Lossless Compression for Text and Images

Alistair Moffat* Timothy C. Bell[†] Ian H. Witten[‡]
October 1995

Abstract Most data that is inherently discrete needs to be compressed in such a way that it can be recovered exactly, without any loss. Examples include text of all kinds, experimental results, and statistical databases. Other forms of data may need to be stored exactly, such as images—particularly bilevel ones, or ones arising in medical and remotesensing applications, or ones that may be required to be certified true for legal reasons. Moreover, during the process of lossy compression, many occasions for lossless compression of coefficients or other information arise.

This paper surveys techniques for lossless compression. The process of compression can be broken down into modeling and coding. We provide an extensive discussion of coding techniques, and then introduce methods of modeling that are appropriate for text and images. Standard methods used in popular utilities (in the case of text) and international standards (in the case of images) are described.

Keywords Text compression, image compression, lossless compression, coding.

CR Reviews Categories E.4 Coding and Information Theory.

^{*}Department of Computer Science, The University of Melbourne, Parkville, Australia 3052. Facsimile: +61 393481184. Internet: alistair@cs.mu.oz.au.

[†]Department of Computer Science, University of Canterbury, New Zealand. Facsimile: +6433642569. Internet: tim@cosc.canterbury.ac.nz.

[‡]Department of Computer Science, University of Waikato, New Zealand. Facsimile: +6478384155. Internet: ihw@waikato.ac.nz.

Contents

1	Intr	Introduction 3				
2	Mo	deling and Coding	3			
	2.1	Separation of Function	4			
	2.2	Overview of Coding Methods	5			
	2.3	Non-Entropy Coding	6			
	2.4	Entropy Coding	7			
	2.5	Minimum-Redundancy Coding	7			
	2.6	Arithmetic Coding	12			
	2.7	Relative performance	16			
	2.8	Binary Alphabet Coding	17			
3	Mo	dels for Text	18			
	3.1	Contextual Models	18			
	3.2	Dictionary Models	20			
	3.3	Model equivalence	22			
	3.4	The PPM algorithm	23			
	3.5	The ZIP algorithm	25			
	3.6	The Compress algorithm	27			
	3.7	Relative performance	27			
	3.8	The future of text compression	28			
4	Mo	dels for Images	28			
	4.1	The CCITT facsimile standards	30			
	4.2	Context models	31			
	4.3	Two-level models	32			
	4.4	JBIG: A standard for bilevel images	34			
	4.5	The GIF format for lossless image compression	37			
	4.6	The FELICS scheme for compression of grayscale images	38			
	4.7	The CALIC scheme for compression of continuous-tone images	40			
	4.8	Performance of lossless image compression methods	41			
5	Oth	er Forms of Data	43			
	5.1	Sound compression	43			
	5.2	Index compression	43			
	5.3	Textual images	44			
	5.4	Filesystem compression	45			
6	Sun	mmary 46				
\mathbf{R}	efere	nces	46			

1 Introduction

Much of the data stored on a computer system is the digital representation of a source that is, by nature, continuous. Video sequences, scanned images, and sound data all fall into this category. Because the form stored is already a quantized version of the original, it is appropriate for further approximation to be permitted, and *lossy* compression techniques can used to obtain extremely compact representations. The user is then free to choose a compression rate—a setting on the "knob" that controls the compressor—that provides the desired balance between the fidelity of the reconstructed data and the cost of storage. For this reason the focus of research into compression of video, image, and sound data is in lossy methods, the aim being to shift the trade-off curve.

Some data types, on the other hand, are intrinsically discrete, or digital. Stored text is the archetypical example of this, but there are many other situations where the original source must be capable of being reconstructed exactly. These applications include image data that must be certified for medical or legal reasons, and data such as remotely-sensed images for which the final use is yet unknown, and for which regenerating an approximation may be inadequate. In the case of satellite imagery there is also a matter of cost—it is far less expensive to store such images than to create them in the first place, and lossy compression might well be a false economy. Archival storage of images of historical documents may also require lossless compression—again, the needs of future scholars cannot be anticipated. For similar reasons the storage of statistical data—such as census tabulations—must be exact. And, as a final example, one can imagine the chaos that might reign were a compressed executable program to be decompressed into a form slightly different from the original.

In this chapter we examine methods currently used for the lossless compression of data. We focus on three main types of data, but the methods we describe are of general applicability. Indeed, one of the characteristics that marks lossless compression methods is that by and large any file can be compressed—and then reconstructed—using any compression method. The only variable is the extent of the compression; because the compression is lossless, faithful reproduction is always assured.

The remainder of this chapter is broken into four sections. In Section 2 we describe the modeling-coding paradigm for lossless compression, and describe a number of coding methods. Section 3 then shows the application of these ideas to text data, including a discussion of two state-of-the-art compression methods. In Section 4 we examine the lossless compression of images. A number of current methods—including the JBIG standard for the representation of bi-level (binary) images and the Group 4 facsimile standard—are described. Compression methods for a number of other types of data are discussed briefly in Section 5. A bibliography of relevant research literature appears at the end of the chapter.

2 Modeling and Coding

Central to an understanding of current lossless compression methods is the "modeling, coding" paradigm. In this section we describe this framework, and give details of coding methods. Sections 3, 4, and 5 describe models appropriate to different types of data, and suggest which of these coders they can be most usefully coupled with.

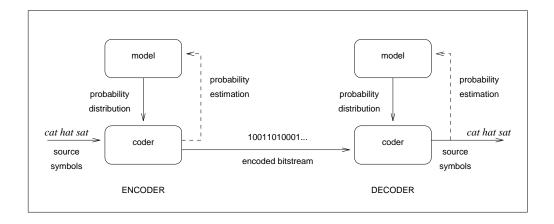


Figure 1: Structure of a lossless compressor

2.1 Separation of Function

Figure 1 shows the logical structure of an adaptive lossless compression system. The left-hand part of the diagram shows the action of the encoder; the central part represents a noiseless communications channel; and the right-hand side depicts the decoder.

At each coding step the *model* is required to estimate a probability for each possible symbol, without any knowledge of the symbol that is about to be represented. The *coder* module takes this probability distribution, together with knowledge of the next symbol in the stream, and transmits a sequence of bits (or symbols in some other channel alphabet) to the decoder. Only after the code for the symbol has been transmitted can the update stage of the model—the *statistics*, or *probability estimation* phase—decide what changes to make to the model. It is these changes that make adaptive coders so powerful. Changes can be made to the probabilities of symbols, in order to incorporate the updated statistics that result from having seen this one symbol; and they can be made to the structure of the model itself.

At the decoding end, the model must supply an identical probability distribution to the coding module, which then extracts and interprets an appropriate number of bits from the transmitted bitstream and determines the corresponding symbol from the source alphabet. This symbol is used in the probability estimation phase to mimic any changes effected in the encoder's model, thereby retaining model synchronization. Since the encoder and decoder now have identical models, the next symbol can be processed in the same manner. The function of the model, then, is to make accurate predictions of the probabilities of forthcoming symbols in the data stream, assuming knowledge of all symbols so far encoded, and perhaps some common a priori information shared by both encoder and decoder.

The function of the probability estimation phase of the model is to revise the probability estimates stored—either explicitly or implicitly—in the model. This revision can be as simple as incrementing a counter that records the frequency of the transmitted symbol; or it might involve considerable adjustment of the structure of the model and the creation of new conditioning classes upon which to base future probability estimates.

The function of the coding module is straightforward, and it is the most "mechanical" of these three processes. Given a particular symbol and a probability distribution on the full alphabet of symbols, the coder must determine and transmit the appropriate code. Given the same probability distribution and the stream of encoded bits, the decoder must determine which symbol was encoded.

This separation of function—model and coder, with the former including explicit probability estimation—was first espoused by Rissanen and Langdon (1981). For reasons of efficiency of implementation, in practical compression systems the distinction may not be as clear-cut as described here. However, for almost all known compression methods a symbol-wise equivalent method can be determined in which probabilities for each symbol are determined, and a coder used to represent that symbol. Langdon (1983) gives an example of this kind of equivalence; others may be found in Bell and Witten (1994).

2.2 Overview of Coding Methods

By Shannon's source coding theorem (Shannon, 1948), optimal coding is achieved when the length of the code assigned to the *i*th symbol of the alphabet is $-\log_2 p_i$ where p_i is the probability of that symbol. Such an assignment of codewords is the goal if compression is to be maximized.

Shannon's interest was primarily in information content, and using this content to define lower bounds. The *entropy* of a memoryless (or Bernoulli) source is a function of the probability distribution $P = [p_i | 1 \le i \le n]$ governing the source, where n is the size of the source alphabet. Shannon defined entropy to be

$$H(P) = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

Measured in bits per symbol, the entropy defines a limit on compressibility for this source that cannot be exceeded by any coder. Below we shall describe a coder that essentially meets this limit.

There are, however, other possible design objectives for a coder. One desirable feature is speed of operation—it should encode and/or decode quickly. Another is economy of memory usage, so that encoding and decoding can be done without the use of large tracts of memory. Because of the tension between these conflicting demands, there is no single "best" coding method. There are, however, a relatively small number of competing alternatives, and these are surveyed in the subsections below. One way of categorizing the alternative methods is as non-entropy coders and entropy coders; the former aiming for speed of operation or economy of memory usage, and the latter aiming for compactness of compressed representation.

Another important way of characterizing coders is whether or not they are *static* or *dynamic*, that is, whether or not they support adaptive models. If an adaptive model is being used the probability distribution is subject to continual change, and pre-calculation of codewords is impossible. Different coding methods react in different ways to the demands of adaptive models, and knowledge of the model is an important factor in the choice of coder if compression speed is to be maximized, and this distinction is also explored further below.

Finally, it is convenient to assume that P, the list of the n symbol probabilities, is provided in sorted order, either non-decreasing or non-increasing. There is no situation in which any compression advantage can be derived by assuming otherwise, and the best that can be done in any given implementation is to map the actual symbols being represented—characters, words, and so on—to a sorted list of ordinal identifiers.

2.3 Non-Entropy Coding

Non-entropy codes use the same representation regardless of the probability distribution of the symbols being encoded. The simplest and fastest non-entropy code is to simply use conventional binary numbers, taking $\lceil \log_2 n \rceil$ bits per codeword to code each symbol in an n-symbol alphabet. This code requires just a few words of memory to implement. An improvement in compression efficiency can be achieved if the codewords for the $2^{\lceil \log_2 n \rceil} - n$ most probable symbols are set to be $\lfloor \log_2 n \rfloor$ bits long, and the change has little impact upon memory usage or throughput.

The binary code can, however, be arbitrarily inefficient in terms of compression. For example, consider the probability distribution $p_i = 2^{-i}$ for $1 \le i < n$, and $p_n = 2^{-(n-1)}$. When n is a power of two the average codelength in a binary code is $\log_2 n$ bits, yet the entropy of the distribution is, in the limit, two bits per symbol.

Elias (1975) considered this problem, and developed a class of universal codes loosely based upon binary, but with the interesting property that they are, to within a constant factor, minimum-redundancy (although, as we shall see below, this does not mean that they are within a constant factor of the entropy) for all decreasing-probability symbol distributions. For example, Elias's C_{γ} code represents integer $x \geq 1$ as $\lfloor \log_2 x \rfloor$ in zero-origin unary, taking $1 + \lfloor \log_2 x \rfloor$ bits; followed by $x - 2^{\lfloor \log_2 x \rfloor}$ in zero-origin binary, taking $\lfloor \log_2 x \rfloor$ bits. That is, integer x is coded using $1 + 2 \lfloor \log_2 x \rfloor$ bits. One useful feature of this code is that it is open ended, and there is no requirement for encoder or decoder to know n, the upper bound on the size of the alphabet. It (and the other codes defined by Elias) can thus be used for infinite alphabets as well as finite ones.

Golomb (1966) and Gallager and Van Voorhis (1975) also considered infinite distributions. They described a parameterized code that again allows representation of arbitrarily large values. Given parameter b, integer x > 0 is coded as (x - 1) div b in a zero-origin unary code, followed by (x - 1) mod b in a zero-origin binary code (using the same modification to binary mentioned earlier) for integers $0 \dots b - 1$. For example, with b = 6 the code for x = 10 is "10101", that is, "10" representing 1 in zero-origin unary, and "101" as an adjusted binary code for 3. The special case when b is a power of two is sometimes known as a Rice code (Rice, 1979).

Golomb codes have one rather surprising property. Suppose that some favorable event occurs with probability p—for example, when a biased coin is being tossed and "heads" is a favorable outcome. Suppose also that the tosses are independent. Then the probability of having exactly x-1 unfavorable events before the next favorable one is given by the geometric distribution, $Pr(x) = (1-p)^{x-1}p$. It turns out that if the Golomb parameter b is chosen as $\lceil -\log(2-p)/\log(1-p) \rceil$ then the resulting Golomb code is a minimum-redundancy code (defined below) for the probabilities of the geometric distribution.

Like binary, Golomb and Rice codes can be generated extremely quickly. Indeed, because they have roughly the same computational cost per bit of output as a binary code, they can actually offer faster performance than binary, since on many probability distributions they result in the output of fewer bits. This combination of attributes makes Golomb codes extremely attractive for applications in which bit-vectors of known sparse density must be represented and independence can be assumed. Such applications include image compression, discussed in Section 4, and database indexes, discussed in Section 5.

2.4 Entropy Coding

If the objective is to maximize compression—that is, to minimize the space required by the encoded form—an entropy coder should be used. For clarity, we consider entropy coders in two families. First, we examine the family of minimum-redundancy coders, the classic example of which is Huffman's algorithm. * If integral-length codewords are to be assigned to individual alphabet symbols these coders offer the best possible compression, hence the designation "minimum-redundancy."

Despite the "best" cachet, minimum-redundancy coders can still be arbitrarily inefficient. Consider an alphabet of two symbols, with $p_1 = 2^{-k}$ and $p_2 = 1 - 2^{-k}$. Clearly, any integral-length code for this alphabet must assign "0" to one of the symbols and "1" to the other; and so the average cost is one bit per symbol. The entropy of the alphabet can, however, be made as close to zero as required through an appropriate choice of k.

Near-optimal compression is possible if the integral-length codewords requirement is dropped, and an example of this family of *arithmetic* coders is also described below. Surprising as it may seem, an arithmetic coder can in effect assign codewords almost exactly $-\log_2 p_i$ bits long, using fractions of bits.

2.5 Minimum-Redundancy Coding

Huffman (1952) used the greedy paradigm to design an algorithm that constructs minimum-redundancy codes. Before describing this algorithm, it is helpful to change focus slightly and ask exactly what it is that must be calculated. In the development above we regarded the set of *codewords* as being "the code", and supposed that the process of constructing a code was the process of assigning, to each symbol, a binary codeword.

