Application of a Finite-State Model to Text Compression

JUKKA TEUHOLA AND TIMO RAITA

Department of Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland

The bit-oriented finite-state model applied in Dynamic Markov Compression (DMC [5]) is here generalized to a larger alphabet. The finite-state machine is built adaptively during compression, by applying two types of modifications to the machine structure: state cloning and shortcut creation. The machine size is kept tolerable by an escape transition mechanism. Similar to DMC, the new method is combined with arithmetic coding, based on the maintained transition frequencies. The experiments show that the new approach produces notably better compression gains for different sorts of texts in natural and formal languages. In some cases the results are better than for any compression technique found in the literature.

Received January 12, 1993, revised June 21, 1993

1. INTRODUCTION

Homogeneous sequences of independent symbols can be encoded optimally using either Huffman coding [7], if distinct codes are required for the symbols, or arithmetic coding [11, 12], if decodability is the only condition. The latter gets arbitrarily close to the information-theoretic lower bound (entropy) when the length of the sequence increases. The same holds for Huffman coding, if sufficiently large symbol groups are taken as coding units.

The independence assumption is hardly ever valid for textual data. The dependencies are strongest between neighbouring characters, but may extend even over 30 characters in a typical English text [6]. Non-contiguous syntactic dependencies may reach even longer, especially in formal languages, but such dependencies will not be considered here. A typical way of describing the dependencies between consecutive symbols is a Markov model. We can imagine that the symbols are generated one by one, and the distribution of the next symbol is determined ('predicted') by a few previous ones. 'A few' means the order of the model, which can be variable, but usually limited. Arithmetic coding is typically used as the final step in compression because, in addition to being optimal, it is well suited for adaptive application.

Various Markov-based compression methods have been proposed in the literature (for surveys, see e.g. [1, 2, 8, 13, 14]). Almost all of them are adaptive, i.e. they make only one pass over the text and gradually refine the model. The same model can be built during decoding, because at each step the model is based on the substring that has already been processed. One of the best predictive methods was presented by Cleary and Witten in [4]. They applied a variable-order model where each coding step starts with a large context (maximum-sized prediction block), which is then reduced, if the context does not predict the symbol to be encoded. When a successful context is found, its successor distribution is applied in the actual (arithmetic) coding. The reduction is called an 'escape' and estimating the escape probability is one of the critical points in the method. After each step, the prediction information related to the context is updated.

Cleary and Witten gather prediction blocks of variable lengths (plus the related distributions), which can be stored, e.g. in a trie. A quite different approach is to apply a *finite-state machine* (FSM) to implement the finite-context model. The machine consists of a set of *states* and symbol-labeled *transitions* from state to state with known (estimated) transition probabilities for each state. For a given state, all outgoing transition labels are distinct. A source sequence determines a unique path through the machine, so that each symbol determines the next transition. Arithmetic coding, based on the transition probabilities, complements the process. The machine can be made adaptive, so that at any step states and transitions may be added, in addition to updating the transition probabilities.

Finite-state models are potentially more powerful than finite-context models. The other advantage is speed—the machine is simple and processing a source sequence means just following the proper transitions. A very successful implementation, called Dynamic Markov Compression (DMC), was presented by Cormack and Horspool [5]. Their source alphabet was binary; character data was handled bitwise by using a suitable initial configuration for the machine. The problem with larger alphabets is that there should exist a path for *any* source sequence. Thus, all states should be equipped with transitions for all symbols, making the machine very space-consuming, and its building rather slow. We cite here Bell *et al.* [2]:

In principle, there is no reason why a symbol-oriented *version of the method should not be used. It is just that in practice, symbol-oriented models of this type tend to be rather greedy in their use of storage, unless a sophisticated data structure is used.*

This challenge has not been widely accepted in the literature. Revuz and Zipstein [10] present one application of a FSM specialized in finding and compressing words of a natural language text. We shall attack the problem in a way that is more faithful to the original DMC technique, without presumptions of the source type.

The rest of this article is organized as follows. The DMC method is described and discussed in Section 2. Section 3 then introduces the generalization to larger alphabets. The details of the actual coding are explained in Section 4. The experimental results, indicating notable improvement compared to DMC, are reported in Section 5, followed by a discussion. Due to the heuristic nature of all these kinds of methods, an analysis of the compression gain is extremely difficult and usually omitted in the literature—this paper is no exception.

