Transactions Letters

Implementing the PPM Data Compression Scheme

ALISTAIR MOFFAT

Abstract—**The "Prediction by Partial Matching" (PPM) data compression algorithm developed by Cleary and Witten is capable of very high compression rates, encoding English text in as little as 2.2 bits/character. Here it is shown that the estimates made by Cleary and Witten of the resources required to implement the scheme can be revised to allow for a tractable and useful implementation. In particular, a variant is described that encodes and decodes at over 4 kbytes/s on a small workstation, and operates within a few hundred kilobytes of data space, but still obtains compression of about 2.4 bits/character on English text.**

I. Introduction

THE "Prediction by Partial Matching" data compression scheme, developed by Cleary and Witten in 1984 [2] is capable of very good compression on a wide variety of source data. The adaptive nature of the scheme, and the flexibility afforded by arithmetic coding, mean that an effective compression model will be built for any input file that is reasonably homogeneous. Cleary and Witten reported that their scheme was capable of representing English text in as little as 2.2 bits/character. In comparison to other compression methods this is very good; for example, using a zero order character model English text will require about 4.7 bits/character, and the Ziv-Lempel [12] algorithms will typically require about 3.3 bits/character.

The drawback of the PPM method is its resource requirements. Cleary and Witten estimated that their scheme could at best be expected to encode and decode at about 1000 characters/s on a VAX 11/780, and might require more than a megabyte of memory when coding a large file. This high resource requirement has made the method unattractive, and the current state of the art for practical use is still the Ziv-Lempel schemes, of which the Unix utility *Compress* is a good example [9], [10].

The estimates of Cleary and Witten were, however, unnecessarily pessimistic. In what follows, it is shown that by trading compression efficiency for running time and data space a practical PPM compression scheme can be implemented. This version runs about three times faster than those first estimates, and operates effectively within a few hundred kilobytes of data space to obtain compression only marginally inferior to that produced by the original PPM. PPM is a tractable technique for high performance data compression.

II. Prediction by Partial Matching

One of the most effective ways to predict symbols, and hence to obtain compression, is to bias the predictions according to the most recent symbols seen. For example, the phrase "One of the most effective ways to predict symbols, and hence to obtain compressio" almost uniquely predicts the next symbol in the stream to be "n." As a general rule, and provided that accurate statistics are known, the higher the order of the model the more accurate will be the predictions, and the better the compression. Using the same example, in the shorter context "sio" a number of other symbols besides "n" must also be regarded as possible, such as in "excelsior" and "physiotherapy."

TABLE I
EXAMPLE CONTEXT SYMBOL COUNTS

order	context	symbol occurrence counts							total
		n	t	u	d	b	c	a	
3	sio	9	2						11
2	io	10	2	3					15
1	o	15	6	18	25	1			65
0	*empty*	30	15	41	68	24	72	32	282

Unfortunately, the sheer magnitude of the sample space for predictions using a long context makes them almost impossible to manage for practical compression. Even restricting the context to 4 prior characters will mean (using typical 8 bit bytes) that there are in excess of 4 billion contexts possible. The alternative is to make the scheme adaptive. Then statistics will be built up as the stream of symbols is processed, without the need for a large model to be stored or transmitted beforehand. This will reduce the space requirements, but does mean that during the initial part of the transmission, while the model is learning the distribution of conditional probabilities, the coding will be relatively inefficient. On the other hand, low-order adaptive models are quick to establish useful statistics, but over a long text will only attain limited compression.

This dilemma is solved in the PPM scheme by using an adaptive model based on a variable length context. At each coding step the longest previously encountered context is used to predict the next character. If the symbol is novel to that context, an escape code is transmitted and the context shortened by dropping one symbol. This process of transmitting an escape code and then shortening the context will continue until the symbol is successfully transmitted. If the current symbol is novel even to the zero order context then a final escape will be transmitted, and the symbol will be encoded as an 8 bit code. The adaptive model can then add the current symbol to all applicable contexts, thereby boot-strapping itself into longer contexts. Cleary and Witten give a full description of this process and give examples. The actual coding of a symbol, given its predicted probability, is performed almost optimally by arithmetic coding. Witten, Neal, and Cleary [11] give a detailed description of arithmetic coding.

By way of example, Table I shows the occurrence counts that may have accumulated at some stage during the encoding process.

