Context model automata for text compression

(published in *The ComputerJournal*, 41 (7), 474-485, 1998)

Pasi Fränti

Department of Computer Science, University of Joensuu,
Box 111, FIN-80101 Joensuu, FINLAND

Timo Hatakka

Department of Information Technology, Lappeenranta University of Technology
Box 20, FIN-53851 Lappeenranta, FINLAND

Abstract

Finite-state automata offer an alternative approach to implement context models where the states in the automata cannot in general be assigned by a single context. Despite the potential of this approach, it makes the design of the modelling more problematic because the exact behaviour of the model is not known. Here we propose a simple formalism – *context model automata* (CMA) – that gives an exact interpretation for the minimum context belonging to each state in the automaton. The formalism is general enough to simulate context models such as PPM and GDMC. Using the CMA formalism as our tool, we study the behaviour of the above two context models.

Key words: Text compression, context modelling, PPM models, Markov models

Number of words:	8100 (including figures and tables)
Number of lines:	1000 (including figures and tables)
Number of figures:	14
Number of tables:	3

Correspondence:
Pasi Fränti
Department of Computer Science, University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, FINLAND
Phone: +358-013-251-3103
Fax: +358-013-251-3290
Email: franti@cs.joensuu.fi

1. Introduction

Text compression is typically performed by *statistical* or *dictionary-based* methods [1, 2]. The input file is processed by a single pass starting with an initial model or dictionary. The method then adapts to the source data during the compression. Only the data that is already compressed (thus also known by the decoder) can be used in the model. In this way the decompression can be done synchronously.

Dictionary-based methods decompose the source data into substrings which are compressed separately either by a pointer to an earlier occurrence of the string in the source data (*LZ77 variants*) [3], or by a pointer to a dynamically constructed dictionary (*LZ78 variants*) [4]. The pointers are stored using a fixed length coding, or an entropy coding method such as *Huffman coding* [5].

Better compression performance is achieved by statistical compression at the cost of lower throughput and higher memory consumption. In symbol-based context models each symbol is predicted by a *context*, which is a combination of a few preceding symbols. A different statistical model is constructed for each context on the basis of the previously coded data. Each model consists of a table of successor symbols and their frequencies. The actual coding can be performed by arithmetic coding using the implementation given by Witten *et al.* [6].

Variable-order contexts have been widely used in PPM models (*prediction by partial match*), originally proposed in [7]. The highest order context is first applied. If the symbol is not predicted by the context, an *escape symbol* is coded and another lower level context is used. If no other context can be applied, the symbol is eventually coded using a zero-order model. The size of the highest order context is kept relatively small (three in [1]) to reduce the memory requirements and the learning cost.

Data compression can conceptually be divided into two separate components: modelling and coding. Modelling is the key question in the design of a compression algorithm whereas the coding can be optimally done using arithmetic coding. From this point of view, it is easy to see that the statistical models are more general than the dictionary-based models since they explicitly separate the two components from each other. In fact, commonly used dictionary-based methods can be simulated by symbol-wise statistical models. For example, Langdon has given a statistical counterpart for LZ78 in [8].

A third approach to text compression is the use of finite state automata. An automaton consists of a set of states and transitions between the states. Each transition corresponds to a single coding event: coding of a symbol or part of it. All states must include outgoing transitions for every possible coding incident that might appear. An implementation known as *Dynamic Markov Compression* (DMC) uses a binary alphabet in the automaton [9]. Symbols of larger alphabets must be handled bitwise. It has been shown that DMC actually implements a finite-context model [10].

Generalized Dynamic Markov Compression (GDMC), proposed by Teuhola and Raita [11], generalizes the idea of DMC to multisymbol alphabets. The states in the automaton correspond to the contexts and the transitions to the selection of the next state. *Escape strategy* is taken to reduce the number of outgoing transitions in each state. GDMC implements a context model with potentially stronger capabilities than the PPM model but its exact behaviour is not understood very well.

We propose a simple formalism, *context model automata* (CMA), for modelling the contexts in a finite-state automaton. It gives an exact interpretation for the minimum context belonging to each state in the automaton. We will show that both PPM and GDMC follow the CMA formalism. We thus obtain a context interpretation for the states of a GDMC automaton. CMA can also be used as a practical tool for analyzing the behaviour of the GDMC and PPM models, and to design new ones.

A more theoretical approach has been taken by Bunton [12] for studying the contexts in a *full-table model* where each state includes a complete set of transitions. The contexts are defined as a set of strings described by regular expressions. This gives an exact but not necessarily an easy interpretation for the contexts of the states. Our approach does not take into account the partial contexts included in the states but it gives a simpler interpretation for the contexts, which can be easily utilized in practical applications. We focus here on models that uses the *escape strategy*, in which only a subset of the entire alphabet may appear in each state. This is more faithful for the traditional context models such as PPM.

Using CMA formalism as our tool, we study the behaviour of the GDMC and PPM context models. CMA gives useful information about the structure of the automaton by giving context interpretation for the states. CMA, however, models only an existing automaton and it does not define the way the automaton is constructed. The dynamic construction of CMA is therefore the key question when designing a new automaton-based compression scheme. In this study, we limit the discussion to the existing models underlying in PPM and GDMC.

In PPM, the creation of new states depends on the context length, and in GDMC, on the statistics of the state. PPM is more greedy in the model expansion and it adapts quicker to the source data. The model growth is controlled by limiting the context length by a predefined maximum. GDMC, on the other hand, has no upper limit for the context length but it uses a frequency based heuristic to control the model growth. Due to its lazier expansion, the resulting contexts are not necessarily unique. Although CMA defines a minimum context length for each state, the automaton has longer partial contexts.

It is noted that the CMA model itself says nothing about the compression performance. It is therefore equally important to consider also the statistical model integrated with the context model. For example, both PPM and GDMC use greedy state selection (the first predictive state is always applied). Better results could be obtained using information theoretic state selection, based on the expected compression performance of the states. However, it is not clear how the expected compression performance should be measured. In the rest of the paper, we will focus on context modelling only, and settle for the existing PPM and GDMC methods in the statistical modelling.

2. Context model automata

The main components of a CMA-based compression algorithm are:

- Context model automaton (CMA)
- Statistical model
- Entropy coder

The object of CMA is to model contexts only. For each input symbol the current state of the automaton determines the context predicting the symbol to be coded. The statistical model then calculates the probability distribution using the symbol frequencies included in the chosen context. The actual coding is performed by an entropy coder (usually arithmetic coding) on the basis of the determined probability distribution. After the coding, the statistics are updated.

