HANDBOOK ON ALGORITHMS AND THEORY OF COMPUTATION

Chapter 10

Text data compression algorithms

Contents

List of Figures

Chapter 10

Text data compression algorithms

MAXIME CROCHEMORE, Gaspard Monge Institute, University of Marne-la-Vallée, France
THIERRY LECROQ, Laboratoire d'Informatique de Rouen, University of Rouen, France

10.1 Text compression

The chapter describes a few algorithms that compress texts. Compression serves both to save storage space and to save transmission time. We shall assume that the text is stored in a file. The aim of compression algorithms is to produce a new file, as short as possible, containing the compressed version of the same text. Methods presented here reduce the representation of text without any loss of information, so that decoding the compressed text restores exactly the original data.

The term "text" should be understood in a wide sense. It is clear that texts can be written in natural languages or can be texts usually generated by translators (like various types of compilers). But texts can also be images or other kinds of structures as well provided the data are stored in linear files.

The interest in data compression techniques remains important even if mass storage systems improve regularly because the amount of data grows accordingly. Moreover, a consequence of the extension of computer networks is that the quantity of data they exchange grows exponentially. So, it is often necessary to reduce the size of files to reduce proportionally their transmission times.

The chapter contains three classical text compression algorithms. Variants of these algorithms are implemented in practical compression software, in which they are often combined together or with other elementary methods. Moreover, we present all-purpose methods, that is, methods in which no sophisticated modeling of the statistics of texts is done. An adequate modeling of a well-defined family of texts may increase significantly the compression when coupled with the coding algorithms of the chapter.

Compression ratio of the methods depend on the input data. However, most often, the size of compressed text vary from 30% to 50% of the size of the input. At the end of the chapter, we present ratios obtained by the methods on several example texts. Results of this type can be used to compare the efficiency of methods in compressing data. But, the efficiency of algorithms is also evaluated by their running times, and sometimes by the amount of memory space they require at run time. These elements are important criteria of choice when a compression algorithm is to be implemented in a telecommunication software.

Two strategies are applied to design the algorithms. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression (Sections 10.2 and 10.3). Section 10.4 presents a refinement of the coding algorithm of Huffman based on the binary representation of numbers. Huffman codes contain new codewords for the symbols occurring in the text. In this method fixed-length blocks of bits are encoded by different codewords. *A contrario* the second strategy encodes variable-length segments of the text (Section 10.5). To put it simply, the algorithm, while scanning the text, replaces some already read segments by just a pointer to their first occurrences. This second strategy often provides better compression ratios.

10.2 Static Huffman coding

The Huffman method is an optimal statistical coding. It transforms the original code used for characters of the text (ASCII code on 8 bits, for instance). Coding the text is just replacing each symbol (more exactly each occurrence of it) by its new **codeword**. The method works for any length of blocks (not only 8 bits), but the running time grows exponentially with the length. In the following, we assume that symbols are originally encoded on 8 bits to simplify the description.

The Huffman algorithm uses the notion of **prefix code**. A prefix code is a set of words containing no word that is a **prefix** of another word of the set. The advantage of such a code is that decoding is immediate. Moreover, it can be proved that this type of code does not weaken the compression.

A prefix code on the binary alphabet $\{0, 1\}$ corresponds to a binary tree in which the links from a node to its left and right children are labeled by 0 and 1 respectively. Such a tree is called a (digital) **trie**. Leaves of the trie are labeled by the original characters, and labels of branches are the words of the code (codewords of characters). Working with prefix code implies that codewords are identified with leaves only. Moreover, in the present method codes are complete: they correspond to complete tries, *i.e.* tree in which internal nodes have all exactly two children.

In the model where characters of the text are given new codewords, the Huffman algorithm builds a code that is optimal in the sense that the compression is the best possible. The length of the encoded text is minimum. The code depends on the input text, and more precisely on the frequencies of characters in the text. The most frequent characters are given short codewords while the least frequent symbols correspond to the longest codewords.

10.2.1 Encoding

The complete compression algorithm is composed of three steps: count of character frequencies, construction of the prefix code, encoding of the text. The last two steps use information computed by their preceding step.

The first step consists in counting the number of occurrences of each character in the original text (see Figure 10.1). We use a special end marker, denoted by `END`, which virtually appears only once at the end of the text. It is possible to skip this first step if fixed statistics on the alphabet are used. In this case however the method is optimal according to the statistics, but not necessarily for the specific text.

The second step of the algorithm builds the tree of a prefix code, called a Huffman tree, using the character frequency $freq(a)$ of each character a in the following way:

- create a one-node tree t for each character a, setting $weight(t) = freq(a)$ and $label(t) = a$,

```
H-COUNT (fin)
for each character a ∈ Σ
    do freq(a) ← 0
while not end of file fin and a is the next symbol
    do freq(a) ← freq(a) + 1
freq(END) ← 1
```

Figure 10.1: Counts the character frequencies.

```
H-BUILD-TREE
for each a ∈ Σ ∪ {END}
    do if freq(a) ≠ 0
          then create a new node t
               weight(t) ← freq(a)
               label(t) ← a
lleaves ← list of all created nodes in increasing order of weight
ltrees ← empty list
while LENGTH (lleaves) + LENGTH (ltrees) > 1
    do (l, r) ← extract two nodes of smallest weight (among the two nodes at the
             beginning of lleaves and the two nodes at the beginning of ltrees)
       create a new node t
       weight(t) ← weight(l) + weight(r)
       left(t) ← l
       right(t) ← r
       insert t at the end of ltrees
return t
```

Figure 10.2: Builds the Huffman coding tree.

```
H-Build-Code (t, length)
if t is not a leaf
   then temp[length] ← 0
        H-Build-Code (left(t), length + 1)
        temp[length] ← 1
        H-Build-Code (right(t), length + 1)
   else codeword(label(t)) ← temp[0 ... length − 1]
```

Figure 10.3: Builds character codewords from the coding tree.

```
H-Encode-Tree (fout, t)
if t is not a leaf
   then write a 0 in the file fout
        H-Encode-Tree (fout, left(t))
        H-Encode-Tree (fout, right(t))
   else write a 1 in the file fout
        write the original code of label(t) in the file fout
```

Figure 10.4: Stores the coding tree in the compressed file.

- repeat
  - Extract the two least weighted trees t_1 and t_2,
  - Create a new tree t_3 having left subtree t_1, right subtree t_2, and weight $weight(t_3) = weight(t_1) + weight(t_2)$,
- until only one tree remains.

