Time Complexity and Compression Rate Analysis of Huffman Coding with Predefined Codes

Wesly Giovano - 13520071[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520071@mahasiswa.itb.ac.id

***Abstract*—Huffman coding is widely used as the fundamental of many compression methods. This study aimed to explore a variety of Huffman coding which uses predefined codes instead of codes that depend on the original data to reduce the cost of building binary tree of Huffman codes. The predefined codes were obtained by building binary tree based on the frequency of each character in Brown Corpus. Simple implementations of the encoder and decoder of Huffman coding were made to analyzed the complexity of Huffman coding with predefined codes. The time complexity was found to be *O(n)* for both encoding and decoding algorithms. To analyze average compression rate of this variant, four sample texts with different lengths were analyzed, and the compression rate was found to be about 42 – 46%.**

***Keywords*—compression rate, Huffman coding, predefined codes, time complexity.**

I. Introduction

As data size is increasing rapidly with time, storage and transmission of data is getting more complex and more expensive. People sought to reduce the cost and time of storing and transmitting data, hence the notion of data compression was born.

Data compression is the process of reducing data size by mean of encoding it so that the resulting size in bits or bytes is smaller than the original data. The encoded representation must be decodable, may or may not be exact same with the original one, also known as lossless and lossy compression. Lossy compression can be used in data that can be presented similarly according to human senses such as image or audio, while lossless compression is commonly used for data that need to be presented exactly the same as original.

One of the simplest yet widely-used compression method is Huffman coding. As the name suggests, the method was found and developed by David A. Huffman in 1951. Huffman coding later became the basis of many other encoding and compression methods, such as deflate and JPEG compression. Huffman coding employs the idea of variable-length coding based on the original data so that common bit representations can be encoded into only few bits. Compression of Huffman coding uses constructed binary tree from original data as the basis of encoding. In this paper, the author would like to analyze the effect of changing the binary tree that is constructed for each data into a fixed binary tree, i.e., predefined codes.

II. Theoretical Framework

A. Tree

By definition, a *tree* is a connected undirected graph with no simple circuits [1]. It is named so because the graph resembles tree. In a tree, a simple path between any two of its vertices is always unique.

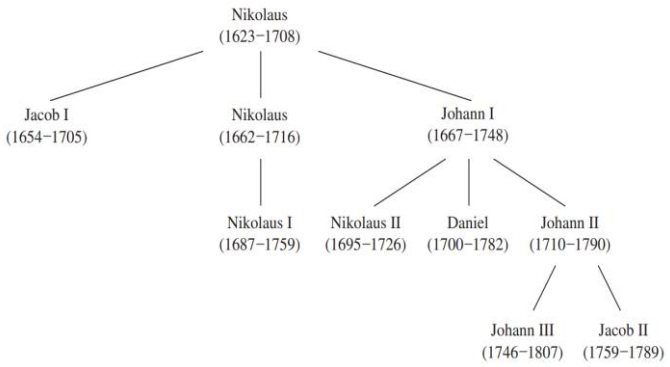

Fig. 1. Example of tree.
Source: K. H. Rosen, *Discrete Mathematics and Its Applications*.

A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root. If the children of each internal vertex are ordered, then the tree is an *ordered rooted tree*. A rooted tree is called an *m-ary tree* if every internal vertex has no more than m children. Hence, a tree with maximum of two children is a *binary tree*.

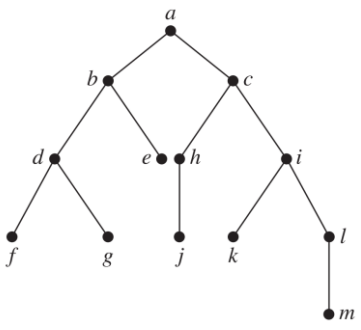

Fig. 2. Example of binary tree.
Source: K. H. Rosen, *Discrete Mathematics and Its Applications*.

