Unbounded length contexts for PPM

John G. Cleary, W. J. Teahan, Ian H. Witten*

Department of Computer Science, University of Waikato, New Zealand

The PPM data compression scheme has set the performance standard in lossless compression of text throughout the past decade. The original algorithm was first published in 1984 by Cleary and Witten [3], and a series of improvements was described by Moffat [6], culminating in a careful implementation, called PPMC, which has become the benchmark version. This still achieves results superior to virtually all other compression methods, despite many attempts to better it. Other methods such as those based on Ziv-Lempel coding [9] are more commonly used in practice, but their attractiveness lies in their relative speed rather than any superiority in compression—indeed, their compression performance generally falls distinctly below that of PPM in practical benchmark tests [1].

Prediction by partial matching, or PPM, is a finite-context statistical modeling technique that can be viewed as blending together several fixed-order context models to predict the next character in the input sequence. Prediction probabilities for each context in the model are calculated from frequency counts which are updated adaptively; and the symbol that actually occurs is encoded relative to its predicted distribution using arithmetic coding. The maximum context length is a fixed constant, and it has been found that increasing it beyond about six or so does not generally improve compression [3, 6].

The present paper describes a new algorithm, PPM*, which exploits contexts of unbounded length. It reliably achieves compression superior to PPMC, although our current implementation—which we have not yet attempted to optimize—uses considerably greater computational resources (both time and space). The next section describes the basic PPM compression scheme. Following that we motivate the use of contexts of unbounded length, introduce the new method, and show how it can be implemented using a trie data structure. Then we give some results that demonstrate an improvement of about 6% over the old method. Finally, a recently-published and seemingly unrelated compression scheme [2] is related to the unbounded-context idea that forms the essential innovation of PPM*.

1 PPM: Prediction by partial match

The basic idea of PPM is to use the last few characters in the input stream to predict the upcoming one. Models that condition their predictions on a few immediately preceding symbols are called "finite-context" models of order k, where k is the number of preceding symbols used. PPM employs a suite of fixed-order context models with different values of k, from 0 up to some pre-determined maximum, to predict upcoming characters.

For each model, a note is kept of all characters that have followed every length-k subsequence observed so far in the input, and the number of times that each has occurred. Prediction probabilities are calculated from these counts. The probabilities associated with each character that has followed the last k characters in the past are

*email {jcleary, wjt, ihw}@waikato.ac.nz

Order $k=2$			Order $k=1$			Order $k=0$			Order $k=-1$		
Predictions	c	p	Predictions	c	p	Predictions	c	p	Predictions	c	p
ab $\rightarrow$ r	2	$\frac{2}{3}$	a $\rightarrow$ b	2	$\frac{2}{7}$	$\rightarrow$ a	5	$\frac{5}{16}$	$\rightarrow$ A	1	$\frac{1}{\lvert A\rvert}$
$\rightarrow$ Esc	1	$\frac{1}{3}$	$\rightarrow$ c	1	$\frac{1}{7}$	$\rightarrow$ b	2	$\frac{2}{16}$			
			$\rightarrow$ d	1	$\frac{1}{7}$	$\rightarrow$ c	1	$\frac{1}{16}$			
ac $\rightarrow$ a	1	$\frac{1}{2}$	$\rightarrow$ Esc	3	$\frac{3}{7}$	$\rightarrow$ d	1	$\frac{1}{16}$			
$\rightarrow$ Esc	1	$\frac{1}{2}$				$\rightarrow$ r	2	$\frac{2}{16}$			
			b $\rightarrow$ r	2	$\frac{2}{3}$	$\rightarrow$ Esc	5	$\frac{5}{16}$			
ad $\rightarrow$ a	1	$\frac{1}{2}$	$\rightarrow$ Esc	1	$\frac{1}{3}$						
$\rightarrow$ Esc	1	$\frac{1}{2}$									
			c $\rightarrow$ a	1	$\frac{1}{2}$						
br $\rightarrow$ a	2	$\frac{2}{3}$	$\rightarrow$ Esc	1	$\frac{1}{2}$						
$\rightarrow$ Esc	1	$\frac{1}{3}$									
			d $\rightarrow$ a	1	$\frac{1}{2}$						
ca $\rightarrow$ d	1	$\frac{1}{2}$	$\rightarrow$ Esc	1	$\frac{1}{2}$						
$\rightarrow$ Esc	1	$\frac{1}{2}$									
			r $\rightarrow$ a	2	$\frac{1}{3}$						
da $\rightarrow$ b	1	$\frac{1}{2}$	$\rightarrow$ Esc	1	$\frac{1}{3}$						
$\rightarrow$ Esc	1	$\frac{1}{2}$									
ra $\rightarrow$ c	1	$\frac{1}{2}$									
$\rightarrow$ Esc	1	$\frac{1}{2}$									

