Performance Analysis Of Several Lossless String Compression Algorithms On Text Files

Arnav Chhabra
Dept. Of Computer Science
PES University,Bangalore
arnav.chh@gmail.com

Shailesh Sridhar
Dept. Of Computer Science
PES University,Bangalore
shailesh.sridhar@gmail.com

Sailesh Gaddalay
Dept. Of Computer Science
PES University,Bangalore
sailesh.gaddalay@gmail.com

***Abstract*— Data compression is the process of modifying , encoding or converting the bits into a form such that data stored takes less space on the disk. There are two types of compression techniques either lossy or lossless. In this report we analyse six different lossless compression algorithms (namely, LZ77,LZ78,RLE,Huffman coding,Shannon Fano coding and LZW) by plotting graphs of their compression ratio vs size of file and time of execution vs size of file.**

I. INTRODUCTION

Data compression refers to reducing the amount of space needed to store data or reducing the amount of time needed to transmit data. The size of data is reduced by removing the excessive information. Data compression can be lossless, only if it is possible to exactly reconstruct the original data from the compressed version . To compress something means that you have a piece of data and you decrease its size .

Each technique has its own advantages and disadvantages. The different types of source data are medical images , text and images being preserved for legal reason , some computer executable files etc.

The general principle with text files is that we transform the string of characters to another string which contains the same information but the length of the string is much smaller. The efficiency is measured based on compression ratio, time of execution based on the size of the text file.

A. Lossless vs Lossy Compression

Lossless compression : Reduces bits by identifying and eliminating statistical redundancy no information is lost in lossless compression. The compressed data once it is uncompressed will be the same as the original data.

Lossy compression: Reduces a file by permanently eliminating certain information. When the file is uncompressed only a part of the original information is still there. Lossy compression is used for video and sound , where a certain amount of information loss is undetected by the users.
Compression ratio = B1/B0
B0 : number of bits in uncompressed file.
B1 : number of bits in compressed file

In this work we implement six different compression algorithms and attempt to draw conclusions on them by visualizing their compression ratios and times of execution.

II. SHANNON FANO ENCODING

In ShannonFano coding, the procedure is done by using a more frequently occurring string which is encoded by a shorter encoding vector and a less frequently occurring string is encoded by a longer encoding vector. Shannon-Fano coding relies on the occurrence of each character or symbol with their frequencies in a list and is also called as a variable length coding.

A. Pseudo Code Compression

```
1: begin
2:          count source units
3:          sort source units to non-decreasing order
4:          SF-SplitS
5:          output(count of symbols, encoded tree, symbols)
6:          write output
7: end
8: procedure SF-Split(S)
9: begin
10:          if (mod(S) greater than 1) then
11:          begin
12:               divide S to S1 and S2 with about same count
of units
13:              add 1 to codes in S1
14:              add 0 to codes in S2
15:               SF-Split(S1)
16:               SF-Split(S2)
17:          end
18: end
```

III. HUFFMAN ENCODING

It is a relatively more successful method used for text compression. It is a compression algorithm used for lossless data compression.It is a more efficient approach as it uses a variable-length representation, where each character

can have a different number of bits to represent it. More specifically we first analyze the frequency of each character in the text, and then we create a binary tree (called Huffman tree) giving a shorter bit representation to the most used characters, so that they can be reached faster and a longer bit representation to the lesser used characters.

A. Pseudo Code Compression

```
1: n := mod(C);
2: Q := C;
3: for i := 1 to n minus 1 do
4:         allocate a new node z
5:         z.left := x := Extract-Min(Q);
6:         z.right := y := Extract-Min(Q);
7:         z.freq := x.freq + y.freq;
8:         Insert(Q, z);
9: end for
10: return Extract-Min(Q); return the root of the tree
```

IV. LZ77

LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a length-distance pair, which is equivalent to the statement each of the next length characters is equal to the characters exactly distance characters behind it in the uncompressed stream. (The distance is sometimes called the offset instead.)

A. Pseudo Code Compression

```
1: while input is not empty do
2:         prefix := longest prefix of input that begins in
window
3:         if prefix exists then
4:             i := distance to start of prefix
5:             l := length of prefix
6:             c := char following prefix in input
7:         else
8:             i := 0
9:             l := 0
10:             c := first char of input
11:         end if

12:         output (i, l, c)

13:         s := pop l+1 chars from front of input
14:         discard l+1 chars from front of window
15:         append s to back of window
16: repeat
```

V. Run Length Encoding (RLE)

Run Length Encoding (RLE) is a simple and popular data compression algorithm. It is based on the idea of replacing a long sequence of the same symbol by a shorter sequence and is a good introduction to the data compression field for newcomers.RLE requires only a small amount of hardware and software resources. This is a result of RLE having been introduced very early, and a large range of derivatives have been developed up to now.

