University of Cincinnati

Date: 2/10/2016

I, Viveka Aggarwal, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:
Lossless Data Compression for Security Purposes Using Huffman Encoding

Student's name: Viveka Aggarwal

This work and its defense approved by:

Committee chair: George Purdy, Ph.D.

Committee member: Rui Dai, Ph.D.

Committee member: Carla Purdy, Ph.D.
Lossless Data Compression for Security Purposes Using Huffman Encoding

A thesis submitted to the Graduate School of University of Cincinnati in a partial fulfillment of requirements for the degree of Master of Science in the College of Engineering and Applied Science department

by

Viveka Aggarwal

Bachelor of Technology
SRM University, India

May 2013

Committee Chair: Prof. George Purdy, Ph. D.
ABSTRACT

The aim of this thesis is to efficiently compress data for security purposes. Here, we analyze and extend the working of Huffman encoding. Our results will ultimately provide us with a list of binary numbers that can be encrypted in AES. The encoding works on reducing the redundancy of the input, enabling us to change the key less often.

We use static and dynamic Huffman encoding, which are forms of compression techniques. We have modified Huffman encoding so that it uses the frequencies of consecutive letter pairs and triplets. The usual way to compress English text is with Huffman encoding on the individual letters. The character frequency used in this encoding schema can be obtained in two ways, either dynamically, that is calculating it on the flow (once the text is obtained as input) or statically (use a pre-created character frequency table). Using the static encoding method, with commonly available frequency table, the effort and the cost of encoding and sending the table can be prevented.
The main idea behind this work is dual and triple character encoding. For the purpose of this thesis, only letters are encoded.

We compare the results of Huffman encoding methods by encoding the input data both dynamically and statically. Through these results we analyze the best method for data compression for security purposes of different types of data (with frequently repeated and non repeated words).
TABLE OF CONTENTS

1. Introduction
   1.1 Goal of research
   1.2 Motivation

2. Input

3. Compression techniques
   3.1 Static Huffman Encoding
   3.2 Extended Static Huffman Encoding
      3.2.1 Pairs
      3.2.2 Triplets
   3.3 Dynamic Huffman Encoding
   3.4 Extended Dynamic Huffman Encoding
      3.4.1 Pairs
      3.4.2 Triplets

4. Results

5. Conclusion and Future work

6. References
1. INTRODUCTION

The usual way to compress text is using Huffman encoding\textsuperscript{[6]}. This minimum redundancy codes construction method was introduced by D. A. Huffman in 1952\textsuperscript{[1][2]}. This encoding scheme works on encoding each letter based on its frequency.

We extended Huffman encoding to encode pairs and triplets and also experimented with some dictionary based encoding techniques too such as LZW\textsuperscript{[3]}, arithmetic encoding\textsuperscript{[13]}. The modified Huffman encoding gave better results.

The reason behind this stems from the fact that the modified Huffman encoding technique creates a new tree for each letter or letter pair, thereby reducing the code length for each letter pair or triplet even though it encodes the same number of pairs or triplets as the number of letters in the input. Time complexity for modified Huffman encoding remains almost the same for the first GB of text.
1.1 GOAL OF RESEARCH

The aim of this thesis is to efficiently compress data to obtain a list of binary numbers that can be encrypted in Advanced Encryption Standard (AES)\(^7\), such that the key can be changed less often. The basis of our thesis is that English language has lot of redundancy, or repeated words, for instance ‘the’ and ‘be’ are the most frequently used English words. We use four lossless encoding techniques to compress data: static Huffman encoding, modified static Huffman encoding, dynamic Huffman encoding and modified dynamic Huffman encoding. The modified Huffman encoding works on encoding groups (pairs and triplets) of characters instead of encoding every character based on its’ frequency as in the traditional Huffman encoding schema. The compression ratio for each technique is calculated for the input data and compared based on the amount of redundancy the data might have.
1.2 MOTIVATION

Data compression, also called source coding is the process of encoding information in fewer number of bits than an un-coded representation, using various encoding schemes.