2. THE BIT-ORIENTED TECHNIQUE (DMC)

As mentioned, DMC is tailored to binary data. Each state of the FSM may have any number of incoming transitions, but precisely two outgoing transitions, labelled 0 and 1. The compression starts with an initial model, which in its simplest form consists of only one state and two transitions (one for each symbol) leading to the state itself. Figure 1 shows another initial model, resulting in a sounder machine configuration (see below). Other more complex initial models may be used, e.g. for bitwise handling of symbols from a non-binary alphabet, see [5].

In the following we shall denote states by symbols $s_0, s_1, \dots, s_i, \dots$, and a transition from s_i to s_j by the expression (s_i, s_j). At most one transition can exist from s_i to s_j for any i, j. The transition label (0 or 1) is marked beside the transition, where necessary.

DMC uses two techniques to adapt the model to the actual data. First, frequency counts are maintained for transitions. This information is fed to the arithmetic coder when leaving a state along some transition. The second technique, called *cloning*, creates a copy (s'_k) of an existing state (s_k), on the basis of a transition (s_j, s_k), see Figure 2. Transition (s_j, s_k) is redirected to s'_k, and copies of the two outgoing transitions, (s_k, s_m) and (s_k, s_n), are created for s'_k. Part of the transition counts of (s_k, s_m) and (s_k, s_n) are given to (s'_k, s_m) and (s'_k, s_n), respectively,

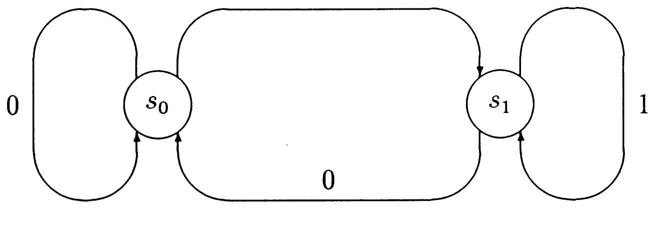

FIGURE 1. A sample initial model of a binary FSM.

in proportion to the counts of (s_j, s_k) and other transitions leading to s_k.

The logical effect of cloning is an increase of context. If several transitions lead to state s_k then, after reaching s_k, the process *forgets* where it came from, i.e. the specific context is lost. Since different predecessors usually imply different distributions of successors, it is reasonable to give a high-traffic transition its own set of successors, which in time is adapted to approach the more precise successor distribution of just that particular transition.

Two properties of the machine should be observed. First, for each state, all incoming transitions have identical labels (either 0 or 1). This condition holds for the initial model of Figure 1. Thus the label could as well be attached to the state itself. Second, the transition counts obey the so called Kirchhoff's Law, i.e. for each node, the sum of incoming counts equals the sum of outgoing counts. This property will be further discussed in the context of the new method.

The key question in DMC is, when to clone. A transition should have a sufficiently high count ($C1$), in order to deserve its own set of successors. However, if the transition is (almost) the only one actually used to enter the state in question, cloning does not pay off. So, another threshold value ($C2$) is required for the sum of other incoming transition counts. Cormack and Horspool [5] have experimentally found that the best results are obtained with as small threshold values as $C1 = C2 = 2$. The interpretation is that it is advantageous to extend the contexts very rapidly, at the cost of imprecision in successor distributions. Unfortunately, this also means that the model grows quickly: for an input of about 100 Kbytes, the graph size was about 150,000 nodes and the number of transitions twice as much. The simplest way to restrict the model size is to stop cloning at some point. Thereafter, the maintenance of frequency counts is the only way to adapt the model.

Although finite-state machines are in principle more powerful tools than variable-order Markov models, DMC is not. Bell and Moffat [3] showed that DMC can generate only *Finite Context Automata* (FCA), which are in fact equivalent to variable-order Markov models. This implies that DMC cannot be more powerful than the Markov-based methods, which has been confirmed also by the experiments. We are not going to step out of FCAs, either, but just device an effective generalization of DMC for larger source alphabets.

