6

Context-Based Compression

6.1 Overview

In this chapter we present a number of techniques that use minimal prior assumptions about the statistics of the data. Instead they use the context of the data being encoded and the past history of the data to provide more efficient compression. We will look at a number of schemes that are principally used for the compression of text. These schemes use the context in which the data occurs in different ways.

6.2 Introduction

In Chapters 3 and 4 we learned that we get more compression when the message that is being coded has a more skewed set of probabilities. By "skewed" we mean that certain symbols occur with much higher probability than others in the sequence to be encoded. So it makes sense to look for ways to represent the message that would result in greater skew. One very effective way to do so is to look at the probability of occurrence of a letter in the context in which it occurs. That is, we do not look at each symbol in a sequence as if it had just happened out of the blue. Instead, we examine the history of the sequence before determining the likely probabilities of different values that the symbol can take.

In the case of English text, Shannon [8] showed the role of context in two very interesting experiments. In the first, a portion of text was selected and a subject (possibly his wife, Mary Shannon) was asked to guess each letter. If she guessed correctly, she was told that she was correct and moved on to the next letter. If she guessed incorrectly, she was told the correct answer and again moved on to the next letter. Here is a result from one of these experiments. Here the dashes represent the letters that were correctly guessed.

Actual Text	THE ROOM WAS NOT VERY LIGHT A SMALL OBLONG
Subject Performance	____ROO______NOT_V_____I______SM____OBL___

Notice that there is a good chance that the subject will guess the letter, especially if the letter is at the end of a word or if the word is clear from the context. If we now represent the original sequence by the subject performance, we would get a very different set of probabilities for the values that each element of the sequence takes on. The probabilities are definitely much more skewed in the second row: the "letter" _ occurs with high probability. If a mathematical twin of the subject were available at the other end, we could send the "reduced" sentence in the second row and have the twin go through the same guessing process to come up with the original sequence.

In the second experiment, the subject was allowed to continue guessing until she had guessed the correct letter and the number of guesses required to correctly predict the letter was noted. Again, most of the time the subject guessed correctly, resulting in 1 being the most probable number. The existence of a mathematical twin at the receiving end would allow this skewed sequence to represent the original sequence to the receiver. Shannon used his experiments to come up with upper and lower bounds for the English alphabet (1.3 bits per letter and 0.6 bits per letter, respectively).

The difficulty with using these experiments is that the human subject was much better at predicting the next letter in a sequence than any mathematical predictor we can develop. Grammar is hypothesized to be innate to humans [64], in which case development of a predictor as efficient as a human for language is not possible in the near future. However, the experiments do provide an approach to compression that is useful for compression of all types of sequences, not simply language representations.

If a sequence of symbols being encoded does not consist of independent occurrences of the symbols, then the knowledge of which symbols have occurred in the neighborhood of the symbol being encoded will give us a much better idea of the value of the symbol being encoded. If we know the context in which a symbol occurs we can guess with a much greater likelihood of success what the value of the symbol is. This is just another way of saying that, given the context, some symbols will occur with much higher probability than others. That is, the probability distribution given the context is more skewed. If the context is known to both encoder and decoder, we can use this skewed distribution to perform the encoding, thus increasing the level of compression. The decoder can use its knowledge of the context to determine the distribution to be used for decoding. If we can somehow group like contexts together, it is quite likely that the symbols following these contexts will be the same, allowing for the use of some very simple and efficient compression strategies. We can see that the context can play an important role in enhancing compression, and in this chapter we will look at several different ways of using the context.

Consider the encoding of the word *probability*. Suppose we have already encoded the first four letters, and we want to code the fifth letter, a. If we ignore the first four letters, the probability of the letter a is about 0.06. If we use the information that the previous letter is b, this reduces the probability of several letters such as q and z occurring and boosts the probability of an a occurring. In this example, b would be the first-order context for a, ob would be the second-order context for a, and so on. Using more letters to define the context in which a occurs, or higher-order contexts, will generally increase the probability of the occurrence of *a* in this example, and hence reduce the number of bits required to encode its occurrence. Therefore, what we would like to do is to encode each letter using the probability of its occurrence with respect to a context of high order.

If we want to have probabilities with respect to all possible high-order contexts, this might be an overwhelming amount of information. Consider an alphabet of size M. The number of first-order contexts is M, the number of second-order contexts is M^2, and so on. Therefore, if we wanted to encode a sequence from an alphabet of size 256 using contexts of order 5, we would need 256^5, or about 1.09951×10^{12} probability distributions! This is not a practical alternative. A set of algorithms that resolve this problem in a very simple and elegant way is based on the *prediction with partial match (ppm)* approach. We will describe this in the next section.