Files are usually compressed to save space, bytes, bandwidth etc. But our motive behind compression is that the compressed version would start to resemble a random file and lists of binary numbers can be encrypted in AES. The compression schema aims on reducing the redundancy\[^{[3]}\] of the input, enabling us to change the key less often. The usual way to compress English text is using Huffman encoding on the individual characters, that is, encoding each character based on its frequency of appearance. But for security purpose and to make use of the redundancy present in English language, we have to modify Huffman encoding so that it randomizes the frequencies of consecutive letter pairs, or triples or more tuples.
2. INPUT

For the purpose of this thesis, we only consider input with upper case characters in the range ‘A’ to ‘Z’. The code designed for this purpose, accepts any form of input, and changes it into this format.

For the static Huffman encoding algorithm, single letter frequencies\textsuperscript{[10]} and letter pair frequencies\textsuperscript{[12]} are obtained from standard data available on the web. Data for the triple letter frequencies is not available on the web, hence for the extended static Huffman encoding for triplets, we created the frequency chart using the frequencies obtained from the e-book “The adventures of Tom Sawyer” by Mark Twain.
3. THE COMPRESSION TECHNIQUES

Data compression is the process of encoding information in fewer number of bits than an uncompressed representation, using various encoding schemes. It is the practice of reducing data quantity without compromising on the quality. It reduces the number of bits required to store and transfer the data, hence making storage of large data feasible.

Compression can either be lossy or lossless. Lossy compression reduces a file by permanently eliminating certain information, especially redundant information. On uncompressing the file, only a part of the original information is retrieved. Whereas, with lossless compression, every single bit of data that was originally in the file remains after the file is uncompressed. In this thesis, we will be covering four lossless compression schemas namely, Static Huffman encoding, Extended Static Huffman encoding (for letter pairs and triplets), Dynamic...
Huffman encoding and, Extended Dynamic Huffman encoding (for letter pairs and triplets).

**Huffman Encoding**

Huffman encoding\(^{[11]}\) is an optimal prefix encoding method that is commonly used for lossless text compression. Prefix encoding refers to a set of words containing no word that is a prefix of another word of a set. The advantage of this technique is that we obtain a uniquely decodable code, such that each word is identifiable without the requirement of a special marker between words. The Huffman model generates a code that is the most optimal in the sense that the compression is best possible. It maps frequency of characters to a unique code. That is, the more common characters are mapped to fewer bits as compared to the lesser common ones.
3.1 STATIC HUFFMAN ENCODING

For the purpose of this thesis, we use the standard character frequency table \cite{10} in the English language. This would help in decoding of data without the need of any extra information from the sender. The complete compression algorithm is based on three steps: getting the character frequencies from the table, generating the prefix code and finally encoding each character with its respective code.

![Figure 1: Character frequency table](image-url)
After obtaining the character frequencies which is input in the program as a single array, the model works on creating a Huffman tree. The Huffman tree is a type of binary search tree.

The pseudo code for building the Huffman tree is as follows:

1. For each alphabet(a) ranging from A-Z:
   a. If frequency is not zero, create a new node t, such that weight(t) is the character frequency and label(t) is the character.

2. List all the created nodes in the increasing order of their weight.
   We used a priority queue for this operation.

3. Extract two nodes with the smallest weight from the above list.
   Do the following till the above list has elements:
   a. Create a new node n with weight as the sum of the above two nodes.
   b. Add the lower frequency node as the left value of n and the other node as the right value of n.
   c. Insert this new node to the above list.
4. Return the root of this tree.

**Encoding:**

1. We take in the first letter from the input text. For example, if the given text is ‘CAT’, the program takes ‘C’ as input first. The following steps are repeated, till the program encounters end of the input string.