In general, for such a code to be uniquely decodable, the code must be prefix-free. That is, no codeword may be a proper prefix of any other codeword. (Such codes are also frequently called prefix codes.) Let l_i be the length in bits of the codeword assigned to the ith symbol of the alphabet. A necessary condition for the code to be prefix-free is that $K = \sum_{i=1}^{n} 2^{-l_i} \leq 1$; this is known as the Kraft inequality. At the same time, for the code to be minimum-redundancy, the quantity $B = \sum_{i=1}^{n} l_i p_i$ must be minimized. It is the point of balance between these two—when K is as large as possible, but not greater than 1, and B is as small as possible—that defines the minimum-redundancy code.

^{*}Indeed, this whole class of codes is sometimes referred to as "Huffman codes" because of the early contribution by Huffman. However it is also true that not all minimum-redundancy codes can be generated by Huffman's algorithm, and for this reason we prefer to use the more general phrase "minimum-redundancy" code.

Having a set of codeword lengths that satisfy the Kraft inequality is not, of itself, a sufficient condition to ensure that the code is prefix-free. For example, consider a two-symbol alphabet with codes "0" and "00". The Kraft sum K is just 0.75, but the code is clearly not prefix-free. However, given a list of codelengths l_i that satisfy the Kraft inequality, a prefix-free code with codewords of the specified lengths can always be designed. To see this, suppose that l_i is the given list of codeword lengths, and that $n_{\ell} = |\{i \mid l_i = \ell\}|$ is the number of codewords of length ℓ , for $1 \leq \ell \leq L$ and L the length of a longest codeword. Then the j+1st of the n_{ℓ} codewords of length ℓ should be the ℓ -bit binary integer $j + base[\ell]$, where

$$base[\ell] = \left\lceil \frac{\left(\sum_{k=\ell+1}^{L} n_k \cdot 2^{L-k}\right)}{2^{L-\ell}} \right\rceil.$$

i	l_i	$\mathit{base}\left[l_i\right]$	j	codeword
1	5	0	0	00000
2	5		1	00001
3	4	1	0	0001
4	4		1	0010
5	4		2	0011
6	2	1	0	01
7	2		1	10
8	2		2	11

Table 1: Assignment of canonical code

For example, consider the example shown in Table 1. From the lengths l_i listed in the second column it can be seen that L=5, and that $n_5=2$, $n_4=3$, $n_3=0$, $n_2=3$, and $n_1=0$. Hence, base[5]=0, base[4]=1, base[3]=2, base[2]=1, and base[1]=2. The fourth column lists the values of j corresponding to each symbol i, counting off (from zero) the codewords of a given length. The l_i -bit integers resulting from $base[l_i]+j$ are listed in the rightmost column; these are the codewords assigned.

Observe the regular pattern—all codewords of a given length are consecutive binary integers. This arrangement is known as a canonical code (Schwartz & Kallick, 1964; Connell, 1973; Hirschberg & Lelewer, 1990), and allows fast encoding and decoding using just two L-word lookup tables, one for the array base, and a second to record the first symbol number of each codelength. In particular, note that it is not necessary for either encoder or decoder to maintain a list of all codewords, and so very large alphabets can be handled efficiently.

For the example shown in Table 1, the second array offset would store the values offset [5] = 1, offset [4] = 3, offset [3] = 6, offset [2] = 6, and offset [1] = 9, indicating that the first 5-bit codeword is assigned to symbol 1, the first 4-bit codeword is assigned to symbol 3, and so on.

Given these two arrays and a function next_input_bit() that returns the next bit from the compressed bitstream, decoding a symbol is effected by the following steps:

1. Set $code \leftarrow next_input_bit()$ and $length \leftarrow 1$.

Table 2: Calculating a minimum-redundancy code

```
2. While code < base[length] do Set code \leftarrow 2 \cdot code + next\_input\_bit() and length \leftarrow length + 1.
```

3. Return offset[length] + code - base[length].

The tight loops and highly localized memory reference pattern mean that canonical codes can be decoded very quickly (Zobel & Moffat, 1995; Moffat & Turpin, 1995).

The problem of designing a minimum-redundancy code, then, is the problem of calculating the *length* of the codeword that should be assigned to each symbol. The ultimate need to assign actual codewords should not be allowed to cloud this problem, and in the remainder of this section we suppose that canonical codes will be generated once codeword lengths have been calculated, and that the primary problem is to find these lengths.

Let us now return to Huffman's algorithm. Each symbol i is assigned an initial codelength of $l_i = 0$, and placed as a singleton with weight p_i into a list of packages. This initial arrangement for the weight distribution P = [1, 2, 3, 4, 4, 10, 11, 12] is shown in the first row of Table 2. In Table 2 the bold-face value for each package of symbols indicates the combined weight of that package. Then, within each package, the codelengths assigned so far to the various symbols are indicated by the list of numbers.

At each stage of Huffman's algorithm the two least weight packages are identified and combined. This results in a single larger package, with weight equal to the sum of the weights of the two contributing packages, and with each codelength within the two contributing packages incremented by one and the two sets of codelengths merged. For example, the first two packages combined are $\mathbf{1}(l_1=0)$ and $\mathbf{2}(l_2=0)$; the resulting package is $\mathbf{3}(l_1=1,l_2=1)$. For reasons that will be explained below, we separate original (or leaf packages) from the packages that result from combining steps; the latter are in the second row in each section

of Table 2.

At the conclusion of the algorithm a single package remains, and that package indicates a minimum-redundancy code. In the example of Table 2 the final codelengths are those that were assumed earlier in the example of Table 1. To a purist, an implementation of Huffman's algorithm should build a code-tree, with edges in the tree labeled by "0" and "1" bits; and then read off the codewords by traversing this code-tree. However, doing so results in an assignment of codewords that may not comply with the canonical pattern required for fast table-driven decoding. For example, for the distribution of weights assumed in Table 2 there is no labeling of tree edges in a code-tree that results in the canonical code shown in Table 1. This is why we insist that the output from a codeword-generator should be a list of corresponding codeword lengths.

Table 2 is designed to illustrate the greedy paradigm behind Huffman's algorithm rather than the mechanism by which the method should be executed. The actual implementation proves an interesting exercise in the use of data structures and the design of algorithms. Van Leeuwen (1976) observed that if the input weights are sorted, then maintaining two lists of packages (as shown in Table 2) allows generation of a minimum-redundancy code to be carried out in O(n) time; that is, at a computational cost that grows linearly in the size of the alphabet. Such efficiency is possible because at any stage of the process the two least weight packages will be amongst the two front packages in each of the two lists, and so can be identified quickly. Van Leeuwen also demonstrated that for the case when the weights are not already sorted it is best simply to sort them and then use the method described above. That is, he showed that it must take $O(n \log n)$ steps to generate a minimum-redundancy code for an unsorted weight distribution. This is why, in this section, we focus upon sorted distributions.

Moffat and Katajainen (1995) further refined van Leeuwen's approach. Their implementation not only executes in O(n) time for a sorted input, but, surprising as it may seem, requires almost no extra memory except for the space occupied by the input array of symbol weights. Given a sorted array of n symbol weights, this in-situ method operates in time linear in n and replaces each symbol frequency by the length in bits of the corresponding minimum-redundancy codeword, using just a small constant number of extra words of memory. The method is very fast in practice, and Moffat and Katajainen (1995) give results showing the calculation of a minimum-redundancy code for an alphabet of over one million symbols in under two seconds.

It is also straightforward to demonstrate through an adversary-based argument that at least linear time must be spent developing a minimum-redundancy code, even when the symbol weights are presented in sorted order. Consider the n symbol weights given by $p_i = 3i$, for $1 \le i \le n-2$, and $p_{n-1} = p_n = (\sum_{i=1}^{n-2} p_i) - 1$. Given these weights, every algorithm that solves this problem must assign $l_{n-1} = l_n = 2$, since if it did not, a simple rearrangement of codewords would result in a more economical code. Suppose then that some algorithm has not examined all n values p_i , which must be the case if it claims to operate in sublinear time. Let p_j be one such value that was not examined. Then provided that $p_{j-1} \le p_j \le p_{j+1}$ is maintained, the adversary is free to reassign p_j , since the algorithm has taken no account of the exact value of p_i . Suppose that the adversary sets $p_i = 3j-2$.

Now the minimum-redundancy code must have either $l_{n-1} = 1$ or $l_n = 1$, since again, if it does not, a simple rearrangement reduces the cost. But if an algorithm does not examine p_j , then it is unable to distinguish between the two cases $p_j = 3j$ and $p_j = 3j - 2$, and so it cannot always construct minimum-redundancy codes. Hence, every symbol weight p_i must be inspected, and every algorithm must spend O(n) time.

Given this lower bound, the calculation method of Moffat and Katajainen (1995), and the use of a canonical assignment of codewords, the minimum-redundancy coding problem is thus essentially solved.

There are, however, a variety of related code construction problems for which the situation is less clear-cut. One important special case is the problem of length-limited coding, where a minimum-redundancy code must be formed subject to the additional constraint that $l_i \leq L$ for some specified limit L. A practical application of this style of coding is when, for reasons of speed, an implementation assumes that all codewords can be manipulated as integers on the computer being used (as was the case, for example, in the canonical decoding pseudo-code given earlier). It was not until 1990 that an efficient solution to the length-limited coding problem was described, the package-merge algorithm of Larmore and Hirschberg (1990). That method requires O(nL) time and O(n) space, and although based upon the same greedy paradigm as Huffman's algorithm, it is an order of magnitude slower. Katajainen et al. (1995) refined the implementation of the package-merge algorithm, and showed that the space overhead can be reduced to $O(L^2)$. In practical usage L is typically a small constant such as 32, and so this latter method can be regarded as being "almost inplace". This boundary package-merge method is also sufficiently fast that it is suited to use in practical compression systems. Katajainen et al. (1995) describe experiments in which a length-limited code for an alphabet of over 1,000,000 symbols was constructed in about 180 seconds of CPU time. The 4-bit length limited code for the example shown in Table 2 is l = [4, 4, 4, 4, 3, 3, 2, 2], with cost B = 128 bits; the cost of the minimum-redundancy code is B = 125 bits.

Asymptotically faster algorithms have been developed for the length-limited coding problem (for example, see Schieber (1995) and the references therein). Whilst in the limit operating in sub-O(nL) time for certain combinations of n and L, it is not at all apparent that these algorithms are suited for practical use. Their development is interesting from a theoretical point of view—in contrast to the unrestricted length problem, no non-trivial lower bound to the complexity of the length-limited coding problem has yet been derived—but they seem unlikely to be of benefit to practitioners.

Moffat et al. (1995b) considered another mechanism whereby minimum-redundancy and length-limited coding problems can be specified. They noted that for large alphabets the list P describing the alphabet weights usually contains many repeated elements, and that a much more succinct representation is to record the number of symbols of each given weight. They further noted that if there are r distinct symbol weights w_i , and that each of those weights appears respectively f_i times (so that $\sum_{i=1}^r f_i = n$), then a minimum-redundancy code can be calculated in $O(r \log(n/r))$ time and space, which is sublinear when r is sublinear in n. (Note that this claim does not contradict the adversary-based lower-bound proof given above, since the proof relies upon all of the p_i s being distinct.)

Symbol	Weight	Codeword
A	2	00
В	10	01
$^{\mathrm{C}}$	16	10
D	2	1100
\mathbf{E}	1	1101
\mathbf{F}	3	111

Table 3: Example of a minimum-redundancy alphabetic code

Moffat et al. (1995b) also showed how the same ideas could be applied to the production of length-limited codes, giving an $O(Lr\log(n/r))$ algorithm. The list of 1,000,000 word frequencies mentioned above has r=11,000 distinct frequencies (indeed, the pigeonhole principle means that a stream of N symbols can have at most $r=\sqrt{2N}$ distinct symbol frequencies, and for this data $N\approx 500,000,000$, and a length-limited code can be calculated in just a few seconds.

A further variant of minimum-redundancy coding is alphabetic coding. The additional constraint in this case is that the assigned codewords should be lexicographically ordered in the same way as the original symbols. This allows compressed strings over the alphabet to be sorted without needing to be decompressed. This is one case when a canonical code cannot be used, since it is not possible to assume as input a sorted list of symbol weights. The classic algorithm for this problem is due to Hu and Tucker (1971)[[Glen's DCC paper, 1992? 1993?]]; again, however, there has been recent activity in this area, and Larmore and Przytycka (1994) consider the situation when both a length-limit is in force and the codes must be alphabetic. Table 3 gives an example of a minimum-redundancy alphabetic code for a six-symbol alphabet. This code costs B=80 bits compared with a B=68 bit minimum-redundancy code.

Finally, we note that Larmore and Przytycka (In press) have developed the first parallel algorithm for minimum-redundancy coding that requires sub- $O(n^2)$ total work (time taken multiplied by processors required). This method is based upon a completely novel paradigm for calculating minimum-redundancy codes, and opens the way for further improvement in algorithms for length-limited coding and parallel minimum-redundancy coding.

2.6 Arithmetic Coding

Given that the bit is the unit of stored data, it appears impossible for codewords to occupy fractional bits. And given that a minimum-redundancy code is the best that can be done using integral-length codewords, it would thus appear that a minimum-redundancy code is *optimal*, that is, is as close to the entropy as can be achieved.

Surprisingly, while true for the coding of a *single* symbol, this reasoning does not hold when *streams* of symbols are to be coded. A single code must clearly be an integral number of bits. But suppose that a stream of symbols is to be represented. Provided that the overall message stream is an integral number of bits, there is no requirement that every bit of the encoded form be assigned exclusively to one symbol or another. For example, if

??

five equi-probable symbols are represented somehow in a total of three bits, then it is not unreasonable to simplify the situation and assert that each symbol occupies 0.6 bits. The output must obviously be "lumpy"—output bits might only be emitted after the second, fourth, and fifth symbols of this stream. However, if the coder has an internal state that is part of the compression process, and if after each symbol is coded the state may be changed, then the total code for each symbol is the number of output bits produced plus the "potential" of the changed internal state. Since the latter might be "fractional" in some way, it is quite conceivable for a coder to represent a symbol of probability p_i in $-\log_2 p_i$ bits. At the end of the stream the internal state must, of course, be represented in some way; in effect, by rounding out to an integral number of bits.

Arithmetic coding (Rissanen, 1976; Rissanen & Langdon, 1979; Langdon, 1984; Witten et al., 1987) is an effective mechanism for doing just this. The internal state of the coder is recorded using two variables L and R, recording the Lower end of a bounding interval, and the width or Range of that interval. For the purposes of the discussion here we will assume that these are real-valued numbers between zero and one. In an actual implementation they are integers, and scaled by some appropriate power of two.

Initially L=0 and R=1. The internal potential of the coder is given by $-\log_2 R$. Suppose that $P=[p_i]$ is a normalized probability distribution, so that $\sum_{i=1}^n p_i=1$. Define $low[i]=\sum_{j=1}^{i-1} p_j$, the cumulative probability of all symbols prior to i in the alphabet. It is also useful to suppose that low[n+1]=1 is defined. Then to code the ith symbol of the alphabet, these two steps are executed:

- 1. Set $L \leftarrow L + R \times low[i]$.
- 2. Set $R \leftarrow R \times p_i$.