6.3 Prediction with Partial Match (*ppm*)

The best-known context-based algorithm is the *ppm* algorithm, first proposed by Cleary and Witten [65] in 1984. It has not been as popular as the various Ziv-Lempel-based algorithms mainly because of the faster execution speeds of the latter algorithms. Lately, with the development of more efficient variants, *ppm*-based algorithms are becoming increasingly more popular.

The idea of the *ppm* algorithm is elegantly simple. We would like to use large contexts to determine the probability of the symbol being encoded. However, the use of large contexts would require us to estimate and store an extremely large number of conditional probabilities, which might not be feasible. Instead of estimating these probabilities ahead of time, we can reduce the burden by estimating the probabilities as the coding proceeds. This way we only need to store those contexts that have occurred in the sequence being encoded. This is a much smaller number than the number of all possible contexts. While this mitigates the problem of storage, it also means that, especially at the beginning of an encoding, we will need to code letters that have not occurred previously in this context. In order to handle this situation, the source coder alphabet always contains an escape symbol, which is used to signal that the letter to be encoded has not been seen in this context.

6.3.1 The Basic Algorithm

The basic algorithm initially attempts to use the largest context. The size of the largest context is predetermined. If the symbol to be encoded has not previously been encountered in this context, an escape symbol is encoded and the algorithm attempts to use the next smaller context. If the symbol has not occurred in this context either, the size of the context is further reduced. This process continues until either we obtain a context that has previously been encountered with this symbol, or we arrive at the conclusion that the symbol has not been encountered previously in *any* context. In this case, we use a probability of $1/M$ to encode the symbol, where M is the size of the source alphabet. For example, when coding the *a* of *probability*, we would first attempt to see if the string *proba* has previously occurred—that is, if *a* had previously occurred in the context of *prob*. If not, we would encode an escape and see if *a* had occurred in the context of *rob*. If the string *roba* had not occurred previously, we would again send an escape symbol and try the context *ob*. Continuing in this manner, we would try the context *b*, and failing that, we would see if the letter *a* (with a zero-order context) had occurred previously. If *a* was being encountered for the first time, we would use a model in which all letters occur with equal probability to encode *a*. This equiprobable model is sometimes referred to as the context of order -1.

As the development of the probabilities with respect to each context is an adaptive process, each time a symbol is encountered, the count corresponding to that symbol is updated. The number of counts to be assigned to the escape symbol is not obvious, and a number of different approaches have been used. One approach used by Cleary and Witten is to give the escape symbol a count of one, thus inflating the total count by one. Cleary and Witten call this method of assigning counts Method A, and the resulting algorithm *ppma*. We will describe some of the other ways of assigning counts to the escape symbol later in this section.

Before we delve into some of the details, let's work through an example to see how all this works together. As we will be using arithmetic coding to encode the symbols, you might wish to refresh your memory of the arithmetic coding algorithms.

Example 6.3.1:

Let's encode the sequence

thisƀisƀtheƀtithe

Assuming we have already encoded the initial seven characters *thisƀis*, the various counts and *Cum_Count* arrays to be used in the arithmetic coding of the symbols are shown in Tables 6.1–6.4. In this example, we are assuming that the longest context length is two. This is a rather small value and is used here to keep the size of the example reasonably small. A more common value for the longest context length is five.

We will assume that the word length for arithmetic coding is six. Thus, $l = 000000$ and $u = 111111$. As *thisƀis* has already been encoded, the next letter to be encoded is *ƀ*. The second-order context for this letter is *is*. Looking at Table 6.4, we can see that the letter *ƀ*

TABLE 6.1 Count array for -1 order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	1	3
s	1	4
e	1	5
ƀ	1	6
Total Count		6

TABLE 6.2 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
b̸	1	7
$\langle Esc \rangle$	1	8
Total Count		8

TABLE 6.3 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
h	*i*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
i	*s*	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3
b̸	*i*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
s	*b̸*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2

is the first letter in this context with a *Cum_Count* value of 1. As the *Total_Count* in this case is 2, the update equations for the lower and upper limits are

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{0}{2} \right\rfloor = 0 = 000000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor - 1 = 31 = 011111.$$

TABLE 6.4 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
hi	*s*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
is	*b̸*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
sb̸	*i*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
b̸i	*s*	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2

As the MSBs of both l and u are the same, we shift that bit out, shift a 0 into the LSB of l, and a 1 into the LSB of u. The transmitted sequence, lower limit, and upper limit after the update are

$$\begin{aligned} \text{Transmitted sequence}: &\ 0 \\ l: &\ 000000 \\ u: &\ 111111 \end{aligned}$$

We also update the counts in Tables 6.2–6.4.

