An Investigation into the Text Compression Capabilities of the Huffman Coding Algorithm

Raphael Bijaoui
rdb4017@ic.ac.uk

Abstract

Huffman Coding has long served as the core of modern compression techniques across the globe. In this project, the Huffman Coding Algorithm is explained, and a possible implementation is expounded upon. After forming a basis on the relevant functions, we test the newly created compression system against three distinctive types of text. Results were then analysed, leading us to the conclusion that the implemented Huffman Coding algorithm is very good for regular written messages, but less reliable for text containing a large amount of ASCII characters.

Introduction

Motivation

To understand our main motivation, we begin with a definition. Data compression is defined as "the process of reducing the amount of data needed for the storage or transmission of a given piece of information, typically by the use of encoding techniques" (Hemmendinger, 2013)

One of the earliest forms of data compression was in the creation of the telegraph, which communicated messages across large distances, within an instant, with a collection of electrical signals representing "codewords" to be deciphered by the receiver, using a table of translations. To improve the speed of sending a message across a channel, Morse Code was created, which took advantage of the fact that the letters used in the English language have varying frequency. As a result, common letters such as "e" and "t" had shorter codewords to improve communication time. (Wolfram, 2013).

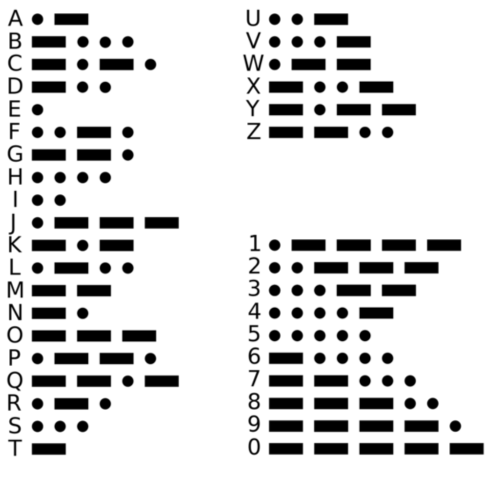


Figure 1: Example of Morse Code table. Note codewords of e and t.

Over time, data compression has evolved to span two main branches- lossy and lossless. A lossy compression reduces a file by removing non-essential information for the intended user and is non-reversible, whereas a lossless compression preserves all information and is fully reversible.

This paper will be focusing on a fundamental algorithm underpinning lossless data compression today- Huffman Coding.

Proposed in 1951 by David Huffman, the algorithm demonstrates a way of breaking down data into a set of unique characters and sorting them as nodes in a binary tree based on their frequency. Since inception, Huffman coding has served as the core of the compression industry, featuring in modern mainstream compression formats such as GZIP, PKZIP, and BZIP2.

This paper will thus simulate and analyse the full transmission (encoding, decoding) of textual data using Huffman encoding in the hopes of better understanding a key component in the world of information theory today.

