Comparison Between (RLE And Huffman) Algorithmsfor Lossless Data Compression

Dr. AMIN MUBARK ALAMIN IBRAHIM
Department of mathematics
Faculty of Education
Shagra University
Afif, Saudi Arabia

Dr. MUSTAFA ELGILI MUSTAFA
Computer Science Department
Community College
Shaqra University
Shaqra, Saudi Arabia

Home affilation
Assistant Professor, Faculty of Computer Science and Information Technology–Neelian University, Khartoum, Sudan

Abstract: **Multimedia field is distinguished from other areas of the need for massive storage volumes. This caused a lot of problems, particularly the speed of reading files when (transmission and reception) and increase the cost (up capacities petition) was to be the presence of ways we can get rid of these problems resulting from the increase Size was one of the successful solutions innovation algorithms to compress files. This paper aims to compare between (RLE and Huffman) algorithms which are also non-compression algorithms devoid texts, according to the standard file size. Propagated the comparison between the original file size and file size after compression using (RLE & HUFFMAN) algorithms for more than (30) text file. We used c++ program to compress the files and Microsoft excel program in the description analysis so as to calculate the compression ratio and others things. The study pointed to the effectiveness of the algorithm (HUFFMAN) in the process of reducing the size of the files.**

Keywords: **RlE,Huffman**

I. INTRODUCTION

In computer science and information theory, data compression, source coding,[1] or bit-rate reduction involves encoding information using fewer bits than the original representation.[2] Compression can be either lossy or lossless. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Lossy compression reduces bits by identifying unnecessary information and removing it.[3] The process of reducing the size of a data file is referred to as data compression. In the context of data transmission, it is called source coding (encoding done at the source of the data before it is stored or transmitted) in opposition to channel coding.[4]Compression is useful because it helps reduce resource usage, such as data storage space or transmission capacity. Because compressed data must be decompressed to use, this extra processing imposes computational or other costs through decompression; this situation is far from being a free lunch. Data compression is subject to a space–time complexity trade-off. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed, and the option to decompress the video in full before watching it may be inconvenient or require additional storage. The design of data compression schemes involves trade-offs among various factors, including the degree of compression, the amount of distortion introduced (e.g., when using lossy data compression), and the computational resources required to compress and uncompressed the data.[5] However, the most important reason for compressing data is that more and more we share data. The Web and its underlying networks have limitations on bandwidth that define the maximum number of bits or bytes that can be transmitted from one place to another in a fixed amount of time. This research aims to compare between (Rle) and (Huffman) algorithm which are lossless compression algorithms , according to the standard file size.

II. RLE ALGORITHM(RUN LENGTH ENCODING)

Run-length encoding is a data compression algorithm that is supported by most bitmap file formats, such as TIFF, BMP, and PCX. RLE is suited for compressing any type of data regardless of its information content, but the content of the data will affect the compression ratio achieved by RLE. Although most RLE algorithms cannot achieve the high compression ratios of the more advanced compression methods, RLE is both easy to implement and quick to execute, making it a good alternative to either using a complex compression algorithm or leaving your image data uncompressed.

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.

RLE schemes are simple and fast, but their compression efficiency depends on the type of image data being encoded. A black-and-white image that is mostly white, such as the page of a book, will encode very well, due to the large amount of contiguous data that is all the same color. An image with many colors that is very busy in appearance, however, such as a photograph, will not encode very well. This is because the complexity of the image is expressed as a large number of different colors. And because of this complexity there will be relatively few runs of the same color.[7]

III. HUFFMAN ALGORITHM

In computer science and information theory, a Huffman code is an optimal prefix code found using the algorithm developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".[10] The process of finding and/or using such a code is called Huffman coding and is a common technique in entropy encoding, including in lossless data compression. The algorithm's output can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). Huffman's algorithm derives this table based on the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code inlinear time to the number of input weights if these weights are sorted.[11] However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

A Huffman encoding can be computed by first creating a tree of nodes:[8]

1. Create a leaf node for each symbol and add it to the priority queue(similar to a queue)
2. While there is more than one node in the queue:
    a. Remove the node of highest priority (lowest probability) twice to get two nodes.
    b. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
    c. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Traverse the constructed binary tree from root to leaves assigning and accumulating a '0' for one branch and a '1' for the other at each node. The accumulated zeros and ones at each leaf constitute a Huffman encoding for those symbols and weights.

