ISSN:2231-0711

Comparitive Analysis of Data Encoding Techniques

Ms. Niharika Poddar
3rd Year, Dept. of Computer Science & Engineering
RNS Institute of Technology
Bengaluru, India
npoddar338@gmail.com

Ms.Varnika Bagaria
3rd Year, Dept. of Computer Science & Engineering
RNS Institute of Technology
Bengaluru, India
varnikabagaria0110@gmail.com

Mrs. Sampada K.S.
Dept. of Computer Science & Engineering
RNS Institute of Technology
Bengaluru, India
k.s.sampada@gmail.com

Abstract— Data Encoding for compression is an area where different methodologies have been defined for the purpose. Hence choosing the best encoding is really important. In addition to different compression technologies and methodologies, selection of a good data compression tool is most important. There is a range of different symbol-based and dictionary-based data compression techniques available. In order to choose the right algorithm, one must determine whether the purpose is to achieve better compression ratio or higher security.

Keywords: Text data compression, Huffman Encoding, Arithmetic Encoding, LZW Encoding

I. INTRODUCTION

Data Compression is a method of encoding rules that allows substantial reduction in the total number of bits to store or transmit a file[1] . Data Compression is possible because most of the real-world data is very redundant.

There are two basic classes of data compression are applied in different areas:

- *Lossy data compression* which is widely used to compress image data files for communication or archives purposes.
- *Lossless data compression* that is commonly used to transmit or archive text or binary files required to keep their information intact at any time. Lossless compression techniques are further divided into symbol-based and dictionary-based techniques.

In this paper, we'll be doing a comparative study between three lossless data compression techniques, that is, Huffman Encoding, Arithmetic Encoding and LZW Encoding, where Huffman and Arithmetic Encoding techniques are symbol-based technique and LZW Encoding technique is a dictionary-based technique.

II. ENCODING ALGORITHMS

A. HUFFMAN ENCODING

Huffman Coding is a symbol-based data compression technique. It generates variable length codes that are integral number of bits [2]. This encoding technique is entropy based i.e., symbols with higher probability or occurrence frequency get shorter codes. The Huffman Code for each symbol has a unique prefix attribute due to which they can be decoded correctly in spite of being variable length.

There are 2 major steps to compress data using Huffman Encoding

1. Build a Prefix Tree (which is essentially a Binary Search Tree) from the input string symbols in a Bottom-Up Manner:
   i. Create a Leaf Node to represent each unique symbol in the input string.
   ii. Each Leaf Node has a weight, which is the probability or frequency of the symbol's occurrence in the input string.
   iii. Two free Nodes with the lowest weight are identified. A parent node is created for these two nodes. The weight of the parent node is assigned to be the sum of its two child nodes.
   iv. The parent node is added to the list of free nodes and the two child nodes are removed from the list
   v. Steps 'c' and 'd' are repeated until only one free node is left. This node is designated as the root of the tree.
2. Traversing the Prefix Tree to assign Prefix Code to each symbol respectively.
   i. Start traversing the Prefix Tree from the Root Node.
   ii. While traversing to a child on the left, write 0.
   iii. While traversing to a child on the right, write 1.

Assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute c.freq giving its frequency. Let Q be the list of free nodes.

Huffman Encoding Algorithm[3]:

```
HUFFMAN(C )
    n = |C|
    Q = C
    for i = 1 to n - 1
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq+y.freq
        INSERT (Q,z)
  return EXTRACT-MIN(Q)
```

Let $d_T(c)$ be the length of the codeword for each character c. The number of bits required to encode the input string is:

$$B(T) = \sum c.freq * dT(c);\ c \in C \quad \text{---(1)}$$

Consider the Table 1 shown below:

Table 1 List of Characters and their Appearance Frequency

Character	A	B	C	D	E
Appearance	15	7	6	6	5

Each of these five nodes A, B, C, D and E are the leaf nodes. Initially, they form the list of free nodes.

The first iteration through the tree finds D (or C) and E with lowest weights 6 and 5. These two are joined to a new parent node with weight = 6+5 = 11. Nodes D and E are then removed from the list of free nodes.

