DATA COMPRESSION USING DYNAMIC MARKOV MODELLING

G.V. Cormack

Department of Computer Science
University of Waterloo

R.N.S. Horspool

Department of Computer Science
University of Victoria

Research Report CS-86-18

May 1986

DATA COMPRESSION USING DYNAMIC MARKOV MODELLING

G. V. CORMACK* AND R. N. S. HORSPOOL†

A method of dynamically constructing Markov chain models that describe the characteristics of binary messages is developed. Such models can be used to predict future message characters and can therefore be used as a basis for data compression. To this end, the Markov modelling technique is combined with Guazzo's arithmetic coding scheme to produce a powerful method of data compression. The method has the advantage of being adaptive: messages may be encoded or decoded with just a single pass through the data. Experimental results reported here indicate that the Markov modelling approach generally achieves much better data compression than that observed with competing methods on typical computer data.

1. INTRODUCTION

All data compression methods rely on *a priori* assumptions about the structure of the source data. Huffman coding[7], for example, assumes that the source data consists of a stream of characters that are independently and randomly selected according to some (possibly unknown) static probabilities. The assumption that we make in this paper is remarkably weak. We assume only that the source data is a stream of bits generated by a discrete-parameter Markov chain model[11]. Such a model is sufficiently general to describe the assumptions of Huffman coding, of run-length encoding, and of many other compression techniques in common use. The survey paper by Severance[10] provides a good introduction to data compression and to various compression techniques that might be employed.

A similar assumption, that there is an underlying Markov model for the data, is made in the Ziv-Lempel [13,14] and the Cleary-Witten[1] techniques. In fact, it is provable that Ziv-Lempel coding approaches the optimal compression factor for sufficiently long messages that are generated by a Markov model.

The new direction taken in our work is an algorithmic attempt to discover a Markov chain model that describes the data. If such a model can be constructed from the first part of a message, it can be used to predict forthcoming binary characters. Each state in the Markov chain supplies probabilities for the next binary character being a zero or a one. After using the probability estimate in a data coding scheme, we can use the actual message character to transfer to a new state

* Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

† Department of Computer Science, University of Victoria, P.O. Box 1700, Victoria, B.C. V8W 2Y2, Canada. (Address for correspondence.)

in the Markov chain. This new state is then used to predict the next message bit, and so on.

If the probability estimates for binary characters deviate from 0.5 and are reasonably accurate, they can be used as the basis of a data compression method. In our implementation, they are used directly to control a minimum-redundancy coding method known as arithmetic coding[9]. We use a particular form of arithmetic coding that was invented by Guazzo[5]. The combination of Markov chain model generation with Guazzo encoding has turned out to be a very powerful method of compressing data. As our experimental results show, it compares very favourably with the competing compression methods. Although some of our performance results are similar to those achievable with the method of Cleary and Witten,[1] we argue that our method is intrinsically simpler and, in any case, requires less storage and consumes less CPU time.

Because the reader may not be familiar with Guazzo's approach to data compression, this paper contains a fairly long and intuitive description of the method. We believe that our introduction will provide insights into the method that are not readily available from reading the original paper. Following this, we present a brief description of adaptive coding and then we explain our method of automatically constructing Markov chain models. Next, we return to the Guazzo coding method and discuss problems associated with the coding and decoding algorithms. Full implementation details of these algorithms are provided in an appendix to the paper. Finally, we present results obtained from an implementation of our data compression algorithm (which we will refer to as *DMC*, for *Dynamic Markov Compression*). These results are compared with other data compression techniques.

The main contribution of this paper is, perhaps, the Markov modelling technique. The fact that this technique can be allied with arithmetic coding to achieve an excellent degree of data compression is a happy accident. As we point out in the conclusions, the modelling algorithm may have other applications. Unfortunately, the modelling technique does not appear to be amenable to analysis. We have no results to prove that the dynamically changing model converges, in some sense, to the *true* model for the data. Such a result is unlikely in any case, because the number of states in the model grows without limit. The modelling technique can be judged only on the basis that it works, and apparently works extremely well.

2. INTRODUCTION TO THE GUAZZO CODING ALGORITHM

Guazzo's algorithm[5] generates minimum-redundancy codes, suitable for discrete message sources with memory. The term *message source with memory* is used to describe a message that has been generated by a Markov chain model. In practical situations, messages almost always exhibit some degree of correlation between adjacent characters and this corresponds to the message source having memory.

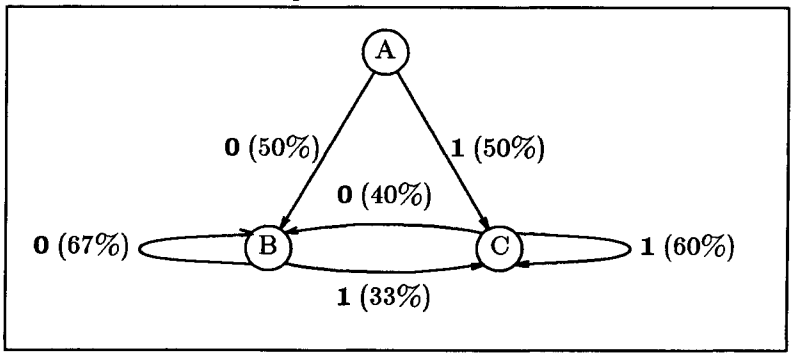
To introduce the subject, we use a simple Markov model as an example. In our example, we assume that the source message string consists of binary digits where

0 is followed by 0 with probability 2/3
0 is followed by 1 with probability 1/3
1 is followed by 0 with probability 2/5
1 is followed by 1 with probability 3/5

and we also assume that the first digit in the message is equally likely to be 0 or 1. The messages generated by this model have the overall property that a zero digit is more likely to be followed by another zero than by a one. Similarly, a one digit is more likely to be followed by a one digit. The state transition diagram for a model that exactly encapsulates these probabilities is shown in Figure 1.

Working with this particular model, we will first examine why the well-known Huffman coding[7] is less than ideal and then we will proceed to see how Guazzo coding can achieve near-optimal codes.

FIGURE 1. An Example Markov Model



2.1. Huffman Coding of Message Sources with Memory

We begin by considering how Huffman coding can be used for messages generated by the Markov model given in Figure 1. Huffman coding cannot be directly applied to a binary source alphabet. Having to provide separate codes for a zero bit and a one bit implies that compression is impossible.

The usual way around the problem is to create a larger source alphabet. This is easily done by grouping message characters together. For example, we can choose to work with groups of three bits, yielding an alphabet of size 8. Through an analysis of the Markov model in Figure 1, involving the computation of equilibrium state probabilities, we have calculated the probability of occurrence for each character in the new source alphabet. These probabilities can be used to derive Huffman codes. Table 1 shows the probabilities and corresponding codes, assuming that bits generated by the model of Figure 1 are grouped into threes.

If our Huffman codes for triples are used to encode a long message generated by the Markov chain model, a moderate degree of compression is achieved. In fact, a long message will, on average, be compacted to approximately 90.3% of its original size. However, this number is considerably larger than the information-theoretic lower bound. A simple calculation of the entropy in the information source shows that compression to 81.9% of the original size should be achievable.