The next letter to be encoded in the sequence is t. The second-order context is *sb̸*. Looking at Table 6.4, we can see that t has not appeared before in this context. We therefore encode an escape symbol. Using the counts listed in Table 6.4, we update the lower and upper limits:

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

Again, the MSBs of l and u are the same, so we shift the bit out and shift 0 into the LSB of l, and 1 into u, restoring l to a value of 0 and u to a value of 63. The transmitted sequence is now 01. After transmitting the escape, we look at the first-order context of t, which is $\not{b}$. Looking at Table 6.3, we can see that t has not previously occurred in this context. To let the decoder know this, we transmit another escape. Updating the limits, we get

$$l = 0 + \left\lfloor (63-0+1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63-0+1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

As the MSBs of l and u are the same, we shift the MSB out and shift 0 into the LSB of l and 1 into the LSB of u. The transmitted sequence is now 011. Having escaped out of the first-order contexts, we examine Table 6.5, the updated version of Table 6.2, to see if we can encode t using a zero-order context. Indeed we can, and using the *Cum_Count* array, we can update l and u:

$$l = 0 + \left\lfloor (63-0+1) \times \frac{0}{9} \right\rfloor = 0 = 000000$$

$$u = 0 + \left\lfloor (63-0+1) \times \frac{1}{9} \right\rfloor - 1 = 6 = 000110.$$

TABLE 6.5 Updated count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
$\not{b}$	2	8
$\langle Esc \rangle$	1	9
Total Count		9

The three most significant bits of both l and u are the same, so we shift them out. After the update we get

Transmitted sequence : 011000

l : 000000

u : 110111

The next letter to be encoded is h. The second-order context ƀt has not occurred previously, so we move directly to the first-order context t. The letter h has occurred previously in this context, so we update l and u and obtain

$$\begin{aligned} \text{Transmitted sequence}: &\ 0110000 \\ l: &\ 000000 \\ u: &\ 110101 \end{aligned}$$

TABLE 6.6 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	2	2
h	2	4
i	2	6
s	2	8
ƀ	2	10
$\langle Esc \rangle$	1	11
Total Count		11

TABLE 6.7 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	h	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3
h	i	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
i	s	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3
ƀ	i	1	1
	t	1	2
	$\langle Esc \rangle$	1	3
	Total Count		3
s	ƀ	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3

TABLE 6.8 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
hi	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
is	*b̸*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
sb̸	*i*	1	1
	t	1	2
	⟨*Esc*⟩	1	3
	Total Count		3
b̸i	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
b̸t	*h*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

The method of encoding should now be clear. At this point the various counts are as shown in Tables 6.6–6.8. ♦

Now that we have an idea of how the *ppm* algorithm works, let's examine some of the variations.

6.3.2 The Escape Symbol

In our example we used a count of one for the escape symbol, thus inflating the total count in each context by one. Cleary and Witten call this Method A, and the corresponding algorithm is referred to as *ppma*. There is really no obvious justification for assigning a count of one to the escape symbol. For that matter, there is no obvious method of assigning counts to the escape symbol. There have been various methods reported in the literature.

Another method described by Cleary and Witten is to reduce the counts of each symbol by one and assign these counts to the escape symbol. For example, suppose in a given

TABLE 6.9 Counts using Method A.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	$\langle Esc \rangle$	1
Total Count		23

TABLE 6.10 Counts using Method B.

Context	Symbol	Count
prob	*a*	9
	l	8
	o	2
	$\langle Esc \rangle$	3
Total Count		22

sequence *a* occurs 10 times in the context of *prob*, *l* occurs 9 times, and *o* occurs 3 times in the same context (e.g., *problem, proboscis*, etc.). In Method A we assign a count of one to the escape symbol, resulting in a total count of 23, which is one more than the number of times *prob* has occurred. The situation is shown in Table 6.9.

In this second method, known as Method B, we reduce the count of each of the symbols *a*, *l*, and *o* by one and give the escape symbol a count of three, resulting in the counts shown in Table 6.10.

The reasoning behind this approach is that if in a particular context more symbols can occur, there is a greater likelihood that there is a symbol in this context that has not occurred before. This increases the likelihood that the escape symbol will be used. Therefore, we should assign a higher probability to the escape symbol.

A variant of Method B, appropriately named Method C, was proposed by Moffat [66]. In Method C, the count assigned to the escape symbol is the number of symbols that have occurred in that context. In this respect, Method C is similar to Method B. The difference comes in the fact that, instead of "robbing" this from the counts of individual symbols, the total count is inflated by this amount. This situation is shown in Table 6.11.