General Overview

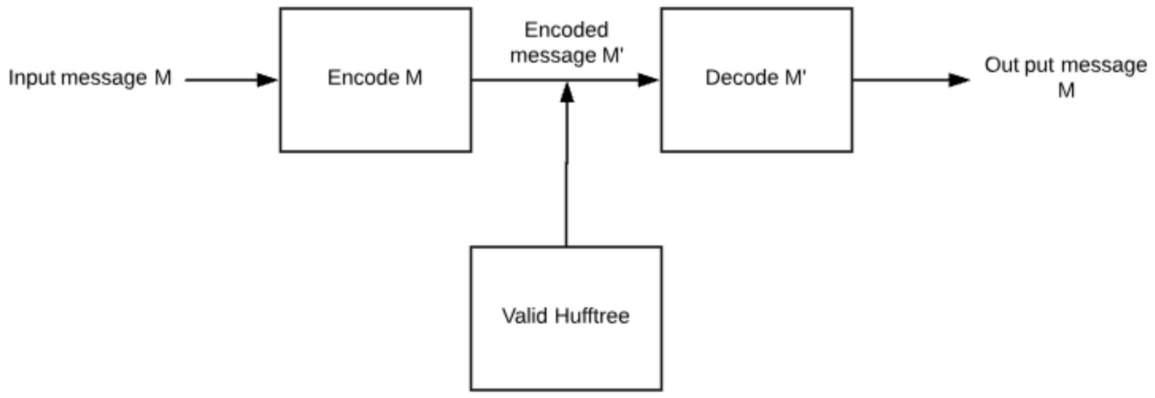


Figure 2: An overview of the compression system to be implemented

The basis of Huffman Coding lies in the creation of a Huffman Tree, which is a special Binary tree that stores the unique characters of a message in leaf nodes, along with their occurrence frequency within the message. Non-leaf nodes are used to hold the sum of the occurrence frequencies of its two subtrees.

The path taken from the root of the Huffman tree to each leaf node is then generated and stored inside a Huffman Code table, which holds each unique character for the input message *M* and it’s respective Huffman code.

The “Valid Hufftree” module is a utility function to check there were no errors in the tree creation process. It traverses all nodes of the Huffman Tree, checking that the nodes meet Huffman Tree conditions.

Finally, we decode the message at the receiver by iterating through all characters the encoded message *M'* , connecting each string of characters to the codewords in the Huffman Code table defined earlier. The resulting output is our original message *M.*

Related Work
There have been several works in the area of Binary Trees that has been covered, and this foundation will prove useful as the Huffman Tree is generated to match the earlier specifications. An important distinction from our previous work is that here, we will be working with an unsorted, unbalanced binary tree that has the unique requirement that each non-leaf nodes has two child nodes.

Previous work on tree traversals may prove useful in this context, as we seek to create a decoding table based on the Huffman Tree generated. Thus, we may apply a level-order traversal in the form of a Breadth-First Search, or an in-order traversal as explored recursively, or with the application of Depth-First Search for our path-finding function.

To generate and make use of a table within the project, a map data structure would be ideal. This is due to its key, value pairing system that will allow us to pair each unique character with its frequency, respectively. Furthermore, since std::map is normally implemented as a self-balancing binary search tree in C++, its favourable O(logN) lookup complexity makes it scalable as we focus on larger messages later on (Gupta & Zaroliagis, 2014).

Lastly, a self-sorting container structure, that makes the two lowest frequency nodes available would help facilitate the Huffman Tree creation process, as nodes can be easily adjoined and inserted once more into the shrinking container.

For these reasons, the implementation of a map and priority queue would be useful additions to the current state-of-the-art as we seek to efficiently compress our message with Huffman Coding.

Describing algorithms
Encoding
We begin the implementation with the encoding algorithm, as was visualised with the first block of *Figure 2*.

Inputs
- Input message *M*, of type std::string
- Empty frequency table *F* of type std::map
- Empty Huffman Tree pointer, to eventually point to the root of Huffman tree *T*
- Empty Huffman Coding table *H*, of type std::map

Output
- Compressed message *M'* as a sequence of binary values of type std::string.

To achieve this output, we will use our inputs in five main steps as follows.
1. <u>Update *F*</u> to represent the unique characters of *M* and their occurrence frequencies
2. <u>Create and populate priority queue</u> *P* with leaf nodes of *T* using the key, value pairs of *F*.
3. <u>Build *T*</u> by:
    - Adjoining the two lowest frequency Huffman Tree nodes from *P*

    - Removing them
    - Inserting adjoined node into the priority queue
    - Repeating until only one node remains in the priority queue.
4. Update *H* to display the unique characters of *M* as the keys, and their respective paths from the root of *T* as the values.
5. Rewrite *M* into *M'* using *H* as a character encoding table.

Having established the high-level conceptual description, let us now delve into the implementation of each of these steps.

1. Update Frequency Table *F*: represented by function update_freq_table

Input message *M* is a string composed of a sequence of ASCII characters. For each ASCII character *C* in *M*, we search for it in frequency table *F*. If *C* is not present in *F,* we emplace a new key, value pair into table *F*, setting the key to *C*, and value (occurrence frequency of character) as 1. If *C* is present in *F,* we add one to the corresponding value at key *C*. Once the message is fully traversed, *F* will represent a decomposition of message *M* as a table of its unique characters and their corresponding occurrence frequencies within *M*.

