Project Report

On

Data Compression Techniques

Submitted By: Kunal Sagar(180001028) and

Jay Bangar(180001022)

Computer Science and Engineering

2nd year

Under the Guidance of

Dr.Kapil Ahuja

Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2020

Contents

Introduction

Data compression is used to reduce the size of a particular file which in turn reduces the required storage space and makes the transmission of data quicker. The extremely fast growth of data that needs to be stored and transferred has given rise to the demands of better transmission and storage techniques. Various lossless data compression algorithms have been proposed and used. Huffman Coding, Arithmetic Coding, Run Length Encoding Algorithm are some of the techniques in use.

Data Compression Methods:

1. **Lossless-** Lossless compression enables the restoration of a file to its original state, without the loss of a single bit of data, when the file is uncompressed. This type of compression is the typical approach with executables, as well as text and spreadsheet files, where the loss of words or numbers would change the information.

2. **Lossy-**Lossy compression permanently eliminates bits of data that are redundant, unimportant or imperceptible. It is useful with graphics, audio, video and images, where the removal of some data bits has little or no discernible effect on the representation of the content.

Objective

- Analyze and compare the following lossless data compression algorithms:
    - Run Length Encoding
    - Arithmetic Coding
    - Huffman Coding
    - Adaptive Huffman Coding
    - Word By Word (Proposed Algorithm)
- Implement all the above mentioned algorithms for text compression.
- Study Compression Ratios on different text files and attempt to improve the compression.

Algorithm Design

- ## Run Length Encoding :

History : "Run-length encoding (RLE) schemes were employed in the transmission of analog television signals as far back as 1967. n 1983, run-length encoding was patented by Hitachi."

Data files frequently contain the same character repeated many times in a row. Such repetition results in unnecessary utilization of storage space. This shortcoming can be overcome by storing the repeated character along with its frequency. This is the natural idea behind Run-Length Encoding, a lossless data compression algorithm.

Example : Given the input sequence as '**AAAAAABBBBBCCCCCCCAAA**' the output generated by the RLE would look like '**A6B5C7A3**'. For the given sequence the algorithm was able to compress the data from **21 characters to 8 characters.**

Pseudo Code :

Encoder :

```
Loop: count = 0
   REPEAT
     get next symbol
     count = count + 1
   UNTIL (symbol unequal to next one)
      output symbol
   IF count > 1
     output count
   GOTO Loop
```

Decoder :

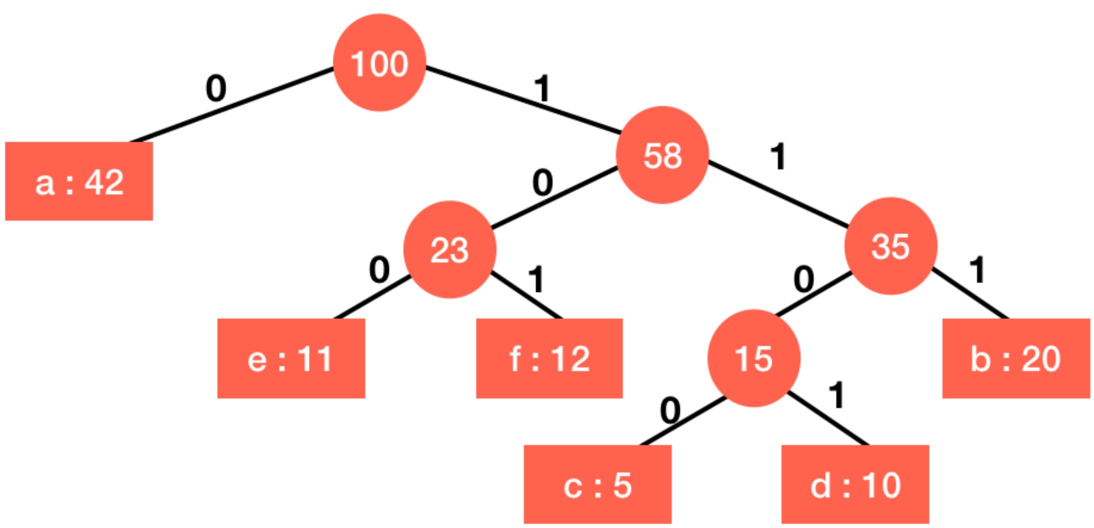
```
Loop:
   REPEAT
      get next symbol
      read the next integer n
      for(i = 1 to n)
          output (symbol)
      end for
```