B. Huffman Coding

Huffman coding is an algorithm to losslessly compress data by encoding the data into prefix code known as Huffman code. Prefix codes are codes that bit string of a code for a letter never occurs as the first part of the bit string for another letter. Huffman coding's algorithm takes frequency of each symbol in the original data as input, constructs a binary tree corresponding to the frequencies, and lastly encodes the original data into Huffman code. If the data is a representation of a string or text, then the input for Huffman coding is frequency of each character that appears in the original text.

The steps of constructing the binary tree are as follows [2]:

1. Pick two symbols with lowest probability or frequency, e.g. *A* and *B*. The two symbols then are combined into a binary tree with the root of the symbols' combination, e.g., *AB*, and children of the two symbols, e.g., *A* and *B*.

2. Pick next two symbols, including the combined symbols in step 1, and repeat the same procedure as step 1.

3. Label the binary tree formed consistently: left side with *0* and right side with *1*.

4. Labels that the path from the root to target leaf shows the prefix code for that symbol.

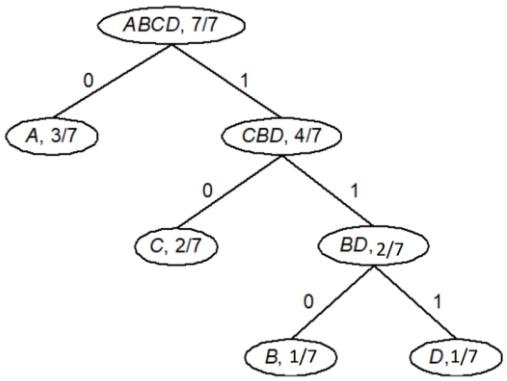

Fig. 3. Example of resulting binary tree in Huffman coding.
Source: R. Munir, lecture slide: Pohon, 2020.

Encoding the original data with Huffman coding is done by replacing each symbol with corresponding prefix code. For example, encoding the string "ABACCDA" with Fig. 3. as the binary results in the Huffman code "0110010101110". To decode the encoded data, read the binary code one by one while following the binary tree's corresponding edge. Once the leaf in the binary tree is reached, then the binary code up until the last one translates to the leaf's symbol.

0	1 1 0	0	1 0	1 0	1 1 1	0
A	B	A	C	C	D	A

Fig. 4. Decoding Huffman code "0110010101110" with binary tree in Fig. 3.

C. Algorithm and Complexity

An algorithm is a method for solving a class of problems on a computer. The complexity of an algorithm is the cost, measured in running time, or storage, or whatever units are relevant, of using the algorithm to solve one of those problems [3]. Hence, there are two kinds of complexity: time complexity, denoted as $T(n)$, and space complexity, denoted as $S(n)$ with n as the data size.

An algorithm typically consists of many different operations, such as input/output, arithmetic operation, assignment, comparison, and function calls. In calculating its time complexity, we only concern some of the operations and omitting the others [4]. For example, in searching algorithm we only concern the comparison and omit the others.

Time complexity of an algorithm is often not presented in an equation of $T(n)$ as the more important question in big data is "How fast does $T(n)$ grow as the data grow bigger?" For large value of n, asymptotic time complexity is used. Most common notation of asymptotic time complexity is Big-O, while the other less-used notations being Big-Omega and Big-Theta.

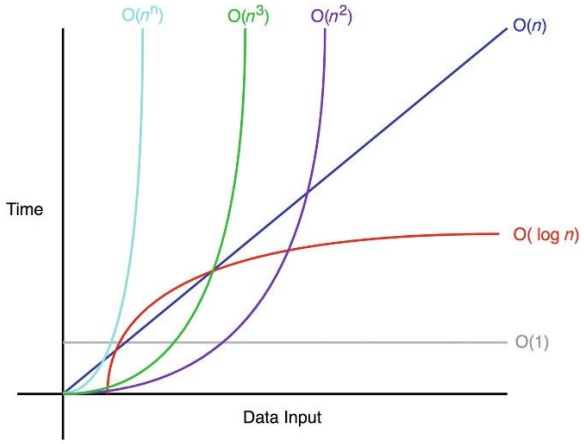

Fig. 5. Graph of speed of growth of functions in Big-O notation.
Source: S. Bae, *JavaScript Data Structures and Algorithms*.