Character	**Frequency**
h	2
u	3
f	6
m	1
a	2
n	2
‘ ‘	3
e	2
d	2
a	2
p	1

Figure 3: Frequency table for message “Huffman huffed and puffed”

Pseudocode

```
for all characters in message
    if frequency table contains character C as key
        Add one to element value. (C, f+1)
    else
        Add new element (C,1) in frequency table
    endif
end loop
```

Complexity

This function iterates through all characters in a message of size *n* once. Thus as *n* is increased, the number of operations of this function should increase accordingly, requiring $O(n)$ operations in the worst and best case scenario.

2. Create & Populate Priority Queue *P*

Create: priority queue condition defined by struct compare

A priority queue is a data structure for maintaining a set of elements, each with an associated value called a key (Cormen et al, 2009). It behaves as a minimum heap, which has conditions that must be declared in a struct upon instantiation. At the top of the priority queue should be our lowest frequency Huffman Tree node pointers, which will help us in identifying the two least-frequent nodes to merge together in stage 3.

Pseudocode

The condition stated in the struct for the insertion of a new node *N* in the priority queue *P*, which can also be thought of as a minimum heap tree, is as follows.

```
insert N as leaf in P
while N.frequency less than Parent.frequency do
        switch leaf node with parent node
endloop
```

This will leave the smallest frequency node at the root of the min-heap tree, which is used to implement the priority queue.

Complexity

If *N* is furthest from root, maximum number of switches will be equal to the height of the tree, which makes this constant in the worst-case. Hence it will require $O(1)$ operations.

Populate: represented by function populate_priority_queue

For each element *E* in frequency table *F*, we create a Huffman Tree node *N* with its character set to the key of *E*, and frequency set to value of *E*. Node *N* is inserted into the priority queue *P* and will represent a leaf node for the Huffman Tree to be created. Whilst adding all *N,* the priority queue automatically sorts these nodes in ascending order of frequency, with the pointers to Huffman tree nodes with lowest frequency at the top of the priority queue.

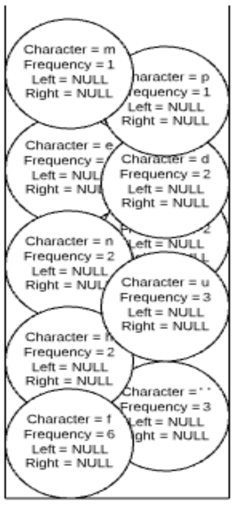


Figure 4: Priority queue after populate_priority_queue with message "Huffman huffed and puffed"

Pseudocode

```
for each E in F
        Allocate new node N
        N.character = E.key
        N.frequency = E.value
        N.left = NULL
        N.right = NULL
        Insert N in P
endloop
```

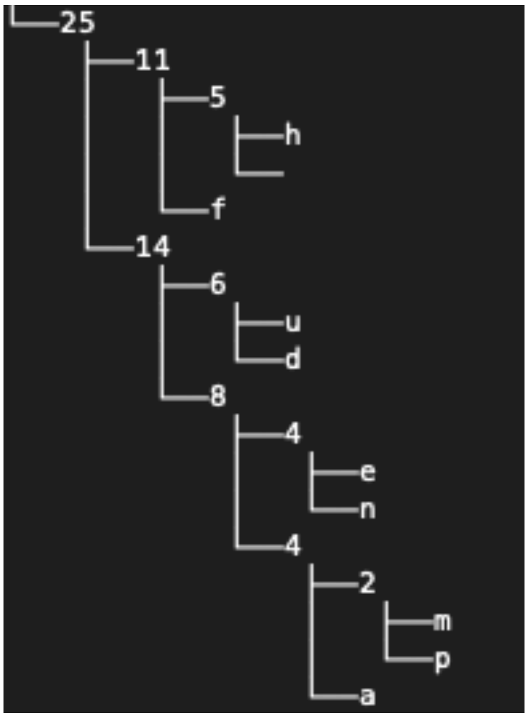
Complexity
The total number of ASCII characters is constant, hence the largest size of our frequency table *F* must also be constant. For that reason, this function will need $O(1)$ operations, as constant operations is a constant value.