The tree is constructed by the algorithm H-Build-Tree in Figure 10.2. The implementation uses two linear lists. The first list, *lleaves*, contains the leaves of the future tree associated each with a symbol. The list is sorted in increasing order of weights of leaves (frequencies of symbols). The second list, *ltrees*, contains the newly created trees. The operation of extracting the two least weighted trees is done by checking the two first trees of the list *lleaves* and the two first trees of the list *ltrees*. Each new tree is inserted at the end of the list of the trees. The only tree remaining at the end of the procedure is the coding tree.

After the coding tree is built, it is possible to recover the codewords associated with characters by a simple depth-first-search of the tree (see Figure 10.3); $codeword(a)$ denotes the binary codeword associated with the character a.

In the third step, the original text is encoded. Since the code depends on the original text, in order to be able to decode the compressed text, the coding tree and the original codewords of symbols must be stored together with the compressed text.

This information is placed in a header of the compressed file, to be read at decoding time just before the decompression starts. The header is written during a depth-first traversal of the

```
H-Encode-Text (fin, fout)
while not end of file fin and a is the next symbol
    do write codeword(a) in the file fout
write codeword(END) in the file fout
```

Figure 10.5: Encodes the characters in the compressed file.

```
H-Encoding (fin, fout)
H-Count (fin)
t ← H-Build-Tree
H-Build-Code (t, 0)
H-Encode-Tree (fout, t)
H-Encode-Text (fin, fout)
```

Figure 10.6: Complete function for Huffman encoding.

tree. Each time an internal node is encountered a 0 is produced. When a leaf is encountered a 1 is produced followed by the original code of the corresponding character on 9 bits (so that the end marker can be equal to 256 if all the 8-bit characters appear in the original text). This part of the encoding algorithm is shown in Figure 10.4.

After the header of the compressed file is made, the encoding of the original text is realized by the algorithm of Figure 10.5.

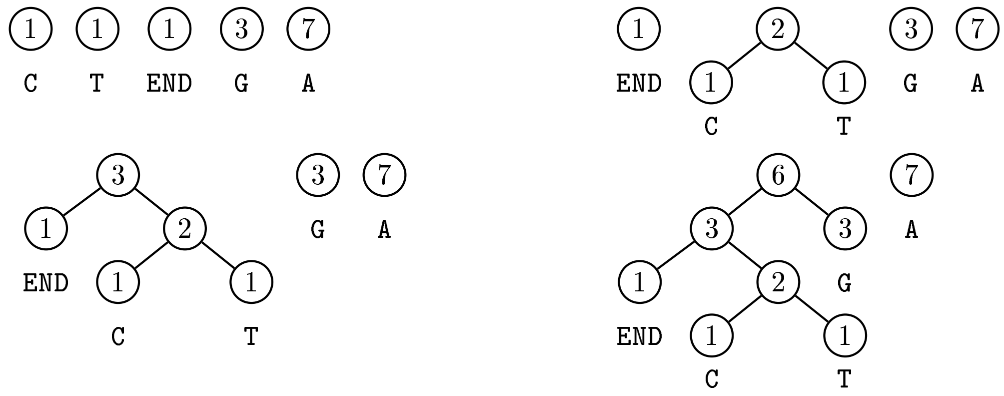
A complete implementation of the Huffman algorithm, composed of the three steps described above, is given in Figure 10.6.

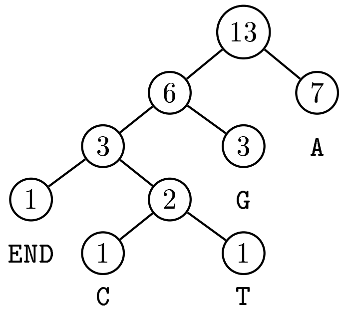
Example 10.1:
y = `ACAGAATAGAGA`
Length of $y = 12 \times 8 = 104$ bits (assuming an 8-bit code)

Character frequencies:

A	C	G	T	END
7	1	3	1	1

Different steps during the construction of the coding tree:





character codewords:

A	C	G	T	END
1	0010	01	0011	000

Encoded tree: `0001`$binary$(`END`,9)`01`$binary$(`C`,9)`1` $binary$(`T`,9)`1`$binary$(`G`,9)`1`$binary$(`A`,9),
which produces a header of length 54 bits:
`0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001`

Encoded text:
$\underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{0010}}_{\texttt{C}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{01}}_{\texttt{G}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{0011}}_{\texttt{T}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{01}}_{\texttt{G}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{01}}_{\texttt{G}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{000}}_{\texttt{END}}$
of length 24 bits

Total length of the compressed file: 78 bits. □

The construction of the tree takes $O(\sigma \log \sigma)$ time if the sorting of the list of the leaves is implemented efficiently. The rest of the encoding process runs in time linear in the sum of the sizes of the original and compressed texts.

```
H-REBUILD-TREE (fin, t)
read bit b from fin
if b = 1
   then make t a leaf
        label(t) ← symbol corresponding to the 9 next bits read from fin
   else create a new node l
        left(t) ← l
        H-REBUILD-TREE (fin, l)
        create a new node r
        right(t) ← r
        H-REBUILD-TREE (fin, r)
```

Figure 10.7: Rebuilds the tree from the header of compressed file.

```
H-DECODE-TEXT (fin, fout, root)
t ← root
while label(t) ≠ END
   do if t is a leaf
      then write label(t) in the file fout
           t ← root
      else read bit b from fin
           if b = 1
              then t ← right(t)
              else t ← left(t)
```

Figure 10.8: Recovers the original text.

H-DECODING (*fin*, *fout*)
1 create a new node *root*
2 H-REBUILD-TREE (*fin*, *root*)
3 H-DECODE-TEXT (*fin*, *fout*, *root*)

Figure 10.9: Complete function for Huffman decoding.

10.2.2 Decoding

Decoding a file containing a text compressed by Huffman algorithm is a mere programming exercise. First, the coding tree is rebuilt by the algorithm of Figure 10.7. Then, the original text is recovered by parsing the compressed text with the coding tree. The process begins at the root of the coding tree, and follows a left edge when a 0 is read or a right edge when a 1 is read. When a leaf is encountered, the corresponding character (in fact the original codeword of it) is produced and the parsing resumes at the root of the tree. The process ends when the codeword of the end marker is encountered. An implementation of the decoding of the text is presented in Figure 10.8.

The complete decoding program is given in Figure 10.9. It calls the preceding functions. The running time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

10.3 Dynamic Huffman coding

The two main drawbacks of the static Huffman method are: first, if the frequencies of characters the source text are not known *a priori*, the source text has to be read twice; second, the coding tree must be included in the compressed file. This is avoided by a dynamic method where the coding tree is updated each time a symbol is read from the source text. The current tree is a Huffman tree related to the part of the text that is already treated. The tree evolves exactly in the same way during the decoding process. The efficiency of the method is based on a characterization of Huffman trees, known as the *siblings property*.