While there is some variation in the performance depending on the characteristics of the data being encoded, of the three methods for assigning counts to the escape symbol, on the average, Method C seems to provide the best performance.

6.3.3 Length of Context

It would seem that as far as the maximum length of the contexts is concerned, more is better. However, this is not necessarily true. A longer maximum length will usually result

TABLE 6.11 Counts using Method C.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	$\langle Esc \rangle$	3
Total Count		25

in a higher probability if the symbol to be encoded has a nonzero count with respect to that context. However, a long maximum length also means a higher probability of long sequences of escapes, which in turn can increase the number of bits used to encode the sequence. If we plot the compression performance versus maximum context length, we see an initial sharp increase in performance until some value of the maximum length, followed by a steady drop as the maximum length is further increased. The value at which we see a downturn in performance changes depending on the characteristics of the source sequence.

An alternative to the policy of a fixed maximum length is used in the algorithm *ppm** [67]. This algorithm uses the fact that long contexts that give only a single prediction are seldom followed by a new symbol. If *mike* has always been followed by *y* in the past, it will probably not be followed by *ƀ* the next time it is encountered. Contexts that are always followed by the same symbol are called *deterministic* contexts. The *ppm** algorithm first looks for the longest deterministic context. If the symbol to be encoded does not occur in that context, an escape symbol is encoded and the algorithm defaults to the maximum context length. This approach seems to provide a small but significant amount of improvement over the basic algorithm. Currently, the best variant of the *ppm** algorithm is the *ppmz* algorithm by Charles Bloom. Details of the *ppmz* algorithm as well as implementations of the algorithm can be found at *http://www.cbloom.com/src/ppmz.html.*

6.3.4 The Exclusion Principle

The basic idea behind arithmetic coding is the division of the unit interval into subintervals, each of which represents a particular letter. The smaller the subinterval, the more bits are required to distinguish it from other subintervals. If we can reduce the number of symbols to be represented, the number of subintervals goes down as well. This in turn means that the sizes of the subintervals increase, leading to a reduction in the number of bits required for encoding. The exclusion principle used in *ppm* provides this kind of reduction in rate. Suppose we have been compressing a text sequence and come upon the sequence *proba*, and suppose we are trying to encode the letter *a*. Suppose also that the state of the two-letter context *ob* and the one-letter context *b* are as shown in Table 6.12.

First we attempt to encode *a* with the two-letter context. As *a* does not occur in this context, we issue an escape symbol and reduce the size of the context. Looking at the table for the one-letter context *b*, we see that *a* does occur in this context with a count of 4 out of a total possible count of 21. Notice that other letters in this context include *l* and *o*. However,

TABLE 6.12 Counts for exclusion example.

Context	Symbol	Count
ob	*l*	10
	o	3
	⟨*Esc*⟩	2
	Total Count	15
b	*l*	5
	o	3
	a	4
	r	2
	e	2
	⟨*Esc*⟩	5
	Total Count	21

TABLE 6.13 Modified table used for exclusion example.

Context	Symbol	Count
b	*a*	4
	r	2
	e	2
	⟨*Esc*⟩	3
	Total Count	11

by sending the escape symbol in the context of *ob*, we have already signalled to the decoder that the symbol being encoded is not any of the letters that have previously been encountered in the context of *ob*. Therefore, we can increase the size of the subinterval corresponding to *a* by temporarily removing *l* and *o* from the table. Instead of using Table 6.12, we use Table 6.13 to encode *a*. This exclusion of symbols from contexts on a temporary basis can result in cumulatively significant savings in terms of rate.

You may have noticed that we keep talking about small but significant savings. In lossless compression schemes, there is usually a basic principle, such as the idea of prediction with partial match, followed by a host of relatively small modifications. The importance of these modifications should not be underestimated because often together they provide the margin of compression that makes a particular scheme competitive.

6.4 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) algorithm also uses the context of the symbol being encoded, but in a very different way, for lossless compression. The transform that is a major part of this algorithm was developed by Wheeler in 1983. However, the BWT compression algorithm, which uses this transform, saw the light of day in 1994 [68]. Unlike most of the previous algorithms we have looked at, the BWT algorithm requires that the entire sequence to be coded be available to the encoder before the coding takes place. Also, unlike most of the previous algorithms, the decoding procedure is not immediately evident once we know the encoding procedure. We will first describe the encoding procedure. If it is not clear how this particular encoding can be reversed, bear with us and we will get to it.