By definition, the algorithm with time complexity $T(n)$ is said to be $O(f(n))$, or denoted as $T(n) = O(f(n))$, if there exists a constant C and n_0 such that $T(n) \leq C \cdot f(n)$ for $n \geq n_0$. The function $f(n)$ should be in simple form without coefficients and other terms, $f(n)$ is only the most significant term. Some examples of $O(f(n))$ are $O(1)$, $O(n)$, $O(n^2)$, $O(n!)$, and $O(n^n)$. As this inequality would hold true for any small $T(n)$ with arbitrary large $f(n)$ such as $f(n) = n^n$, the notation $O(f(n))$ would lose its meaning with such $f(n)$. For that reason, we need to choose smallest $f(n)$ that makes the inequality true.

The algorithm with time complexity $T(n)$ is said to be $\Omega(g(n))$, or denoted as $T(n) = \Omega(g(n))$, if there exists a constant C and n_0 such that $T(n) \geq C \cdot g(n)$ for $n \geq n_0$. Similarly with Big-O, we need to choose largest $g(n)$ that makes the inequality true.

The algorithm with time complexity $T(n)$ is said to be $\Theta(h(n))$, or denoted as $T(n) = \Theta(h(n))$, if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.

If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then:

1. $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$
2. $T_1(n) \cdot T_2(n) = O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
3. $O(c \cdot f(n)) = O(f(n))$, where c is a constant.
4. $f(n) = O(f(n))$

III. Limitations of the Study

The limitation of the study conducted in this paper include the sample size is limited (only around six million characters were analyzed from Brown Corpus), and taken from only American English texts. For that reason, the study result may be inaccurate to a certain point. Moreover, the study result would only relevant to American English texts, and may not be relevant to other languages and uncommon texts, for example paper of mathematics study. The result may also be not relevant to texts that use a lot of uncommon characters or words. The concept of predefined codes would also fail if a text containing an undefined symbol in the methodology is to be compressed.

IV. Methodology

The author used Brown Corpus as the basis to get the frequency of symbols commonly used in English texts. Brown Corpus is an electronic collection of text samples of American English with variety of genres from nonfiction texts to fiction texts. The Brown Corpus was analyzed by using Python program with NLTK module. The author also made assumption that each paragraph is separated by one new line character.

V. Huffman Coding with Predefined Codes

A. Construction of Huffman Codes

With the Brown Corpus as reference, the following frequency table of characters was obtained, shown in Table. 1. The *c* column means *character* while *f* column means *frequency* of the character.

Table. 1. Frequency table of characters, sorted in descending *f*.

c	*f*	*c*	*f*	*c*	*f*
\s	1003303	I	12543	(	2464
e	589980	A	11385	5	2144
t	423392	'	10983	9	2125
a	371418	S	10322	:	1987
o	357020	x	9379	3	1732
i	333212	H	8015	U	1640
n	332908	C	7776	Y	1610
s	300431	M	7455	!	1597
r	287337	B	6527	K	1494
h	249219	W	6003	4	1452
l	192894	;	5566	6	1451
d	184215	1	5182	8	1265
c	139434	P	5162	7	1065
u	127159	q	4862	V	1055
m	113186	j	4748	$	579
f	106409	?	4694	Q	241
p	90770	0	4458	/	236
g	89140	z	4431	*	173
w	83137	F	4263	&	166
y	80164	D	4080	%	147
b	66277	N	3798	Z	122
,	58982	R	3663	X	56
.	55578	G	3444	{	16
v	46206	O	3267	}	16
k	29685	L	3252	[	2
"	17687	E	3166	]	2
\n	15667	J	3008	+	1
T	15568	2	2621		
-	15401	)	2495		

Note: \s means space character and \n means new line character.

The frequency table of the characters were then converted into probability table, shown in Table. 2 with *P* column as the probability value of appearance.

