PPM Performance with BWT Complexity: A New Method for Lossless Data Compression [*]

Michelle Effros
California Institute of Technology
effros@caltech.edu

Abstract

This work combines a new fast context-search algorithm with the lossless source coding models of PPM to achieve a lossless data compression algorithm with the linear context-search complexity and memory of BWT and Ziv-Lempel codes and the compression performance of PPM-based algorithms. Both sequential and nonsequential encoding are considered. The proposed algorithm yields an average rate of 2.27 bits per character (bpc) on the Calgary corpus, comparing favorably to the 2.33 and 2.34 bpc of PPM5 and PPM* and the 2.43 bpc of BW94 but not matching the 2.12 bpc of PPMZ9, which, at the time of this publication, gives the greatest compression of all algorithms reported on the Calgary corpus results page. The proposed algorithm gives an average rate of 2.14 bpc on the Canterbury corpus. The Canterbury corpus web page gives average rates of 1.99 bpc for PPMZ9, 2.11 bpc for PPM5, 2.15 bpc for PPM7, and 2.23 bpc for BZIP2 (a BWT-based code) on the same data set.

I Introduction

The Burrows Wheeler Transform (BWT) [1] is a reversible sequence transformation that is becoming increasingly popular for lossless data compression. The BWT rearranges the symbols of a data sequence in order to group together all symbols that share the same (unbounded) history or "context." Intuitively, this operation is achieved by forming a table in which each row is a distinct cyclic shift of the original data string. The rows are then ordered lexicographically. The BWT outputs the last character of each row (which is the character that precedes the first character of the given cyclic shift in the original data string). For a finite memory source, this ordering groups together all symbols with the same conditional distribution and leads to a family of low complexity universal lossless source codes [2, 3].

While the best of the universal codes described in [2, 3] converges to the optimal performance at a rate within a constant factor the optimum, the performance of these codes on finite sequences from sources such as text fails to meet the performance of

[*]M. Effros is with the Department of Electrical Engineering (MC 136-93) at the California Institute of Technology, Pasadena, CA 91125. This material is based upon work partially supported by NSF Award No. CCR-9909026.

some competing algorithms [4]. This failure stems in part from the fact that in grouping together all symbols with the same context, the BWT makes the context information inaccessible to the decoder.

Originally, the work described in this paper set out to modify the BWT-based codes of [2, 3, 4] in order to make the context information accessible to the decoder. What evolved, however, was effectively a variation on the Prediction by Partial Mappings (PPM) algorithms of [5, 6]. Thus the work presented here can be viewed either as a modification of the BWT to include context information or as a modification of PPM to allow for the computational efficiency of the BWT. The description that follows takes the latter viewpoint.

The PPM algorithm and its descendants are extremely effective techniques for sequential lossless source coding. In tests on data sets such as the Calgary and Canterbury corpora, the performance of PPM codes consistently meets or exceeds the performance of a wide array of algorithms, including techniques based on the BWT and Ziv-Lempel style algorithms such as LZ77 [7], LZ78 [8], and their descendants.

Yet despite their excellent performance, PPM codes are far less commonly applied than algorithms like LZ77 and LZ78. The Ziv-Lempel codes are favored over PPM-based codes for their relative efficiencies in memory and computational complexity [9]. Straightforward implementations of some PPM algorithms require $O(n^2)$ computational complexity and memory to code a data sequence of length n. While implementations requiring only $O(n)$ memory have been proposed in the literature [6, 10], the high computational complexity (and encoding time) of PPM algorithms, remains an impediment for their more widespread use.

This paper introduces a new context-search algorithm. While the proposed algorithm could also be employed in Ziv-Lempel and BWT-based codes, its real distinction is its applicability within PPM-based codes. The PPM code used is a minor variation on an existing PPM algorithm [10]. Our code achieves average rates of 2.27 and 2.14 bits per character (bpc), respectively, on the Calgary and Canterbury corpora and is efficient in both space and computation. The algorithm uses $O(n)$ memory and achieves $O(n)$ complexity in its search for the longest matching context for each symbol of a length-n data sequence. The $O(n)$ complexity is a significant improvement over the worst-case $O(n^2)$ complexity of a direct context search. Several variations on the given approach are presented. The variations include both sequential and non-sequential encoding techniques and allow the user to trade off encoder memory, delay, and computational complexity.