- ## Huffman Encoding :

History: The Huffman algorithm was developed by David A. Huffman while he was a Sc.D. student at MIT, and published it in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

It is a variable length coding technique. Since the total number of characters are fixed (i.e. 256), the repetition of each character in a text file is inevitable. Therefore assigning a code, small in length, to a character which has high frequency is a **Greedy Approach.** As Huffman encoding is a two pass algorithm so it takes additional time for building a Huffman tree, and provides better compression ratio. Since the frequencies of symbols vary across messages, there is no one Huffman coding that will work for all messages. In most of the cases, the length of encoded message will always be less than the original message due to fixed number of characters(i.e.256) as the maximum possible height of tree is $Log_2(256) = 8$ (size of a character), hence it guarantees compression.

Construction of Tree : We construct a Huffman tree by maintaining a min heap for frequencies. Initially the min heap contain leaf nodes with characters and their frequencies. In each iteration we pop two minimum frequency nodes and connect them with a newly formed internal node whose frequency is equal to sum of frequency of its child. We repeat until min heap contains a single node i.e. root node.
Assigning codes : '0' bit is assigned to the left child and '1' bit to the right child until all the nodes are traversed.
Example: Value near the character represent their frequencies.



Link for Image

Codes: a **– 0** , b **-111** , c – **1100** ,d – **1101** ,e – **100** , f – **101**.

Pseudo Code :

Encoder :

```
Procedure Huffman(C):     // C is the set of n characters with its frequency.
n = C.size()
Q = priority_queue()
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
while Q.size() is not equal to 1
    Z = new node()
    Z.left = x = Q.pop
    Z.right = y = Q.pop
    Z.frequency = x.frequency + y.frequency
    Q.push(Z)
end while
Return Q
```

Assign codes :

```
traverseNode(n, code) :
if (leftChild(n) != NULL and rightChild(n) != NULL) then
        traverseNode(leftChild(n), code+'0')     //traverse through the left child
        traverseNode(rightChild(n), code+'1')    //traverse through the right child
else  store the character and data of current node.
```

Decoder :

```
Procedure Decompress (root, S): // S refers to bit-stream to be decompressed
n := S.length()
for i := 1 to n
    curr = root
    while curr.left != NULL and curr.right != NULL
        if S[i] is equal to '0'
            curr := curr.left
        else
            curr := curr.right
        endif
        i := i+1
    endwhile
    print curr.symbol
endfor
```

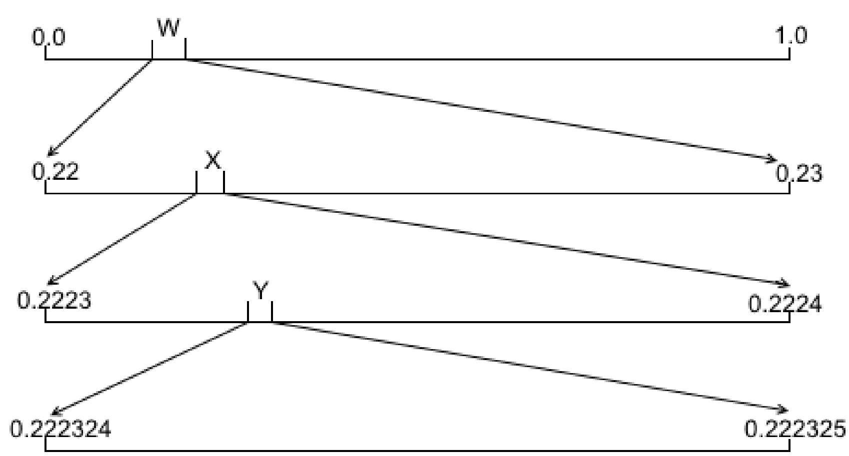
- ## Arithmetic Encoding :

History: Arithmetic codes were invented by Elias, Rissanen and Pasco, and subsequently made practical by Witten in 1987.

Arithmetic coding takes a message composed of symbols and converts it to a floating point number greater than or equal to zero and less than one. This algorithm relies on a model to characterize the symbols it is processing. The job of the model is to tell the encoder what the probability of a character is, in a given message. Huffman Coding is not efficient when the probability distribution of characters is highly unsymmetrical while the Arithmetic Coding works fine in such cases.