At the end of the stream the transmitted code is any number c such that $L \leq c < L + R$. By this time $R = \prod_{k=1}^m p_{s_k}$, where s_k is the kth input symbol, and there are m symbols in the stream. The potential has similarly increased to $-\sum_{k=1}^m \log_2 p_{s_k}$, and so to guarantee that it is within the specified range the number c must be at least this many bits long. For example, suppose that some alphabet has n=2 symbols, and that $p_1=3/4$ and $p_2=1/4$. Then a sequence with m=6 and $s_1=s_2=s_3=s_4=s_5=1$ and $s_6=2$ results in L=729/4096=0.17798 and R=243/4096=0.05933. These give, in binary, $L=0.001011011001_2$ and $L+R=0.001111001100_2$, so an appropriate message string unambiguously between these two bounds is c=0.00110, that is, the bitstream "00110". In this case five bits suffice to represent six symbols. This can be verified: $\lceil -\log_2((3/4)^5(1/4))\rceil = \lceil 4.08 \rceil = 5$.

To actually make a workable implementation, several problems must be addressed. The greatest of these is that the process just described requires arbitrary precision arithmetic. If the output file is to be (say) 125 Kb, then L and R must be maintained to about one million bits of precision, a substantial imposition. Fortunately, it is also possible to produce bits on an incremental basis and thereby limit the precision needed for L and R. After each potential-increasing step (above), the following renormalization (or potential-reducing step) should be executed:

3. While R < 0.25 do

- (a) If L + R < 0.5/* first bit of all values c such that $L \le c < L + R$ is a "0" */ $bit_plus_follow(0)$.
- (b) Else, if $0.5 \le L$ then /* first bit all values c such that $L \le c < L + R$ is a "1" */ $bit_plus_follow(1),$ Set $L \leftarrow L - 0.5$.
- (c) Else, /* Polarity of bit is opposite to next bit */ Set $bits_outstanding \leftarrow bits_outstanding + 1$, Set $L \leftarrow L - 0.25$.
- (d) Then, for all cases, set $L \leftarrow 2 \times L$ and $R \leftarrow 2 \times R$.
- 4. To perform $bit_plus_follow(b)$, do:
 - (a) $write_one_bit(b)$.
 - (b) Use $write_one_bit(1-b)$ to output $bits_outstanding$ bits of the opposite polarity.
 - (c) Set $bits_outstanding \leftarrow 0$.

This mechanism maintains $R \geq 0.25$ prior to the coding of each symbol, and so the precision required for L and R is limited to two bits more than the precision to which the probabilities p_i are maintained.

Figure 2 shows the three alternatives catered for at steps 3a, 3b, and 3c respectively. In the first case (Figure 2a) the next output bit is clearly a zero, and the values L and R can be adjusted accordingly. The second case (Figure 2b) allows for the situation when the next bit is definitely a one. The third case, at step 3c and shown in Figure 2c, is somewhat more complex. When R < 0.25 and L and L + R are on opposite sides of 0.5 the polarity of the immediately next output bit cannot yet be known, since it depends upon future symbols that have not yet been coded. What can be known, however, is that the bit after that next bit will be of opposite polarity to the next bit, since all binary numbers in the range 0.25 to 0.75 start either with "01" or with "10". Hence, in this third case, the renormalization—doubling of L and R—can still take place, provided a note is made (using bits_outstanding) to output an additional opposite bit the next time a bit of defined polarity is produced. This why each time a bit is output at step 4, it is followed up by any bits_outstanding still extant.

The decoder must reverse this process. Given a code c, the decoder must determine the sequence of m symbols that resulted in c being transmitted. This is done by the following method. Quantity V is assumed to be the current window into c, of the same precision as used for L and R. The bounds L and R are again initialized to 0 and 1 respectively prior to the first symbol, and V must be initialized to the appropriate number of bits from the beginning of the bitstream c.

- 1. Determine i such that $low[i] \leq (V L)/R < low[i + 1]$.
- 2. Set $L \leftarrow L + R \times low[i]$.

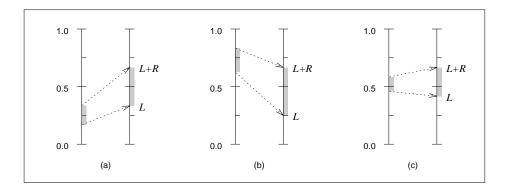


Figure 2: Renormalization in arithmetic coding

- 3. Set $R \leftarrow R \times p_i$.
- 4. Renormalize as in step 3 of the encoder, treating V identically to L, except that each time L is doubled, V should be doubled and a bit from the coded bitstream added into the low order position using next_input_bit().
- 5. Output symbol i.

In a practical implementation for use with an adaptive model the probabilities are subject to continual revision, and so the cumulative probabilities assumed in the array low cannot be calculated in advance. Nor can they be assumed to be normalized. Instead, they must be calculated and updated on the fly. Normalization can be achieved at the expense of extra divisions during the coding process; if c_i is the count of symbol i in a stream of m symbols, then $p_i = c_i/m$ is readily calculated.

The problem of efficiently maintaining cumulative frequency counts so as to handle an adaptive model is a more challenging one. With an alphabet of n symbols, as much as O(n) time might be required if low[i] must be calculated afresh at each coding step. Instead, various structures have been proposed to allow computation of low[i] in sublinear time. These include the move-to-front list organization described by Witten *et al.* (1987), the splay tree approach of Jones (1988), the skewed-tree approach of Moffat (1990b) (which has an access structure analogous to the Elias C_{γ} code discussed earlier), and the balanced tree method of Fenwick (1994).

Of these, only the splay tree and skewed-tree techniques are linear-time. Using either of these structures all required operations can be effected in O(m+b) time, where b is the number of bits produced by the coder and m the number of symbols coded. The balanced tree method (Fenwick, 1994), whilst sub-linear in extreme situations, is also very fast in practice (Moffat $et\ al.$, 1994), and has the added advantage of requiring just n words of memory for an alphabet of n symbols.

The biggest drawback of arithmetic coding is the computation involved. Each symbol coded incurs a cost of several multiplicative operations, and on typical hardware these are expensive. Worse, most of these operations are required even for static probability distributions. For this reason one of the main themes of current research into arithmetic coding is methods by which some or all of these multiplicative operations can be avoided (Rissanen

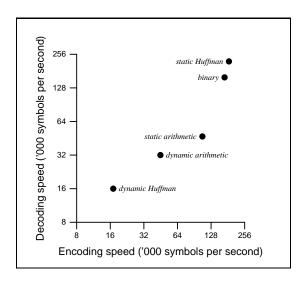


Figure 3: Coding speed, thousands of symbols per second

& Mohiuddin, 1989; Chevion et al., 1991; Feygin et al., 1994; Moffat et al., 1995a). Each of these variants in some way or another approximates the result of the coding calculation, trading exactness (and hence, potentially, compression optimality) for speed. For example in some of these methods R is approximated by a number $R' \leq R$ of a special form, and multiplications and divisions by R are approximated by multiplications and divisions by R', implemented as a small number of shift/add operations.

2.7 Relative performance

The astute reader will have observed that we focussed on static coding methods while discussing minimum-redundancy codes, but concentrated more on adaptive coding when discussing arithmetic coding. This was no accident. Clearly, arithmetic coding can also be used for static probability distributions; and conversely a variety of methods for "dynamic Huffman" coding have been described in the literature (Gallager, 1978; Cormack & Horspool, 1984; Knuth, 1985; Vitter, 1989; Lu & Gough, 1993). However, static arithmetic coding is not much faster than dynamic arithmetic coding, while dynamic Huffman coding is substantially slower than static (canonical) Huffman coding. Figure 3, derived from the results presented by Moffat et al. (1994), shows the relative speed of some of the coding methods described in this section. The test data used was a list of 1,000,000 integers corresponding to distinct words in a sample of English text.

The conclusion to be drawn from Figure 3 is clear: if a dynamic coder is required, arithmetic coding should be preferred; but for static coding, a canonical minimum-redundancy code should be used. Also worth noting is that the difference between these two alternatives is sufficiently great that practical compression systems (see, for example, the discussion of ZIP in Section 3.5 below) operate adaptively at the level of multi-kilobyte blocks, but process each block of data in a semi-static manner, thereby achieving the both the versatility of an adaptive model and the speed of a static coder.

Memory usage requirements reinforce these observations. Even for dynamic models, n words of memory suffice for arithmetic coding (Fenwick, 1994) on an alphabet of n symbols, whereas dynamic Huffman coders require between 10n and 15n words of memory (Moffat et al., 1994). Semi-static coding is memory-economical for both minimum-redundancy and arithmetic coding.

In terms of compression, there is often little to choose between the two classes of entropy coders, particularly if the alphabet is large and the most probable symbol relatively infrequent. In principle arithmetic coding will never give worse compression than minimum-redundancy codes, but because of termination overheads and the need to specify symbol probabilities more precisely there are a range of practical situations in which a Huffman-style minimum-redundancy code can result in slightly better compression than an arithmetic code (Bookstein & Klein, 1993; Moffat et al., 1994). A variety of authors have considered analytic bounds on the inefficiency of minimum-redundancy codes, and shown them to be very accurate for most purposes (Gallager, 1978; Capocelli et al., 1986; Capocelli & De Santos, 1991; Manstetten, 1992).

2.8 Binary Alphabet Coding

One important domain in which arithmetic coding is essential is binary alphabet coding, and methods for this problem are worthy of special mention. When the input alphabet is binary—pixels in a bi-level image, or bits in a text data file—the only coding method that can obtain any compression at all is arithmetic coding. But this environment also shows conventional arithmetic coding (as described above) at its slowest. Multiplications and divisions on a per-input-bit basis are a virtual guarantee of slow throughput.

For this reason binary arithmetic coding has received considerable attention in the literature. To see how this differs from multi-symbol arithmetic coding, consider as an example the situation when $p_1 \geq p_2$ and $p_1 + p_2 = 1$. Suppose that L' and R' represent the values of L and R after the range-narrowing steps in the coder, but before any renormalization. When symbol 1 is coded, L' = L and $R' = R - p_2 \times R$. When symbol 2 is coded, $L' = R - p_2 \times R$ and $R' = p_2 * R$. Hence, if p_2 is approximated by a negative power of two, then all of the operations involved can be carried out by shifting.

The IBM Q-coder (Pennebaker et al., 1988) represents the culmination of a great deal of research along these lines. In the Q-coder it is R that is approximated. The renormalization process is organized so that R is constrained in the range $0.75 \le R < 1.5$. Then, for all multiplications, R is assumed to be 1, thereby obviating the need to actually perform those multiplications. This results in compression loss for some symbols, and a compression gain for others, depending on whether the actual value of R is less than or greater than 1. On average, there is a small but tolerable compression loss.

The Q-coder also incorporates an elegant probability estimation regime (Pennebaker & Mitchell, 1988). Instead of recording symbol frequencies c_i , and using c_i/m as the estimate of p_i , the Q-coder revises its estimations only when a bit is output—that is, when the renormalization loop fires. If the symbol that caused the output bit happens to be the current most probably symbol (MPS, with the less probable symbol being the LPS), the probability of the MPS is increased. On the other hand, if the bit was caused by an LPS,

the probability of the LPS is increased. If the two probabilities were assumed to be equal, the symbol that caused the output bit is taken to be the MPS for the next sequence of compression steps. Probability values are restricted to a small set of allowed values, and all changes in probability are table-driven. In total, each Q-coder (more precisely, QM-coder) "context" requires just eight bits of storage: one to indicate which symbol is the current MPS; and seven to record an index that notes the currently estimated probability. By avoiding the division inherent in frequency-based probability estimation, and by adjusting the probability estimates only once per bit (which, for a binary coder, might represent a vary large number of symbols if compression is good), the Q-coder provides good compression at high speed.

3 Models for Text

Text compression is achieved by forming a model of the text being compressed, and using that model to supply probability distributions to a coder as shown in Figure 1. In the previous section we discussed the coding problem and showed how it is possible to get optimal compression for a given probability distribution using the method of arithmetic coding. In contrast, developing suitable models is something of an art, drawing on ideas from linguistics and statistics, and research into improved compression performance is focussed on designing better models for text.

There are two general approaches to modeling text. One is to use a *contextual* model, in which the probability distribution for each symbol is estimated based on the context in which that symbol occurs. The other is a *dictionary* model, in which a list (or dictionary) of common substrings is maintained, and the text is coded by substituting references to the dictionary. Contextual models generally give better compression performance, while dictionary models are usually computationally less demanding. Current commercial systems most often use dictionary-based methods, but because of increases in computing power, contextual models are becoming increasingly attractive.

3.1 Contextual Models

The best text compression methods are based on the idea of "predicting" a symbol using a small number of immediately preceding symbols as a context. A prediction is simply an estimated probability distribution for what might come next. For example, in one text the context h is followed by an e in 17,470 out of 38,398 occurrences of the context, or 45.5% of the time. If this is used as a probability estimate for the letter, then an e will be coded in $-\log_2 0.455 = 1.14$ bits (assuming that an arithmetic coder is being used). The decoder uses exactly the same model as the encoder, and can therefore recover the character from the bits transmitted by the arithmetic coder. The best compression is obtained if the character that actually occurs in the context has a high probability. The challenge of modeling is to make this happen frequently.

One way to make more accurate predictions is to use a larger context. For example, in the context of the two characters th, an e occurs 56.5% of the time, and would be coded in only 0.82 bits using this model.

A model that makes predictions based on the previous n symbols is referred to as an order-n finite context model, where $n \geq 0$. Even the trivial case of an order-zero context can provide reasonable compression in some circumstances, but usually higher-order contexts are needed to get good compression. It might seem desirable to set n as large as possible. However, if the context is too large then it becomes unlikely that it has ever been seen in the sample of text on which the probability distribution is based. Also, the prediction probabilities will depend very strongly on the particular sample text. In practice, the best approach is to use a variety of context sizes, and have the model switch between them depending on the level of confidence in the predictions of the various contexts. One of the best text compression techniques, the PPM method (described below), uses this principle.

The above description has ignored the issue of finding a suitable sample of text from which to estimate statistics. There are three main possibilities: static, semi-static, and adaptive modeling. In a static model, statistics are measured using some representative sample of text, and then remain fixed for any future data that is compressed. This approach is not widely used in practice because it is not usually possible to be sure that the sample will be representative of all data that will be encoded in the future. A semi-static model avoids this problem by estimating the statistics from the data that is being compressed. This requires two passes over the data, one to glean the statistics and another to encode the data. It also requires that the model is stored or transmitted with the text. This kind of model is appropriate when the text itself is static. For example, a large amount of text stored on CD-ROM can be compressed very effectively by forming a semi-static order-zero model of English words in the text (Zobel & Moffat, 1995).