Cleary and Witten discussed two methods for calculating symbol and escape probabilities. Their method A allows a count of one for the escape code, and correspondingly inflates the total count for the context by one. For example, using method A in the above example gives $p_A(\text{"n"} \mid \text{"sio"}) = (9/12)$, and $p_A(\textit{escape} \mid \text{"sio"}) = (1/12)$ where $p_M(c \mid \textit{string})$ is the conditional probability assigned to symbol c by method M in the context given by *string*, and $p_M(c)$ is the overall probability assigned to symbol c by method M. Method B created space for the escape event by subtracting one from each count and retaining the same total, so that $p_B(\text{"n"} \mid \text{"sio"}) = (8/11)$ and $p_B(\textit{escape} \mid \text{"sio"}) = (2/11)$.

Should the escape be transmitted while in the context "sio," both "n" and "t" can be excluded from consideration in the context

Paper approved by the Editor for Coding Theory and Applications of the IEEE Communications Society. Manuscript received August 8, 1988; revised December 18, 1989.

The author is with the Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia.

IEEE Log Number 9039069.

"io," and increased probabilities can be allocated to the remaining symbols, making the coding more efficient. Thus, $p_A(\text{"}u\text{"} \mid \text{"}io\text{"}) = (3/4)$ and $p_B(\text{"}u\text{"} \mid \text{"io"}) = (2/5)$. This process of exclusion will be discussed further below. The overall probability assigned to each symbol is calculated by including the escape probabilities, so that

$$p_A(\text{"}a\text{"}) = (1/12) \times (1/4) \times (1/27) \times (32/105), \text{ and}$$

$$p_B(\text{"}a\text{"}) = (2/11) \times (3/5) \times (5/29) \times (31/132).$$

Note that, using method B, symbol "b" cannot be encoded using the first-order context (it has only occurred once) and so cannot be excluded from the zero order context.

Method A would generate a 12.05 bit code for "a," and method B would generate a 7.82 bit code. This does not mean that method B is better: method A requires 2.58 bits for "t," while method B generates 3.46 bits. Which is actually better depends on which escape mechanism best captures the "true" behavior of the data. This will be discussed further below.

III. Implementing PPM

Central to all of the PPM implementations was a digital search tree, or trie [5]. Each list of the trie represented one context, tracking the number of times that context had been encountered. Each node in the list recorded the number of times a particular symbol had occurred in that context, and was itself the head node for a corresponding list representing a context one symbol longer. In simplest form, each node of the trie required 18 bytes—three 4 byte pointers and three 2 byte integers. The pointers recorded the next node in the same context list, the first node in the "child" context set, and the root node of the context set one symbol shorter. For example, the node for "sio" would have pointers to the node for perhaps "sig," the node for perhaps "sion," and the node for "io," respectively. The final pointer (the vine pointer, looping back up the tree) was included to allow speedy context dropping after the transmission of an escape.

The three integers in a node were used to record the symbol represented at that node, the number of times that symbol had appeared in its parent's context set, and the number of times it had been used as a context set. Two distinct values were needed as it was necessary to keep the occurrence counts bounded, and the count scaling was applied to individual lists when required, not to the whole trie.

This simplest trie was only used for method A coding. For method B another count field was added, recording the length of the corresponding list. This enabled the escape probability to be known without scanning the entire context list.

At each stage of the coding a "hand" of pointers into the trie was maintained, with one finger of the hand pointing at each of the current contexts. The context list of the deepest finger of the hand, representing the longest permitted context, was searched first, then, after the transmission of an escape code, the second longest, and so on, until either the symbol was found or even the list indicated by the root finger (the thumb) had failed to find the symbol. In this latter case, the symbol was finally transmitted as an unweighted 1 in 257 code, with the 257th symbol reserved for an end of file marker.

While the list pointed at by the deepest finger (i.e., the longest permitted context) of the hand was being scanned there was no need to allow for exclusions, and so in fact two different searching procedures were used: when it was known *a priori* that no symbols had been excluded a successful search could terminate without scanning the remainder of the list. Both versions of the list searching used a "move-to-front" policy to reduce the searching time.

The symbols excluded at any stage were recorded by setting a bit in an array. The final step in processing each symbol was to reset this array, and there was a marked speed advantage in only resetting bits that had been set, by scanning the appropriate context list again rather than reinitializing the whole array to zero.