3. THE GENERALIZED MACHINE

From now on, the new compression method will be called GDMC (Generalized DMC). Just like DMC, also GDMC builds a FSM dynamically, starting from an initial model. The transitions are labelled by symbols from the general (e.g. ASCII) alphabet. A set of transitions sharing the same source state is called a *fanset*. If no confusion arises, also the set of related symbols (labels) is called a fanset. In what follows, Freq(s_i, s_j)

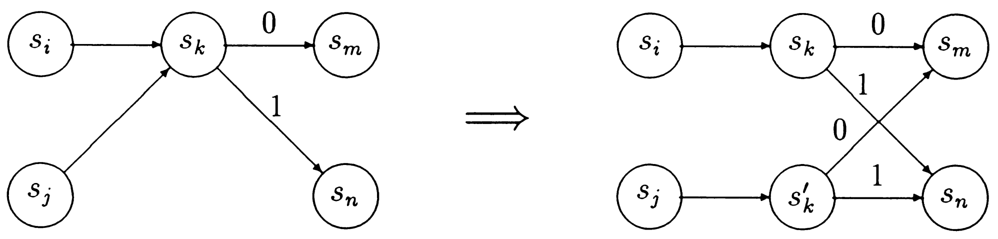

FIGURE 2. An example of cloning a state in DMC.

denotes the frequency of transition (s_i, s_j), and Fsum(s) the sum of frequencies of the transitions in the fanset of s.

The problem incurred by a large alphabet is 2-fold. First, since any input string must be accepted by the machine, *all* symbols must be represented in each fanset, consuming a lot of memory. Second, the distribution of symbols in a fanset is difficult to estimate, because most of the symbols will never actually occur after a certain state. Yet, all transitions must be assigned a non-zero frequency, to ensure the correct functioning of arithmetic coding.

For solving the problem, we adopt the well-known *escape* mechanism, *cf.* [4]. Each state is followed by a few (possibly zero) *actual* symbol transitions, and an escape transition, representing the rest of the alphabet. This reduces the number of transitions considerably, but estimating the escape frequency must still be solved. A simple solution is to maintain it in the same way as the actual transition frequencies (as if the escape were a symbol in the alphabet).

The initial model consists of the initial state s_0, plus transitions and states for all symbols of the alphabet, see Figure 3. Escapes lead back to s_0, but s_0 needs no escape transition. All transitions are assigned the frequency 1. An even simpler model would consist of s_0 and transitions for each symbol, leading from s_0 to itself. However, this model contradicts the following two properties that we want to maintain:

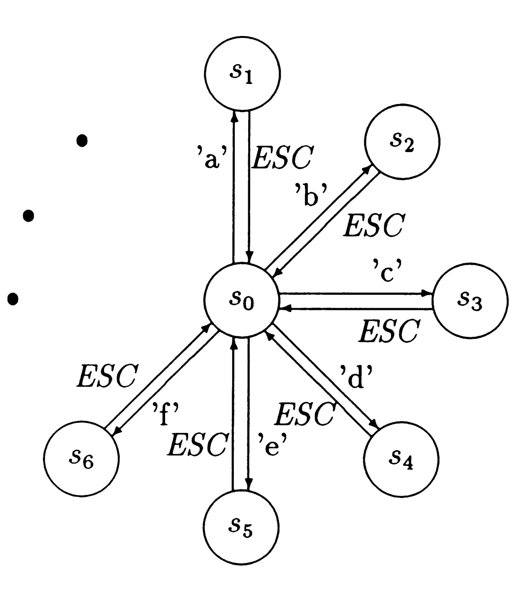

FIGURE 3. The initial configuration of the FSM.

- Only one transition may exist from s_i to s_j, for any i, j.
- Each state is entered with identically labelled transitions and/or escapes.

Actually, the second property implies the first one, because all transitions leaving a state have different labels.

Two structure-modifying operations are applied during the dynamic building of the finite-state machine:

1. *Cloning* a state s_k (see Figure 4a), based on an incoming transition (s_i, s_k), is even simpler than in DMC: a new state s'_k is created, and (s_i, s_k) is redirected to it. An escape transition is set from s'_k to s_k. Note that the new machine is functionally equivalent to the old one. The advantages of cloning will be substantiated only after applications of the second type of operation.
2. A *shortcut* is a new transition ((s'_k, s_m) in Figure 4b) that bypasses a combination of an escape (s'_k, s_k) and an actual transition (s_k, s_m), representing the input symbol ('b'). When 'b' occurs the next time in state s'_k, the shortcut is taken, instead of the escape. The set of symbols represented by the escape is thus reduced. The combination of cloning and shortcut creation has the effect of increasing the context—the encoding of 'b' now depends on the path traversed (leading to either s_k or s'_k). When creating a shortcut, some frequencies should be updated, but the best policy is not obvious. The discussion on this question is deferred to the next section.