3. **Build Huffman Tree *T*:** represented by function build_huffman_tree

   The top two node pointers in the priority queue *P* are temporarily stored and become the left and right child nodes of a new non-leaf Huffman tree node *H*, which holds a null character and a frequency consisting of the sum of the left and right child node frequencies. The top two node pointers are popped off the priority queue, and the new node *H* is inserted.

This process of popping off the top two nodes and inserting an adjoined node *H* is repeated until there remains only one node in the priority queue.

The last node in the queue represents the root to our new Huffman Tree *T*, which can be visualised as follows:



```
└──25
    ├──11
    │   ├──5
    │   │   ├──h
    │   │   └──
    │   └──f
    └──14
        ├──6
        │   ├──u
        │   └──d
        └──8
            ├──4
            │   ├──e
            │   └──n
            └──4
                ├──2
                │   ├──m
                │   └──p
                └──a
```

Figure 5: Example of Huffman Tree for "huffman huffed and puffed"

Pseudocode

```
BUILDTREE($T$)
while $P$.size is greater than 1
    allocate new node $N$
    $N.left$ = $l$ = extracted node from top of $P$
    $N.right$ = $r$ = extracted node from top of $P$
    $N.frequency = l.frequency + r.frequency$
    insert $N$ in priority queue
endloop
return extracted node from top of $P$
```

Complexity

With each iteration of the while loop, we remove two nodes from the priority queue and add one. This reduces the size of the priority queue by one each time. Furthermore, given that the maximum number of ASCII characters is constant, this also means that the maximum amount of nodes in our priority queue will be constant. This function therefore needs a constant number of operations in the worst case, giving it a complexity of $O(1)$.

4. **Update Huffman Coding Table *H*:** represented by function update_hufftable

Now that we have created the Huffman Tree *T*, we update the Huffman Coding Table *H*, which holds the same keys as the Frequency Table *F*, but has values equivalent to the Huffman Code instead of the occurrence frequency.

```
for all elements E in F
        emplace (E.key, “”) in H
endloop
```

Again, since the maximum number of elements *E* in *F* is constant, our worst-case complexity here will be constant with *O*(1) operations.

To find the path from root to leaf, we take the root node of *T* and traverse all nodes. Once we reach a leaf node, we lookup the character in our Huffman Code Table *H*, and set the value of this element to the path taken to reach the leaf node from the root.

Pseudocode

```
FINDCODE(node N, string str, hufftable H)
        if N is null
                return
        endif
        if N is a leaf node
        H[N.character].value = str
        endif
        FINDCODE (N.left, str+”0”, H); //traverse the left subtree
        FINDCODE (N.right,str+”1”,H); //traverse the right subtree
end
```

In the worst-case scenario, say we have 128 leaf nodes present in our Huffman tree. Since a Huffman Tree may be unbalanced, the largest height of the tree would therefore be 127, and might look something like this:

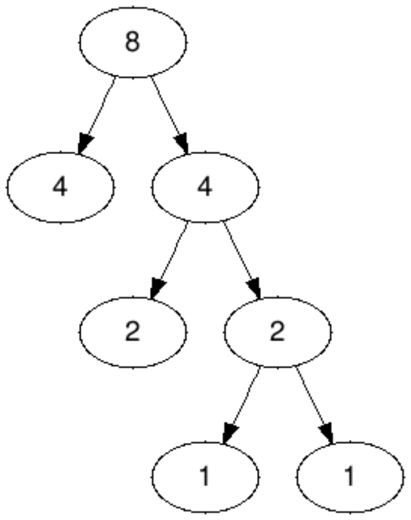


Figure 6: Unbalanced Huffman Tree (Lopes, 2016)

This would make the maximum number of nodes in our Huffman Tree equal to 2*128 – 1 which is a constant number independent of the size *n* of our message, giving a worst case complexity of *O*(1)

5. <u>Rewrite *M* into *M'*</u>: represented by function encode_message

Having created our Huffman Code Table *H,* we now iterate through each character *C* in message *M,* searching *H* for *C.* When the corresponding element is found, we append the value to encoded message *K.*

Pseudocode

```
for all C in M
        K = K + H[C].value ;
endloop
```

K now holds our fully Huffman encoded message.