CMA implements the context model as follows. Each state in the automaton corresponds to a single context having its own statistics. The states include also transitions for all possible coding events that might appear. A transition leads to the state to be used next. The automaton is deterministic: its behaviour depends only on the input (source data and parameters), i.e., the automaton works always in the same way for the same input.

The states do not necessarily include transitions for all possible symbols and the escape strategy is applied instead. Only transitions for already seen symbols can appear in the states. The rest are coded by an *escape*-transition (*ESC*) leading to another state. If the symbol does not exist in the following state, an escape path is followed until a proper state is reached and the symbol can be coded. The escape strategy eliminates also the zero-frequency problem [1]. An example of a CMA and its application for coding a sample source string are illustrated in Figure 1.

The main phases of a CMA-based compression algorithm are described in Figure 2. The input data is processed symbol by symbol and the automaton is constructed dynamically during the coding process. The compression starts with an initial automaton from its initial state. The initial state typically includes transitions for all possible symbols and it carries no context. After a symbol has been coded and statistics updated, the automaton can be updated applying two basic *cloning* operations:

- Create a new state
- Create a new transition

The modification of the automaton thus corresponds to the dynamic modelling in PPM-based compression schemes. The initial automaton does not need to be very complicated. It is more important that the automaton adapts to the input data. The key questions of CMA are when and how to clone. Solutions for these questions will be discussed in Section 3.

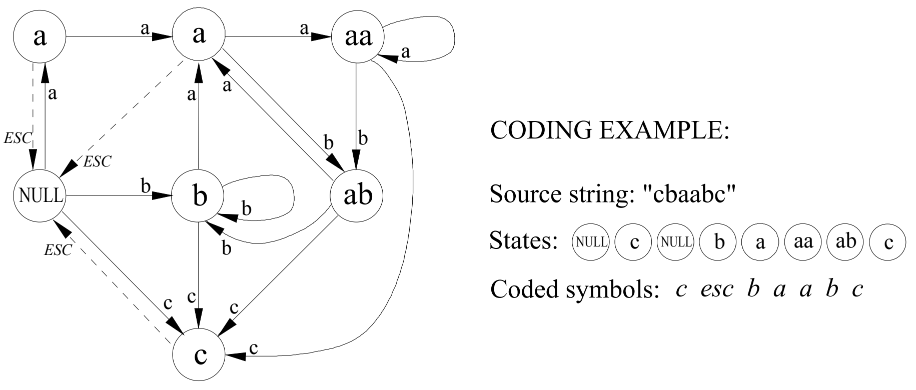

NULL
a
a
aa
b
ab
c
ESC
CODING EXAMPLE:
Source string: "cbaabc"
States: NULL c NULL b a aa ab c
Coded symbols: c esc b a a b c

Figure 1: Example of a context model automaton for alphabet $\Sigma = \{a, b, c\}$.

```
Compress(File):
    {
    C = C0   /* initial state */
    REPEAT
        Symbol = GetNextSymbol(File);
        C = CompressSymbol(C, Symbol);
    UNTIL Symbol=EOF;
    }

CompressSymbol(C, Symbol): returns next state.
    {
    IF EXISTS( C.Transition(Symbol) ) THEN
        EncodeSymbol(C, Symbol);
        UpdateStatistics(C, Symbol);
        nextC = C.Transition(Symbol);
        Update automaton
    ELSE
        EncodeSymbol(C, ESC);
        UpdateStatistics(C, ESC);
        nextC = CompressSymbol(C.transition(ESC), Symbol);
        Update automaton
    RETURN nextC;
    }
```

Figure 2: Pseudo code for the CMA coding process.

2.1. Formal description of CMA

The following symbols and definitions will be used to describe CMA:

Alphabet of all symbols	Σ
State in the automaton	C
Initial state	C_0
Order of the context in C	$\|C\|$
Any transition ($S \rightarrow D$)	T
Source state for a transition	S
Destination state for a transition	D
Escape transition	*ESC*
Label of a transition	*Label*(T)

There are two kinds of transitions in the automaton: actual transitions and escape transitions. *Label*(*ESC*) is defined to be an empty string. Note that the *ESC*-symbol still needs to be coded. *Path* describes a sequence of transitions corresponding to a traversal in the automaton. *Label of path* denotes the catenation of all non-*ESC* labels of the transitions along the path. The label of traversed path corresponds to a fixed number of previously coded symbols.

Using the above symbols and definitions, an automaton is defined to be a context model automaton (CMA) if it meets the following criteria:

> **Definition 1: (deterministicity property)**: *There is only one outgoing transition from each state with the same label. Furthermore, only one possible outgoing ESC-transition exists in each state.*
>
> **Definition 2: (termination property)**: *The automaton must terminate in a finite number of steps for any source string.*
>
> **Definition 3: (context property)**: *The context in each state C is the longest common suffix of the labels of all possible paths from C_0 to C. The order of the context in C is the length of that suffix string, and is denoted by $|C|$.*
>
> **Definition 4: (suffix property)**: *Consider a transition T in the form $S \rightarrow D$. The context of D is Suffix(S)+Label(T). In other words, the next context after an actual transition is always a suffix of the context of the previous state, concatenated with the symbol just coded.*

The termination property states that the automaton must be able to encode any source string and terminate within a finite time, i.e., it recognizes all the languages over the alphabet. This means that all escape paths from any state must eventually lead to a state that is capable of encoding the input symbol. In practice, this is usually solved so that all escape paths end up in the initial state of the automaton, which is capable of coding all alphabet symbols.

The context property defines the context of a state *C* as the longest uniquely determined context. For example, consider the state *ab* in Figure 1. The labels of all possible incoming paths of length 3 are *aab* and *bab*. Their longest common suffix is *ab*, which is therefore the context of that state due to the context property. The actual context, however, may still be longer than $|C|$. In the previous example, there exists a longer (but only partially unique) context (*a* or *b*)+*ab* because the incoming path with the label *cab* cannot appear.