Now we should decide, when to clone and when to make a shortcut. In cloning we adopt only one of the two criteria used in DMC: the target state s_j of a non-escape transition (s_i, s_j) is cloned, if the difference Fsum(s_j) − Freq(s_i, s_j) exceeds the threshold value CC ('Cloning Constant'). In our experiments, the first cloning condition in DMC, i.e. a threshold value for Freq(s_i, s_j), was found to impair the compression result! The interpretation is that the optimal threshold value is one, setting no restriction to cloning. The tradeoff between compression gain and main memory demand (during processing) can be sufficiently tuned by the value of CC only.

A shortcut is created always when an escape is followed by a non-escape transition. This is useful, because as soon as we get one evidence of a successor symbol for a state, we should record it—otherwise the informa-

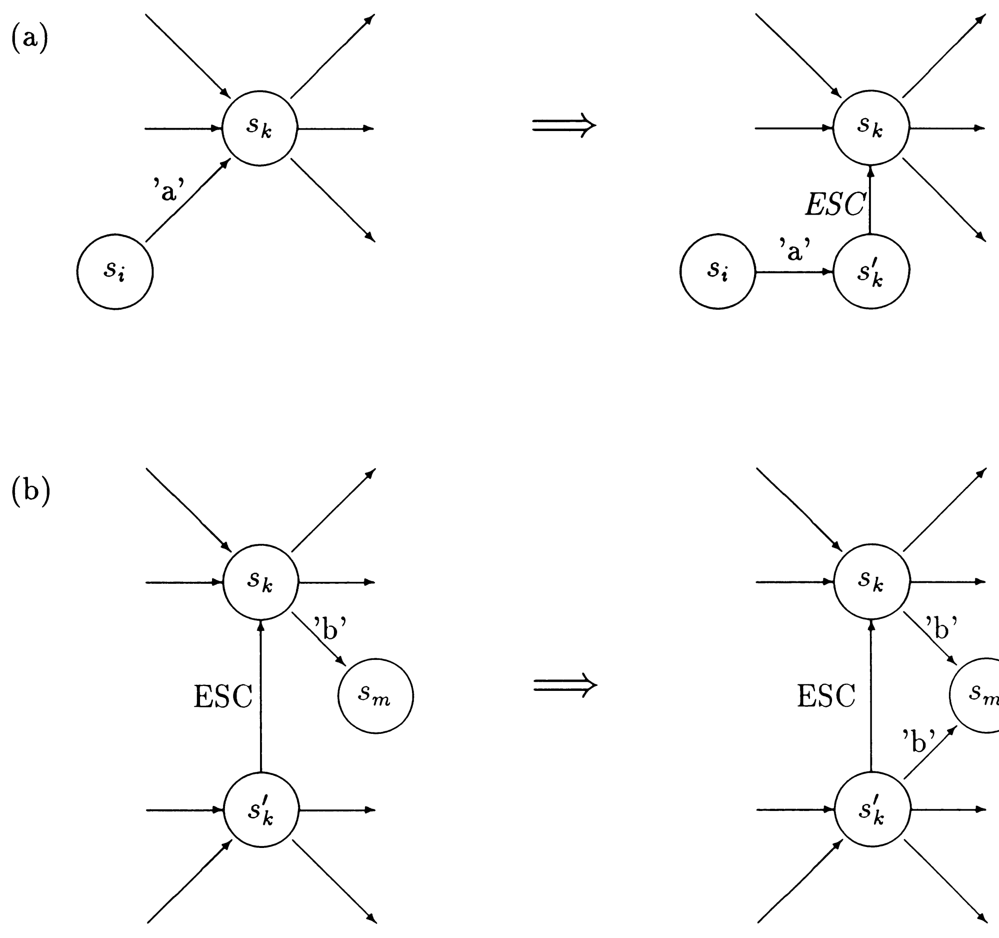

FIGURE 4. Extension operations on the machine: (a) cloning and (b) creating a shortcut.

tion gets lost. Often we have to take several escape transitions in succession, before the current symbol is found among the actual transitions. For the transition chain (s_p, s_{p+1}), ..., (s_{q-1}, s_q), (s_q, s_r) in Figure 5, *only* the last escape is bypassed by the new shortcut (s_{q-1}, s_r). This rule has been set for practical reasons. The other two alternatives would have been the following:

- Create the direct shortcut (s_p, s_r). This seems the most obvious solution, but empirical observations do not support it. Moreover, it would make the application of the *exclusion principle* (*cf.* next section) more difficult.
- Create all shortcuts (s_p, s_r), ..., (s_{q-1}, s_r) on the escape path. This would, in addition to waste memory, be too hasty: Traversing a long escape chain is an indication of an unfrequent symbol (in the current context); otherwise shortcuts would have been created before. Thus it is unwise to create transitions that will probably never be used. Moreover, we have no ground to estimate their frequencies.