For a message *M* of length *n*, we will complete *n* iterations as the function sequentially moves through the message. The lookup timing will be constant since the size of table *H* is limited to the number of valid ASCII characters. Hence, this function needs *O*(n) operations in the worst-case.

Valid Huffman Tree

Now that we have encoded the message, it is in our best interests to check there had been no errors in the encoding process. One good way of doing so is to check that we are working with a valid Huffman Tree. The conditions for a valid Huffman tree are stated as follows:

```
      all the terminal nodes (leaves) have characters,
      all the non-leaf nodes have two sub-trees each,
   and the occurrence frequency of a non-leaf node equals
the sum of the occurrence frequencies of its two sub-trees.
```

Figure 7: Specifications for valid Huffman tree (Ben Ismail, 2018)

Input

- Pointer *N* pointing to the root node of the Huffman tree in question.

Output

- Boolean *valid* set to true if it is a valid Huffman tree, and false for vice-versa.

This was implemented as follows.

Pseudocode

```
VALIDHUFFTREE(hufftree node)
    valid = true
    CHECKTREE(node,valid) //Recursively traverses all nodes. If rule broken, valid = false
    return valid
end

CHECKTREE (node,valid) //used for tree traversal
    if node != NULL
        CHECKTREE (node->left, valid)
        CHECKROOT(node, valid) //checks the root for all the rules of Huffman tree.
                              // If rule broken, "valid" set to false
        CHECKTREE (node->right, valid)
    endif
end
```

This function checks all nodes of our Huffman tree, and thus will have the same worst-case complexity as update_hufftable which is constant at $O(1)$.

Decoding Algorithm

Huffman Encoding's lossless compression means that by using the Message M' and Huffman Table *H* generated in the Encoding process, it is possible for us to fully retrieve our original message *M*. This algorithm sequentially processes *M'* checking if parts of the string *M'* map to a value in *H*. If it does, the key corresponding to that value is retrieved from *H* and added to the output message.

Inputs

- Encoded message *M'* in the form of a string of bits
- Pointer to the root of the Huffman tree *N* built during the encoding

Output

- The original message *M* as a string of characters

Pseudocode

```
string M = “”
t = N
if M’ is empty
    for i = 1 to t.frequency
        M = M + t.character
    endloop
else
    for i = 1 to M’.size
        if M’[t.character] == ‘0’
            t = t.left
        else
            t = t.right
        endif
        if t is a leaf node
            M = M + t.character
            t = N
        endif
    endloop
endif
return M
end
```

In the case that M’ is empty, we may have an input string such as “AAAA”, which would have only one node in the Huffman Tree. The function will iterate *t*.frequency times (in this case, 4), which is equivalent to the length *n* of the input message M.

If M’ is not empty, the first for loop will require *m* iterations, where *m* is the size of the encoded message. Since there is a limit on the number of possible characters, size *m* will be equal to k**n*, where k is a constant. For each lookup of the frequency table, we require *O*(log*a*) operations in the worst-case scenario, where *a* is the total number of possible ASCII characters (constant). The logarithm in place for the table lookup is due to the nature of C++ maps as self-balancing binary search trees, as referenced in ***Related Work***.

Thus, if *M’* is not empty, our decoding function will have a worst-case complexity of $O(k*n*\log a)$, which can be simplified to $O(n)$ in big-O notation as k log*a* is a constant value.

Overall Complexity

To find the overall complexity, we must sum the complexities of each function.

Encoding algorithm: $O(n+1+1+1+1) = O(n)$
Valid Huffman Tree: $O(1)$
Decoding algorithm: $O(n)$

Total complexity = $O(n+n+1) = O(2n+1) = O(n)$

Thus, we expect to see a linear increase in the complexity of our algorithm as *n* is increased.

Experimental Results

In this section, we shift our focus from how our compression system works internally, to now externally.

In the world of data compression systems, we are concerned with a few key metrics. Notably: running time (i.e. how long it takes to compress the data) and compression ratio (how large the compressed data is relative to the input data).

