

Data Compression
Erick Oduniyi (eeoduniyi@gmail.com)

Data Compression

Erick Oduniyi (erickoduniyi@ku.edu)

Department of Electrical Engineering & Computer Science, University of Kansas
1450 Jayhawk Blvd, Lawrence, KS 66045, USA

1. Nature of Data Compression

Data compression or *source coding* is the process of achieving reduced file sizes by using a smaller number of bits to encode information. Reducing file sizes are chiefly important to communication systems as it enables or improves the speed of transmission over communication channels.

Data compression is accomplished by *identifying redundancy* or *removing* “insignificant” information from the original file. Data compression that involves eliminating statistical redundancy is *lossless compression* whereas explicitly eliminating bits from the original representation is *lossy compression*. Both methods work to minimize the physical storage and transmission of data and accordingly impose their respective trade-offs. For instance, sound and audio applications make use of lossless encoding formats like .wav and .flac, which maintain reversibility but are often larger file sizes than their smaller, but irreversible lossy counterparts such as .mp3 and .aac. On one hand, the average music listener may be fine with most .mp3 or .aac encoded files as there may be no perceptible quality degradation. On the other hand, the mere mention of .mp3 may cause visible distress for the self-proclaimed audiophile.¹

1.1. History of data compression

The history of data compression can be traced back to 1838 with Samuel Morse’s *Morse codes*, which encode text characters as sequences of symbols $\in \{., -, \cdot\cdot\cdot, \cdot\cdot\cdot\cdot, \cdot\cdot\cdot\cdot\cdot, \dots\}$ and takes advantage of frequencies in the English language, using shorter codewords for frequently used letters like e (·) and t (—).

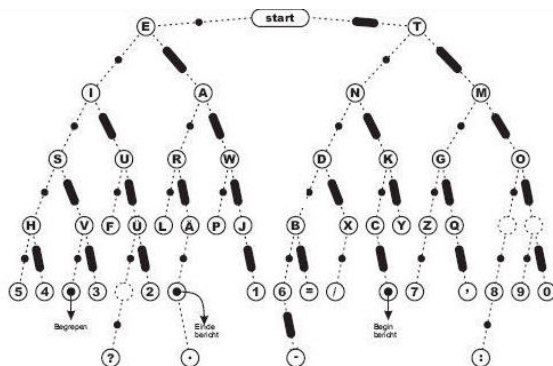


Fig. 1 – Morse decoder

¹See <https://www.gearslutz.com/board/high-end/724128-lossy-audio-bad-you.html> for a modern review on lossy audio encoding.

The theoretical framework for modern data compression schema was developed in the late 1940s along with the field of information theory. Together with Robert Fano, Claude Shannon, the chief architect of information theory, authored fundamental papers on coding theory, outlining how code words could be routinely assigned based on probabilistic blocks. In 1951 David Huffman provided an optimal scheme for this routine with his *Huffman coding algorithm*. In the late 1970s Abraham Lempel and Jacob Ziv ushered a generation of encoding schemes based on pointers and dictionaries. A decade later, Terry Welch made improvements on Lempel and Ziv’s *LZ78 algorithm* which became known as the *LZW algorithm*, one of the most popular general purpose compression schemes. The late 1980s and 1990s introduced audio coding, digital video coding, .zip, and the .rar file formats. Today, more data is being generated than ever before, and as such, communication systems must rely on sophisticated data compression algorithms, various heuristics, and innovations in hardware to transmit, store, and decode enormous amounts of information.

2. Data Compression Algorithms

As a point to become more familiar with the history of data compression and information theory, the algorithmic outline and implementation of the *Huffman coding algorithm*, *Adaptive Huffman algorithm*, *Shannon-Fano-Elias algorithm*, *Tree-based Lempel Ziv algorithm (LZ78)*, and their associated *definitions*, *complexity*, *compression ratio*, and amount of *space saved* are presented. Each data compression algorithm was implemented in the MATLAB programming language, using the ASCII (American Standard Code for Information Interchange) encoding of characters, and the U.S Constitution as the file to be compressed:

United States Constitution

“We the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution for the United States of America....”