In the case when the majority of the symbols were being successfully predicted in the maximum permitted context, the processing of each symbol involved only a search in a relatively short move-to-front linked list, with early termination when the symbol was found; an arithmetic coding step using the parameters generated; and then resetting the fingers of the hand using the sequence of vine pointers starting at the node found. In this case it was expected that the coding would not be significantly slower than comparable zeo order methods, which must also undertake these same basic steps, including searching in a longer list.

TABLE II
SPACE REQUIRED AND PREDICTION EFFECTIVENESS

	file type	order 1	2	3	4	5
trie nodes	VAX object	4470	12012	21819	32882	44743
	English text	1330	8265	27374	63461	116950
predictions	VAX object	74%	53%	40%	32%	27%
	English text	99%	95%	86%	74%	61%

TABLE III
COMPRESSION PERFORMANCE OF METHODS A AND B

		order 1	2	3	4	5
compression (bits per char)	method A	3.50	2.81	2.55	2.50	2.51
	method B	3.51	2.82	2.54	2.47	2.46
throughput (kbyte per sec)	method A	4.08	3.50	2.74	2.26	–
	method B	3.75	3.04	2.44	1.96	–

The programs were written in C and run on a Sun-3/50 under BSD4.2 Unix. Each program required about 1000 lines of code, and so they were relatively compact. The experiments used a suite of nine test files, including a VAX object file, spreadsheet and database documents from an IBM PC, a transcript of an edit-compile-run terminal session, C source code, and English text in several forms, with and without embedded formatting commands. File sizes in the suite ranged from 16384 bytes to 139521 bytes, and they totalled 400325 bytes. Each 8-bit byte of the file was taken to be a symbol for compression purposes.

A. Performance of Figures for Methods A and B

The first section of Table II lists for the smallest and largest files in the test suite the number of trie nodes required during the compression. The second section of the table lists, for the same two files, the success rate of the searches in the maximum context, with the maximum context varying from 1 to 5. For example, using a fourth order model on the text file, 63461 trie nodes were required, corresponding to 1.1 megabytes using method A and 1.2 megabytes using method B. In the same fourth-order model 74% of the text file (using method A) was coded using fourth order contexts. The figures show that for English text a large fraction of the characters can be predicted using relatively long contexts, but that the space requirements might become very large. The object file is harder to compress, but even so, more than half of the characters could be coded using second order contexts.

Table III lists compression performance, measured as output bits per input character, and throughput, measured in kilobytes of source data per second, for methods A and B. The compression figure was calculated as a weighted average over the nine input files, and the throughput was measured as twice the total size of the nine files, divided by the total time required to both encode and decode them all. In almost all cases encoding and decoding required very similar running times, and so a single throughput figure was appropriate. There were individual discrepancies from the averages shown, and, for example, throughput on the object file was slower than the average and compression worse. However, in practice a data compression scheme will be called upon to compress a mixture of file types, and it seemed not unreasonable to compare the methods based upon such an average. The fifth-order models were affected by significant page faulting, and the times were not reliable.

In terms of compression, the two methods were very close, and the results agree with those of Cleary and Witten. However method B requires slightly more time than method A, caused primarily by

TABLE IV
COMPRESSION IMPROVEMENTS

	modification	order 1	2	3	4	5
compression (bits per char)	method A	3.50	2.81	2.55	2.50	2.51
	count scaling	3.41	2.71	2.42	2.35	2.36
	single counting	3.48	2.74	2.42	2.35	2.36
	hybrid probability	3.47	2.71	2.38	2.28	2.27
	all modifications	3.37	2.61	2.28	2.19	2.19

the need for each novel character to be transmitted twice in a lower order model.

B. Improving Compression Performance

A number of variations on the PPM theme were also tested. This section describes these modifications, and details the improved compression achieved by them.