Table 1: PPM model after processing the string *abracadabra* (maximum order 2)

used to predict the upcoming character. Thus from each model, a separate predicted probability distribution is obtained.

These distributions are effectively combined into a single one, and arithmetic coding is used to encode the character that actually occurs, relative to that distribution. The combination is achieved through the use of "escape" probabilities. Recall that each model has a different value of k. The model with the largest k is, by default, the one used for coding. However, if a novel character is encountered in this context, which means that the context cannot be used for encoding it, an "escape" symbol is transmitted to signal the decoder to switch to the model with the next smaller value of k. The process continues until a model is reached in which the character is not novel, at which point it is encoded with respect to the distribution predicted by that model. To ensure that the process terminates, a model is assumed to be present below the lowest level, containing all characters in the coding alphabet. This mechanism effectively blends the different order models together in a proportion that depends on the values actually used for escape probabilities.

As an illustration of the operation of PPM, Table 1 shows the state of the four models with $k = 2, 1, 0$, and -1 after the input string *abracadabra* has been processed. For each model, all previously-occurring contexts are shown with their associated predictions, along with occurrence counts c and the probabilities p that are calculated from them. By convention, $k = -1$ designates the bottom-level model that predicts all characters equally; it gives them each probability $|A|$ where A is the alphabet used.

Some policy must be adopted for choosing the probabilities to be associated with the escape events. There is no sound theoretical basis for any particular choice in

character	probabilities encoded (without exclusions)	probabilities encoded (with exclusions)	code space occupied
c	$\frac{1}{2}$	$\frac{1}{2}$	$-\log_2 \frac{1}{2} = 1$ bit
d	$\frac{1}{2}, \frac{1}{7}$	$\frac{1}{2}, \frac{1}{6}$	$-\log_2(\frac{1}{2} \cdot \frac{1}{6}) = 3.6$ bits
t	$\frac{1}{2}, \frac{3}{7}, \frac{5}{16}, \frac{1}{\lvert A\rvert}$	$\frac{1}{2}, \frac{3}{6}, \frac{5}{12}, \frac{1}{\lvert A\rvert-5}$	$-\log_2(\frac{1}{2} \cdot \frac{3}{6} \cdot \frac{5}{12} \cdot \frac{1}{251}) = 11.2$ bits

Table 2: Encodings for three sample characters using the model in Table 1

the absence of some *a priori* assumption on the nature of the symbol source; some alternatives are evaluated in [8]. The method used in the example, commonly called "Method C," gives a count to the escape event equal to the number of different symbols that have been seen in the context so far [6]; thus, for example, in the order-0 column of Table 1 the escape symbol receives a count of 5 because five different symbols have been seen in that context.

Sample encodings using these models are shown in Table 2. As noted above, prediction proceeds from the highest-order model ($k = 2$). If the context successfully predicts the next character in the input sequence, the associated probability p is used to encode it. For example, if c followed the string *abracadabra*, the prediction $ra \rightarrow c$ would be used to encode it with a probability of $\frac{1}{2}$, that is, in one bit.

Suppose instead that the character following *abracadabra* were d. This is not predicted from the current $k = 2$ context ra. Consequently, an escape event occurs in context ra, which is coded with a probability of $\frac{1}{2}$, and then the $k = 1$ context a is used. This does predict the desired symbol through the prediction $a \rightarrow d$, with probability $\frac{1}{7}$. In fact, a more accurate estimate of the prediction probability in this context is obtained by noting that the character c cannot possibly occur, since if it did it would have been encoded at the $k = 2$ level. This mechanism, called "exclusion," corrects the probability to $\frac{1}{6}$ as shown in the third column of Table 2. Finally, the total number of bits needed to encode the d can be calculated to be 3.6.