Example1:[9]

Lets say you have a set of numbers and their frequency of use and want to create a Huffman encoding for them:

FREQUENCY	VALUE
5	1
7	2
10	3
15	4
20	5
45	6

Creating a huffman tree is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies:

```
      12:*
      /  \
   5:1   7:2
```

The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

10:3

12:*

15:4

20:5

45:6

You then repeat the loop, combining the two lowest elements. This results in:

```
                22:*
               /    \
           10:3      12:*
               /    \
            5:1      7:2
```

and the list is now:

15:4

20:5

22:*

45:6

You repeat until there is only one element left in the list.

```
                35:*
               /    \
           15:4      20:5
```

```
22:*

35:*

45:6
```

```
                57:*
          _____/     \_____
         /                 \
      22:*                 35:*
     /    \               /    \
 10:3      12:*       15:4      20:5
                      /    \
                   5:1      7:2
```

```
45:6

57:*
```

```
                       102: *
                  _____/     \__
                 /              \
              57:*              45:6
          ____/   \____
         /             \
      22:*             35:*
     /    \           /    \
 10:3      12:*   15:4      20:5
  /    \
5:1     7:2
```

Now the list is just one element containing 102:*, you are done.

This element becomes the root of your binary Huffman tree. To generate a Huffman code you traverse the tree to the value you want, outputting a 0 every time you take a left handbranch, and a 1 every time you take a right hand branch. (normally you traverse the tree backwards from the code you want and build the binary Huffman encoding string backwards as well, since the first bit must start from the top).

Decoding a Huffman encoding is just as easy : as you read bits in from your input stream you traverse the tree beginning at the root, taking the left hand path if you read a 0 and the right hand path if you read a 1. When you hit a leaf, you have found the code.

Generally, any Huffman compression scheme also requires the Huffman tree to be written out as part of the file, otherwise the reader cannot decode the data. For a static tree, you don't have to do this since the tree is known and fixed.

The easiest way to output the Huffman tree itself is to, starting at the root, dump first the left hand side then the right hand side. For each node you output a 0, for each leaf you output a 1 followed by N bits representing the value. For example, the partial tree in my last example above using 4 bits per value can be represented as follows:

000100 fixed 6 bit byte indicates how many bits the value

for each leaf is stored in. In this case, 4.

0 root is a node

left hand side is

10011 a leaf with value 3

right hand side is

0 another node

recurse down, left hand side is

10001 a leaf with value 1

right hand side is

10010 a leaf with value 2

recursion return

So the partial tree can be represented with 00010001001101000110010, or 23 bits. Not bad

Example(2):

File consisting of symbols (AAABAABABCDEDDBCDADA) required a file size using Huffman coding and compare it with the original file size if you use a fixed length encoding (3bit to represent each character)

Solution:

Probability of each symbol or number of frequency as follows:

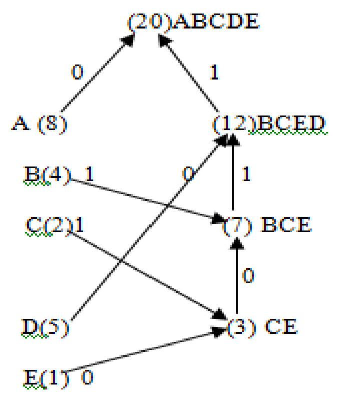
symbols	**Number of frequency**	**probability**
A	8	0.4
B	4	0.2
C	2	0.1
D	5	0.25
E	1	0.0

Table (1) shows number of frequency for all symbols

Note:

Probability=(number of frequency / summation of symbols)*

Through the table note that the E, C symbols are the least frequent, and so the huffman tree is configured as follows:



Figure(1): Huffman TreeFrom Huffman tree extract binary representation of each code as shown in the following table

symbols	code
A	**0**
B	111
C	1101
D	10
E	1100

Table (2) each symbol has its own coding

To calculate the file size using Huffman coding is as follows:

symbols	**Frequency value**	**Length of code**	**Overall size of the code in the file**
A	8	1	8
B	4	3	12
C	2	4	8
D	5	2	10
E	1	4	4
total			***42***

Table (3) shows the account file size

Hence the file size after compression algorithm using Huffman 42 bits, while the original file size before compressing (3 * 20 = 60 bits). Note that 20 is the number of characters.