On the next iteration, the two nodes with lowest weights are the nodes B and C. These are attached to another new parent node altogether. The new parent node is assigned weight 13. B and C are removed from the list of free nodes.

In the succeeding iteration, the B/C and D/E parent nodes (with weights 13 and 11 respectively) are identified to be the lowest and tied to a new parent node with weight = *13+11 = 24.*

Finally, in the last iteration, only free nodes left are A (weight = 15) and the parent node for B/C and D/E (weight = 24). These two are attached to a new parent node with weight *15+24 = 39* and is the only free node remaining in the free node list, signifying that the tree is complete.

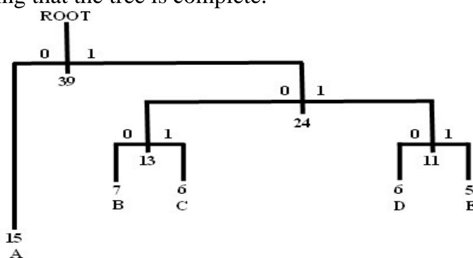

Figure 1: The Huffman Tree

To determine the unique prefix code for each symbol, we traverse the Huffman Tree as mentioned under I.2. This technique assigns the following prefix codes to the characters:

Table 2 The Huffman Code Table

A	0
B	100
C	101
D	110
E	111

Using equation (1), the total number of bits required to encode the input string:

*= 15 * 1 + 7 * 3 + 6 * 3 + 6 * 3 + 5 * 3 = 15 + 21 + 18 + 18 + 15*

= 87 bits.

In order to decode the encoded string, it is required to send the List of Characters and their Appearance Frequency along with the Huffman Codes.

B. *ARITHMETIC ENCODING ALGORITHM*

Arithmetic coding is a data compression technique that encodes data string by creating a code string which represents a fractional value on the number line between 0 and 1[4]. The coding algorithm is symbol-wise recursive; that is, it operates upon and encodes one data symbol per iteration. On each iteration, 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 structure used here is HashMap.

Arithmetic coding is the process of subdividing a unit interval such that:

- Each codeword is the sum of probabilities of preceding symbols.
- The width of each subinterval to the right gives the probability of that symbol.

Arithmetic Coding Algorithm[5]:

```
encode_symbol(symbol, cum_prob):
        range = high – low
        high = low + range * cum_prob[symbol -1]
        low = low + range * cum_prob[symbol]
```

The function encode_symbol should be called repeatedly for each symbol in the file until you encounter a terminator.

The input to the algorithm is file that is to be compressed and a probability table having the probabilities of each character in the file. We have calculated the probabilities by finding the relative frequency of each character in the file. All the probabilities will be in range [0,1). The characters are also arranged in order of their ASCII values.

The output of the program will be the encoded string which will consist of floating-point values between 0 and 1.

Consider the Table 3 as shown:

Table 3 List of characters and their respective probability

Character	a	b	c	d
Probability	0.5	0.25	0.125	0.125

Consider the Figure shown below:

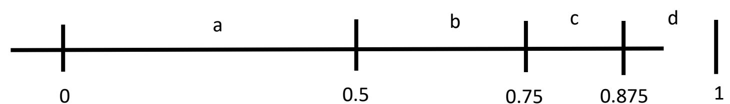

Figure 2: Codewords of Table 1 in unit interval

Consider an example string of *"abcd"*, let the probability of each character be as shown in Table 3 .

The first symbol in the string is 'a' so the corresponding interval to be considered is [0, 0.5). This is interval is to be further divided into a number line similar to that of Figure 2 with the end points being 0 and 0.5.

Let *'r'* be the range of the interval. The range of new interval is 0.5. The new range of each symbol is calculated by:

*Range of symbol (R) = l : l + r * (probability of symbol)*
where, *l* is lower limit -----(2)

In the above example, for *'a'* the new range is given by:

$R_a = 0 : 0 + 0.5 * 0.5 = [0 : 0.25)$

for 'b' the new range is given by:

$R_b = 0.25 + 0.5 * 0.25 = [0.25 : 0.375)$

Similarly, for *'c'* and *'d'* we get $R_c = [0.375, 0.4375)$ and $R_d = [0.4375, 1)$

Now the next symbol is *'b'* so the corresponding interval to be considered is [0.25, 0.375).