Table. 2. Probability table of characters, sorted in descending *P*.

c	*P*	*c*	*P*	*c*	*P*
\s	0.1681373	I	0.0021020	(	0.0004129
e	0.0988711	A	0.0019079	5	0.0003593
t	0.0709536	'	0.0018406	9	0.0003561
a	0.0622436	S	0.0017298	:	0.0003330
o	0.0598308	x	0.0015718	3	0.0002903
i	0.0558409	H	0.0013432	U	0.0002748
n	0.0557900	C	0.0013031	Y	0.0002698
s	0.0503474	M	0.0012493	!	0.0002676
r	0.0481530	B	0.0010938	K	0.0002504
h	0.0417651	W	0.0010060	4	0.0002433
l	0.0323259	;	0.0009328	6	0.0002432
d	0.0308714	1	0.0008684	8	0.0002120
c	0.0233669	P	0.0008651	7	0.0001785
u	0.0213098	q	0.0008148	V	0.0001768
m	0.0189681	j	0.0007957	$	0.0000970
f	0.0178324	?	0.0007866	Q	0.0000404
p	0.0152116	0	0.0007471	/	0.0000395
g	0.0149384	z	0.0007426	*	0.0000290
w	0.0139324	F	0.0007144	&	0.0000278
y	0.0134342	D	0.0006837	%	0.0000246
b	0.0111069	N	0.0006365	Z	0.0000204
,	0.0098844	R	0.0006139	X	0.0000094
.	0.0093140	G	0.0005772	{	0.0000027
v	0.0077434	O	0.0005475	}	0.0000027
k	0.0049747	L	0.0005450	[	0.0000003
"	0.0029641	E	0.0005306	]	0.0000003
\n	0.0026255	J	0.0005041	+	0.0000002
T	0.0026089	2	0.0004392		
-	0.0025810	)	0.0004181		

A binary tree for Huffman coding was constructed according to the probability table of characters. The resulting binary tree was then tabulated into Table. 3.

Table. 3. Predefined Huffman codes.

c	Huffman code	Length
A	110000000	9
B	1100000111	10
C	1101110110	10
D	0001100000	10
E	11000000111	11
F	0001100010	10
G	11001100010	11
H	1101110111	10
I	110000010	9
J	11000000110	11
K	011101011111	12

L	11000001101	11
M	1100110010	10
N	11001100111	11
O	11001100000	11
P	0111010010	10
Q	00011000010001	14
R	11001100011	11
S	011101000	9
T	110111001	9
U	110011000010	12
V	1100110011001	13
W	1100000010	10
X	11001100110001101	17
Y	110000011001	12
Z	1100110011000111	16
a	1001	4
b	1101111	7
c	00010	5
d	10100	5
e	001	3
f	110001	6
g	011110	6
h	11010	5
i	0110	4
j	0001101101	10
k	11001101	8
l	10101	5
m	110010	6
n	0101	4
o	1000	4
p	011111	6
q	0001101110	10
r	0000	4
s	0100	4
t	1011	4
u	110110	6
v	0111011	7
w	011100	6
x	000110101	9
y	000111	6
z	0001100011	10
0	0001101000	10
1	0111010011	10
2	00011011111	11
3	110011000011	12
4	011101011110	12
5	00011010010	11
6	011101011101	12
7	000110000101	12
8	011101011100	12
9	00011000011	11
'	011101010	9
"	00011001	8
\s	111	3
\n	110111010	9
+	110011001100011001010	21
-	110111000	9
*	110011001100010	15
/	00011000010000	14
.	1100001	7
,	1100111	7
?	0001101100	10
!	110000011000	12
:	110011001101	12
;	0111010110	10
(	00011010011	11
)	00011011110	11
{	110011001100011000	18
}	110011001100011001	19
[	11001100110001100100	20
]	110011001100011001011	21
$	0001100001001	13
%	110011001100000	15
&	110011001100001	15

B. Time Complexity of Huffman Coding

An encoder and decoder were written in Python language based on the predefined Huffman codes as shown in Table. 3. The core algorithms to be shown do not include the conversion of Huffman code from plaintext file to data structure in Python with two reasons:

1. The concerned data size is the size of text instead of the Huffman codes.
2. It could be assumed that the Huffman codes have been implemented in the program since the codes were predefined.