There are two reasons why a Huffman code does not achieve the lower bound. The first reason is that Huffman codes can be free of redundancy only if character frequencies in the source

Source Form	Probability	Huffman Code
000	0.2424	10
001	0.1212	010
010	0.0727	1101
011	0.1091	000
100	0.1212	011
101	0.0606	1100
110	0.1091	001
111	0.1636	111

TABLE 1. Huffman Codes for 3-bit Groups

alphabet are integer powers of 1/2. The second reason is that the Huffman code is taking advantage of correlations only between the bits that we have grouped together. If, for example, our source message contains the two adjacent groups

... 001 110 ...

the Huffman scheme encodes both groups independently. But this ignores correlation between the last bit of the first group and the first bit of the second group. Therefore, the second group is encoded in a sub-optimal manner. These two problems with the Huffman scheme are ameliorated only by choosing larger groups of bits for constructing a source alphabet. As we choose larger and larger groups, the coding efficiency comes closer to the lower-bound but the alphabet size grows exponentially. The storage for tables needed to hold Huffman encodings also grows exponentially, while the computational cost of creating these tables becomes infeasible.

An alternative to imprudent expansion of the alphabet size is to use multiple sets of Huffman codes. The choice of which set of codes to use for the next message character is determined by the preceding message characters. Such an approach can be tuned to achieve a reasonable compromise between compression performance and the total amount of storage needed to hold tables of Huffman codes[2].

2.2. Guazzo Encoding applied to the Markov Model

The Guazzo method does not suffer from either of the defects noted for Huffman coding. For sufficiently long messages, the method can generate encodings that come arbitrarily close to the information-theoretic lower bound. The first step in understanding the binary version of the Guazzo method is to consider the output encoding as being a binary fraction. For example, if the output encoding begins with the digits

0 1 1 0 1 ...

we should consider this as being the binary number that begins '0.01101...'. (This is a number which has a value close to 13/64 when expressed as a decimal fraction). The job of the encoding algorithm is, in effect, to choose a fractional number between zero and one which encodes the entire source message.

Given that the Guazzo algorithm has access to the Markov model shown in Figure 1, we can trace the algorithm and see how it would choose an encoding for an indefinitely long source message that begins

0 1 1 1 0 0 1 ...

Initially, the Guazzo algorithm has freedom to choose binary fractions that lie between

0.000... and 0.111...

inclusive. Guazzo's algorithm determines from the Markov model that the first digit of the source message is equally likely to be zero or one. It therefore divides the space of binary fractions into two equal halves, choosing all fractions in the closed interval [0.000..., 0.0111...] to represent a source message that begins with '0', and all fractions in the closed interval [0.1000..., 0.111...] to represent messages that begin with '1'. Our source message begins with zero and therefore we must pick the first sub-interval. Since all binary fractions in the selected subrange begin with the digit '0', this effectively determines that the first output digit is zero.

The first source digit takes us to the state labelled B in our Markov model. In this state, the next digit is twice as likely to be a zero as a one. Guazzo's algorithm therefore determines that the range of available binary fractions should be subdivided into two parts in the ratio two to one. After the split, the sub-interval [0.000..., 0.010101...) represents source messages that begin '00...' and the sub-interval [0.010101..., 0.0111..] represents messages beginning with '01'. The first of these two sub-intervals has exactly twice the range of the second. Since the second digit of the source message is one, we are restricted to the second interval. And since this source digit leads us to state C in the model, we now have to split the available range of message encodings in a three to

two ratio.

Rather than proceeding through the example at this level of detail, we will skip a few steps. Continuing along the same lines as before, the Guazzo algorithm will eventually determine that our source message beginning with '0111001' should be represented by some binary fraction in the interval

[0.011100110101..., 0.01110100101111...]

Since the lower and upper bounds of the interval begin with the same five fractional digits, the encoding algorithm could have already generated these digits (namely '01110').

If we consider the requirements of a practical computer implementation, it is clear that interval bounds should not be computed as binary fractions containing unlimited numbers of digits. An implementation that requires only a small, finite, number of bits to be retained in the calculations of the interval bounds is essential. We defer consideration of this and other practical issues to a later section of this paper.

To add some further intuition to our intuitive description of the Guazzo algorithm, we offer a simple observation. The way that the available space of binary encodings is subdivided at each step of the algorithm distributes encoded messages as evenly as possible throughout this space. It is important to distribute encodings evenly, otherwise some encodings will be unnecessarily close together – and two values that are almost the same will usually require more bits to differentiate them than two values that are farther apart.

The encoding process is easily reversible. The decoding algorithm has access to the same Markov model for the information source as was used by the encoder. It starts by constructing the same interval [0.000..., 0.111...] of possible encodings and by subdividing it in the same way as the encoder subdivided it. Inspection of the leading bit(s) in the encoded message then determines which of the two partitions must have been used by the encoder and therefore determines what the first source digit must have been. Once this first digit is known, the decoding algorithm can select the appropriate partition and repeat the division into two parts. Inspection of the encoded message now determines the second digit, and so on.

3. ADAPTIVE CODING

Data compression has traditionally been implemented as a two-pass technique. An initial pass through the source message is performed to discover its characteristics. Then, using knowledge of these characteristics, a second pass to perform the compression is made. For example, if we are using Huffman coding, the first pass would count frequencies of occurrence of each character. The Huffman codes can then be constructed before the second pass performs encoding. To ensure that the compressed data can be decoded, either a fixed coding scheme must be used or else details of the compression scheme must be included with the compressed data. Although a two-pass implementation for our new data compression technique would be easy to develop, we prefer to proceed directly to a one-pass *adaptive* data compression implementation. One-pass implementations are preferable because they do not require the entire message to be saved in on-line computer memory before encoding can take place.

There is a one-pass technique for data compression that, in practice, achieves compression very close to that obtained with two-pass techniques. The basic idea is that the encoding scheme changes dynamically while the message is being encoded. The coding scheme used for the k-th character of a message is based on the characteristics of the preceding k—1 characters in the message. This technique is known as *adaptive coding.* For example, adaptive versions of Huffman coding have been proposed and implemented[3,4,8]. In practice, adaptive Huffman coding achieves data compression that differs insignificantly from conventional two-pass Huffman coding, but at the expense of considerably more computational effort.

The Ziv-Lempel[12,13,14] and the Cleary-Witten[1] methods are also adaptive coding techniques. The basic idea of Ziv-Lempel coding is that a group of characters in the message may be replaced

by a pointer to an earlier occurrence of that character group in the message. After a short learning period, these pointers will consistently occupy fewer bits than the character groups they replace. This algorithm can be implemented in such a way as to be much faster than adaptive Huffman coding, while achieving much better data compression. Nevertheless, the Ziv-Lempel algorithm takes advantage only of correlations among source characters that happen to be grouped together: all context is discarded between such groups. The scheme therefore suffers from the same defect described in Section 2.1; the loss of context is ameliorated only by using longer and longer groups, with commensurate consumption of memory resources.