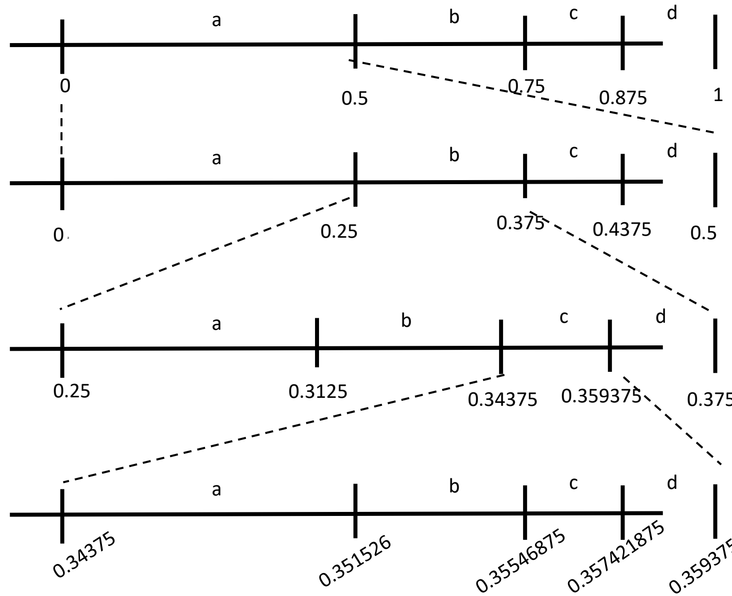

Figure 3 Successive division of interval

Using equation (2), new values of ranges of each symbol is calculated repeatedly until the last character is reached as shown in Figure 3. In the string, the last character is *'d'*, so from the last partition the interval of *'d'* is selected; that is [0.357421875, 0.359375). So, the code word of the string will be any number that lies between the range: $0.357421875 \leq codeword < 0.359375$

The iteration is terminated when the method encounters an end of file or a predefined termination character.

C. LZW ENCODING ALGORITHM

The Lempel-Ziv-Welch or LZW Encoding Technique is a dictionary-based compression scheme whereby groups of symbols that appear in an adaptive dictionary are looked for in the input string [6]. If a pattern is found in the dictionary, the index of the pattern is the output instead of the code for the symbols. The longer the matching pattern, better the compression ratio.

- An adaptive dictionary always saves all ASCII characters from 0-255 with respective index. Initially the dictionary contains only these 256 entries.
- LZW tends to identify repeated sequences in the input data and on encountering a new pattern, it adds it to the dictionary.

LZW Algorithm[7]:

```
LZW (input string S)
Initialize table with single character strings
P = first input character
WHILE not end of input stream
    C = next input character
    IF P + C is in the string table
        P = P + C
    ELSE
        output the code for P
        add P + C to the string table
        P = C
END WHILE
output code for P
```

Evidently, the most essential pre-requisite for the LZW Algorithm is a well – modelled dictionary. This can be achieved by using the Hash Table data structure.

An important point to note is that the same copy of the above stated dictionary must be available with both the encoder and decoder for this technique to work correctly.

Therefore, in order to achieve a complete LZW encoder:

- Initialize the dictionary with codes 0-255
- Insert newly discovered patterns in the string, generating a new index or code for them.

With the above two points in mind, consider an example string[8] : "ABBABB". After reading the input string, the dictionary has the following inclusions:

Table 4 Dictionary additions after processing "ABBABB"

String	Code
AB	257
BB	258
BA	259
ABB	260

To understand the generation of encoded output for "ABBABB", consider the following table:

Table 5 Generation of encoded output for "ABBABB"

Input Symbol	Working	Output Code
A	Current String = "A", already exists in dictionary	
B	Current String = "AB", added to dictionary. Last substring of the current string matched in dictionary = "A", therefore index of "A" is the corresponding output code. Current String = "B".	65
B	Current String = "BB", added to dictionary. Last substring of the current string matched in dictionary = "B", therefore index of "B" is the corresponding output code. Current String = "B"	66
A	Current String = "BA", added to dictionary. Last substring of the current string matched in dictionary = "B", therefore index of "B" is the corresponding output code. Current String = "A"	66
B	Current String = "AB", already exists	
B	Current String = "ABB" added to dictionary. Last substring of the current string matched in the dictionary = "AB", therefore index of "AB" is the corresponding output code.	257

Therefore, the encoded output for the input string "ABBABB" is 65 66 66 257.