The most versatile compression systems use adaptive modeling, where the statistics are estimated from the data that has already been encoded. The decoder is able to calculate identical estimates to the encoder because it will have decoded the same symbols. In practice, the model is usually updated immediately after each symbol is encoded and decoded, rather than being rebuilt from scratch each time. An adaptive system will give poor compression towards the beginning of a file because the statistics are based on a very small sample. However, later on in the file the compression will improve greatly, to the extent that the start-up overhead will typically be similar to the cost of prepending the model that would have been formed in a semi-static situation (Cleary & Witten, 1984a). Thus, adaptive modeling gives similar compression performance to a semi-static model, but only requires a single pass over the data.

Context-based models are usually linked to an entropy coder. For reasons discussed in the last section, an arithmetic coder is most suitable for adaptive modeling. Minimum-redundancy coding is generally more suited to semi-static models, because the code is only generated once for each file being compressed, compared with an adaptive model, which can require a new code to be generated every time the statistics change. Minimum-redundancy coding is particularly suited to the order-zero word based model mentioned above, because the model contains relatively low probabilities, and so the coder compresses at a rate close to the entropy of the alphabet.



Figure 4: A coding step for the LZ77 compression method

3.2 Dictionary Models

Dictionary models use a paradigm where several consecutive symbols are replaced by a single code. The "dictionary" is a list of phrases available for substitution. These phrases are typically common substrings, and the dictionary construction can be static, semi-static, or adaptive.

By far the most widely used dictionary compression methods are adaptive. In an adaptive situation, the dictionary is constructed using the previously encoded text. Initially the dictionary is empty, or contains some trivial substrings such as the symbols in the input alphabet. As the text is compressed, "useful" strings are added to the dictionary, and in some cases entries that are not likely to be productive are removed.

Almost all adaptive dictionary methods fall under the general umbrella of "Ziv-Lempel" coding, a family of methods that derive from two seminal papers by Ziv and Lempel (Ziv & Lempel, 1977; Ziv & Lempel, 1978). Both papers describe adaptive dictionary methods, although the techniques have significant differences. They are sometimes labeled LZ77 and LZ78, or LZ1 and LZ2, respectively.

The LZ77 family of methods uses a sliding window of recently-seen symbols as a dictionary, as illustrated in Figure 4. The window typically stores 1 to 8 Kbytes of characters immediately preceding the current encoding point. The coder searches the window for the longest match for the next few characters. The match is coded by transmitting the offset from the coding point to where the match occurs, and the length of the match. For example, in Figure 4 a match for four characters has been found, starting seven characters back from the encoding position, so the pointer (7,4), representing the offset and length, would be transmitted. The coded characters are then added to the window, and the same number of "old" characters are removed from its other end.

Decoding is very efficient for LZ77 methods, because the decoder simply looks up the matched substring in its own window and copies the indicated characters. This makes the technique particularly suitable for "distribution" situations, where files are encoded once, distributed to many locations, and decoded many times.

Two of the factors that distinguish members of the LZ77 family from each other is how the pointers are encoded, and what happens if no suitable match is found. A simple approach to coding is to represent pointers into a window of size N using $\lceil \log_2 N \rceil$ bits. However, considerable improvements in compression can be achieved by using shorter codes for pointers to recent characters and longer codes for ones that reach far back into the

window. Likewise, the match length could be coded by setting a maximum length of F, and coding the length in $\lceil \log_2 F \rceil$ bits, but shorter matches are much more common than longer ones, and better compression is achieved by varying the code length.

Sometimes, particularly at the beginning of coding, no match will be found for the characters being coded, or the match may be so small that the pointer would consume more space than the characters it replaces. LZ77 methods typically require a minimum match length of two or three characters to make coding worthwhile. If no suitable match is found in the window, there are three main techniques for ensuring that coding can proceed. The method used in Ziv and Lempel's original paper (Ziv & Lempel, 1977) is to require that a character is always transmitted after each pointer. Better compression is achieved by transmitting a character only when necessary. This requires transmitting an extra bit at each coding step to indicate whether the next code represents a pointer or a literal character. Storer and Szymanski (1982) and Bell (1986a) give examples of this technique. A third method takes advantage of the observation that novel characters tend to occur in groups. In this method, a code is transmitted to indicate the number of non-matching characters, which are then transmitted consecutively (Fiala & Green, 1989).

Most LZ77 methods assume that greedy searching is used to perform matching, that is, the *longest* match for the next few characters is used. This is not necessarily optimal because taking a long match may preclude a better match that is coming up. Some systems will evaluate the benefit of coding the next character (or more) explicitly, to see if a longer match is found starting at a later position in the input. If taken too far, this slows down the encoder considerably, but most of the compression improvement can be achieved by looking ahead just one step (Horspool, 1995).

Fast searching data structures can be applied to the encoding phase, with particularly good performance being achieved by hashing. The combination of very fast decoding, fast encoding, and relatively good compression performance has made LZ77 techniques very popular as general purpose compression methods, and they currently enjoy wide popularity because they work efficiently even on modest personal computers. LZ77 is used by general purpose compression programs, file archiving software, and on-the-fly disk compression systems. One particularly fast LZ77 variant is the LZRW1 technique (Williams, 1991).

They build an explicit dictionary by breaking the input up into phrases and storing them in a data structure for rapid matching. Usually the phrases are non-overlapping, so fewer dictionary strings are available for coding than for LZ77 methods. Although this means that fewer substitutions are possible, the codes will be shorter. Overall, LZ78-based methods tend to give worse compression performance than corresponding LZ77-based ones, and decoding is more expensive because the decoder must build up its own dictionary. However, encoding tends to be faster because less work is required to maintain and search the dictionary.

Ziv and Lempel's original LZ78 method (Ziv & Lempel, 1978) is illustrated by the example in Figure 5. The previously coded characters are broken up into phrases (numbered underneath) using a simple heuristic, and the coding step searches them for the longest phrase that matches the characters that are about to be encoded. The commas show how

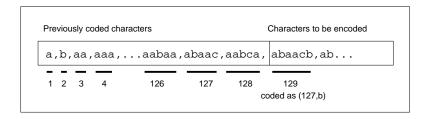


Figure 5: A coding step for the LZ2 compression method

the previous text has been broken up into phrases for the dictionary. In this case phrase number 127, abaac, is the longest match for the upcoming characters. It is represented by transmitting the number 127, followed by the character immediately following the matched ones, which in this case is a b. The string just transmitted (aabaacb in the example) is then added to the dictionary. In other words, the parsing algorithm is simply to add one character to an existing phrase from the dictionary. The algorithm can be implemented naturally using a trie data structure (Langdon, 1983).

As for LZ77, many variations of the original LZ78 method exist, using different parsing heuristics and different methods for representing the matches. Some of the more widely used LZ78 variants are based on a method proposed by Welch (1984), dubbed LZW. The LZW method eliminates the explicit character in LZ78's (code, character) pair. This is done by initializing the dictionary to contain all the characters in the alphabet, which guarantees that a match will always be found—even if only a single character is matched. New phrases for the dictionary are created by extending a coded phrase by appending the first character of the next phrase to it.

This is illustrated in Figure 6, where the commas show the groups of characters that have been coded as single phrases. Under each phrase is shown the new one constructed from it, which is done by adding the first character from the next phrase. The longest phrase that matches the characters about to be coded is number 109, aabaa. This is transmitted, and the new phrase, aabaaa is added to the dictionary as number 112. One catch to this approach is that the decoder cannot construct a phrase until one step after it has been used, since it will not have the first character of the next phrase. For example, if the next coding step found that phrase number 112 was the longest match, the decoder would not have the phrase available. Fortunately there is a simple solution to this problem, as the decoder does not need to know all of phrase 113 to be able to construct phrase 112, but only the first character. This is already known—it must be the same as the first character of phrase 112, and so decoding can proceed despite the apparent circular reference.

3.3 Model equivalence

Although we have distinguished between context-based methods and dictionary methods, the line between the two is blurred. For example, it is possible to use contexts to predict dictionary entries (Gutmann & Bell, 1994; Hoang et al., 1995), and it is possible for context-based systems to use strings as their symbols (such as the word-based coder mentioned

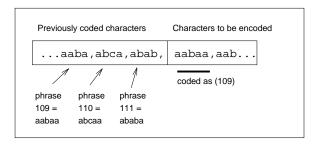


Figure 6: A coding step for the LZW compression method

earlier).

Furthermore, considerable work has been done to examine a duality between the two approaches, using the techniques of blocking (combining the coding of several consecutive characters), and decomposition (breaking the coding of one symbol into several steps). A context-based method can be converted to a dictionary method by blocking. As the blocks get larger, dependencies between blocks become less significant, and the dictionary method becomes a good approximation to the context-based one. However, the dictionary becomes huge, and will not usually be practical to implement.

Conversely, several techniques have been given for decomposing the phrases in a dictionary method into discrete symbol codes. The code lengths for the individual symbols in a phrase are chosen so that they add up to the same length as the code for the phrase in the dictionary method. Decompositions have been found for the original LZ78 method (Langdon, 1983), LZ77 (Bell, 1986b), and for some more general cases (Rissanen & Langdon, 1981; Bell & Witten, 1994). Again, this transformation does not lead to practical methods, but it demonstrates a close relationship between the two approaches. Usually a decomposition exposes inefficiencies in the dictionary method, which help to explain why context-based methods usually give better compression performance.

In general, dictionary-based methods are faster because they process several characters at once, while context-based methods give better compression because they make good use of contextual information.

The following sections describe some practical methods based on the modeling techniques described above.

3.4 The PPM algorithm

PPM stands for *Prediction by Partial Matching*, a method first proposed in 1984 (Cleary & Witten, 1984b). It is an adaptive method based on context models, and gives extremely good compression performance on a range of file types, particularly text files. On files of English text it typically achieves compression rates of a little over 2 bits per character. Its main drawback is that it is considerably slower than the fast Ziv-Lempel methods.

The general principle of PPM is to try to use a long context to make a prediction, but to switch to a shorter one if the context has not been seen before, or if the next character has not been seen in that context. The context is truncated repeatedly until the character

Previously coded characters	Characters to be encoded
abccabccbcbccacaabaabc	aba

Figure 7: A PPM coding step

Context		Character	Total count
		counts	
order-4	aabc	0	
$\operatorname{order-3}$	abc	2	c(2)
$\operatorname{order-2}$	bc	4	c(3), b(1)
$\operatorname{order-1}$	c	8	c(3), b(2), a(3)
order-0		22	c(9), b(6), a(7)

Table 4: A table of contexts for the coding step in Figure 7.

can be predicted, at which point it is coded. The predictions made by PPM can be coded conveniently using arithmetic coding.

Figure 7 shows what happens to encode one character using PPM. The system is about to code the character a, and the context is ... abaabc. PPM starts with some maximum context length, typically about four characters. In the case of the example, this would give the order-4 context aabc. The encoder searches a data structure built from the previously encoded text for earlier occurrences of the context. If none is found then it automatically shifts to an order-3 context. The decoder can do the same, since it has seen exactly the same prior text. Eventually both encoder and decoder will settle on the largest context that has occurred previously, without any communication between the two being necessary. Table 4 shows the number of times each context has been seen for the example in Figure 7. The order-3 context, abc, is the largest one that has been seen before, and so it will be chosen initially to code the next character. Both encoder and decoder maintain a count of the characters that have occurred in this context. In the example, only the character c has occurred, twice. This would seem to require that a c should be predicted with a probability of 100%, but that would prevent any other character from being coded, and in the example, would make it impossible to code the upcoming a because it has never been seen in this context before. To allow for this situation, an additional, notional, character is added to the context, called an escape character. The encoder transmits an escape signal to indicate to the decoder that this context cannot be used, and both shift down to the next smaller context. For the meantime we will give the escape character a count of one, artificially inflating the number of times the order-3 context has been seen to three, and giving the escape character a probability of one third. If a c were being coded, it would be represented using the remaining two-thirds of the probability space.

At this stage in the example, both the encoder and decoder are now using the order-2

context. This context can be used to predict a c or a b, but not an a. Another escape is called for, but first note that a c would never be coded in this situation, because if the next character were a c then it would have been coded in the order-3 context. It is therefore pointless to count the c in the order-2 encoding, and so we end up having codes for only the escape character and the character b. This simple improvement is called exclusion, and it will always improve compression performance because the probabilities in the distribution increase, so the code sizes decrease. If the escape character is given an artificial count of one, each of the two characters will have a probability of one half (that is, they would be coded in one bit by the arithmetic coder).

Once the encoder reaches the order-1 context, the character a can be encoded because it has a non-zero count. In this context, b and c will be excluded, and the only characters in the probability distribution will be a and the escape. The escape is necessary because the decoder needs to allow for the possibility that a different character might occur. If the count for an escape is one, then the a will be coded with probability 3/4.

If a new character, d say, were to be encoded, PPM would escape right down to an order-0 context, then escape from that context and encode the new character using a uniform probability distribution over all possible characters. This will occur frequently near the beginning of a text, but escapes will seldom be used once sufficient text has been seen to build a good model.

An important design decision is how escape probabilities are determined. This involves estimating the probability that something previously unseen is likely to occur, which is referred to as the zero-frequency problem (Witten & Bell, 1991). Several approaches have been proposed, including the simple one mentioned above of allocating a count of one to the escape character. A method that seems to consistently work well in practice is based on the number of distinct characters already seen in the current context. If r distinct characters have been seen in a total of n occurrences of the context, then the probability of an escape is estimated to be r/(n+r) and the remaining probability of n/(n+r) is distributed proportionally over the other characters. This approach has been dubbed "method C," and the version of PPM that uses it is called PPMC. A full description of PPMC, including a data structure for implementing it efficiently, is given by Moffat (1990a). The PPMC method typically gives compression of just over 2 bits per character for English text. A further method "D" has since been described (Howard & Vitter, 1992) that consistently yields slightly better compression than method C.

Cleary et al. (1995) describe a variation of PPM called PPM*, which works with arbitrarily long contexts. A heuristic is used to determine how long the context should be for each prediction. Burrows and Wheeler (1994) have developed another algorithm which at first sight appears to be quite different, but is shown by Cleary et al. (1995) to be a semi-static equivalent of PPM*. The Burrows and Wheeler method does, however, have the advantage of being much faster in practice.

3.5 The ZIP algorithm

ZIP is an algorithm used in a number of freeware compression programs primarily made

available by the Info-Zip group[†]. Programs that use it include zip, gzip, pgp and most PNG implementations. The pkzip system uses the same file format, and therefore similar encoding and decoding algorithms.

The ZIP method is a highly optimized variation of the LZ77 compression method, and uses relatively modest processing resources to achieve very good compression. The tradeoff between speed and compression can be altered by the user, making it useful in a variety of situations.