The following corollaries are deduced directly from the CMA definitions:

> **Corollary 1**: *If there is an actual transition from S to D, then $|D| \leq |S| + 1$.*
>
> **Corollary 2**: *If there is an ESC-transition from S to D, then $|D| \leq |S|$.*
>
> **Corollary 3**: *All actual transitions incoming to D have the same label. Furthermore, if there is at least one actual transition incoming to D then the last non-ESC transition along all paths leading to D must have the same label.*

The corollary 1 states that the order of the context cannot be increased by more than one due to a single transition. This can be verified both by the context and suffix properties. The suffix property imposes also that the context cannot expand due to an escape transition (corollary 2). The suffix property is rather strict. Consider an *ESC*-transition between two states with a context of order 1 or greater. The suffix property implies that these contexts must have the same last symbol (corollary 3). A weaker definition would have stated that *D* = *Suffix*(*S*+*Label*(*T*)) but we adopt here the stronger definition.

2.2. Dynamic construction of CMA

CMA gives context interpretation for the states of automaton. However, CMA models only an existing automaton and does not say anything about how the automaton is constructed. The dynamic construction of CMA must therefore be defined by an initial automaton and by a set of cloning operations. We define a CMA-based compression method recursively by the following three criteria:

Requirement 1: *The initial automaton is CMA.*

Requirement 2: *The new automaton due to a cloning operation is CMA.*

Requirement 3: *Cloning does not change the contexts of existing states.*

These first two requirements are rather loose and they give great freedom in the design of the context model. The third requirement, however, defines that cloning cannot change the contexts of an existing automaton. Thus, the context of a state is determined at the time of its creation and cloning operations may generate only local changes in the automaton. In this way, we can easily keep track of the way the automaton develops.

3. CMA implementations of GDMC and PPM models

In the following section we will show how the GDMC and PPM models can be implemented under the CMA concept. We will first discuss the cloning operations introduced in GDCM and show that they directly implement a CMA. The PPM model is then simulated using the same cloning operations, and by making only minor modifications to the *CompressSymbol* function.

3.1. GDMC model

The initial model of GDMC is given in Figure 3. It is clearly a deterministic automaton and it can encode all symbols by a finite number of steps because all states have an escape path leading to the initial state, which contains transitions for all alphabet symbols. If the states are assigned by the contexts shown in Figure 3, the automaton will also meet the context and suffix properties. Thus, the first requirement for a dynamic CMA is fulfilled.

3.1.1. Cloning operations in GDMC:

GDMC involves two cloning operations: (1) creating new states, and (2) creating new transitions (*shortcuts*). A new state can be created after a symbol has been coded in the THEN-branch of the *CompressSymbol* function (see Figure 2). The objects of the operation are the current state and the next state in the automaton. Denote the last coded symbol by *a* and the corresponding transition by $S \rightarrow D$. The cloning is performed by creating a new context *Sa* and redirecting the original transition from $S \rightarrow D$ to $S \rightarrow Sa$. An *ESC*-transition $Sa \rightarrow D$ is also created, see Figure 4. The decision when to clone depends on the frequency distribution in the context, and is issued in Section 3.3. Note that the cloning in GDMC is performed only if the source state directly predicts the next symbol. It is thus never applied after an escape path.

GDMC involves also the so-called *shortcut* operation. Shortcut is a new transition that bypasses a combination of an *ESC*-transition followed by an actual transition, see Figure 5. Note that it is possible to traverse a long *ESC*-path until a correctly labelled transition is found. In this case, only the last *ESC*-transition is involved in the shortcut operation, see Figure 6. Alternative shortcut strategies have been discussed in [11]. For example: (1) creating a direct shortcut $S_1 \rightarrow D$, or (2) creating all possible shortcuts $S_1 \rightarrow D$, $S_2 \rightarrow D$, and so on. It is also noted that in GDMC, it is allowed to create a new state directly after a shortcut has been created.

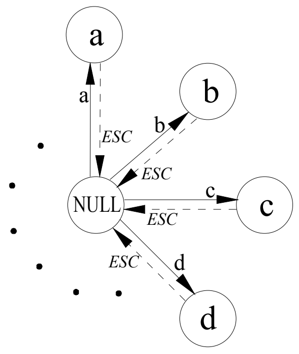


Figure 3: The initial model of GDMC.

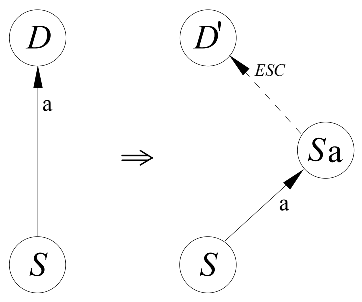


Figure 4: Cloning of GDMC.

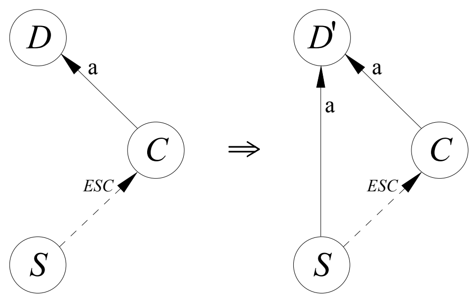


Figure 5: Shortcut of GDMC.

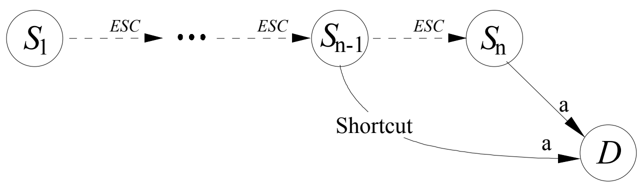


Figure 6: Long path of *ESC*-transitions.

3.1.2. CMA properties after context cloning:

The new automaton after the context cloning is deterministic and the inclusion of the escape transition does not break the termination property. The context of the source state *S* is not affected by the cloning and so we can limit the examination of the context and suffix properties to *Sa* and *D*' only. *D*' denotes here the state *D* after cloning. The context property states that the context of a state is the longest common suffix of the labels of all paths from C_0 to the state. As the only incoming transition to *Sa* is *S*→*Sa*, the context of the new state can be defined by *Sa* (catenation of *S* with *a*). This fulfills both the context and suffix properties.

The label of the longest common path incoming to *D*' is either the original context *D*, or *Sa* due to cloning. Thus, due to the context property *D*' = *Min*(*D*, *Sa*), where *Min* is defined as the shorter of the two strings. Considering that the automaton was CMA before cloning, we know that: *D* = *Suffix*(*S*)+*a* (suffix property). It follows that *D*' = *Min*(*Suffix*(*S*)+*a*, *Sa*) = *Suffix*(*S*)+*a* = *D*. In other words, the context of *D* does not change due to cloning and therefore *D*' fulfills the context property. Furthermore, the cloning has no effect on any successor of *D*. The suffix property of *D*' can also be verified easily: *D*' = *Suffix*(*Sa*)+Label(*ESC*) = *Suffix*(*Sa*) which is true because *D*' = *Suffix*(*S*)+*a*. The modified automaton therefore meets the CMA properties also after the cloning of a new state.