2.1. Source files

In addition to compressing the *U.S constitution*, the first chapter of *Alice's Adventures in Wonderland*, aggregated transcripts of CHILDES' English child-directed speech from ages 0 - 36 months, and aggregated transcripts of Voxforge's normative adult speech are used as files to be compressed. The English Voxforge corpus is a freely available data repository that contains 119 hours of audio, 14,590 unique words, and 84,541 word tokens. The child-directed speech corpus was selected from a subset of the American English CHILDES corpus (MacWhinney, 1998). This subset contains a collection of conversations between children (13 to 84 months of age) and their parents or guardians. From this subset we stripped the parent utterances directed to the target children ultimately deriving a corpus composed of 217 hours of audio, 400 unique word and 4,384 word tokens. In order to carry out similar compression task, all of the source files have been normalized such that they only contain characters $\in \{A...Z\} + \{\text{SPACE}\}$.

thomas_cover.txt

WE NOW PUT CONTENT IN THE DEFINITION OF ENTROPY BY ESTABLISHING THE FUNDAMENTAL LIMIT FOR THE COMPRESSION OF INFORMATION DATA COMPRESSION CAN BE ACHIEVED BY ASSIGNING SHORT DESCRIPTIONS TO THE MOST FREQUENT OUTCOMES OF THE DATA SOURCE ...

alice.txt

ALICE S ADVENTURES IN WONDERLAND LEWIS CARROLL THE MILLENNIUM FULLCRUM EDITION CHAPTER I DOWN THE RABBIT HOLE ALICE WAS BEGINNING TO GET VERY TIRED OF SITTING BY HER SISTER ON THE BANK AND OF HAVING NOTHING TO DO ONCE OR TWICE SHE HAD PEEPED INTO THE BOOK HER SISTER WAS READING BUT IT HAD NO PICTURES OR CONVERSATIONS IN IT AND WHAT IS THE USE OF ...

voxforge.txt

A B D E F G H I K ETCETERA BUT A B C D A BABBLING PROCESSION OF THE BEST STORIES IN THE WORLD SENT FROM THE HEART OF THE EARTH A BABY MUST BE ALLOWED TO PUT A SPOON IN ITS MOUTH POINTED END FOREMOST A BABY OF TWO OR AT LEAST BY THE TIME HE IS THREE A BALL DRESS MAY BE EXQUISITE IN DETAIL BUT IT IS OFTEN MERELY EFFECTIVE A BAND OF MONOS SURPRISED ...

childes1.txt

OPEN THERE ARE SO MANY TOYS OH MAMA TOO MANY TOYS OKAY WAS THAT GOOD FUNNY EATING NOISE DO THESE SQUEAK THERE BUILD TH EM UP OKAY WHAT TAKE THEM DOWN WHAT OH BOOM LAUGH YEAH BOOM LAUGH WHERE S DADDY DADDY NOT HERE DADDY S NOT HERE PRETTY PRETTY HERE OH LAUGH WHAT BOOM DADDY BOOM WHERE DID IT GO BOOM YOU WANT THAT ONE OKAY DADDY I LL GO GET YOUR ...

childes2.txt

THEY FELL DOWN BECAUSE YOU KNOCKED THEM DOWN SIT DOWN SIT SIT ALL GONE WHAT DO YOU WANT ALICE LOOK DOWN THERE TELL ME WHAT YOU WANT YOU RE GONNA FLY WELL CLIMB UP HERE I LL MORE THESE OVER CLIMB UP WADI LAUGHS THAT S GOOD THAT S GOOD SQUEAL THERE SHE IS ON MOMMY S LAP NO YOU RE NOT LEAVING YOU VE GOT TO STAY CLOSE WHY DON T WE CALL DADDY LET S CALL ...

childes3.txt

OKAY RIGHT OH HOW NICE THANK YOU THAT S GOOD DID YOU PUT MILK AND SUGAR IN IT DID YOU PUT MILK AND SUGAR IN IT THANK YOU VERY GOOD GOOD I M FINISHED I M FINISHED I FINISHED IT ALL GONE GIVING ME MORE YEAH THANK YOU OH THANK YOU WHAT S THIS MORE COFFEE THIS MORE COFFEE HOW A BOUT LOIS MAYBE LOIS WANTS SOME LAUGHS WHERE S LOIS PETER PETER WHERE S LOIS ...