Of course, when traversing the same path anew, the next shortcut (s_{q-2}, s_r) is created, etc. Note also that traversing a plain escape chain (with otherwise empty fansets) costs nothing, in view of compression.

Cloning and shortcut creation are the *only* operations on the structure of the machine. It is difficult to imagine,

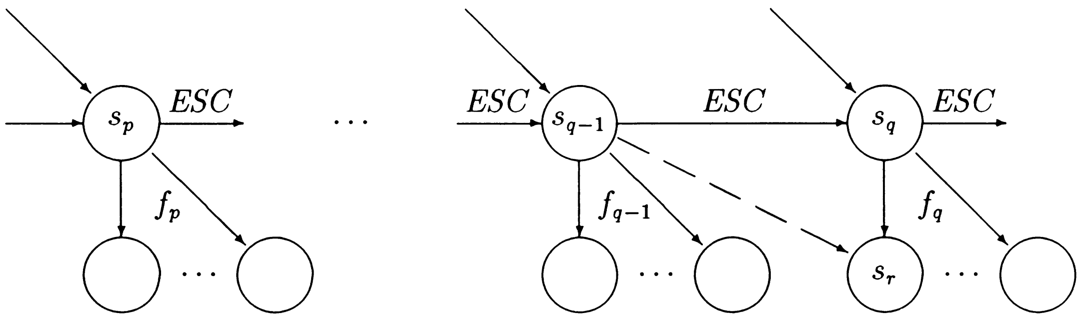

FIGURE 5. An escape chain.

what the graph will look like after a large number of operations, but the following interpretation can be given. Taking an escape transition means reducing the context and enlarging the fanset. This results from the shortcut policy: For two states, s_i and s'_i, with fansets f_i and f'_i, respectively, and an escape from s'_i to s_i, it holds that $f'_i \subseteq f_i$. We shall take advantage of this property in the coding method. Several escapes can be followed in a sequence, possibly up to the initial state s_0, which represents the null context, and has the whole alphabet as its fanset. There exists an escape path to s_0 from *each* state of the machine. The length of this path represents the (minimal) *order* of the Markov model at the current state. The order of the initial state s_0 is zero, i.e. it carries no context.

Compared with the original DMC, we can say that GDMC is *lazy* in creating new transitions. DMC immediately connects the clone to all the targets of the original state, whereas GDMC does it one by one, according to the need. This observation is also a sufficient proof for the fact that, like DMC, also GDMC generates only Finite-Context Automata [3].

4. THE CODING PROCESS

The compression of a source string is accomplished by starting from s_0 and following the transitions one by one, according to the source symbols. At each state, the encoding is based on the frequencies of possible symbols in the fanset. For the purposes of arithmetic coding, we must define some (e.g. alphabetic) order within the symbols, say, $\{a_1, \ldots, a_k, \ldots, a_m\}$, of which a_k is the current source symbol and a_m is the escape. Three values are transmitted to the arithmetic encoder: $\Sigma_{i=1}^{k-1} \text{Freq}(a_i)$, $\Sigma_{i=1}^{k} \text{Freq}(a_i)$ and $\Sigma_{i=1}^{m} \text{Freq}(a_i)$. The details of arithmetic coding are skipped here—a lucid implementation is given in [15].

In some cases we can exclude some transitions from a fanset. If, after visiting state s_i, an escape is made to s_j, we know that the current symbol is not within the fanset f_i of s_i. So we can exclude the related symbols from the fanset f_j of s_j. (Here f_i and f_j represent *symbols* in the fanset.) This so called *exclusion principle* (see [2]) is based on the above mentioned property: $f_i \subseteq f_j$. The general case is illustrated in Figure 5, where the states $s_p, \ldots, s_q$ have fansets $f_p, \ldots, f_q$, respectively, and $f_p \subseteq f_{p+1} \subseteq \cdots \subseteq f_{q-1} \subseteq f_q$. After traversing the whole chain, the *effective fanset* of s_q is obtained by excluding $f_p, \ldots, f_{q-1}$ from f_q, which simply means the difference of sets f_q and f_{q-1}.

It is important that extensions to the machine are based on the data already encoded, to enable the decoding process to do the same modifications. Therefore, cloning a state is performed *after* having arrived at it. Similarly, a shortcut of two transitions is created *after* taking them both.