In the previous examinations we have assumed that no transition is directed to the state itself. It is possible that actual transitions in GDMC are sometimes directed to the state itself. Even so, all the above conclusions are still valid (proof is omitted here).

3.1.3. CMA properties after shortcut:

An *ESC*-transition is traversed in GDMC only when there is not outgoing transition labelled by the current symbol to be coded. Duplicate labels are thus never created and the automaton remains deterministic. The termination property is also preserved since new escape transitions are never created by shortcuts.

The creation of a shortcut does not affect the states *S* and *C* of Figure 5 in any way, and so only the state *D*' needs to be examined. The label of the longest common path incoming to *D*' is either *D* or *Suffix*(*S*)+*a* due to the new transition. Considering that the automaton was also CMA before the modification, we know that $D = Suffix(C)+a$ and $C = Suffix(S)$. It thus follows that

$$\begin{aligned} D' &= Min(D, Suffix(S)+a) \\ &= Min(Suffix(C)+a, Suffix(S)+a) \end{aligned} \qquad \textbf{(1)}$$

The fact that $C = Suffix(S)$ implies $|Suffix(C)+a| \leq |Suffix(S)+a|$. The above equation therefore simplifies to:

$$D' = Suffix(C)+a = D. \qquad \textbf{(2)}$$

This gives a proof to the context and suffix properties for *D*'. Moreover, since *D*' = *D*, the shortcut does not affect the succeeding states of *D* in any way. Thus, the modified automata due to the creation of a shortcut meet the CMA properties. Note that the alternative shortcut strategies discussed in [11] would meet all the *CMA* criteria, too.

3.2. PPM model

In PPM, the coding of a symbol starts always at the highest level $N_{\max}$. If the symbol cannot be coded by the current context, lower level contexts are reached using escape transitions. After a proper context has been found and the symbol has been coded, the process should then continue at the highest level again. The corollary 1 of the CMA properties, however, states that the order of context cannot increase by more than one due to a single transition. For example, suppose that the last three coded symbols were *abc* (assuming $N_{\max}=3$), but the next symbol *d* is predicted only by the context *c*. After *d* has been coded, the next context according to the PPM model should be *bcd*. But there cannot be a direct transition $c \rightarrow bcd$ in CMA.

In order to perform the PPM simulation, the meaning of a transition is more relaxed than in GDMC. Originally a transition corresponds not only to the coding of a symbol but it also indicates the state to be used for the next symbol. The cloning and creation of transitions are defined in the same way as in GDMC but the coding is not necessarily continued from the state pointed out by the transition. Instead, the next state is the new state due to cloning.

The *CompressSymbolPPM* function simulating the PPM model is defined in Figure 7. The function takes the symbol to be coded and an *N*-level context as input, and outputsthe next context to be used, which is located at level *N*+1 (or at level $N_{\max}$ in case of $N=N_{\max}$). If transition for the input symbol exists, the symbol is compressed and the transition is followed for obtaining the context for the next symbol. The resulting state is then returned as the output of the function. This is a trivial case where no cloning is needed.

In the case when the correct transition does not exist, the routine follows an escape path recursively through lower levels and when a proper state is found and the symbol coded, the function begins backtracking among the traversed escape path. New states are cloned during the backtracking and the result of the recursion is a state at the same level as the current one. If the current state is at level $N_{\max}$, a shortcut is created for pointing to the newly obtained state. Otherwise (which happens at the lower levels during the recursion) a new state is cloned at the level $N+1$. This process is illustrated in Figure 8 where *d* is the input symbol and *abc* the context. Since transition *d* does not exist the function is recursively called and the state *bcd* is eventually obtained as a result.

The simulation is now almost complete but we still need to define the initial automaton for PPM. In principle, the automaton of Figure 3 could be applied but the original algorithm in [7] uses slightly different initialisation. This is implemented by the automaton shown in Figure 9. The first state ("static") has transitions for all possible symbols in the alphabet and the statistical model assigned to the state is assumed to be static. This is sometimes referred as "-1 level context" in the literature.

The second state ("adaptive") has only an escape transition to the "static" state. In this way the model may better adapt to the input data because it has no transitions for unused symbols as a burden. In our implementation, we let also the "static" state to adapt to the input data.

The initial model of Figure 9 is theoretically not clean because the "adaptive" state contradicts the suffix property of CMA (incoming transitions should not have different labels). This has no practical consequences because the exception remains local and the rest of the dynamically constructed automaton will still meet all the CMA criteria. If we had adopted the weaker version of the suffix property (stating that $D = Suffix(S+Label(T))$ the initial model would have been a proper CMA.

```
CompressSymbolPPM(C, Symbol): returns next state.
    {
    IF EXISTS( C.Transition(Symbol) ) THEN
        EncodeSymbol(C, Symbol);
        UpdateStatistics(C, Symbol);
        nextC = C.Transition(Symbol);
    ELSE
        EncodeSymbol(C, ESC);
        nextC = CompressSymbolPPM(C.transition(ESC), Symbol);
        CreateTransition(C, nextC, Symbol);
        IF |C| < $N_{\max}$ THEN
            nextC = CloneContext(C, nextC, Symbol);
    RETURN nextC;
    }
```

Figure 7: Compression function for CMA in order to simulate PPM. The operations modifying the automaton are enhanced.

<Figure 8 is not included in this file due to technical problems. Only paper copy exists!!!>

Figure 8: Example of the use of the *CompressSymbolPPM* function.

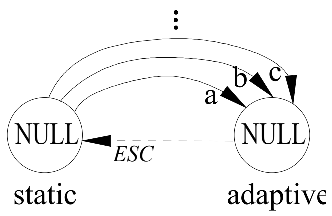


Figure 9: Initial model for PPM.

3.3. Properties of the GDMC and PPM models

In the following, we will study the behaviour of the GDMC and PPM context models using the CMA formalism as our tool. The contexts in CMA can be easily determined during the coding process by assigning the states with their level number. Both in GDMC and PPM, the level number $|C|$ of a new state C is already known at the time it is cloned. Due to the suffix property, it is always one higher than that of the source state involved in the cloning, see Figure 4. The actual context is then the $|C|$ last coded symbols.