Siblings property: Let T be a Huffman tree with n leaves (a complete binary weighted tree built by the procedure H-BUILD-TREE in which all leaves have positive weights). Then the nodes of T can be arranged in a sequence $(x_0, x_1, \ldots, x_{2n-2})$ such that:

1. the sequence of weights $(weight(x_0), weight(x_1), \ldots, weight(x_{2n-2}))$ is in decreasing order;
2. for any i $(0 \leq i \leq n-2)$, the consecutive nodes x_{2i+1} and x_{2i+2} are siblings (they have the same parent).

The compression and decompression processes initialize the dynamic Huffman tree by a one-node tree that correspond to an artificial character, denoted by `ART`. The weight of this single node is 1.

DH-INIT

$root \leftarrow 0$
$child(root) \leftarrow$ `UNDEFINED`
$weight(root) \leftarrow 1$
$parent(root) \leftarrow$ `UNDEFINED`
for each letter $a \in \Sigma \cup \{$`END`$\}$
  do $leaf[a] \leftarrow$ `UNDEFINED`
$leaf[$`ART`$] \leftarrow root$

Figure 10.10: Initializes the dynamic Huffman tree.

10.3.1 Encoding

Each time a symbol a is read from the source text, its codeword in the tree is sent. However this happens only if a appeared previously. Otherwise the code of `ART` is sent followed by the original codeword of a. Afterwards, the tree is modified in the following way: first, if a never occurred before, a new internal node is created and its two children are a new leaf labeled by a and the leaf `ART`; then, the tree is updated (see below) to get a Huffman tree for the new prefix of text.

Implementation

Each node is identified with an integer n, the root is the integer 0. The invariant of compression and decompression algorithms is that, if the tree has m nodes, the sequence of nodes $(m-1,\ldots,1,0)$ satisfies the siblings property. The tree is stored in a table, and we use the next notations, for a node n:

- $parent(n)$ is the parent of n ($parent(root) =$ `UNDEFINED`),
- $child(n)$ is the left child of n (if n is an internal node, otherwise $child(n) =$ `UNDEFINED`), and $child(n)+1$ is its right child (this is useful only at decoding time),
- $label(n)$ is the symbol associated with n when n is a leaf,
- $weight(n)$ is the weight of n (it is the frequency of $label(n)$ if n is a leaf).

Indeed, the child link is useful only at decoding time so that the implementation may differ between the two phases of the algorithm. But, to simplify the description and give a uniform treatment of the data structure, we assume the same implementation during the encoding and the decoding steps. Weights of nodes are handled by the procedure DH-UPDATE.

The correspondence between symbols and leaves of the tree is done by a table called *leaf*: for each symbol $a \in \Sigma \cup \{$`END`$\}$ $leaf[a]$ is the corresponding leaf of the tree, if any.

The coding tree initially contains only one node labeled by symbol `ART`. The initialization is given in Figure 10.10.

Encoding the source text is a succession of three steps: read a symbol a from the source text, encode the symbol a according to the current tree, update the tree. It is described in Figure 10.11.

Encoding a symbol a already encountered consists in accessing its associated leaf $leaf[a]$, then computing its codeword by a bottom-up walk in the tree (following the *parent* links up to the root). Each time a node n ($\neq$ root) is encountered if n is odd a 1 is sent (n is the right

```
DH-Encoding (fin, fout)
DH-Init
while not end of file fin and a is the next symbol
   do   DH-Encode-Symbol (a, fout)
        DH-Update (a)
DH-Encode-Symbol (END, fout)
```

Figure 10.11: Complete function for dynamic Huffman encoding.

```
DH-Encode-Symbol (a, fout)
S ← empty stack
n ← leaf[a]
if n = UNDEFINED
   then n ← leaf[ART]
while n ≠ root
   do   if n is odd
           then Push (S, 1)
           else Push (S, 0)
        n ← parent(n)
Send (S, fout)
if leaf[a] = UNDEFINED
   then write in fout the original codeword of a on 9 bits
        DH-Add-Node (a)
```

Figure 10.12: Encodes one symbol.

```
DH-Add-Node (a)
transform leaf[ART] into
   an internal node of weight 1 with
      a left child of weight 0 for leaf[a],
      a right child of weight 1 for leaf[ART].
```

Figure 10.13: Adds a new symbol in the tree.

```
DH-Decoding (fin, fout)
DH-Init
a ← DH-Decode-Symbol (fin)
while a ≠ END
   do    write a in fout
         DH-Update (a)
         a ← DH-Decode-Symbol (fin)
```

Figure 10.14: Complete function for dynamic Huffman decoding.

```
DH-Decode-Symbol (fin)
n ← root
while child(n) ≠ UNDEFINED
   do read bit b from fin
         n ← child(n) + b
a ← label(n)
if a = ART
   then a ← symbol corresponding to the next 9 bits read from fin
         DH-Add-Node (a)
return (a)
```

Figure 10.15: Decodes one symbol from the input file.

child of its parent), and if n is even a 0 is sent (n is then the left child of its parent). As the codeword of a is read in reverse direction a stack S is used to temporarily store the bits and send them properly. The procedure Send (S, *fout*) send the bits of the stack S in the correct order to the file *fout*.
If a has not been encountered yet, then the code of `ART` is sent followed by the original codeword of a on 9 bits, and a new leaf is created for a (see Figures 10.12 and 10.13).

10.3.2 Decoding

At decoding time the compressed text is parsed with the coding tree. The current node is initialized with the root like in the encoding algorithm, and then the tree evolves symmetrically. Each time a 0 is read from the compressed file the walk down the tree follows the left link, and it follows the right link if a 1 is read. When the current node is a leaf, its associated symbol is written in the output file and the tree is updated exactly as it is done during the encoding phase.

Implementation

As for the encoding process the current Huffman tree is stored in a table, and it is initialized with the artificial symbol `ART`. The same elements of the data structure are used during the decompression. Note that the next node when parsing a bit b from node n is just $child(n) + b$

```
DH-UPDATE (a)
n ← leaf[a]
while n ≠ root
   do   weight(n) ← weight(n) + 1
        m ← n
        while weight(m − 1) < weight(n)
           do   m ← m − 1
        DH-SWAP-NODES (m, n)
        n ← parent(m)
weight(root) ← weight(root) + 1
```

Figure 10.16: Updates the current Huffman tree.

with the convention on left-right links and 0-1 bits. The tree is updated by the procedure DH-UPDATE used previously and that is described in the next section. Figures 10.14 and 10.15 display the decoding mechanism.