2. We start at the root of the previously created Huffman tree, and check the left and the right branches.
   
   a. If the input character is a leaf node, and is on the left branch, add ‘0’ to the output string, if it’s on the right branch, add ‘1’ to the output string. Move to the next input character from the input string.
   
   b. Else, move to the left branch (the left sub tree) and add ‘0’ to the output string.
   
   c. Repeat steps a through c, till we reach the the required leaf node.
Example working of the Huffman encoding process:

INPUT (x): “THECATHASTHEBAT”

Length of x: 15 * 8 = 120 bits

Frequency table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.12</td>
<td>1.49</td>
<td>2.71</td>
<td>12.02</td>
<td>5.92</td>
<td>6.28</td>
<td>9.1</td>
</tr>
</tbody>
</table>

Diagram:

```
  8  1  3  12  6  9  6
A  B  C  E  H  T  S
```

```
  8  4  12  6  9  6
A  E  H  T  S

  1  3
B  C
```
Fig 2 Huffman tree for static single letter encoding
Decoding:

1. We start at the root of the previously created Huffman tree and start reading the binary input (the previously compressed data) For example: “1010010100”

2. If the character is ‘0’ we move towards the left branch and, and towards the right branch if the character encountered is ‘1’.

3. Repeat step 2, till we reach a leaf node.

4. We replace the whole string of binary characters used in the path with the letter of the leaf node.
5. We move towards the next binary character from the input string.

   The whole process is repeated till we reach the end of the input string.
3.2 EXTENDED STATIC HUFFMAN ENCODING

3.2.1 LETTER PAIRS

Extended Huffman compression for letter pairs deals with compression of data, taking into consideration the frequency for pairs of characters. The advantage of this technique is that it effectively takes into consideration the redundancy in the English language. For example, In the English language, the words ‘to’, ‘be’, ‘of’ are the most frequently used\[9\]. With single letter Huffman encoding schema, we encoded each letter according to the probability of it’s occurrence, but with double letter Huffman encoding schema, we consider the frequency of every possible pair of letters. Using these frequencies, we create 26 different Huffman trees.

The pseudo code for generating these trees is very similar to the single letter Huffman encoding scheme.
Input: A 26*27 matrix, with each matrix[0][0] representing the frequency of A-A, and matrix[25][25] representing the frequency of Z-Z. The last element in each row is -1, placed as an error check.

Output: 26 Huffman trees

1. For each alphabet(a) ranging from A-Z, create a Huffman tree:
   a. We start with 26 leaf nodes, each depicting its frequency with respect to the previously chosen character. For example, if we are creating a tree for letter ‘B’ then the node with label ‘A’ would depict the frequency of letter ‘A’ in the English language, given letter ‘B’ precedes it.
   b. Place all these nodes in a priority queue, in increasing order of their frequency values.
   c. Extract the first two nodes with the smallest weight from the above list. Do the following till the above list has elements:
      d. Create a new node n with weight as the sum of the above two nodes.
e. Add the lower frequency node as the left value of n and the other node as the right value of n.

f. Insert this new node to the above list.

g. Return the root of this tree.

Fig 3 Huffman tree for letter A
Encoding:

1. The first letter of the input string is encoded using single letter Huffman encoding schema.

2. We now take the first letter of the input string as a reference for the Huffman tree to look into, and search for the second character’s code, based on the tree obtained. For example: If our input string is “CAT” we encode ‘C’ using single simple Huffman encoding, and then consider the pair ‘CA’. In this case we search the Huffman[2] tree or the Huffman tree for letter ‘C’ and search for the encoding of A in it.

3. This process is repeated until the entire string is encoded.

Example working of the Extended Static Huffman (Pairs) encoding process:

INPUT (x): “THECATHASTHEBAT”

Length of x: 15 * 8 = 120 bits
Frequency table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>20</td>
<td>33</td>
<td>0</td>
<td>5</td>
<td>95</td>
<td>104</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>47</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>31</td>
<td>0</td>
<td>9</td>
<td>38</td>
<td>38</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>110</td>
<td>23</td>
<td>45</td>
<td>48</td>
<td>33</td>
<td>115</td>
<td>83</td>
</tr>
<tr>
<td>H</td>
<td>114</td>
<td>2</td>
<td>2</td>
<td>302</td>
<td>6</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>S</td>
<td>67</td>
<td>11</td>
<td>17</td>
<td>74</td>
<td>50</td>
<td>43</td>
<td>109</td>
</tr>
<tr>
<td>T</td>
<td>59</td>
<td>10</td>
<td>11</td>
<td>75</td>
<td>330</td>
<td>34</td>
<td>56</td>
</tr>
</tbody>
</table>