Initially we start with a range from 0 to 1 and after reading each character the range is sliced further according to its probability. The final encoded number is in between 0 and 1.
As floating point variables have limited precision hence we have implemented the algorithm with **Dynamic Model** in **16 bit binary form**.

Example: Let's encode a string "**wxy**".
The arithmetic coder maintains two numbers, low and high, which represents a subinterval [low,high) of the range [0,1). Initially low = 0 and high = 1. Let the static model range of w = [0.22,0.23), x = [0.23,0.24) and y = [0.24, 0.25).After encoding first letter 'w' range [0,1) sliced up to [0.22, 0.23). Encoding 2nd letter 'x' further sliced the range to [0.2223,0.2224). Finally, after encoding 'y' the final range of encoded string lies between [0.222324, 0.222325).
Final encoded message = 0.22232450



Link for image

Pseudo Code :

cum_freq[0] = total number of characters

Encoder :

```
  begin encode_each_symbol(symbol)
     range = high - low;
     high = low + (range * cum_freq [symbol - 1]) / cum_freq [0];
     low = low + (range * cum_freq [symbol]) / cum_freq [0];
     loop :
       if (high is less than half ) output_bit(0)
       else if (low is greater than equal to 0.5)
           output_bit(0)
           subtract 0.5 from low and high
       else if (low is greater than equal to 0.25  and high is less than 0.75)
           penfding_bits++
          subtract 0.25 from low and high
       else end loop;
    left_shif low and high
    update model()
end
```

Decoder :

```
function decodeSymbol ()
     range = high – low
     cum = ((((value - low) + 1) * cum_freq[0] - 1) / range);
     symbol_index = search_symbol_index();
     //slice up the range
     high = low + (range * cum_freq[symbol_index - 1]) / cum_freq[0];

     low = low + (range * cum_freq[symbol_index]) / cum_freq[0];

     loop :

        if (high is less than half)               // do nothing

        else if (low is greater than equal to 0.5)
          subtract 0.5 from low, high and value
       else if (low is greater than equal to 0.25  and high is less than 0.75)
            subtract 0.25 from low, high and value
       else end loop;

     shift left low, high and value
     add new bit from stream to value
     return symbol_index;
```

- ## Adaptive Huffman Encoding :

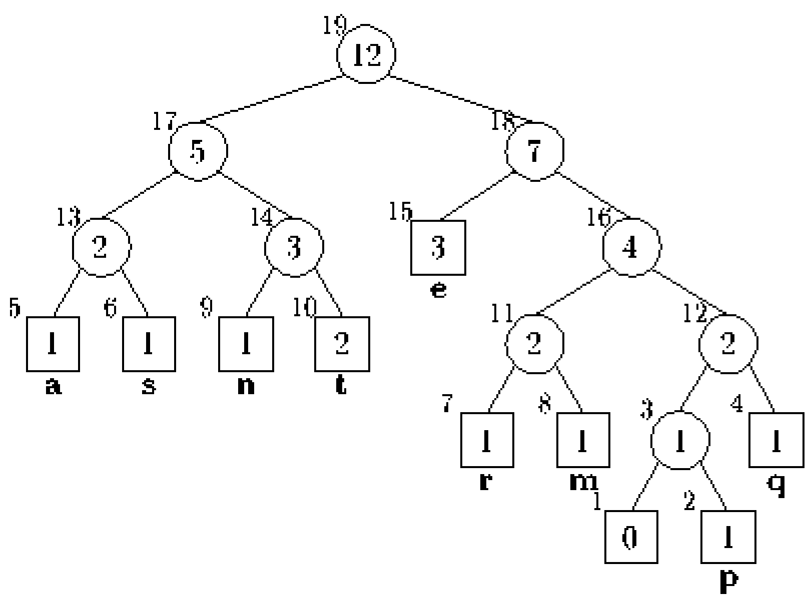
History: Algorithm is based on the classical Huffman coding method. The oldest adaptive algorithm was published by Faller (1973) and later Gallager (1978), independently. At 1985 Knuth made a little modification, and so the algorithm was called FGK(Faller-Gallager-Knuth).

Since Huffman is a 2 pass (one pass for collecting the frequency and other for the encoding) algorithm and also decoder for the same require some additional information of the frequency table beforehand but there are many cases which require live compression i.e. compressing the data(text) as soon as it has been received without being concerned for the upcoming data. It is a one pass algorithm designed for such cases. In this **Greedy Algorithm** the data encountered is added to the tree and the tree is updated accordingly to provide small length code for more frequent characters. Decompression also works in similar way and does not require any information of data beforehand .At any moment the state of tree is same in compression and decompression.