10.3.3 Updating

During encoding and decoding phases the current tree has to be updated to take into account the correct frequency of symbols. When a new symbol is considered the weight of its associated leaf is incremented by 1, and the weights of ancestors have to be modified correspondingly. The procedure that realizes the operation is shown in Figures 10.16. Its proof of correctness is based on the siblings property.

We explain how the procedure DH-UPDATE works. First, the weight of the leaf n corresponding to a is incremented by 1. Then, if point 1 of the siblings property is no longer satisfied, node n is exchanged with the closest node m ($m < n$) in the list such that $weight(m) < weight(n)$. Doing so, the nodes remain in decreasing order of their weights. It is important here that leaves have positive weights because this guarantees that m is not a parent nor an ancestor of node n. Afterwards, the same operation is repeated on the parent of n until the root of the tree is reached.

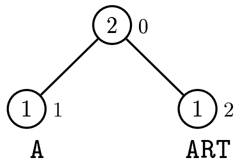
The aim of procedure DH-SWAP-NODES used in Figures 10.16 is to exchange the subtrees rooted at its input nodes m and n. In concrete terms, this remains to exchange the records stored at the two nodes in the table. It is meant that nothing is to be done if $m = n$.

Example 10.2:
$y =$ `ACAGAATAGAGA`

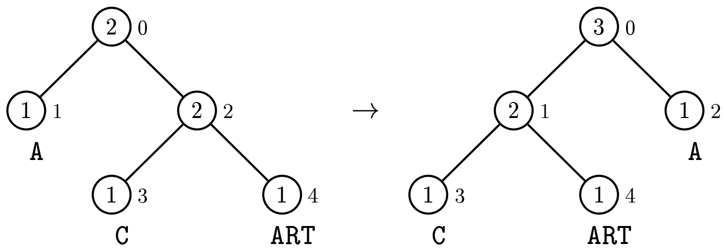
Initial tree:

1 0
ART

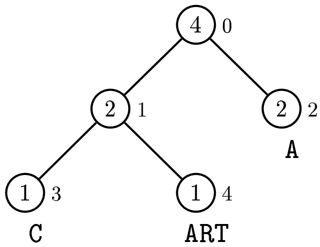
Next symbol is `A`:
The ASCII code of `A` is sent on 9 bits;
Bits sent: `001000001`



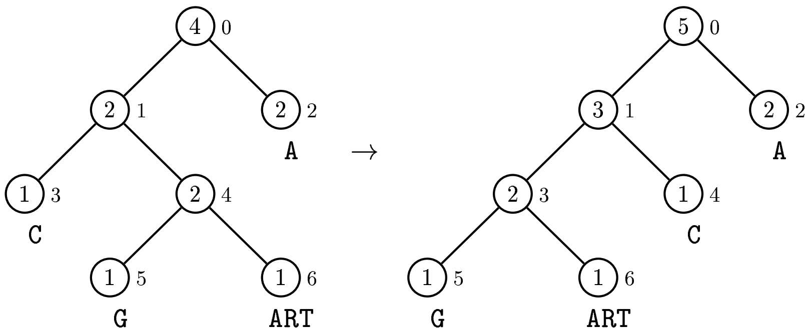
Next symbol is `C`:
The code of `ART` is sent followed by the ASCII code of `C` on 9 bits;
Bits sent: `1 001000011`
Nodes 1 and 2 are swapped.



Next symbol is `A`:
The code of `A` is sent;
Bit sent: `1`



Next symbol is `G`:
The code of `ART` is sent followed by the ASCII code of `G` on 9 bits;
Bits sent: `01 001000111`
Nodes 3 and 4 are swapped.



Finally, the entire sequence of bits sent is the following:

```
AR-Encode (fin)
  l ← 0
  h ← 1
  while not not end of file fin and a_i is the next symbol
     do   l ← l + (h − l) * l_i
          h ← l + (h − l) * h_i
  return(l)
```

Figure 10.17: Basic arithmetic encoding.

$$\underbrace{\texttt{001000001}}_{\texttt{A}}\ \underbrace{\texttt{1}}_{\texttt{ART}}\ \underbrace{\texttt{001000011}}_{\texttt{C}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{01}}_{\texttt{ART}}\ \underbrace{\texttt{001000111}}_{\texttt{G}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{1}}_{\texttt{A}}\ \underbrace{\texttt{101}}_{\texttt{ART}}$$

$$\underbrace{\texttt{001010100}}_{\texttt{T}}\ \underbrace{\texttt{0}}_{\texttt{A}}\ \underbrace{\texttt{100}}_{\texttt{G}}\ \underbrace{\texttt{0}}_{\texttt{A}}\ \underbrace{\texttt{100}}_{\texttt{G}}\ \underbrace{\texttt{0}}_{\texttt{A}}\ \underbrace{\texttt{111}}_{\texttt{ART}}\ \underbrace{\texttt{100000000}}_{\texttt{END}}$$

The total length of the compressed text is 66. □

10.4 Arithmetic coding

10.4.1 Encoding

The basic idea of arithmetic coding is to consider symbol as digits of a numeration system, and texts as decimal parts of numbers between 0 and 1. The length of the interval attributed to a digit (it is 0.1 for digits in the usual base 10 system) is made proportional to the frequency of the digit in the text. The encoding is thus assimilated to a change in the base of a numeration system. To cope with precision problems, the number corresponding to a text is handled via a lower bound and an upper bound, which remains to associate with a text a sub-interval of $[0,1[$. The compression comes from the fact that large intervals require less precision to separate their bounds.

More formally, the interval associated with each symbol $a_i \in \Sigma$ $(1 \leq i \leq \sigma)$ is denoted $I(a_i) = [l_i, h_i[$. The intervals satisfy the conditions: $l_1 = 0$, $h_\sigma = 1$, and $l_i = h_{i-1}$ for $1 < i \leq \sigma$. Note that $I(a_i) \cap I(a_j) = \emptyset$ if $a_i \neq a_j$.

The encoding phase consists in computing the interval corresponding to the input text. The basic step that deals with a symbol a_i of the source text transforms the current interval $[l,h[$ into $[l',h'[$ where $l' = l + (h-l) * l_i$ and $h' = l + (h-l) * h_i$, starting with the initial interval $[0,1[$ (see Figure 10.17). Indeed, in a theoretical approach, l only is needed to encode the input text.