ZIP uses a hash table to locate previous occurrences of strings. The next three characters to be coded are hashed, and the resulting value is used to look up a table entry. This is the head of a linked list that contains places where the three characters have occurred in the window. The length of the linked list is restricted to prevent search time from growing too much—at the expense of a small loss in compression. Since the limit on list length can be chosen at encoding time, the user can trade speed against compression. Having a limit is particularly important if the input contains long runs of the same character, because the runs will produce a very long list of references. Long lists are time-consuming to maintain, and are usually unnecessary because the first few items in the list will more often than not be sufficient to find a good match. Compression can be improved slightly by storing recent occurrences at the beginning of the linked list, in order to favor recent matches. Matches are represented in the encoded file by a pointer consisting of an offset and a length. If no suitable previous occurrence can be found, a "raw" character is transmitted. The offset of a pointer is represented using a minimum-redundancy code, so that more frequent offsets (usually recent ones) are coded in fewer bits. The match length is represented by another prefix code; and this code is also used when raw characters are to be represented. It may seem contradictory to combine the match lengths and raw characters into one code, but this can give better compression than separating them, because in this latter case an extra bit would need to be transmitted to indicate whether the next input is a match length or a character, whereas the combined code can use less than one bit on average to send this same unit of information. The match length is sent before the offset of a pointer so that the decoder can tell whether a pointer or a raw character is being transmitted.

If the user specifies that compression is more important than speed, ZIP uses a simple form of non-greedy parsing to improve compression performance by transmitting a raw character even though a pointer could be used. This occurs when the use of a raw character gives a better match for the characters immediately following the one about to be coded.

The codes for ZIP are generated semi-statically. Blocks of up to 64 K bytes are processed at a time. The appropriate canonical codes are generated for the pointers and raw characters, and a code table is prepended to the compressed form of the block. This means that ZIP is not really a single-pass method. However, the blocks are small enough to be held in memory. As a result, the input file need only be read once, and the program behaves as if it operates in one pass.

Because of the fast searching algorithm and compact output representation based upon minimum-redundancy codes, ZIP outperforms most other Ziv-Lempel methods in terms of both speed and compression effectiveness. It typically compresses files of English text to

[†]http://quest.jpl.nasa.gov/Info-ZIP

just under 3 bits per character. This is remarkably close to the performance of the better context-based models, and yet ZIP runs very quickly.

3.6 The COMPRESS algorithm

The Unix utility Compress is one of the more widely-used variants of the LZW method. It began as an implementation of LZW, but was tuned to make it suitable as a general-purpose compression system. Its widespread popularity bears testimony to its high performance, although recently it has been displaced by ZIP, which can get better compression and speed performance under most circumstances, particularly if decoding throughput is important.

In Compress, the number of bits used to represent phrases is gradually increased during encoding, so that only just enough codes are available to represent the number of entries in the dictionary. Compress also places a maximum on the number of phrases that are constructed, which in turn limits the amount of memory required for coding. When the dictionary is full, adaptation ceases. Compression performance is monitored, and if it deteriorates significantly, the dictionary is cleared and rebuilt from scratch. The dictionary is stored in a trie data structure, but access to the trie is made very fast by using hashing. The child of a node in the trie is found by hashing the node number and the label on the branch that is being followed.

The combination of hashing and the fine tuning of the codes makes Compress relatively fast, and yet it gives good compression. It typically reduces files of English text to around 3.5 bits per character. This is well under half their original size, but a long way from the performance of methods like PPM. However, it is now being superseded by ZIP which, with the right parameters, can give similar performance, but can also be used to give either better compression or better speed by altering the extent of its search.

3.7 Relative performance

The ultimate test of the usefulness of a compression program is its performance. Figure 8 shows the behaviour of several of the methods described here on a two gigabyte collection of English text. The figure plots decoding speed (in megabytes per second, on a Sun Microsystems SPARC 10 Model 512) versus the compression rate achieved (measured as the percentage remaining after compression). In most cases encoding speed is similar to decoding speed. The ZIP options were set for speed rather than compression effectiveness, and slightly better compression is possible if the alternative setting is made. The HUFF-WORD method is a zero-order word-based model using semi-static canonical coding (Zobel & Moffat, 1995).

There is a clear tradeoff between throughput and compression efficiency. The best method is PPM, but it is also the slowest. The fastest method is *null*, that is, no compression at all and a straight file copy. The best of the LZ78 methods—Compress—is outperformed by ZIP in terms of both compression efficiency and speed. To be interesting, future methods must lie outside the "lower-right" envelope on Figure 8, or have some other attribute to recommend their use.

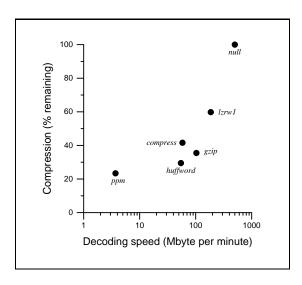


Figure 8: Compression performance and decoding rate

3.8 The future of text compression

Each year researchers find slightly better adaptive text compression methods, although this is mainly achieved through fine tuning, as the main breakthroughs were made in the late 1970s (Ziv-Lempel coding and arithmetic coding) and the early 1980s (context models such as PPM). It appears to be particularly difficult to break the barrier of compressing in less than 2 bits per character for "typical" English text.

Some early experiments by Shannon (1951) and Cover and King (1978) attempted to find a lower bound for the compressibility of English text by having human subjects make the predictions for a compression system. These authors concluded that English has an inherent entropy of around 1 bit per character, and that we are unlikely ever to be able to compress it at a better rate than this.

Models based on finite-state machines have been proposed (Cormack & Horspool, 1987; Teuhola & Raita, 1993), but to date these appear to be no more powerful than context-based models (Bell & Moffat, 1989; Bunton, 1995). Recent innovations include dictionary-based methods that store multiple dictionaries, and choose which one to use based on the current context (Hoang et al., 1995), and methods based on non-greedy parsing (Horspool, 1995). However, these only achieve small improvements. More powerful models based on grammars and semantics have been touted, but practical systems are yet to emerge in the quest for sub-two bit per character compression.

4 Models for Images

There are three principal domains in which lossless compression methods are used for image transmission. The first comprises bilevel images—notably faxes—for which several standards have been developed. The second is for specialist applications of grayscale images (or ones with a limited color palette), such as medical X-rays, where the image informa-

tion is subtle and the consequences of error so dire that users are not prepared to accept any chance of loss. The third is simply for reasons of convenience: in practice images are frequently stored on computers, and transmitted between them, in losslessly compressed formats because it is just not worth worrying about the extent of approximation that can be tolerated. Lossy techniques are used for most other image applications, because as the amount of data associated with each pixel grows from 1 bit for bilevel images to a few bits for grayscale images to several bytes for full-color images, the improvement in compression that results from accepting lossy representation increases and the perceptual effect of slight degradations tends to decrease.

In the late 1970s, the international communication standards body CCITT recognized the potential of emerging technologies for facsimile transmission and sought a standard for fax. This was finalized in 1980 as the "Group 3" standard and forms the basis of the fax machines that are in everyday use. The standard incorporates simple data compression techniques based on run-length coding, prefix coding, and differential coding to exploit line-to-line coherence (Hunter & Robinson, 1980). Shortly thereafter the "Group 4" standard was published, which is essentially the same as Group 3 but is intended for use on digital networks rather than conventional analog telephone circuits. Both these techniques are described in the next section.

During the 1980s, the gradual adoption of arithmetic coding removed any artificial constraints on what kinds of models could be used, prompting researchers to begin to investigate the use of context models for bilevel image compression (Langdon & Rissanen, 1982). As explained earlier, context models of text condition the probability distribution of the upcoming character using the previous few characters. For each different context that occurs, both encoder and decoder maintain a next-character probability distribution. The actual next character is coded with respect to this distribution using arithmetic coding, and recovered without error by the decoder since it has access to exactly the same distribution. This works for bilevel pictures too, as described in Sections 4.2 and 4.3. Context modeling forms the basis of a recent CCITT/ISO standard for lossless compression of images called JBIG, which is covered next.

For continuous-tone—that is, grayscale and color—pictures, some lossy compression methods incorporate lossless modes in which the difference between the lossy version of the picture and the original is somehow coded and transmitted. For example, the recent JPEG standard reproduces continuous-tone pictures with excellent image quality at around 1 bit/pixel (Pennebaker & Mitchell, 1993). One bit per pixel—the starting point for fax—is impressive compression for grayscale or color images, which generally have anything from 8 to 36 bit/pixel when digitized. Although it is a "lossy" method that does not reconstruct the original image exactly, JPEG does include an option of lossless compression for use in situations where this is felt necessary.

Lossy image compression is generally based on quite different principles to lossless methods: using, for example, spectral analysis rather than context-based prediction. However, there has recently been a surge of interest in lossless methods of image compression. We describe the popular GIF format, which is based on Ziv-Lempel coding of a sequence of pixel values and is widely used for compressing images in practice, in Section 4.5. Specially

designed methods that take into account the two-dimensional context give much better results, and we describe a method developed specifically for lossless compression of grayscale pictures in the following section. Despite being far simpler than lossless JPEG, its performance compares very favorably. Finally, in Section 4.7, we outline a much more elaborate method that has been introduced very recently and represents the current state of the art in compression performance.

4.1 The CCITT facsimile standards

The CCITT standards begin by specifying details like paper size and scanning resolution. A one-page document at standard 200×100 dpi resolution contains approximately 1700×1200 black-or-white pixels, or 2 Mbit of information before compression. The transmission rate is normally 4800 bit/sec, which at the time the standard was introduced was what could comfortably be achieved over an ordinary telephone line. If no compression were used it would take 430 seconds, or about seven minutes, to send a one-page document. This contrasts with an average time, for typical documents, of around one minute using the Group 3 standard.

The standard specifies two coding methods: a one-dimensional scheme that treats each scan line independently, and a two-dimensional one that exploits coherence between successive lines. Basically, the latter scheme identifies the positions along each line at which the image changes from black to white, or from white to black, and codes them relative to the corresponding positions on the previous line. To get things started the one-dimensional scheme is used to send the first line.

Each scan line is regarded as a sequence of alternating black and white runs. Lines are assumed to begin with a white run so that the receiver can maintain synchronization; lines that start with black are treated as though they begin with a zero-length white run. In the one-dimensional scheme, a line is represented by coding the length of each run using a pre-specified, non-adaptive prefix code. Separate code tables are used for black and white runs because their statistical distribution is different—textual images are normally black on white, and black runs of a few pixels are much more common than white runs of the same length. Each code table can represent a run-length value of up to 63 pixels, and special "make-up" codewords allowed groups of multiples of 64 pixels to be handled. The code tables were obtained by optimizing the codes for a particular set of test documents.

Coded lines are terminated by a special end-of-line codeword. This is chosen to be a unique binary sequence that cannot occur in a valid line of coded data. Even if transmission errors corrupt the scan line data and destroy the receiver's synchronization with the codewords, the end of the line will still be detected reliably.

The two-dimensional code is a line-by-line scheme in which one row of pixels, the "current line," is coded with respect to the previous, already-transmitted, row, the "reference line." This is called "vertical mode." The position of each color change (white to black or black to white) in the current line is coded with respect to a nearby change position (of the same color) on the reference line, if one exists. "Nearby" is taken to mean within three pixels, and so the vertical-mode code can take on one of seven values: -3, -2, -1, 0, +1, +2, +3. If there is no nearby change position on the reference line, the ordinary one-dimensional

code—called "horizontal mode"—is used. A third condition is when the reference line contains a run that has no counterpart in the current line, and then a special "pass code" is sent to signal to the receiver that the next complete run of the opposite color in the reference line should be skipped.

This two-dimensional coding scheme is exceedingly susceptible to errors, for any transmission error in a line will likely propagate all the way down the rest of the page. Consequently some lines are transmitted using one-dimensional coding, and the two-dimensional code is only used for the remaining lines. The Group 3 standard provides for at least every kth line to be transmitted using the one-dimensional method, and k is set to 2 so that an error can destroy at most two consecutive lines. The standard also includes a high-resolution mode with twice the vertical dot density $(200 \times 200 \text{ dpi})$, and for this k is set to 4 so that greater compression can be achieved. Errors can then destroy as many as four lines, but this corresponds to only two lines at standard resolution.

The compression improvement of Group 3 two-dimensional coding is limited by the fact that every kth line is transmitted using one-dimensional coding, to localize the effect of transmission errors. In digital transmission systems where errors are detected and corrected at a lower level of protocol, this is unnecessary: the two-dimensional scheme can be used for every line on the page, assuming an all-white reference line at the beginning to get things off the ground. This is the "Group 4" facsimile standard. Thus only real difference between the Group 3 and 4 methods is that the latter dispenses with features that are designed to give robustness in the face of transmission errors, on the assumption that these are corrected by lower-level protocols. Because error correction and detection ability implies redundancy of representation, the omission of these gives rise to a more compact encoding, and hence faster transmission. Further information on the standards, and results for compressing the CCITT test images using them, can be found in (Pratt $et\ al.$, 1980; Yasuda $et\ al.$, 1985; Bodson $et\ al.$, 1985).

4.2 Context models

A context model conditions the probability distribution of a pixel being black or white on which of the preceding pixels are black and which are white. Whereas in text it is clear that the best letters on which to condition the distribution are the directly preceding ones, for pictures we must decide whether to use the pixels above the current one or to the left of it—or, more likely, both.

The best plan is to use as context a template of pixels surrounding the current one but preceding it in transmission order. Experiments have shown that good results are achieved with the templates shown in Figure 9. A black dot marks each pixel included in the template and a bull's-eye marks the position of the pixel about to be coded. The gray pixels are ones whose values are not yet known by the decoder and so cannot be included in the context, and the open circles indicate pixels whose values are known but are not included in the context.

For example, if there are seven pixels in the template, there are at most 128 different contexts. The number of occurrences of a white pixel and a black pixel in each context are recorded—256 counts in all. Then, during coding, the pixels under the template are used as

the context to determine which counts to use. This can be implemented by concatenating the seven bits together to form a context number between 0 and 127, which is used to index the list of occurrence counts. The counts are used to drive an arithmetic coder as described in Section 2.6. Note that this is one application in which arithmetic coding really is required—no prefix-free code can represent the symbols of a binary alphabet in fewer than one bit per symbol, and blocking to make runs and thereby enlarge the alphabet is not effective because the probability of a 1 is different in every context.

Sixteen bits is sufficient to store the counts, for nothing is gained by making them more precise, and all counts can be halved whenever one threatens to exceed $2^{16} - 1$. Eight bits may be insufficient, for often much of the image is white, and performance can be degraded quite noticeably because the maximum probability that can be represented is 255/256. The counts can be predetermined based on analysis of representative images, or computed individually for each image and transmitted to the decoder before coding begins. Alternatively, they can be accumulated adaptively and transmitted implicitly, as described in Section 2.1. Under the adaptive scheme, each pixel is coded according to the current set of counts and then used by both encoder and decoder to update the appropriate count. This raises the problem of how to code a pixel in a context in which it has never occurred before. The simplest solution is to start all counts at 1 instead of 0, and to round up when halving.