3.3.1. Cloning states:

The key questions for creating new states are: when to clone and how to clone. PPM clones always after a symbol has been coded. Greedy context expansion is usually advantageous, see [1, p. 149] and [2], but the growth of the automaton must be limited in some way to keep the memory consumption reasonable. The straightforward solution of PPM limits the context lengths to a predefined maximum $N_{\max}$. GDMC, on the other hand, has no upper limit for the context length but it uses the following heuristic to control the growth. Consider that the transition $S \to D$ was traversed. A new state is created if the difference

$$\sum_i f\left(D \to DD_i\right) - f(S \to D) \qquad \textbf{(3)}$$

exceeds a predefined *cloning constant* (*CC*). Here DD_i refers to a successor state of *D*, and *f* to transition frequency. This gives a feasible solution for limiting the growth of the automaton. Best compression results were reported with very small CC-values [11]. Note that unbounded context lengths have also been proposed for PPM in [13]. This method limits the model growth by cloning only when a context is used second time.

Another significant difference between the GDMC and PPM models lies in the way the automaton is traversed. When coding a long sequence of escape transitions, the PPM model allows backtracking among the traversed escape path. New states are also cloned during the backtracking. In this way, PPM creates all possible contexts from the current level up to the level N_{max} whereas GDMC expands the automaton only by one context at a time. PPM is therefore more greedy in the context cloning.

After a new state has been cloned, there is a question whether the coding process should continue from the current state *D* (like in GDMC), or from the new state that was created in the cloning (like in PPM). At first sight there seems to be no point to continue from the newly created state because it has not yet learned any transitions. The context cannot therefore predict the next symbol and an escape transition must be followed back to the state *D*. The traversed escape transition, on the other hand, does not cost any bits since its probability is 1. Besides, if the coding continues from the new state it can directly learn the next symbol to be coded, and in this way the state adapts faster to the source data.

Figure 10 demonstrates the behaviour of the GDMC and PPM models when a sample string "abcaba" is coded. PPM learns all existing contexts up to the upper limit, whereas GDMC expandeds only up to the second level. None of the second level contexts has learned any transitions yet. The faster model growth of PPM can be summarised in the following three properties: (1) greedy cloning criterion, (2) multiple cloning operations in a single coding step, (3) coding is continued directly from the new state after cloning.

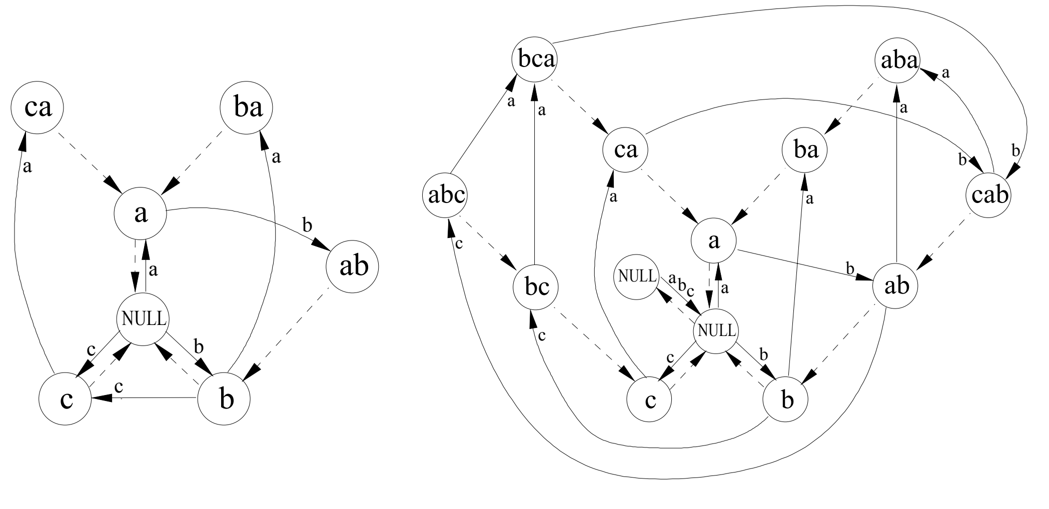


Figure 10: GDMC (left) and PPM (right) automata after the coding of a string *abcaba*.
In GDMC, cloning was performed when a context was used second time.
In PPM, N_{max} was set to 3.

3.3.2. Transitions:

Consider a transition $S \rightarrow D$ (assuming $|S|<N_{\max}$) and denote $|D|$-$|S|$ by the *increase of a context length* due to the transition. In PPM, all such transitions increase the context exactly by one. This is a consequence of the greedy cloning criterion in which the creation of a new transition is directly followed by a cloning operation. Only exceptions are the states locating at the level $N_{\max}$. Escape transitions decrease the context always by one.

In GDMC, transitions can also be directed to the same, or to a lower level in the automaton. For example, when coding the symbol *c* in the state *ab* of Figure 10, transition $ab \rightarrow c$ will be created. If the state *c* is not immediately cloned the transition will remain. This is possible due to the lazy cloning criterion. Transitions can also be directed to themselves. It may happen when the same symbol is repeated in the source data.

Escape transitions can be directed in GDMC to any lower level. For example, if the state *c* was cloned after traversing the transition $ab \rightarrow c$ (the new state being *abc*), the resulting escape transition ($abc \rightarrow c$) would decrease the context by two. These kind of arbitrary jumps between different levels are possible because the cloning criterion does not take into account the contexts of the states but it depends on the frequencies only.

3.3.3. Partial contexts:

The context property defines the minimum order of the context and implies that there might be a higher order *partial context* in the state. Consider a state *C* with an *N*-order context. Denote all possible incoming (*N*+1)-length paths to *C* by *xC*, and the set of all possible symbols *x* by *X*. The state *C* is defined to include a *partial context* of order *N*+1 if the set *X* is a true subset of the full symbol set ($X \subset \Sigma$). In other words, there exists at least one symbol in the alphabet that cannot precede the context.

In the case of full table strategy (where all states include transitions for all alphabet symbols), each longer context implies a partial context in all of its suffix contexts [12]. For example, if the state *ab* exists in the automaton, the partial context in *b* is $(\Sigma \backslash a)+b$ because the state *b* will never be used for the input string *ab*. Similarly, all other longer contexts reduce certain string combinations from the context of *b*.