Example 10.3:
$\Sigma = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$, $\sigma = 4$

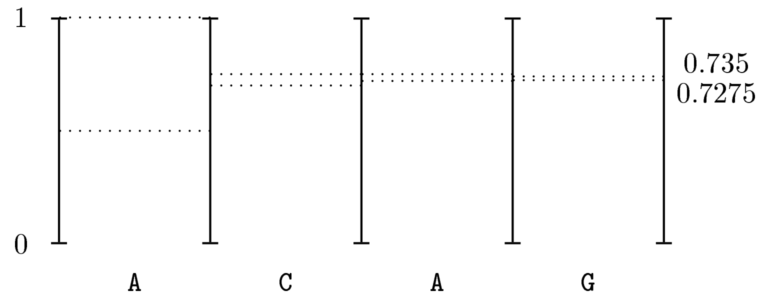
$$I(\texttt{A}) = [0.5, 1[,\quad I(\texttt{C}) = [0.4, 0.5[,\quad I(\texttt{G}) = [0.1, 0.4[,\quad I(\texttt{T}) = [0, 0.1[$$

Encoding `ACAG` gives:

```
AR-Decode (l, fout)
while l ≠ 0
    do  find a_i such that l ∈ I(a_i)
        write a_i in file fout
        l ← (l − l_i)/(h_i − l_i)
```

Figure 10.18: Basic arithmetic decoding.

symbol	l	h
	0	1
`A`	0.5	1
`C`	0.7	0.75
`A`	0.725	0.75
`G`	0.7275	0.735



□

10.4.2 Decoding

The reverse operation of decoding a text compressed by the previous algorithm theoretically requires only the lower bound l of the interval. Decoding the number l is done as follows: first find the symbol a_i such that $l \in I(a_i)$, produced the symbol a_i, and then replace l by l' defined by:

$$l' \leftarrow \frac{l - l_i}{h_i - l_i}.$$

The same process is repeated until $l = 0$ (see Figure 10.18). Indeed, the implementation of the decoding phase simulates what is done on the current interval considered at encoding time.

Example 10.4:
$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $\sigma = 4$

$$I(\mathtt{A}) = [0.5, 1[, \quad I(\mathtt{C}) = [0.4, 0.5[, \quad I(\mathtt{G}) = [0.1, 0.4[, \quad I(\mathtt{T}) = [0, 0.1[$$

Decoding $l = 0.7275$:

AR-ENCODING (*fin*, *fout*)
Initialize tables *freq* and *cum-freq*
$l \leftarrow 0$
$h \leftarrow 2^N - 1$
waiting-counter $\leftarrow 0$
while not end of file *fin* **and** a_i is the next symbol
  do AR-ENCODE-SYMBOL (a_i)
    Update tables *freq* and *cum-freq*
    Maintain symbols in decreasing order of *cum-freq*
AR-ENCODE-SYMBOL (`END`)
waiting-counter $\leftarrow$ *waiting-counter* $+ 1$
AR-SEND-BIT (leftmost bit of h)

Figure 10.19: Complete arithmetic encoding function.

l	a_i
0.7275	`A`
0.455	`C`
0.55	`A`
0.1	`G`
0	

□

10.4.3 Implementation

The main problem when implementing the arithmetic coding compression algorithm is to cope with precision on real numbers operations. The $[0, 1[$ interval of real numbers is substituted by the interval of integers $[0, 2^N - 1[$, where N is a fixed integer.

So, the algorithms work with integral values of size N. During the process, each time the binary representation of bounds l and h have a common prefix this prefix is sent and l is shifted to the left and filled with `0`'s while h is shifted to the left and filled with `1`'s.

The intervals associated with symbols of the alphabet are computed with the help of symbol frequencies, in a dynamic way: each character frequency is initialized with 1 and is incremented each time the symbol is encountered.

We denote by $\mathit{freq}[i]$ the frequency of symbol a_i of the alphabet. We also consider the cumulative frequency of symbols, and set $\mathit{cum\text{-}freq}[i] = \sum_{j=i+1}^{\sigma} \mathit{freq}[j]$. Then $\mathit{cum\text{-}freq}[0]$ is the cumulative frequency of all symbols. Note that $\mathit{cum\text{-}freq}[0] - \sigma - 1$ is the length of the prefix of the input scanned so far. The symbols are maintained in decreasing order of their frequencies. This obviously save on the expected number of operations to update the table *cum-freq*.

Then when a symbol a_i is read from the source text, the current interval $[l, h[$ is updated in the following way:

$$l \leftarrow l + \frac{(h - l + 1) * \mathit{cum\text{-}freq}[i]}{\mathit{cum\text{-}freq}[0]}$$

```
AR-Encode-Symbol (a_i)
l ← l + ((h − l + 1) * cum-freq[i])/cum-freq[0]
h ← l + ((h − l + 1) * cum-freq[i − 1])/cum-freq[0] − 1
repeat
    if leftmost bit of l = leftmost bit of h
        then AR-Send-Bit (leftmost bit of h)
                l ← 2 * l
                h ← 2 * h + 1
        else  if h − l < cum-freq[0]
                    then l ← 2 * (l − 2^(N−2))
                         h ← 2 * (h − 2^(N−2)) + 1
                         waiting-counter ← waiting-counter + 1
until leftmost bit of l ≠ leftmost bit of h and h − l ≥ cum-freq[0]
```

Figure 10.20: Encodes one symbol.

```
AR-Send-Bit (bit, fout)
write bit in fout
while waiting-counter > 0
    do    if bit = 0
             then write 1 in fout
                  write 0 in fout
          waiting-counter ← waiting-counter − 1
```

Figure 10.21: Sends one bit followed by the waiting bits, if any.

$$h \leftarrow l + \frac{(h-l+1) * \textit{cum-freq}[i-1]}{\textit{cum-freq}[0]} - 1$$

The common prefix (if any) of l and h is sent and l and h are shifted and filled (respectively with 0's and 1's). At this point, if the interval is too short (if $h-l < \textit{cum-freq}[0]$) it is extended to $[2*(l-2^{N-2}), 2*(h-2^{N-2})+1[$ and a waiting counter is incremented by 1. And the same operation is repeated as long as the interval is too short. After that, when a bit is sent, the reverse bit is sent the number of times indicated by the waiting counter.