Figure 9 shows the results achieved using adaptive coding with templates of different sizes on a set of 80 test images, ranging from line art (mostly white) to full-detail scanned images. (Ignore the segments to the right of the lower bars for now.) The values listed are the mean ratio of original size to compressed size, averaged over the 80 images. For each size, the template shown is the one that yields the best overall compression for the test suite. Using templates with seven or more pixels, this method generally outperforms the CCITT Group 3 and 4 standards. Context-based compression can also be used for gray-scale images by treating them as related series of bit-planes and using a three-dimensional context in which to predict each bit (Tischer et al., 1993).

4.3 Two-level models

The tradeoff between the prediction accuracy attained and the "learning" cost of using a large model can be clearly seen in Figure 9. For models based on templates of fewer than about 12 pixels the learning cost is small, and so compression improves if the template is enlarged. With 12-pixel templates there are 8192 counts, two for each possible template, and with an average of 400,000 pixels in each test image there is a good chance that most different contexts will occur reasonably often—often enough to make a good model. On the other hand, when more pixels are used in the template the model does not converge to a useful state within the number of pixels being encoded, and so compression degrades as the template grows.

The learning cost of a large model can be quantified by determining the improvement in compression when the final values of the counts are used as a static model to compress the same image on a second pass. This measures the "self-entropy" of the image according to that template. It is not a physically possible compression method because it does not take into account the information needed to specify the set of counts, which is very large—16 megabytes for 16-bit counts and a 22-bit template. Rather, it is an upper bound to the amount of compression that can be obtained when using that particular template. The incremental improvement is shown in the right-hand bars in Figure 9 for the templates with 12 or more bits. It is miniscule for the smaller templates.

The 18- and 22-bit templates offer a very detailed context in which the next bit can be

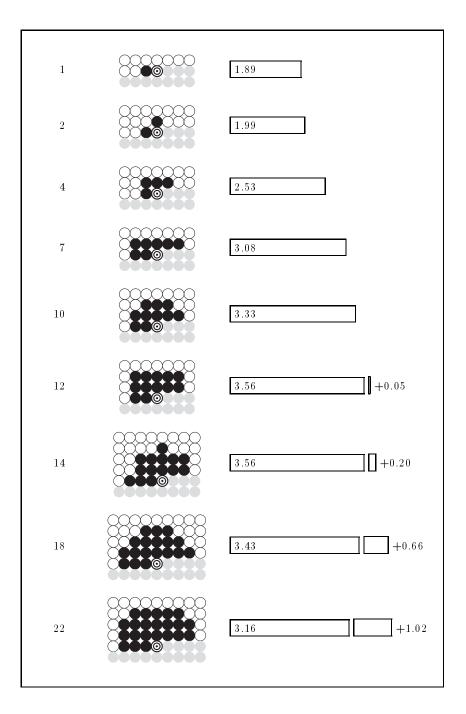


Figure 9: Compression factors for context-based compression

predicted, but each of these large contexts needs to be initialized before it can be relied on, and this is why compression with these templates is actually worse than it is with the smaller 12- and 14-bit models. To learn the counts for a large template without paying a high learning cost, a two-level coding scheme can be used (Moffat, 1991). This is done by coding each pixel in the full context only if that context has already been observed a certain number of times before. If, because of insufficient prior occurrences, the full context is not regarded as being a reliable predictor, a subset is used to generate a smaller template. Figure 10 shows the compression that can be achieved with suitable two-level templates. The pixels used for the subordinate context are black, and the extra pixels for the larger context are gray. Again, the templates shown are those that give the best results out of a wide range of templates that were tested.

Small improvements are obtained by two-level coding for the 12 and 14 pixel templates, but the overall compression is still only a little better than that given by the corresponding one-level templates in Figure 9. More marked improvement was obtained with the 18 and 22 pixel templates, although the net effect of the gain is relatively modest because of the lesser starting position.

4.4 JBIG: A standard for bilevel images

JBIG is a recent standard for lossless compression of images (CCITT, 1993). The acronym stands for "Joint Bilevel Image Experts Group." The "Joint ... Group" refers to the fact that it was produced jointly by the CCITT (now called ITU) and ISO international stan-

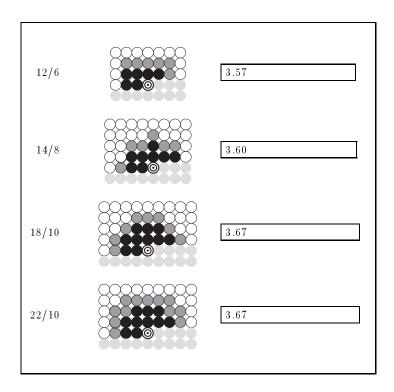


Figure 10: Compression factors for two-level compression

dards bodies. As the "Bilevel Image" suggests, the standard is designed for bilevel images, although it can also be used for grayscale images with a small number of bits per pixel by compressing each bitplane separately.[‡] Once the resolution exceeds about six bits/pixel, however, more effective compression can be obtained using other algorithms.

The compression method can be applied to the full image, or it can be used in a "progressive" mode, giving an approximate version first whose fidelity is improved as time goes by. This is achieved by starting with a low-resolution image and then successively doubling the resolution. The starting image is called the "base" layer and the others are called "differential" layers. The user can choose the base resolution and the number of differential layers that occur before the final layer is reached. It is anticipated that the coarsest base resolution used in practice will be either 12.5 or 25 dpi, which gives an image whose page layout is discernible but whose details are illegible. Setting the base resolution to be the same as the final one, and specifying that there are no differential layers, achieves an ordinary non-progressive or "sequential" transmission.

One drawback with progressive compression is that it makes greater demands on the decoder's memory space than normal sequential transmission. The decoder needs a buffer in which it can store the previous resolution level while it is decoding the next one. In order to reduce the amount of memory required, there is provision in the standard to divide the image into stripes, each one being a horizontal bar with a user-definable height. Stripes are coded and transmitted separately, and the user can specify the order in which stripes, resolutions, and bitplanes are intermixed in the coded data.

The JBIG method is essentially a context-based encoder as described in the previous section, adapted to incorporate progressive transmission. At the lowest level, coding is based on a template model and adaptive arithmetic coding is used to encode the predictions. Because only 1-bit quantities (pixel values) are being encoded, the arithmetic coding process can be streamlined: in fact, a variant of the Q-coder described in Section 2.8 is used.

One interesting feature of the JBIG standard is the method that it suggests for computing lower-resolution images (Yoshida et al., 1992). The obvious way to halve the resolution of an image is to group the pixels into 2×2 blocks and average the four values in each block. Unfortunately, with bilevel pictures it is not clear what to do when two of the pixels are 1's and the other two are 0's. Consistently rounding up or down tends to wash out the image very quickly. Another possibility is to round the value in a random direction each time, but this adds a considerable amount of noise to the image, particularly at the lower resolutions. JBIG incorporates a sophisticated resolution-reduction method that preserves the overall gray values. It also preserves lines, and isolated pixels of the kind that may be produced when a grayscale picture is converted to a black-white halftone one by the standard "dithering" process.

When an image's resolution is reduced by a particular reduction algorithm, it sometimes happens that the receiver can determine a pixel's value unambiguously from pixels that have been received previously. When this occurs, the pixel is said to be "deterministically predictable." An algorithm can be designed that spots such pixels and assigns them their

[‡]Better performance is achieved by encoding pixel values using Gray codes before dividing the image into individual bitplanes. These codes have the property that consecutive numeric values are represented by codes that differ in just one bit position.

value without the need for anything to be transmitted. This improves compression by around 7%.

The JBIG standard defines two kinds of contexts, one for the base or lowest-resolution layer and the other for the differential or higher-resolution layers. Figure 11 shows the templates that can be used for coding the base layer, which as noted at the beginning of this section will often be either 12.5 or 25 dpi in practical applications. There are two options depending on whether the context is to be taken from one or two preceding lines of pixels, and both involve 10 pixels. The encoder can specify which is being used. Not surprisingly, one of them is the 10-bit template of Figure 9.

The single gray pixel in each template of Figure 11 has a special status. Known as an "adaptive" pixel, its position is allowed to change during the course of processing an image. Figure 11 shows its default position, but at any time the encoder can specify that a different pixel is to be used instead. The intention of the adaptive pixel is to improve compression efficiency on images containing certain kinds of halftones, which have a regular grid structure that is usually much larger than the template size. By setting the adaptive pixel to the previous position on the current scan line in the halftone grid, significantly more effective contexts can be obtained for coding.

As well as pixels in the layer being coded, differential-layer templates contain pixels in corresponding positions in the lower-resolution layer. Progressive transmission works in the other direction from resolution reduction because lower resolution layers are known when higher ones are being transmitted, whereas with resolution reduction higher layers are known and lower ones are computed from them. Four templates are used for progressive transmission, depending on the phase of the pixel being coded, as shown in Figure 12. In each case the template includes four pixels of the lower-resolution layer, marked by black circles, and six pixels of the current layer, five of which are marked by black squares and the sixth by a gray square. In Figure 12, as in Figure 11, pixels whose values are not yet known are lightly shaded and have no outlines. There are 10 pixels in these templates, and four phases; thus $2^{12} = 4096$ different contexts are maintained by the encoder and decoder.

The gray pixel in each part of Figure 12 is an "adaptive" pixel. Like the adaptive pixel in the base-layer templates, its position is allowed to change as the image is processed. Figure 12 shows its default position.

According to the earlier description of template models, for each of the 4096 different contexts two associated counts should be stored—one for black target pixels and the other for white ones. However, JBIG does not represent the counts explicitly but buries them inside the arithmetic coding process itself, principally for reasons of efficient implementa-

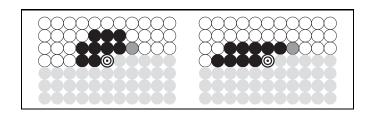


Figure 11: The base-layer templates in the JBIG standard

tion, as described in Section 2.8. Pixel statistics often alter from one region of an image to another, and the design of the JBIG probability estimation mechanism ensures that changing probabilities can be tracked at an appropriate rate. This rate is controlled by the amount of precision in the less probable symbol (LPS) probability, which in turn governs the size of the tables. The lower the precision, the more quickly the estimator responds to a change of probability. Conversely, the greater the precision the better the code performs if the statistics are stationary. There is thus a tradeoff between response in non-stationary situations and coding efficiency in stationary ones.

The advantage of this coding and probability estimation technique is that it is very fast. No multiplication is involved in the arithmetic coding. The estimation is only invoked when renormalization is needed—that is, when a compressed bit is being generated—and even then, it is table driven and requires only an index and memory access. And, despite the approximations involved, little coding efficiency is lost.

4.5 The GIF format for lossless image compression

Probably the most commonly-used lossless image compression format used today is GIF, the Graphics Interchange Format (pronounced *jiff*). This was introduced by CompuServe in 1987 in order to minimize the time required to download pictures over modem links; it was intended as an exchange medium for graphic images that could be displayed on a variety of graphics hardware platforms.§

GIF applies to images in which each pixel is represented by eight bits or less. The code for a pixel is an index into a table that specifies a color map for the entire image. Thus with eight-bit pixel descriptors, there are only 256 possible colors for every pixel in the image. However, the color map may itself contain colors chosen from a far larger palette. The GIF format allows the color table to be specified along with each image, or for a group of images to share the same color map, or for the color map to be omitted entirely. If it is included, it forms an uncompressed prefix to the image file, and may specify up to 256 color table entries each of 24 bits—eight for the three primary colors red, green, and blue. The color table is uncompressed.

The compression scheme used for the sequence of pixel values is not tailored for images

[§]A problem with the GIF format that became apparent only recently is that the LZW compression scheme that it uses is patented by Unisys, and developers offering GIF-based products must pay royalties. A new image format, PNG for Portable Network Graphics, is being designed specifically for the public domain; see http://quest.jpl.nasa.gov/PNG for details. PNG has the further advantage over GIF that it accommodates images with more than eight bits per pixel.

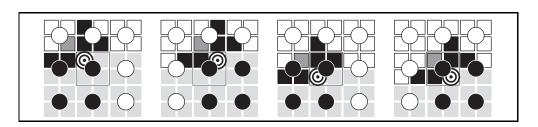


Figure 12: The differential-layer templates in the JBIG standard

at all: it is the LZW scheme that is designed for text compression. Recall from Section 3.2 that LZW is a variant of LZ78, the dictionary-based Ziv-Lempel compression method, that initializes the dictionary to contain the alphabet and then parses off successive phrases from the input string, each phrase being one that is found in the dictionary, at each stage adding to the dictionary the phrase augmented by one additional character.

Suppose we are working with eight-bit pixel descriptors. Then the GIF encoding scheme initializes the dictionary with the 256 possible pixel codes—though this is only notional; it is not necessary to actually load the dictionary with these values—and two extra codes, a "clear" code and an "end-of-information" code, and proceeds to LZW-encode the sequence of pixel values in raster-scan order.

One additional feature is included to make it easy to skip quickly over a particular image in an image file—for one file may contain several images. The LZW-coded information is grouped into blocks of up to 255 bytes, each preceded by a byte count. This means that an image can be skipped over without actually being decompressed.

4.6 The FELICS scheme for compression of grayscale images

FELICS, an acronym for "fast, efficient, lossless image compression system," is a simple and remarkably effective technique for lossless compression of grayscale images that compares well with the lossless variant of JPEG (Howard & Vitter, 1993; Howard, 1993). The idea is to code each pixel based on its two neighbors and use a specially-tailored non-adaptive scheme to represent its value. Tested on an extensive set of Landsat test images, and some general images, the scheme has been reported to provide slightly more compression than the lossless modes of JPEG (arithmetically coded version) and to run considerably faster, at least on general-purpose architectures. However, it does not work well for highly compressible images, and in particular can never compress to less than 1 bit/pixel no matter how redundant the image.

When a picture is transmitted in ordinary raster-scan order, each pixel has two nearest neighbors whose values are known, illustrated in Figure 13a. For interior pixels these are the one above and the one to the left. For pixels on the top row, they are the two preceding ones. And for pixels on the left-hand side of the image, they are the first two pixels of the preceding line (an equally good alternative would be to use the first pixels of the two preceding lines instead). The very first two pixels in the image, which are not handled by these rules, can be encoded using the standard binary representation with negligible impact on performance.

Suppose that pixel P, with neighbors P_1 and P_2 , is being encoded. Overloading the notation slightly, we also use P, P_1 , and P_2 to represent the grayscale values of these pixels. There are three cases: either the value P lies between the other two, or it lies above their maximum H, or it lies below their minimum L. These three situations are illustrated schematically in Figure 13b, in which $L = \min\{P_1, P_2\}$ and $H = \max\{P_1, P_2\}$. Typically P lies in the central region about half the time, and this can be conveniently encoded in 1 bit. A special case occurs when the neighboring pixels P_1 and P_2 have grayscale values that are equal. Then, the probability that P has the same value tends to be somewhat less than half, because only one value is available, and it is advantageous to widen the central region

artificially to include, say, three gray levels rather than just one. When P lies outside the central region, the above/below situations are symmetrical and they can be distinguished by a second bit.

