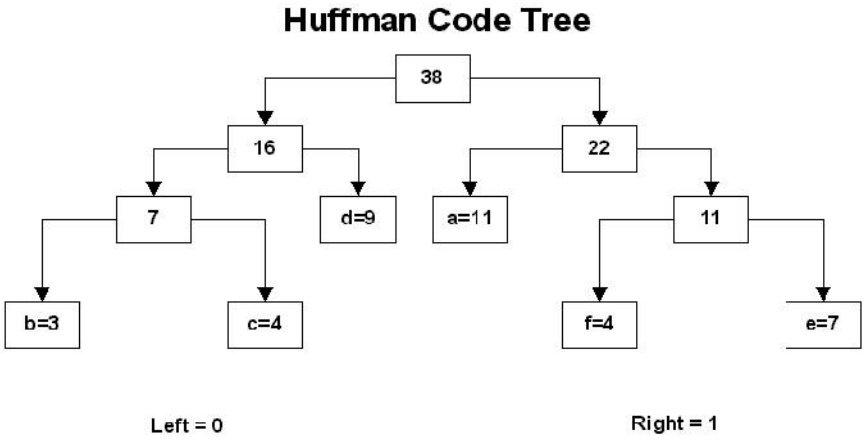


Fig.1

The Huffman algorithm is a so-called "greedy" approach to solv this problem in the sense that at each step, the algorithm chooses the best available option. It turns out that this is sufficient for finding the best encoding. The basic idea behind the algorithm is to build the tree bottom-up. First, every letter starts off as part of its own tree and the trees are ordered by the frequency of the letters in the original string. Then the two least-frequently used letters are combined into a single tree, and the frequency of that tree is set to be the combined frequency of the two trees that it links together. For instance, if we started out with two characters that showed up once, L and T, in our sample string, they would be recombined into a new tree that has a "super node" that links to both L and T, and has a frequency of 2:

```
  X, 2
  / \
 /   \
L, 1  T, 1
```

This new tree is reinserted into the list of trees in its sorted position. The process is then repeated, treating trees with more than one element the same as any other trees except that their frequencies are the sum of the frequencies of all of the letters at the leaves. (This is just the sum of the left and right children of any node because each node stores the frequency information about its own children.) The process completes when all of the trees have been combined into a single tree -- this tree will describe a Huffman compression encoding.

Essentially, a tree is built from the bottom up -- we start out with 256 trees (for an ASCII file) -- and end up with a single tree with 256 leaves along with 255 internal nodes (one for each merging of two trees, which takes place 255 times). The tree has a few interesting properties -- the frequencies of all of the internal nodes combined together will give the total number of bits needed to write the encoded file (except the header). This property comes from the fact that at each internal node, a decision must be made to go left or right, and each internal node will be reached once for each time a character beneath it shows up in the text of the document.

To go from plain text to compressed text, you would have to do a traversal of the tree and store the path to reach each leaf node as a string of bits (0 for going left, 1 for going right) and associate that bit with the particular character at the leaf. Once this is done, converting a plain text file into a compressed file is just a matter of replacing each letter with an appropriate bit string and then handling the possibility of having some extra bits that need to be written (this is discussed more fully in the implementation notes). Notice that two different data structures likely need to be used here -- a list of trees, and those binary trees themselves. It might make sense to use several data structures such as:

```
struct tree_t
{
   tree_t *left;
   tree_t *right;
   char character;
};
```

to store the tree elements (note that if left and right are NULL, then we would know that the node is a leaf and that character stores a valid char of interest) and

```
struct list_t
{
   list_t *next;
   int total_frequency;
   tree_t *tree;
}
```

to store the list of trees that still need to be merged together as a linked list.

A Huffman code is designed by merging together the two *least probable* characters, and repeating this process until there is only one character remaining. A code tree is thus generated and the Huffman code is obtained from the labeling of the code tree. An example of how this is done is shown below.