The remainder of the paper is organized as follows. Section II contains a review of PPM algorithms. The review focuses on PPM* [6] and its exclusion-based variation from [10]. A short introduction to suffix trees and McCreight's suffix tree construction algorithm [11] follows in Section III. McCreight's algorithm is used in implementations of Ziv-Lempel [12] and BWT [1] codes. The algorithm description is followed by a brief discussion of the difficulties inherent in applying McCreight's algorithm in PPM-based codes. Section IV describes a new method for combining PPM data compression with a new suffix-tree algorithm. Section V gives experimental results and conclusions.

II PPM Algorithms

The lossless compression algorithms in the PPM family of codes combine sequential arithmetic source coding (see, for example, [13], [14], or texts such as [9]) with adaptive Markov-style data models. Given a probability model $p(x^n)$ for symbols $x_1, \ldots, x_n$ in some finite source alphabet $\mathcal{X}$, arithmetic coding guarantees a description length $\ell_n(x^n)$ such that $\ell_n(x^n) < -\log p(x^n) + 2$ for all possible $x^n = (x_1, \ldots, x_n) \in \mathcal{X}^n$ [13]. Thus, given a good source model, arithmetic codes yield excellent lossless source coding performance.

A simple approach to source modeling is the Markov model approach. For any finite integer k, a Markov model of order k conditions the probability of the next symbol on the previous k symbols. Thus we code symbol x_n using the probability $p(x_n|x^{n-1}) = p(x_n|x_{n-k}^{n-1})$, where the string $x_{n-k}^{n-1} = (x_{n-k}, x_{n-k+1}, \ldots, x_{n-1})$ describes the "context" of past information on which our estimation of likely future events is conditioned.

In an adaptive code, the probability estimates are built using information about the previously coded data stream. Thus the encoder may update its probability estimates $p_n(a|s)$ for all $a \in \mathcal{X}$ and all $s \in \mathcal{X}^k$ at each time step n in order to better reflect its current knowledge of the source. The subscript n on $p_n(a|s)$ here makes explicit the adaptive nature of the probability estimate; $p_n(a|s)$ is the estimate of probability $p(a|s)$ at time n – just before the nth symbol is coded. The estimate $p_n(a|s)$ is based on the $n-1$ previously coded symbols in the data stream. Let $N_n(a|s)$ denote the number of times that symbol a has followed sequence $s \in \mathcal{X}^k$ in the previous data stream, where $N_n(a|s) = \sum_{i=k+1}^{n-1} 1(x_{i-k}^i = sa)$ for each $a \in \mathcal{X}$. If the probability model $p_n(a|s)$ relies only on the conditional symbol counts $\{N_n(a|s) : a \in \mathcal{X}, s \in \mathcal{X}^k\}$ and if the decoder can sequentially decipher the information sent to it, then the decoder can track the changing probability model p_n by keeping a tally of symbol counts identical to the one used at the encoder.

PPM source models generalize adaptive Markov source models by replacing the *single* Markov model of fixed order k by a *collection* of Markov models of varying orders. For example, given some finite memory constraint M, the original PPM algorithm uses Markov models of all orders $k \in \{-1, 0, \ldots, M\}$, where the order $k = -1$ model refers to a fixed uniform distribution on all symbols $x \in \mathcal{X}$. (Typical values of M for text compression satisfy $M \leq 6$ [5, 15].) PPM uses "escape" events to combine the prediction probabilities of its $M + 1$ Markov models. The escape mechanism is employed on symbols that have not previously been seen in a particular context. Let Esc denote the escape character, and use $\mathcal{Y}$ to denote the modified alphabet $\mathcal{X} \cup \{Esc\}$. Use $N_n(Esc|s)$ to denote the number of distinct symbols that have followed context s, which equals the number of times an escape was used in the given context. PPM builds its probability estimate for symbol a at time n recursively. It begins by finding the longest context of the given symbol that has previously appeared in the data stream. It then uses the escape character to back off to lower order models if necessary to find a model in which symbol a is not novel in the given context.