In addition to the machine structure, the maintenance of frequencies is equally important for successful coding. After taking a transition, its frequency count is increased by one. When cloning a state s_k (into s'_k), based on transition (s_i, s_k) (Figure 4a), the escape from s'_k to s_k gets the frequency $\text{Freq}(s_i, s_k)$, because both represent the same flow.

When creating a shortcut, the update of frequencies is not so trivial. The frequency of the shortcut transition itself is easy to estimate. Referring to Figure 4(b), the following formula is applied

$$\text{Freq}(s'_k, s_m) = \frac{\text{Freq}(s'_k, s_k) \cdot \text{Freq}(s_k, s_m)}{\text{Effsum}(s_k \mid s'_k)}$$

where $\text{Effsum}(s_k \mid s'_k)$ is the sum of transition counts in the effective fanset of s_k (i.e. in the difference of fansets of s_k and s'_k). In our implementation, the frequencies are maintained as *real* numbers, not integers, i.e. the above ratio is not truncated/rounded. This proved to produce better results.

The ratio $\text{Freq}(s_k, s_m)/\text{Effsum}(s_k \mid s'_k)$ represents the proportion of the escape frequency to be assigned to the shortcut (s'_k, s_m), because the path (s'_k, s_k, s_m) will never be traversed anymore. Accordingly, the escape frequency is reduced by $\text{Freq}(s'_k, s_m)$. Kirchhoff's law would imply that also $\text{Freq}(s_k, s_m)$ should be reduced by the same amount. However, it can be shown that a long sequence of machine steps may then create *negative* frequencies. This non-obvious phenomenon was detected in the experiments, and results from an unfortunate combination of machine updates. The problem would not occur, if $\text{Effsum}(s_k \mid s'_k)$ were replaced by $\text{Fsum}(s_k)$, but this is unacceptable in terms of compression performance. The illegal frequencies could of course be captured by additional tests and handled in some special manner. However we obtained the best results by not reducing $\text{Freq}(s_k, s_m)$ at all. It seems that the reduction somehow tampers the successor distribution of other transitions leading to s_k. A somewhat annoying consequence from the selected rule is that we *abandon Kirchhoff's law* (for state s_k in this case). It must be emphasized that, though being a natural property, Kirchhoff's law is no necessity for coding—all we need is a realistic distribution of successors for each state.

Very little needs to be said about decoding. From the machine building point of view, the procedure is identical. Into arithmetic decoding we have to transmit the whole distribution (cumulative frequencies) of the effective fanset, from which the decoder picks up the correct one. Thereafter, cloning and shortcut creation can take place.

A skeleton of the encoding program in Pascal is given in the Appendix.

5. EXPERIMENTS

In testing GDMC, we used files from the Calgary Corpus, which is a widely used set of benchmark data for compression, see [1]. First we tried to find the optimal value for the threshold parameter *CC*, which determines the rate of cloning. A sample set of results is given in Table 1 for 'paper 1' (53 161 bytes) of the Corpus. The best compression results are achieved with very small values of *CC*. The conclusion is the same as in [5]: the increase of context is more important than the precision of the successor distribution. A straightforward solution would be to clone *always*. However, as can be seen from Table 1, with less eager cloning, memory consumption can be dropped to half, without essentially degrading the compression results. Note that frequencies can really be non-integers, due to the calculations made in creating a shortcut. Moreover, even $CC = 0.0$ does not always cause cloning, because Kirchhoff's Law does not hold.

In the second test all the files of the Corpus were compressed, using the value $CC = 0.25$, which is close to optimal in all cases. In Table 2, the compression results of GDMC are compared with those of DMC [5] and PPMC ('Prediction of Partial Match', presented by Cleary and Witten [4] and tuned by Moffat [9]). The latter is generally regarded as the best practical text compression method. The results of DMC and PPMC originate from [1]. We can see that, especially for longer texts in natural language, GDMC is clearly the best of the three.

It must be admitted that we did not set any constraints on the main memory consumption of GDMC during the coding process. For the largest test files about 10–20 Mbytes were used (with the optimal *CC* value).