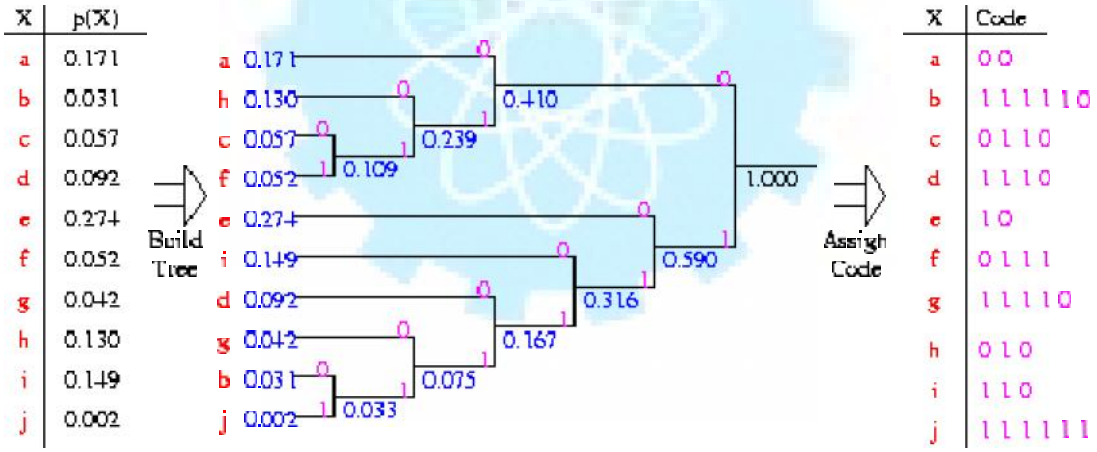


Fig 2

III. RUN LENGTH ENCODING

It is a very simple form of data compression in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs. For example: simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

RLE may also be used to refer to an early graphics file format supported by CompuServe for compressing black and white images, but was widely supplanted by their later Graphics Interchange Format. RLE also refers to a little-used image format in Windows 3.x, with the extension **rle**, which is a Run Length Encoded Bitmap, used to compress the Windows 3.x startup screen.

Typical applications of this encoding are when the source information comprises long substrings of the same character or binary digit

RLE works by reducing the physical size of a repeating string of characters. This repeating string, called a *run*, is typically encoded into two bytes. The first byte represents the number of characters in the run and is called the *run count*. In practice, an encoded run may contain 1 to 128 or 256 characters; the run count usually contains as the number of characters minus one (a value in the range of 0 to 127 or 255). The second byte is the value of the character in the run, which is in the range of 0 to 255, and is called the *run value*.

Uncompressed, a character run of 15 A characters would normally require 15 bytes to store:

AAAAAAAAAAAAAAA

The same string after RLE encoding would require only two bytes:

15A

The 15A code generated to represent the character string is called an *RLE packet*. Here, the first byte, 15, is the run count and contains the number of repetitions. The second byte, A, is the run value and contains the actual repeated value in the run.

A new packet is generated each time the run character changes, or each time the number of characters in the run exceeds the maximum count. Assume that our 15-character string now contains four different character runs:

AAAAAAbbbxxxxxt

Using run-length encoding this could be compressed into four 2-byte packets:

6A3b5x1t

Thus, after run-length encoding, the 15-byte string would require only eight bytes of data to represent the string, as opposed to the original 15 bytes. In this case, run-length encoding yielded a compression ratio of almost 2 to 1.

IV. VARIANTS ON RUN-LENGTH ENCODING

There are a number of variants of run-length encoding. Image data is normally run-length encoded in a sequential process that treats the image data as a 1D stream, rather than as a 2D map of data. In sequential processing, a bitmap is encoded starting at the upper left corner and proceeding from left to right across each scan line (the X axis) to the bottom right corner of the bitmap . But alternative RLE schemes can also be written to encode data down the length of a bitmap (the Y axis) along the columns , to encode a bitmap into 2D tiles , or even to encode pixels on a diagonal in a zig-zag fashion . Odd RLE variants such as this last one might be used in highly specialized applications but are usually quite rare.