More precisely, for each $a \in \mathcal{Y}$ and each $k \in \{0, \ldots, M\}$ let

$$p(n,k,a) = \frac{N_n(a|x_{n-k}^{n-1})}{\sum_{b \in \mathcal{Y}} N_n(b|x_{n-k}^{n-1})},$$

and define

$$p(n,-1,a) = \frac{1}{|\mathcal{X}|},$$

where $|\mathcal{X}|$ is the size of alphabet $\mathcal{X}$. Define $K(n)$ to be the length of the largest context for symbol n that has previously appeared in the given data stream. For each $a \in \mathcal{X}$ let $k(n,a)$ be the largest $k \in \{0, \ldots, K(n)\}$ such that $N_n(a|x_{n-k}^{n-1}) > 0$. The initial context index $K(n)$ equals zero if the context of symbol n has not previously appeared in the data stream; the final context index $k(n,a)$ is set to -1 if symbol a has not previously appeared in the given data string. PPM uses probability model

$$p_n(a|x^{n-1}) = p_n(a|x_{n-K(n)}^{n-1}) = p(n, k(n,a), a) \prod_{k=k(n,a)+1}^{K(n)} p(n, k, Esc).$$

This model is inefficient in its normalization of $p(n,k,a)$ for $k(n,a) \leq k < K(n)$. By describing an escape character in model k, the encoder tells the decoder that the observed symbol satisfies the equation $N_n(a|s) = 0$. As a result, both the encoder and the decoder may remove from their low order probability estimates all symbols $a \in \mathcal{X}$ such that $N_n(a|s) > 0$. Modifying the normalizing constants accordingly improves system performance, and thus this modification is assumed in the work that follows.

PPM* [6] modifies PPM by removing the a priori memory constraint M to allow for unbounded contexts. Models are kept for *all* contexts of *all* lengths that have previously appeared in the data stream. In choosing its initial context, PPM* uses the shortest matching deterministic context, where a context is deterministic if the number of symbols that have previously appeared in that context is exactly equal to one. If no such matching deterministic context exists, then PPM* uses the longest matching context instead.

In [10], Bunton considers a collection of variations on PPM*. One of those variations uses the "update exclusion" mechanism of [15]. The update exclusion method replaces count $N_n(a|s)$ in the calculation of $p_n(a|s)$ with a modified count $N'_n(a|s)$. While N increments by one each time symbol a is seen in context $s \in \mathcal{X}^*$, N' increments by one *only* if symbol a is either novel in context s or s is a suffix of some longer context $t \in \mathcal{X}^*$ such that $|t| = |s| + 1$ and a is novel in context t.

III Suffix Trees

In the original PPM* algorithm [6], the contexts are stored in a "context trie." In [10], the contexts are stored in a suffix tree. This paper follows an approach closer to that of the latter work. A suffix tree for data sequence $x^n = (x_1, x_2, \ldots, x_n)$ describes all suffixes of x^n. We denote those suffixes by $\{s_i\}_{i=1}^n$, where $s_i = x_i^n = (x_i, \ldots, x_n)$. Using $\diamond$ to denote a unique end-of-string character, the suffix tree for a fixed string

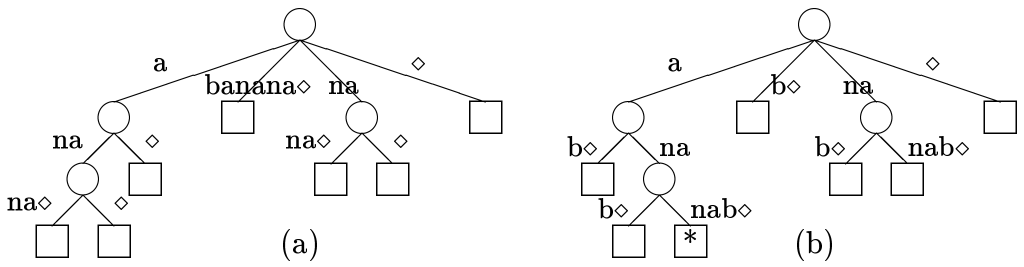

Figure 1: (a) A suffix tree for the string banana◇. (b) A suffix tree for ananab◇, or, equivalently, a "prefix tree" for ◇banana.