The method of Cleary and Witten uses the preceding k characters in a message to predict the probabilities for the current message character and uses these probabilities to drive an arithmetic coding scheme. To this end, a table of all previous occurrences of strings of length k is kept, along with the counts of characters following each string. However, if the particular combination of k characters has not appeared previously in the message, we will be unable to estimate the probabilities using the table. In this case, the Cleary and Witten method escapes to use a table of the preceding k-1 characters which is more likely to be able to make an estimate of probabilities. If the combination of k-1 preceding characters has not previously occurred either, another escape will take place, and so on. (The null string will always make a prediction.) Each table of size k may be regarded as a sparse representation of a (very large) k th-order Markov model. The main point is that the probabilities used in the various Markov models are learned as the message is being processed. Since the decoder can be programmed to learn in the same way as its decodes characters, it is easy to construct an adaptive compression algorithm.

The Guazzo coding algorithm is eminently suitable for use in adaptive coding schemes. The only aspect of the algorithm that need change dynamically is the source of probability estimates for message characters. At each step of the encoding process, the algorithm requires probability estimates for each of the possibilities for the next message character. It does not matter to the Guazzo algorithm whether these probability estimates are derived from a static Markov model or from a dynamically changing Markov model. In such a dynamic model, both the set of states and the transition probabilities may change, based on message characters seen so far. The next section of this paper explains our method for dynamically building a Markov model for the source message.

Decoding a message produced by an adaptive coding implementation of the Guazzo algorithm should not prove to be a problem either. All that the decoding algorithm needs to do is to recreate the same sequence of changes to the dynamically changing Markov model as were made by the encoding algorithm. Since the decoding algorithm sees exactly the same sequence of unencoded digits as the encoding algorithm, there is no difficulty.

4. DYNAMIC CONSTRUCTION OF PREDICTIVE MARKOV MODELS

A Markov chain model can be characterized as a directed graph with probabilities attached to the graph edges. We can distinguish two different aspects to the problem of creating such a Markov model automatically. One part is the determination of suitable probabilities to place on the edges of the graph. The other part is the determination of the structure of the graph itself. We consider these two parts separately, beginning with the easier problem of choosing probabilities for the transitions in a given model.

4.1. Choosing Edge Probabilities

To begin our explanation, we assume that we already have a Markov model but that there are no probabilities attached to the transitions. We further assume, as a starting point, that correlations exist between a message character and the immediately preceding characters.

If we have been reading binary digits from a source message, we could have been following the corresponding transitions in the model and have been counting how many times each transition in the model has been taken. These counts provide reasonable estimators of the probabilities for those transitions that have been taken many times. More precisely, if the transition out of state A

for digit 0 has been taken n_0 times and the transition for digit 1 has been taken n_1 times, then the following probability estimates are reasonable:

$$\text{Prob } \{ \text{ digit} = 0 \mid \text{current state} = \text{A } \} = n_0 / (n_0 + n_1)$$

$$\text{Prob } \{ \text{ digit} = 1 \mid \text{current state} = \text{A } \} = n_1 / (n_0 + n_1)$$

The more often we have visited state A, the more confidence we would have in these probability estimates.

Unfortunately, the adaptive coding technique requires us to begin to make probability estimates before these transition counts have grown to significant values. Furthermore, the above formulae are undefined for a first visit to a state and yield transition probabilities of 0% and 100% on the next few visits. We must be especially careful not to supply the Guazzo algorithm with a 0% probability for any bit because the generated encoding would have an infinite length if that bit were actually observed. There are many ways in which the probability formulae can be adjusted to take account of these two concerns. The method which we used in our implementation is, perhaps, the simplest. We adjusted the formulae to be

$$\text{Prob } \{ \text{ digit} = 0 \mid \text{current state} = \text{A } \} = (n_0 + c) / (n_0 + n_1 + 2c)$$

$$\text{Prob } \{ \text{ digit} = 1 \mid \text{current state} = \text{A } \} = (n_1 + c) / (n_0 + n_1 + 2c)$$

where c is a positive constant. Using small values for c is equivalent to having confidence in probability estimates based on small sample sizes, whereas large values correspond to having little confidence. On the other hand, an adaptive algorithm will seem to 'learn' the characteristics of a source file faster if small values for c are used, but at the expense of making poor predictions more often. If very large files are being compressed, the choice of c becomes largely irrelevant.

4.2. Building the State Transition Graph

The method by which probabilities attached to transitions in the Markov model change dynamically has just been explained. What has not been explained is the method by which the set of states in the model changes dynamically. We will try to explain this method through consideration of a simple scenario. Suppose that we have a partially constructed model which includes states named A, B ... E, as drawn in Figure 2(a). The figure shows that there are transitions from both A and B to C, and transitions from C to both D and E. Now, whenever the model enters state C, some contextual information is lost. In effect, we forget whether we reached state C from A or from B. But it is quite possible that the choice of next state, D or E, is correlated with the previous state, A or B. An easy way to learn whether such a correlation exists is to duplicate state C (we call this process *cloning*), generating a new state C′. This creates a revised Markov model as drawn in Figure 2(b). After this change to the model, the counts for transitions from C to D or E will be updated only when state C is reached from A, whereas the counts for C′ to D or E will be updated only when C′ is reached from B. Thus the model can now learn the degree of correlation between the A, B states and the D, E states.

If the above cloning process is performed when, in fact, no correlation between the previous state and the next state exists, little has been lost. We have simply made the model more complicated (more states) and made our probability estimates more susceptible to statistical fluctuations (because each state is visited less often). If such correlations do exist, the improvements in the probability estimates can be dramatic. Carrying on with the model of Figure 2(b), it is possible that the choice of next state (D or E) is not correlated with the previous state being A or B but *is* correlated with the states immediately before state A or state B. If this is the case, then cloning state A, cloning state B, cloning state C′ and recloning state C will enable our model to discover the correlations. In general, the more cloning that is performed, the longer the range of correlations that can be discovered and be used for predictive purposes.

