Chapter 2

Combining Huffman and Run-Length Coding

Gal, amant de la reine, alla, tour magnanime,
gallament de l'arène à la Tour Magne, à Nîmes.
— Robert DESNOS

2.1 Introduction

We now turn to Jakobsson's [34] method for the compression of sparse bit-strings and show how it can be improved. It uses Huffman coding, and since this algorithm will also reappear as one of the main subjects in all the subsequent chapters, a brief description will now follow. The term 'code' is used throughout as abbreviation for 'set of codewords'.

We are given a set of n non-negative weights $\{w_1, \ldots, w_n\}$, which are the frequencies (or probabilities) of occurrence of the letters of some alphabet. The problem is to generate a binary variable-length code, consisting of codewords with lengths l_i bits, $1 \leq i \leq n$, with optimal compression capabilities, i.e., such that $\sum_{i=1}^{n} w_i l_i$ is minimized. Moreover, to allow unique decipherability, the code should have the prefix property, that is, no codeword is the prefix of any other. In 1952, Huffman [32] proposed the following algorithm which solves the problem.

The Huffman encoding algorithm

1. **If** $n = 1$ **then** the codeword corresponding to the only weight is the nullstring; **return**.
2. Let w_1 and w_2, without loss of generality, be the two smallest weights.
3. Solve the problem recursively for the $n - 1$ weights $w_1 + w_2, w_3, \ldots, w_n$; let α be the codeword assigned to the weight $w_1 + w_2$.

4. The code for the n weights is obtained from the code for $n-1$ weights generated in point 3 by replacing α by the two codewords $\alpha 0$ and $\alpha 1$; **return**.

In the straightforward implementation, the weights are first sorted and then every weight obtained by combining the two which are currently the smallest, is inserted in its proper place in the sequence so as to maintain order. This yields an $O(n^2)$ time complexity. Using two queues, Van Leeuwen [59] has shown how to reduce it to $O(n \log n)$. In fact, the dominating part of the time is spent on sorting the weights w_i, requiring $\Omega(n \log n)$. If the weights are already given in order, the algorithm can be implemented in time $O(n)$.

For decompression, one uses either a *translation table* giving for each codeword the corresponding encoded letter of the alphabet, or one uses the *Huffman tree* in the leaves of which these letters are stored. Efficient techniques for the decoding of Huffman encoded messages are presented in Chapter 3.

When a natural language text is to be encoded, the Huffman algorithm is usually applied on the frequencies of the different letters; however, one can also encode the different letter pairs or triplets, etc. Compression will be improved, but at the cost of much larger translation tables. The basic idea of the new methods for bit-vector compression which are presented in this chapter, is to use Huffman codes to represent items of completely different nature, provided there is an unambiguous way to decompose a given file into these items. In the present application, we apply Huffman coding to *bit patterns* of constant size on one hand and to *lengths* of 0-bit-runs on the other. These lengths are chosen in a systematic way, using new exotic numeration systems. Some of the methods consistently yield compression factors superior to the previously known ones, and even improve on method PRUNE of Chapter 1.

2.2 Incorporating 0-Run-Length Coding

In what follows, each method is identified by a label which appears in typewriter font and usually between parenthesis following the explanations, together with a number referring to the corresponding line in Table 2.4 (page 27), which displays the experimental results. Methods `TREE` and `PRUNE` of Chapter 1 were added for comparison to the table (lines 1 and 2), as well as Jakobsson's Huffman coding method, which is referred to as method `NORUN` (line 3). We use the same notations as in Chapter 1, i.e., the vector to be compressed is v_0 and has length l_0 bits.

As we are interested in sparse bit-strings, we can assume that the probability p of a block of k consecutive bits being zero is high. If $p \geq 0.5$, method `NORUN` assigns to this 0-block a codeword of length one bit, so we can never expect a better compression factor than k. On the other hand, k cannot be too large since we must generate codewords for 2^k different blocks.

In order to get a better compression, we extend the idea of method `NORUN` in

the following way: there will be codewords for the $2^k - 1$ non-zero blocks of length k, plus some additional codewords representing runs of zero-blocks of different lengths. In the sequel, we use the term 'run' to designate a run of zero-blocks of k bits each.

The length (number of k-bit blocks) of a run can take any value up to l_0/k, so it is impractical to generate a codeword for each: as was just pointed out, k cannot be very large, but l_0 is large for applications of practical importance. On the other hand, using a fixed-length code for the run length would be wasteful since this code must suffice for the maximal length, while most of the runs are short. The following methods attempt to overcome these difficulties.

2.2.1 Definition of classes of run-lengths

Starting with a fixed-length code for the run-lengths, we like to get rid of the leading zeros in the binary representation $B(\ell)$ of run-length ℓ, but we clearly cannot simply omit them, since this would lead to ambiguities. We *can* omit the leading zeros if we have additional information such as the position of the leftmost 1 in $B(\ell)$. Hence, partition the possible lengths into classes C_i, containing run-lengths ℓ which satisfy $2^{i-1} \leq \ell < 2^i$, $i = 1, \ldots, \lfloor \log_2(l_0/k) \rfloor$. The $2^k - 1$ non-zero block-patterns and the classes C_i are assigned Huffman codewords corresponding to the frequency of their occurrence in the file; a run of length ℓ belonging to class C_i is encoded by the codeword for C_i, followed by $i - 1$ bits representing the number $\ell - 2^{i-1}$. For example, a run of 77 0-blocks is assigned the codeword for C_7 followed by the 6 bits 001101. Note that a run consisting of a single 0-block is encoded by the codeword for C_1, without being followed by any supplementary bits.

The Huffman decoding procedure has to be modified in the following way: The table contains for every codeword the corresponding class C_i as well as $i - 1$. Then, when the codeword which corresponds to class C_i is identified, the next $i - 1$ bits are considered as the binary representation of an integer m. The codeword for C_i followed by those $i-1$ bits represent together a run of length $m + 2^{i-1}$; the decoding according to Huffman's procedure resumes at the i-th bit following the codeword for C_i. Summarizing, we in fact encode the length of the binary representation of the length of a run (method `LLRUN`, line 4).

2.2.2 Representing the run-length in some numeration system

Method `LLRUN` seems to be efficient since the number of bits in the binary representation of integers is reduced to a minimum and the lengths of the codewords are optimized by Huffman's algorithm. But encoding and decoding are admittedly complicated and thus time consuming. We therefore propose other methods for which the encoded file will consist only of codewords, each representing a certain string of bits. Even if their compression factor is lower than `LLRUN`'s, these methods are justified by their simpler processing.

To the $2^k - 1$ codewords for non-zero blocks, a set S of t codewords is adjoined representing $h_0, h_1, \ldots, h_{t-1}$ consecutive 0-blocks. Any run of zero-blocks will now be encoded by a suitable linear combination of some of these codes. The number t depends on the numeration system according to which we choose the h_i's and on the maximal run-length M, but should be low compared to 2^k. Thus in comparison with method `NORUN`, the table used for compressing and decoding should only slightly increase in size, but long runs are handled more efficiently. The encoding algorithm now becomes:

Step 1: Collect statistics on the distribution of run-lengths and on the set NZ of the 2^k-1 possible non-zero blocks. The total number of occurrences of these blocks is denoted by N_0 and is fixed for a given set of bit-maps.

Step 2: Decompose the integers representing the run-lengths in the numeration system with set S of "basis" elements; denote by $\mathrm{TNO}(S)$ the total number of occurrences of the elements of S.

Step 3: Evaluate the relative frequency of appearance of the 2^k-1+t elements of $\mathrm{NZ} \cup S$ and assign a Huffman code accordingly.

For any $x \in (\mathrm{NZ} \cup S)$, let $p(x)$ be the probability of the occurrence of x and $\ell(x)$ the length (in bits) of the codeword assigned to x by the Huffman algorithm. The weighted average length of a codeword is then given by $\mathrm{AL}(S) = \sum_{x \in (\mathrm{NZ} \cup S)} p(x)\ell(x)$ and the size of the compressed file is

$$\mathrm{AL}(S) \times (N_0 + \mathrm{TNO}(S)).$$

After fixing k so as to allow easy processing of k-bit blocks, the only parameter in the algorithm is the set S. In what follows, we propose several possible choices for the set $S = \{1 = h_0 < h_1 < \ldots < h_{t-1}\}$. To overcome coding problems, the h_i and the bounds on the associated digits a_i should be so that there is a unique representation of the form $L = \sum_i a_i h_i$ for every natural number L.

Given such a set S, the representation of an integer L is obtained by the following simple procedure:

for $i \leftarrow t-1$ **to** 0 **by** -1

$a_i \leftarrow \lfloor L/h_i \rfloor$
$L \leftarrow L - a_i \times h_i$
end

The digit a_i is the number of times the codeword for h_i is repeated. This algorithm produces a representation $L = \sum_{i=0}^{t-1} a_i h_i$ which satisfies

$$\sum_{i=0}^{j} a_i h_i < h_{j+1} \qquad \text{for} \quad j = 0, \ldots, t-1. \tag{2.1}$$

Condition (2.1) guarantees uniqueness of representation (see [16]).

2.2.3 Selection of the set of basis elements

A natural choice for S is the standard binary system, $h_i = 2^i$, $i \geq 0$, or higher base numeration systems such as $h_i = m^i$, $i \geq 0$ for some $m > 2$. If the run-length is L it will be expressed as $L = \sum_i a_i m^i$, with $0 \leq a_i < m$ and if $a_i > 0$, the codeword for m^i will be repeated a_i times. Higher base systems can be motivated by the following reason.

If p is the probability that a k-bit block consists only of zeros, then the probability of a run of r blocks is roughly $p^r(1-p)$, i.e., the run-lengths have approximately geometric distribution. The distribution is not exactly geometric since the involved events (some adjacent blocks contain only zeros, i.e., a certain word does not appear in some consecutive documents) are not independent. Nevertheless the experiments showed that the number of runs of a given length is an exponentially decreasing function of run-length (see Figure 2.1 below). Hence with increasing base of the numeration systems, the relative weight of the h_i for small i will rise, which yields a less uniform distribution for the elements of $\mathrm{NZ} \cup S$ calculated in Step 3. This has a tendency to improve the compression obtained by the Huffman codes. Therefore passing to higher order numeration systems will reduce the value of $\mathrm{AL}(S)$.

On the other hand, when numeration systems to base m are used, $\mathrm{TNO}(S)$ is an increasing function of m. Define r by $m^r \leq M < m^{r+1}$ so that at most r m-ary digits are required to express a run-length. If the lengths are uniformly distributed, the average number of basis elements needed (counting multiplicities) is proportional to $(m-1)r = (m-1)\log_m M$, which is increasing for $m > 1$. For our nearly geometric distribution this is also the case as can be seen from the experiments. Thus from this point of view, lower base numeration systems are preferable. Numeration systems to base m were checked for $m = 2, \ldots, 10$ (`POW2`, ..., `POW10`; lines 5–13). For $m = 3$, we experimented with another variant: instead of using certain codewords twice where a digit 2 is needed in the ternary representation, we added a special codeword indicating that the following codeword has to be doubled (`POW3M`; 14).

2.2.4 Trying to reduce the total number of codewords

As an attempt to reduce $\mathrm{TNO}(S)$, we pass to numeration systems with special properties, such as systems based on Fibonacci numbers

$$F_0 = 0, \quad F_1 = 1, \qquad F_i = F_{i-1} + F_{i-2} \quad \text{for} \quad i \geq 2.$$

(a) The binary Fibonacci numeration system: $h_i = F_{i+2}$ (`FIBBIN`; 15). Any integer L can be expressed as $L = \sum_{i \geq 0} b_i F_{i+2}$ with $b_i = 0$ or 1, such that this binary representation of L consisting of the string of b_i's contains no adjacent 1's (see Knuth [41, Exercise 1.2.8–34], Fraenkel [16]). This fact for a binary Fibonacci system is equivalent to condition (2.1), and reduces the number of codewords we need to represent a specific run-length, even though the number of added codewords is larger than for `POW2` (instead of $t(\texttt{POW2}) = \lfloor \log_2 M \rfloor$ we have $t(\texttt{FIBBIN}) = \lfloor \log_\phi(\sqrt{5}M) \rfloor - 1$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio). For example, when all the run-lengths are equally probable, the average number of codewords per run is assymptotically (as $k \to \infty$) $\frac{1}{2}(1 - 1/\sqrt{5})t(\texttt{FIBBIN})$ instead of $\frac{1}{2}t(\texttt{POW2})$.

(b) A ternary Fibonacci numeration system: $h_i = F_{2(i+1)}$, i.e., we use only Fibonacci numbers with even indices. This system has the property that there is at least one 0 between any two 2's ([16]). This fact for a ternary Fibonacci system is again equivalent to (2.1) (`FIBTER`; 16).

(c) Like **(b)**, but with a special codeword for the digit 2 like in `POW3M` (`FIBTERM`; 17).

2.2.5 Trying to reduce the average codeword length

New systems with similar properties are obtained from generalizations of the Fibonacci systems to higher order. The idea is to lower $\mathrm{AL}(S)$ at the cost of $\mathrm{TNO}(S)$, while $\mathrm{TNO}(S)$ is kept smaller than for the `POW`m systems.

(a) Methods which are generalizations of method `FIBBIN`, based on the following sequence of integers which is defined recursively:

$$\begin{aligned} u_{-1}^{(m)} &= 1, \qquad u_0^{(m)} = 1 \\ u_i^{(m)} &= m u_{i-1}^{(m)} + u_{i-2}^{(m)}, \quad i \geq 1, \end{aligned} \tag{2.2}$$

(This is the recursion satisfied by the convergents of the continued fraction $[1,\dot{m}]$, where $\dot{m}$ represents the infinite concatenation of m with itself.) For $m = 1$ we get `FIBBIN`. The system based on the sequence $u_0^{(m)}, u_1^{(m)}, \ldots$ is an $(m+1)$-ary numeration system with the following property: there exists a unique representation of any integer L as $L = \sum_i a_i u_i^{(m)}$, such that $0 \leq a_i \leq m$, and such that if a_{i+1} reaches its maximal value m, then a_i is zero. These methods were checked for $m = 2, \ldots, 10$ (`REC2`, ..., `REC10`; 18–26).

(b) Methods which are generalizations of method `FIBTER`, based on the sequence:

$$\begin{aligned} u_{-1}^{(ab)} &= 1-\epsilon a, \qquad u_0^{(ab)} = 1 \\ u_i^{(ab)} &= (ab+2)\,u_{i-1}^{(ab)} - u_{i-2}^{(ab)}, \quad i \geq 1, \end{aligned} \tag{2.3}$$

where a,b and ϵ are parameters. This is the recursion satisfied by the convergents p_{2i}/q_{2i} of the continued fraction $[1, b, a, b, a, \dots]$. Specifically, (2.3) is the recursion for the q_{2i} when $\epsilon = 0$ and for the p_{2i} when $\epsilon = 1$. The numeration system based on the q_{2i} is denoted by `AaBbQ` and that based on the p_{2i} by `AaBbP`. Any nonnegative integer L can be represented in these numeration systems in the form $L = \sum_{i=0}^{n} c_i u_i^{(ab)}$, where

$$0 \leq c_0 \leq a(b+\epsilon), \qquad 0 \leq c_i \leq ab+1 \quad (i>0),$$

and the following condition holds: if for some $0 \leq i < r \leq m$, c_i and c_r assume their maximal values, then there exists an index j satisfying $i < j < r$, for which $c_j < ab$. In particular for `A1B1P`, which is our ternary system `FIBTER`, we have $c_j = 0$.

From (2.3) follows that the numeration system `AaBbQ` is equivalent to the system `AcBdQ` if $ab = cd$. In particular, `AaBbQ` is equivalent to `AbBaQ`. We have checked the following methods, which are listed in lexicographically increasing order of their basis elements: `A1B2Q` (27), which is ternary in c_0, but 4-ary in c_j for every $j > 0$; `A1B2P` (28), a 4-ary numeration system; `A1B3Q` (29), `A2B1P` (30), `A1B3P` (31), `A2B2Q` (32), `A1B4P` (33), `A3B1P` (34), `A2B2P` (35) and `A4B1P` (36).

Table 2.1: *Sequences defining the numeration systems*

Method	Arity	Sequence of basis elements									
FIBBIN	2	1	2	3	5	8	13	21	34	55	89
				144	233	377	610	987	1597	2584	4181
FIBTER	3	1	3	8	21	55	144	377	987	2584	
REC2	3	1	3	7	17	41	99	239	577	1393	3363
REC3	4	1	4	13	43	142	469	1549			
REC4	5	1	5	21	89	377	1597				
REC5	6	1	6	31	161	836	4341				
REC6	7	1	7	43	265	1633					
REC7	8	1	8	57	407	2906					
REC8	9	1	9	73	593	4817					
REC9	10	1	10	91	829						
REC10	11	1	11	111	1121						
A1B2Q	4,3	1	3	11	41	153	571	2131			
A1B2P	4	1	4	15	56	209	780	2911			
A1B3Q	5,4	1	4	19	91	436	2089				
A2B1P	4,5	1	5	19	71	265	989	3691			
A1B3P	5	1	5	24	115	551	2640				
A2B2Q	6,5	1	5	29	169	985					
A1B4P	6	1	6	35	204	1189					
A3B1P	5,7	1	7	34	163	781	3742				
A2B2P	6,7	1	7	41	239	1393					
A4B1P	6,9	1	9	53	309	1801					

Table 2.1 lists the sequences of the first few basis elements of the various numeration systems. A system that is ℓ-ary in c_0 and m-ary in all other digits contains m, ℓ in the column labeled "Arity".

2.2.6 Error detection

As was briefly mentioned at the end of Chapter 1, one of the weak points of Huffman codes is their sensitivity to errors: a single wrong bit may render the code useless. In order to locally restrict the damage caused by errors, one could add some redundant bits so as to ensure that for some constant block-size r, the bits indexed $ir+1$, $i \geq 0$, start a new codeword. A single error can then affect at most r bits of the coded file. This issue will be addressed later in Section 5.2.3. Since for small r the loss of compression may be significant, we compare the error-detecting capabilities of our methods supposing that no "synchronizing" bits are added and restrict ourselves to the case when a single error occurred.

For `NORUN`, suppose the error occurred in x, which was the i-th codeword of the compressed file. Now the i-th codeword is interpreted as some codeword y. If x and y are of the same length, the decoding from the $(i+1)$-st codeword on is correct and the error will not be detected. However, only a single k-bit block of the decompressed file is garbled. If x and y are not of the same length, chances are good to reveal the existence of an error at the end of the string either because the last bits do not form a codeword or because the size of the decompressed file is not as expected.

For `LLRUN`, error detection can be enhanced by checking that codewords for classes C_i, representing runs of zero-blocks, do not appear consecutively. This may happen after a wrong bit which transforms the following into a sequence of independent codewords.

For all the methods using codewords for the basis elements $\{h_0, h_1, \ldots, h_{t-1}\}$ of a numeration system, these codewords should appear for every given run-length in monotone order, e.g., by decreasing basis elements. An error will quickly be discovered by checking that the decoded run-lenghts indeed appear in decreasing order. A wrong bit would tend to mix up the codewords and there are even better chances that an error will break this rule than for `LLRUN`.

2.3 Experimental Results

All the above mentioned methods were checked on the 56588 bitmaps of RRP used also in Chapter 1. For all the methods using Huffman coding, the block-size was chosen as $k = 8$, i.e., one byte. Thus codewords were generated for 255 non-zero bytes and for 3 to 18 run-lengths, depending on the numeration system chosen. For `LLRUN`, we had classes C_1 to C_{13}. By examining the statistics of the distribution of non-zero k-bit blocks, we found that although blocks with a fixed number s of 1-bits were nearly equiprobable, the frequency decayed rapidly as s increased. One exception should be noted: as s passed from $k-1$ to k, the frequency rose. This fact can be explained by the "clustering effect": adjacent bits represent usually documents written by the same author and there is a positive correlation for a word

to appear in consecutive documents because of the specific style of the author or simply because such documents often treat the same subject.

The Responsa maps were divided into three files of different size and density. The first file corresponds to words of length 1 to 4 characters, which are the words with highest frequency of occurrence; the second file contains words of length 5–8 and the third of length 8–13 (lowest frequency); for technical reasons, the 8-letter words were split between the two last files. Finally, the three files were unified.

In order to evaluate the influence of the clustering effect on the compression, a test was designed on the randomly generated bitmaps of Section 1.3.2.

A digitized picture can also be considered as a long bit-string, and for pictures of technical drawings for example, these maps are usually sparse. The problem of image compression is not quite the same as the one treated in this paper, since often several bits can be changed without losing the general impression of the picture. Thus methods can be applied which use noise removal, edge detection, void filling, etc. in both dimensions (see Ramachandran [51]), and are thus not generalizable to information retrieval bit-maps, where every single bit must be kept. Nevertheless, we wanted to see how our methods perform on digitized pictures and to compare them with other methods used for image compression.

The picture we chose was used by several authors to test their compression methods (e.g., Hunter and Robinson [33]). After digitizing and after thresholding, a bit-map of $512 \times 512 = 262144$ bits was obtained, which is reproduced on page 4. Other authors used a much higher resolution (1728×2376). Moreover even when different scanners operate at the same resolution and scan the same document, they can give significantly different statistics and compression factors (see [33]). Since our equipment did not allow a higher resolution, we decided to compare our methods with the "Modified Huffman Codes" (MHC) proposed in [33], applying all the methods to our low-resolution picture.

The latter method consists of coding runs of 0-bits. There are 64 so-called "Terminal codes" TC(i) representing a run of i zeros followed by a 1, $i = 0, \ldots, 63$, and some "Make-up" codewords MU(j) standing for a run of $64j$ 0-bits, $j \geq 1$. A run of length ℓ is encoded by TC(ℓ) if $\ell < 64$, otherwise by MU(j) followed by TC(i) such that $64j + i = \ell$. The codewords are generated using the Huffman algorithm. One could also add codewords for runs of 1-bits, but for the sparse bit-strings we are interested in for our information retrieval applications, their influence will be negligible.

Table 2.2 is a summary of statistical information on the three files of Responsa-maps, on the unified file, on the file of random maps and the picture.

Table 2.2: *Statistics of the bit-maps*

Contents of the file: words of length	Number of maps	Number of non-zero bytes (N_0)	Nbr of runs	Density: avg nbr of 1-bits per map	Percentage of 1-bits per map	Average length of run (in byte)	Average number of runs per map
1–4	15378	6,091,250	3,214,770	720.15	1.70%	23.38	209.05
5–8	36004	5,951,808	4,336,727	218.09	0.52%	42.50	120.45
8–13	5206	518,919	427,947	117.69	0.28%	63.08	82.20
unified (1–13)	56588	12,561,977	7,979,444	345.29	0.817%	35.90	141.01
random	5664	1,468,517	1,053,433	342.38	0.810%	27.02	185.99
picture	1	3135	1470	11398	4.35%	20.16	1470

Figure 2.1 displays the relative frequency (in %) of runs of a certain length as a function of run-length (number of k-bit blocks), for both the unified and the random files.

Figure 2.1: *Run-length distribution*

As we started to get results, we saw that the idea of adding a special codeword for the digit 2 in ternary systems (methods `POW3M` and `FIBTERM`) was not a good one, therefore we did not expand it to numeration systems with larger base.

As expected, `LLRUN` was the best method, but `REC2` which was the second best on the Responsa and random files used only about 5–8% additional space. One of the striking results was that for several methods there are on the average less than

2 codewords per run; if the run-lengths were uniformly distributed, the expected number for `POW2` for example was 6. Table 2.3 is a sample for some of the good methods.

Table 2.3: *Average number of codewords needed per run*

File	POW2	FIBBIN	FIBTER	REC2	A1B2Q
Unified	2.17	1.84	2.46	2.39	2.79
Random	2.17	1.85	2.47	2.40	2.79
Picture	2.22	1.70	2.29	2.20	2.38

Table 2.4 gives the results of the experiments on the unified, the random and the picture files. The methods are listed by order of their appearance in the explanations. The values for AL are given in bytes (8-bit blocks). For `LLRUN`, the average length is calculated as follows: let $p(\mathrm{NZ}_i)$ denote the probability of occurrence of the i-th non-zero block and $\ell(\mathrm{NZ}_i)$ the length (in bits) of the corresponding Huffman code; define $p(C_i)$ and $\ell(C_i)$ similarly for the classes of zero-block runs; then

$$\mathrm{AL} \quad = \quad \frac{1}{8}\left(\sum_i p(\mathrm{NZ}_i)\,\ell(\mathrm{NZ}_i) \;+\; \sum_i p(C_i)\,(\ell(C_i)+i-1)\right).$$

As can be seen, $\mathrm{TNO}(S)$ is indeed increasing and $\mathrm{AL}(S)$ decreasing when passing to higher order numeration systems, with few exceptions on the picture file, which is a very small sample (the size of the picture file is only 0.1% of the size of the random file). The average length of a single codeword was low, about half a byte. This was due to the Huffman procedure, which takes advantage of the great differences in the frequencies of the non-zero blocks.

When the methods are sorted according to the compression they yield, one obtains a similar order on all the files. The compression factors for methods 5–36 vary only slightly for a given file; on the three files of Responsa maps, the CF varies from 10.03 to 10.61 for the words of length 1–4, from 20.68 to 22.28 for the words of length 5–8 and from 30.16 to 33.13 for the words of length 8–13, thus the CF is a decreasing function of the density of the file (see Table 2.2).

One can see that the results of the experiment on random maps are similar to the results obtained on the Responsa maps. Generally, the compression is lower in the random case so we conclude that the clustering effect in the Responsa maps improves the performance of our compression methods.

Though the digitized picture is less sparse than the other maps — the probability for a 1-bit is 4.35% , see Table 2.2 — the strong correlation between the bits yields a high compression factor of up to 9.2 approximately with the best method, which is again `LLRUN`. The MHC method of [33] gave a compression factor of only 8.29 (which is less than for some of our methods).

Table 2.4: *Results of the experiments on three files*

Method	**Unified** File			**Random** File			**Picture** File		
	TNO	AL	CF	TNO	AL	CF	TNO	AL	CF
1 TREE	–	–	10.533	–	–	8.451	–	–	5.560
2 PRUNE	–	–	17.451	–	–	15.906	–	–	5.809
3 NORUN	286449015	0.151	6.568	28460059	0.152	6.585	29633	0.189	5.284
4 LLRUN	7979444	0.776	18.768	1053433	0.730	16.251	1470	0.770	9.241
5 POW2	17314002	0.587	17.039	2289097	0.543	14.664	3259	0.615	8.332
6 POW3	21355385	0.517	17.047	2826247	0.471	14.798	3683	0.562	8.553
7 POW4	24769215	0.473	16.924	3277474	0.430	14.675	4865	0.501	8.182
8 POW5	27794543	0.445	16.642	3688763	0.403	14.414	5225	0.476	8.227
9 POW6	30650875	0.418	16.545	4058894	0.375	14.458	5078	0.473	8.428
10 POW7	33231411	0.396	16.507	4401817	0.353	14.456	5441	0.451	8.465
11 POW8	35581405	0.377	16.489	4717956	0.335	14.447	6764	0.407	8.139
12 POW9	37871111	0.361	16.443	5030555	0.320	14.382	6377	0.415	8.300
13 POW10	39961965	0.348	16.380	5310817	0.308	14.328	6773	0.398	8.317
14 POW3M	21355385	0.541	16.293	2826247	0.499	13.964	3683	0.586	8.202
15 FIBBIN	14716716	0.631	17.360	1944357	0.586	14.966	2504	0.663	8.766
16 FIBTER	19648611	0.533	17.410	2598588	0.489	15.040	3369	0.581	8.666
17 FIBTERM	19648611	0.558	16.637	2598588	0.518	14.208	3369	0.601	8.377
18 REC2	19089063	0.542	17.442	2523209	0.497	15.080	3231	0.588	8.750
19 REC3	23063838	0.494	17.001	3055201	0.446	14.825	3989	0.534	8.612
20 REC4	26531683	0.455	16.800	3515323	0.410	14.663	4921	0.491	8.292
21 REC5	29577710	0.423	16.766	3918559	0.380	14.617	4988	0.477	8.463
22 REC6	32385333	0.399	16.668	4294663	0.358	14.517	5411	0.451	8.500
23 REC7	34921144	0.380	16.559	4630351	0.338	14.500	6456	0.413	8.277
24 REC8	37284439	0.364	16.491	4957755	0.323	14.432	6353	0.415	8.329
25 REC9	39481131	0.350	16.427	5249356	0.310	14.365	6782	0.397	8.329
26 REC10	41572115	0.338	16.334	5528749	0.299	14.284	7273	0.380	8.287
27 A1B2Q	22284147	0.503	17.073	2943319	0.455	14.898	3497	0.565	8.740
28 A1B2P	23927839	0.481	17.037	3167224	0.436	14.794	4522	0.513	8.346
29 A1B3Q	25647807	0.463	16.916	3395956	0.420	14.643	4649	0.503	8.370
30 A2B1P	25811643	0.450	16.949	3423471	0.414	14.786	4883	0.492	8.314
31 A1B3P	27306536	0.449	16.719	3621359	0.405	14.502	5275	0.476	8.180
32 A2B2Q	28601027	0.438	16.588	3785631	0.397	14.346	5345	0.473	8.174
33 A1B4P	30315745	0.420	16.602	4016360	0.377	14.486	5132	0.471	8.411
34 A3B1P	31045059	0.404	16.962	4127455	0.361	14.807	5147	0.462	8.556
35 A2B2P	32098563	0.400	16.736	4261207	0.358	14.577	5033	0.464	8.644
36 A4B1P	35720479	0.368	16.817	4761363	0.328	14.641	5869	0.425	8.558

It has been proposed to drop the codewords for the 255 non-zero blocks and to apply our methods to 0-*bit* runs as in the MHC-method, instead of 0-*byte* runs. A special codeword for a run of length 0 must be added. The advantage of this variant is that it considerably reduces the size of the Huffman translation tables. On the other hand, execution time for encoding and decoding will increase due to the bit-manipulations. On the random file, this method gave slightly inferior compression factors than the byte-oriented variants, for all the methods. On the Responsa maps, the difference was more evident, as well as for the picture, which was not surprising, since there we have relatively long runs of consecutive 1's which are associated with much waste in the bit-oriented method.

2.4 Concluding Speculations

Some of the new techniques presented in this chapter gave higher compression factors on sparse bit-strings than other known methods, while being easy to implement and allowing for fast decoding. Nevertheless, the experiments showed that the "optimal" numeration system depends on the statistics of the bit-map file. Therefore, given a specific file, one should evaluate the CF for several systems and finally compress with the system giving the best results. It is even not necessary to stick to one of the above numeration systems, nor to any well-known system at all. As a matter of fact, *every* increasing sequence of integers $h_0 = 1, h_1, h_2, \ldots$ can be considered as the set of basis elements of some numeration system, and the representation of any integer in it will be unique if we use the algorithm of Section 2.2.2. If we make only small perturbations in the sequences of Table 2.1, changing one of the elements slightly and continuing with the same recurrence formula to obtain the following, the desirable properties of the digits of the representation will be preserved at all places where the recurrence formula holds.

We thus suggest the following heuristic to improve the CF:

Step 1: Start the search with the numeration system giving the best CF among the methods 5–36; $i \leftarrow 1$.

Step 2: Create new sequences of basis elements by increasing or decreasing h_i; for $j > i$, h_j are obtained by the recurrence formula of the system found in Step 1; for each of the sequences evaluate its CF; continue until a first value of h_i is found which gives a local optimum.

Step 3: Repeat Step 2 for $i \leftarrow i+1$ until the obtained improvement is smaller than a certain predetermined amount.

There is of course no certainty that the optimal sequence will be found or even approached. Perhaps the local optima in Step 2 are not global; perhaps starting the Step 3 iteration with i greater than 1 can lead to better results, but since long runs are rare, perturbations in h_i for larger i will have less influence on the CF.

This heuristic was applied to the unified file, which led to the following sequences:

	Sequence of basis elements							CF
REC2	1	3	7	17	41	99	...	17.442
	1	3	7	18	43	104	...	17.463
	1	3	7	18	44	106	...	17.478
	1	3	7	18	44	107	...	17.480

Note that the last few systems give a slightly better compression than method PRUNE of Chapter 1. However, in addition to compression efficiency and robustness against errors, the simplicity of the decoding procedure should also be taken as criterion when deciding which compression method to use. The decoding of Huffman codes involves bit-manipulations, whereas the block sizes for methods TREE and PRUNE were primarily chosen so as to process only entire bytes (see Section 1.3.1). In the next chapter, we show how Huffman codes can also be decoded without bit-manipulations.