Example 10.5:
$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $N = 8$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	5	4	3	2	1	0
freq	0	1	1	1	1	1

$[0,255[\xrightarrow{\mathtt{A}} [0 + \frac{(255-0+1)*4}{5}, 0 + \frac{(255-0+1)*5}{5} - 1[= [204,255[= [\mathtt{11001100}_2, \mathtt{11111111}_2[$
`11` is sent and $[\mathtt{00110000}_2, \mathtt{11111111}_2[= [48,255[$ is the next interval;

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	6	4	3	2	1	0
freq	0	2	1	1	1	1

$[48,255[\xrightarrow{\mathtt{C}} [96,231[= [\mathtt{10011000}_2, \mathtt{10111001}_2[$
`10` is sent and $[\mathtt{01100000}_2, \mathtt{11100111}_2[= [96,231[$ is the next interval;

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	7	5	3	2	1	0
freq	0	2	2	1	1	1

$[96,231[\xrightarrow{\mathtt{A}} [193,231[= [\mathtt{11000001}_2, \mathtt{11100111}_2[$
`11` is sent and $[\mathtt{00000100}_2, \mathtt{10011111}_2[= [4,159[$ is the next interval;

Next symbol is `G`: `001` is sent;
Next symbol is `A`: `1` is sent;
Next symbol is `A`: `1` is sent;
Next symbol is `T`: `0111` is sent;
Next symbol is `A`: `10` is sent;
Next symbol is `G`: nothing is sent and the current interval becomes $[111,139[$;

$[111,139[\xrightarrow{\mathtt{A}} [127,139[= [\mathtt{01111111}_2, \mathtt{10001011}_2[$, nothing is sent
The interval is too short and replaced by $[2*(127-2^{N-2}), 2*(139-2^{N-2})+1[= [126,151[$ and one bit is waiting, $[\mathtt{01111110}, \mathtt{10010111}[= [126,151[$ is the next interval;

$[126,151[\xrightarrow{\mathtt{G}} [134,138[= [\mathtt{10000110}_2, \mathtt{10001010}_2[$
`1+0+000` are sent and $[\mathtt{01100000}_2, \mathtt{10101111}_2[= [96,175[$ is the next interval;

```
AR-DECODING (fin, fout)
Init the tables freq and cum-freq
value ← 0
for i ← 1 to N
    do  read bit b from fin
        value ← 2 * value + b
l ← 0
h ← 2^N − 1
repeat
    i ← AR-DECODE-SYMBOL (fin)
    if a_i ≠ END
       then write a_i in fout
            Update the tables freq and cum-freq
            Maintain symbols in decreasing order of cum-freq
until a_i = END
```

Figure 10.22: Complete arithmetic decoding function.

$[96, 175[\xrightarrow{\texttt{A}} [141, 175[= [\texttt{10001101}_2, \texttt{10101111}_2[$
`10` is sent and $[\texttt{00110100}_2, \texttt{10111111}_2[= [52, 191[$ is the next interval;

$[52, 191[\xrightarrow{\texttt{END}} [52, 59[= [\texttt{00110100}_2, \texttt{00111011}_2[$
`0011` is sent;

`1+0` are sent in order to finish the encoding process. □

The decoding process is exactly the reverse of the coding process. It uses a window of size N on the compressed file. First, the window is filled with the first N bits of the compressed file and *value* is the corresponding base 2 number. The current interval is initialized with $l = 0$ and $h = 2^N - 1$.

Then, the symbol a_i to be produced is the first character such that:

$$\mathit{cum\text{-}freq}[i] > \frac{(\mathit{value} - l + 1) * \mathit{cum\text{-}freq}[0] - 1)}{h - l + 1},$$

and l and h are then updated exactly in the same way than during the coding process. If the binary representations of l and h have a common prefix of length p they are both shifted p binary position to the left and l is filled by `0`'s, h is filled with `1`'s. The window on the compressed file is shifted p symbols to the right and the variable *value* is updated correspondingly. The tables *freq* and *cum-freq* are updated and the symbols are maintained in decreasing order of the frequencies as in the coding process.

This is repeated until the symbol `END` is produced.

Example 10.6:
Decoding the text `11101100111011110100001000111 0`

```
AR-Decode-Symbol (fin)
cum ← ((value − l + 1) ∗ cum-freq[0] − 1)/(h − l + 1)
i ← 1
while cum-freq[i] > cum
   do   i ← i + 1
l ← l + ((h − l + 1) ∗ cum-freq[i])/cum-freq[0]
h ← l + ((h − l + 1) ∗ cum-freq[i − 1])/cum-freq[0] − 1
repeat
   if leftmost bit of l = leftmost bit of h
      then l ← 2 ∗ l
           h ← 2 ∗ h + 1
           read bit b from fin
           value ← 2 ∗ value + b
      else if h − l < cum-freq[0]
              then l ← 2 ∗ (l − 2^(N−2))
                   h ← 2 ∗ (h − 2^(N−2)) + 1
                   read bit b from fin
                   value ← 2 ∗ (value − 2^(N−2)) + b
until leftmost bit of l ≠ leftmost bit of h and h − l ≥ cum-freq[0]
return i
```

Figure 10.23: Decodes one symbol.

$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	5	4	3	2	1	0
freq	0	1	1	1	1	1

$value = 236 = \mathtt{11101100}_2$
$cum = 4$
$[0, 255[\xrightarrow{\mathtt{A}} [204, 255[= [\mathtt{11001100}_2, \mathtt{11111111}_2[$: shift by 2
The next interval is $[\mathtt{00110000}_2, \mathtt{11111111}_2[$ and $value = \mathtt{10110011}_2 = 179$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	6	4	3	2	1	0
freq	0	2	1	1	1	1

$value = 179 = \mathtt{10110011}_2$
$cum = 3$
$[48, 255[\xrightarrow{\mathtt{C}} [152, 185[= [\mathtt{10011000}_2, \mathtt{10111001}_2[$: shift by 2
The next interval is $[\mathtt{01100000}_2, \mathtt{11100111}_2[$ and $value = \mathtt{11001110}_2 = 206$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	7	5	3	2	1	0
freq	0	2	2	1	1	1

$value = 206 = \mathtt{11001110}_2$
$cum = 5$
$[96, 231[\xrightarrow{\mathtt{A}} [193, 231[= [\mathtt{11000001}_2, \mathtt{11100111}_2[$: shift by 2
The next interval is $[\mathtt{00000100}_2, \mathtt{10011111}_2[$ and $value = \mathtt{00111011}_2 = 59$

Next symbols are **GAATAG** and the current interval becomes $[111, 139[$ and $value = 132 = \mathtt{10000100}_2$

$cum = 10$
$[111, 139[\xrightarrow{\mathtt{A}} [127, 139[= [\mathtt{01111111}_2, \mathtt{10001011}_2[$: no shift
The interval is too short and is replaced by $[\mathtt{01111110}_2, \mathtt{10010111}_2[$ and $value = \mathtt{10001000}_2 = 136$

$value = 136 = \mathtt{10001000}_2$
$cum = 6$
$[126, 151[\xrightarrow{\mathtt{G}} [134, 138[= [\mathtt{10000110}_2, \mathtt{10001010}_2[$: shift by 4
The next interval $[\mathtt{01100000}_2, \mathtt{10101111}_2[$ and $value = \mathtt{10001110}_2 = 142$

$value = 142 = \mathtt{10001110}_2$
$cum = 9$
$[96, 175[\xrightarrow{\mathtt{A}} [141, 175[= [\mathtt{10001101}_2, \mathtt{10101111}_2[$: shift by 2
The next interval $[\mathtt{00110100}_2, \mathtt{10111111}_2[$ and $value = \mathtt{00111000}_2 = 56$

$value = 56 = \texttt{00111000}_2$
$cum = 0$
The symbol is `END`, the decoding process is over. □

Maintaining the symbols in decreasing order of frequencies can be done in $O(\log \sigma)$ using a suitable data structure.

10.5 LZW Coding

Ziv and Lempel designed a compression method using encoding **segments**. These segments of the original text are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the text it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv-Lempel compression scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols).

The dictionary is the central point of the algorithm. It has the property of being prefix-closed (every prefix of a word of the dictionary is in the dictionary), so that it can be implemented efficiently as a trie. Furthermore, a hashing technique makes its implementation efficient. The version described in this section is called the Lempel-Ziv-Welsh method after several improvements introduced by Welsh. The algorithm is implemented by the `compress` command existing under the UNIX operating system.

10.5.1 Encoding

We describe the scheme of the coding method. The dictionary is initialized with all strings of length 1, the characters of the alphabet. The current situation is when we have just read a segment w of the text. Let a be the next symbol (just following the given occurrence w). Then we proceed as follows:

- If wa is not in the dictionary, we write the index of w in the output file, and add wa to the dictionary. We then reset w to a and process the next symbol (following a).
- If wa is in the dictionary we process the next symbol, with segment wa instead of w.

Initially, the segment w is set to the first symbol of the source text, so that it is clear that "w belongs to the dictionary" is an invariant of the operation described above.

Example 10.7:
The alphabet is the 8-bit ASCII alphabet, $y =$ `ACAGAATAGAGA`
The dictionary initially contains the ASCII symbols, their indices are their ASCII codewords.

A	C	A	G	A	A	T	A	G	A	G	A		w	written	added
	↑												A	65	AC, 257
		↑											C	67	CA, 258
			↑										A	65	AG, 259
				↑									G	71	GA, 260
					↑								A	65	AA, 261
						↑							A	65	AT, 262
							↑						T	84	TA, 263
								↑					A		
									↑				AG	259	AGA, 264
										↑			A		
											↑		AG		
												↑	AGA	264	
														256	

□

10.5.2 Decoding

The decoding method is symmetric to the coding algorithm. The dictionary is recovered while the decompression process runs. It is basically done in this way:

- read a code c in the compressed file,
- write in the output file the segment w having index c in the dictionary,
- add the word wa to the dictionary where a is the first letter of the next segment.

In this scheme, the dictionary is updated after the next segment is decoded because we need its first symbol a to concatenate at the end of the current segment w. So, a problem occurs if the next index computed at encoding time is precisely the index of the segment wa. Indeed, this happens only in a very special case, in which the symbol a is also the first symbol of w itself. This arises if the rest of the text to encode starts with a segment $azazax$ (a a symbol, z a string) for which az belongs to the dictionary but aza does not. During the compression process the index of az is output, and aza is added to the dictionary. Next, aza is read and its index is output. During the decompression process the index of aza is read while the first occurrence of az has not been completed yet, the segment aza is not already in the dictionary. However, since this is the unique case where the situation occurs, the segment aza is recovered by taking the last segment az added to the dictionary concatenated with its first letter a.

Example 10.8:
Decoding the sequence 65, 67, 65, 71, 65, 65, 84, 259, 264, 256
The dictionary initially contains the ASCII symbols, their indices are their ASCII codewords.

read	written	added
65	`A`	
67	`C`	`AC`, 257
65	`A`	`CA`, 258
71	`G`	`AG`, 259
65	`A`	`GA`, 260
65	`A`	`AA`, 261
84	`T`	`AT`, 262
259	`AG`	`TA`, 263
264	`AGA`	`AGA`, 264
256		

The critical situation occurs when reading the index 264 because, at that moment, no word of the dictionary has this index. □

10.5.3 Implementation

We describe how the dictionary, which is the main data structure of the method, can be implemented. It is natural to consider two implementations adapted for the two phases respectively because the dictionary is not manipulated in the same manner during these phases. They have in common a dictionary implemented as a trie stored in a table D. A node p of the trie is just an index on the table D. It has the three following components:

- $parent(p)$, a link to the parent node of p,
- $label(p)$, a character,
- $code(p)$, the codeword (index in the dictionary) associated with p.

During the compression algorithm shown in Figure 10.24, for a node p we need to compute its child according to some letter a. This is done with by hashing, with a hashing function defined on pairs in the form (p, a). This provides a fast access to the children of a node.

The function HASH-SEARCH, with input $(D, (p, a))$, returns the node q such that $parent(q) = p$ and $label(q) = a$, if such a node exists and `NIL` otherwise. The procedure HASH-INSERT, with input $(D, (p, a, c))$, inserts a new node q in the dictionary D with $parent(q) = p, label(q) = a$ and $code(q) = c$.

For the decompression algorithm, no hashing technique is necessary on the table representation of the trie that implements the dictionary. Having the index of the next segment, a bottom-up walk in the trie produces the mirror image of the expected segment. A stack is then used to reverse it. We assume that the function $string(c)$ performs this specific work for a code c. The bottom-up walk follows the parent links of the data structure. The function $first(w)$ gives the first character of the word w. These features are part of the decompression algorithm displayed in Figure 10.25.

The Ziv-Lempel compression and decompression algorithms run both in time linear in the sizes of the files provided the hashing technique is implemented efficiently. Indeed, it is very fast in practice, except when the table becomes full and should be reset. In this situation, usual implementations also reset the whole dictionary to its initial value.

The main advantage of Ziv-Lempel compression method, compared to Huffman coding, is that it captures long repeated segments in the source file and thus often yields better compression ratios.

```
LZW-Coding (fin, fout)
count ← −1
for each character a ∈ Σ
    do count ← count + 1
        Hash-Insert (D, (−1, a, count))
count ← count + 1
Hash-Insert (D, (−1, END, count))
p ← −1
while not end of file fin and a is the next symbol
        q ← Hash-Search (D, (p, a))
        if q = NIL
            then write code(p) on 1 + log(count) bits in fout
                    count ← count + 1
                    Hash-Insert (D, (p, a, count))
                    p ← Hash-Search (D, (−1, a))
            else  p ← q
write p on 1 + log(count) bits in fout
write code(Hash-Search (D, (−1, END)) in 1 + log(count) bits in fout
```

Figure 10.24: LZW encoding algorithm.