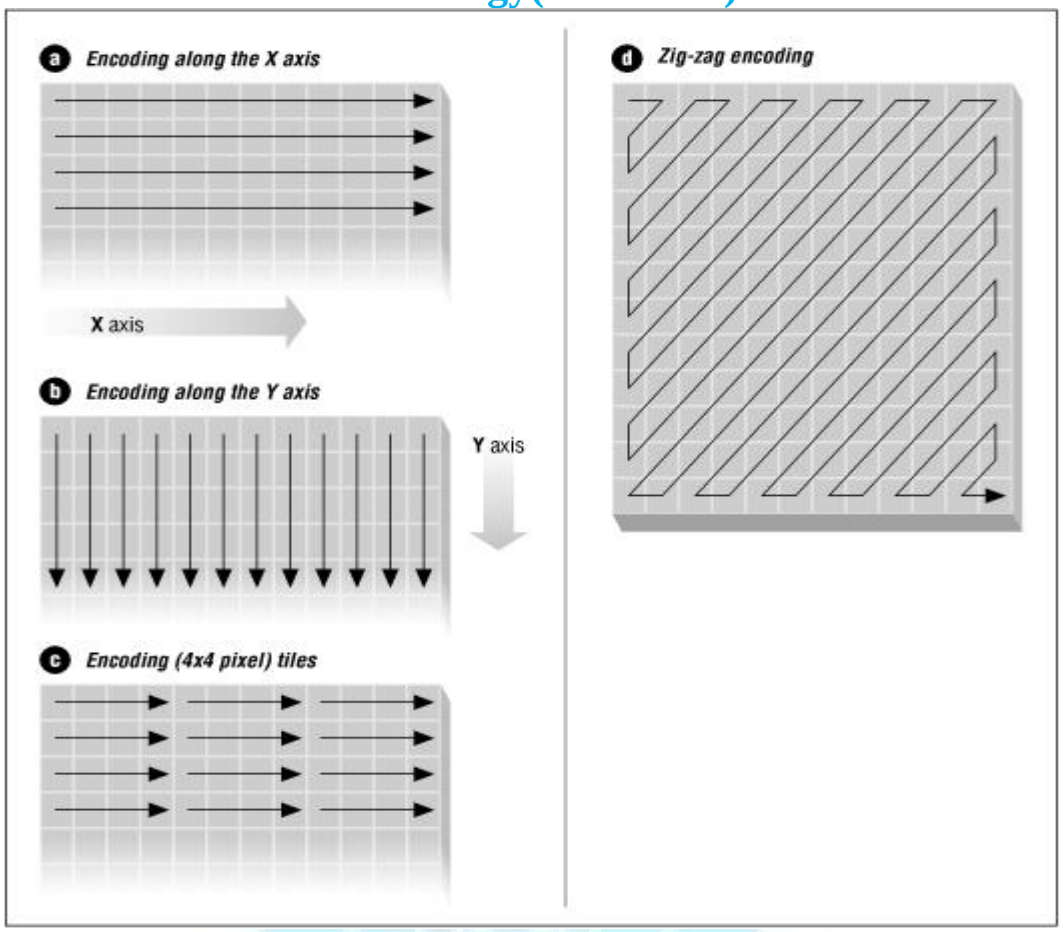


Fig. 3

RLE algorithms are normally lossless in their operation. However, discarding data during the encoding process, usually by zeroing out one or two least significant bits in each pixel, can increase compression ratios without adversely affecting the appearance of very complex images. This RLE variant works well only with real-world images that contain many subtle variations in pixel values.

Make sure that your RLE encoder always stops at the end of each scan line of bitmap data that is being encoded. There are several benefits to doing so. Encoding only a simple scan line at a time means that only a minimal buffer size is required. Encoding only a simple line at a time also prevents a problem known as *cross-coding*.

Bit-, Byte-, and Pixel-Level RLE Schemes

The basic flow of all RLE algorithms is the same, as illustrated in fig.