IV. METHOD OF COMPARISON BETWEEN HUFFMAN AND RLE

The comparison process between compression algorithms(Huffman and RLE) using c++ program to compress the files[13]and[14] and a mathematical formula to calculate the compression ratio with Microsoft excel program , which are as follows:[12]

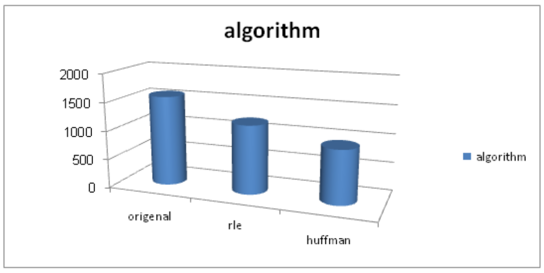
Compression ratio=size after compression/size before compression

V. RESULT AND DISCUSSION

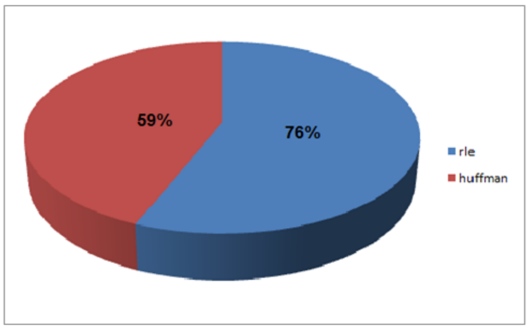
Has the testing process on 30 file in different sizes, and we compress itswith Huffman algorithm and RLE algorithm, using compression ratio coefficient and then we carried out a comparison between these files size with the original size(bytes). The results as shown in Table (4) and figure (2 and 3).Where results showed a high compression ratio on the Huffman algorithm more than RLE algorithm when compared to the size of the original file.

Original	**RLE**	**Huffman**
9	5	2
12	5	2
15	10	5
18	10	5
21	15	8
24	15	8
27	20	12
30	20	16
33	25	16
36	25	21
39	30	21
42	30	25
45	35	25
48	35	24
51	40	29
54	40	29
57	45	34
60	45	34
63	50	39
66	50	39
69	55	44
72	55	44
75	60	49
78	60	49
81	65	54
84	65	54
87	70	59
90	70	59
93	75	63
96	75	68
	total	
1575	1200	937

Table(4): Shows the original file size and then compressed using an algorithm (Huffman and Rle)



Figure(1): Shows the comparative sizes of the compressed files algorithms with the original file



Figure(2): Shows the compression ratio of each algorithm (Huffman and rle) with the original file size

VI. CONCLOSION

Search results indicated that we have acquired them by using the compression ratio coefficient that Huffman algorithm with greater efficiency in file compression. So that it compresses the original file by more than 40% while Rle algorithm compresses the original file by less than 23%. This result proves that preference Huffman algorithm more than Rle algorithm subject to further study.

VII. REFERENCES

[1] Wade, Graham (1994). Signal coding and processing (2 ed.). Cambridge University Press. p. 34. ISBN 978-0-521-42336-6. Retrieved 2011-12-22. The broad objective of source coding is to exploit or remove 'inefficient' redundancy in the PCM source and thereby achieve a reduction in the overall source rate R.

[2] Mahdi, O.A.; Mohammed, M.A.; Mohamed, A.J. (November 2012). "Implementing a Novel Approach an Convert Audio Compression to Text Coding via Hybrid Technique".International Journal of Computer Science Issues 9 (6, No. 3): 53–59. Retrieved 6 March2013.

[3] Pujar, J.H.; Kadlaskar, L.M. (May 2010). "A New Lossless Method of Image Compression and Decompression Using Huffman Coding Techniques". Journal of Theoretical and Applied Information Technology 15 (1): 18–23.

[4] Salomon, David (2008). A Concise Introduction to Data Compression. Berlin: Springer.ISBN 9781848000728.

[5] Tank, M.K. (2011). Implementation of Limpel-Ziv algorithm for lossless compression using VHDL. Thinkquest 2010: Proceedings of the First International Conference on Contours of Computing Technology. Berlin: Springer. pp. 275–283.

[6] Dale, Nell B., and John Lewis. "Chapter 3.1." Computer Science Illuminated. Boston: Jones and Bartlett, 2002. 54-55. Print.

[7] http://www.fileformat.info/mirror/egff /ch09_03.htm

[8] http://rosettacode.org/wiki/Huffman_ coding#JavaScript

[9] https://www.siggraph.org/education/ materials/HyperGraph/video/mpeg/mpegfaq/ huffman_tutorial.html

[10] Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes". Proceedings of the IRE 40 (9): 1098–1101. doi:10.1109/JRPROC.1952.273898. edit

[11] van Leeuwen, Jan (1976). "On the construction of Huffman trees". ICALP: 382–410. Retrieved 20 February 2014.

[12] Ze-Nian Li and Mark S. Drew , Fundamentals of Multimedia , ISBN: 0130618721, Prentice-Hall, 2004

[13] http://rle.codeplex.com/

[14] http://code.activestate.com/recipes/577480-huffman-data-compression/