In light of the previous observation, our implementation performs cloning as soon as practicable. Whether it is practicable to clone a particular state depends on whether that state has been

FIGURE 2(a). Part of a Markov Model

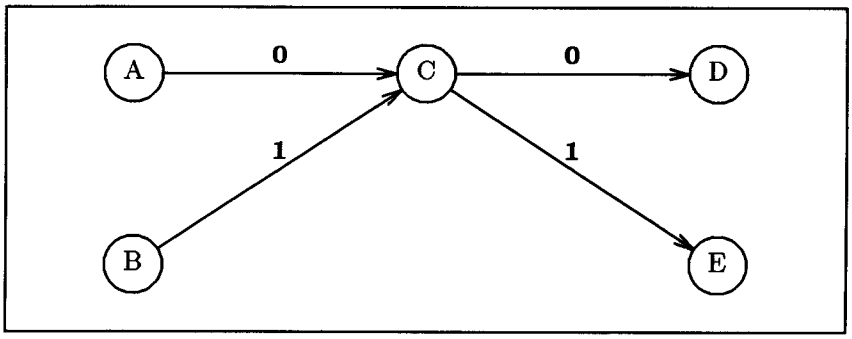
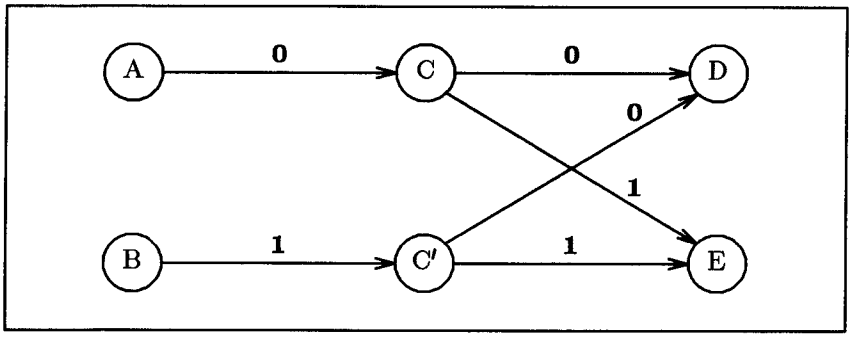


FIGURE 2(b). The Markov Model after 'cloning'



visited a reasonable number of times from each of two (or more) different predecessor states. Referring to Figure 2(a), again, let us assume that the current state is A and a transition is about to be made to state C. The desirability of cloning state C should depend on whether the AC and BC transition counts are both reasonably large. If, say, the BC count is zero or is small compared to the AC count, then the probabilities associated with the transitions leaving C will reflect a correlation with the predecessor being A. Cloning state C would enable correlations with state B to be discerned, but there is little benefit to be gained if the BC transition is rarely taken.

It is assumed that the next transition to be followed in the Markov chain model would transfer from the current state to the candidate state, a state that is eligible for cloning. Generalizing the scenario of Figure 2(a), we have the following cloning criterion. The candidate state is cloned if and only if

> the number of observed transitions from the current state to the candidate state is greater than *MIN_CNT1*, and the number of observed transitions from all states other than the current state to the candidate state is greater than *MIN_CNT2*.

The full algorithm for implementing state cloning appears in the appendix to this paper as Figure A.1. The algorithm also shows how transition counts are apportioned when a state is cloned. By apportioning these counts, Kirchoff's Laws* are maintained. The assumption of Kirchoff's Laws simplifies the logic needed to determine how often the candidate state has been visited from state other than the current state. The choice of suitable values for *MIN_CNT1* and *MIN_CNT2* is discussed later in the paper.

---

* By this we mean that the count of transitions into some state from all its predecessors should be the same as the count of transitions out of that state to all its successors. In practice, the two counts may differ by one because there is always one more transition into the current state than there are transitions that have been taken out of it.

4.3. Starting and Stopping the Model Construction

Two important questions have not yet been answered. The first question is: What Markov model should we begin with? The simple answer is that we need only begin with a minimal model capable of generating any message sequence. It contains only one state. Both transitions out of this state (for the digits 0 and 1) loop back to this single state. This model is diagrammed in Figure 3(a). After operation of the cloning algorithm, this single state model rapidly grows into a complex model with thousands of states.

In practice, there is some benefit to be gained from beginning with a slightly less trivial initial model. Almost all computer data is byte or word aligned. Correlations tend to occur between adjacent bytes more so than between adjacent bits. If we begin with a model that corresponds to byte structure, the process of learning source message characteristics occurs faster, leading to slightly better data compression. A simple model for byte structure has 255 states, arranged as a binary tree, with transitions from each leaf node returning to the root of the tree. The general shape of this tree, but with a smaller number of nodes, is diagrammed in Figure 3(b).

FIGURE 3(a). An Initial One-State Markov Chain Model

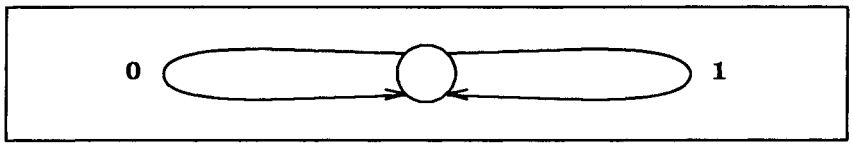
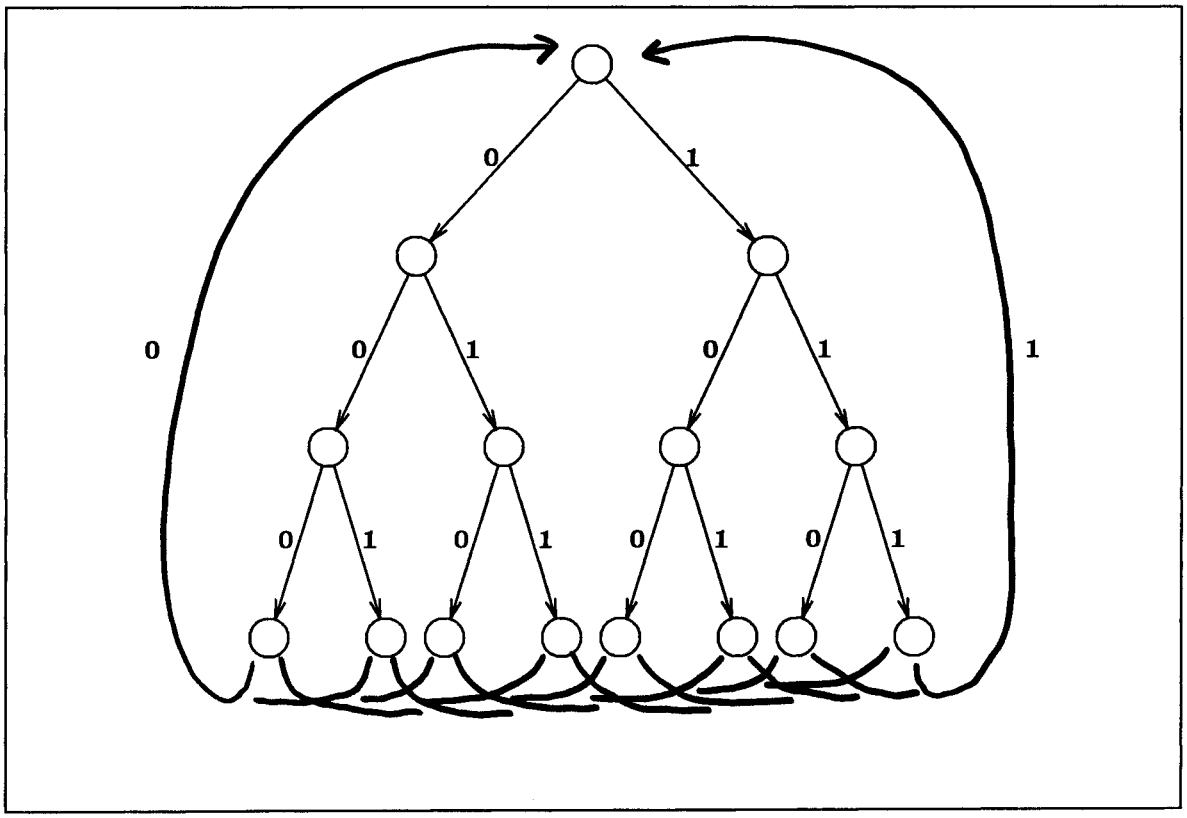