A. Pseudo Code Compression

```
1: Loop: count = 0
2:         REPEAT
3:             get next symbol
4:             count = count + 1
5:         UNTIL (symbol unequal to next one)
6:         output symbol
7:         IF count greater than 1
8:             output count
9:         GOTO Loop
```

VI. Lempel-Ziv-Welch (LZW)

The LZW algorithm is a very common compression technique. This algorithm is typically used in GIF and optionally in PDF and TIFF. Unixs compress command, among other uses. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the widely used Unix file compression utility compress, and is used in the GIF image format. The Idea relies on reoccurring patterns to save data space. LZW is the foremost technique for general purpose data compression due to its simplicity and versatility. It is the basis of many PC utilities that claim to double the capacity of your hard drive.

A. Pseudo Code Compression

```
1: s = empty string;
2: while (there is still data to be read)
3:         ch = read a character;
4:         if (dictionary contains s+ch)
5:             s = s+ch;
6:         else
7:             encode s to output file;
8:             add s+ch to dictionary;
9:             s = ch;
10: encode s to output file;
```

VII. LZ78

LZ78-based schemes work by entering phrases into a dictionary and then, when a repeat occurrence of that particular phrase is found, outputting the dictionary index instead of the phrase.

Every step LZ78 will send a pair (i,a) to the output, where i is an index of the phrase into the dictionary and a is the next symbol following immediately after the found phrase. In most implementations the dictionary is represented like the trie with numbered nodes. If we go from the root to a certain node, we will get phrase from the input text.

In each step we look for the longest phrase in dictionary, that would correspond to the unprocessed part of the input text. Index of this phrase together with the symbol, which

follows the found part in input text, are then send to the output. The old phrase extended by the new symbol is then put into dictionary. This new phrase is numbered by the smallest possible number.

A. Pseudo Code

```
1: begin
2:          initialize a dictionary by empty phrase P
3:          while (not EOF) do
4:          begin
5:              readSymbol(X)
6:              if (F.X¿ is in the dictionary) then
7:                  F = F.X
8:              else
9:                begin
10:                   output(pointer(F),X)
11:                   encode X to the dictionary
12:                   initialize phrase F by empty character
13:                 end
14:           end
15: end
```