The algorithm can be summarized as follows. Given a sequence of length N, we create $N-1$ other sequences where each of these $N-1$ sequences is a cyclic shift of the original sequence. These N sequences are arranged in lexicographic order. The encoder then transmits the sequence of length N created by taking the last letter of each sorted, cyclically shifted, sequence. This sequence of last letters L, and the position of the original sequence in the sorted list, are coded and sent to the decoder. As we shall see, this information is sufficient to recover the original sequence.

We start with a sequence of length N and end with a representation that contains $N+1$ elements. However, this sequence has a structure that makes it highly amenable to compression. In particular we will use a method of coding called move-to-front (*mtf*), which is particularly effective on the type of structure exhibited by the sequence L.

Before we describe the *mtf* approach, let us work through an example to generate the L sequence.

Example 6.4.1:

Let's encode the sequence

thisƀisƀthe

We start with all the cyclic permutations of this sequence. As there are a total of 11 characters, there are 11 permutations, shown in Table 6.14.

TABLE 6.14 **Permutations of *thisƀisƀthe*.**

0	*t*	*h*	*i*	*s*	*ƀ*	*i*	*s*	*ƀ*	*t*	*h*	*e*
1	*h*	*i*	*s*	*ƀ*	*i*	*s*	*ƀ*	*t*	*h*	*e*	*t*
2	*i*	*s*	*ƀ*	*i*	*s*	*ƀ*	*t*	*h*	*e*	*t*	*h*
3	*s*	*ƀ*	*i*	*s*	*ƀ*	*t*	*h*	*e*	*t*	*h*	*i*
4	*ƀ*	*i*	*s*	*ƀ*	*t*	*h*	*e*	*t*	*h*	*i*	*s*
5	*i*	*s*	*ƀ*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*ƀ*
6	*s*	*ƀ*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*ƀ*	*i*
7	*ƀ*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*ƀ*	*i*	*s*
8	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*ƀ*	*i*	*s*	*ƀ*
9	*h*	*e*	*t*	*h*	*i*	*s*	*ƀ*	*i*	*s*	*ƀ*	*t*
10	*e*	*t*	*h*	*i*	*s*	*ƀ*	*i*	*s*	*ƀ*	*t*	*h*

TABLE 6.15 Sequences sorted into lexicographic order.

0	ƀ	i	s	ƀ	t	h	e	t	h	i	s
1	ƀ	t	h	e	t	h	i	s	ƀ	i	s
2	e	t	h	i	s	ƀ	i	s	ƀ	t	h
3	h	e	t	h	i	s	ƀ	i	s	ƀ	t
4	h	i	s	ƀ	i	s	ƀ	t	h	e	t
5	i	s	ƀ	i	s	ƀ	t	h	e	t	h
6	i	s	ƀ	t	h	e	t	h	i	s	ƀ
7	s	ƀ	i	s	ƀ	t	h	e	t	h	i
8	s	ƀ	t	h	e	t	h	i	s	ƀ	i
9	t	h	e	t	h	i	s	ƀ	i	s	ƀ
10	t	h	i	s	ƀ	i	s	ƀ	t	h	e

Now let's sort these sequences in lexicographic (dictionary) order (Table 6.15). The sequence of last letters L in this case is

$$L : sshtthƀiiƀe$$

Notice how like letters have come together. If we had a longer sequence of letters, the *runs* of like letters would have been even longer. The *mtf* algorithm, which we will describe later, takes advantage of these runs.

The original sequence appears as sequence number 10 in the sorted list, so the encoding of the sequence consists of the sequence L and the index value 10. ♦

Now that we have an encoding of the sequence, let's see how we can decode the original sequence by using the sequence L and the index to the original sequence in the sorted list. The important thing to note is that all the elements of the initial sequence are contained in L. We just need to figure out the permutation that will let us recover the original sequence.

The first step in obtaining the permutation is to generate the sequence F consisting of the first element of each row. That is simple to do because we lexicographically ordered the sequences. Therefore, the sequence F is simply the sequence L in lexicographic order. In our example this means that F is given as

$$F : ƀƀehhiisstt$$

We can use L and F to generate the original sequence. Look at Table 6.15 containing the cyclically shifted sequences sorted in lexicographic order. Because each row is a cyclical shift, the letter in the first column of any row is the letter appearing after the last column in the row in the original sequence. If we know that the original sequence is in the k^{th} row, then we can begin unraveling the original sequence starting with the k^{th} element of F.