FIGURE 3(b). An Initial Markov Chain Model for 4-bit Characters



Although a tree-structured initial graph works well, it is possible to do a little better. When the transition counts have reached reasonable values, the counts have the potential to show correlations between the individual bits of a byte. For example, if it is the case that the third bit in a byte being a zero implies that the seventh bit is always a one, the tree model would be able to describe this correlation. In general, a state in the k-th level of the tree represents correlations with the preceding k-1 bits. The amount of left context that is correlated with the current state builds up from 0 bits to 7 bits and then the correlation is discarded when a transition to the top (root) of the tree is made. A more satisfactory model is one that retains a constant amount of left context. Apart from being esthetically pleasing, such a model has the capability of learning correlations between the last few bits of one byte and the first few bits of the next byte. A model that keeps the amount of left context constant is, unfortunately, difficult to show in a diagram. It would be easiest to draw on the surface of a torus. We call the model a ***braid*** because of the way in which the transitions interweave when drawn out in two dimensions. An algorithm for the construction of the braid model appears in the appendix to this paper as Figure A.4. The particular form of braid that would be generated by this algorithm is also known as an *omega* network or as a *perfect shuffle* network.

The second question is: when should the cloning process be halted? If it is not halted, there is no bound on the amount of memory needed by our compression algorithm. On the other hand, if it is completely halted, we lose the ability for our algorithm to adapt if some characteristics of the source message change. A possible solution is to set a limit on the number of states. When the limit is reached, the Markov model is discarded and we begin again with the initial model. This drastic solution is more effective than it might appear. However, a less drastic variation on the approach is easily implemented. We can retain the last k bytes of the source message that have been read in a cyclic buffer. When the limit on the number of states is reached, the model is discarded as before. Then, without adding to the encoded message, a new model is constructed by processing the k bytes in the buffer. This should yield a new model with a relatively small number of states that corresponds to the characteristics of the last k message bytes. Although some loss of data compression performance occurs at these storage reclamations, the loss is not very great in practice and the compression algorithm retains its adaptability.

5. PRACTICAL USAGE OF THE GUAZZO ALGORITHM

Guazzo encoding was previously discussed without much regard to the problems associated with a practical implementation. There are two main problem areas which must be addressed. The first problem is that the lower and upper bounds of the interval are rational numbers which, as the algorithm proceeds, must apparently be calculated to ever greater precision. The solution adopted by Guazzo is to weaken the requirement that these bounds be calculated precisely.

At each step of the encoding algorithm, the interval is divided into two parts. If the division is not performed exactly in the same proportions as the ratio of the probabilities of the next source digit being a zero or a one, there is no great loss. The coding technique still works, in the sense that decoding can be uniquely performed. All that is lost is a little coding efficiency. Guazzo therefore proposed that a *fidelity criterion* should be used to determine how precisely the division point between the two subranges is to be calculated. The tighter this criterion is, the better the message encoding is, but at the computational expense of having to calculate the division point more accurately.

For our implementation of the algorithm, we chose the more practical approach of retaining as many significant bits in the calculation of the division point as will conveniently fit in one computer word. And, as soon as message bits are generated (when both bounds of the interval have one or more identical leading digits), they are removed from both variables that record the interval bounds. (They may be removed by logical *shift-left* operations.) Thus, a fairly constant degree of accuracy (about 30 significant bits) is maintained by our implementation. An algorithm organized along these lines is provided in the Appendix as Figure A.2.

The second issue is one that was not addressed by Guazzo. He assumed the message source to be unending. However, this clearly does not suit typical computer applications for data compression. In the example of the previous section, we assumed that the source message was infinite and began with the sequence

0 1 1 1 0 0 1 ...

and we discovered that the encoded message should be some infinite sequence in the interval

[0.011100110101..., 0.01110100101111...]

Suppose, however, that the source message contains only the 7 digits listed above. That is, when the encoding algorithm attempts to read an eighth bit, it receives an end-of-input indication. What should the algorithm do? Our encoded message could be the finite sequence '011101' or it could be '0111010' or '011100111' or ... Which of these encoded messages will be correctly decoded?

If the decoding algorithm treats the finite encoded message as though it were a normal binary fraction, that is, if it treats '011101' and '0111010' and '01110100' ... as being synonymous, then none of these messages will be decoded correctly. The binary fraction '011101' has an *exact* value and therefore the decoding algorithm can continue sub-dividing intervals and generating nonsensical message bits forever.

Our solution to the problem is, in effect, to consider the encoded message sequence '0111010' as representing a range of values [0.0111010000..., 0.01110101111...]. In other words, we treat the encoded message as being '0111010xxxx...', where 'x' represents an unknown bit value. The decoding algorithm should sub-divide intervals and generate message bits as long as there is an unambiguous choice as to which half of the interval the value '0111010xxx...' belongs in. As soon as the choice depends on the value of one or more of the digits denoted by 'x', the decoding algorithm halts.

Clearly it is the responsibility of the encoder to make sure that enough bits are present in the encoded message for the decoder to reconstruct the entire original message without ambiguity. This detail is taken care of in two places in the algorithm of Figure A.2. At the end of the algorithm (at label *999*), the algorithm simply outputs all remaining bits held in the variable *MP* up to, but not including, the rightmost one bit. *MP* holds the tail of the binary encoding for the partition point between the next two subranges. Since the decoding algorithm will discover an exact match between its input data and this partition point, it cannot choose between the upper part of the lower part of the interval. Thus, no superfluous output bit(s) will be created. Note that we stop just before outputting the rightmost one bit in *MP*. If this bit were output, the decoder would be able to deduce that the encoded message is ***greater than or equal*** to *MP* and this would cause the decoder to generate a spurious one bit.

It is not sufficient to observe that no superfluous output bit is created by the decoder. We must also guarantee that that the decoder can reconstitute the entire source message, without losing any of the last few bits. To guarantee that no message bits are lost, we force the calculation of *MP* to make the rightmost bit a one. Thus, at the end of the algorithm, all but the rightmost bit of *MP* must be output. Maximizing the number of bits output by the encoder like this disambiguates any pending selections of interval partitions in the decoder.

The decoding algorithm closely mirrors the encoding algorithm. Indeed, except for some delays when it cannot immediately decide whether to select the upper or lower half of the range partition, its calculations proceed almost in step with the calculations of the encoder. The decoding algorithm is given in the appendix as Figure A.3.