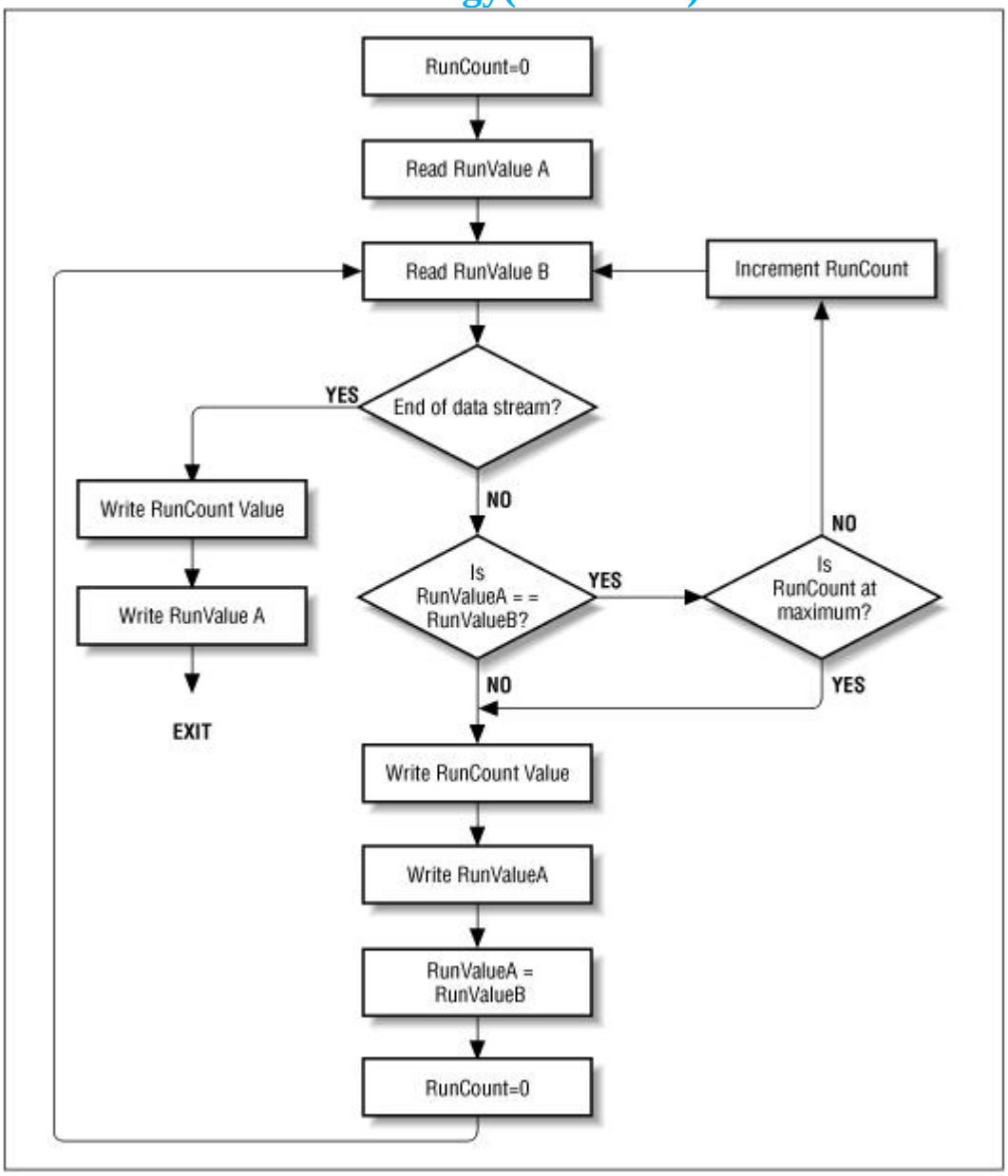


Fig. 4

The parts of run-length encoding algorithms that differ are the decisions that are made based on the type of data being decoded (such as the length of data runs). RLE schemes used to encode bitmap graphics are usually divided into classes by the type of atomic (that is, most fundamental) elements that they encode. The three classes used by most graphics file formats are bit-, byte-, and pixel-level RLE.

Bit-level RLE schemes encode runs of multiple bits in a scan line and ignore byte and word boundaries. Only monochrome (black and white), 1-bit images contain a sufficient number of bit runs to make this class of RLE encoding efficient. A typical bit-level RLE scheme encodes runs of one to 128 bits in length in a single-byte packet. The seven least significant bits contain the run count minus one, and the most significant bit contains the value of the bit run, either 0 or 1 A run longer than 128 pixels is split across several RLE-encoded packets.