If the next character were one that had never been encountered before, say t, escaping would take place repeatedly right down to the base level $k = -1$. Once this level is reached, all symbols are equiprobable—except that, through the exclusion device, there is no need to reserve probability space for symbols that already appear at higher levels. Assuming a 256-character alphabet, the t is coded with probability $\frac{1}{251}$ at the base level, leading to a total requirement of 11.2 bits including those needed to specify the three escapes.

It may seem that PPM's performance should always improve when the maximum context length is increased, because the predictions are more specific. Figure 1 shows how the compression ratio varies when different maximum context lengths are used, for the text of Thomas Hardy's novel *Far from the Madding Crowd* (file *book1* in the Calgary text compression corpus [1]). The graph shows that the best compression is achieved when a maximum context length of five is chosen and that it deteriorates slightly when the context is increased beyond this.

This general behavior is quite typical. The reason is that while longer contexts do provide more specific predictions, they also stand a much greater chance of not giving rise to any prediction at all. This causes the escape mechanism to be used more frequently to reduce the context length down to the point where predictions start to appear. And each escape operation carries a small penalty in coding efficiency.

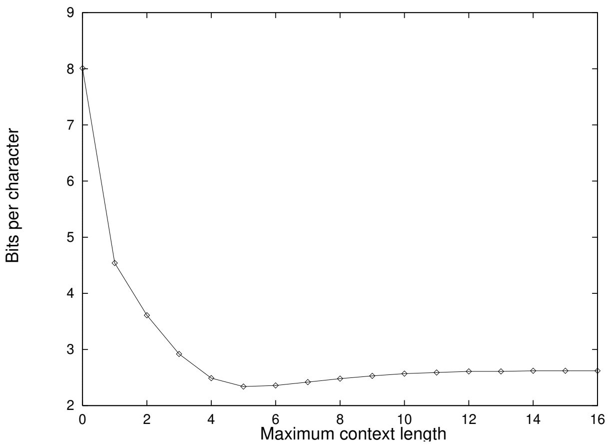


Figure 1: How the PPM compression ratio varies with maximum context length

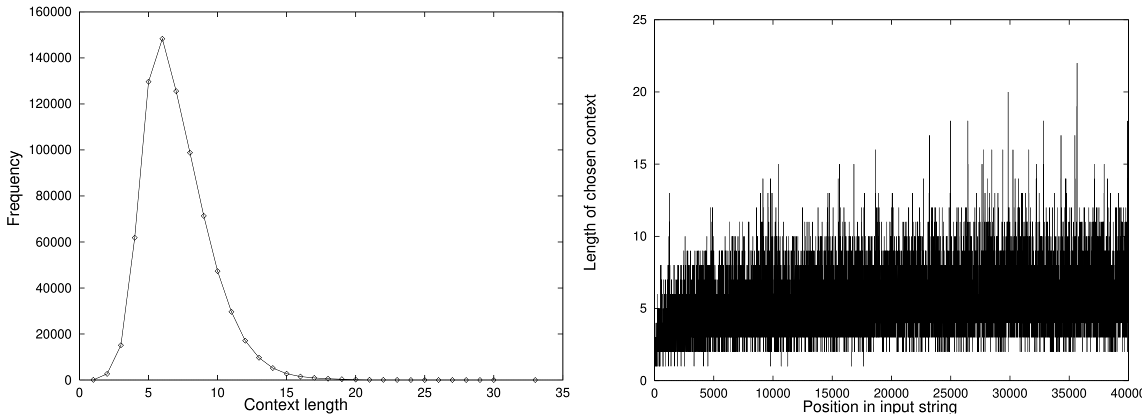
2 PPM*: Exploiting longer contexts

An alternative to PPM's policy of imposing a universal fixed maximum upper bound on context length is to allow the context length to vary depending on the coding situation. It is possible to store the model in a way that gives rapid access to predictions based on *any* context, eliminating the need for an arbitrary bound to be imposed. We call this approach, in which there is no *a priori* bound on context length, PPM*. It bestows the freedom to choose any policy for determining the context to be used for prediction, subject only to the constraint that the decoder must be able to make the same choice despite the fact that it does not know the upcoming character.

