Data Compression

Erick Oduniyi (erickoduniyi@ku.edu)

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* wheares 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 $\in \{\cdot, -, \cdot\cdot, --, \cdot-, -\cdot, \ldots\}$ and takes advantage of frequencies in the English language, using shorter codewords for frequently used letters like e ($\cdot$) and t ($-$).

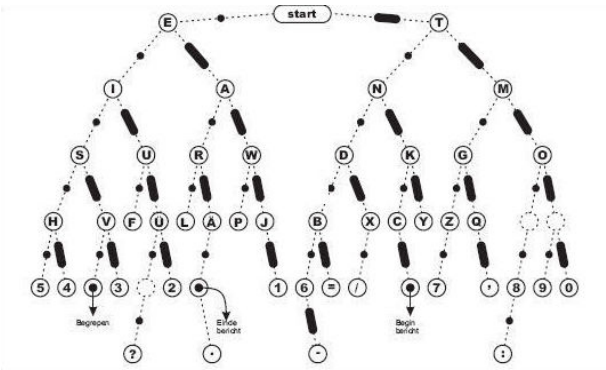


Fig. 1 – Morse decoder

[1] See *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. 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).

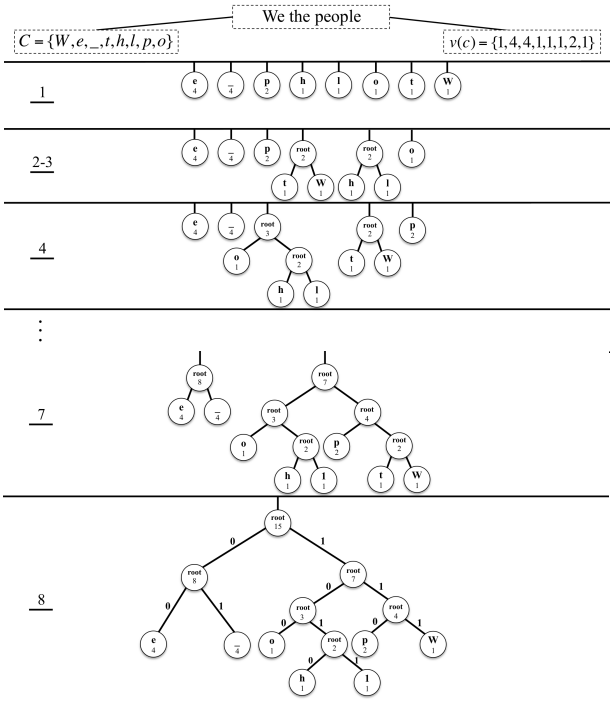


Fig. 2 – Static Huffman tree construction: "We the people"

2.1.1. 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 $\{(c_1, v_1), ..., (c_n, v_n)\}$ 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*

procedure HUFFMAN($C, v(c)$)
  $H_{tree} = \{(c_1, v_1), ..., (c_n, v_n)\}$
  for $n : 2$ in C **do**
    $H_{tree} = \text{SORT}(H_{tree})$
    $root(c_{n-1} + c_n,\ t_1 w + t_2 w)$
    $H_{tree} = H_{tree} - \{(c_{n-1}, t_{1w}), (c_n, t_{2w})\}$
    $H_{tree} = \{H_{tree}, root\}$
  end for
  Return H_{tree}
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*

procedure CODEBOOK(H_{tree})
  for each *char* in C **do**
    $p_{char} = H_{tree}(\text{'}char\text{'})$
    $code = \lambda$
    while $char \neq H_{tree}(p_{char})$ **do**
      if RIGHT_TRAVERSE(H_{tree}) **then**
        $code = code : 1$
      else
        $code = code : 0$
      end if
    end while
    $Codebook(\text{'}char\text{'}) = code$
  end for
  Return *Codebook*
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