In the case of escape strategy, the question of partial contexts becomes more complicated because of the escape transitions. For example, the state *b* in Figure 11 (left) does not include a partial context (*b* or *c*)+*b* despite the state *ab* exists in the automaton. The state *b* is indeed not reached with the input *abc*. However, the context of a state is defined merely on the basis of the history. The exclusion of certain string combinations depending on the future symbols is known as the *exclusion principle*. Nevertheless, it is not considered as a partial context because the state *b* can still be reached by all possible input strings *xb* ($x \in \Sigma$).

Variable order context models using escape strategy are not so strict about what is the exact context of a state. When a longer context (*ab*) adapts to more and more different transitions, the effect of the string *ab* is reduced in the shorter context *b*. It is therefore gradually transferring towards the context (*b* or *c*)+*b* but it may never reach that status.

Despite the above discussion, the CMA automata can still include partial contexts in the strict sense. Figure 11 illustrates how this kind of context is created by the cloning operation of GDMC. The new state *bc* includes a partial context (*b* or *c*)+*bc* because it cannot be reached when coding the string *abc* (due to the state *ab* and the transition *ab*→*c*). This originates from the lazy cloning criterion of GDMC, which allows states to be cloned after they have already been bypassed by shortcuts. The direct consequence is that the longest possible context is not always utilized. In PPM model, this kind of partial contexts cannot appear because of the greedy cloning criterion.

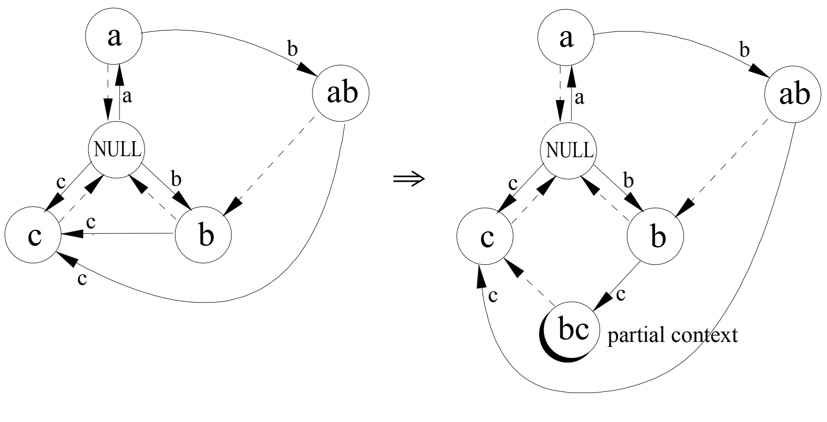


Figure 11: Example of a partial context in GDMC.

3.3.4. Duplicate contexts:

Consider the automaton of Figure 11 (left). Assume that the transition *NULL*→*c* was traversed and the state *c* is then cloned. This results in two states with identical contexts, see Figure 12. The original state *c* will be reached using the existing transitions *ab*→*c* and *b*→*c*, the new one using the redirected transition *NULL*→*c*.

Due to the suffix property, the cloning operation creates a new context whose order is one higher than that of the source transition. Duplicate contexts are therefore generated always when the traversed transition already increases the context by one: $|D|=|S|+1$. The main effect of the duplicates is that the new state will include a partial context. For example, the new context *c* in Figure 12 includes a partial context (*a* or *c*)+*c*. Duplicates may also be generated for $|D|<|S|+1$ if the source state *S* is already a duplicate context. In this case, there exist two branches in the automaton with similar contexts.

It is unclear what is the advantage/disadvantage of the duplicate contexts. Presumably they slow down the learning process because the same set of transitions must first be learned in the original context, and only then they can be learned in the new (duplicate) context. If so desired, the existence of duplicates could be easily avoided by allowing cloning only when $|D|<|S|+1$. Note that duplicate contexts cannot appear in PPM model due to the greedy cloning criterion.

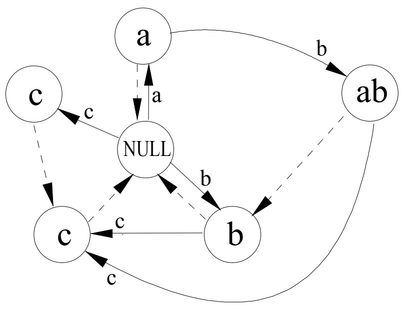


Figure 12: Example of duplicate contexts.

3.4. Probability estimation

Both PPM and GDMC use frequency-based probability model. The probability for a transition is calculated by dividing its frequency count by the sum of all frequency counts of the outgoing transitions. The only exception is the escape transition. In GDMC, escape transitions are treated as any other transitions and they thus has their own frequency counters. In PPM the probability for an escape transition is calculated by a predefined function known as *Escape method C* [14]:

$$p(ESC) = \frac{q}{q+c} \tag{4}$$

Here q is the number of outgoing transitions in the state, and c is the sum of their frequency counts, i.e., the total number of times the state has been visited. It is noted that q equals to the number of times the escape transition has been used. Both methods thus have the same intuition but the actual probability values may differ because of different initialisation and update strategies.

In PPM, the statistics for a new context start from scratch and the frequency of a new transition is always set to 1. In GDMC, the frequency of a new transition (referring to Figure 5) is estimated by:

$$f(S \to D) = \frac{f(S \to C) \cdot f(C \to D)}{Effsum(C|S)} \tag{5}$$

where $Effsum(C|S)$ is the sum of all transition counts in C excluding the ones existing also in S. This corresponds to the exclusion principle (see below). Due to this initialisation the frequencies are stored as real numbers instead of integer domain. See [11] for details of this probability modelling method.

Both PPM and GDMC apply the so-called *exclusion principle* when calculating the effective probability for a symbol. If escape transition was just traversed, it is known that the current symbol cannot be any of the symbols predicted by the previous state. All these symbols can therefore be excluded from the probability calculations. The exclusion principle always improves the compression because otherwise a proportion of the probability space would be wasted for impossible coding events.

After the symbol has been coded, its frequency count is incremented by one. The update is restricted only to the predictive state; the counters in the lower order states are not altered. This principle, known as *update exclusion*, is commonly used in PPM and also in GDMC. An alternative strategy would be to follow the escape path to the initial state and update the frequencies in all states along the path. For example, if the symbol was predicted by the context *abc*, the update would then consider also the states *bc* and *c*.