TABLE 1. The effects of varying *CC*

CC	*Result (bits/character)*	*No. of states*	*No. of transitions*
0.0	2.61	52223	98595
0.5	2.58	36447	73821
1.0	2.59	29483	62964
2.0	2.61	22420	51101
5.0	2.65	15403	38569
10.0	2.71	11774	31437
20.0	2.79	8943	25644

TABLE 2. A comparison of results from compressing files of the Calgary Corpus

File	*Size*	*DMC*	*PPMC*	*GDMC*
bib	111261	2.28	2.11	**2.05**
book1	768771	2.51	2.48	**2.32**
book2	610856	2.25	2.26	**2.02**
geo	102400	**4.77**	4.78	5.16
news	377109	2.89	2.65	**2.60**
obj1	21504	4.56	**3.76**	4.40
obj2	246814	3.06	**2.69**	2.82
paper1	53161	2.90	**2.48**	2.58
paper2	82199	2.68	**2.45**	**2.45**
pic	513216	0.94	1.09	**0.80**
progc	39611	2.98	**2.49**	2.67
progl	71646	2.17	1.90	**1.83**
progp	49379	2.22	**1.84**	1.90
trans	93695	2.11	1.77	**1.73**

The speed of compression/decompression varied a lot, depending on the source type. For large text files, the compression speed was about 9 Kbytes/s and the decompression speed about 7 Kbytes/s, in a Sun Sparcserver 690M (25 MIPS), excluding the I/O time. The programs were written in Pascal, without any special optimization.

6. DISCUSSION AND FURTHER WORK

The presented GDMC method produces remarkably good compression results for most textual data types. However, we do not claim it to be 'optimal' in any sense, even within the framework of finite-state models. There are many heuristic features in the method, which could perhaps be done differently. The most delicate question is related to the estimation of transition frequencies when modifying the machine with the basic transformations (clone and shortcut). A thing that still bothers us is the abandonment of Kirchhoff's law. This makes the machine theoretically unclean, but we did not find a satisfactory solution without sacrificing the compression performance.

It is probably possible to invent different ways of extending the machine, to better conform with the properties of the source. The transformations should enhance the differentiation of contexts, without consuming too much memory.

The execution speed might be improved in several ways at the cost of compression gain. Most of the processing time in the current version is consumed in

- searching for the correct transition in the list of successors;
- determining the difference of fansets (applying the exclusion principle);
- calculating the frequencies and their sums;
- arithmetic coding.

By attacking these points we aim to develop a faster practical variant of the presented method.

REFERENCES

[1] T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression*. Prentice Hall, Englewood Cliffs, NJ (1990).

[2] T. C. Bell, I. H. Witten and J. G. Cleary, Modeling for text compression. *ACM Computing Surveys*, **21**, pp. 557–591 (1989).

[3] T. C. Bell and A. Moffat, A note on the DMC data compression scheme, *The Computer Journal*, **32**, pp. 16–20 (1989).

[4] J. G. Cleary and I. H. Witten, Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, **COM-32**, pp. 396–402 (1984).

[5] G. V. Cormack and R. N. S. Horspool, Data compression using dynamic Markov modelling. *The Computer Journal*, **30**, pp. 541–550 (1987).

[6] S. Guiasu, *Information Theory with Applications*. McGraw-Hill, London (1977).

[7] D. A. Huffman, A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, **40**, pp. 1098–1101 (1952).

[8] D. A. Lelewer and D. S. Hirschberg, Data compression. *ACM Computing Surveys*, **19**, pp. 261–296 (1988).

[9] A. Moffat, *A Note on the PPM Data Compression Algorithm.* Research Report 88/7, Department of Computer Science, University of Melbourne, Victoria, Australia (1988).

[10] D. Revuz and M. Zipstein, *A Text Compression Algorithm for Natural Languages.* Preprint, Université de Marne la Vallée, Noisy le Grand, France (1992). (To appear in *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching.*)

[11] J. J. Rissanen, Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research & Development*, **20**, pp. 198–203 (1976).

[12] J. J. Rissanen and G. G. Langdon, Arithmetic coding. *IBM Journal of Research & Development*, **23**, pp. 149–162 (1979).

[13] J. A. Storer, *Data Compression: Methods and Theory.* Computer Science Press, Rockville, MD (1988).

[14] R. N. Williams, *Adaptive Data Compression.* Kluwer Academic, Norwell, MA (1991).

[15] I. H. Witten, R. M. Neal and J. G. Cleary, Arithmetic coding for data compression. *Communications of the ACM*, **30**, pp. 520–540 (1987).

APPENDIX

A skeleton of the GDMC encoding program in Pascal is given below, together with type definitions. Since all transitions leading to a given state are related to the same symbol, we have attached the symbol to the state record, thus saving space. The initial state is an exception: it is entered only via an escape. Escape transitions are also stored directly in state records, because there is only one escape per state.