V. ARITHMETIC ENCODING

Arithmetic encoding is the most powerful compression techniques. This converts the entire input data into a single floating point number. A floating point number is similar to a number with a decimal point, like 4.5 instead of **4**1/2. However, in arithmetic coding we are not dealing with decimal number so we call it a floating point instead of decimal point [4]. Let‟s take an example we have string:

BE_A_BEE

E	B	_	A
3	2	2	1

Step 1: in the first step we do is look at the frequency count for the different letters:

Step 2: In second step we encode the string by dividing up the interval [0, 1] and allocate each letter an interval whose size depends on how often it count in the string. Our string start with a „B‟, so we take the „B‟ interval and divide it up again in the same way:

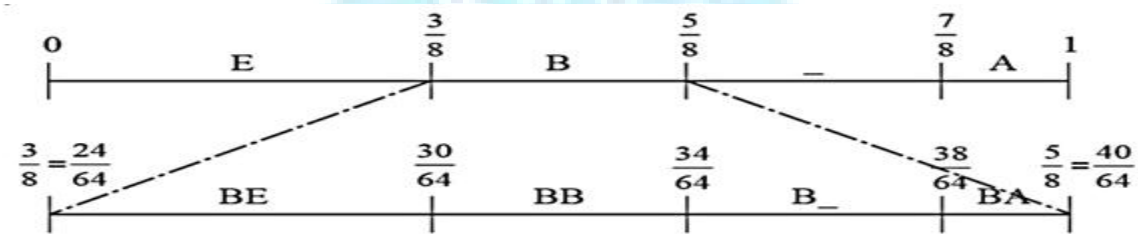
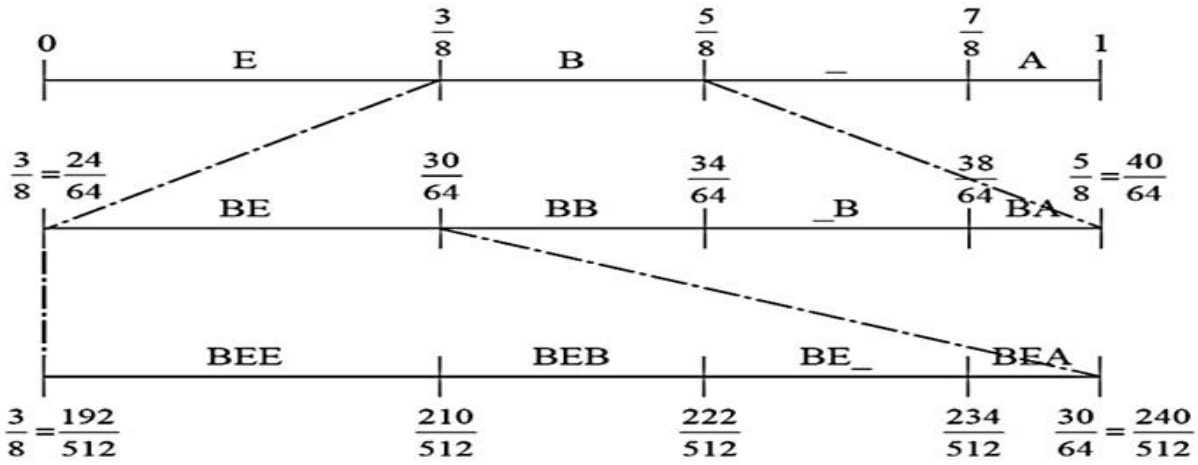
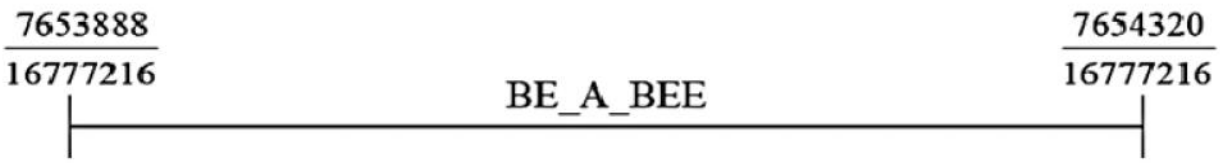


Fig 5

The boundary between „BE‟ and „BB‟ is 3/8 of the way along the interval, which is itself 2/3 long and starts at 3/8. So boundary is 3/8 + (2/8) * (3/8) = 30/64. Similarly the boundary between „BB‟ and „B_‟ is 3/8+ (2/8) * (5/8) = 34/64, and so on. **Step 3:** In third step we see next letter is now „E‟, so now we subdivide the „E‟ interval in the same way. We carry on through the message….And, continuing in this way, we eventually obtain



and continuing in this way, we obtain:



So we represent the message as any number in the interval [7653888/16777216, 7654320/16777216]

However, we cannot send numbers like 7654320/16777216 easily using computer. In decimal notation, the rightmost digit to the left of the decimal point indicates the number of units; the one to its left gives the number of tens: the next one along gives the number of hundred, and so on. So 7653888 = $(7*10_6) + (6*10_5) + (5*10_4) + (3*10_3) + (8*10_2) + (8*10) + 8$ Binary numbers are almost exactly the same, only we deal with powers of 2 instead of power of 10. The rightmost digit of binary number is unit (as before) the one to its left gives the number of 2s, the next one the number of 4s, and soon. So 110100111 = $(1*2_8) + (1*2_7) + (0*2_6) + (1*2_5) + (0*2_4) + (0*2_3) + (1*2_2) + (1*2_1) + 1 = 256 + 128 + 32 + 4 + 2 + 1 = 423$ in denary (i.e. base 10).

VI. MEASURING COMPRESSION PERFORMANCES

Performance measure is use to find which technique is good according to some criteria. Depending on the nature of application there are various criteria to measure the performance of compression algorithm. When measuring the performance the main thing to be considered is space efficiency [5]. and the time efficiency is another factor. Since the compression behavior depends on the redundancy of symbols in the source file, it is difficult to measure performance of compression algorithm in general. The performance of data compression depends on the type of data and structure of input source. The compression behavior depends on the category of the compression algorithm: lossy or lossless. Following are some measurements use to calculate the performances of lossless algorithms.

A. COMPRESSION RATIO: compression ratio is the ratio between size of compressed file and the size of source file.

$$Compression\ ratio = \frac{Size\ after\ compression}{Size\ before\ compression}$$

B. COMPRESSION FACTOR: compression factor is the inverse of compression ratio. That is the ratio between the size of source file and the size of the compressed file.

$$Compression\ factor = \frac{Size\ before\ compression}{Size\ after\ compression}$$

Saving percentage calculates the shrinkage of the source file as a percentage

$$saving\ percentage = \frac{size\ before\ compression - size\ after\ compression}{size\ before\ compression}\%$$

C. COMPRESSED PATTERN MATCHING: compressed pattern matching is the process of searching of pattern in compressed data with little or no decompression shown in following table.