2.1.2. Implementation

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

2.1.3. Encoding results

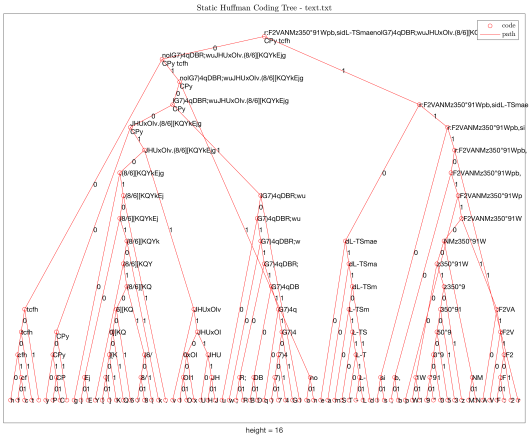
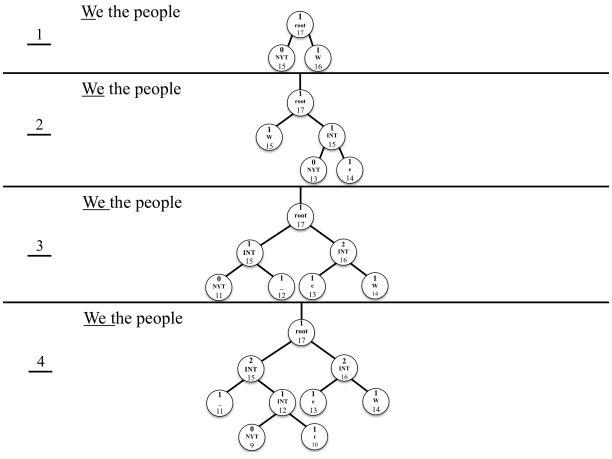


Fig. 3 – MATLAB Static Huffman: U.S Constitution

2.2. Adaptive Huffman

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*.



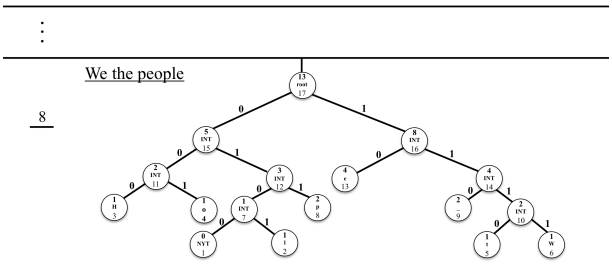


Fig. 4 – Adaptive Huffman tree construction: "We the people"

2.2.1. 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 children[2] 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:

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

Algorithm 3 *Adaptive Huffman coding*

procedure ADAPTIVE_HUFFMAN(F)
  $size = 2(\text{UNIQUE}(F)) + 1$
  $order = size$
  $root_{node} = (\text{'}root\text{-}node\text{'}, 0, size)$
  $H_{tree} = \{root_{node}\}$
  for all $char$ in F **do**
    if '$char$' $\notin H_{tree}.names$ **then**
      if $H_{tree}.length > 1$ **then**
        $INT_{node} \rightarrow \text{LEFT\_CHILD}(NYT)$
        $INT_{node} = (\text{'}INT\text{-}node\text{'}, this.node_{weight}, this.node_{order})$
        $NYT = \{(\text{'}char\text{'}, 1, order-1), (\text{'}NYT\text{-}node\text{'}, 0, order-2)\}$
        $H_{tree} = \{H_{tree}, NYT\}$
        $order = order - 2$
      else
        $NYT = \{(\text{'}char\text{'}, 1, order-1), (\text{'}NYT\text{-}node\text{'}, 0, order-2)\}$
        $H_{tree} = \{H_{tree}, NYT\}$
        $order = order - 2$
      end if
    else
      $p_{char} = H_{tree}(\text{'}char\text{'})$
      $LEAF_{node} \rightarrow H_{tree}(p_{char})$
      $LEAF_{node} = (\text{'}char\text{'}, this.node_{weight} + 1, this.node_{order})$
    end if
    H_{tree} = UPDATE($\{root_{node}, ..., NYT\}$)
  end for
  Return H_{tree}
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 2 – Example Codebook − Adaptive Huffman

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

2.2.2. Implementation

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

2.2.3. Encoding results

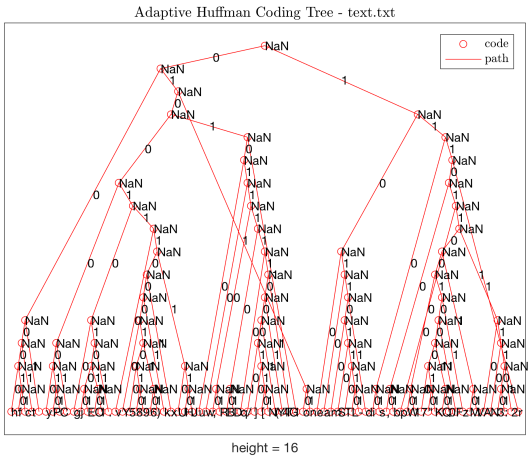


Fig. 5 – MATLAB Adaptive Huffman: U.S Constitution

TBD

2.3. LZ78

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.