Example 6.4.2:

In our example

$$F = \begin{bmatrix} \not{b} \\ \not{b} \\ e \\ h \\ h \\ i \\ i \\ s \\ s \\ t \\ t \end{bmatrix} \quad L = \begin{bmatrix} s \\ s \\ h \\ t \\ t \\ h \\ \not{b} \\ i \\ i \\ \not{b} \\ e \end{bmatrix}$$

the original sequence is sequence number 10, so the first letter in of the original sequence is $F[10] = t$. To find the letter following t we look for t in the array L. There are two t's in L. Which should we use? The t in F that we are working with is the lower of two t's, so we pick the lower of two t's in L. This is $L[4]$. Therefore, the next letter in our reconstructed sequence is $F[4] = h$. The reconstructed sequence to this point is th. To find the next letter, we look for h in the L array. Again there are two h's. The h at $F[4]$is the lower of two h's in F, so we pick the lower of the two h's in L. This is the fifth element of L, so the next element in our decoded sequence is $F[5] = i$. The decoded sequence to this point is thi. The process continues as depicted in Figure 6.1 to generate the original sequence.

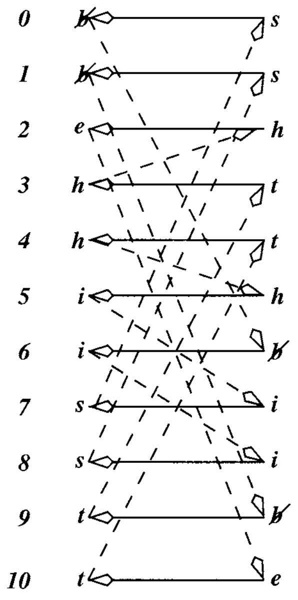

FIGURE 6. 1 **Decoding process.** ♦

Why go through all this trouble? After all, we are going from a sequence of length N to another sequence of length N plus an index value. It appears that we are actually causing expansion instead of compression. The answer is that the sequence L can be compressed much more efficiently than the original sequence. Even in our small example we have runs of like symbols. This will happen a lot more when N is large. Consider a large sample of text that has been cyclically shifted and sorted. Consider all the rows of A beginning with *heb̸*. With high probability *heb̸* would be preceded by t. Therefore, in L we would get a long run of ts.

6.4.1 Move-to-Front Coding

A coding scheme that takes advantage of long runs of identical symbols is the move-to-front (*mtf*) coding. In this coding scheme, we start with some initial listing of the source alphabet. The symbol at the top of the list is assigned the number 0, the next one is assigned the number 1, and so on. The first time a particular symbol occurs, the number corresponding to its place in the list is transmitted. Then it is moved to the top of the list. If we have a run of this symbol, we transmit a sequence of 0s. This way, long runs of different symbols get transformed to a large number of 0s. Applying this technique to our example does not produce very impressive results due to the small size of the sequence, but we can see how the technique functions.

Example 6.4.3:

Let's encode $L = sshtthb̸iib̸e$. Let's assume that the source alphabet is given by

$$\mathcal{A} = \{b̸, e, h, i, s, t\}.$$

We start out with the assignment

0	1	2	3	4	5
$b̸$	e	h	i	s	t

The first element of L is s, which gets encoded as a 4. We then move s to the top of the list, which gives us

0	1	2	3	4	5
s	$b̸$	e	h	i	t

The next s is encoded as 0. Because s is already at the top of the list, we do not need to make any changes. The next letter is h, which we encode as 3. We then move h to the top of the list:

0	1	2	3	4	5
h	*s*	*b̸*	*e*	*i*	*t*

The next letter is *t*, which gets encoded as 5. Moving *t* to the top of the list, we get

0	1	2	3	4	5
t	*h*	*s*	*b̸*	*e*	*i*

The next letter is also a *t*, so that gets encoded as a 0.

Continuing in this fashion, we get the sequence

$$4\ 0\ 3\ 5\ 0\ 1\ 3\ 5\ 0\ 1\ 5$$

As we warned, the results are not too impressive with this small sequence, but we can see how we would get large numbers of 0s and small values if the sequence to be encoded was longer. ♦

6.5 Associative Coder of Buyanovsky (ACB)

A different approach to using contexts for compression is employed by the eponymous compression utility developed by George Buyanovsky. The details of this very efficient coder are not well known; however, the way the context is used is interesting and we will briefly describe this aspect of ACB. More detailed descriptions are available in [69] and [70]. The ACB coder develops a sorted dictionary of all encountered contexts. In this it is similar to other context based encoders. However, it also keeps track of the *contents* of these contexts. The content of a context is what appears after the context. In a traditional left-to-right reading of text, the contexts are unbounded to the left and the contents to the right (to the limits of text that has already been encoded). When encoding the coder searches for the longest match to the current context reading right to left. This again is not an unusual thing to do. What is interesting is what the coder does after the best match is found. Instead of simply examining the *content* corresponding to the best matched context, the coder also examines the *contents* of the coders in the neighborhood of the best matched contexts. Fenwick [69] describes this process as first finding an anchor point then searching the *contents* of the neighboring contexts for the best match. The location of the anchor point is known to both the encoder and the decoder. The location of the best *content* match is signalled to the decoder by encoding the offset δ of the context of this *content* from the anchor point. We have not specified what we mean by "best" match. The coder takes the utilitarian approach that the best match is the one that ends up providing the most compression. Thus, a longer match farther away from the anchor may not be as advantageous as a shorter match closer to the anchor because of the number of bits required to encode δ. The length of the match λ is also sent to the decoder.