For clarity and ease, the author omitted the time complexity of assignment operations, focusing on string concatenation and hash table access operations with each of them has the value of $T_{concat}(1)$ and $T_{access}(1)$ respectively in time complexity.

To speed up the algorithm, the author used hash table as the data structure for the binary tree, which is `huffmancode` in the algorithms. The hash table's key is character, and its value is the corresponding Huffman code. Another hash table named `flipped_huffmancode` was also made by flipping the key and value in `huffmancode`.

Core algorithm of the encoder is as follows:

```
encodedtext = ''
for char in plaintext:
    encodedtext += huffmancode[char]
```

Let the variable `plaintext` be the string that is to be converted into Huffman code and contains *n* characters. The first line initializes the variable `encodedtext` as empty string, and the variable would later hold the resulting Huffman code. The second line is a loop that traverses each of the character in `plaintext`. The third line contains two concerned operations, hash table access in `huffmancode[char]` and string concatenation in `encodedtext + huffmancode[char]`. The algorithm's complexity could be modelized into (1).

$$T_{encode(n)} = \sum_{i=1}^{n} \left(T_{access}(1) + T_{concat}(1)\right) = n \cdot \left(T_{access}(1) + T_{concat}(1)\right) \quad (1)$$

As there is a constant $C = T_{access}(1) + T_{concat}(1)$, the author could choose *f(n)* = *n* so that $T_{encode}(n) \leq Cf(n) = O(f(n))$. Hence, the asymptotic time complexity of the encoding algorithm is $O(n)$.

Core algorithm of the decoder is as follows:

```
plaintext = ''
sequence = ''
for bit in encodedtext:
    sequence += bit
    if sequence in flipped_huffmancode:
        char = flipped_huffmancode[sequence]
        sequence = ''
        plaintext += char
```

Let the variable `encodedtext` be the string of *m* bits that is to be converted back into plaintext which would contain *n* characters. The first and second line initializes the variable `plaintext` and `sequence` as empty string, and the variable would later hold the resulting plaintext and current bit sequence respectively. The third line is a loop that traverses each of the bit in `encodedtext`. There are two paths of operations inside the loop: first, if there is no key of `sequence`, and second, if there is key of `sequence` in `flipped_huffmancode`. The first path contains only one concatenation operation, while the second path contains two concatenation operations and one hash table access. The algorithm's complexity could be modelized into (2).

$$T_{decode}(\mathrm{n}) = \sum_{i=1}^{n} \left(T_{access}(1) + 2T_{concat}(1)\right) + \sum_{i=1}^{m} \left(T_{concat}(1)\right) = n \cdot T_{access}(1) + (2n + m)\, T_{concat}(1) \quad (2)$$

Because each character's Huffman code is at least 1 bit and at most 21 bits, the relation between *m* and *n* could be written as $n \leq m \leq 21n$ or $m = Cn$ in which $1 \leq C \leq 21$. Then (2) could be rewritten as (3).

$$T_{decode}(n) = n \cdot \left(T_{access}(1) + (2 + C)\, T_{concat}(1)\right) \quad (3)$$

As there is a constant $K = T_{access}(1) + (2 + C)\, T_{concat}(1)$, the author could choose *f(n)* = *n* so that $T_{decode}(n) \leq Kf(n) = O(f(n))$. Hence, the asymptotic time complexity of the decoding algorithm is $O(n)$.

C. Compression Rate of Huffman Coding

To calculate the average compression rate in American English text, the equations below are used assuming plaintext characters are encoded in 8-bits ASCII with the data obtained from Brown Corpus:

$$m = \sum f \quad (5)$$

$$n = \sum (f \times len) \quad (6)$$

$$compression\ rate = \frac{8m - n}{8m} \times 100\% = \frac{8m - \Sigma(f \times len)}{8m} \times 100\% \quad (7)$$

with *m* is number of characters in plaintext, *n* is number of total bits in Huffman codes obtained, *f* is frequency of each character in Brown Corpus, and *len* is length of Huffman code for the corresponding character.