$x^n\diamond$ is uniquely determined by three properties [11]. (1) Each tree arc may represent any finite nonempty string $s \in \mathcal{X}^*$. (2) Each internal tree node except for the root must have at least two children. (3) The strings on sibling arcs must begin with different characters. The suffix tree for *banana*◇ appears in Figure 1(a).

A suffix tree on a string of length n has n leaves and no more than $2n$ nodes, giving a linear storage requirement. Each arc string, described by its start and end points in the original data sequence, is stored at the node to which the given arc descends. In the implementation of [11], each node contains a single "sibling" pointer and a single "descendant" pointer to link it to the rest of the tree.

Suffix trees are very popular data structures for use in lossless codes. In particular, suffix tree implementations of both Ziv-Lempel style codes (e.g., [12]) and BWT-based codes [1] appear in the literature. In Ziv-Lempel codes, suffix trees may be used to find longest matches in the previously coded data stream. In fact, the suffix tree for x^n records *all* matches of *all* lengths in x^n. Each match is an internal node of the tree. The number of times that the match occurs in x^n equals the number of leaves descending from that node. For example, from the context tree in Figure 1(a) we learn that the substring "a" appears three times in the data string while the substrings "ana" and "na" each appear twice.

To understand the relationship between suffix trees and the BWT, suppose that the tree arcs descending from each node in the tree are ordered alphabetically – with branch "a" falling to the left of branch "b◇" and so on. Assuming this lexicographic ordering of tree arcs, note that the suffix tree gives a lexicographic ordering of all suffixes of the string x^n. Note further that, given the existence of the end-of-string symbol, the lexicographic ordering of all suffixes is equivalent to the lexicographic ordering of all cyclic shifts of the original data stream. As a result, if we modify the suffix tree of Figure 1(a) by adding the symbol x_{i-1} to the leaf corresponding to suffix $s_i = (x_i, \ldots, x_n)$, then performing the BWT on x^n is equivalent to reading off the x_{i-1} values from left to right through the tree. This similarity to the BWT is not coincidental. Both PPM* and the BWT rely on the same unbounded contexts in building probability models for use in adaptive arithmetic coding. The key difference between these two algorithms is that PPM* manages to preserve context information in a manner that is sequentially available to both encoder and decoder, while the BWT makes this information inaccessible to the decoder (except through explicit or implicit context description as in some of the algorithms of [2, 3]).

The suffix tree implementation of PPM* given in [10] is closely related to the context trie approach of [6], and the memory required by both algorithms grows linearly in n. At time n, the encoder stores both the tree structure associated with all contexts $s_1, s_2, \ldots, s_{n-1}$ and a list of $K(n)+1$ pointers. These $K(n)+1$ pointers point to the nth symbol's contexts of lengths $0, 1, \ldots, K(n)$, all of which are represented by either states or "virtual states" in the context tree. While the contexts are suffixes of each other, there is no natural order to their locations in the context tree.

In [11], McCreight introduces an $O(n)$ complexity technique for suffix tree construction. The $O(n)$ complexity is a huge savings over the $O(n^2)$ worst-case complexity of direct tree construction. Unfortunately, McCreight's technique is not directly applicable to PPM*, as the following discussion illustrates.

In [11], suffix tree construction is performed by adding suffixes to the tree one by one in order of decreasing length. Thus in the example of Figure 1(a), the suffix "banana$\diamond$" would be added to the tree first, followed by the suffix "anana$\diamond$", followed by the suffix "nana$\diamond$" and so on. While each subsequent suffix could be added to the tree by simply starting at the root and searching down for the longest match, the time required by this straightforward approach is superlinear. The two key observation used in [11] are:

- Context s_i differs from context s_{i+1} in only its first character ($s_i = x_i s_{i+1}$).
- The longest match for s_i is known in searching for the longest match for s_{i+1}.

Together, these two observations imply that if the longest match for s_i in the previous data string is $x_i s$ for some $s \in \mathcal{X}^*$, then the longest match for suffix s_{i+1} must begin with string s. Combining this observation with an efficient approach for extending s to find the full match for s_{i+1} yields a linear tree construction algorithm.

The efficiency of McCreight's suffix tree construction technique results from its use of information about s_{i-1} in adding suffix s_i to the tree. An unfortunate consequence of this approach, however, is that suffix tree construction using McCreight's algorithm requires access to the complete data string. While the complete data string could be made available to PPM*'s encoder if we assume a nonsequential code, the decoder does not have the full data string during suffix tree construction. In fact, the decoder cannot decode the binary source description without access to at least part of the suffix tree. As a result of this predicament – that the decoder needs the data sequence to get the suffix tree and needs the suffix tree to get the data sequence – McCreight's algorithm cannot be directly applied to PPM*.

IV Algorithm

The algorithm considered here is a memory- and computation-efficient implementation of PPM*. Like the implementation of [10], the implementation given here uses suffix trees for storing and computing the information used in PPM*'s probability model. Unlike the prior implementation, however, the algorithm given here relies on a suffix tree of the *reversed* data stream or, equivalently, a "prefix tree" for the data stream under consideration. (Note, again, the similarities between this approach and the BWT approach, where string reversal is often employed prior to performance of the transform.) A suffix tree for the reversed data string *ananab*$\diamond$, which is also a prefix tree for the data string $\diamond$banana, appears in Figure 1(b).

Prefix trees are useful for the following property. All contexts of a given symbol are nested subsets of a single branch through the prefix tree. For example, suppose that we are constructing a prefix tree as we code each subsequent symbol of a data string x^n, and suppose that the symbols seen so far are "banana". Then the prefix tree available for coding the next symbol is the prefix tree of Figure 1(b). The contexts for this symbol are: "a", "na", "ana", "nana", "anana", and "banana". All of these contexts are represented along the branch whose terminating leaf is labeled by a * in Figure 1(b). The reversed contexts ("a", "an", "ana", "anan", "anana", and "ananab") label the partial substrings of this branch. Using this approach, we can replace the earlier list of context pointers by a single context pointer pointing to the longest context of interest in coding the nth symbol.

As discussed in the previous section, the use of suffix (or, equivalently, prefix) trees in implementing the probability models in PPM* is motivated by their space-efficiency. The goal here is to develop a fast suffix tree construction algorithm that can be applied directly within PPM*. Unfortunately, the order in which PPM* makes the suffixes of the reversed data string available for addition to the suffix tree is exactly opposite the order in which strings are added to the suffix tree in McCreight's algorithm. More specifically, in PPM*, symbols become available to the decoder sequentially, where symbol x_i can be decoded only after the context tree based on symbols $x_1, \ldots, x_{i-1}$ has been constructed. Thus using p_i to denote the ith prefix $p_i = (x_1, \ldots, x_i)$ of string x^n, PPM* requires an algorithm for adding suffixes (of the reversed data stream) in order from shortest to longest – that is order $p_1 = s_n = (x_1), p_2 = s_{n-1} = (x_1, x_2), \ldots, p_n = s_1 = (x_1, \ldots, x_n)$. In contrast, McCreight's algorithm adds the longest suffix first and then follows with shorter and shorter suffixes of that original suffix – giving order $s_1, s_2, \ldots, s_n$. The following observations parallel the observations on which McCreight's algorithm is built.

- Prefix p_i differs from prefix p_{i-1} by only its last character $p_i = p_{i-1}x_i$. Thus the longest context for symbol x_{i+1}, here denoted by $\mathrm{suf}_i(p_i)$, is at most one character longer than the longest context for symbol x_i and can rely on no characters prior to those found in the context for x_i. More precisely, $\mathrm{suf}_i(p_i) = \mathrm{suf}_i(p_{i-1}x_i) = \mathrm{suf}_i(\mathrm{suf}_{i-1}(p_{i-1})x_i)$.

- Since we add the prefixes to the tree in order $p_1, p_2, \ldots$, the longest match for prefix p_{i-1} is known prior to the search for the longest match for prefix p_i.

