UC San Diego
UC San Diego Previously Published Works

Title
Universal lossless compression via multilevel pattern matching

Permalink
https://escholarship.org/uc/item/39k54514

Journal
IEEE Transactions on Information Theory, 46(4)

ISSN
00189448

Authors
Kieffer, J. C.
Yang, En-Hui
Nelson, G. J.
et al.

Publication Date
2000-07-01

DOI
10.1109/18.850665

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Universal Lossless Compression Via Multilevel Pattern Matching

John C. Kieffer, *Fellow, IEEE*, En-hui Yang, *Member, IEEE*, Gregory J. Nelson, *Member, IEEE*, and Pamela Cosman, *Member, IEEE*

***Abstract*—A universal lossless data compression code called the multilevel pattern matching code (MPM code) is introduced. In processing a finite-alphabet data string of length n, the MPM code operates at $O(\log \log n)$ levels sequentially. At each level, the MPM code detects matching patterns in the input data string (substrings of the data appearing in two or more nonoverlapping positions). The matching patterns detected at each level are of a fixed length which decreases by a constant factor from level to level, until this fixed length becomes one at the final level. The MPM code represents information about the matching patterns at each level as a string of tokens, with each token string encoded by an arithmetic encoder. From the concatenated encoded token strings, the decoder can reconstruct the data string via several rounds of parallel substitutions. A $O(1/\log n)$ maximal redundancy/sample upper bound is established for the MPM code with respect to any class of finite state sources of uniformly bounded complexity. We also show that the MPM code is of linear complexity in terms of time and space requirements. The results of some MPM code compression experiments are reported.**

***Index Terms*—Arithmetic coding, entropy, lossless data compression, redundancy, universal codes.**

I. Introduction

UNIVERSAL lossless data compression algorithms based upon pattern matching have been studied in the source coding literature since the 1970's, beginning with the Lempel–Ziv code [17]. It is the purpose of this paper to put forth a new universal lossless data compression algorithm based upon pattern matching, which has some attractive features both with regard to data compression performance and implementation complexity. This new data compression algorithm is called the Multilevel Pattern Matching code (MPM code, for short). In this introductory section of the paper, we give a nontechnical description of the workings of the MPM code—a detailed description shall be presented in subsequent sections.

Manuscript received July 19, 1996; revised June 6, 1999. This work was supported in part by the National Science Foundation under Grants NCR-9304984, NCR-9508282, NCR-9627965, and by the Natural Sciences and Engineering Research Council of Canada under Grant RGPIN203035-98. The material in this paper was presented in part at the IEEE International Symposium on Information Theory, Cambridge, MA, August 16–22, 1998.

J. C. Kieffer is with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA.

E.-h. Yang is with the Department of Electrical & Computer Engineering, University of Waterloo, Waterloo, Ont., Canada N2L 3G1.

G. J. Nelson is with Anoka-Ramsey Community College, Coon Rapids, MN 55433 USA.

P. Cosman is with the Department of Electrical and Computer Engineering, University of California at San Diego, San Diego, CA 92037 USA.

Communicated by N. Merhav, Associate Editor for Source Coding.

Publisher Item Identifier S 0018-9448(00)04275-9.

For some fixed positive integer $r \geq 2$, and each data string x of length at least r over a fixed finite alphabet, let there be specified a positive integer I for which r^I is less than or equal to the length of x. (The choice of the integer I is dependent upon the length of x; we shall discuss the nature of this dependence later in the paper.) For each integer i satisfying $0 \leq i \leq I$, the MPM code extracts from x a certain sequence $S_i(x)$ consisting of some nonoverlapping substrings of x of length r^{I-i}. For each sequence $S_i(x)$ $(i = 0, 1, \cdots, I-1)$, let the substrings of x forming the entries of $S_i(x)$ be called "patterns." If $S_i(x) = (s_1, s_2, \cdots, s_k)$, the set of distinct patterns in $S_i(x)$ is the set $\{u : u = s_i \text{ for some } i = 1, 2, \cdots, k\}$. In each $S_i(x)$ $(i = 0, 1, \cdots, I-1)$, the MPM code detects the distinct patterns in $S_i(x)$. For each distinct pattern u appearing in $S_i(x)$, the MPM code performs a pattern-matching task consisting of determining which entries of $S_i(x)$ match u (i.e., coincide with u)—each appearance in $S_i(x)$ of a pattern matching u is replaced with a "token" from an abstract token alphabet $\{t_0, t_1, t_2, t_3, \cdots\}$, so that distinct patterns in $S_i(x)$ are assigned distinct tokens. In this way, each sequence $S_i(x)$ $(i = 0, 1, \cdots, I-1)$ is "tokenized" via pattern matching to yield a "token sequence" T_i containing the same number of terms as the sequence $S_i(x)$.