How to choose which context is the best for prediction is an area of intense research. One attractive-sounding possibility is to keep a record, for each context, of how well it compressed in the past. The same record could be maintained independently by both encoder and decoder, and they could use the context with the best average compression. Curiously, this policy does not perform well in practice. This can be explained by considering its behavior under random input. Then some contexts will perform better than others purely by chance, and the best-performing ones will be selected for prediction. Of course, with random input good performance in the past is no guarantee of good performance in the future. The best policy is to use a zero-length context, and the worst thing one can do is to use a relatively "extreme" context, even if its historical performance does lie markedly above that of its competitors!

A simple but effective strategy is as follows. A context is defined to be "deterministic" when it gives only one prediction. We have found in experiments that for such contexts the observed frequency of the novel characters is much lower than expected based on a uniform prior distribution. This can be exploited by using such contexts for prediction. The strategy that we recommend is to choose the shortest deterministic context currently in the context list. If there is no deterministic context, then the longest context is chosen instead.

A histogram of the context lengths chosen by this strategy is shown in Figure 2(a) for the file *book1*. The histogram peaks sharply at a context length of five to six; not surprisingly the best context length (for this file) is five (Figure 1). Notwithstanding this peak, however, the length of the shortest deterministic context varies widely: Figure 2(b) plots it for the first 40,000 character positions in the file *book1*. The graph demonstrates that deterministic contexts much longer than five or six occur



(a) (b)

Figure 2: (a) Histogram of the length of shortest deterministic contexts (b) Length of the deterministic context at each character position of *book1*

frequently and gradually increase in length as more input is seen, providing evidence that longer contexts provide improved compression.

The main problem associated with the use of unbounded contexts is the amount of memory necessary to store them. It has often been noted that it is impractical to extend PPM to models with a substantially higher order because of the exponential growth of the memory that is required as k increases. For PPM*, the problem is even more daunting, as it demands the ability to access all possible contexts right back to the very first character. Although this can be done by simply scanning back through the input string, the $O(N^2)$ execution time incurred rules that out in practice.

2.1 Context tries

A key insight in solving this problem is that the trie structure used to store PPM models can operate in conjunction with pointers back into the input string. In particular, a leaf node can point into the input string whenever a context is unique. Then, if the context needs to be extended, it is only necessary to move the input pointer forward by one position. To update the trie, a linked list of pointers to the currently active contexts can be maintained, with the longest context at the top. We call the resulting data structure a "context trie."

Figure 3 illustrates the context trie for the string *abracadabra*. The root node of the trie (the null string "Λ") is at the top. Contexts that have occurred before in the input string extend downward until they become unique, at which point a pointer, shown by a dashed line in the diagram, is stored back into the input string. For example, looking to the very left of the tree, none of *a*, *ab*, *abr*, *abra* are unique—they all appear two or more times in the input string—whereas *abrac* is unique. Consequently it is at this level that a pointer into the input string is substituted for further refinement of the trie structure.

The context list is shown at the lower right. It relates to the current position in the input string, and contains pointers to the contexts that are currently active. These are labelled 0 to 4 in the boxes on the left, and the corresponding nodes are marked with numbered arrows. The longest active context *abra* is placed at the top of the list, and each context below it is missing one further character. The number of elements in the context list is the length of the longest context, plus one for the root node. The list always contains at least one node—the root.

As each character is processed, the context trie is updated by updating each node pointed at by the context list. There are four possibilities when updating a node, depending on the new symbol in the input string and the state of the node.

The first two cases correspond to a situation where the next character is already predicted by the context trie. If there is link to a lower node in the trie for the new symbol, the context pointer is replaced by a pointer to that node. Suppose, in Figure 3, that the next character is *c*. Because this has occurred before in all the contexts, the structure can be updated by moving each of the five context pointers down one level to the corresponding nodes for the letter *c*. However, if the link pointed into the input string instead of to a lower node in the trie, then a new node is created along one position in the input string. Both the original link and the context are updated to point to the new node. Suppose that the next character after the *c* is *a*. This is already predicted by the *abrac* context to the left of the trie, as well as by the *brac*, *rac*, *ac*, and *c* contexts, so that five new nodes are created all pointing at the letter *d*.