2.2. Evaluation criteria

The *compression ratio* CR is defined as the ratio between the uncompressed size and compressed size:

$$CR = \frac{\text{Uncompressed Size}}{\text{Compressed Size}} \quad (1)$$

Thus a representation that compresses a 10 MB file to 2 MB has a compression ratio of $10/2 = 5$, often notated as an explicit ratio, 5:1 (read "five" to "one"), or as an implicit ratio, 5/1. Note that this formulation applies equally for compression, where the uncompressed size is that of the original; and for decompression, where the uncompressed size is that of the reproduction. Alternatively, the amount of *space saved* SS is defined as the reduction in size relative to the uncompressed

size:

$$SS = 1 - \frac{1}{CR} \quad (2)$$

Thus a representation that compresses a 10MB file to 2MB would yield a space savings of $1 - 2/10 = 0.8$, often notated as a percentage, 80%. Finally, the expected codeword length $L(C)$ is given as:

$$L(C) = \sum_{x=i}^I p(x)l(x) \quad (3)$$

This is the average codeword length when encoding a single source symbol. But since averaging is a linear operation, the average length of a coded message with N source symbols

2.3. Huffman coding

The *Huffman coding algorithm* accomplishes compression by assigning fewer bits for more frequently occurring characters. This is done by constructing a *Huffman tree*, which is a binary tree that stores the characters at the leaves, and whose root-to-leaf paths provide the bit sequence used to encode the characters. Hence, the algorithm works to reduce the traversal through the tree to characters with high frequencies (weights).

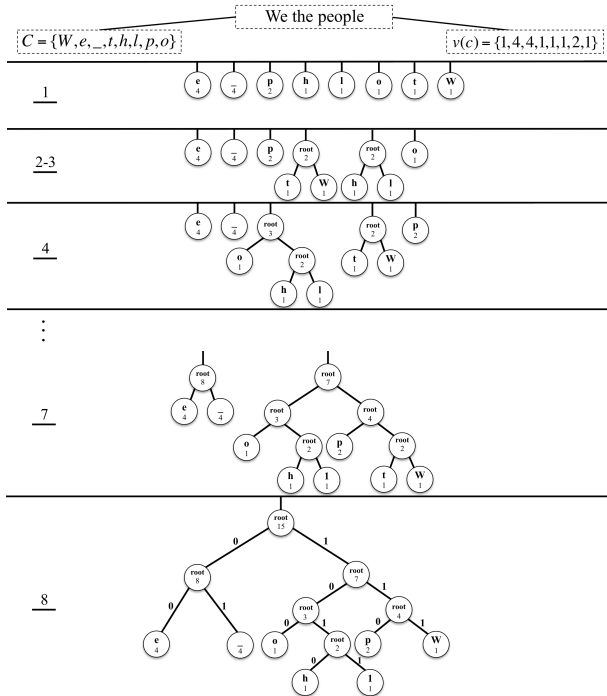


Fig. 2 – Static Huffman tree construction: “We the people”

2.3.1. Algorithm overview

First, the Huffman scheme requires a vector, $v(c)$ of representational weights v_i for n unique characters in the file alphabet C . Using $v(c)$ a forest of n 1-node trees $\{(c_1, v_1), \dots, (c_n, v_n)\}$

are generated. Then, the two trees with the lowest weights t_{1w} , and t_{2w} are combined until the optimal encoding tree H_{tree} is left. The Huffman coding algorithm can be given as:

Algorithm 1 Huffman coding

```

1: procedure HUFFMAN( $C, v(c)$ )
2:    $H_{tree} = \{(c_1, v_1), \dots, (c_n, v_n)\}$ 
3:   for  $n : 2$  in  $C$  do
4:      $H_{tree} = \text{SORT}(H_{tree})$ 
5:      $root(c_{n-1} + c_n, t_{1w} + t_{2w})$ 
6:      $H_{tree} = H_{tree} - \{(c_{n-1}, t_{1w}), (c_n, t_{2w})\}$ 
7:      $H_{tree} = \{H_{tree}, root\}$ 
8:   end for
9:   Return  $H_{tree}$ 
10: end procedure

```

After the tree is constructed, a unique bit-string representation (*prefix code*) for every distinct character can be generated by traversing their root-to-character path p_{char} within the H_{tree} .

Algorithm 2 Codebook generator

```

1: procedure CODEBOOK( $H_{tree}$ )
2:   for each  $char$  in  $C$  do
3:      $p_{char} = H_{tree}('char')$ 
4:      $code = \lambda$ 
5:     while  $char \neq H_{tree}(p_{char})$  do
6:       if RIGHT_TRAVERSE( $H_{tree}$ ) then
7:          $code = code : 1$ 
8:       else
9:          $code = code : 0$ 
10:      end if
11:    end while
12:     $Codebook('char') = code$ 
13:  end for
14:  Return  $Codebook$ 
15: end procedure

```

Table 1 – Example Codebook – Static Huffman

character	code
e	00
-	01
o	100
p	110
h	1010
l	1011
t	1110
W	1111

2.3.2. Implementation

The implementation of the static or vanilla Huffman coding algorithm was done in MATLAB.

2.3.3. Encoding results

Table 2 – Results– Static-Huffman

<i>file</i>	<i>length</i> [chars]	<i>H(x)</i> [bits]	<i>L(C)</i> [bits]	<i>T</i> [secs]	<i>CR</i>	<i>SS</i>
thomas_cover.txt	e	2/13	8/13	7/13	0.100	3
american_constitution.txt	h	4/13	4/13	2/13	0.00	0
alice.txt	-	2/13	6/13	5/13	0.011	3
voxforge.txt	o	1/13	9/13	17/26	0.1010	4
childes1.txt	p	1/13	10/13	19/26	0.1011	4
childes2.txt	l	1/13	11/13	21/26	0.1100	4
childes3.txt	t	1/13	12/13	23/26	0.1110	4

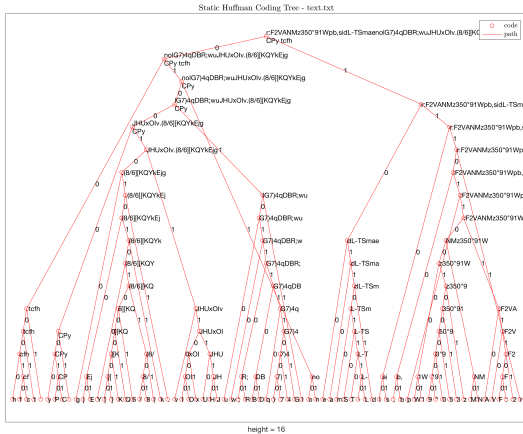


Fig. 3 – MATLAB Static Huffman: U.S Constitution

2.4. Adaptive Huffman coding

Unlike the standard Huffman coding algorithm, the *Adaptive Huffman coding algorithm* assumes no initial representational weights or probabilities on the input symbols. Hence, it performs “online” encoding by dynamically adjusting the Huffman tree as symbols are being streamed from the input file. Typically, a weightless node (or *Not Yet Transferred* node) is used to represent new incoming symbols, while if the encoding system has encountered the symbol before it outputs the path to the symbol in the current tree. Once the root-to-path within the tree has been identified for the symbol, the tree is adjusted as necessary; updating the frequencies of the related nodes and reorganizing the tree such that it maintains the *sibling-property*.