4. Empirical results

The files of Calgary corpus [1] were compressed using the GDMC and PPM models with the parameter setup of Table 1. We want to find out what kind of contexts (and how long) the models create, how fast they adapt to the source, and what kind of transitions there exist in the automata. Additional tests were performed by varying the main parameters of the models. The aim is to find out how the parameter setup affects the performance of the compression, and how well the context model can be controlled by the parameters. The throughput of the CMA-based implementation does not significantly differ from the previous implementations of PPM and GDMC.

In the literature, the maximum context length of PPM is often set to 3 but we use $N_{\max}$=4 because it gives lower bit rate for all files in Corpus, except the file *geo*. The average bit rates for various $N_{\max}$-values (1 to 7) are (3.62, 2.85, 2.46, 2.34, 2.32, 2.33, 2.34). In addition, we tested also GDMC with $N_{\max}$=4 (referred as GDMC-4). The results are summarised in Table 2. The average bit rates for GDMC and GDMC-4 are 2.52 and 2.55. The GDMC bit rates are 7 % higher than that of PPM on average. Slightly better results were obtained only for the three longest files (*book1*, *book2* and *pic*). The benefit of the unlimited context length remains marginal: 2 % on average and the maximum improvement is 6 % for the file *pic*.

PPM learns all contexts up to the maximum length that there are to be learned. GDMC is more picky in cloning and learns only about 35-80 % of the same contexts. It also learns a large number of longer contexts. The total number of contexts in GDMC is typically about three times as much as in PPM but only about 60 % of the contexts are unique; the rest are duplicates.

Table 1: Summary of the context and probability model parameters.

		PPM	**GDMC**
Context model	Context length limit:	$N_{\max}$ = 4	No upper limit
	Cloning criterion:	Clone always	Depends on statistics
	Multiple cloning:	Yes	No
Statistical model	Probability estimation:	Frequency based	Frequency based
	Initial statistics:	Starting from scratch	Derived from successor
	Escape probability:	PPM model *C*	Own frequency counter for ESC

Table 2: Performance comparison (bit rates) of the various methods.

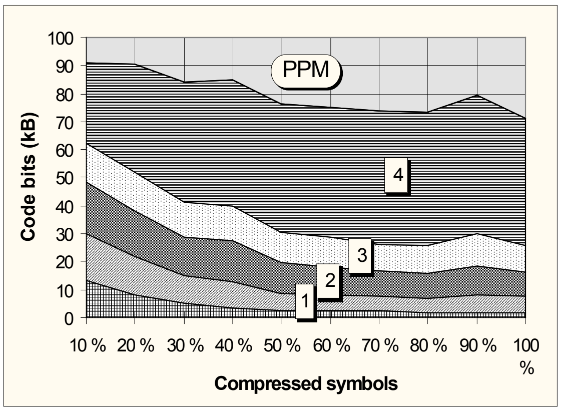
File	**Size**	**PPM-4**	**GDMC**	**GDMC-4**
bib	111261	**1.94**	2.04	2.08
book1	768771	2.33	**2.32**	2.33
book2	610856	2.04	**2.02**	2.07
geo	102400	**4.72**	5.16	5.16
news	377109	**2.43**	2.60	2.63
obj1	21504	**3.73**	4.40	4.41
obj2	246814	**2.51**	2.81	2.86
paper1	53161	**2.37**	2.58	2.59
paper2	82199	**2.35**	2.45	2.46
pic	513216	0.82	**0.80**	0.85
progc	39611	**2.41**	2.67	2.68
progl	71646	**1.77**	1.82	1.89
progp	49379	**1.76**	1.90	1.95
trans	93695	**1.58**	1.73	1.79
Average	224401	**2.34**	2.52	2.55

In PPM, the input symbols were predicted by a context length of 3.27 on average. In GDMC and GDMC-4, the corresponding numbers were 4.38 and 2.94 (excluding *obj1* and *pic*). In *pic*, the average length of the predicted context was as large as 1823 due to the high repetition of zero bytes (large homogenous white areas in the binary image). We also observed that GDMC uses almost always the longest available context in the automaton. This indicates that, even if there are partial contexts in the automaton (besides the duplicates), they are rarely used.

It was shown in Section 3.3 that duplicates in GDMC can be avoided by modifying the cloning criterion accordingly. Our experiments show that this modification does not have any significant effect on the compression performance. Small improvement (0-3 %) was achieved in case of 9 files out of 14, but the bit rate increased in the case of *geo* (4 %) and *pic* (16 %). This is most likely explained by the nature of their data, which is not taken into account in the compression. The file *pic*, for example, is a binary image but the context models process the data one byte at a time. GDMC utilizes the bitwise dependencies across the byte boundaries somewhat better since it is capable of modelling partial contexts (involved also in the duplicate contexts).

Figure 13 illustrates the source of the code bits. In PPM, 55 % of the code bits originate from the highest level (N=4). The code bits in GDMC originate from a wider variety of levels, but overall, the highest levels ($N\geq 4$) contribute by a lesser amount (38 %) than the highest level in PPM. Even if there are a great number of longer contexts in GDMC, they are only suffix extensions of those appearing also at the levels $N\leq 4$. GDMC thus relies more on the lower level contexts than PPM.

Figure 13 demonstrates also the growth of the models. The lower level contexts are used more often in the early stage of compression. When the compression proceeds and the models adapt to the data, the higher levels become more commonly used. The overall bit rate decreases over the course of time. PPM expands the contexts faster than GDMC, which is shown in a better compression performance in the early stages, see Figure 14.



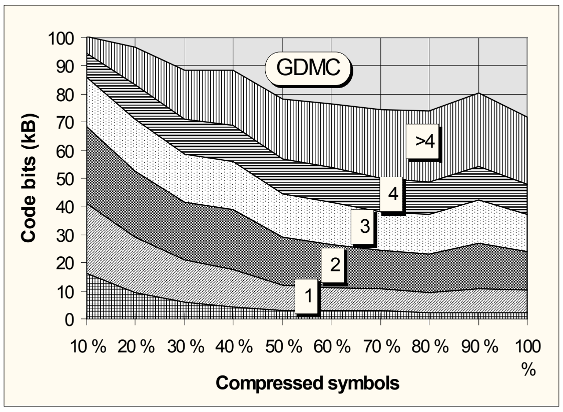


Figure 13: Source level of code bits as a function time (all files).

Error! Not a valid link.

Figure 14: Bit rates of GDMC relative to PPM as a function of time (average values).