Based on the same strategy as for the single Huffman encoding, we create seven trees. Huffman tree for letter ‘A’ is illustrated below.
Fig 4 Huffman tree for letter ‘A’
We encode the first letter from the previous example’s T’s code: ‘01’

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10000</td>
<td>1001</td>
<td>101</td>
<td>Null</td>
<td>10001</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
<td>0010</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>000</td>
<td>0011</td>
</tr>
<tr>
<td>C</td>
<td>111</td>
<td>Null</td>
<td>11001</td>
<td>0</td>
<td>10</td>
<td>11000</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>11110</td>
<td>1110</td>
<td>110</td>
<td>11111</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>H</td>
<td>01</td>
<td>000100</td>
<td>000101</td>
<td>1</td>
<td>0000</td>
<td>00011</td>
<td>001</td>
</tr>
<tr>
<td>S</td>
<td>110</td>
<td>11100</td>
<td>11101</td>
<td>10</td>
<td>1110</td>
<td>11111</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>010</td>
<td>011000</td>
<td>011001</td>
<td>00</td>
<td>1</td>
<td>01101</td>
<td>0111</td>
</tr>
</tbody>
</table>

Encoded string: 01111110111010111011111110010

Length of encoded string: 30 bits

Space saving percentile: 75.00%
Decoding:

1. We start at the root of the previously created Huffman tree and start reading the binary input (the previously compressed data) For example: “1010010100”. We decode the first character with the help of previously stated decoding schema in the single letter Huffman encoding method.

2. Now we refer the tree of the character obtained in the previous step. This character is added to the output string.

3. We search for the next possible substring from the remaining part of the input string to obtain a valid character as a leaf node.

4. We replace the whole string of binary characters used in the path with the letter of the leaf node.

5. Repeat from step two to step four until we reach the end of the input string.
3.2.2 TRIPLETS

On having worked on extending the Huffman encoding to character pairs we noticed that, more than 25 percent of the top 100 most frequent words\textsuperscript{[2]} in English language are three lettered. The most frequent word in the language being “the” and the fifth being “and”. Hence, we worked on extending the Huffman encoding to triplets. Huffman encoding for triplets works on generating 26*26 different trees, while considering the frequency of every three possible character combination in the English language. In triple letter Huffman encoding each tree would represent a pair of letters (A-Z) and each node of the tree would represent the frequency of the tree name plus the node character together. For example, the node ‘E’ in the Huffman tree for pair ‘TH’ would represent the frequency of ‘THE’ in the English language.

The frequency has been pre-obtained and input to the program in the form of 26*27*27 matrix. The frequency has been obtained by forming
triplets and noting their frequency from the English book “The Adventures of Tom Sawyer” by Mark Twain.

The algorithm for creating Huffman trees as per extended Huffman encoding for triplets is as follows:

1. For each alphabet(a) pair ranging from AA-ZZ, create a Huffman tree, that is a total of 676 trees:
   
a. We start with 26 leaf nodes, each depicting its frequency with respect to the previously chosen character pair. For example, if we are creating a tree for the pair ‘AB’ then the node with label ‘A’ would depict the frequency of letter ‘A’ in the English language, given letters ‘AB’ precedes it.
   
b. Place all these nodes in a priority queue, in increasing order of their frequency values.
   
c. Extract the first two nodes with the smallest weight from the above list. Do the following till the above list has elements:
   
d. Create a new node n with weight as the sum of the above two nodes.
e. Add the lower frequency node as the left value of n and the other node as the right value of n.

f. Insert this new node to the above list.

g. Return the root of this tree.

2. We encode the first character using the character – prefix map obtained from the single static Huffman encoding, followed by the first two characters being encoded using the character-prefix map obtained from the static Huffman encoding for pairs.