10.6 Experimental results

The table of Figure 10.26 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts. The table is extracted from (Zipstein, 1992).

The source files are: French text, C sources, Alphabet, and Random. Alphabet is a file containing a repetition of the line `abc...zABC...Z`. Random is a file where the symbols have been generated randomly, all with the same probability and independently of each others.

The compression algorithms reported in the table are: the Huffman algorithm of Section 10.2, the Ziv-Lempel algorithm of Section 10.5, and a third algorithm called Factor. This latter algorithm encodes segments of the source text as Ziv-Lempel algorithm does. But the segments are taken among all segments already encountered in the text before the current position. The method gives usually better compression ratio but is more difficult to implement. Compression based on arithmetic as presented in the chapter gives compression ratios slightly better than Huffman coding.

The table of Figure 10.26 gives in percentage the sizes of compressed files. Results obtained by Ziv-Lempel and Factor algorithms are similar. Huffman coding gives the best result for the Random file. Finally, experience shows that exact compression methods often reduce the size of data to 30%–50% of their original size.

10.7 Research Issues and Summary

The statistical compression algorithm is from Huffman (1951). The UNIX command `pack` implements the algorithm.

```
LZW-Decoding (fin, fout)
count ← −1
for each character a ∈ Σ
    do count ← count + 1
        Hash-Insert (D, (−1, a, count))
count ← count + 1
Hash-Insert (D, (−1, END, count))
c ← first code on 1 + log(count) bits from fin
write string(c) in fout
a ← first(string(c))
repeat
    d ← next code on 1 + log(count) from fin
    if d > count
        then count ← count + 1
             parent(count) ← c
             label(count) ← a
             write string(c)a in fout
             c ← d
        else a ← first(string(d))
             if a ≠ END
                then count ← count + 1
                     parent(count) ← c
                     label(count) ← a
                     write string(d) in fout
                     c ← d
                else exit
forever
```

Figure 10.25: LZW decoding algorithm.

Source texts	French	C sources	Alphabet	Random
Sizes in bytes	62816	684497	530000	70000
Huffman	53.27%	62.10%	72.65%	**55.58**%
Ziv-Lempel	**41.46**%	34.16%	2.13%	63.60%
Factor	47.43%	**31.86**%	**0.09**%	73.74%

Figure 10.26: Sizes of texts compressed with three algorithms.

The dynamic version has been discovered independently by Faller (1973) and Gallager (1978). Practical versions are given by Cormack and Horspool (1984) and Knuth (1985). A precise analysis leading to an improvement is presented in (Vitter, 1987). The command `compact` of UNIX implements the dynamic Huffman coding.

It is unclear to whom precisely should be attributed the idea of data compression using arithmetic coding. It is sometimes refer to Elias (1963), and has become popular after the publication of the article of Witten, Neal, and Cleary. An efficient data structure for the tables of frequencies is due to Fenwick (1994). The main interest in arithmetic coding for text compression is that the two different processes "modeling" the statistics of texts and "coding" can be made independent modules.

Several variants of the Ziv-Lempel algorithm exist. The reader can refer to the books of Bell, Cleary, and Witten (1990) or Storer (1988) for a discussion on them.

The books of Held (1991) and Nelson (1992) present practical implementations of various compression algorithms, while the book of Crochemore and Rytter (1994) overflows the strict topic of text compression and described more algorithms and data structures related to texts manipulations.

10.8 Defining Terms

Codeword: sequence of bits of a code corresponding to a symbol.
Prefix: a word $u \in \Sigma^*$ is a prefix of a word $w \in \Sigma^*$ if $w = uz$ for some $z \in \Sigma^*$.
Prefix code: set of words such that no word of the set is a prefix of another word contained in the set. A prefix code is represented by a coding tree.
Segment: a word $u \in \Sigma^*$ is a segment of a word $w \in \Sigma^*$ if u occurs in w, i.e. $w = vuz$ for two words $v, z \in \Sigma^*$. (u is also referred to as a factor or a subword of w)
Trie: tree in which edges are labeled by letters or words.

10.9 References

Cormack, G.V., Horspool, R.N.S. 1984. Algorithms for adaptive Huffman Codes. *Inf. Process. Lett.* 18:159–165.

Faller, N. 1973. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*. 593-597.

Fenwick, P.M. 1994. A new data structure for cumulative probability tables. *Software—Practice and Experience*. 24:327–336.

Gallager, R.G. 1978. Variations on a theme by Huffman. *IEEE Trans. Inform. Theory.* 24(6):668–674.

Knuth, D.E. 1985. Dynamic Huffman coding. *J. Algorithms*. 6:163–180.

Vitter, J.S. 1987. Design and analysis of dynamic Huffman codes. *J. ACM*. 34(4):825–845.

Welch, T.A. 1984. A technique for high-performance data compression. *IEEE Computer*. 17(6):8-19.

Witten, I.H., Neal, R., Cleary, J.G. 1987. Arithmetic coding for data compression. *Comm. ACM*. 30(6):520–540.

Ziv, J., Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory.* 23(3):337–343.

10.10 Further Information

The set of algorithms presented in the chapter provides the basic methods for data compression. In commercial software they are often combined with other more elementary techniques that are described in textbooks. A wider panorama of data compression algorithms on texts may be found in several books such as:

- Bell, T.C., Cleary, J.G., and Witten, I.H. 1990. *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey.
- Crochemore, M., and Rytter, W. 1994. *Text Algorithms*, Oxford University Press.
- Held, G. 1991. *Data Compression.* John Wiley & Sons.
- Nelson, M. 1992. *The Data Compression Book*, M&T Books.
- Storer, J.A. 1988. *Data Compression: Methods and Theory*, Computer Science Press.

Research papers in text data compression are disseminated in a few journals, among which are: *Communications of the ACM*, *Journal of the ACM*, *Theoretical Computer Science*, *Algorithmica*, *Journal of Algorithms*, *SIAM Journal on Computing*, *IEEE Trans. Information Theory.*

An annual conference presents the latest advances of this field of research:

- *Data Compression Conference*, which is regularly held at Snowbird (Utah) in spring.

Two other conferences on pattern matching also present research issues in the domain:

- *Combinatorial Pattern Matching*, which started in 1990 and was held in Paris (France), London (England), Tucson (Arizona), Padova (Italy), Asilomar (California), Helsinki (Finland), Laguna Beach (California).
- *Workshop on String Processing*, which started in 1993 and was held in Belo Horizonte (Brasil), Valparaiso (Chile), and Recife (Brasil).

But general conferences in computer science often have sessions devoted to data compression algorithms.