The first modification was to the count scaling. Because the arithmetic coding routines used had a limit of 16383 on the maximum frequency count that could be handled (see [11] for a discussion of this), the initial implementations of methods A and B halved all of the counts in a particular context whenever this limit was reached. The scaling also brought the useful side effect of making the model self-adapting to changing symbol distributions in the input stream. For example, halving the count of the number of distinct following symbols (method B), has the effect that, after a learning phase when the escape has a relatively high probability, it is scaled back toward zero more quickly than just by the sheer accumulation of statistics. Experiments with different values of the limit showed that as a general rule, the more often the counts were halved, the better the compression. A limit of 64 gave marginally better compression than other values tested. Because there might be up to 256 items in any particular context list, and to avoid rounding errors, extra precision was achieved by incrementing the count variables by 8 rather than 1 at each symbol occurrence, and in fact scaling the counts when their total exceeded the threshold of $512 = 8 \times 64$. This combination meant that low probability events could still be allocated very small regions of codespace, but that the counts adapted rapidly. Results using this strategy in connection with method A are given as the second row of Table IV, and show a 5% improvement in compression.

The second modification tested was to count each symbol only in context levels at or above the context in which it was successfully predicted. For example, if "n" is coded in context "sio" it is redundant to add 1 to the count of "*n*" in the zero order context, the context "o," and the context "io." Adaptive coding is effective because what is happening is counted and used as the basis of subsequent coding, and so the only count that needs to be incremented is the count for "*n*" in the context "sio." The alternative full counting strategy counts predictions that are in fact *not* being coded. Implemented independently, this technique also improved the method *A* compression by about 5%.

The third change was to reconsider the method for calculating the escape probability. It seemed wasteful to only start using a context for predictions when it had already occurred twice (method B), but it seemed desirable, at least in the initial stages of the coding when novel symbols are relatively frequent, to allocate more than a count of 1 to the escape (method A). As a compromise between these two a hybrid, method C, was developed in an attempt to get the best of both. In method C, the escape is counted as having occurred a number of times equal to the number of distinct symbols encountered in the context, with the total context count inflated by the same amount. For example, using the symbol counts of Table I, $p_C(escape \mid \text{"sio"}) = (2/13)$, $p_C(\text{"n"} \mid \text{"sio"}) = (9/13)$ and, taking into account the exclusions to get an overall probability,

$$p_C(\text{"}a\text{"}) = (2/13) \times (3/6) \times (5/31) \times (32/111),$$

which yields an 8.13 bit code. The fourth row of Table IV lists the compression attained by this hybrid probability calculation, still with the original policies of full counting and count halving at 16383. The hybrid probability calculation significantly improved the compression obtained.

The final row of the table shows the combined effect of these three modifications, and represents the best compression obtained by tuning the PPM strategy. This final method will be called method C. Strictly speaking, method C is the hybrid scheme for estimating the escape probabilities, but it will be convenient in what follows to refer to the package of improvements as method C.

C. The Effect of Bounding the Data Space

One of the biggest obstacles to use of PPM compression is the large amount of data space required. Even in experiments on reasonably small files there were cases when there was insufficient real memory available to support the model. To develop a practical PPM it was necessary to investigate the sensitivity of the model to a bounded workspace.

To limit the data space, a strategy described by Cormack and Horspool [3] was adopted. A fixed amount of space was allocated to trie nodes, and when that space had been filled the encoding was temporarily halted and the entire trie discarded. To avoid very inefficient coding while the trie was being rebuilt, the last 2048 characters transmitted were maintained in a circular buffer, and the trie was rebuilt from the buffer before transmission was resumed.

To further reduce the space required the 32 bit pointers of the original trie were replaced by 16 bit array indexes. This then meant that each node was reduced to 14 bytes (method C), and that the trie was limited to 65536 nodes, or 896 kbytes. This was accepted as a reasonable compromise between node size and trie size. The results in Table V arise from a restricted space implementation of method C, and can be compared to the final row of Table IV. It was not practical to run the fourth and fifth order models with very small data spaces because of thrashing.

Within the range of data space allocations considered the order three model was the best, suffering little by being restricted to as little as 224 kbytes. Encoding became slower as the memory was restricted because of an increased fraction of nonproductive time that was spent rebuilding the trie when it became full. Encoding was also slower compared to the unrestricted space method C because of the overhead of repeatedly converting 16 bit indexes to 32 bit pointers and vice-versa.

Other strategies for reclaiming space from the trie could also be considered, such as garbage collection on nodes with counts of 1, garbage collection on leaves, or some sort of least recently used strategy [7]. However, because of the success of the simple method described, and the complexity of the alternatives, these were not tested. It is possible that they would provide a very slight compression advantage, but could be expected to require more running time.