The two rules for updating the tree are:

1. All nodes in our tree (except for the root) must have a sibling.
2. The nodes must be listed (from left to right, bottom to top)in order of increasing frequency and order.

Example of a FGK Tree:



Link for image

Pseudo Code :

Updating the tree:

```
procedure update_tree(U)
begin
  while (U!=root) do
   begin
     if (exists node U1 with same value and greater order) then
        change U1 and U
     increment value of U
     U := parent(U)
   end
  increment value of U, update leaf codes
end
```

Encoder :

```
  // NYT -> Not Yet Transferred
  begin
    create NYT node
    readSymbol(X)
    while (X!=EOF) do
    begin
      if (first_read_of(X))
        output(NYT)
        output(X)
        create new node U with next nodes NYT and new node X
        update_tree(U);
     else
        output(X)
        update_tree(X)
     readSymbol(X)
   end
end
```

Decoder :

```
begin
    n := S.length()
    for i := 1 to n
       curr = root
       while curr.left != NULL and curr.right != NULL
          if S[i] is equal to '0'
             curr := curr.left
          else
             curr := curr.right
          endif
          i := i+1
       endwhile
       if(curr == NYT)
            char C = read_next_8_bits
            create new node U with next nodes NYT and new node C
            output(C)
       else
            char C = curr.symbol
            output(C)
       end if
    update_tree(C)
    endfor
```

- ## Word By Word Adaptive Huffman (Proposed Algorithm):

The existing Adaptive Huffman compresses the input text file by considering the number of times a particular byte(character) has occurred and works in such a way that the frequently occurred byte get shorter code word. In this **Greedy Optimization** we devise a mechanism which compresses the given text file by taking one word at a time rather than one byte. While encoding we maintain a uniform dictionary and an intermediate file which stores all the unique words from the input text file. The intermediate file is passed along with the encoded message to the decoder which helps in making its own dictionary.

Pseudo Code :

Encoder :

```
process dictionaryUpdate()
begin
 readWord(W)                          // reading from input file
 while(W != EOF)do
     begin
       if(first_read_of(W))
         add W to dictionary and Intermediate file
 m = total number of words in dictionary
 e = log2(m)
 r = m – 2^e          // 0 <= r < 2^e
for (first 2*r words)
  code(W) :=  binary value of index(w) in (e+1) bits          // index starts from zero.
end for
for (rest of the words)
  code(W) :=  binary value of (index(w) – r) in (e) bits
end for

// NYT -> Not Yet Transferred
Process encoding()
begin
     create NYT node
     readWord(W)
     while (W!=EOF) do
      begin
        if (first_read_of(W)) then
         begin
           output_code(NYT)
           generate_index_code(W)
           create new node U with next nodes NYT and new node W
           update_tree(U);
         end
        else
         begin
           output_code(W)
           update_tree(W)
         end
        readWord(W)
      end
  end
```

```
procedure update_tree(U)
       begin
          while (U!=root) do
           begin
             if (exists node U1 with same value and greater order) then
                change U1 and U
             increment value of U
             U := parent(U)
           end
          increment value of U, update leaf codes
       end
```

Decoder :

```
     process dictionary_update()
     begin
      readWord(W)                              // reading from intermediate file
      while(W != EOF)do
          begin
            if(first_read_of(W))
              add W to dictionary
            else continue
      m = total number of words in dictionary
      e = log2(m)
     r = m – 2^e        // 0 <= r < 2^e
     for (first 2*r words)
       code(W) :=  binary value of index(w) in (e+1) bits             // index starts from zero.
     end for
     for (rest of the words)
       code(W) :=  binary value of (index(w) – r) in (e) bits
     end for

  process decoding()
     begin
          n := S.length()
          for i := 1 to n
              curr = root
```

```
      while curr.left != NULL and curr.right != NULL
         if S[i] is equal to '0'
            curr := curr.left
         else
            curr := curr.right
         endif
         i := i+1
      end while
      if(curr == NYT)
           q = read_next_e_bits
           if (q<r)
             left shift (q)
             q = q + read_next_bit()
           else
             q = q + r
           word W = dictionary[q]          //word at the index q
           create new node U with next nodes NYT and new node W
           output(W)
      else
           word W = curr.word
           output(W)
      end if
   update_tree(W)
   end for
```

```
procedure update_tree(U)
 begin
   while (U!=root) do
    begin
      if (exists node U1 with same value and greater order) then
        change U1 and U
      increment value of U
      U := parent(U)
    end
   increment value of U, update leaf codes
 end
```

Implementation

Click here to view the implementation of all the above mentioned algorithms along with test files.