Testing Environment

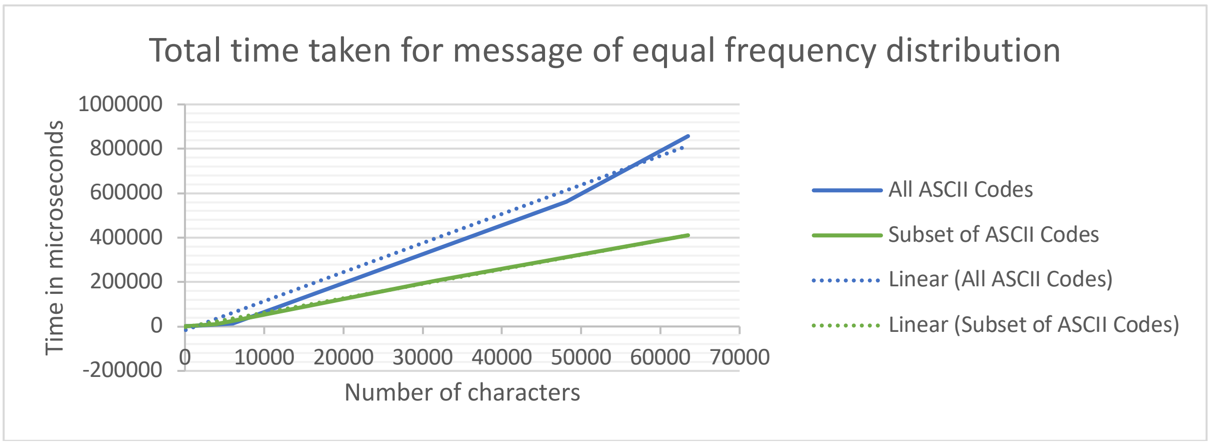
Since Huffman Coding improves the transmitted message size by working with the relative frequencies of characters in a text, it is important to test our compression system with text files that inherently have different relative frequencies for each character.

We will thus test our system's running time for the two following cases:

1) Equal occurrence frequency distribution
2) Variable occurrence frequency distribution

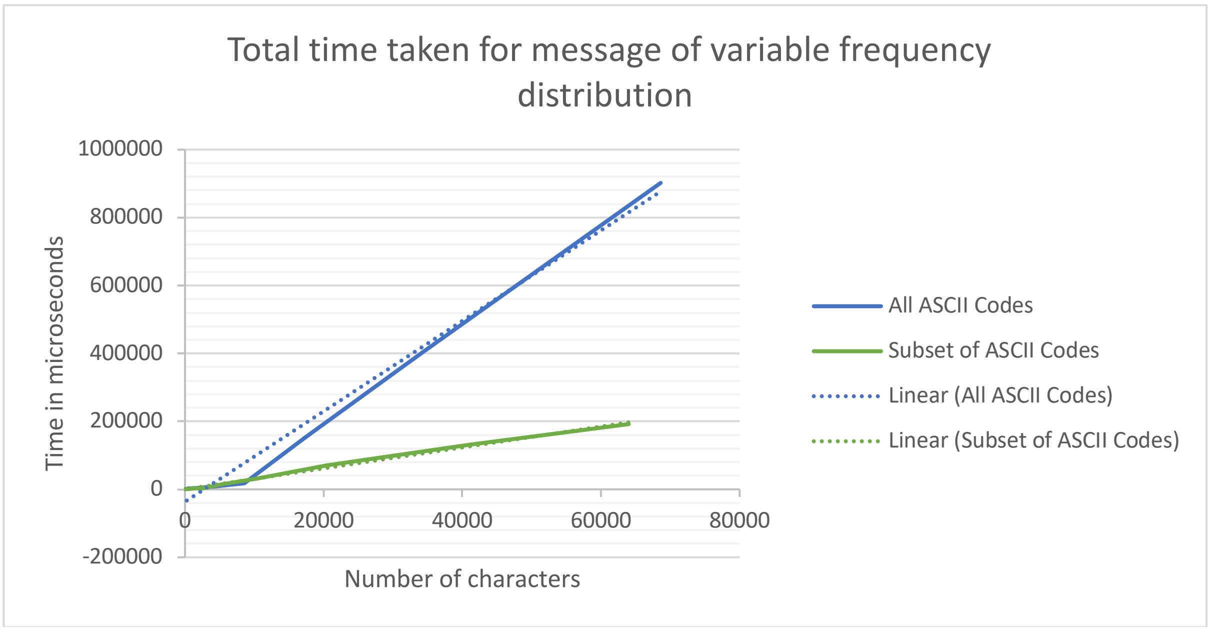
For each case we will test the system for all ASCII characters, and for a subset of all ASCII characters.