A strategy similar to that of [11] exploits these observations to decrease the complexity associated with searching for the longest context for each subsequent symbol. The basic idea is to add auxiliary links to the data structure. These links provide "shortcuts" in the search for the longest match. A description of the algorithm follows.

The tree is initialized to a single node corresponding to the length-0 (empty) string. The initial tree is labeled T_0. The suffix tree is then built up one leaf at a time, by adding one prefix at each time step. After the $(i-1)$th addition, the tree contains prefixes $p_1, p_2, \ldots, p_{i-1}$ and is called T_{i-1}. Associated with every node in the tree except for the root and the most recently added leaf (p_{i-1}) is an array of "short-cut" pointers. The number of elements in a short-cut array equals the number of symbols seen so far in the corresponding context. The short-cut pointer at node s for symbol a points to the tree node corresponding to context sa and is created the first time symbol a appears in context s. (Short-cut pointers are not necessary at the root since the symbol-a short-cut pointer from the root would always point from the root to the root's child (if any) labeled by "a".)

At time $(i-1)$, the algorithm visits $\mathrm{suf}_{i-1}(p_{i-1})$ in order to add the leaf for prefix p_{i-1}. Thus the algorithm for step i assumes node $\mathrm{suf}_{i-1}(p_{i-1})$ as its starting point. At step i, the PPM* encoder describes symbol x_i using tree T_{i-1} and then adds prefix p_i to the tree. In making this addition, the algorithm begins at node $\mathrm{suf}_{i-1}(p_{i-1})$, (which is the longest possible context for symbol x_i). If symbol x_i has previously appeared in context $\mathrm{suf}_{i-1}(p_{i-1})$, then $\mathrm{suf}_{i-1}(p_{i-1})$ has a short-cut pointer for symbol x_i. Traversing this pointer leads to node $\mathrm{suf}_i(p_i) = \mathrm{suf}_{i-1}(p_{i-1})x_i$. If x_i has not previously appeared in context $\mathrm{suf}_{i-1}(p_{i-1})$ then the algorithm considers a context one symbol shorter and looks for a context pointer there. This procedure continues until the algorithm either finds and traverses a short-cut pointer or ends up at the root. The final node in this procedure is an extension of $\mathrm{suf}_i(p_i)$. The algorithm creates a node for $\mathrm{suf}_i(p_i)$ if necessary and then adds leaf p_i to $\mathrm{suf}_i(p_i)$. Each context of x_i visited along the way is given a short-cut pointer pointing to the new leaf. Since symbol x_i is novel in each such context, the counts $N'(x_i|s)$ are incremented for each of these nodes and each of their parents.

Let k_i denote the number of contexts of x_i visited in searching for $\mathrm{suf}_i(p_i)$. Finding the complexity of the tree construction algorithm is equivalent to finding $\sum_{i=1}^n k_i$. Repeated application of the equality $|\mathrm{suf}_i(p_i)| = |\mathrm{suf}_{i-1}(p_{i-1})x_i| - k_i$ accomplishes that goal, giving finally $\sum_{i=1}^n k_i = n - |\mathrm{suf}_n(p_n)| \leq n$. Thus in total at most n nodes must be visited in the search for $\mathrm{suf}_i(p_i)$ given $\mathrm{suf}_{i-1}(p_{i-1})$, giving $O(n)$ complexity.

The resulting algorithm is $O(n)$ in memory; the constant here is larger than that of [10] due to the additional pointers. The motivation for the pointers is the reduction, from (worst-case) $O(n^2)$ to $O(n)$, in the complexity associated with finding the longest context. The same approach could be used in suffix-tree construction for other algorithms as well, including, for example, PPM.