3. We now proceed to finding all the triplets in the input sentence and replacing each set with its respective prefix string. For example in the input sentence “HELLO” the triplets will be: “HEL”, ”ELL”, “LLO”. The final string of all these prefixes is then returned as the encoded output of the input string.
Example working of the Extended Static Huffman (Triplets) encoding process:

INPUT (x): “THECATHASTHEBAT”

Length of x: 15 * 8 = 120 bits

Frequency table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>281</td>
<td>76</td>
<td>58</td>
<td>119</td>
<td>207</td>
<td>283</td>
<td>546</td>
</tr>
<tr>
<td>AT</td>
<td>157</td>
<td>63</td>
<td>167</td>
<td>372</td>
<td>346</td>
<td>307</td>
<td>411</td>
</tr>
<tr>
<td>BA</td>
<td>0</td>
<td>10</td>
<td>144</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>25</td>
</tr>
<tr>
<td>CA</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>36</td>
<td>72</td>
</tr>
<tr>
<td>EB</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>227</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>EC</td>
<td>219</td>
<td>0</td>
<td>1</td>
<td>53</td>
<td>94</td>
<td>4</td>
<td>150</td>
</tr>
<tr>
<td>HA</td>
<td>0</td>
<td>33</td>
<td>25</td>
<td>0</td>
<td>8</td>
<td>95</td>
<td>1395</td>
</tr>
<tr>
<td>HE</td>
<td>462</td>
<td>383</td>
<td>406</td>
<td>152</td>
<td>350</td>
<td>806</td>
<td>365</td>
</tr>
<tr>
<td>TH</td>
<td>1467</td>
<td>27</td>
<td>22</td>
<td>6189</td>
<td>116</td>
<td>70</td>
<td>138</td>
</tr>
<tr>
<td>ST</td>
<td>429</td>
<td>47</td>
<td>17</td>
<td>308</td>
<td>611</td>
<td>82</td>
<td>138</td>
</tr>
</tbody>
</table>
Fig 5 Huffman tree for letter ‘AS’
Prefix table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>110</td>
<td>11111</td>
<td>11110</td>
<td>1110</td>
<td>1110</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>AT</td>
<td>11111</td>
<td>11110</td>
<td>1110</td>
<td>10</td>
<td>110</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>BA</td>
<td>Null</td>
<td>000</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>CA</td>
<td>Null</td>
<td>Null</td>
<td>000</td>
<td>001</td>
<td>Null</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>EB</td>
<td>01</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>00</td>
<td>Null</td>
</tr>
<tr>
<td>EC</td>
<td>0</td>
<td>Null</td>
<td>11000</td>
<td>1101</td>
<td>110</td>
<td>11001</td>
<td>10</td>
</tr>
<tr>
<td>HA</td>
<td>Null</td>
<td>001</td>
<td>0011</td>
<td>Null</td>
<td>0010</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>HE</td>
<td>10</td>
<td>1110</td>
<td>110</td>
<td>11110</td>
<td>11111</td>
<td>0</td>
<td>1110</td>
</tr>
<tr>
<td>TH</td>
<td>01</td>
<td>001101</td>
<td>001100</td>
<td>1</td>
<td>0010</td>
<td>00111</td>
<td>000</td>
</tr>
<tr>
<td>ST</td>
<td>10</td>
<td>110101</td>
<td>110100</td>
<td>111</td>
<td>0</td>
<td>11011</td>
<td>1100</td>
</tr>
</tbody>
</table>

Encrypting ‘T’ as obtained from static Huffman encoding tree for single character: 01
Encrypting ‘TH’ as obtained from static Huffman encoding tree for
character pairs: 1

Encoded string: 01111001110010100111100101
Length of encoded string: 27 bits
Space saving percentile: 77.5%

Decoding:

1. For decoding the input binary string, we start at the root of the
   previously created Huffman tree and start reading the binary input
   (the previously compressed data) For example: “1010010100”. We
   decode the first character with the help of previously stated
   decoding schema in the single letter Huffman encoding method.
   We store the result as the first char of the output string.