Another implementation difficulty has, for simplicity, been ignored in the implementations reproduced in Figures A.2 and A.3. These two algorithms assume that the message can be encoded as an arbitrary number of bits. In practice, the encoded message would usually have to contain an integral number of bytes (or, perhaps, words). If we simply truncate the encoded message,

dropping up to 7 bits, the decoder may not be able to reproduce the last few bits of the original message. And we cannot append extra bits to the encoded message because these will, almost certainly, be converted into spurious bits at the end of the decoded message. Provided that the original source message contains an integral number of bytes, we have a solution to the problem.

The solution is to append seven extra bits to the source message during the encoding process. Each of these seven source bits is chosen to pessimize the encoding. The encoding algorithm computes *MP*, the dividing point in the interval [*LB*, *HB*], as before. Then it determines which of *LB* or *HB* has more leading bits in common with *MP* (there cannot be a tie). If it is *LB*, the extra bit is chosen to be zero; otherwise it is chosen to be one. Apart from the way the extra bit is generated, it is treated like any other message bit for encoding purposes. With this choice of source bit, at least one new bit is appended to the encoded message. The encoded message may now be safely truncated. Any bits that are lost at the end of the encoded message may cause some bits to be lost from the decoded message. However, bits lost from the decoded message are only some of the extra bits that were added by the encoder. And because our seven extra bits were chosen to pessimize the encoding process, the loss of one encoded message bit can, at worst, cause the loss of one bit in the decoded message. Thus when the message is decoded, zero to seven extra bits will be found at the end of the message. The decoder ignores any incomplete byte at the end of the message, so that no genuine information is lost and no spurious information is gained.

6. RESULTS

Four different data compression algorithms were tested with a variety of data files found on the Berkeley UNIX system (running on a VAX-11/780). These files were chosen because of their large size, making them prime candidates for compression, and because they were fairly typical of files on this system. The file types included formatted documents, unformatted documents (*i.e.* input to the *troff* formatting program), program source code (in the C language), and executable object files.

Our new coding algorithm, *DMC* (for *Dynamic Markov Compression*), was compared against three other compression programs. One program was an adaptive Huffman coding algorithm, as implemented in the UNIX *compact* command[8]. It should be noted that this program yields compression results that are almost indistinguishable from a two-pass (non-adaptive) Huffman coding algorithm.

The second program[†] was a variation, due to T. Welch[12], on the Ziv-Lempel compression algorithm. This variation is labelled LZW in the table. We actually tried out two versions of the LZW program. The standard version remembers sequences of bytes that have occurred in the file and replaces subsequent occurrences of these same sequences with pointers back to the first occurrences. Our second version of LZW remembers sequences of bits (rather than bytes). This bit-oriented version of Liv-Zempel coding is labelled LZ-2 (*2* for binary) in the table of results.

The third program was the Cleary-Witten compression algorithm[1] (called *CW* in the remainder of this paper). As explained previously, this technique involves the construction of k th-order Markov models, where k is a (small) number that must be selected in advance. For the results shown in the table, k was chosen to be 4.

The version of DMC used in the table started with a braid-structured initial model and was not subjected to any memory size constraints. Furthermore, we set the parameters that control the cloning of states in the Markov model to values that give good results. We will say what these values are after discussing the main results.

We compared the five different compression programs on several data files. The resulting compression factors are shown in Table 2. A compression factor is computed as the ratio between

[†] We used version 2.0 of the *compress* program. This program was originally authored by S.W. Thomas of the University of Utah and enhanced by J.M. Orost of Perkin-Elmer Ltd.

the size of the encoded (compressed) file and the size of the original file. For example, a figure of 40% in our table would indicate that a file was compressed to two fifths of its original size.

It is fairly easy to rank the different compression programs in the order of their effectiveness. The only difficulty is in comparing DMC against CW. For ASCII source files, DMC is very slightly behind CW. But for object code files, DMC is well ahead. In fact, DMC is very well suited for files that do not have a homogenous nature, and UNIX object files have a non-homogenous structure.‡ This is because our algorithm is more flexible, continually adapting itself to the data. Although the Adaptive Huffman algorithm, the Ziv-Lempel algorithm, and the Cleary-Witten method can all adapt themselves after a change in file characteristics, they take a relatively long time to adapt. Adaptability is important in some applications, such as in compressing data sent over a communication link. Since this data would normally be formed from a long series of short, unrelated, messages, we would expect the characteristics of the data to change frequently.

We now return to some of the practical details of our compression algorithm. When the dynamic Markov modelling method was described earlier, two free parameters (*MIN_CNT1* and *MIN_CNT2*) were present in the algorithm. Our experiments have shown that these parameters should be chosen so that cloning occurs very early. The results of one such experiment are shown in Table 3, below. In this experiment, we set the two parameters equal and tried values of 1, 2, 4, 8 ... up to 128. Smaller values lead to earlier cloning and therefore to more states in the Markov model. However, the table also shows that small values give the best compression factors. Similar results are obtained if the parameters are varied independently. Our general observation is that promotion of rapid growth in the model leads to the best results. All other results involving our compression algorithm reported in this paper used parameter values of (2,2).

Another detail to be considered is exactly how the number of states in the Markov model should be limited. As Table 3 illustrated, the number of states exceeded 150,000 for an input file

Compression Program	*Source File* Formatted Text[1]	Unformatted Text[2]	Object Code[3]	C Source Code[4]
Adaptive Huffman (*compress*)	59.7%	61.6%	79.6%	62.9%
Normal Ziv-Lempel (*LZW*)	38.2%	42.9%	98.4%	40.8%
Bit-oriented Ziv-Lempel (*LZ-2*)	74.2%	83.6%	91.3%	86.7%
Cleary and Witten (*CW*)	26.5%	30.2%	69.4%	26.3%
Dynamic Markov (*DMC*)	27.2%	31.8%	54.8%	27.5%

[1] Formatted manual entry for the *csh* command (74510 bytes).

[2] Unformatted version of *csh* manual entry (61536 bytes).

[3] Object code for *csh* command (68608 bytes).

[4] Source code for *finger* command (31479 bytes).

TABLE 2. Comparative Compression Results

‡ These files have several sections – including an instruction area part, a data area part, a relocation dictionary and a symbol table.

Parameter Values	Max. Graph Size (number of nodes)	Compression Performance
(1,1)†	>194,000	34.7%
(2,2)	150,901	33.8%
(4,4)	84,090	35.8%
(8,8)	44,296	38.9%
(16,16)	23,889	42.7%
(32,32)	12,089	46.5%
(64,64)	6,347	50.6%
(128,128)	3,211	54.6%
(256,256)	1,711	58.6%

The subject file contained 97,393 characters of ASCII text. (It was the terminal capability database */etc/termcap*.)

† The compression program ran out of storage for graph nodes in the first experiment with parameter values (1,1). The program had compressed more than 90% of the source file when it aborted.

TABLE 3. Varying the Cloning Parameters

holding fewer than 100,000 bytes. A scheme for limiting the model size was previously outlined. We impose an upper limit on the number of nodes in the graph. When the graph grows to reach this limit, the entire graph is discarded and we start over again with the initial small graph. To avoid losing too much compression performance while the compression algorithm 'relearns' the structure of the source data, we buffer the last k bytes of the source input. These k bytes are used to re-build a reasonably small Markov model after a storage reclamation. This leads to two, related, practical questions. First, how is compression performance affected when the number of states in the model is limited? Second, how large should the buffer be? Table 4 may be helpful in providing some answers to these questions.