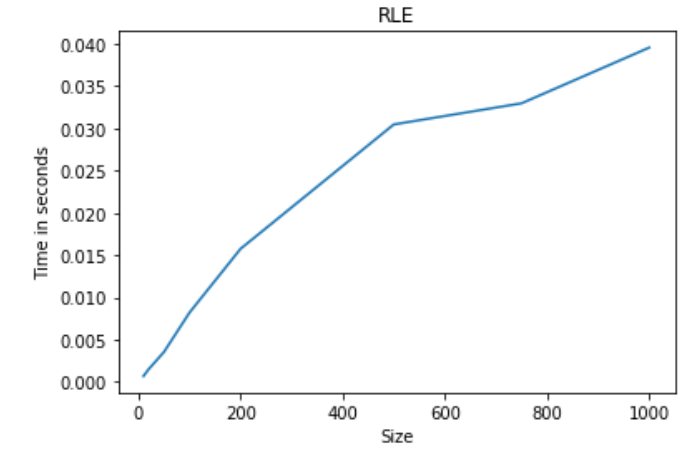
VIII. Approach

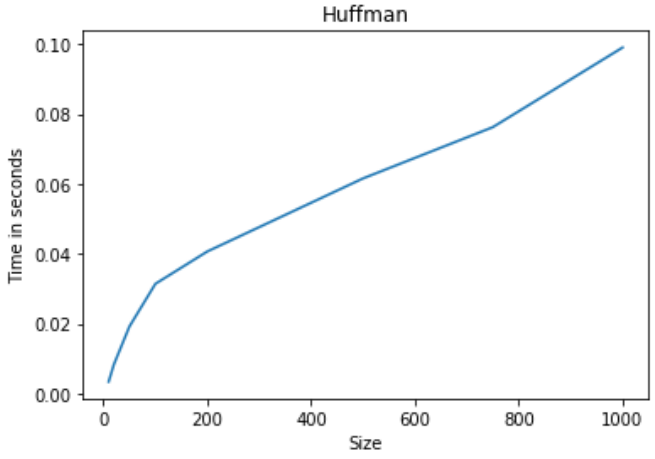
In order to analyze the performance of these algorithms we created four different types of input files:
Type 0:
Standard everyday text, like that you would see in a book.
Type 1:
Contains long strings of repeated characters as patterns.
Type 2:
Contains many repeated patterns, but not necessarily strings of repeated patterns as in Type 2
Type 3:
Contains completely random text generated using a random text generator
For each of these types, we created 8 different files of different sizes;10kb,20kb,50kb,100kb,200kb,500kb,750kb and 1000kb. We then ran these algorithms using these files, plotting the compression ratio with respect to size for each different file type, as well as the time taken by each algorithm w.r.t file size when run on type 0 files.

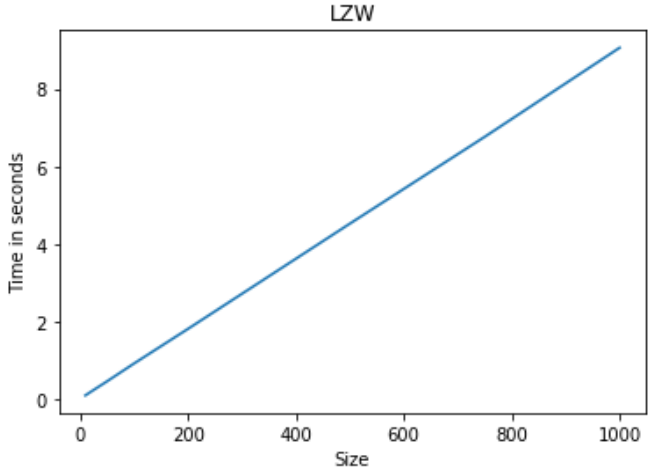
Finally we analyze these results and attempt to arrive at meaningful conclusions.It is to be noted that n refers to the size of the input.