Case 1: Equal Occurrence Frequency Distribution



Type of message	Compression Ratio
All ASCII Codes	0.829787234
Subset of ASCII Codes	0.741935484

Case 2: Variable Occurrence frequency distribution

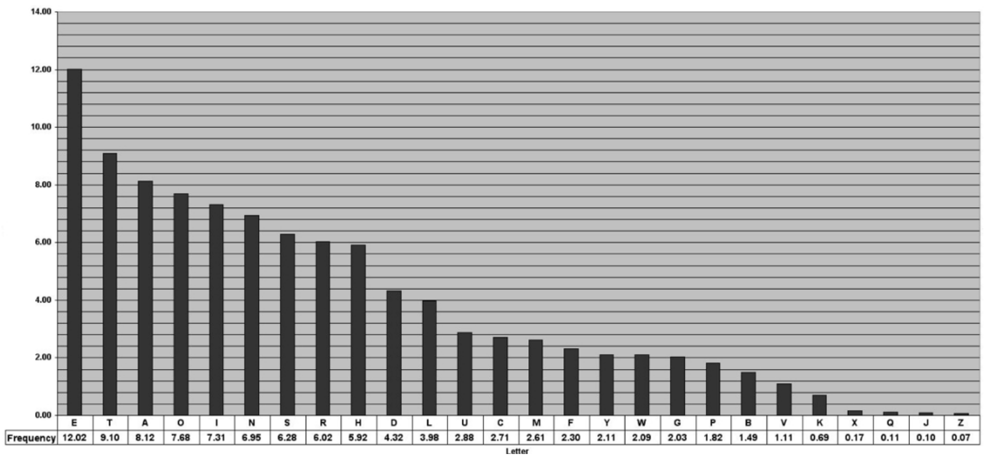


Type of message	Compression Ratio
All ASCII Codes	0.677238806
Subset of ASCII Codes	0.525390625

Having tested the running time and compression ratios of our compression system under different input conditions, we will now investigate how our compression system works for the most common case of text compression.

Compression Ratio for written English texts

Whilst written English consists of a variety of different words and meanings, when observed on a large-scale it begins to display key characteristics. Below is a chart showing how letter (occurrence) frequency varies from a sample of 40,000 words in a dictionary.



	E	T	A	O	I	N	S	R	H	D	L	U	C	M	F	Y	W	G	P	B	V	K	X	Q	J	Z
Frequency	12.02	9.10	8.12	7.68	7.31	6.95	6.28	6.02	5.92	4.32	3.98	2.88	2.71	2.61	2.30	2.11	2.09	2.03	1.82	1.49	1.11	0.69	0.17	0.11	0.10	0.07

Figure 8: Frequency of letters in the alphabet (MEC, 2003)

Therefore, should we compress a number of large texts, we should expect to see a similar compression ratio as the large texts begin to resemble one another.

Our test data must be large and resemble real texts. For that reason, popular English language books from *Project Gutenberg* were selected and processed as follows:

Text	**Compression Ratio**
Adventures of Sherlock Holmes	0.56444238
A Doll's House	0.59013
Metamorphosis	0.56042497
Great Expectations	0.57771573
The Republic	0.54445311
Average	**0.56743324**

From the table above, we see that for regular English text compression, our system displays about a 56.7% reduction in file size.

Evaluation

Our results show that as the number of characters in our input message grows, the running time of our system increases linearly. This held true for both cases of equal and variable frequency distribution, and it matches the worst-case complexity calculation calculated in ***Overall Complexity***, where we hypothesized our system to function in $O(n)$ time.