2. Using the character obtained in the previous step we find the next
   character using the static Huffman decoding schema for character
   pairs. We store the result as the second character of the output
   string.
3. Now, using these two characters, we approach the tree of character first + second. In the obtained tree we find the next possible character, moving left for zero and right for one. Using this previous step, we get the first three characters of our decoded string.

4. Use the last two characters of the output string as reference for the tree to be considered to decode the next character. Add the obtained character to the output string.

5. Repeat step four, till we reach the end of the binary input string.

Return the decoded string.
3.3 DYNAMIC HUFFMAN ENCODING

Dynamic Huffman encoding works exactly like static Huffman encoding except, it doesn’t use a pre obtained frequency table. Dynamic Huffman encoding algorithm works on creating a frequency table ‘on the fly’ based on the input string. It has its own set of advantages and disadvantages when compared with static Huffman encoding algorithm. The output of Huffman’s algorithm is a variable length code table which is then used to encode the input string. The algorithm derives this table from the probability of frequency of letters which it generates from the input string.

As in other entropy encoding methods the more frequent or common characters are represented using fewer bits as compared to those which are comparatively less frequent. The Huffman algorithm can be implemented in linear time if the character frequencies are sorted. For this purpose, our program places the obtained frequencies in a priority queue where they are available in an ascending order.
The dynamic Huffman algorithm is a three step process:

1. Create a frequency table based on the input string.

2. Using the above created frequency table, create a tree for all the characters involved in the given input.

3. Using the tree create a map relating each character to its prefix.

4. Using the map, encode the input string, into a binary string.

5. While communicating this text with another person or group, the map too has to be communicated, we plan on doing so using AES encryption.

Creating the Huffman tree:

1. Read through the entire string to create a frequency table, where each index represents the ASCII value of it’s character and the array value represents its frequency.

2. This frequency table is then used to create a Huffman tree, each index from out frequency table is used to create a leaf node, which contains the character value and its frequency.
3. These leaves are then put into a priority queue, which on each poll gives us the current lowest frequency leaf.

4. We start with the first two lowest frequency leaves and build a Huffman tree, with its root representing a null value node, with frequency equivalent to the sum of the two input nodes.

5. We continue doing this till out queue becomes empty.

**Encode:**

1. We start at the root of the previously created Huffman tree, and check the left and the right branches.

   a. If the input character is a leaf node, and is on the left branch, add ‘0’ to the output string, if it’s on the right branch, add ‘1’ to the output string. Move to the next input character from the input string.

   b. Else, move to the left branch (the left sub tree) and add ‘0’ to the output string.

   c. Repeat step 2, till we reach the the required leaf node.
d. Each time we reach our required character, we add it to our map, along with its prefix.

**Example working of the Huffman encoding process:**

INPUT (x): “THECATHASTHEBAT”

Length of x: 15 * 8 = 120 bits

Creating the frequency table based on input string:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

```
1 1 1 2 3 3 4
B S C E A H T
```
Fig 6 Huffman tree for static single letter encoding
Prefix table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>11110</td>
<td>11110</td>
<td>1110</td>
<td>10</td>
<td>11111</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoded string: 01011101111011001011011111011011101111101100

Length of encoded string: 44 bits

Space saving percentile: 63.33%

Decoding:

1. We are already given the tree for the process of decoding. The tree is initially encrypted, and should be decrypted using AES.

2. We then start at the root of the Huffman tree and start reading the binary input (the previously compressed data). For example: “1010010100”

3. If the character is ‘0’ we move towards the left branch and, towards the right branch if the character encountered is ‘1’.
4. Repeat step 2, till we reach a leaf node.

5. We replace the whole string of binary characters used in the path with the letter of the leaf node.

6. We then move our initial pointer to the end of the previously encountered binary string, as encountered in the previous step.