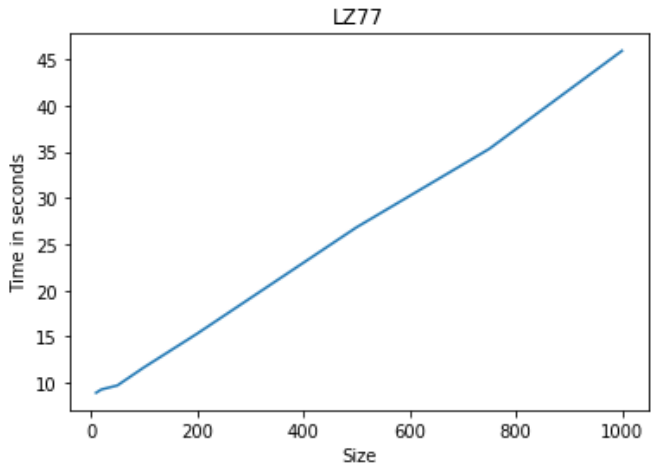
IX. Results and Analysis

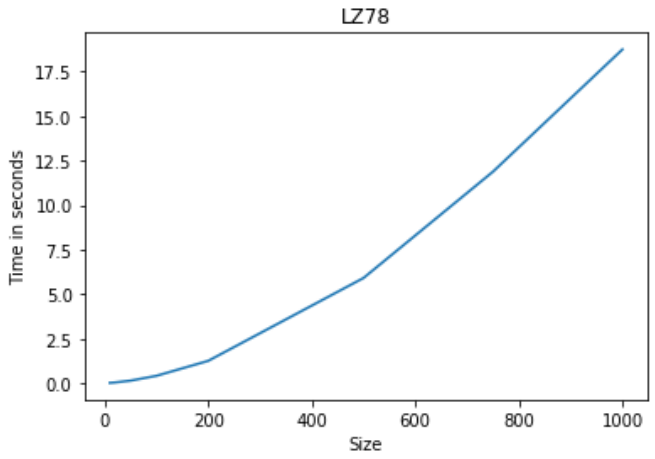
A. Execution Time VS Size Comparison

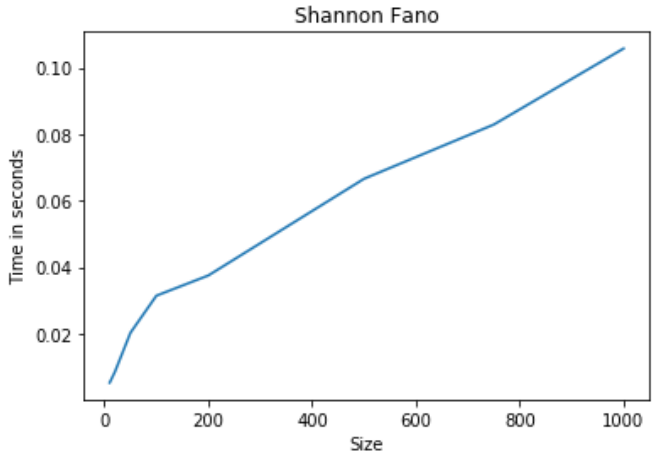






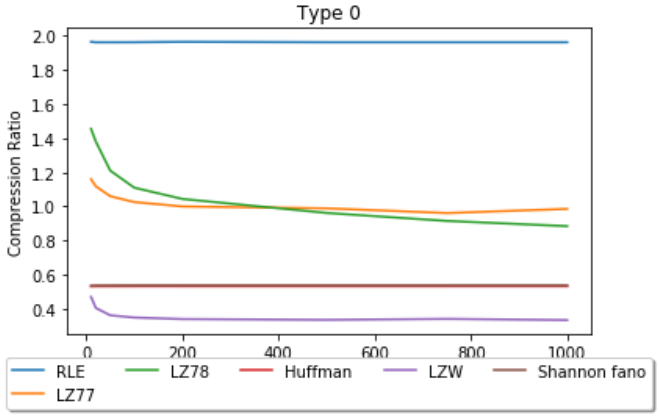






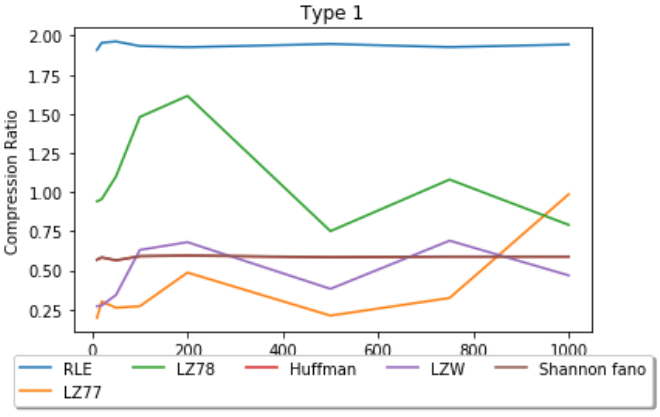
The general pattern noticed in the above graphs is that the time taken for compression for all the algorithms increases linearly with the increase of the size of the files. This matches the expectation that all these algorithms have O(n) time complexity.

B. Compression Ratio vs Size Comparison

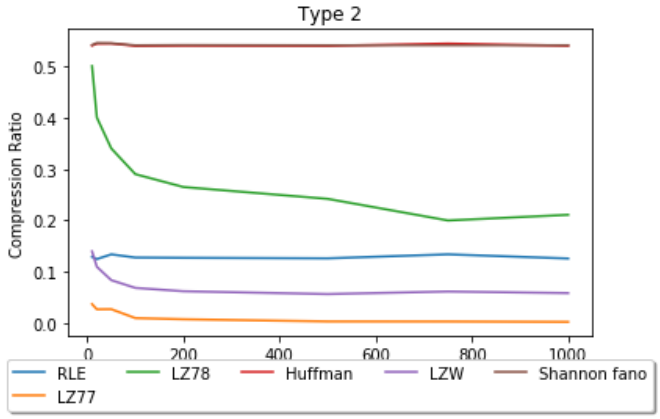


The graph of compression ratios for input 0 is the perhaps the most important of the four, as it represents the performance of these compression algorithms for general every day text.