The second pair of cases correspond to the situation where the next character is new in this context, that is, when there is no prediction out of the current node corresponding to that character. Suppose first that there are links to lower levels in the trie, but that they correspond to other characters. Then a new node is created for the new character, containing that symbol and a pointer to the next input position and it is dropped from the context list. For example, in Figure 3, if the next character is *b* then the contexts at pointers 2, 3, and 4 will be updated by adding child nodes for the character *b*. The contexts at positions 0 and 1, however, already have *b* predictions and so do not need to be changed. Finally, if there is a link out of the current node into the input string, but the next character is not the expected one, then *two* new trie nodes will have to be created, one for the expected character and the other for the new one. Both of these will have pointers into the input string, the former to (one past) the original position, and the latter to the new position at the end of the string. For example, if first *c* and then *x* were added to Figure 3, the five *c* nodes at the leaves of the trie would each gain two children, an *a* child pointing to around the middle of the input string and an *x* child pointing to the end.

2.2 Implementation issues

Using a PATRICIA*-style trie.* Substantial space can be saved in the context trie by collapsing non-branching sub-paths into single nodes, just like the standard PATRICIA trie data structure [7]. For each collapsed node, only one branch emanates from it. In Figure 3 there are three such paths, two with the letters *brac* and the third with *rac*. Collapsing non-branching paths requires two extra pointers to be stored with each node: the length of the string that the node represents, and a pointer to where it ends in the input string. In addition, an extra pointer associated with each position in the context list gives the current position in the non-branching path. The effect of collapsing all such paths into single nodes is to make the number of nodes in the trie linear with the size of the input string. Note that deterministic contexts, defined earlier, correspond to the non-branching paths in the context trie.

Storing the counts. Prediction in PPM* is based upon the frequencies of the characters that follow each context. These counts are stored with each node in the context trie, and are incremented whenever the node is updated. Collapsing non-branching paths causes some complication with count storage, because several different contexts

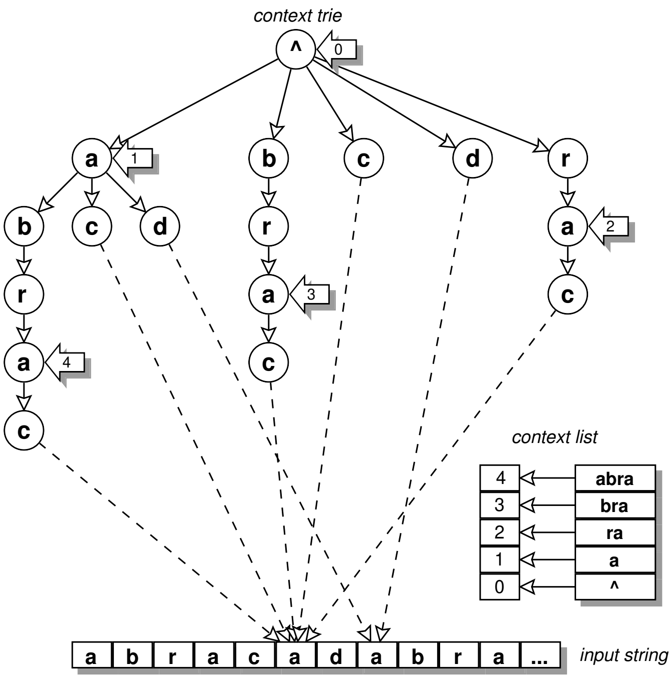


Figure 3: Context trie for the string *abracadabra*

may point to the same trie node. In this case a different count will be associated with each context, and so the count needs to be stored with the context list entry. When a collapsed node is first entered, this count is set to the current node count, which is then incremented. In the event that the non-branching path becomes split later on, the count from the context list is stored with the new node.

Scaling the counts. As is usual in PPM implementations, the frequency counts must be scaled downward periodically to prevent overflow in the arithmetic coder. This is normally done by halving the counts when a certain threshold has been exceeded, which has generally been found to improve compression because of its locally adaptive effect. For PPM*, counts are best scaled at the encoding stage because of the large number of 1 counts that occur throughout the context trie. Experimental results show that low scaling thresholds consistently reduce the compression obtained, and so the largest scaling threshold consistent with the requirements of the arithmetic encoder is used.

Escaping. In PPM* it is still necessary to cope with novel characters that have not been encountered before in the chosen context. Escaping can be used here, just as in PPM, by dropping down to the next member of the context list until a context that predicts the character is found. If the bottom of the list is reached, a uniform fixed distribution ($k = -1$) is used. We are presently using escape method C, which needs an extra "escape" count, equal to the number of branches, to be stored with every trie node. Other methods such as the ones proposed in [5, 8] may improve compression, although this remains to be investigated.

Escaping is even more crucial to PPM* than it is to PPM. With a fixed maximum context length, use of the escape mechanism in PPM will decay with time as all the contexts' predictions fill up. However, in PPM*—particularly with our current policy of choosing deterministic contexts—escaping will always be frequently used no matter

file	size (bytes)	PPMC (bpc)	PPM* (bpc)	BW94 (bpc)
bib	111261	2.11	**1.91**	2.07
book1	768771	2.48	**2.40**	2.49
book2	610856	2.26	**2.02**	2.13
geo	102400	4.78	4.83	**4.45**
news	377109	2.65	**2.42**	2.59
obj1	21504	**3.76**	4.00	3.98
obj2	246814	2.69	**2.43**	2.64
paper1	53161	2.48	**2.37**	2.55
paper2	82199	2.45	**2.36**	2.51
pic	513216	1.09	0.85	**0.83**
progc	39611	2.49	**2.40**	2.58
progl	71646	1.90	**1.67**	1.80
progp	49379	1.84	**1.62**	1.79
trans	93695	1.77	**1.45**	1.57
average	224402	2.48	**2.34**	2.43

Table 3: Compression ratios for the Calgary corpus

how much input is processed, possibly at a constant rate. It is interesting to note that escaping down contexts one character at a time may not be the best strategy. Since there are numerous contexts to choose from, it may be better to select the new context on grounds other than its length. Such strategies have not yet been investigated.

Exclusions. PPM* makes use of exclusions when performing prediction by excluding predictions found at a higher order when calculating the probabilities for lower-order contexts, just as described previously for PPM (and illustrated in Table 2). Update exclusion, a feature of PPMC (not described above) in which the count for a predicted character is not incremented if it is already predicted by a higher-order context, is not incorporated into PPM* because to do so would require significant re-organization of the data structure. Update exclusion improves the performance of PPMC by about 2%; whether it provides similar gains for PPM* is an open question.

3 Results

Table 3 shows the result of running PPM* on the Calgary corpus [1], along with the standard PPMC implementation and a recently-published, and extremely competitive, non-adaptive scheme labeled BW94 [2] (described in the following section). The best figure for each file is printed in bold.

Averaged over the entire corpus, PPM* yields a 5.6% improvement over PPMC, and a 3.7% improvement over BW94. It performs relatively poorly on just three files—*obj1*, *geo* and *pic*. It tends to perform less well on smaller files (e.g. *obj1*) and on files that are binary rather then character-based (e.g. *geo* and *pic*). It is interesting to note that of all the compression schemes listed in [1], PPMC is the only one that ever outperforms either BW94 or PPM*, and then only on a single file (*obj1*).

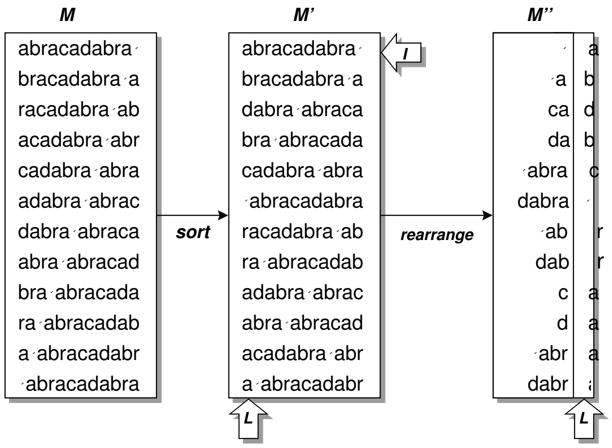


Figure 4: BW94 compression of the string *abracadabra*□

4 Related work

The archetypical compression scheme that uses unbounded contexts is, of course, LZ78, which parses the text into ever-growing phrases and encodes it as phrase numbers alternating with explicitly-represented single characters [9]. This scheme is known to be asymptotically optimal for an ergodic source, although convergence is very slow and in practice the method does not perform particularly well. Much effort has gone into devising improvements to the basic method, and one of the later ones, LZFG [4], is not too far behind PPMC in compression performance (and greatly superior to it in speed).