In GDMC, 32 % of the code bits originates from escape symbols whereas the corresponding number in PPM is 14 %. By comparing the transition statistics, it was shown that GDMC learns fewer transitions than PPM. The number of transitions (excluding escape transitions) per context are 2.00 (PPM) and 1.04 (GDMC) on average. This is partly explained by the longer contexts of GDMC. They can predict certain symbols more accurately than the lower level contexts but the total number of transitions (predicted symbols) remains smaller. Escape transitions are therefore used more often. The destination level of the transitions, however, did not significantly differ between the models. Only very few transitions lead to the lower levels in GDMC, see Table 3.

Table 3: Destination level of transitions in respect to the source level.

	+1	same	-1	lower
PPM	34 %	66 %	0.0 %	0.0 %
GDMC	95 %	4 %	0.7 %	0.0 %
GDMC-4	42 %	56 %	2.4 %	0.0 %

PPM and GDMC have their differences also in the probability model. For example, they use different initial automata (see Figures 3 and 9). The initial automaton of PPM gives 1 % lower bit rate on average. A more significant difference is the initialisation of the frequency counter when a new transition is created. In PPM, the frequencies start from scratch whereas GDMC uses a more sophisticated frequency estimation technique (see Section 3.4). According to our experiments, the results of GDMC are about 10 % worse if the PPM-like probability model is applied. This implies that the model might be utilized also in PPM but unfortunately this is not the case. The GDMC probability model has been designed and optimized merely within the GDMC context model and we were not able to adopt the method in PPM.

5. Conclusions

A simple formalism was introduced to describe context model automata (CMA). It provides a strong tool to simulate existing context modelling methods such as PPM and GDMC. The behaviour of the above two methods were studied both analytically and experimentally from the CMA point of view. The main advantages of the PPM and GDMC models can be summarised as follows:

PPM:	**GDMC:**
• Fast adaptation to source	• No upper limit for context size
• Better initial model	• Partial contexts
• Accuracy of statistics	• Sophisticated probability model

The main advantage of PPM over GDMC is its faster adaptation to the source data. This originates from the following properties: (1) greedy cloning criterion, (2) multiple cloning operations in a single coding step, (3) coding is continued directly from the new state after cloning. PPM can learn up to $N_{\max}$ new contexts per coded symbol whereas GDMC can learn only one. The PPM model expands quicker in the early part of compression but the growth becomes smoother over the course of time. The GDMC model, on the other hand, keeps on growing with the same rate throughout the compression because there is no upper limit for the context length.

The initial model of PPM is slightly better (1 %) than that of GDMC because the zero-order context has no transitions for unused symbols as a burden. Furthermore, the statistics of the PPM contexts are more accurate than that of GDMC. The highest level contexts in PPM are created immediately when they first appear in the source data. The statistics thus concentrate on the relevant contexts only. In GDMC it takes longer to build the same contexts. There are also a great number of higher level and duplicate contexts in the GDMC automaton. The learning is therefore inefficiently distributed over a wider number of contexts resulting in less accurate statistics.

The unbounded context length of GDMC is both an advantage and a disadvantage. It allows more flexibility in the context length but it also results in unnecessarily long contexts because the context expansion is not properly controlled. This defects the learning of the statistics and a better cloning criterion should therefore be designed.

The other benefits of GDMC are also questionable. The potential of partial/duplicate contexts is their ability to model more efficiently contexts beyond the byte boundary (such as the binary data of the file *pic*). Unfortunately extra contexts break up the statistics into too many contexts resulting in less accurate prediction in the case of normal data files. There is also no direct control of the conditions under which the duplicates are created and they are created more or less in a random fashion. Moreover, we were not able to transfer the sophisticated estimation of transition frequencies into the PPM model. Its main benefit is most likely to compensate the slower learning efficiency of GDMC, i.e., the method tries to predict the frequency that the PPM model would have already learned due to its faster adaptation.

To sum up, CMA provides a formalism to describe context model automata. It gives useful information about the structure of the automaton. Potential weaknesses, such as duplicates and arbitrarily long contexts, can therefore be easily controlled if so desired. CMA, on the other hand, models only the structure of the automaton without giving any direct information about the compression performance. It is therefore important to consider also the statistical model when designing a CMA-based compression method. In particular, information theoretic state selection and the mixing of the probability estimates from different states have been shown to improve compression performance of a PPM model by 5-12 % [15].

Acknowledgements

The work of Pasi Fränti was supported by a grant from the Academy of Finland, and the work of Timo Hatakka by a grant from the Finnish Cultural Foundation.

References

1. Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) *Text Compression*, Prentice Hall, Englewood Cliffs, NJ.
2. Williams, R.N. (1991) *Adaptive Data Compression*, Kluwer Academic Publishers, Norwell, Massachusetts.
3. Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **23**, 337-343.
4. Ziv, J. and Lempel, A. (1978) Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, **24**, 530-536.
5. Huffman, D. (1952) A method for the reconstruction of minimum redundancy codes. *Proc. of the IRE*, **40**, 1098-1101.
6. Witten, I.H., Neal, R.M. anb Cleary, J.G. (1987) Arithmetic coding for data compression. *Communications of the ACM*, **30**, 520-540.
7. Cleary, J.G. and Witten, I.H. (1984) Data compression using adaptive coding and partial string matching, *IEEE Transactions on Communications*, **32**, 396-402.
8. Langdon, G.G. (1983) A note on the Ziv-Lempel model for compressing individual sequences. *IEEE Transactions on Information Theory*, **29**, 284-287.
9. Cormack, G.V. and Horspool, R.N.S. (1987) Data compression using dynamic markov modelling, *The Computer Journal*, **30**, 541-550.
10. Bell, T.C. and Moffat, A. (1989) A note on the DMC data compression scheme, *The Computer Journal*, **32**, 16-20.
11. Teuhola, J. and Raita, T. (1993) Application of a finite-state model to text compression, *The Computer Journal*, **36**, 607-614.
12. Bunton, S. (1995) The structure of DMC, *Proceedings Data Compression Conference*, Snowbird, Utah, 72-81.
13. Cleary, J.G. and Teahan, W.J. (1997) Unbounded length contexts for PPM, *The Computer Journal*, **40**, 67-75.
14. Moffat, A. (1990) Implementing the PPM data compression scheme, *IEEE Transactions on Communications*, **38**, 1917-1921.
15. Bunton, S. (1997) Semantically motivated improvements for PPM variants, *The Computer Journal*, **40**, 76-93.