7. The whole process is repeated till we reach the end of the input string.
3.4 EXTENDED DYNAMIC HUFFMAN ENCODING

3.4.1 LETTER PAIRS

Extended dynamic Huffman encoding for character pairs works on creating 26 Huffman trees. That is each letter has its own Huffman tree, where the leaves of a tree represent the frequency of the pair tree value and leaf value. We assume that if a particular pair is the only leaf in a tree, we consider its binary prefix equivalent to be ‘1’. That is if ‘E’ is the only leaf in the Huffman tree for ‘Q’ then binary prefix for ‘QE’ will be ‘1’. This will enable us to decode the binary string more efficiently and be able to represent each character pair in the encoded string. When there is a single leaf in a tree, it cannot actually create a Huffman tree, and hence the modification.
**Encoding:**

1. Create a 26*26 matrix and traverse through the input string. For each character pair “XY” go to array index for ‘X’ and increment the array index of ‘Y’.

2. Using the above 26 by 26 matrix, for each row of the matrix create a Huffman tree using its column values.

3. The Huffman trees for each row can be created using the same method as described in the dynamic Huffman encoding method.

4. The above obtained 26 Huffman trees depict frequencies for pairs contained in our input string, ranging from ‘AA’ to ‘ZZ’. Using these Huffman trees, create an array of maps.

5. The index of the array is the character value of the first character of an input pair, and the second character of the input pair can be searched for in the map obtained. The resultant prefix would describe the encoding of the input character pair.

6. Return the maps’ array.
7. Now encode the first character as the prefix of the character obtained in the dynamic Huffman encoding for single character.

8. The rest of the pairs are replaced as per their values from the map.

Example working of the Huffman encoding process:

INPUT (x): “THECATHASTHEBAT”

Length of x: 15 * 8 = 120 bits

Creating the frequency table based on input string:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Fig 7 Huffman tree for letter ‘A’

Fig 8 Huffman tree for letter ‘E’
Prefix table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>E</td>
<td>Null</td>
<td>0</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>S</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

Encrypting ‘T’ as obtained from static Huffman encoding tree for single character: 0

Encoded string: 01111110011111

Length of encoded string: 14 bits

Space saving percentile: 88.33%
Decoding:

1. For the process of decoding, we are provided with 26 Huffman trees, each representing a different letter in the range A-Z.

2. We decode the first prefix using the Huffman tree obtained from the algorithm dynamic Huffman encoding for single letter encoding.

3. We keep a pointer at the beginning of the input binary string initially. One pointer is kept at the root of the Huffman tree from step 2.

4. We keep moving the pointer forward in the input string, until we find a match. For each binary character from our input string, we move left if we encounter a zero and right if we encounter a one. Once we reach a leaf node, we return that leaf value, which would be a character ranging between A-Z. Print this character.

5. Using the character obtained from the previous step, we move to that character’s Huffman tree, and follow steps three and four till we reach the end of our input string.
3.4.2 TRIPLETS

Extended dynamic Huffman encoding for groups of three letters works on creating 576 Huffman trees. Each tree leaf reflects the frequency of characters ‘A’ - ‘Z’ from the tree value. The tree value reflects character pairs in the range ‘AA’ - ‘ZZ’ and their respective frequencies. Hence, each character pair has its own Huffman tree. The output of this code is a 26*26 matrix representing the 576 frequency tables which contain the binary string for each character triplet.

If any of the Huffman trees has a single node, then the binary code for that triplet is considered as ‘1’ to avoid perplexity.

Encoding:

1. For encoding the input string, we first encode the first character through the dynamic Huffman encoding and the first two characters using the dynamic Huffman encoding for character pairs algorithms.
2. We now build a $26 \times 26 \times 26$ matrix and store the frequency of every three-character group in the string in it.

3. Using the above created matrix, create a Huffman tree for each character pair, as illustrated in the dynamic Huffman encoding for single character.

4. For each of the character pair trees, create a matrix of prefix tables, which maps each three-character group to its respective binary code.

5. Character pairs, having just one node to form triplet, are encoded as ‘1’. For example, if for character pair ‘AB’, the only triplet group that exists is ‘ABC’ then ABC is encoded as ‘1’.

6. Using the prefix tables obtained in the previous step, encode each three letter group with its respective binary code.