III. EXPERIMENTAL RESULTS

The tables 7-9 shows the time taken for the various algorithms and Fig 4-6 shows the graph of varying file size and the corresponding time taken by various algorithms.

a. For Huffman encoding algorithm:

Table 7 Tabulated values of time taken to encode text of various file sizes by Huffman Encoding

Size of file (in bytes)	*Time taken by Huffman Coding (in μs)*
24603	3660.5
100000	19381
426754	21443.6
1000000	46316.4
1029744	3735.7
2473400	3781
4638690	87685.6

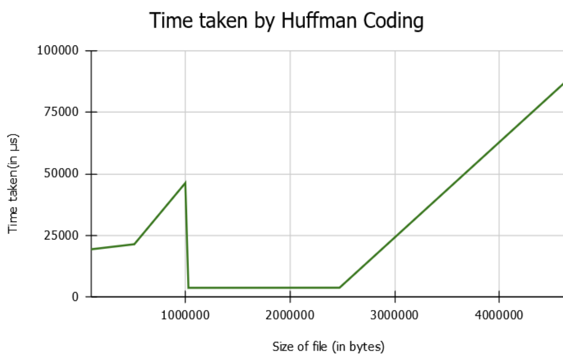

Figure 4 Graph of Time taken vs Size of file by Huffman Encoding

b. For Arithmetic encoding algorithm:

Table 8 Tabulated values of time taken to encode text of various file sizes by Arithmetic Encoding

Size of file (in bytes)	*Time taken by Arithmetic Coding(in μs)*
24603	73720
100000	138249
426754	658400
1000000	608258
1029744	71496
2473400	2051045
4638690	1496322

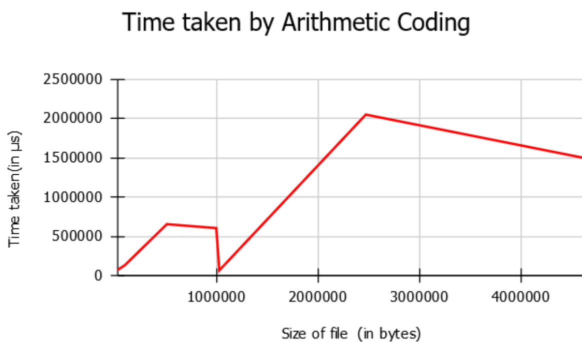

Figure 5 Graph of Time taken vs Size of file by Arithmetic Encoding

c. For LZW encoding algorithm:

Table 9 Tabulated values of time taken to encode text of various file sizes by LZW Encoding

Size of file (in bytes)	*Time taken by LZW Coding (in μs)*
24603	550.4
100000	483204.5
426754	63526
1000000	2120581.3
1029744	3240.1
2473400	1682
4638690	8006460

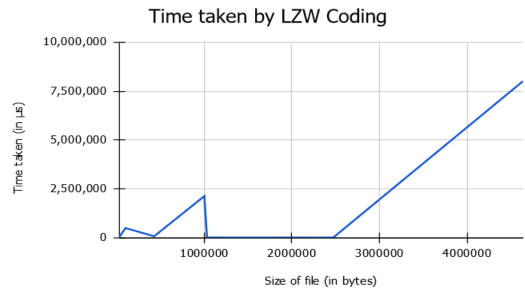

Figure 6 Graph of Time taken vs Size of file by LZW Encoding

On careful examination of graphs and tables, it is observed that, although the theoretical Time Complexity for Huffman Encoding is more than that of Arithmetic and LZW Encoding, the former technique outperforms the latter.

IV RESULT DISCUSSION

A. *Space Complexity*

Following are the conclusions drawn about the Space Complexity of the Compression Algorithms discussed:

- For Huffman Encoding, the Space Complexity is 2*O(|Σ|) = O(|Σ|) (input string is passed twice: once to calculate frequency of occurrence, and again to construct the Prefix Tree), where |Σ| = no. of unique characters in the input string.
- For Arithmetic Encoding, the Space Complexity depends on number of different input symbols where maximum is O(n), where n is the length of message or file.
- For LZW Encoding, the Space Complexity is O(n) as the initial dictionary size is fixed and independent of the input length. Each byte is read only once and the complexity of operating each character is constant.