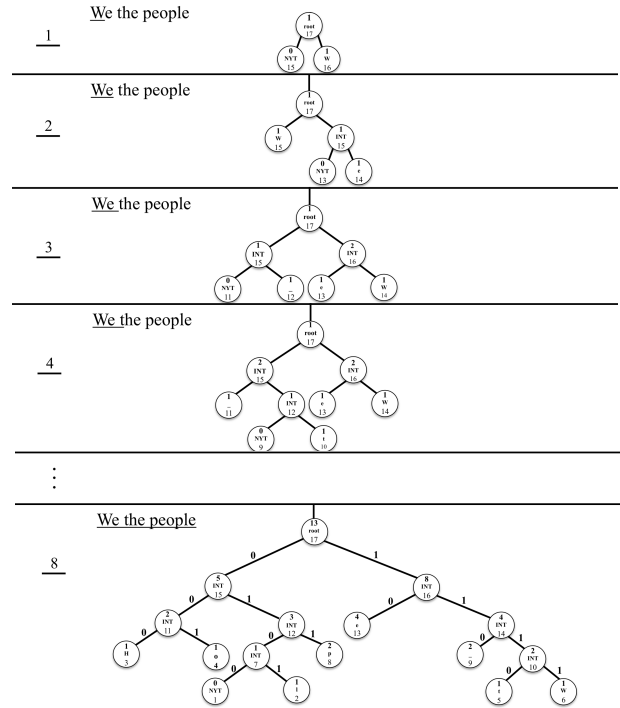


Fig. 4 – Adaptive Huffman tree construction: “We the people”

2.4.1. Algorithm overview

First, the Adaptive Huffman encoder starts with a root node called the Not Yet Transferred (*NYT*) node. Each node is given an id *name*, a *weight* to denote its frequency, and some integer i between 1 and $2n - 1$ to denote the *precedence* or *order* it falls within the tree. The *root* node maintains the highest order and *weight*. Then, as the first symbol is read from the input file F the *NYT* node spawns the first left and right child node. At this point, the *root* node *weight* swaps from being the *NYT* node *weight* to the sum of the newly inputted symbol node *weight* (e.g. *weight* = 1) and the *NYT* node with a *weight* of 0. From this point, the *NYT* node always maintains the lowest order and *weight* (e.g. *weight* = 0; $i = 1$). The *NYT* node will continue to spawn children² for all newly inputted nodes. Repeated nodes (symbols already in the tree) are processed to their respective root-to-character paths p_{char} swapping orders within the tree as to maintain the *sibling-property*. Finally, the symbol node's *weight* is incremented reflecting its frequency. This process is carried out until the last character in F is read, eventually returning some optimal encoding tree H_{tree} . The Adaptive Huffman coding algorithm can be given as:

²Internal nodes (INT_{node}) serve as non-symbol nodes, where they can be the sum of internal or symbol nodes.

Algorithm 3 Adaptive Huffman coding

```

1: procedure ADAPTIVE_HUFFMAN( $F$ )
2:    $size = 2(\text{UNIQUE}(F)) + 1$ 
3:    $order = size$ 
4:    $root_{node} = ('root-node', 0, size)$ 
5:    $H_{tree} = \{root_{node}\}$ 
6:   for all  $char$  in  $F$  do
7:     if ' $char$ '  $\notin H_{tree}.names$  then
8:       if  $H_{tree}.length > 1$  then
9:          $INT_{node} \rightarrow \text{LEFT.CHILD}(NYT)$ 
10:         $INT_{node} = ('INT-node', this.node_{weight}, this.node_{order})$ 
11:         $NYT = \{('char', 1, order - 1), ('NYT-node', 0, order - 2)\}$ 
12:         $H_{tree} = \{H_{tree}, NYT\}$ 
13:         $order = order - 2$ 
14:      else
15:         $NYT = \{('char', 1, order - 1), ('NYT-node', 0, order - 2)\}$ 
16:         $H_{tree} = \{H_{tree}, NYT\}$ 
17:         $order = order - 2$ 
18:      end if
19:    else
20:       $p_{char} = H_{tree}('char')$ 
21:       $LEAF_{node} \rightarrow H_{tree}(p_{char})$ 
22:       $LEAF_{node} = ('char', this.node_{weight} + 1, this.node_{order})$ 
23:    end if
24:     $H_{tree} = \text{UPDATE}(\{root_{node}, \dots, NYT\})$ 
25:  end for
26:  Return  $H_{tree}$ 
27: end procedure

```

After the tree is constructed, a unique bit-string representation (*prefix code*) for every distinct character can be generated by traversing their root-to-character path p_{char} within the H_{tree} .