Some formulae in the encoding program are slightly different from those in the text above, adding/subtracting 1.0 is due to the fact that at the current state the in-count has been increased, whereas the out-count has not.

Details of arithmetic coding are skipped—three procedure calls are related to it, i.e. initialization, encoding a transition and finishing the coding. The well-known ending problem is handled by an additional eof-symbol in the fanset of the initial state.

Decoding differs from encoding in that the next transition is not determined by the input symbol, but by arithmetic decoding, which now needs the distribution of the whole effective fanset. Otherwise, from the model construction point of view, decoding is analogous, and therefore skipped.

```
type
  symbol_type = char;
  symbol_file = packed file of symbol_type;
  state_ptr = ^ state_rec;
  trans_ptr = ^ trans_rec;
  state_rec = record
                  symbol: symbol_type;
                  fanset: trans_ptr;
                  esc_target: state_ptr;
                  esc_freq: real
                end;
trans_rec = record
                freq: real;
                target: state_ptr;
                next: trans_ptr
              end;

procedure search(symb: symbol_type;         { symbol to be encoded }
                 prev_state: state_ptr;     { previous state }
                 state: state_ptr;          { current state }
                 var trans: trans_ptr;      { matching transition }
                 var cum_freq: real;        { cumul. frequency of symb }
                 var eff_sum: real);        { sum of included frequencies }
{
  Scans the fanset of the current state and searches for symbol symb.
  The transition, if found, is returned in parameter trans.
  Two frequency sums are counted from the effective fanset of the current state (symbols in the fanset of prev_state excluded):
      1) the cumulative frequency of symbols < symb
      2) sum of all frequencies in the effective fanset
}
begin ··· end;

procedure insert_trans(shortcut: trans_ptr; state: state_ptr);
{
  Inserts the shortcut transition in the fanset of the given state.
  The position is determined e.g. by the alphabetic order of symbols—any well defined order will do.
```

```
 }
 begin ··· end;

 procedure encode(source: symbol_file; CC: real);
 var
   symb: symbol_type;
   state, prev_state, clone: state_ptr;
   from, trans, shortcut: trans_ptr;
   cum_freq, eff_sum: real;
 begin
   initialize_arithmetic_encoding();
   create_initial_model(state);          { The start state }
   from := nil;                          { The transition last taken }
   while not eof(source) do
   begin
     read(source, symb);
     prev_state := nil;                  { Previous state in the esc-chain }
     search(symb, prev_state, state, trans, cum_freq, eff_sum);
     if from < > nil                     { Not at the start? }
     then if (eff_sum + 1.0—from^.freq) > = CC
       then begin                        { The current state must be cloned }
         new(clone);
         clone^.symbol := state^.symbol; { Represents the same symbol }
         from^.target := clone;          { New target for the from-transition }
         clone^.fanset := nil;           {No actual successor symbols yet }
         clone^.esc_target := state;     { Escape transition to the old state }
         clone^.esc_freq := from^.freq   { The old frequency is copied }
       end;
     while trans = nil do
     begin                               { Loop for escapes }
       arith_encode((eff_sum—state^.esc_freq), 1.0, eff_sum);
       state^.esc_freq := state^.esc_freq + 1.0;
       prev_state := state;
       state := state^.esc_target;       { Proceed to the escape target }
       search(symb, prev_state, state, trans, cum_freq, eff_sum)
     end;
     arith_encode(cum_freq, (cum_freq + trans^.freq), eff_sum);
     if prev_state = nil
     then begin                          { No escapes, thus no shortcuts }
       trans^.freq := trans^.freq + 1.0;
       state := trans^.target;           { Proceed to the next state }
       from := trans                     { Remember where we came from; needed in cloning }
     end
     else begin                          { One or more escapes were taken }
       new(shortcut);                    { Create a shortcut to trans^.target }
       shortcut^.target := trans^.target;
       shortcut^.freq := (p^.esc_freq – 1.0)*trans^.freq / eff_sum + 1.0;
       prev_state^.esc_freq := prev_state^.esc_freq – shortcut^.freq;
       insert_trans(shortcut, state);    { The shortcut starts from the current state }
       state := shortcut^.target;        { Proceed to the next state }
       from := shortcut                  { Remember where we came from }
     end
   end;
   finish_arithmetic_encoding(state)     { Encode the fictitious eof-symbol }
 end;
```