Algorithm Analysis

- ## Run Length Encoding

Time Complexity:

Encoding :

- Consider an input file of total 'X' characters. So the length would be |X|.
- Traversing through the input file takes O(|X|) time.
- And other operations like maintaining a counter and printing the encoded message take constant time.

- **Hence, the total time in all cases : O(|X|)**

Decoding

- Consider the Encoded file of total length |E| .
- Traversing through the encoded file takes O(|E|) time.
- And other operations like maintaining a counter and string to number conversion takes constant time.

- **Hence, the total time in all cases : O(|E|)**

Space Complexity :

- The only space required is for maintaining local variables which does not depend on the input file size.

- **Hence, the total space : O(1)** (constant space).

- ## Huffman Encoding

  - Average / Best case : When the tree is balanced
  - Worst Case : When the tree is skewed.

  - Input File Length : |X| (Total 'X' symbols)
  - Number of Distinct characters: N

Time Complexity:

- Calculating the frequencies: O(|X|) time (Traversing the input File)
- Storing the frequency: O(|X| *Log N) (Search and Insertion takes O(Log N) in C++ STL Map)
- Insertion of Leaf Node in Priority Queue: O(N) (Insertion of N leaf nodes)
- For making the Huffman tree
- Total number of steps : N - 1 (total number of nodes in PQ)
- Insertion and Heapify : O(log N)
- Total time for building the tree : (N-1)*O(log N) < = c*N*log N = O (N*log N)

Encoding

- Traversing each symbol in Input File: O(|X|)
- Searching for a symbol in tree : O (log N) [O(N) in worst case]

- **Total time for encoding : O(|X| * log N) [O(|X|*N) in worst case]**

Decoding

- Number of Tree Traversal : |X| (Number of characters in Input File)
- Traversing for a symbol in tree : O (log N) [O(N) in worst case]

- **Total time for decoding : O(|X| * log N) [O(|X|*N) in worst case]**

Space Complexity:

- The space required for Huffman Tree = N
- The extra space required (maps, other local variables, etc) is bounded by N.

- **Hence, total space complexity : O(N).**

• Arithmetic Coding

- Input File Length : |X| (Total 'X' symbols)
- Number of Distinct characters: N
- Encoded File Length: |E|

Time Complexity:
Encoding

- Time for traversing the input file : O(|X|)
- Encoding symbols and normalization : O(1) (Constant Time)
- Updating the model : O(N)

- **Hence, the total time in Encoding : O(|X| *N)**

Decoding

- Time for traversing the bit stream : O(|E|)
- Normalization : O(1)   (Constant Time
- Updating the model : O(N)

- **Hence, the total time in Decoding: O(|E|*N)**

Space Complexity :

- An array is used for maintaining the frequency table.
- There are two more arrays for index to character conversion and vice-versa.

- **Hence, the total space complexity : O(N).**

Adaptive Huffman Encoding

- Average Case : When the tree is nearly balanced.
- Worst Case : When the tree is skewed.
- Input File Length : |X| (Total 'X' symbols)
- Number of Distinct characters: N
- Encoded File Length: |E|

Time Complexity:

Encoding

- Traversing the input file: O(|X|)
- Searching the Symbol: O (log N)   [O(N) in worst case]
- If Not Found
  - Generating NYT Code : O (log N)   [O(N) in worst case]
  - Printing NYT and ASCII of Symbol : O(1)
- Else
  - Generating the Symbol Code: O(log N)   [O(N) in worst case]
  - Updating the FGK Tree: O (log N)   [O(N) in worst case]
- **Total Time for Encoding : O (|X| * log N)**   **[O(|X| * N) in worst case]**

*****(Worst case is more frequent in practical scenarios)

Decoding

- Traversing the input file: O(|E|)
- Reach a Leaf Node
- If node is NYT => read next 8 bits : O(1)   (Constant Time)
- Print the Symbol: O(1)   (Constant Time)
- Updating the FGK Tree: O (log N)   [O(N) in worst case]