Table 3 – Example Codebook – Adaptive Huffman

character	code
e	10
-	110
h	000
o	001
p	011
l	0101
t	1110
W	1111

2.4.2. Implementation

The implementation of the Adaptive Huffman coding algorithm was done in MATLAB.

2.4.3. Encoding results

Table 4 – Results – Adaptive-Huffman

file	length [chars]	$H(x)$ [bits]	$L(C)$ [bits]	T [secs]	CR	SS
thomas_cover.txt	h	2/13	8/13	7/13	0.100	3
american_constitution.txt	e	4/13	4/13	2/13	0.00	0
alice.txt	-	2/13	6/13	5/13	0.011	3
voxforge.txt	o	1/13	9/13	17/26	0.1010	4
chilides1.txt	p	1/13	10/13	19/26	0.1011	4
chilides2.txt	l	1/13	11/13	21/26	0.1100	4
chilides3.txt	t	1/13	12/13	23/26	0.1110	4

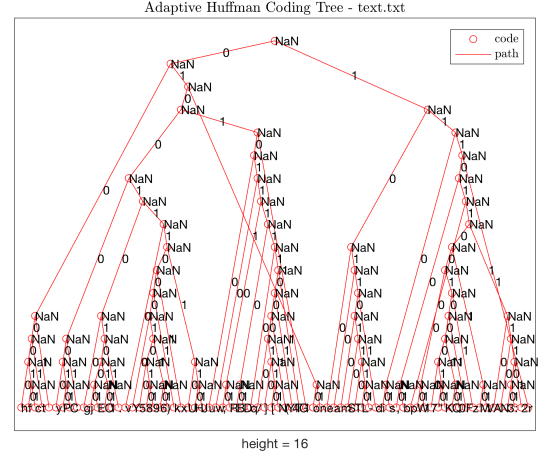


Fig. 5 – MATLAB Adaptive Huffman: U.S Constitution

2.5. LZ78 coding

The *LZ78 algorithm* achieves compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input file. LZ78-based schemes work by registering phrases into a dictionary and then, when a repeat occurrence of that particular phrase is found, outputting the dictionary index instead of the phrase. That is, every iteration will output a pair $(index, a)$, where $iindex$ is an index of the phrase into the dictionary and a is the next symbol following immediately after the found phrase. Hence, the coding algorithm works to build a dictionary in the form of a tree, where the root-to-node path will generate a phrase from the input file.

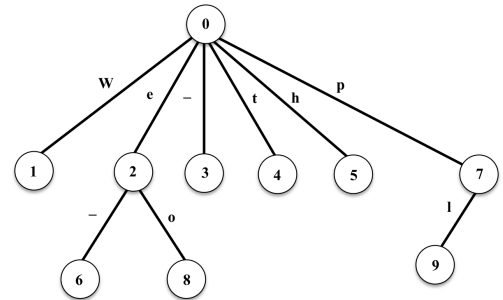


Fig. 6 – LZ78 tree construction: “We the people”

2.5.1. Algorithm overview

The encoding system starts with a one node tree that denotes the empty string λ . Then, each dictionary entry is of the form $\text{Dictionary}(index) = (index, character)$, where $index$ is the index to a previous phrase, dictionary entry, and $character$ is appended to the string represented by $\text{Dictionary}(index)$. For each $character$ of the input file F , the dictionary is searched for a match: $(LAST_{index}, character)$. If a match is found, then last matching $index$ is set to the $index$ of the matching

entry, and nothing is output. If a match is not found, then a new dictionary entry is created: $Dictionary(NEXT_{index}) = (LAST_{index}, character)$, and the algorithm outputs last matching $index$, followed by a $character$, then resets last matching $index = 0$ and increments next available $index$. When the end of F is reached, the algorithm outputs the last matching $index$. Lempel-Ziv coding algorithm can be given as:

Algorithm 4 LZ78 coding

```

1: procedure LZ78( $F$ )
2:    $Dictionary = \{\lambda\}$ 
3:    $index = 1$ 
4:    $prefix = \lambda$ 
5:   while  $F \neq \emptyset$  do
6:      $character = F(NEXT_{character})$ 
7:     if  $prefix + character \in Dictionary$  then
8:        $prefix = prefix + character$ 
9:     else
10:      if  $prefix = \lambda$  then
11:         $index = 0$ 
12:      else
13:         $index = Dictionary(prefix)$ 
14:         $entry = (index, character)$ 
15:         $Dictionary = \{(index, prefix + character)\}$ 
16:         $prefix = \lambda$ 
17:      end if
18:    end if
19:  end while
20:  if  $prefix \neq \lambda$  then
21:     $index = Dictionary(prefix)$ 
22:     $entry = (index,)$ 
23:  end if
24:  Return  $Dictionary$ 
25: end procedure

```

Table 5 – Example Codebook – LZ78

$index$	$entry$	$code$	$binary$
1	W	(0, W)	0,000
2	e	(0, e)	0,001
3	-	(0, -)	0,010
4	t	(0, t)	0,011
5	h	(0, h)	0,100
6	e-	(2, o)	10,101
7	p	(0, p)	0,110
8	eo	(2, o)	10,101
9	pl	(7, l)	111,111
10	e	2	10

2.5.2. Implementation

The implementation of the LZ78 coding algorithm was done in MATLAB

2.5.3. Encoding results

Table 6 – Results– LZ78

$file$	$length$ [chars]	$H(x)$ [bits]	$L(C)$ [bits]	T [secs]	CR	SS
thomas_cover.txt	h	2/13	8/13	7/13	0.100	3
american_constitution.txt	e	4/13	4/13	2/13	0.00	0
alice.txt	-	2/13	6/13	5/13	0.011	3
voxforge.txt	o	1/13	9/13	17/26	0.1010	4
chldes1.txt	p	1/13	10/13	19/26	0.1011	4
chldes2.txt	l	1/13	11/13	21/26	0.1100	4
chldes3.txt	t	1/13	12/13	23/26	0.1110	4

2.6. Shannon-Fano-Elias coding

The *Shannon-Fano-Elias* coding algorithm (SFE) is a coding scheme that performs compression by using the distribution over a source of symbols to generate codewords. As a notable achievement, this algorithm generates prefix code words for any distribution over an arbitrarily large sized alphabet. Concretely, the SFE compression procedure is applied to a sequence of random variables and their respective probabilities to calculate the cumulative distribution function (CDF), the modified CDF, and then the modified-CDF's binary representation. Being the predecessor to arithmetic coding, SFE requires arithmetic which precision grows with block size. Then, the codeword $c(x)$ consists of the first $\log \left(\frac{1}{p(x)} \right) + 1$ bits of the binary expansion of the sum:

$$\sum_{j < x} p(j) + \frac{1}{2} p(x)$$

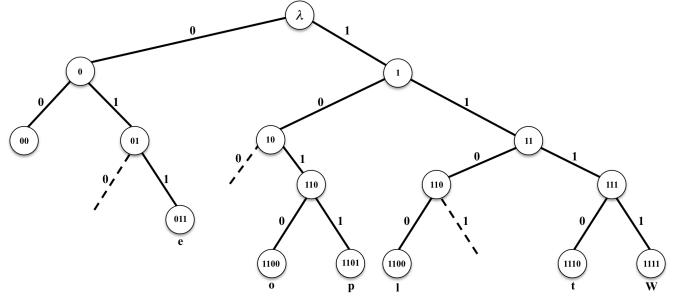


Fig. 7 – SFE tree construction: “We the people”