Figure 13b also shows the distribution of P relative to L and H. When P is within the central region its values are distributed almost uniformly, and so a standard binary encoding gives nearly optimal compression in this case. When it is outside the central region its probability drops off sharply as the value moves further away, and the Golomb code described in Section 2.3 is a good choice. We consider each of these two situations a little further.

Define $\Delta = H - L$ to be the size of the central region, as marked in Figure 13b. To encode a pixel P between P_1 and P_2 , the offset P - L is coded within the range $[0, \Delta]$. Ordinary binary coding is used, with the simple modification described at the beginning of Section 2.3 in the case where $\Delta + 1$ is not a power of two.

Now consider the use of Golomb codes to represent values above and below the central region. Recall from Section 2.3 that the Golomb code has a parameter b, and the number x > 0 is coded as q + 1 in unary, where $q = \lfloor (x - 1)/b \rfloor$, followed by r = x - qb - 1 coded in binary, requiring either $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits. Very little coding efficiency is lost by using the special case of a Golomb code where b is restricted to a power of two, $b = 2^m$. Then

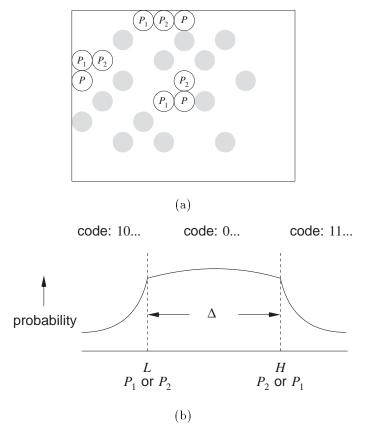


Figure 13: How a pixel's coding region depends on its two neighbors: (a) defining a pixel's neighbors; (b) the three coding regions

the binary part is the lower m bits of x, and the unary part is the number represented by the upper bits.

The Golomb code works best when the value of m is chosen to suit the particular distribution that is being coded. It seems reasonable to employ Δ as a context to determine what value of m to use. In order to optimize m for each context, a record can be kept of how each value performed in the past. For an 8-bit grayscale image there are only 256 possibilities for Δ , and experience shows that it is seldom necessary to use anything other than m = 0, 1, 2, 3 and 3, which correspond to parameters of b = 1, 2, 3, and 8.

It is a simple matter to maintain a table that records, for every value of Δ and m, the cumulative total code length that would have ensued if parameter m were used in the context Δ . The parameter with the smallest cumulative code length is selected to encode the next value encountered in that context. Of course, the decoder maintains the same table so that it can decode the Golomb codes successfully. In this way a suitable parameter value is used for each value of Δ , and very little overhead is needed to maintain the necessary information.

4.7 The CALIC scheme for compression of continuous-tone images

A recent entrant into the field of lossless image compressors is CALIC, for "context-based adaptive lossless image codec" (Wu, 1995; Wu et al., 1995). This has been reported to outperform all other lossless schemes, and was ranked top among nine lossless image compression schemes that were evaluated by ISO/JPEG in July 1995. Although it is capable of efficient implementation, CALIC is conceptually rather an elaborate scheme, and only the main ideas will be sketched here. An important practical feature is that implementations only need sufficient storage space to hold three rows of the image, along with a small amount of memory which is independent of image size.

CALIC encodes an image in conventional raster-scan order, and bases all its computation on a small local neighborhood of twelve pixels that precede the current one in transmission order. Like FELICS, it predicts the value of a pixel based on those pixels in its neighborhood, and then transmits a correction that represents the discrepancy between the actual pixel value and its predicted value. However, both the prediction mechanism and the means of error transmission are considerably more involved than FELICS.

To predict a pixel value, local estimates of the pixel gradient are obtained from the neighboring pixels, in four directions, horizontally, vertically, and in both diagonal directions. These estimates are used to detect the magnitude and orientation of edges in the input image so that adjustments can be made to the prediction accordingly. A total of eight different cases are distinguished, depending on whether sharp or not-so-sharp edges are detected in these directions, and the actual prediction formula is different for each case. Adjusting the prediction to take account of gradient is an explicit attempt to cater for lines and edges in images.

Once this "gradient-adjusted" prediction is obtained, a correction is applied based on the particular pattern that the neighboring pixels exhibit. This is necessary because gradients alone cannot adequately characterize some of the more complex relationships between the predicted pixel and its surroundings. Context modeling can exploit higher-order structures such as texture patterns in the image. This is done by quantizing the neighborhood into several hundred different conditioning classes, and estimating the expected prediction error in each of these classes separately. Conditional expectations are estimated rather than conditional density functions to ensure that sample counts are sufficiently large to obtain good estimates, and to reduce the time and space complexity. The conditional error expectation estimate is then used to correct the gradient-adjusted prediction. The net effect is a non-linear, context-based predictor that corrects itself by adapting to errors made in the same context in the past.

Finally, the difference between the actual pixel value and the corrected prediction is entropy-encoded. In driving the entropy coder, the probability distribution of the prediction error is estimated within (only) eight different conditioning classes. These conditioning classes allow the scheme to adapt to different spatial texture patterns. The appropriate conditioning class is determined by an estimate of the predictability of the signal at that point. This predictability is measured by a linear function of the horizontal and vertical gradient and the prediction error at the preceding pixel. Optimal coefficients can be calculated using off-line linear regression to yield a further improvement in coding efficiency.

If this final correction stage is omitted, a lossy coding scheme results which is competitive with the JPEG lossy compression standard: in fact, it yields significantly higher compression than JPEG for the same objective quality measure.

Another important innovation in CALIC is that it distinguishes between binary and continuous-tone regions of pictures on a local, rather than a global, basis. It does this by examining a neighborhood of six local preceding pixels: if they have no more than two different values (which need not necessarily be black and white), the upcoming pixel is coded in binary mode, otherwise it is coded using the predictive method described above. In binary mode, the six-pixel neighborhood is used to condition the distribution of the upcoming pixel values, as described in Section 4.2. There is, of course, provision to "escape" to a pixel value that is different from the two that occur in the neighborhood.

4.8 Performance of lossless image compression methods

To assess the relative performance of the lossless image compression methods that have been described, tests were conducted on some standard implementations. Image compression algorithms are quite sensitive to a number of factors, and these tests are not designed as a comprehensive comparison but as a rough indication of the performance of the different methods.

The images used for bilevel compression were the eight CCITT test images that were introduced many years ago to assess the performance of the Groups 3 and 4 facsimile standards (Hunter & Robinson, 1980); these contain a total of 33 million 1-bit pixels. They are intended to be representative of office documents. However, the performance of compression algorithms is very sensitive to the type of image being compressed. For example, the static codes of the Group 3 and 4 facsimile methods will *expand* detailed bi-level images, and so those two schemes include an escape flag to indicate that a given scanline is being transmitted uncompressed.

The compression schemes tested were publicly-available implementations of Group 3,

Group 4, JBIG, and bilevel GIF. The JBIG images were compressed using sequential coding, with no differential layer; this is designed to give the best compression. However, the Huffman coding version of JBIG was used because the arithmetic coding module is not available in public-domain implementations.

Figure 14 shows the results. Compression ratio is plotted horizontally; for example, the best compression ratio is around 20:1 for the JBIG method, and GIF provides slightly worse compression than CCITT Group 3 coding. Speed is plotted vertically, in millions of pixels per second on a Sun SPARCCentre 1000. In most cases the encoding and decoding speeds are very similar (within 10 percent of each other); the graph shows the average. Bilevel GIF compression is extremely slow: this is an artifact of the program we used, which assumes that the images are color and naively seeks a good palette to use in the color table. For the

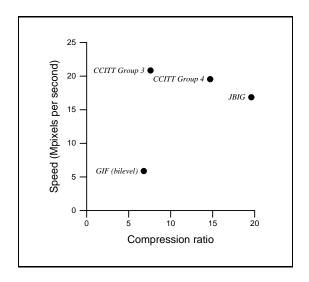


Figure 14: Performance of bilevel compression methods

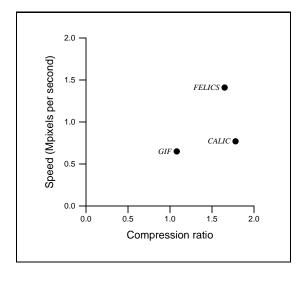


Figure 15: Performance of grayscale compression methods

other three methods, the trade-off between compression and speed is evident. It is likely that while the JBIG implementation is fairly well optimized for speed, the others could be improved considerably by paying more attention to coding details.

Nine images were used to compare the continuous-tone compression schemes, totaling 2.2 million eight-bit pixels. The algorithms tested were standard implementations of GIF, FELICS, and CALIC. Figure 15 shows the results. As expected, CALIC gives slightly better compression than FELICS, though with an appreciable speed penalty. GIF is worse than both the other methods. Some care had been taken with coding each of these implementations, and so the speed comparison more accurately conveys the intrinsic complexity of the methods than in the bilevel case.

5 Other Forms of Data

In this section we briefly describe some other applications of lossless compression methods.

5.1 Sound compression

Most schemes for compressing audio waveforms are lossy, and this is really the most appropriate way to compress audio—even when very high fidelity results are sought. However, lossless audio schemes have been developed, and although the compression they obtain is greatly inferior to lossy schemes tailored for audio, they do get better results than those obtained by applying general-purpose compression programs to audio files.

Generally, simply applying Huffman encoding to the differences between successive samples seems to be as good a method as any. Much better results are achieved with lower-quality (e.g. 8-bit/sample) than with high-quality (e.g. 16-bit/sample) signals; moreover, the higher the sample rate the better the compression ratio.

For example, VOCPACK¶ is a compressor/decompressor for 8-bit digital sound. It can compress signed and unsigned data, sampled at any rate, mono or stereo. Since the method used is not lossy, it is not even necessary to strip file headers before compressing. Although developed for use with SoundBlaster (.VOC) and Windows (.WAV) files, any 8-bit sound file can be compressed since the program takes no assumptions about the file structure. Compression ratios obtained (fraction of original space required by the compressed representation) typically vary from 80% of the original size for files sampled at 11 KHz to 40% for 44 KHz files.

5.2 Index compression

It is becoming increasingly common for large corpuses of text to be available on-line for searching and browsing. Indeed, most computer users quickly create their own personal information space occupying tens or hundreds of megabytes of storage in which they store documents and mail items. In the commercial arena, it is not unusual for such information bases to expand into the multi-gigabyte range.

 $[\]P \, A \, vailable \, \, at \, ftp://oak.oakland.edu/simtel/msdos/sound/vocpak20.zip.$

Compression has clear application to the storage of these repositories, and the techniques described in Section 3 can be adapted for them. Indeed, the use of compression can in some cases save both space and time—it is possible for the cost of decompressing a compressed document to be more than recompensed by the reduced cost of fetching the compressed representation (Zobel & Moffat, 1995).

Compression can also be used to facilitate storing the indexes that are necessary if efficient searching is to be supported. An inverted index for a document collection stores, for each word (and number) that appears anywhere in the collection, a list of identifiers indicating the documents in which that word appears. Stored naively, such an index might occupy a substantial fraction of the space occupied by the collection itself, since on average each word in English text corresponds to 4–6 characters, and a 32-bit integer document identifier also occupies 4 bytes.

There is, however, a great deal of structure in such an inverted index. If it is assumed that each list of document identifiers may be stored in increasing order, then each list can be stored as a sequence of document-gaps rather than as a list of absolute document numbers. For example, if the word cat appears in documents 5, 6, 8, 10, 20, 21, and 31 of some collection, then its inverted list can be represented as 5, 1, 2, 2, 10, 1, 10. Terms that appear in many documents must, ipso facto, correspond to lists containing many small gaps, while rare terms will be stored as a relatively small number of possibly larger gaps. Hence, any prefix code for the integers that favors small values is likely to give a more compact representation than a flat binary code.

This is indeed the case, and even the simple Elias and Golomb codes described in Section 2.3 give remarkably good compression. Word usage in a document collection is hardly random, with words moving in and out of use as time goes by, and more elegant models that attempt to exploit clustering effects yield better compression (Bookstein et al., 1994). For typical document collections storing page-length articles of a few kilobytes in length, it is possible to represent the index information in approximately 5–10% of the space occupied by the text being indexed, a remarkable saving indeed (Moffat & Zobel, 1992; Bell et al., 1993; Witten et al., 1994a).

5.3 Textual images

Most image compression algorithms are general-purpose, applying equally to any kind of image. However, there are many situations, such as facsimile and document archiving applications, in which bilevel images are coded that contain mainly printed text; we call these textual images. It is possible to obtain excellent compression on textual images by taking their nature into account. One option is to perform optical character recognition on the text and transmit the ASCII codes for the characters, along with some information about their position on the page. The problem with this is that recognition is not completely reliable, and mistakes can be catastrophic. An alternative is to identify repeated patterns, usually characters, in the image and replace them by pointers into a library of patterns (Witten et al., 1994b). In general, the process comprises these steps.

1. Find, isolate and extract all the *marks* in the image, which are connected groups of black pixels.

- 2. Construct a library containing the different marks that appear.
- 3. Identify the symbol in the library corresponding to each mark in the image, and measure the coordinate offsets between one mark and the next.
- 4. Compress and transmit the library, the symbol sequence, and the offsets.

From the information in the fourth step, an approximation to the original image called the reconstructed text can be created. This gives a lossy method of compression. To make the reconstruction lossless, which may be necessary for legal, medical, or historical purposes, a fifth step may be added:

5. Transmit enough additional information to restore the original image from the reconstructed text.

Combining the two modes gives a two-stage procedure for progressive transmission which is particularly attractive in practical applications involving browsing through large databases. In lossy mode this scheme outperforms other lossy schemes of image compression, and yet retains a very sharp, high-quality, rendition of the image. In lossless modes it outperforms other lossless schemes such as JBIG.

5.4 Filesystem compression

On-the-fly disk compression systems intercept data that is being written to disk and compress it before it is stored. From the user's point of view, the data takes a little longer to read and write, but the disk stores considerably more data than before. Clearly it is essential that such systems are lossless. They also need to be very fast so that they do not seriously impact system performance—in fact, in some cases performance actually improves because less data is being transferred to and from disk.

LZ77-based methods are ideal for this application because they decode very quickly, and can be made to encode very quickly by using an efficient searching data structure, such as a hash table, and limiting how much searching is done. Limiting the search means that some good matches may be overlooked, and compression will deteriorate, but this is a reasonable compromise to make the system as transparent as possible. Improved compression can be achieved by recompressing files when the processor is idle, using a more thorough search for matches. The decompression algorithm is identical, regardless of how the matches were found, so recompression is an ideal "background" process.

A compressed file system is particularly vulnerable to errors, and so reliability is a major concern. Data is normally stored on disk in fixed size blocks, but compression makes the size of a block unpredictable, which complicates file management. Furthermore, a small error in a compressed file can render it unreadable. Thus, minor problems that might not seriously affect an uncompressed system can have a major impact on a compressed one. Consequently, the most successful filesystem compressors have been the ones that are resiliant to errors, rather than those that give the best compression—avoiding having a corrupted disk is much more important to users than squeezing an extra few percent of data onto it.