As one would expect, compression performance improves both as the maximum graph size is increased and as the buffer size is increased. Therefore, the best choices of limits depend on trade-

Buffer Size	Maximum Number of Nodes in Graph						
	5000	10000	15000	20000	25000	30000	35000
100 bytes	53.6%	48.3%	46.3%	45.4%	44.5%	44.0%	43.0%
200 bytes	51.9%	48.8%	47.4%	43.4%	44.8%	44.2%	42.0%
500 bytes	50.9%	46.7%	45.6%	43.9%	44.5%	43.8%	42.8%
1000 bytes	50.0%	46.1%	45.0%	43.5%	43.5%	42.7%	41.4%

The subject file contained 97,394 characters of ASCII text. (It was the terminal capability database */etc/termcap*.) For results in the rightmost column, 4 storage reclamations occurred. For results in the leftmost column, as many as 50 reclamations occurred.

TABLE 4. Choosing Limits on the Markov Model Size

offs between compression efficiency, storage size and execution speed. Increasing the maximum graph size improves compression performance and reduces execution time (because storage reclamations are less frequent). Increasing the buffer size improves compression performance too, while increasing both the storage requirements and the execution time (because more model rebuilding work is performed at each storage reclamation and because reclamations will occur more frequently). However, if the buffer size is made too large, the maximum graph size may be reached while rebuilding the graph.

7. DISCUSSION AND CONCLUSIONS

DMC is a general data compression algorithm that, to the best of our knowledge, achieves some of the best compression results reported in the literature. Text files, for example, are compressed to such an extent that each character requires little more than two bits, on average. Depending on the file, we observed figures in the range of 2.2 to 2.6 bits.

Of the compression algorithms compared in Table 2, the LZW algorithm is, by far, the fastest while the CW method is the slowest. In terms of storage requirements, the adaptive Huffman algorithm uses the least amount of storage and the CW method normally uses the most. However, this observation must be qualified by the fact that the storage used by the LZW and DMC algorithms increases without bound as the source message is processed. As a practical requirement, the amount of storage available for use by LZW and DMC must be artificially limited.

Although DMC has strong competition from the LZW and CW methods, we argue that DMC is a more general approach that has several advantages. Both the LZW and CW methods are, for practical purposes, byte-oriented. It is indeed possible to implement bit-oriented versions of LZW and CW but, in the case of LZW (as seen in Table 2), the results are poor. This is because the learning period for LZW becomes much longer, too long for LZW to achieve reasonable compression on typical files. A bit-oriented version of CW is impractical for a different reason. If it is desired to achieve the same effect as a fourth-order Markov model for bytes, it would be necessary to use a 32nd-order Markov model for bits. The amount of memory required for table storage would simply be astronomical.

On the other hand, DMC is not strongly biased towards byte-oriented data at the expense of bit-oriented data. The use of a byte-oriented starting model (the braid structure) does not cause DMC to perform poorly if the data is not, in fact, byte-oriented. Therefore we argue that DMC is more general and can be applied to files, such as files that have already been subjected to some form of compression, that do not preserve byte-alignment. To some extent, this also explains why DMC outperforms the other compression methods on object code files. The intruction area part of an object file contains small groupings of bits, such as instruction operands, that are not generally aligned within a byte.

However, the implementation of the DMC algorithm described in this paper requires a considerable amount of computation and requires a large amount of memory. We have performed some preliminary work on a fast method of performing Guazzo encoding and have obtained some excellent performance figures. This method, which uses a finite state automaton instead of performing range calculations, will be reported in a future paper. But further work needs to be done to see if it is possible to reduce the storage requirements of DMC without losing too much compression performance. Further experimentation is also needed to study the best choices of values for the parameters that control the cloning process in the DMC algorithm. There is, of course, no reason to require that these values should be static.

A slightly different direction for further research lies in generalizing the algorithm to compress two-dimensional images, such as those generated for raster-type devices. The problem here is that we would like the model to take account of correlations between adjacent scan lines as well as between adjacent points on the same scan line.

We also see applications of the Dynamic Markov modelling method to problems other than data compression. For example, a computer system could use the method to predict accesses to

records in a database and use these predictions to prefetch records. Another possible use of the modelling technique is in game playing programs to model the playing strategy of a human opponent. This idea was used in a program to play the children's game "Rock, Scissors, Paper"[6].

Acknowledgements

Financial support for this research was received from the Natural Sciences and Engineering Research Council of Canada. Programming help for several of the experiments was provided by Peter Ashwood-Smith. Finally, we thank John Cleary of the University of Calgary for supplying results for the Cleary-Witten compression method applied to our sample files.

REFERENCES

1. J.G. Cleary and I.H. Witten, Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* **COM-32**(4) (1984).
2. G.V. Cormack, Data compression for a data base system. *Communications of the ACM* **28**(12) (1985).
3. G.V. Cormack and R.N. Horspool, Algorithms for adaptive Huffman codes. *Information Processing Letters* **18**(3) (1984).
4. R. Gallager, Variations on a theme by Huffman. *IEEE Transactions on Information Theory* **IT-24**(6) (1978).
5. M. Guazzo, A general minimum-redundancy source-coding algorithm. *IEEE Transactions on Information Theory* **IT-26**(1) (1980).
6. R.N. Horspool and G.V. Cormack, Dynamic Markov modelling – a prediction technique. *Proceedings of 19th Hawaii International Conference on the System Sciences*, Honolulu (1986).
7. D.A. Huffman, A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* **40** (1952).
8. C.L. McMaster, On-line manual entry and source code for "compact" and "uncompact" commands. Berkeley UNIX 4.2bsd system documentation (1985).
9. J. Rissanen and G.G. Langdon Jr., Arithmetic coding. *IBM Journal of Research and Development* **23**(2) (1979).
10. D.G. Severance, A practitioner's guide to data base compression. *Information Systems* **8**(1) (1983).
11. K.S. Trivedi, *Probability and statistics with reliability, queuing and computer science applications.* Prentice-Hall, Englewood Cliffs, N.J. (1982).
12. T.A. Welch, A technique for high-performance data compression. *IEEE Computer* **17**(6) (1984).
13. J. Ziv and Lempel, A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* **IT-23**(3) (1977).
14. J. Ziv and Lempel, Compression of individual sequences via variable-rate encoding. *IEEE Transactions on Information Theory* **IT-24**(5) (1978).

APPENDIX

The algorithms for cloning states, for implementing Guazzo encoding and decoding, and for constructing a braid as the initial graph model appear below. These algorithms are expressed in Pascal, but we have used some additional operators for bit-manipulation where desirable. These operators are **'shl'** and **'shr'** for left and right logical shifts, and '&' and '|' for bitwise logical **'and'** and **'or'** operations. The identifier N represents the number of significant bits that the algorithms use for range calculations. When implemented on a computer with 32-bit integers and with double-length integer multiplication and division instructions N can be chosen to be 31.