The compression ratio for our system works best for input messages of variable frequency that do not use all ASCII characters, as shown in the compression ratio for the subset of ASCII characters in case 2 (0.525) and for the large English text compression (0.567), which also exhibits a variable frequency distribution at a large scale and consists of a subset of ASCII codes. As we introduce more ASCII characters to our input message, the compression ratio worsens. This is because the size of our Huffman Tree increases, causing longer codewords for each of our characters and hence a larger compressed message.

It was also observed that the system is approximately 14% less performant when the occurrence frequencies of the system are equal to one another, compared to when they differ.

Conclusion

We may conclude that for input messages consisting of a smaller subset of ASCII characters and a range of character occurrence frequencies, Huffman's coding algorithm is very good. With regards to the task of text compression, our algorithm displays the lowest run time and compression ratio when messages use a subset of the available ASCII characters, and have variable occurrence frequencies. This results in a linear run time with a lower gradient, and a lower compression ratio (52.5% of original file size).

However, as the number of ASCII characters increases, and as character occurrence frequencies become more equal, the algorithm starts performing less optimally. One way we can improve the running time of the algorithm is by creating the table of codewords prior to compression. It may differ slightly from the optimal set of codewords, but it could significantly reduce our execution time as it directly translates our characters into codewords. This removes the need to iterate through the whole message in the encoding process, and also removes the need to generate the Frequency and Code Tables in real-time. This would be especially useful for the compression of larger English texts, which were shown to possess similar properties to one another.

We can implement this pre-determined table of codewords by making use of the letter frequencies in the English language as our frequency table values, and thus estimating the shape of our Huffman Tree for a large text. We can test this performance with more text files from *Project Gutenberg's* catalogue of over 58,000 text files, to give us not only a large range of test data, but also a good benchmark for character frequency.

Beyond text compression, this algorithm could be applied to any other file with data that can be broken down into a set of smaller components that repeat itself across a file. For example, we could apply this algorithm for image compression, setting pixels as our smallest unit, and their RGB values as their own "ASCII character".

In further explorations of text compression, it would be interesting to build upon the Huffman Coding Algorithm with an implementation of the Lempel-Ziv algorithm, which breaks our Huffman Code into smaller pieces, and then encodes the new message by having each piece refer to an earlier piece in the code.

It has been a fascinating exploration of text compression- one can only look forward to building up on this knowledge. But for now… time to "decompress"!

Works Cited

"Advanced Design and Analysis Techniques." *Introduction to Algorithms*, by Thomas H. et al Cormen, MIT Press, 2009, pp. 428–437.

Ben Ismail, Sahbi. *Text Compression Using Huffman's Coding Algorithm*. Imperial College London, 2019, bb.imperial.ac.uk/bbcswebdav/pid-1527033-dt-content-rid-5058689_1/courses/DSS-EE1_08-18_19/EE1-08_EE2-18_ADS_CW2_spring-2019.pdf.

"English Letter Frequency (Based on a Sample of 40,000 Words)." *Frequency Table*, pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html.

Gupta, Prosenjit, and Christos Zaroliagis. "Applied Algorithms - First International Conference, ICAA 2014, Kolkata, India, January 13-15, 2014. Proceedings Front Cover." *Google Books*, Google, 8 Jan. 2014, books.google.co.uk/books?id=JrG5BQAAQBAJ&dq=why%2Bstd%3A%3Amap%2Bis%2Bo%28logn%29&source=gbs_navlinks_s%2B%29.

Hemmendinger, David. "Data Compression." *Encyclopædia Britannica*, Encyclopædia Britannica, Inc., 23 Apr. 2013, www.britannica.com/technology/data-compression.

Juan Lopes. "Does Huffman Encoding Necessarily Result in a Balanced Binary Tree?" *Stack Overflow*, 2016, stackoverflow.com/questions/34990201/does-huffman-encoding-necessarily-result-in-a-balanced-binary-tree.

Wolfram, Stephen. "SOME HISTORICAL NOTES." *Historical Notes from Stephen Wolfram's A New Kind of Science*, 2013, www.wolframscience.com/reference/notes/1069b.