7. Return the binary encoded string.
Example working of the Huffman encoding process:

INPUT (x): “THECATHASTHEBAT”

Length of x: 15 * 8 = 120 bits

Creating the frequency matrix based on input string:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>AT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EB</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EC</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>HE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TH</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Fig 9 Huffman tree for pair ‘HE’

Fig 10 Huffman tree for pair ‘TH’
Prefix table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>H</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
</tr>
<tr>
<td>AT</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>BA</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
</tr>
<tr>
<td>CA</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
</tr>
<tr>
<td>EB</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>EC</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>HA</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
</tr>
<tr>
<td>HE</td>
<td>Null</td>
<td>0</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>TH</td>
<td>0</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>ST</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>1</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

Encrypting ‘T’ as obtained from static Huffman encoding tree for single character: 0
Encrypting ‘TH’ as obtained from static Huffman encoding tree for character pairs: 1

Encoded string: 01111101111011

Length of encoded string: 15 bits

Space saving percentile: 87.5%

Decoding:

1. In the beginning we are provided with Huffman trees in the range 0-576, each representing a character pair in the range “AA-ZZ”, where the leaf of each tree represents the node’s value’s distance from the tree value.

2. We encode the first character using dynamic Huffman encoding and the first two characters using Huffman encoding for character pairs. Print these values and add the characters to the output string.
3. Now, using the two alphabets obtained in the previous step, access the tree with the same same value. Define a pointer ‘p’ at the beginning of the input string.

4. Move left on encountering ‘0’ and right on encountering ‘1’.
   Increment the pointer. Continue doing this till we reach a leaf node.
   Print the value of the leaf node.

5. Using, the last two characters of the output string, access the tree with the same value.

6. Repeat steps four and five till ‘p’ reaches the end of input string.
4. RESULTS

**Input:** The Quick brown fox jumps over the lazy dog.

**Edited:** THEQUICKBROWNFOXJUMPSTHELAZYDOG

<table>
<thead>
<tr>
<th></th>
<th>STATIC</th>
<th>DYNAMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE</td>
<td>37.14%</td>
<td>42.86%</td>
</tr>
<tr>
<td>PAIRS</td>
<td>45.71%</td>
<td>84.28%</td>
</tr>
<tr>
<td>TRIPLETS</td>
<td>61.07%</td>
<td>86.78%</td>
</tr>
</tbody>
</table>

**Input:** how I need a drink alcoholic of course after the heavy chapters involving quantum mechanics.

**Edited:** HOWINEEDADRINKALCOHOLICOFCOURSEAFTERTHEHEAVYCHAPTERSINVOLVINGQUANTUMMECHANICS

<table>
<thead>
<tr>
<th></th>
<th>STATIC</th>
<th>DYNAMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE</td>
<td>45.61%</td>
<td>47.56%</td>
</tr>
<tr>
<td></td>
<td>PAIRS</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>51.94%</td>
<td>77.60%</td>
</tr>
</tbody>
</table>

**Input:** resume.txt

<table>
<thead>
<tr>
<th></th>
<th>STATIC</th>
<th>DYNAMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE</td>
<td>46.50%</td>
<td>47.19%</td>
</tr>
<tr>
<td>PAIRS</td>
<td>51.77%</td>
<td>55.56%</td>
</tr>
<tr>
<td>TRIPLETS</td>
<td>54.47%</td>
<td>72.70%</td>
</tr>
</tbody>
</table>
5. CONCLUSION AND FUTURE WORK

The results obtained were surprisingly good, and in most cases for the given input we noted a 20-30% increased compression for modified dynamic and a 10-15% increased compression for modified static Huffman encoding over the pre-existing techniques. This was a lot better than expected.

We would like to conclude by saying extended dynamic Huffman encoding would work better if the input varied a good deal.

Future work, involves extending the dynamic and static modified Huffman encoding algorithm to quadruplets and further till we cannot compress the data any further. The input too could be widened to include small case letters and numbers. Currently, the compression concentrates only on text files[8], this could be extended to include images and videos.
6. REFERENCES