D. Tuning for Speed

The bounded space variant of method C was then tuned for speed. Because of the good performance of the third order model, the maximum context was fixed at three, and a number of optimizations implemented. These primarily involved unrolling loops and replacing some procedure calls by in-line code, trading versatility for speed. This made the source code about 150 lines longer, and somewhat less understandable. Other parts of the code were also slightly rewritten, and a number of variables, such as the fingers into the trie, were updated lazily, that is, only when required. The combined effect of all of these changes was to increase the speed of the bounded space method C by about 25%. Compression was unaffected by these changes.

Finally, a fourth method of calculating the probabilities was considered, and in a deliberate attempt to trade compression for improved speed, the calculation of exclusions was completely dispensed with. Doing so meant that codespace would be wasted and compression would degrade. However not having to calculate exclusions meant that the searches at all context levels could make effective use of the move-to-front lists. The escape probability was still calculated using method C, and so this fourth program was called method Cnx, for no exclusions. For example, again using the

TABLE V
THE EFFECT OF LIMITING THE DATA SPACE

	data space	order 1	2	3	4	5
compression (bits per char)	56 kb	3.37	2.71	2.73	–	–
	112 kb	3.37	2.61	2.51	2.60	–
	224 kb	3.37	2.61	2.36	2.42	2.51
	448 kb	3.37	2.61	2.28	2.30	2.38
	896 kb	3.37	2.61	2.28	2.19	2.26

TABLE VI
PERFORMANCE OF METHODS C AND CNX

	data space	method C order=3	method Cnx order=3
compression (bits per char)	56 kb	2.73	2.94
	112 kb	2.51	2.68
	224 kb	2.36	2.51
	448 kb	2.28	2.40
throughput (kbyte per sec)	56 kb	1.7	2.4
	112 kb	2.3	3.4
	224 kb	3.0	4.2
	448 kb	3.3	4.5

figure of Table I, $p_{Cnx}(escape \mid \text{``sio''}) = (2/13)$, $p_{Cnx}(\text{``n''} \mid \text{``sio''}) = (9/13)$, and, as an overall probability

$$p_{Cnx}(\text{``}a\text{''}) = (2/13) \times (3/18) \times (5/70) \times (32/289)$$

giving a code of 12.27 bits. In this latter case method Cnx is relatively bad. On the other hand, should the current symbol be an "n" or a "t," both approaches would generate the same code. Table VI lists the compression results and throughput of the final method C, and an equivalent implementation of method Cnx.

Method Cnx runs about 40% faster than method C while giving compression only 5% worse. Moreover, the order three version of method Cnx outperforms even fourth and fifth order versions of methods A and B; can be implemented to run effectively in only 200 kbyte of data space; and is capable of encoding/decoding at speeds in excess of 4 kbytes/s. This then is the justification for the initial claim that Cleary and Witten were somewhat pessimistic when originally describing their PPM data compression method. (When run on a Vax 11/780, method Cnx processed the test suite at 2.73 kilobytes/s).

By way of comparison, the Unix utility *Compact*, implementing a zero order model with adaptive Huffman coding [6], was both slower (3.36 kbytes/s) and less efficient (4.72 bits/character) than method Cnx, while the utility Compress, implementing a form of Ziv-Lempel coding, ran about 8 times faster but also obtained inferior compression (3.37 bits/character) on the test files. Methods C and Cnx can be seen to yield very good compression at a reasonable resource cost.

IV. PPM ON LARGE ALPHABETS

The previous sections considered the application of the PPM paradigm to a stream of characters. The technique can also be applied to the encoding of other streams where the alphabet, the set of allowable symbols, is not so restricted. In particular, we have been interested in encoding streams of integers representing words for word based compression schemes [1], [4]. Each integer in the stream will be in the range 1 to $n + 1$ where n is the maximum integer appearing in the stream so far. Such a stream is generated by mapping distinct words onto sequentially allocated integers, so that, for example, "to be or not to be" is represented by "1 2 3 4 1 2." The actual characters associated with each word must also be transmitted in such a data compression model, as well as the nonwords that separate the words, but those aspects will be ignored here, and only the stream of words is considered. Further details can be found in [8].