The interesting aspect of this scheme is that it moves away from the idea of exactly matching the past. It provides a much richer environment and flexibility to enhance the compression and will, hopefully, provide a fruitful avenue for further research.

6.6 Dynamic Markov Compression

Quite often the probabilities of the value that the next symbol in a sequence takes on depend not only on the current value but on the past values as well. The *ppm* scheme relies on this longer-range correlation. The *ppm* scheme, in some sense, reflects the application, that is, text compression, for which it is most used. Dynamic Markov compression (DMC), introduced by Cormack and Horspool [71], uses a more general framework to take advantage of relationships and correlations, or contexts, that extend beyond a single symbol.

Consider the sequence of pixels in a scanned document. The sequence consists of runs of black and white pixels. If we represent black by 0 and white by 1, we have runs of 0s and 1s. If the current value is 0, the probability that the next value is 0 is higher than if the current value was 1. The fact that we have two different sets of probabilities is reflected in the two-state model shown in Figure 6.2. Consider state A. The probability of the next value being 1 changes depending on whether we reached state A from state B or from state A itself. We can have the model reflect this by *cloning* state A, as shown in Figure 6.3, to create state A'. Now if we see a white pixel after a run of black pixels, we go to state A'. The probability that the next value will be 1 is very high in this state. This way, when we estimate probabilities for the next pixel value, we take into account not only the value of the current pixel but also the value of the previous pixel.

This process can be continued as long as we wish to take into account longer and longer histories. "As long as we wish" is a rather vague statement when it comes to implementing the algorithm. In fact, we have been rather vague about a number of implementation issues. We will attempt to rectify the situation.

There are a number of issues that need to be addressed in order to implement this algorithm:

1. What is the initial number of states?
2. How do we estimate probabilities?

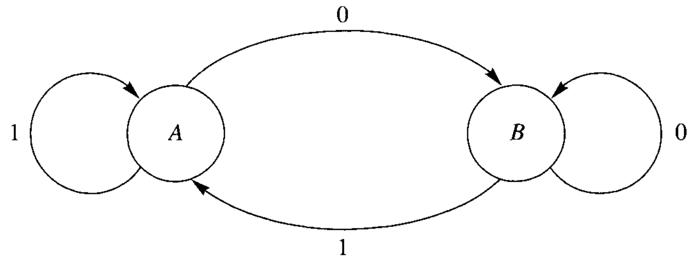

FIGURE 6.2 A two-state model for binary sequences.

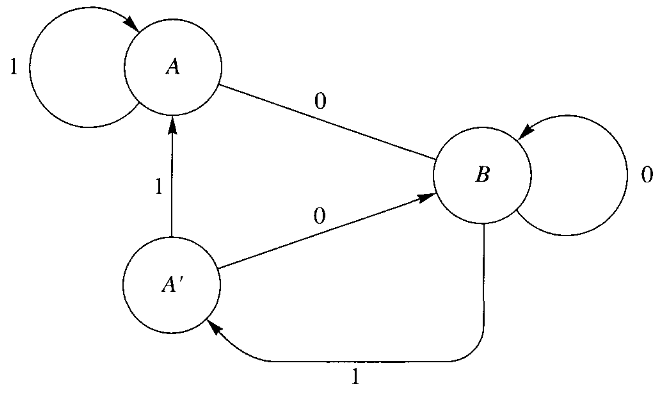

FIGURE 6. 3 **A three-state model obtained by cloning.**

3. How do we decide when a state needs to be cloned?
4. What do we do when the number of states becomes too large?

Let's answer each question in turn.

We can start the encoding process with a single state with two self-loops for 0 and 1. This state can be cloned to two and then a higher number of states. In practice it has been found that, depending on the particular application, it is more efficient to start with a larger number of states than one.