Very recently, a novel block-sorting algorithm has been described that achieves compression as good as context-based methods such as PPM but at execution speeds closer to Ziv-Lempel techniques [2]; we call this BW94 after its inventors Burrows and Wheeler. This method, unlike all others considered in the present paper, is not adaptive: first the complete input sequence is transformed and then the resulting output is encoded. The algorithm is effective because the transformed string contains more regularities than the original one.

At first glance, BW94 seems completely different to the context-based approach taken by PPM*. However, it can indeed be viewed as a context-based method, with no predetermined upper bound to context length. We illustrate it using "abracadabra□" again (note the inclusion of □ as end of file symbol).

Using the algorithm described in [2], first generate the matrix of strings M in Figure 4, then sort the reversed strings alphabetically to produce M'. Two parameters are extracted from the sorted matrix. The first, I, is an integer that records which row number corresponds to the original string. The second, L, is the character string that constitutes the first column. In this example, $I = 0$ and $L = abdbc\square rraaaa$. Strange as it may seem, the input string is completely specified by I and L: the reverse transformation for reconstructing the original is explained in [2]. Moreover, L can be transmitted very economically because it has the property that the same letters often fall together into long runs.

M'' is the same as M' but with L moved from the left to the right and symbols not needed to form unique contexts suppressed. It is clear then that the characters of L correspond to the predictions of the unbounded contexts lying to their left. It is also interesting to note that the context trie used to implement PPM* can equally well be applied to generate the BW94 transform, instead of the suffix trees proposed in [2]. This is accomplished by building the context trie from the input text as for PPM*. Then, after the complete string has been processed, one simply writes out for each leaf node the character in the input string that precedes the context pointed to by that node. The output can be encoded in the same manner as described by Burrows and Wheeler.

In summary, whereas BW94 can be viewed as exploiting contexts of unbounded length by sorting them after the whole input string has been processed, PPM* works adaptively by predicting the next character from previous, unbounded-length, contexts.

5 Conclusions

A new method of text compression, PPM*, has been described that outperforms all others on test files such as the Calgary corpus. The method revolves around the use of ever-growing contexts, and a data structure has been detailed that permits arbitrarily long contexts to be represented efficiently.

Also described is another, seemingly quite different, method of compression that has been introduced very recently. Surprisingly, this also appears to gain its power from its ability to utilize unbounded contexts.

Although there are a number of obvious areas in which further investigation will probably result in improvements to PPM*, it already provides a 5.6% performance increase over its predecessor, PPM. While this is not a large practical gain, we are clearly in an area where diminishing returns are to be expected. The most important contribution of PPM* is in pointing the way towards a general treatment of unbounded contexts.

References

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

[2] Burrows, M. and Wheeler, D.J. (1994) "A block-sorting lossless data compression algorithm," Technical Report, Digital Equipment Corporation, Palo Alto, California.

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

[4] Fiala, E.R. and Green, D.H. (1989) "Data compression with finite windows," *Communications of the ACM*, **32**(4), 490–505.

[5] Howard, P.G. (1993) "The design and analysis of efficient lossless data compression systems," Report CS–93–28, Brown University, Providence, Rhode Island.

[6] Moffat, A. (1990) "Implementing the PPM data compression scheme," *IEEE Transactions on Communications*, **38**(11), 1917–1921.

[7] Sedgewick, R. (1988) *Algorithms.* Addison-Wesley, Reading, Massachusetts.

[8] Witten, I.H. and Bell, T.C. (1991) "The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression," *IEEE Transactions on Informaton Theory*, **37**(4), 1085–1094.

[9] Ziv, J. and Lempel, A. (1978) "Compression of individual sequences via variable rate coding," *IEEE Transactions on Information Theory*, **24**(5), 530–536.