FIGURE A.1. The Cloning Algorithm

```
{ NEXT_STATE[S,D]  =  state reached from S after transition on digit D
  TRANS_CNT[S,D]   =  number of observations of input D when in state S
  STATE            =  number of current state
  LAST_STATE       =  largest state number used so far
  MIN_CNT1         =  minimum number of transitions from the current state
                      to state S before S is eligible for cloning
  MIN_CNT2         =  minimum number of visits to a state S from all
                      predecessors of S other than the current state be-
                      fore S is eligible for cloning.  }
while not eof do
  begin
    read( B );                    { read one input bit }
    TRANS_CNT[STATE,B]  :=  TRANS_CNT[STATE,B] + 1;
    NXT                 :=  NEXT_STATE[STATE,B];
    NXT_CNT             :=  TRANS_CNT[NXT,0] + TRANS_CNT[NXT,1];
    if  (TRANS_CNT[STATE,B] > MIN_CNT1)  and
          ((NXT_CNT - TRANS_CNT[STATE,B]) > MIN_CNT2) then
      begin
        LAST_STATE  :=  LAST_STATE + 1;
        NEW         :=  LAST_STATE;   { Obtain a new state number }
        NEXT_STATE[STATE,B]  :=  NEW;
        RATIO  :=  TRANS_CNT[STATE,B] / NXT_CNT;
        for  B := 0  to  1  do
          begin
            NEXT_STATE[NEW,B]  :=  NEXT_STATE[NXT,B];
            TRANS_CNT[NEW,B]   :=  RATIO * TRANS_CNT[NXT,B];
            TRANS_CNT[NXT,B]   :=  TRANS_CNT[NXT,B] - TRANS_CNT[NEW,B]
          end;
        NXT  :=  NEW
      end;
    STATE  :=  NXT
  end;
```

FIGURE A.2. The Guazzo Encoding Algorithm

```
{  MEANINGS OF VARIABLES:
   The binary message interval is LB to HB, inclusive.  MP is the dividing
   point for the interval partion.  LB, HB and MP are all scaled by 2**N.
   P0 and P1 are integers that give the relative probabilities for the
   next message bit being 0 or 1.  }
MSBIT  :=  1 shl (N-1);   MSMASK  :=  (1 shl N) - 1;
LB  :=  0;                HB  :=  MSMASK;
repeat
   {  At this point, knowledge of the Markov model or some other external
      source of information is used to estimate the relative probabilities
      of the next source digit being 0 or 1.  These (unnormalized) proba-
      bilities are assigned to P0 and P1 in the statements below.  }
   P0  :=  ... ;   P1  :=  ... ;
   {  Calculate the range partition )
   MP  :=  (P1 * LB + P0 * HB + P0 + P1 - 1) div (P0 + P1);
   if  MP = LB  then  MP  :=  MP + 1;
   MP  :=  MP | 1;     { force rightmost bit to 1 }
   {  Assertion:    LB  <  MP  ≤  HB   }
   if  eof  then  goto 999;
   read( B );                  { read one bit }
   if  B = 1  then
      LB  :=  MP                  { pick upper part of range }
   else
      HB  :=  MP - 1;             { pick lower part of range }
   while  (LB & MSBIT) = (HB & MSBIT)  do
      begin
         write( LB shr (N-1) );                  { output one bit }
         LB  :=  (LB shl 1) & MSMASK;            { remove the bit }
         HB  :=  (HB shl 1) & MSMASK + 1;
      end;
until  false;

999:                { Output all but rightmost bit in MP }
while  MP <> MSBIT  do
   begin
      write( MP shr (N-1) );                 { output one bit }
      MP  :=  (MP shl 1) & MSMASK            { remove the bit }
   end
```

FIGURE A.3. The Guazzo Decoding Algorithm

```
{ MEANINGS OF VARIABLES:
   LB, HB, MP, P0 and P1 have the same meanings as in Figure 5.
   IN_MSG holds a sequence of encoded message bits; the sequence begins at
   the high-order end.  LAST_BIT holds a single bit in the same position as
   the last significant bit of the sequence in IN_MSG.  }
MSBIT  :=  1 shl (N-1);    MSMASK  :=  (1 shl N) - 1;
LB  :=  0;                 HB  :=  MSMASK;
IN_MSG  :=  0;             LAST_BIT  :=  1 shl N;
repeat
   { At this point, we estimate the relative probabilities for the next
     decoded bit to be zero or one.  This estimation process uses exactly
     the same information as was available to the encoder at the same
     point in the original source message.  }
   P0  :=  ... ;   P1  :=  ... ;
   { Calculate the range partition }
   MP  :=  (P1 * LB  +  P0 * HB  +  P0  +  P1  -  1) div (P0 + P1);
   if  MP = LB  then  MP  :=  MP + 1;
   MP  :=  MP | 1;
   { Assertion:   LB  <  MP  ≤  HB   }
   repeat
      if  (IN_MSG  |  (LAST_BIT - 1) )  <  MP  then
         begin     B  :=  0;   HB  :=  MP - 1     end
      else  if  IN_MSG  ≥  MP  then
         begin     B  :=  1;   LB  :=  MP  end
      else
         begin
            if  eof  then  goto  999;   { exit at end-of-file }
            read( B );                  { read one bit }
            LAST_BIT  :=  LAST_BIT shr 1;
            if  B <> 0  then  IN_MSG  :=  IN_MSG | LAST_BIT;
            B := - 1
         end
   until  B  ≥  0;
   write( B );      { output one bit }
   while  (LB & MSBIT)  =  (HB & MSBIT)  do
      begin
         LB  :=  (LB shl 1) & MSMASK;
         HB  :=  ((HB shl 1) & MSMASK)  |  1;
         IN_MSG  :=  (IN_MSG shl 1) & MSMASK;
         LAST_BIT  :=  LAST_BIT shl 1
      end
until  false;
999:     { exit here when finished decoding }
```

FIGURE A.4. The Braid Construction Algorithm

```
{ LAST_STATE, TRANS_CNT[S,D] and NEXT_STATE[S,D] have the same definitions
  as in Figure A.1; The following code initializes the arrays for S = zero
  up to S = NBITS*STRANDS - 1.   }

const
  NBITS = 8;          { Number of bits per byte }
  STRANDS = 256;      { 2 ** NBITS }

for I := 0 to NBITS-1 do
  for J := 0 to STRANDS-1 do
    begin
      STATE := I + NBITS*J;
      K := (I+1) mod NBITS;
      NEXT_STATE[STATE,0] := K + (( 2*J ) mod STRANDS) * NBITS;
      NEXT_STATE[STATE,1] := K + ((2*J+1) mod STRANDS) * NBITS;
      TRANS_CNT[STATE,0]  := 1;
      TRANS_CNT[STATE,1]  := 1
    end;

LAST_STATE := NBITS*STRANDS - 1;
```