By using Table. 1. and Table. 3., (6) yielded compression rate of 0.4408, equivalent to 44.08% for average American English text.

To validate the compression rate of Huffman coding with predefined codes, four sample texts with varying length, from short to long, were tested.

First, the author used a short tweet by John Cena (retrieved from John Cena on Twitter/ Twitter on 13 December 2021), "Put forth honest effort not just to do good but to be good." From 472 bits (59 characters), the Huffman coding with predefined codes compressed it to 263 bits or equivalently 44.28% compression rate.

Second, the introduction part of this paper, "As data size is increasing ... i.e., predefined codes.", was tested with the Huffman coding. Original length of the plaintext in ASCII bits was 11816 bits (1447 characters), and the compression successfully brought the number down to 6441 bits which translated to 45.49% compression rate.

Third, the author used an article published by National Geographic titled "To beat Omicron, the race is on to tweak existing vaccines" (retrieved from To beat Omicron, the race is on to tweak existing vaccines (nationalgeographic.com) on 13 December 2021) as the sample text with some changes from non-ASCII characters into its similar counterpart in ASCII, including the character 'é'into 'e' and en dash into hyphen. From the original 73112 bits (9139 characters), Huffman coding encoded the text into 41513 bits, which translated to 43.22% compression rate.

Fourth, the author used a story book titled "The Merry Adventures of Robin Hood" (2006) (retrieved from https://www.gutenberg.org/cache/epub/964/pg964.txt on 13 December 2021). The Huffman coding algorithm originally failed because the text contained an unregistered ASCII character '_'. Since the underscore character was most likely the representation of italic font style, the character was deleted in the test. From the original 4717672 bits (589709 characters), the Huffman code encoded it into 2698327 bits, which translated to 42.80% compression rate.

From the four sample texts used, the compression rate was observed relatively consistent in value, ranging from 42.80% to 45.49%.

VI. Conclusion

From the study, it was understood that using Huffman coding with predefined codes may reduce time complexity to *O(n)* in both encoding and decoding steps. The usage of the algorithm with predefined codes may compress any American English text to about 42 – 46% given that the text does not contain any undefined characters in the predefined codes.

VII. Appendix

The Python code used to obtain and analyze Brown Corpus is as follows:

```
from nltk.corpus import brown
from sacremoses import MosesDetokenizer
from collections import Counter

mdetok = MosesDetokenizer()

text = ''
paras = brown.paras()
for para in paras:
    sents_array = [mdetok.detokenize
       (' '.join(sent).replace('``',
       '"').replace("''", '"').replace('`',
       "'").split(), return_str=True) for
       sent in para]

    paragraph = ' '.join(sents_array)
    text += paragraph + '\n'

print(Counter(text))
```


VIII. Acknowledgment

The author would like to express gratitude to family members and friends who provided help supports during the writing of this paper. Author would also like to thank Dra. Harlili, M.Sc. as class lecturer and Dr. Ir. Rinaldi Munir, M.T. as head lecturer for IF2120 Discrete Mathematics course for their spirit and determination to educate their students with enthusiasm.

References

[1] K. H. Rosen, *Discrete Mathematics and Its Applications*. New York: McGraw-Hill, 2019, ch. 11.

[2] R. Munir, School of Electrical Engineering and Informatics, Bandung Institute of Technology, lecture slide: Pohon, 2020.

[3] H. S. Wilf, *Algorithm and Complexity* (Internet Edition). Philadelphia: University of Pennsylvania, 1994, ch. 0.

[4] R. Munir, School of Electrical Engineering and Informatics, Bandung Institute of Technology, lecture slide: Kompleksitas Algoritma, 2020.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 13 Desember 2021

Wesly Giovano (13520071)