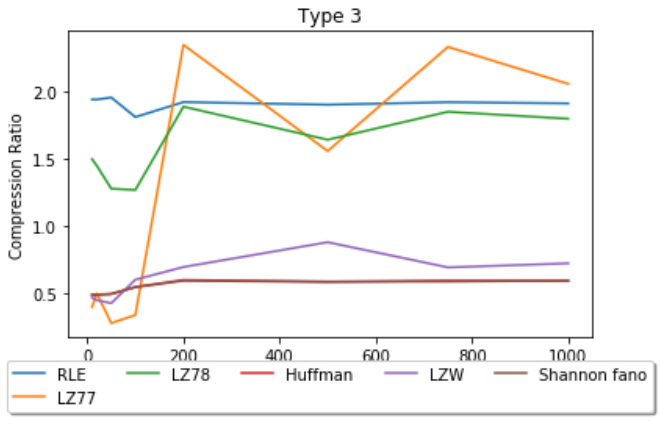
- As there are close to no strings of repeating patterns, RLE only increases the size of the output file, making its performance very bad.
- LZ77 and LZ78 perform reasonably the same.
- Shannon fano and Huffman both perform almost identically, owing to the similarity in the algorithms.
- LZW performs the best among the six algorithms, with a compression ratio of roughly 0.34.



While RLE,Huffman and Shannon Fano seem to show reasonable behaviour and smooth graphs, LZW, LZ77 and LZ78, the three look-up based algorithms seem to show no clear trend in their graphs. Type 1 input files contain randomly repeated patterns of characters. As the performance of look-up based algorithms depends strongly on the previous appearance of patterns, the compression ratio is affected greatly by the exact string in the graph and can not be expected to show a smooth trend.



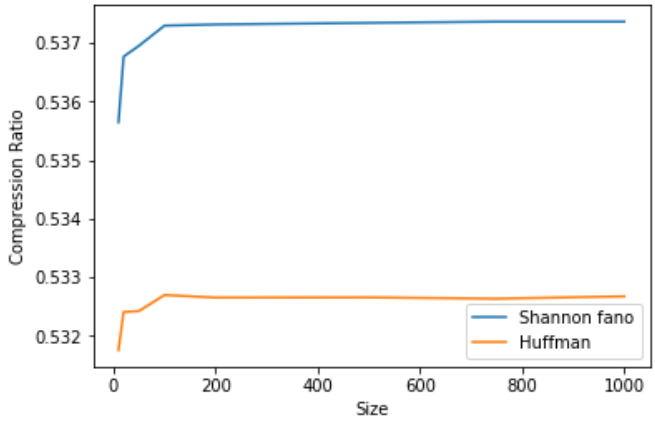
The most important feature of this graph is the huge improvement in performance of RLE. RLE depends heavily on long strings of repeated patterns in order to be successful. As type 2 input files contain text conforming to this description, RLE is able to perform well. Another observation is that the look-up based algorithms have almost constant compression ratio. This is owing to the fact that even in larger files, the same patterns are repeated and are already present in the dictionary.



While RLE,Shannon Fano and Huffman showcase predictably smooth behaviour, the three look-up based algorithms have completely haphazard graphs, again displaying their dependence on the exact nature of the input. Shannon Fano and Huffman are the best performers for the input type 3.

X. Observations

- RLE performs poorly in all cases except when there are long repeated strings of characters.
- Shannon fano and Huffman maintain the same performance for all the input types and sizes, with a compression ratio between 0.5 and 0.55. This is can attributed to the fact that they depend on only character frequency and not on patterns whatsoever



- In all the compression ratio graphs, the performances of Shannon-Fano and Huffman are so similar that their plots completely overlap. However upon closer inspection, it seems that Huffman always performs slightly better than Shannon Fano.
- The look-up based algorithms LZ77, LZ78 and LZW depend heavily upon the nature of the text in the input text and the patterns present in them.LZW was proposed as an improvement to LZ77 and LZ78, and in general performs better.These three algorithms show an increase in performance when the file size is sufficiently large, as a larger size implies that it is more likely for a pattern to be repeated.
- LZ77 only looks at the last 'k' characters corresponding to the window size, while LZ78 maintains a dictionary. LZ77 discards information as its window moves past it in the text.This is why LZ78 shows a larger improvement in performance than LZ78 as input size increases. However, when there are long repeaed patterns LZ77 tends to perform better because of the nature of its algorithm.

XI. Conclusions

- For standard text, such as articles and textbooks LZW performs the best among the six algorithms.
- For use cases when the input text is almost random or when the type of input is unknown, we would recommend Shannon Fano or Huffman as they provide a good compression ratio regardless of the patterns appearing in the text.
- For use cases where there are several repeated patterns such as in genome sequences, we would recommend LZ77.