Huffman Encoding and Decoding

Omar Yousef Al-Khathlan[#1], Mohammed Al-Wohayyeb[*2]

Computer Science Department, King Saud University

Saudi Arabia

[1]438104887@student.ksu.edu.sa

[2]438103387@student.ksu.edu.sa

***Abstract*— This document serves as our documentation for our project submission.**

***Keywords*— Encoding: convert a message into a coded form. Decoding: convert a coded message into intelligible language.**

I. Introduction

This document outlines our work on the Huffman encoding and decoding algorithm as part of our project.

II. Huffman algorithm

Huffman encoding and decoding is an algorithm to assign binary codes to symbols which reduces the overall number of binary bits needed.

A. Pseudocode for Huffman Tree Construction

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

Fig. 1 Pseudocode for the Huffman algorithm for building the Huffman tree [1].

The tree will store the symbols as leaf nodes and the binary code to each leaf is the path from the root to the particular leaf node with 1 representing a right turn while 0 represents a left turn.

Through this tree, the length of each binary code is assigned based on how frequently a particular symbol occurs within a sentence, such that symbols with higher frequencies get shorter binary codes thus resulting in the compression of the coded message.

III. Implementation

We started by implementing our own data structures as per the request of the project description, Then started to implement the Huffman algorithm through the usage of the aforementioned data structures.

A. Data Structure Implementation

We implemented the following data structures:

- Binary Search Tree (BST).
- Binary Tree (BT: by the name Pair).
- Minimum Heap (used as Priority Queue).

B. Huffman Implementation

We started by parsing the input message and counted the occurrence of each symbol which was later used to construct the Huffman tree, Then we traversed the Huffman tree and generated each symbol's binary code, Afterwards, we encoded the input message, After which we decode it back to its original form.

C. Input and Output

In our codebase, we used a file to input the string to the Huffman class to perform the encoding and decoding it then would output the results onto the command line.

Further explanation can be found as comments within the code itself.

IV. Experiment Description

In our experiment, we tested how effective our implementation of the Huffman compression algorithm against randomly generated input with respect to the ASCII code table which ranges from 0 to 127.

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

Fig. 2 A table representing symbols with its ASCII and HEX code [2].

We started by generating a random starting string and then would replicate that string to ensure that the same letters are encoded and decoded to minimize unwanted fluctuations within the data set that our program was tested against, furthermore we ran multiple tests on each singular test sample and would discard the first run and then would take the average of the remaining test samples.

V. Experimental Data

We tested our implementation of the Huffman algorithm against a randomly generated string with 500 characters and duplicated it until we reached a maximum input length of 20,000 characters. Each sample size has had 11 runs in total which we took the average of the latter 10 runs to ensure that the running time is meaningful.

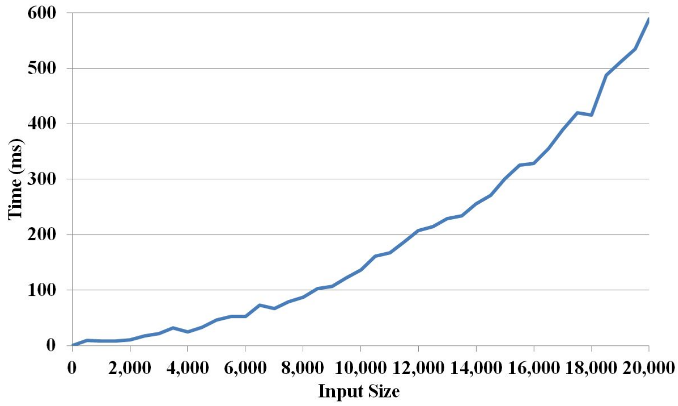

Fig. 3 A line graph representing the time of execution of our Huffman algorithm against string size.

VI. Theoretical Analysis

After a thorough examination of our codebase, we concluded that our worst links in the call chain are the encoding and the decoding calls with theoretical time complexity of *O(n)* and *O(m)* respectively where n is the length of the input string and *m* is the length of the encoded message.

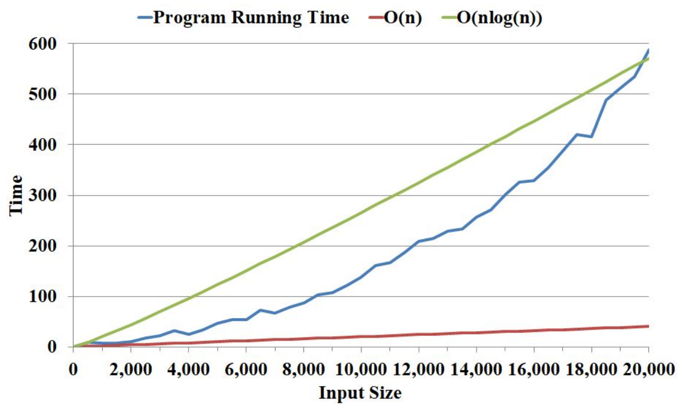

Fig. 4 A line graph representing the program execution time of our Huffman algorithm along with the theoretical complexities *O(nlog(n))* and *O(n)*.

VII. Conclusion

Through our experiment, we were taken back at the level of compression that the Huffman algorithm provides.

We performed several tests some of which were random and some were bespoke and were amazed that when given very large inputs which are randomly generated over a small interval of characters the encoded message reaches a maximum compression value consistently.

REFERENCES

[1] H. Cormen, T., E. Leiserson, C., L. Rivest, R. and Stein, C., 2009. Introduction To Algorithms. 3rd ed.

[2] Ascii.cl. 2020. ASCII Codes - Table Of Ascii Characters And Symbols. [online] Available at: <https://ascii.cl/> [Accessed 23 April 2020].