- **Total Time for Encoding : O (|E| * log N)   [O(|E| * N) in worst case]**

Space Complexity :

- The space required for the FGK Tree = N
- The extra space required (maps, other local variables, etc) is bounded by N

- **Hence, total space complexity : O(N).**

- ## Word To Word (Proposed Algorithm)

  - Average Case : When the tree is nearly balanced.
  - Worst Case : When the tree is skewed.
  - Input File Length : |X| (Total 'X' Words)
  - Number of Distinct Words: N
  - Encoded File Length: |E|
  - Intermediate File Length: |N|
  - Total time for making the dictionary (map): O(|X|*logN)   (Traversing the input file)
  - Assigning index to each word in dictionary : O(N) time.

Encoding

- Traversing the input file: O(|X|)
- Searching the Word: O (log N)   [O(N) in worst case]
- If Not Found
  - Generating NYT Code : O (log N)   [O(N) in worst case]
  - Generate Index Code: O(log N)
- Else
  - Generating the Word Code: O(log N)   [O(N) in worst case]
- Updating the Tree: O (log N)   [O(N) in worst case]

- **Total Time for Encoding : O (|X| * log N)   [O(|X| * N) in worst case]**

*(Worst case is more frequent in practical scenarios)

Decoding

- Total time for making the dictionary (map) from intermediate file : O(|N| *logN)
- Assigning index to each word in dictionary : O(N) time.
- Traversing the encoded stream: O(|E|)
- Reach a Leaf Node
- If node is NYT :
    - read next (e) or (e+1) bits : O(e) (e<=64 ~ constant time)
- Print the Word: O(1)   (Constant Time)
- Updating the Tree: O (log N)   [O(N) in worst case]

- **Total Time for Encoding : O (|E| * log N)   [O(|E| * N) in worst case]**

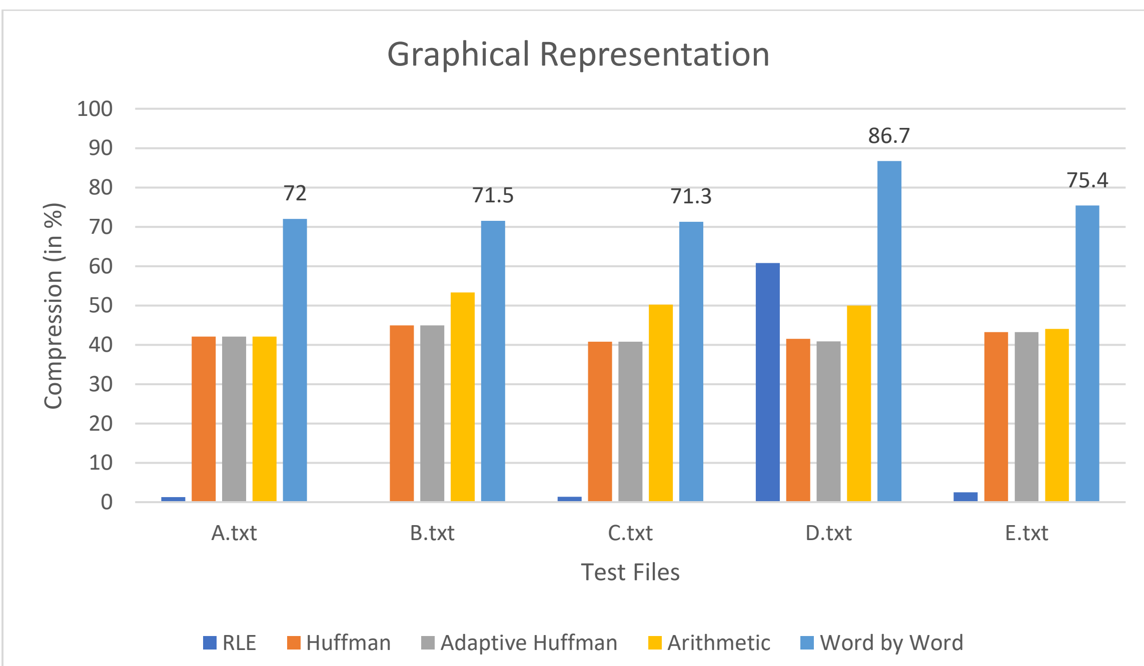
****

Observation

*Each cell in the table contains Size of the Encoded File, Compression Ratio in % and the Execution time