B. *Time Complexity*

Fig 7. Shows the graph of time taken by various algorithms for varying size of file. Following are the conclusions drawn about the Time Complexity of the Compression Algorithms discussed:

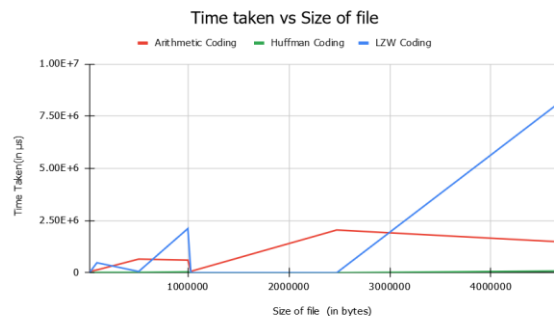

Figure 7 Graph of Time taken vs Size of file by all algorithm

- For Huffman Encoding, the Time Complexity is O(nlogn), where n is the no. of characters in the input string. Using Heap Sort, each iteration requires log(n) time to determine the lowest weight node.
- For Arithmetic Encoding, the Time Complexity depends on the number of different symbols and length of symbols, that is, $n + n^*|\Sigma|$, where Σ is unique symbols set. In the comparison we are limiting the n(|Σ|) to ASCII range of [0,255]. So, the time complexity is $O(n^*|\Sigma|)=O(n)$ where |Σ| is a constant set of values and $n(|\Sigma|)$ is ≤ 256.
- For LZW Encoding, the Time Complexity of operation of each character is a constant

V Conclusion

On careful examination of the results obtained by encoding our Example Data using all the three Encoding Techniques, we concluded that choosing the right encoding technique for data compression depends on the data to be compressed. For diverse data, symbol-based encoding techniques are more efficient. For massive data with higher pattern repetitions, dictionary-based encoding techniques prove to be a better choice.

Huffman encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length. It generates shorter binary codes for encoding symbols that appear more frequently in the input string. Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output. Hence, this feature is beneficial from security perspective.

Arithmetic coding typically has a better compression ratio than Huffman coding, as it produces a single symbol rather than several separate codewords. Arithmetic coding differs from other forms of entropy encoding as rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, a fraction 'n' where $0 \leq n < 1$. There are a few disadvantages of arithmetic coding. One is that the whole codeword must be received to start decoding the symbols, and if there exists a corrupt symbol then entire message is corrupted. Another is that there is a limit to the number of symbols to encode within a codeword.

LZW technique is simple, and there is no need to analyze the input text. But the implementation becomes difficult in terms of managing the dictionary. The overhead of storing a variable length string and added to the stated issue. Files with 0 repetitions although encoded, are not compressed at all.

VI References

[1] Amandeep Singh Sidhu, Er. Meenakshi Garg "Research Paper on Text Data Compression Algorithm using Hybrid Approach" International Journal of Computer Science and Mobile Computing

[2] Mark Nelson, Jean-loup Gailly (1995) The Data Compression Book (2nd Edition). MIS:PressSubs. of Henry Holt, & Co. 115 W. 18th St. New York, United States

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) Introduction To Algorithms (3rd Edition). The MIT Press Cambridge, Massachusetts, London, England

[4] Glen G. Langdon, Jr "An Introduction to Arithmetic Coding"

[5] Ian H. Willen, Radford M. Neal, And John G. Cleary "Arithmetic Coding For Data Compression"

[6] Mark Nelson, Jean-loup Gailly (1995) The Data Compression Book (2nd Edition). MIS:PressSubs. of Henry Holt, & Co. 115 W. 18th St. New York, United States

[7] Amartya Ranjan Saikia "LZW(Lempel–Ziv–Welch) Compression technique"

[8] Mark Nelson "LZW Data Compression Revisited"

VII Acknowledgement

We are grateful to Mrs. Sampada K. S., Assistant Professor, CSE Department, RNSIT for mentoring us to present this paper successfully.