The probabilities from a given state can be estimated by simply counting the number of times a 0 or a 1 occurs in that state divided by the number of times the particular state is occupied. For example, if in state V the number of times a 0 occurs is denoted by n_0^V and the number of times a 1 occurs is denoted by n_1^V, then

$$P(0|V) = \frac{n_0^V}{n_0^V + n_1^V}$$
$$P(1|V) = \frac{n_1^V}{n_0^V + n_1^V}.$$

What if a 1 has never previously occurred in this state? This approach would assign a probability of zero to the occurrence of a 1. This means that there will be no subinterval assigned to the possibility of a 1 occurring, and when it does occur, we will not be able to represent it. In order to avoid this, instead of counting from zero, we start the count of 1s and 0s with a small number c and estimate the probabilities as

$$P(0|V) = \frac{n_0^V + c}{n_0^V + n_1^V + 2c}$$
$$P(1|V) = \frac{n_1^V + c}{n_0^V + n_1^V + 2c}.$$

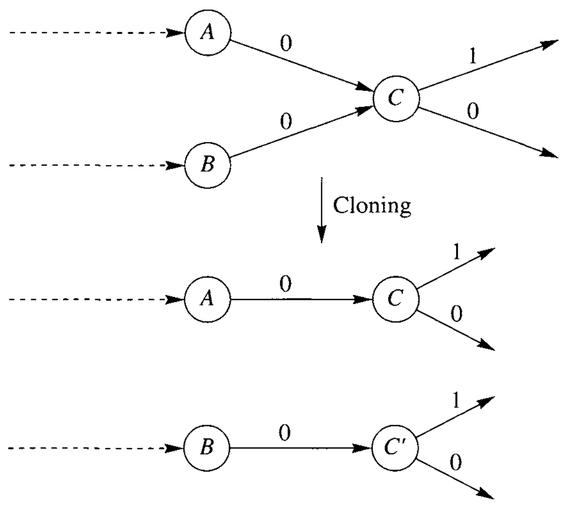

FIGURE 6.4 **The cloning process.**

Whenever we have two branches leading to a state, it can be cloned. And, theoretically, cloning is never harmful. By cloning we are providing additional information to the encoder. This might not reduce the rate, but it should never result in an increase in the rate. However, cloning does increase the complexity of the coding, and hence the decoding, process. In order to control the increase in the number of states, we should only perform cloning when there is a reasonable expectation of reduction in rate. We can do this by making sure that both paths leading to the state being considered for cloning are used often enough. Consider the situation shown in Figure 6.4. Suppose the current state is A and the next state is C. As there are two paths entering C, C is a candidate for cloning. Cormack and Horspool suggest that C be cloned if $n_0^A > T_1$ and $n_0^B > T_2$, where T_1 and T_2 are threshold values set by the user. If there are more than three paths leading to a candidate for cloning, then we check that both the number of transitions from the current state is greater than T_1 and the number of transitions from all other states to the candidate state is greater than T_2.

Finally, what do we do when, for practical reasons, we cannot accommodate any more states? A simple solution is to restart the algorithm. In order to make sure that we do not start from ground zero every time, we can train the initial state configuration using a certain number of past inputs.

6.7 Summary

The context in which a symbol occurs can be very informative about the value that the symbol takes on. If this context is known to the decoder then this information need not be encoded: it can be inferred by the decoder. In this chapter we have looked at several creative ways in which the knowledge of the context can be used to provide compression.

Further Reading

1. The basic *ppm* algorithm is described in detail in *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1].
2. For an excellent description of Burrows-Wheeler Coding, including methods of implementation and improvements to the basic algorithm, see "Burrows-Wheeler Compression," by P. Fenwick [72] in *Lossless Compression Handbook.*
3. The ACB algorithm is described in "Symbol Ranking and ACB Compression," by P. Fenwick [69] in the *Lossless Compression Handbook*, and in *Data Compression: The Complete Reference* by D. Salomon [70]. The chapter by Fenwick also explores compression schemes based on Shannon's experiments.

6.8 Projects and Problems

1. Decode the bitstream generated in Example 6.3.1. Assume you have already decoded *thisƀis* and Tables 6.1–6.4 are available to you.
2. Given the sequence *theƀbetaƀcatƀateƀtheƀcetaƀhat*:
   (a) Encode the sequence using the *ppma* algorithm and an adaptive arithmetic coder. Assume a six-letter alphabet $\{h, e, t, a, c, \not{b}\}$.
   (b) Decode the encoded sequence.
3. Given the sequence *etaƀcetaƀandƀbetaƀceta*:
   (a) Encode using the Burrows-Wheeler transform and move-to-front coding.
   (b) Decode the encoded sequence.
4. A sequence is encoded using the Burrows-Wheeler transform. Given $L = elbkkee$, and index = 5 (we start counting from 1, not 0), find the original sequence.