Input File Name	Input File Size	Run Length Encoding	Huffman Encoding	Adaptive Huffman Encoding	Arithmetic Encoding	Word by Word (Optimization)
A.txt	368 KB	363 KB (1.3 %) (0.188 s)	213 KB (42.1 %) (0.823 s)	213 KB (42.1 %) (1.599 s)	213 KB (42.1 %) (1.368 s)	103 KB (72.0 %) (58.436 s)
B.txt	510 KB	509 KB (0.0 %) (0.270 s)	281 KB (44.9 %) (0.987 s)	281 KB (44.9 %) (2.175 s)	238 KB (53.3 %) (1.551 s)	145 KB (71.5 %) (105.452 s)
C.txt	213 KB	210 KB (1.4 %) (0.715 s)	126 KB (40.8 %) (0.502 s)	126 KB (40.8 %) (1.038 s)	106 KB (50.2 %) (0.612 s)	61 KB (71.3 %) (38.734 s)
D.txt	166 KB	65 KB (60.8 %) (0.129 s)	97 KB (41.5 %) (0.382 s)	98 KB (40.9 %) (0.895 s)	83 KB (50.0 %) (0.450 s)	22 KB (86.7 %) (0.620 s)
E.txt	118 KB	115 KB (2.5 %) (0.133 s)	67 KB (43.2 %) (0.207 s)	67 KB (43.2 %) (0.666 s)	66 KB (44.0 %) (0.395 s)	29 KB (75.4 %) (7.293 s)

Quantitative Analysis of all the Algorithms



* Testing on large files size was omitted as the computation time in word by word (optimization) was taking several minutes due to the large number of words.

Conclusion

Run Length Encoding(RLE)

The compression achieved from the text file A, B, C and E is very insignificant because the repetition of character in consecutive manner is very low (barely more than 3). But the compression is significant in text file D as this file is specially generated to contain notable repetition of characters consecutively. Therefore, we can say that RLE works good when the consecutive repetition of characters is significant. The normal text files are not favourable for RLE compression due to less repetition. Hence, RLE is mostly used in Image Compression.

Huffman Encoding

The Huffman algorithm works well when the frequencies of the characters are almost equally distributed. The equal distribution make a balanced Huffman tree which implies the optimal length of code assigned to each character. In the worst case the probability of a symbol exceeds 0.5, resulting in a skewed binary tree.

Adaptive Huffman Coding

Adaptive Huffman is very sensitive to symbol as a small mistake in encoding a symbol results in permanent damage to the later encoded message. Adaptive Huffman generally compare well with static Huffman in average cases. In performance the Adaptive Huffman algorithm is never much worse than twice the optimal of Huffman Encoding.

Arithmetic Coding

The compression ratio of arithmetic coding is better than Huffman because it doesn’t use a discrete number of bits for each symbol. Low probability symbols use many bits, high probability use fewer bits and the probability changes dynamically resulting in better compression. The execution time is slightly greater than Huffman as Huffman uses static table whereas Arithmetic updates the frequency table dynamically.

Word By Word (Proposed Algorithm)

Word by Word compression mechanism is achieved by using a dictionary which is dynamically updated both by the sender and the receiver. This enables one to assign shorter codes to each word in the input file as compared to Huffman where the codes are assigned to each character. As a word contains multiple characters hence the code length assigned to a word would be higher in Huffman in comparison to the proposed algorithm. Hence we get higher compression in Word by Word. In most practical applications, the total number of unique words in the input file will always be more than the number of unique bytes. Hence the proposed algorithm has greater memory requirements for execution. The number of distinct words in this algorithm is not bounded like max number of characters(256) in Adaptive Huffman, hence the tree

building and traversal takes more time(linear) as the ‘N’ is very high. This results in higher execution time.
In conclusion, the proposed algorithm provides better compression ratio at the cost of greater memory requirement and execution time.

Future Work

This report focuses on application of algorithms on text file compression. But there is also a scope of image compression using all the algorithms for the future work.

References

- Analysis and Comparison of Algorithms for Lossless Data Compression
- Enhancing Adaptive Huffman Coding Through Word By Word Compression for Textual Data
- Analysis and comparison of Adaptive Huffman Coding and Arithmetic Coding Algorithms
- The Data Compression Book by Mark Nelson & Jean-Loup Gailly

********THE END********