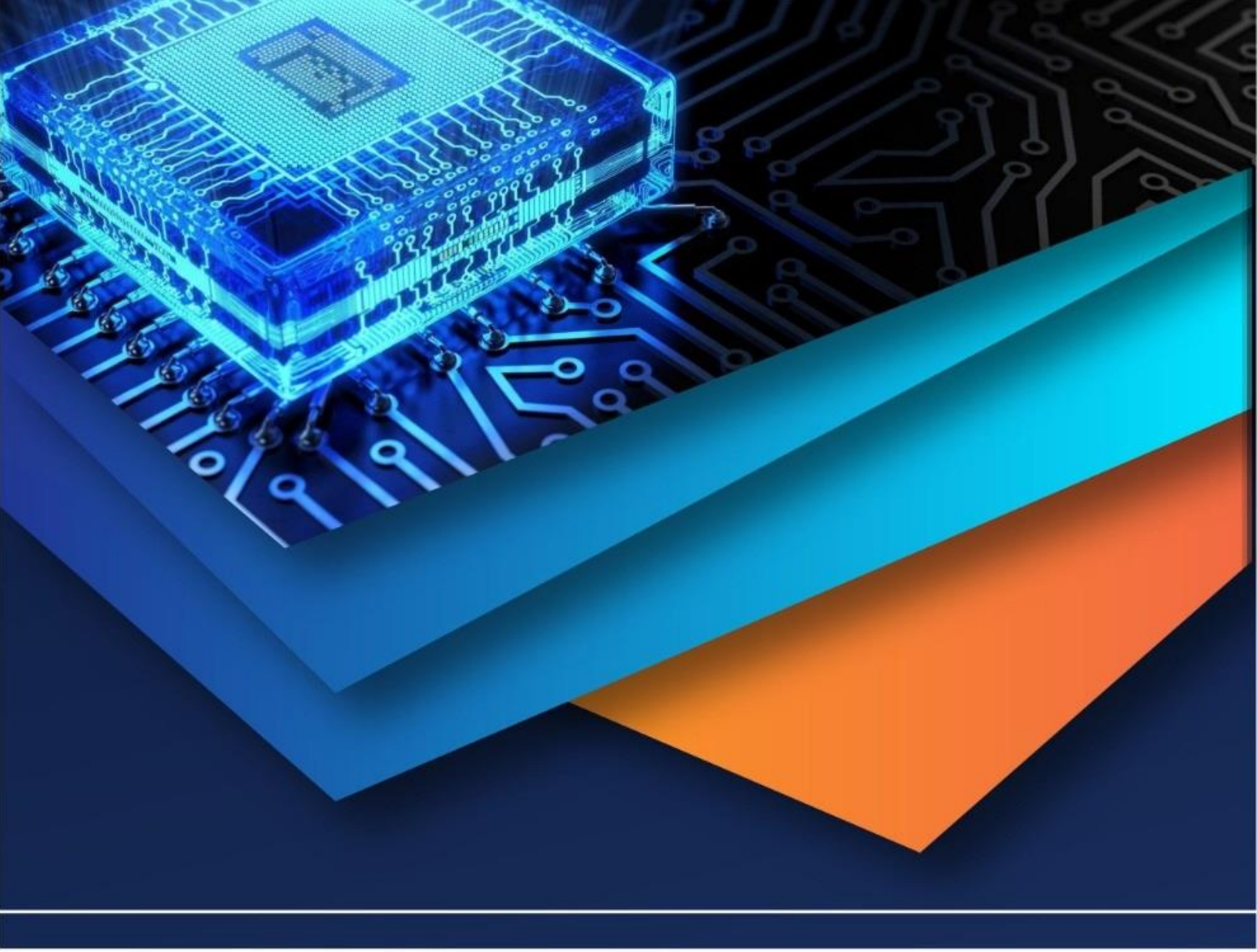
COMPRESSION METHOD	ARITHMATIC	HUFFMAN
Compression Ratio	Very Good	Poor
Compression Speed	Slow	Fast
Decompression Speed	Slow	Fast
Memory Space	**Very low**	Low
Compressed Pattern Matching	NO	Yes
Permits Random Access	No	Yes

VII. CONCLUSION

In this paper we have find out that arithmetic encoding methodology is very powerful over Huffman encoding methodology. In comparison we came to know that compression ratio of arithmetic encoding is better. And furthermore arithmetic encoding reduces channel bandwidth and transmission time

REFERENCES

[1] Introduction to Data Compression, Khalid Sayood, Ed Fox (Editor), March 2000.
[2] Burrows M., and Wheeler, D. J. 1994. A Block-Sorting Lossless Data Compression Algorithm. SRC Research Report 124, Digital Systems Research Center.
[3] Ken Huffman. Profile: David A. Huffman, Scientific American, September 1991, pp. 54–58.
[4] Blelloch, E., 2002. Introduction to Data Compression, Computer Science Department, Carnegie Mellon University.
[5] Cormak, V. and S. Horspool, 1987. Data compression using dynamic Markov modeling, Comput. J., 30: 541–550.
[6] Cleary, J., Witten, I., "Data Compression Using Adaptive Coding and Partial String Matching", IEEE Transactions on Communications, Vol. COM-32, No. 4, April 1984, pp 396-402.
[7] Mahoney, M., "Adaptive Weighting of Context Models for Lossless Data Compression", Unknown, 2002.
[8] Bloom, C., "LZP: a new data compression algorithm", Data Compression Conference, 1996. DCC '96. Proceedings, p. 425 10.1109/DCC.1996.488353.
[9] Capocelli, M., R. Giancarlo and J. Taneja, 1986. Bounds on the redundancy of Huffman codes, IEEE Trans. Inf. Theory, 32: 854–857.
[10] Kaufman, K. and T. Shmuel, 2005. Semi-lossless text compression, Intl. J. Foundations of Computer Sci., 16: 1167-1178.
[11] Pu, I.M., 2006, Fundamental Data Compression, Elsevier, Britain.
[12] D. S. Taubman and M. W. Marcellin, JPEG200



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY