Unbounded Length Contexts for PPM

JOHN G. CLEARY AND W. J. TEAHAN

Department of Computer Science, University of Waikato, Hamilton, New Zealand
Email: jcleary@cs.waikato.ac.nz, wjt@cs.waikato.ac.nz

The PPM data compression scheme has set the performance standard in lossless compression of text throughout the past decade. PPM is a finite-context statistical modelling technique that can be viewed as blending together several fixed-order context models to predict the next character in the input sequence. This paper gives a brief introduction to PPM, and describes a variant of the algorithm, called PPM*, which exploits contexts of unbounded length. Although requiring considerably greater computational resources (in both time and space), this reliably achieves compression superior to the benchmark PPMC version. Its major contribution is that it shows that the full information available by considering all substrings of the input string can be used effectively to generate high-quality predictions. Hence, it provides a useful tool for exploring the bounds of compression.

Received June 28, 1996; revised July 25, 1997

1. INTRODUCTION

The prediction by partial matching (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 [1], and a series of improvements was described by Moffat, culminating in a careful implementation, called PPMC, which has become the benchmark version [2]. 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 [3, 4] 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 [5].

Prediction by partial matching, or PPM, is a finite-context statistical modelling 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 [6, 7]. The maximum context length is a fixed constant, and it has been found that increasing it beyond about 5 does not generally improve compression [1, 2, 8].

The present paper[1] describes an algorithm, PPM*, which exploits contexts of unbounded length. It reliably achieves compression superior to the benchmark PPMC version, although our current implementation uses considerably greater computational resources (in both time and space). The next section describes the basic PPM compression scheme. Following that we give our motivation for 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 benchmark PPMC. Finally, other seemingly unrelated compression schemes are related to the unbounded-context idea that forms the essential innovation of PPM*.

This paper uses the compression achieved on the standard Calgary text compression corpus [5] as a measure of how good the PPM* model is. The importance of this goes beyond the incremental improvement in the size of the compressed text. Having a computer model that achieves close to human performance is critical in areas such as speech recognition, spell-checking, OCR and language identification. Teahan and Cleary [9] show how the PPM scheme can be used to build a character-based computer model that can predict English text almost as well as humans. They performed experiments on the same text that Claude E. Shannon used in a famous experiment to estimate the entropy of English [10], and found that performance was close to, and in some cases superior to, human-based results. It is also well-known in cryptography that removing redundancy is important prior to encryption to prevent statistical attacks [11]. It is important here that there are no models (human or otherwise) that are significantly better than the model used to remove the redundancy.

2. 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

[1] A preliminary form of this paper [25] was presented at the 1995 IEEE Data Compression Conference.

TABLE 1. PPMC model after processing the string *abracadabra* (maximum order 2)

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	$1/\lvert A\rvert$
$\rightarrow$ 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	$\frac{1}{2}$									

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 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 to encode 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 $1/|A|$ 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 the absence of some *a priori* assumption on the nature of the symbol source; some alternatives are evaluated in [8, 12]. 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 [2]; 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

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

	Probabilities encoded		
Char.	(without exclusions)	(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	$\frac{1}{2}, \frac{3}{7}, \frac{5}{16}, 1/\lvert A\rvert$	$\frac{1}{2}, \frac{3}{6}, \frac{5}{12}, 1/(\lvert A\rvert - 5)$	$-\log_2(\frac{1}{2} \cdot \frac{3}{6} \cdot \frac{5}{12} \cdot \frac{1}{251}) = 11.2$ bits

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* was 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 was 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 [5]). The graph shows that the best compression is achieved when a maximum context length of 5 is chosen and that it deteriorates slightly when the context is increased beyond this.

This general behaviour 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 operate 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.

3. 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.

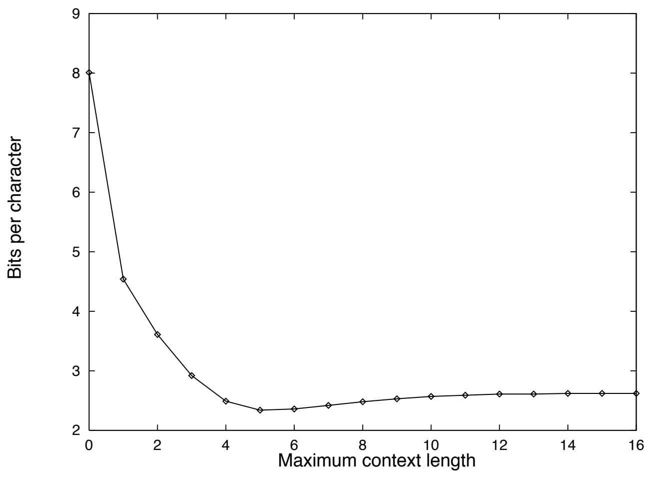

FIGURE 1. How the PPM compression ratio varies with maximum context length.

The results in Figure 1 show that there is a likely problem with extending the length of contexts in this way. Although there is more information available in the PPM* model than in any finite-context model, this is of no use if it cannot be employed effectively. The fact that the performance worsens beyond a certain order implies that the probability estimates are being computed in a sub-optimal way.

In the remainder of this paper, we use the following simple escape strategy based on the PPMC mechanism. 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 [13]. 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

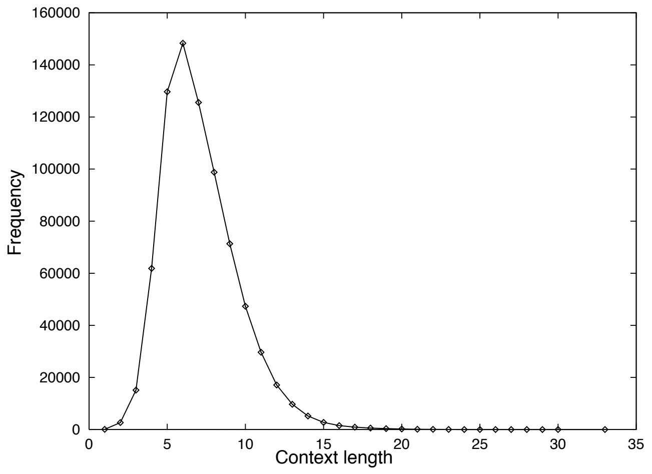

FIGURE 2. Histogram of the length of shortest deterministic contexts of file `book1`.

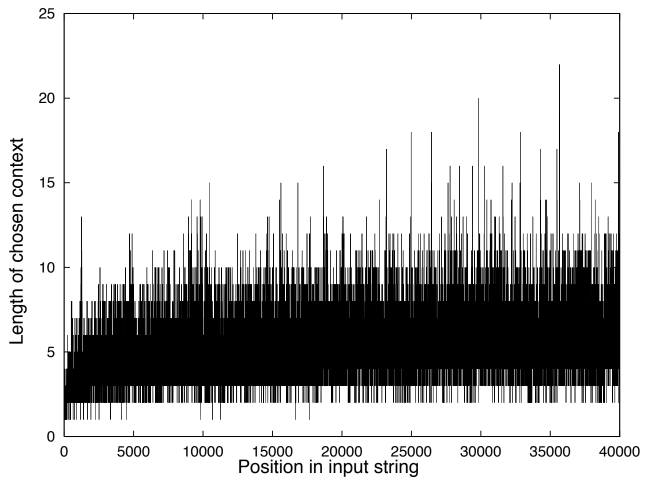

FIGURE 3. Length of the deterministic context at each character position of file `book1`.

longest context is chosen instead. The PPMC mechanism is then used, starting from this context.

A histogram of the context lengths chosen by this strategy is shown in Figure 2 for the file `book1`. The histogram peaks sharply at a context length of 5–6; not surprisingly the best context length (for this file) is 5 (Figure 1). Notwithstanding this peak, however, the length of the shortest deterministic context varies widely: Figure 3 plots it for the first 40 000 character positions in the file `book1`. The graph demonstrates that deterministic contexts much longer than 5 or 6 occur frequently and gradually increase in length as more input is seen.

A key problem associated with the use of unbounded contexts is the amount of memory and time necessary to maintain 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.

3.1. 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 (these correspond to the 'excited states' in the suffix-tree implementation of PPM and PPM* of Bunton [8]) with the longest context at the top. We call the resulting data structure a 'context trie'.

Figure 4 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 previously 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 to the start of the context. For example, looking to the very left of the tree, none of *a*, *ab*, *abr*, *abra* is 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 a 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 4, 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 points into the input string instead of to a lower node in the trie, then the pointer is redirected to a new node which in turn points to the same position in the input string. Both the original link and the context are updated to point to

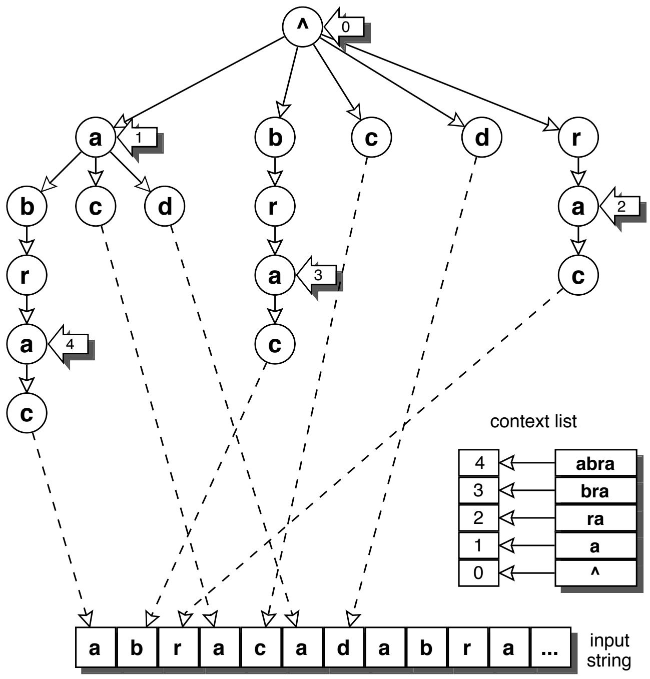

FIGURE 4. Context trie for the string *abracadabra*.

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 and their input pointers remain unchanged.

The second pair of cases corresponds 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 the input pointer copied from the parent node, and it is dropped from the context list. For example, in Figure 4, 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 a copy of the parent node's original input position, and the latter to the start position of the new context. For example, if first *c* and then *x* were added to Figure 4, the five *c* nodes at the leaves of the trie would each gain two children, an *a* child with a copy of the parent node's input pointer and an *x* child pointing to the start of each new context.

3.2. Implementation issues

3.2.1. *Using a PATRICIA-style trie*

Substantial space can be saved in the context trie by collapsing non-branching sub-paths into single nodes, just as in the standard PATRICIA trie data structure [14]. For each collapsed node, only one branch emanates from it. In Figure 4 there are three such paths, two with the letters *brac* and the third with *rac*. These collapsed sub-paths are shown in Figure 5.

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 starts 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.

3.2.2. *Memory requirements*

Using the collapsed-node representation of the PATRICIA trie, there will be at most $2N$ nodes in the trie for N inputs. There are at most N leaves as each leaf points to a unique position in the input, and each internal node points to at least two descendants, so that there are at most $N - 1$ internal nodes.

Bunton [8] showed how the techniques introduced in this paper can also be used to reduce the memory requirements of PPM implementations, thus making higher-order PPM implementations more feasible.

3.2.3. *Using a suffix trie*

As described, inserting the next symbol into a PATRICIA trie can take time up to O(N) (this worst case occurs when the input contains a string where the same symbol is repeated N times). As shown by Ukkonen [15], the addition of 'suffix

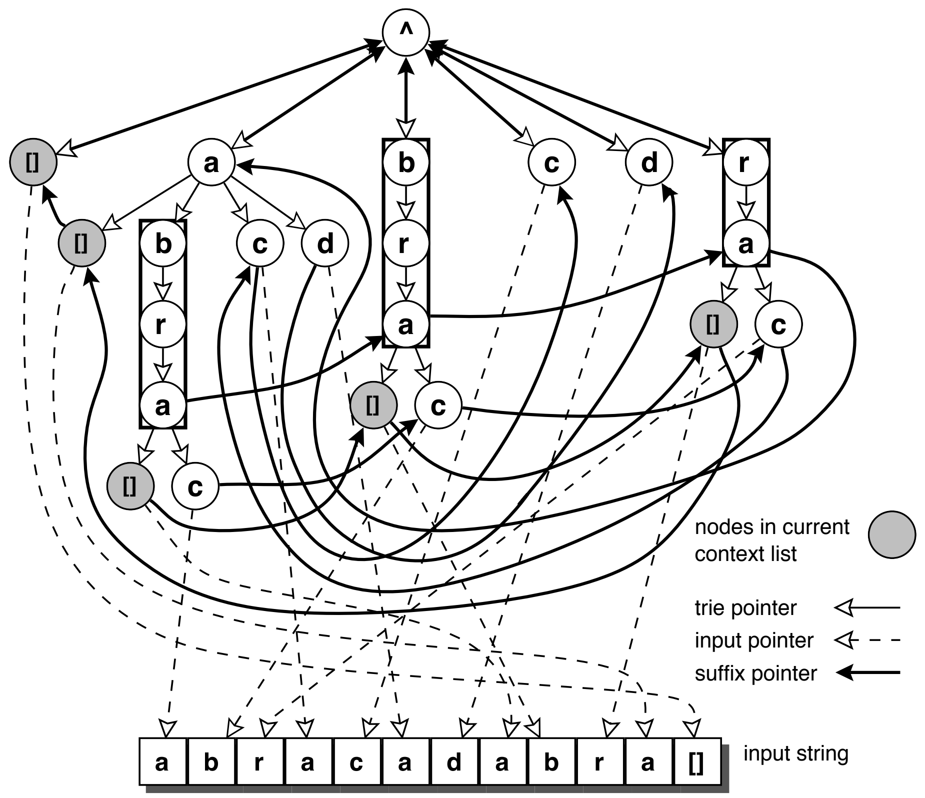

FIGURE 5. Suffix trie for the string *abracadabra*[].

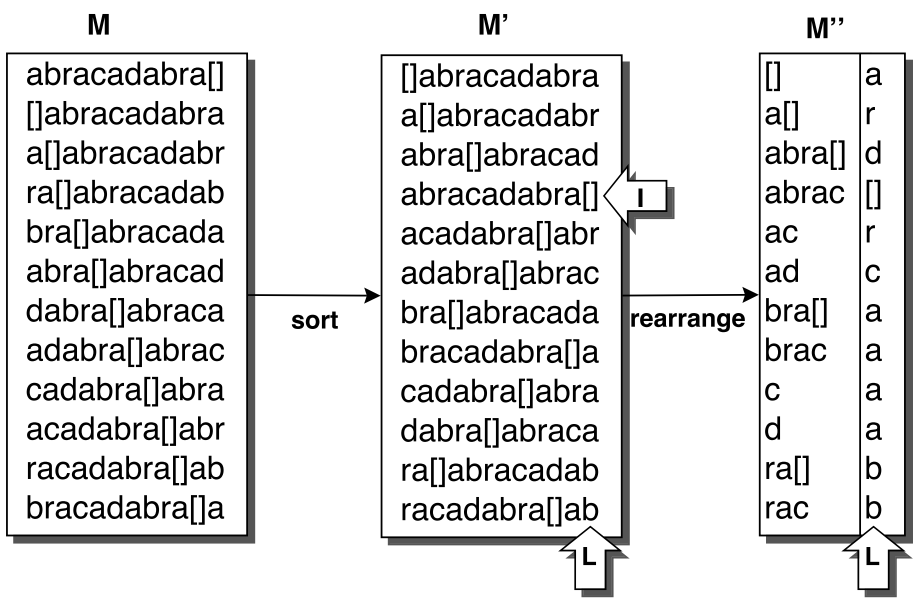

FIGURE 6. BW compression of the string *abracadabra*[].

pointers' to each node can provide an amortized O(1) insertion time. A suffix pointer points from the node representing string $s_1s_2\ldots s_n$ to the node representing the string with the leftmost symbol deleted: $s_2s_3\ldots s_n$. (That is, the suffix pointers point up the tree.) Figure 5 shows the tree of Figure 4 with collapsed nodes and suffix pointers included, and with the end of string symbol [] appended to the input string.

The nodes on the context list lie along a single chain of suffix pointers. Thus it is unnecessary to store the context list separately; a single pointer to the current longest context suffices. The use of a unique terminating symbol emphasizes the one-to-one correspondence between input positions and leaves and also corresponds more closely to the example of Figure 6.

TABLE 3. PPM*C model after processing the string *abracadabra*

Predictions			c	p	Predictions			c	p	Predictions			c	p
Order $k = 5$:					Order $k = 2$:					c	$\rightarrow$	a	1	$\frac{1}{2}$
abrac	$\rightarrow$	a	1	$\frac{1}{2}$	ab	$\rightarrow$	r	2	$\frac{2}{3}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{3}$	d	$\rightarrow$	a	1	$\frac{1}{2}$
					ac	$\rightarrow$	a	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$
Order $k = 4$:						$\rightarrow$	*Esc*	1	$\frac{1}{2}$	r	$\rightarrow$	a	2	$\frac{2}{3}$
$\Rightarrow$abra	$\rightarrow$	c	1	$\frac{1}{2}$	ad	$\rightarrow$	a	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	3	$\frac{1}{3}$
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$					
brac	$\rightarrow$	a	1	$\frac{1}{2}$	br	$\rightarrow$	a	2	$\frac{2}{3}$	Order $k = 0$:				
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{3}$	$\Rightarrow$	$\rightarrow$	a	5	$\frac{5}{16}$
					$\Rightarrow$ra	$\rightarrow$	c	1	$\frac{1}{2}$		$\rightarrow$	b	2	$\frac{2}{16}$
Order $k = 3$:						$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	c	1	$\frac{1}{16}$
abr	$\rightarrow$	a	1	$\frac{1}{2}$							$\rightarrow$	d	1	$\frac{1}{16}$
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$	Order $k = 1$:						$\rightarrow$	r	2	$\frac{2}{16}$
$\Rightarrow$bra	$\rightarrow$	c	1	$\frac{1}{2}$	$\Rightarrow$a	$\rightarrow$	b	2	$\frac{2}{7}$		$\rightarrow$	*Esc*	5	$\frac{5}{16}$
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	c	1	$\frac{1}{7}$					
rac	$\rightarrow$	a	1	$\frac{1}{2}$		$\rightarrow$	d	1	$\frac{1}{7}$	Order $k = -1$:				
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	3	$\frac{3}{7}$	$\Rightarrow$	$\rightarrow$	A	1	$1/\lvert A\rvert$
					b	$\rightarrow$	r	2	$\frac{2}{3}$					
						$\rightarrow$	*Esc*	1	$\frac{1}{3}$					

Larsson [16] shows that Ukkonen's algorithm can be extended so that only a window of the most recent M inputs is used. Given that there will be at most $2M - 1$ nodes, this permits the total memory used to be fixed precisely, which is of great importance for large files on small computers. Larsson's algorithm deletes the appropriate nodes as the input drops out of the window, and it has been shown to take amortized O(1) time for the deletions.

Unfortunately, the Ukkonen–Larsson algorithm ignores the need to update counts and to perform escape calculations. Because in the worst case these require a scan from the leaf to the root of the tree, they add an $O(N)$ component back into the computation time. It is unclear whether there exist effective probability estimation techniques that can avoid this $O(N)$ scan time.

The use of suffix pointers for PPM* was inspired by the work of Bunton [8, 17], which applies suffix trees as a common representation for a large class of compression techniques including PPM and PPM*.

3.2.4. *Encoding 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. A representation of the PPM* model with escape method C is shown in Table 3 after the string *abracadabra* has been processed. The contexts correspond to the nodes in the context trie of Figure 4, and are listed from the bottom up, moving from left to right. Each currently active context on the context list is marked by $\Rightarrow$ in the table. Prediction proceeds from the shortest deterministic context (i.e. only one prediction) on the list, if there is one, otherwise from the highest-order model at the head of the list ($k = 4$). In this case, there are three deterministic contexts (the contexts *abra*, *bra* and *ra*), so *ra* is chosen. If this 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 $\frac{1}{2}$. Sample encodings in this example are the same as for the PPMC model given in Table 2.

4. RESULTS

Table 4 shows the result of running PPM* on the Calgary corpus [5], along with the benchmark PPMC implementation which uses order 3, and a recently published, and extremely competitive, non-adaptive scheme labelled BW (Burrows and Wheeler [18]) (described in the following section). The best figure for each file is printed in bold. The present PPM*C implementation uses the context trie data structure with the escape algorithm C and the deterministic state-selection strategy.

Averaged over the entire corpus, PPM*C yields a 5.6% improvement over PPMC and a 3.7% improvement over BW. 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 than character based (e.g. `geo` and `pic`). It is interesting to note that of all the

TABLE 4. Compression ratios for the Calgary corpus

File	Size (bytes)	PPMC (bpc)	PPM*C (bpc)	BW (bpc)
bib	111 261	2.11	**1.91**	2.07
book1	768 771	2.48	**2.40**	2.49
book2	610 856	2.26	**2.02**	2.13
geo	102 400	4.78	4.83	**4.45**
news	377 109	2.65	**2.42**	2.59
obj1	21 504	**3.76**	4.00	3.98
obj2	246 814	2.69	**2.43**	2.64
paper1	53 161	2.48	**2.37**	2.55
paper2	82 199	2.45	**2.36**	2.51
pic	513 216	1.09	0.85	**0.83**
progc	39 611	2.49	**2.40**	2.58
progl	71 646	1.90	**1.67**	1.80
progp	49 379	1.84	**1.62**	1.79
trans	93 695	1.77	**1.45**	1.57
Weighted average		2.27	**2.09**	2.18
Average		2.48	**2.34**	2.43

compression schemes listed in [5], PPMC is the only one that ever outperforms either BW or PPM*C, and then only on a single file (obj1).

A number of experiments with much larger English texts than those in the Calgary corpus are reported in [13]. They show that execution speeds and compression performance for PPM* are comparable to higher-order PPM implementations (order 5 and above), although with greater memory requirements. A word-based variation of the algorithm described in [13] requires much less memory than word trigram models commonly used in speech recognition and machine translation, but still at the same time provides access to word-based contexts of unbounded length.

5. RELATED WORK

PPM* encodes in its model an explicit list of minimal strings that have occurred one or more times in the input. In a sense, this is all the information that can be extracted from the input. This information is sufficient to implement all compression techniques that have the finite-context property (that is, the probability estimate depends only on some finite length of preceding context). This includes a very wide range of compression techniques. DMC is a technique originally described in [19] and has been shown to be a finite-context model in [20].

The archetypical compression schemes that use unbounded contexts are LZ77 and LZ78, which parse the text into non-overlapping phrases from the input [3, 4]. These schemes are known to be asymptotically optimal for an ergodic source, although convergence is 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 [21], is not too far behind PPMC in compression performance (and greatly superior to it in speed).

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 [18]; we call this BW 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, BW 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 the end of string symbol).

Using the algorithm described in [18], first generate the matrix of strings M in Figure 6, then sort the 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 last column. In this example, $I = 4$ and $L = ard[]rcaaaabb$. 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 [18]. 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 highlighted and symbols not needed to form unique contexts suppressed. It is clear then that the unique strings in M'' have a one-to-one correspondence with the leaves of the trie in Figure 5. The characters in L are those that lie immediately before each of the unique contexts—thus BW can be seen as very similar to the process of predicting from right to left—always predicting the next character to the left and using the same contexts as PPM*.

In summary, whereas BW 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.

6. CONCLUSIONS

A new lossless compression mechanism, PPM*, has been described. Its major contribution is that it shows that the full information available by considering all substrings of the input string can be used effectively to generate high-quality predictions. The information in the PPM* model subsumes that used in many current high-quality models including the LZ family [3, 4], DMC [19] and BW [18]. Significant work remains to be done in effectively extracting this information, however. In another paper in this issue, Bunton [8] shows a number of significant improvements to the escape mechanism used in this paper, and further improves performance with a new information-theoretic state-selection technique. Åberg *et al.* [22] also investigate several ways of improving the escape probabilities in PPM.

Bloom [23] reports a result of 2.19 bpc for the Calgary corpus using an order bound of 8. There is clearly a need for a better understanding of what is possible; for example, a strong lower bound on the compression of a PPM* model would be very interesting.

Despite the foregoing, there are classes of information that are not taken into account by PPM* (or by other related techniques). There is evidence, for example, that in English text there are words which show strong recency effects [24]. If the word occurs once then there is a high probability that it will occur again soon. It is unclear, at this time, what models and estimation techniques can effectively take advantage of such information.

REFERENCES

[1] Cleary, J. G. and Witten, I. H. (1984) Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, **COM-32**, 396–402.

[2] Moffat, A. (1990) Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, **COM-38**, 1917–1921.

[3] Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, **IT-23**, 337–343.

[4] Ziv, J. and Lempel, A. (1978) Compression of individual sequences via variable rate coding. *IEEE Trans. Inform. Theory*, **IT-24**, 530–536.

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

[6] Moffat, A., Neal, R. and Witten, I. H. (1995) Arithmetic coding revisited. *Proc. Data Compression Conf.* IEEE Computer Society Press, Los Alamitos, CA.

[7] Witten, I. H., Neal, R. M. and Cleary, J. G. (1987) Arithmetic Coding for Data Compression. *Commun. ACM*, **30**, 520–540.

[8] Bunton, S. (1997) Semantically motivated improvements for PPM variants. *Comp. J.*, **40**, in press.

[9] Teahan, W. J. and Cleary, J. G. (1996) The entropy of English using PPM based models. *Proc. Data Compression Conf.*, pp. 53–62. IEEE Computer Society Press, Los Alamitos, CA.

[10] Shannon., C. E. (1951) Prediction and entropy of printed English. *Bell Syst. Tech. J.*, **30**, 50–64.

[11] Wilson, W. J. (1994) Chinks in the armor of public key cryptosystems. *Technical Report* 94/3, University of Waikato, Hamilton, New Zealand.

[12] Witten, I. H. and Bell, T. C. (1991) The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inform. Theory*, **IT-37**, 1085–1094.

[13] Teahan, W. J. (1997) *Modelling English Text*. D.Phil. Thesis, University of Waikato, New Zealand.

[14] Sedgewick, R. (1988) *Algorithms*. Addison-Wesley, Reading, MA.

[15] Ukkonen, E. (1995) On-line construction of suffix trees. *Algorithmica*, **14**, 249–260.

[16] Larsson, N. J. (1996) Extended application of suffix trees to data compression. *Proc. Data Compression Conf.*, pp. 190–199. IEEE Computer Society Press, Los Alamitos, CA.

[17] Bunton, S. (1996) *On-line Stochastic Process in Data Compression*. Ph.D. Thesis, University of Washington.

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

[19] Horspool, R. N. and Cormack, G. V. (1986) Dynamic Markov modelling—a prediction technique. *Proc. Int. Conf. on the System Services*, Honolulu, Hawaii.

[20] Bell, T. C. and Moffat, A. M. (1989) A note on the DMC data compression scheme. *Comp. J.*, **32**, 16–20.

[21] Fiala, E. R. and Green, D. H. (1989) Data compression with finite windows. *Commun. ACM*, **32**, 490–505.

[22] Åberg, J., Shtarkov, Y. M. and Smeets, B. J. M. (1997) Towards understanding and improving escape probabilities in PPM. *Proc. Data Compression Conf.* IEEE Computer Society Press, Los Alamitos, CA.

[23] Bloom, C. (1996) PPMZ. http://wwwvms.utexas.edu/~cbloom

[24] DeMori, R. and Kuhn, R. (1990) A cache-based natural language model for speech recognition. *IEEE Trans. Patt. Anal. Machine Intell.*, **PAMI-12**, 570–583.

[25] Cleary, J. G., Teahan, W. J. and Witten, I. H. (1995) Unbounded length contexts for PPM. *Proc. Data Compression Conf.*, pp. 52–61 IEEE Computer Society Press, Los Alamitos, CA.