6 Summary

Lossless compression is appropriate for many types of data. In particular, text and some forms of images must be stored exactly, and approximations cannot be tolerated. In this chapter we have surveyed a wide spectrum of lossless compression techniques, including a variety of coding methods, and compression models suited for text, bi-level, gray-scale, and document images, and database indexes.

Software

Many of the techniques described in this chapter have been implemented by the current authors, in collaboration with J. Zobel (RMIT, Australia). A document and image compression and retrieval system—named MG—has been made publicly available, and may be copied from ftp://munnari.oz.au/pub/mg. Witten et al. (1994a) describe the various components of this system, and give a tutorial introduction to the facilities it offers.

Acknowledgements

This work was supported by the Australian Research Council and the Natural Sciences and Engineering Research Council of Canada. We are grateful to Stuart Inglis who performed the tests described in Section 4.8.

References

- Bell, T.C. (1986a). Better OPM/L text compression. *IEEE Transactions on Communications*, COM-34:1176-1182.
- Bell, T.C. (1986b). A Unifying Theory and Improvements for Existing Approaches to Text Compression. Ph.D. thesis, University of Canterbury, Christchurch, New Zealand.
- Bell, T.C., & Moffat, A. (1989). A note on the DMC data compression scheme. The Computer Journal, 32(1):16-20.
- Bell, T.C., & Witten, I.H. (1994). The relationship between greedy parsing and symbolwise text compression. *Journal of the ACM*, 41(4):708-724.
- Bell, T.C., Moffat, A., Nevill-Manning, C.G., Witten, I.H., & Zobel, J. (1993). Data compression in full-text retrieval systems. Journal of the American Society for Information Science, 44(9):508-531.
- Bodson, D., Urban, S.J., Deutermann, A.R., & Clarke, C.E. (1985). Measurement of data compression in advanced group 4 facsimile systems. *Proc. IEEE*, 73(4):731–739.
- Bookstein, A., & Klein, S.T. (1993). Is Huffman coding dead? Computing, 50(4):279-296.

- Bookstein, A., Klein, S.T., & Raita, T. (1994). Markov models for clusters in concordance compression. In Storer, J.A., & Cohn, M. (eds), *Proc. IEEE Data Compression Conference*. Snowbird, Utah: IEEE Computer Society Press, Los Alamitos, California, pages 116–125.
- Bunton, S. (1995). The structure of DMC. In (Storer & Cohn, 1995), pages 72-81.
- Burrows, M., & Wheeler, D.J. (1994). A block-sorting lossless data compression algorithm.

 Tech. rept. 124. Digital Equipment Corporation, Palo Alto, California.
- Capocelli, R.M., & De Santos, A. (1991). New bounds on the redundancy of Huffman codes. IEEE Transactions on Information Theory, IT-37:1095-1104.
- Capocelli, R.M., Giancarlo, R., & Taneja, I.J. (1986). Bounds on the redundancy of Huffman codes. *IEEE Transactions on Information Theory*, IT-32:854-857.
- CCITT (1993). Draft Recommendation T.82 & ISO DIS 11544: Coded Representation of Picture and Audio information—Progressive Bi-Level Image Compression.
- Chevion, D., Karnin, E.D., & Walach, E. (1991). High efficiency, multiplication free approximation of arithmetic coding. In Storer, J.A., & Reif, J.H. (eds), *Proc. IEEE Data Compression Conference*. Snowbird, Utah: IEEE Computer Society Press, Los Alamitos, California, pages 43–52.
- Cleary, J.G., & Witten, I.H. (1984a). A comparison of enumerative and adaptive codes. *IEEE Transactions on Information Theory*, IT-30(2):306-315.
- Cleary, J.G., & Witten, I.H. (1984b). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32:396-402.
- Cleary, J.G., Teahan, W., & Witten, I.H. (1995). Unbounded length contexts for PPM. In (Storer & Cohn, 1995), pages 52-61.
- Connell, J.B. (1973). A Huffman-Shannon-Fano code. Proc. IEEE, 61(7):1046-1047.
- Cormack, G.V., & Horspool, R.N. (1984). Algorithms for adaptive Huffman codes. Information Processing Letters, 18:159-165.
- Cormack, G.V., & Horspool, R.N. (1987). Data compression using dynamic Markov modeling. The Computer Journal, 30(6):541-550.
- Cover, T.M., & King, R.C. (1978). A convergent gambling estimate of the entropy of English. *IEEE Transactions on Information Theory*, IT-24:413-421.
- Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194-203.
- Fenwick, P.M. (1994). A new data structure for cumulative probability tables. Software— Practice and Experience, 24(3):327-336. Errata published in 24(7):677, July 1994.

- Feygin, G., Gulak, P.G., & Chow, P. (1994). Minimizing excess code length and VLSI complexity in the multiplication free approximation of arithmetic coding. *Information Processing & Management*, 30(6):805-816.
- Fiala, E.R., & Green, D.H. (1989). Data compression with finite windows. Communications of the ACM, 32(4):490-505.
- Gallager, R.G. (1978). Variations on a theme by Huffman. IEEE Transactions on Information Theory, IT-24(6):668-674.
- Gallager, R.G., & Van Voorhis, D.C. (1975). Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory*, IT-21(2):228-230.
- Golomb, S.W. (1966). Run-length encodings. *IEEE Transactions on Information Theory*, IT-12(3):399-401.
- Gutmann, P.C., & Bell, T.C. (1994). A hybrid approach to text compression. In Storer, J.A., & Cohn, M. (eds), *Proc. IEEE Data Compression Conference*. Snowbird, Utah: IEEE Computer Society Press, Los Alamitos, California, pages 225–234.
- Hirschberg, D., & Lelewer, D. (1990). Efficient decoding of prefix codes. Communications of the ACM, 33(4):449-459.
- Hoang, D.T., Long, P.M., & Vitter, J.S. (1995). Multiple-dictionary compression using partial matching. In (Storer & Cohn, 1995), pages 272-281.
- Horspool, R.N. (1995). The effect of non-greedy parsing in Ziv-Lempel compression methods. In (Storer & Cohn, 1995), pages 302-311.
- Howard, P.G. (1993). The Design and Analysis of Efficient Lossless Data Compression Systems. Ph.D. thesis, Brown University, Rhode Island. Available as Technical Report CS-93-28.
- Howard, P.G., & Vitter, J.S. (1992). Practical implementations of arithmetic coding. In Storer, J.A. (ed), *Image and Text Compression*. Norwell, Massachusetts: Kluwer Academic.
- Howard, P.G., & Vitter, J.S. (1993). Fast and efficient lossless image compression. In Storer, J.A., & Cohn, M. (eds), Proc. IEEE Data Compression Conference. Snowbird, Utah: IEEE Computer Society Press, Los Alamitos, California, pages 351-360.
- Hu, T.C., & Tucker, A.C. (1971). Optimal computer search trees and variable length alphabetic codes. SIAM Journal of Applied Mathematics, 21:514-532.
- Huffman, D.A. (1952). A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Engineers*, 40(9):1098-1101.
- Hunter, R., & Robinson, A.H. (1980). International digital facsimile coding standards. Proc. IEEE, 68(7):854-867.

- Jones, D.W. (1988). Application of splay trees to data compression. Communications of the ACM, 31(8):996-1007.
- Katajainen, J., Moffat, A., & Turpin, A. (1995). A fast and space-economical algorithm for length-limited coding. In *Proc. International Symposium on Algorithms and Computation*. Cairns, Australia: Springer-Verlag. To appear.
- Knuth, D.E. (1985). Dynamic Huffman coding. Journal of Algorithms, 6:163-180.
- Langdon, G.G. (1983). A note on the Ziv-Lempel model for compressing individual sequences. IEEE Transactions on Information Theory, IT-29(2):284-287.
- Langdon, G.G. (1984). An introduction to arithmetic coding. IBM Journal of Research and Development, 28(2):135-149.
- Langdon, G.G., & Rissanen, J. (1982). A simple general binary source code. *IEEE Transactions on Information Theory*, IT-28(5):800-803.
- Larmore, L.L., & Hirschberg, D.S. (1990). A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464-473.
- Larmore, L.L., & Przytycka, T.M. (1994). A fast algorithm for optimum height-limited alphabetic binary trees. SIAM Journal on Computing, 23(6):1283-1312.
- Larmore, L.L., & Przytycka, T.M. (In press). Constructing Huffman trees in parallel. SIAM Journal on Computing. To appear.
- Lu, W.W., & Gough, M.P. (1993). A fast-adaptive Huffman coding algorithm. IEEE Transactions on Information Theory, 41(4):535-538.
- Manstetten, D. (1992). Tight upper bounds on the redundancy of Huffman codes. *IEEE Transactions on Information Theory*, 38(1):144-151.
- Moffat, A. (1990a). Implementing the PPM data compression scheme. *IEEE Transactions* on Communications, 38(11):1917-1921.
- Moffat, A. (1990b). Linear time adaptive arithmetic coding. *IEEE Transactions on Information Theory*, 36(2):401-406.
- Moffat, A. (1991). Two-level context-based compression of binary images. In Storer, J.A., & Reif, J.H. (eds), *Proc. IEEE Data Compression Conference*. Snowbird, Utah: IEEE Computer Society Press, Los Alamitos, California, pages 382–391.
- Moffat, A., & Katajainen, J. (1995). In-place calculation of minimum-redundancy codes. In *Proc. Workshop on Algorithms and Data Structures*. Kingston University, Canada: Springer-Verlag.
- Moffat, A., & Turpin, A. (1995). On the implementation of minimum-redundancy prefix codes. Submitted.

- Moffat, A., & Zobel, J. (1992). Parameterised compression for sparse bitmaps. In Belkin, N., Ingwersen, P., & Pejtersen, A.M. (eds), Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval. Copenhagen: ACM Press, New York, pages 274-285.
- Moffat, A., Sharman, N., Witten, I.H., & Bell, T.C. (1994). An empirical evaluation of coding methods for multi-symbol alphabets. *Information Processing & Management*, 30(6):791-804.
- Moffat, A., Neal, R., & Witten, I.H. (1995a). Arithmetic coding revisited. In (Storer & Cohn, 1995), pages 202-211.
- Moffat, A., Turpin, A., & Katajainen, J. (1995b). Space-efficient construction of optimal prefix codes. In (Storer & Cohn, 1995), pages 192–201.
- Pennebaker, W.B., & Mitchell, J.L. (1988). Probability estimation for the Q-coder. *IBM Journal of Research and Development*, 32(6):737-752.
- Pennebaker, W.B., & Mitchell, J.L. (1993). JPEG: Still Image Data Compression Standard. New York: Van Nostrand Reinhold.
- Pennebaker, W.B., Mitchell, J.L., Langdon, G.G., & Arps, R.B. (1988). An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, 32(6):717-726.
- Pratt, W.K., Capitant, P.J., Chen, W.H., Hamilton, E.R., & Wallis, R.H (1980). Combining symbol matching facsimile data compression system. *Proc. IEEE*, 68(7):786-796.
- Rice, R.F. (1979). Some practical universal noiseless coding techniques. Tech. rept. 79-22. Jet Propulsion Laboratory, Pasadena, California.
- Rissanen, J., & Langdon, G.G. (1979). Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149-162.
- Rissanen, J., & Langdon, G.G. (1981). Universal modeling and coding. *IEEE Transactions on Information Theory*, IT-27(1):12-23.
- Rissanen, J., & Mohiuddin, K.M. (1989). A multiplication-free multialphabet arithmetic code. *IEEE Transactions on Communications*, 37(2):93–98.
- Rissanen, J.J. (1976). Generalised Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198-203.
- Schieber, B. (1995). Computing a minimum-weight k-link path in graphs with the concave Monge property. In *Proc. 6th Ann. Symp. Discrete Algorithms*. San Francisco, California: SIAM, Philadelphia, Pennsylvania, pages 405-411.
- Schwartz, E.S., & Kallick, B. (1964). Generating a canonical prefix encoding. Communications of the ACM, 7(3):166-169.

- Shannon, C.E. (1948). A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379-423, 623-656.
- Shannon, C.E. (1951). Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:55.
- Storer, J.A., & Cohn, M. (eds) (1995). Proc. IEEE Data Compression Conference. Snow-bird, Utah: IEEE Computer Society Press, Los Alamitos, California.
- Storer, J.A., & Szymanski, T.G. (1982). Data compression via textual substitution. Journal of the ACM, 29:928-951.
- Teuhola, J., & Raita, T. (1993). Application of a finite-state model to text compression.

 The Computer Journal, 36(7):607-614.
- Tischer, P., Worley, R., Maeder, A.J., & Goodwin, M. (1993). Context-based lossless image compression. *The Computer Journal*, 36(1):68-77.
- van Leeuwen, J. (1976). On the construction of Huffman trees. In *Proc. 3rd International Colloquium on Automata, Languages, and Programming*. Edinburgh University, Scotland: Edinburgh University, pages 382-410.
- Vitter, J.S. (1989). Algorithm 673: Dynamic Huffman coding. ACM Transactions on Mathematical Software, 15(2):158-167.
- Welch, T.A. (1984). A technique for high performance data compression. *IEEE Computer*, 17:8–20.
- Williams, R.N. (1991). An extremely fast Ziv-Lempel data compression algorithm. In Storer, J.A., & Reif, J.H. (eds), *Proc. IEEE Data Compression Conference*. Snowbird, Utah: IEEE Computer Society Press, Los Alamitos, California, pages 362–371.
- Witten, I.H., & Bell, T.C. (1991). The zero frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085-1094.
- Witten, I.H., Neal, R., & Cleary, J.G. (1987). Arithmetic coding for data compression.

 Communications of the ACM, 30(6):520-541.
- Witten, I.H., Moffat, A., & Bell, T.C. (1994a). Managing Gigabytes: Compressing and Indexing Documents and Images. New York: Van Nostrand Reinhold.
- Witten, I.H., Bell, T.C., Emberson, H., Inglis, S., & Moffat, A. (1994b). Textual image compression: Two-stage lossy/lossless encoding of textual images. *Proc. IEEE*, 82(6):878-888.
- Wu, X. (1995). Context selection and quantization for lossless image coding. In (Storer & Cohn, 1995), page 453.
- Wu, X., Memon, N., & Sayood, K. (1995). A context-based, adaptive, lossless/nearly-lossless coding scheme for continuous-tone images. Proposal submitted to ISO/IEC JTC SC 29/WG 1 for evaluation for a new international standard for lossless image compression.

- Yasuda, Y., Yamazake, Y., Kamae, T., & Kobayashi, K. (1985). Advances in FAX. *Proc. IEEE*, 73(4):706–730.
- Yoshida, T., Endoh, T., Kawamura, N., & Kato, H. (1992). Image reduction system. US Patent 5,159,468.
- Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337-343.
- Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530-536.
- Zobel, J., & Moffat, A. (1995). Adding compression to a full-text retrieval system. Software—Practice and Experience, 25(8):891-903.