2.6.1. Algorithm overview

First, the SFE source encoder arranges the symbols in the alphabet $|\mathcal{X}|$ according to their decreasing probabilities $p(x) = p(x_1) > p(x_2) > \dots > p(x_n)$. Using the alphabets distribution, the cumulative probability distribution is computed as $F(x)$:

$$F(x) = \sum_{j < x} p(x)$$

Then, the modified CDF is calculated as $\bar{F}(x)$:

$$\bar{F}(x) = \sum_{j < x} p(j) + \frac{1}{2} p(x)$$

The length of the code word can be determined by $l(x)$:

$$l(x) = \left\lceil \log \left(\frac{1}{p(x)} \right) + 1 \right\rceil$$

Finally, the code words $C(x)$ is generated, converting the symbol sequence into a binary sequence of 0s and 1s. The Shannon-Fano-Elias coding algorithm can be given as:

Algorithm 5 *Shannon-Fano-Elias coding*

```

1: procedure SHANNON_FANO_ELIAS( $p(x), l(x)$ )
2:   for each  $x$  in  $p(x), l(x)$  do
3:      $p(x) = \text{SORT}(p(x))$ 
4:      $F(x) = \sum_{x_i \leq x} p(x)$ 
5:      $\bar{F}(x) = \sum_{j < x} \frac{1}{2} p(j) + p(x)$ 
6:      $binary_{rep} = \text{BINARY}(\bar{F}(x))$ 
7:      $Codebook(x) = \text{CODE}(binary_{rep}, l(x))$ 
8:   end for
9:   Return  $Codebook$ 
10: end procedure

```

Table 7 – Example Codebook – Shannon-Fano-Elias

x_i	c_i	$p(x)$	$F(x)$	$\bar{F}(x)$	$\bar{F}(x)_{binary}$	$l(x)$	$C(x)$
x_1	e	4/13	4/13	2/13	0.00	2	00
x_2	-	2/13	6/13	5/13	0.011	3	011
x_3	h	2/13	8/13	7/13	0.100	3	100
x_4	o	1/13	9/13	17/26	0.1010	4	1010
x_5	p	1/13	10/13	19/26	0.1011	4	1011
x_6	l	1/13	11/13	21/26	0.1100	4	1100
x_7	t	1/13	12/13	23/26	0.1110	4	1110
x_8	W	1/13	13/13	25/26	0.1111	4	1111

2.6.2. Implementation

The implementation of the Shano-Fano-Elias algorithm was done in MATLAB

2.6.3. Encoding results

Table 8 – Results– Shannon-Fano-Elias

file	length [chars]	$H(x)$ [bits]	$L(C)$ [bits]	T [secs]	CR	SS
thomas_cover.txt	h	2/13	8/13	7/13	0.100	3
american_constitution.txt	e	4/13	4/13	2/13	0.00	0
alice.txt	-	2/13	6/13	5/13	0.011	3
voxforge.txt	o	1/13	9/13	17/26	0.1010	4
chldes1.txt	p	1/13	10/13	19/26	0.1011	4
chldes2.txt	l	1/13	11/13	21/26	0.1100	4
chldes3.txt	t	1/13	12/13	23/26	0.1110	4

3. Discussion

The problem of data compression is to choose codeword lengths that minimize the average number of bits per symbol, in addition to the codes are instantaneous. The four algorithms accomplish this, however they each prescribe their respective sets of pros and cons. In general, the *Huffman* coding algorithm provides a simple and fast coding scheme, where the algorithm complexity grows exponentially with the block length, but requires a probability distribution. Further, it is difficult to use Huffman coding for extended sources with large alphabets or when the symbol probabilities vary with time. *Shannon-Fano-Elias* coding is a type of arithmetic coding where complexity grows linearly with the block size, but also requires source of symbol statistics. Finally, the *LZ78* scheme utilizes a dictionary to store patterns and transmit the buffer offset and length. These universal coders do not need to know the symbol probability distribution and is instead estimated in a single pass of the data.