Homework 4 – Lossless Compression

Mobile Programming and Multimedia

Gabriel Rovesti – 2103389

1 TABLE OF CONTENTS

2 Assignment

Encode the string:

abcabcabcabcfffffffffff000000000000fffffffffffffffffffffff

using:

1. *the LZW algorithm and*
2. *choose an algorithm between Shannon-Fano and Huffman.*

Compare the two results in terms of compression ratio.

3 LZW Algorithm

Let's first start considering the LZW algorithm on the string given in input. Here, the table is created to give an overview representation.

w	k	Output	Code	Symbol
$NULL$	a			
a	b	a	256	ab
b	c	b	257	bc
c	a	c	258	ca
a	b			
ab	c	256	259	abc
c	a			
ca	b	258	260	cab
b	c			
bc	a	257	261	bca
a	b			
ab	c			
abc	f	259	262	$abcf$
f	f	f	263	ff
f	f			
ff	f	263	264	fff
f	f			
ff	f			
fff	f	264	265	$ffff$
f	f			
ff	f			
fff	f			
$ffff$	f	265	266	$fffff$
f	0	263	267	$f0$
0	0	0	268	00
0	0			
00	0	268	269	000
0	0			

00	0			
000	0	269	270	0000
0	0			
00	0			
000	0			
0000	0	270	271	00000
0	0			
0	f	268	272	$0f$
f	f			
ff	f			
fff	f			
$ffff$	f			
$fffff$	f	266	273	$ffffff$
f	f			
ff	f			
fff	f			
$ffff$	f			
$fffff$	f			
$ffffff$	f	273	274	$fffffff$
f	f			
ff	f			
fff	f			
$ffff$	f			
$fffff$	f			
$ffffff$	f			
$fffffff$	f	274	275	$ffffffff$
f	f			
ff	f			
fff	f			
$ffff$	f			
$fffff$	f			
$ffffff$	EOF	273		

The algorithm is applied step by step based on the algorithm code taken from the slides. Basically, we encode each character starting from the ASCII table (0-255) and see if the word exists inside the dictionary; if not, the concatenation is then added inside of the dictionary and the code for the specific word is given in output, then adding the code for the considered string. This way, the vocabulary is dynamically built, encoding the variable-length strings each time.

So, the encoded sequence is:
$a\ b\ c$ 256 258 257 259 f 263 264 265 263 0 268 269 270 268 266 273 274 273

The original size is given by the number of bits of the whole original encoding multiplied by the number of bits given the representation, so $60 * 8 = 480\ b$.

We are using two bytes, given the number of bits occupied is 274, so we have the number of characters occupied by the encoding (21). Given the encoding is multiplied by the number of bytes occupied ($2\ B = 16\ b$), we would have $21 * 16 = 336\ b$ for the total encoded size.

Quoting the formula of data compression ratio present here:

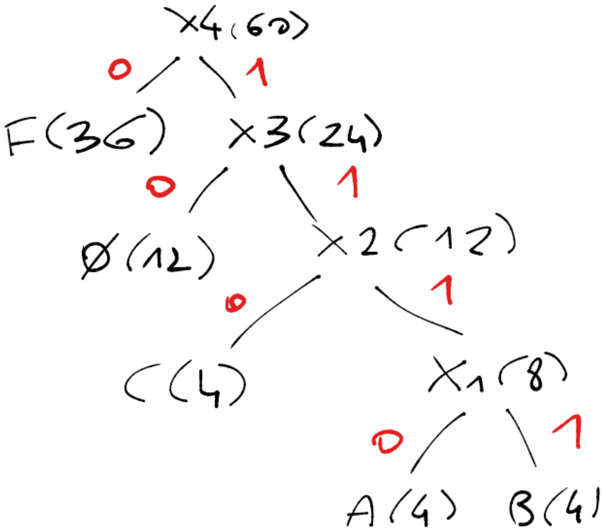
$$Compression\ Ratio = \frac{Uncompressed\ Size}{Compressed\ Size} = \frac{480}{336} = 1.43$$

4 Huffman Algorithm

In this section, the Huffman algorithm is chosen and applied, with the following table describing each symbol, occurrences and the encoding, obtained looking at the tree obtained (0 for left children, 1 for the right children), hence considering the total number of those.

Symbol	*N. of occurrences (#)*	$\log_2(\frac{1}{p_i})$	*Code*	*N. of bits*
A	4	$\log_2(\frac{1}{15})$	1110	16
B	4	$\log_2(\frac{1}{15})$	1111	16
C	4	$\log_2\left(\frac{1}{15}\right)$	110	12
F	36	$\log_2(\frac{3}{5})$	0	36
0	12	$\log_2\left(\frac{1}{5}\right)$	10	24

The corresponding tree is represented here representing the encoding is shown here:



The algorithm is bottom-up, so we start from the lowest-occurrences nodes, in this case A, B, forming a new node as sum. Given the tree would be unbalanced, the character C is then summed subsequently, forming a sum node of 12. Continuing this way, we sum all occurrences of the nodes, reaching the root of 60.

The algorithm gives in output as encoded string:

- Original: abcabcabcabcffffffffffff000000000000ffffffffffffffffffffffff
- Encoded: 1110 1111 110 1110 1111 110 1110 1111 110 1110 1111 11 0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

We then compute how many bits are occupied, considering this is computed multiplying the number of occurrences with how many bits the single code occupies:

$$4 * 4 + 4 * 4 + 4 * 3 + 36 * 1 + 12 * 2 = 104$$

The occupation is given from the number of bits needed by the ratio (8 bits) multiplied by the different chars found inside the encoding (5) multiplying by 2 to represent the column of the encoded sequence (as much as the number of symbols). Combining all of this we get $(8 * 5) * 2 = 80\ b$. This is then summed with the result of the encoding, specifically $80 + 104 = 184\ b$ for the encoded size.

To calculate the effect of the entropy, we use the formula $H(S) = \mu = \sum_i p_i \log_2\left(\frac{1}{p_i}\right) = 1.69$, obtaining as theoretical occupation $H(S)$ multiplied by the number of characters (60), $1.69 * 60 = 101.27$.

The original uncompressed ratio is given by the total number of original bits multiplied by 8 bits, so $60 * 8 = 480$ (so, number of bits occupied multiplied by 8 bits).

Quoting the formula of data compression ratio present here:

$$Compression\ Ratio = \frac{Uncompressed\ Size}{Compressed\ Size} = \frac{480}{184} = 2.61$$

5 CONCLUSIONS

The LZW algorithm achieved a compression ratio of 1.43, compressing the original data from 480 bits to 336 bits.

The Huffman algorithm performed better, achieving a compression ratio of 2.61, compressing the original data from 480 bits to 104 bits (plus an additional 80 bits for the encoding table). So, in this specific case, it provided significantly better compression.

This is due to the Huffman algorithm's ability to assign shorter codes to more frequent symbols, effectively exploiting the skewed symbol distribution in the input data. This is based on construction of a code tree based on symbol frequencies, which can be computationally expensive for large input data.

LZW started becoming more efficient after more appearances of the same characters and their combinations. This, on the other hand, does not require any prior knowledge of symbol frequencies and can adapt to the input data dynamically, making it more suitable for scenarios where the input data is not known in advance or has varying symbol distributions.

As a matter of fact, recall LZW does not need to memorize the table, reducing the overhead needed in saving it but building it dynamically during the decompression phase. So, in the end, Huffman always obtains the optimum in terms of compression for each string length, while LZW needs data of at least 100 kb to obtain efficient results comparable to Huffman's.