The given context tree design algorithm may be applied in a number of different ways to yield a PPM* variant using probability estimates based entirely on the update exclusion counters $N'(a|s)$. Sequential and non-sequential methods are proposed here. The sequential approach uses the above algorithm in both its encoder and its decoder; thus the sequential codes' encoder and decoder use the same procedure to independently update the tree with each subsequent symbol. A non-sequential approach, using McCreight's algorithm at the encoder and the above algorithm at the decoder is also proposed. The algorithms differ in their memory and computational complexity, but both algorithms are $O(n)$ in both the memory required to store the model and the number of computations needed to find the longest context of an incoming data symbol. Note that the given complexity describes only the complexity

Calgary Corpus				
File	B97 (bpc)	NEW (bpc)	PPM* (bpc)	BW94 (bpc)
bib	1.79	1.84	1.91	2.07
bk1	2.18	2.39	2.40	2.49
bk2	1.86	1.97	2.02	2.13
geo	4.46	4.75	4.83	4.45
news	2.29	2.37	2.42	2.59
obj1	3.68	3.79	4.00	3.98
obj2	2.28	2.35	2.43	2.64
ppr1	2.25	2.32	2.37	2.55
ppr2	2.21	2.33	2.36	2.51
pic	0.78	0.87	0.85	0.83
prgc	2.29	2.34	2.40	2.58
prgl	1.55	1.59	1.67	1.80
prgp	1.53	1.56	1.62	1.79
trns	1.33	1.38	1.45	1.57
AVG	2.18	2.27	2.34	2.43

Canterbury Corpus				
File	PPMZ9 (bpc)	NEW (bpc)	PPM7 (bpc)	BZIP2 (bpc)
text	2.08	2.18	2.26	2.27
fax	0.79	0.87	0.94	0.78
csrc	1.87	1.95	2.08	2.18
excl	1.01	1.49	0.97	1.01
sprc	2.45	2.67	2.58	2.70
tech	1.83	1.95	2.01	2.02
poe	2.22	2.40	2.46	2.42
html	2.19	2.29	2.35	2.48
lisp	2.25	2.33	2.43	2.79
man	2.87	2.94	3.01	3.33
play	2.34	2.47	2.55	2.53
AVG	1.99	2.14	2.15	2.23

Table 1: Compression results on the Calgary and Canterbury corpora.

associated with finding the longest matching context but does not include the cost of moving from the longest context to the shortest deterministic context in PPM*.

V Results and Conclusions

This paper introduces a new implementation of PPM* with update exclusions that reduces the worst-case $O(n^2)$ complexity of the tree update mechanism to $O(n)$. The new algorithm maintains the $O(n)$ memory of earlier PPM* algorithms but increases the constant in that term. Table 1 shows the rate results achieved by the proposed algorithm (labeled as "new") as compared to those of a variety of alternative algorithms. The results for competing algorithms on the Calgary corpus are quoted from [6] and [10] (B97). The results on the Canterbury corpus are quoted from the Canterbury corpus web page. While the results given in Table 1 are by no means exhaustive, they give a reasonable picture of how the performance of the proposed algorithm compares to current alternatives. While the proposed approach is not the best possible algorithm in rate performance, it surpasses both PPM* and the BWT-based codes in compression capabilities using only $O(n)$ complexity in the tree design.

References

[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC 124, Digital Systems Research Center, Palo Alto,

CA, May 1994.

[2] M. Effros. Universal lossless source coding with the Burrows Wheeler transform. In *Proceedings of the Data Compression Conference*, pages 178–187, Snowbird, UT, March 1999. IEEE.

[3] M. Effros. Universal lossless source coding with the Burrows Wheeler transform. 1999. Submitted to the *IEEE Transactions on Information Theory* June 28, 1999.

[4] M. Effros. Theory meets practice: universal source coding with the Burrows Wheeler transform. In *Proceedings of the Allerton Conference on Communication, Control, and Computing*, Monticello, IL, September 1999. IEEE.

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

[6] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In *Proceedings of the Data Compression Conference*, pages 52–61, Snowbird, UT, March 1995. IEEE Computer Society.

[7] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23:337–343, May 1977.

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

[9] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, New Jersey, 1990.

[10] S. Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2/3):76–93, 1997.

[11] E. M. McCreight. A space-economical suffix tree construction algorithm. *Jornal of the ACM*, 23(2):262–272, April 1976.

[12] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the Association for Computing Machinery*, 28(1):16–24, January 1981.

[13] F. Jelinek. Buffer overflow in variable length coding of fixed rate sources. *IEEE Transactions on Information Theory*, 14:490–501, May 1968.

[14] J. Rissanen and G. G. Langdon Jr. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, March 1979.

[15] A. Moffat. Implementating the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.