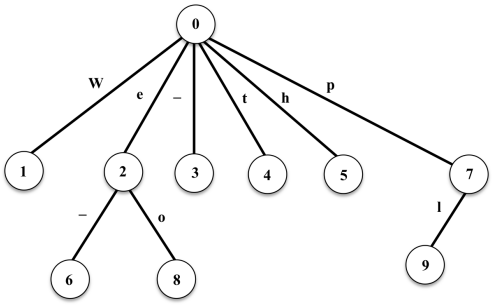


Fig. 6 – LZ78 tree construction: “We the people”

2.3.1. 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 $Dictionary(index) = (index, character)$, where *index* is the

index to a previous phrase, dictionary entry, and *character* is appended to the string represented by $Dictionary(index)$. 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 = {λ}
3:      index = 1
4:      prefix = λ
5:      while F ≠ ∅ do
6:          character = F(NEXT_character)
7:          if prefix + character ∈ Dictionary then
8:              prefix = prefix + character
9:          else
10:             if prefix = λ then
11:                 index = 0
12:             else
13:                 index = Dictionary(prefix)
14:                 entry = (index, character)
15:                 Dictionary = {(index, prefix + character)}
16:                 prefix = λ
17:             end if
18:         end if
19:     end while
20:     if prefix ≠ λ then
21:         index = Dictionary(prefix)
22:         entry = (index, )
23:     end if
24:     Return Dictionary
25: end procedure
```

Table 3 – 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

2.3.2. Implementation

The implementation of the LZ78 coding algorithm was done in MATLAB

2.3.3. Encoding results

TBD

2.4. Shannon-Fano-Elias

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 cod eword $c(x)$ consists of the first $\log\left(\frac{1}{p(x)}\right)+1$ bits of the binary expansion of the sum:

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

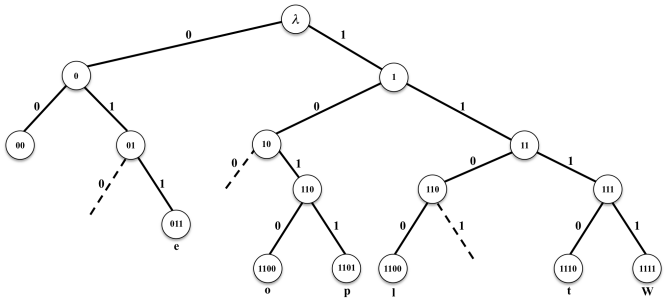


Fig. 7 – SFE tree construction: "We the people"

2.4.1. Algorithm overview

First, the SFE source encoder arranges the symbols in the alphabet $|\mathscr{X}|$ according to their decreasing probabilities $p(x) = p(x_1) > p(x_2) > ... > p(x_n)$. 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 $\overline{F}(x)$:

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

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*

procedure SHANNON_FANO_ELIAS($p(x), l(x)$)
  for each x in $p(x), l(x)$ **do**
    $p(x) = \text{SORT}(p(x))$
    $F(x) = \sum_{x_i \leq x} p(x)$
    $\overline{F}(x) = \sum_{j<x} \frac{1}{2} p(j) + p(x)$
    $binary_{rep} = \text{BINARY}\left(\overline{F}(x)\right)$
    $Codebook(x) = \text{CODE}(binary_{rep}, l(x))$
  end for
  Return *Codebook*
end procedure

Table 4 – Example Codebook − Shannon-Fano-Elias

x_i	c_i	$p(x)$	$F(x)$	$\overline{F}(x)$	$\overline{F}(x)_{binary}$	$l(x)$	*code*
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

2.4.2. Implementation

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

2.4.3. Encoding results

TBD