As an experiment into the effectiveness of the various PPM models, each was modified to allow for numeric rather than character input, and then applied to the stream of word numbers generated for a large file of English text. This file contained 26257 integers, representing occurrences of 2408 distinct words. The results of these experiments are listed in Table VII. Methods A and B performed badly, with the poor compression being caused by the too small escape probability (method A) and the need to transmit each number twice in an unweighted raw form (method B). This transmission was avoided in the other methods by making use of the implicit knowledge that when the escape appears in the zero order context it must indicate a word number one larger than the largest number previously encountered. Method C included the single counting optimisation, but the threshold value for count scaling was retained at 16383—it made no sense to halve the counts early when the maximum number of distinct items in the list could not be known in advance. Methods C and Cnx obtained compressions up to 15% better than method B and 25% better than method A. The zero-order entropy of the word distribution in the test file was 8.68 bits/word.

TABLE VII
PPM PERFORMANCE ON WORDS

	method	order 1	2	3
trie nodes		15778	37051	60970
successful searches		49%	18%	8%
compression (bits per word)	method A	8.42	8.63	8.65
	method B	7.84	7.76	7.76
	method C	6.71	6.60	6.58
	method Cnx	6.75	6.72	6.76
throughput (words per sec)	method A	190	170	160
	method B	140	130	120
	method C	190	170	170
	method Cnx (list)	860	750	660
	method Cnx (tree)	1740	1330	1080

Character PPM is based on an alphabet of 256 distinct 8 bit symbols. To encode a stream of integers representing words using PPM, no such *a priori* bound can be placed on the size of the alphabet, and the asymptotic efficiency of the data structure used for the context searching must be considered. Searching in a list of n items to calculate exclusions will take $O(n)$ time, and will be very time consuming when n becomes large. For this reason, methods A, B, and C when applied to an integer input stream proved to be very slow.

On the other hand, method Cnx makes full use of the MTF lists and throughput was significantly better. Even more effective was to use a binary search tree for each context set within the trie, creating a tree trie; this required an additional pointer in each node and an additional counter, to record the sum of the context counts of nodes in the left subtree of the node. Use of a tree rather than a list reduced the searching time within each context to $O(\log n)$, and in practical terms, resulted in a two-fold speed increase without altering the compression. Method Cnx, using a tree trie, is much better suited to this type of input data than methods A, B, and C.

ACKNOWLEDGMENT

The author is grateful to T. Bell, who read and commented on an early draft of this work, and to the referees, who made several helpful suggestions.

REFERENCES

[1] J. Bentley, D. Sleator, R. Tarjan, and V. Wei, "A locally adaptive data compression scheme," *CACM 29*, no. 4, pp. 320-330, Apr. 1986.

[2] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. COM-32, pp. 396-402, Apr. 1984.

[3] G. Cormack and R. Horspool," Data compression using dynamic markov modelling," *Comput. J.*, vol. 30, no. 6, pp. 541-550, Dec. 1987.

[4] P. Elias, "Interval and recency-rank source coding: Two on-line adaptive variable-length schemes," *IEEE Trans. Inform. Theory*, vol. IT-33, pp. 3-10, Jan. 1987.
[5] D. Knuth, *Fundamental Algorithms Volume 3: Sorting and Searching*. New York: Addison-Wesley, 1975.
[6] ——"Dynamic Huffman coding," *J. Algorithms*, vol. 6, no. 2, pp. 163-180, 1985.
[7] V. Miller and M. Wegman, "Variations on a theme by Ziv and Lempel," in *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds. New York: Springer-Verlag, 1985, pp. 131-140, vol. 12.
[8] A. Moffat, "Word based text compression," *Software Practice and Experience*, vol. 19, no. 2, pp. 185-198, Feb. 1989.
[9] S. Thomas and J. Orost, *Compress (version 4.0) program and documentation*, available from petsd!joe@RUTGERS.EDU, 1985.
[10] T. Welch, "A technique for high performance data compression," *IEEE Comput.*, vol. 17, pp. 8-20, June 1984.
[11] I. Witten, R. Neal, and J. Cleary, "Arithmetic coding for data compression," *CACM*, vol. 30, no. 6, pp. 520-541, June 1987.
[12] J. Ziv and A. Lempel, "Compression of individual sequences via variable rate coding," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530-536, Sept. 1976.