The sequences $T_0, T_1, \cdots, T_{I-1}$, together with the sequence $T_I = S_I(x)$ consisting of some individual entries of the given data string x, form the sequence $(T_0, T_1, \cdots, T_I)$, called the *multilevel representation* of x. Each data string can be fully recovered from its multilevel representation. Via a simple adaptive arithmetic encoder, the MPM code separately encodes each token sequence T_i in the multilevel representation $(T_0, T_1, \cdots, T_I)$ of the data string x into a binary string B_i. The binary codeword for the data string x generated by the MPM code is then obtained by concatenating the strings $B_0, B_1, \cdots, B_I$ together from left to right; we write this codeword as $B_0B_1 \cdots B_I$. From the codeword $B_0B_1 \cdots B_I$, the MPM code can decode the multilevel representation $(T_0, T_1, \cdots, T_I)$, from which the data string x is reconstructed by means of parallel substitutions.

The structure of the MPM code is depicted in the block diagrams in Figs. 1 and 2, where we assume that $I = 3$ for simplicity. The encoding part of the MPM code is given in Fig. 1. The mappings Ψ, Φ, and Π are string processing functions, defined in Section II, that allow for the recursive computation of the sequences $S_0(x)$, $S_1(x)$, $S_2(x)$, $S_3(x)$ from the input data x. The "tokenization map" t, also described in Section II, converts $(S_0(x), S_1(x), S_2(x), S_3(x))$ into the multilevel representation (T_0, T_1, T_2, T_3). The bit strings B_0, B_1, B_2, B_3 are

0018–9448/00$10.00 © 2000 IEEE

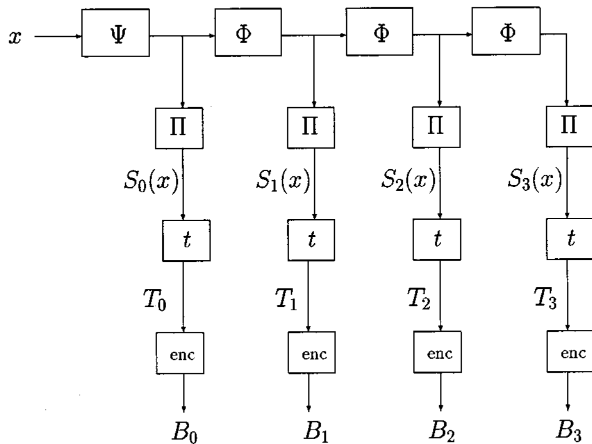

Fig. 1. Encoding part of MPM code.

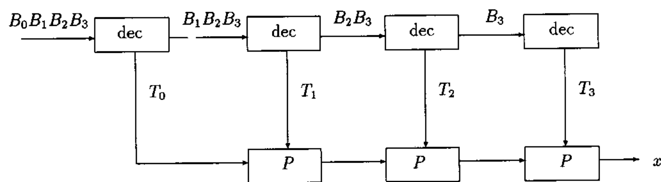

Fig. 2. Decoding part of MPM code.

then obtained by encoding T_0, T_1, T_2, T_3, respectively. Fig. 2 gives the decoding part of the MPM code. The mapping P, described in Section II, is used to perform parallel substitutions, and allows the reconstruction of x in a recursive manner.

Example 1: We illustrate the workings of the MPM code using a simple example. For simplicity, we take $r = 2$ and take the length of the data string to be a power of two. (The case of general r and general data lengths will be considered in Section II.) We take the data string x to be the following binary string of length 32:

$$x = 00100100100000010010000010010010.$$

Let us suppose that $I = 4$. We need to describe how to form the sequences $S_i(x)$, $i = 0, 1, 2, 3, 4$. First, form the following sequences:

$$\begin{aligned} u_0 &= x \\ u_1 &= (0010010010000001,\ 0010000010010010) \\ u_2 &= (00100100,\ 10000001,\ 00100000,\ 10010010) \\ u_3 &= (0010,\ 0100,\ 1000,\ 0001,\ 0000,\ 1001) \\ u_4 &= (00,\ 10,\ 01). \end{aligned}$$

The entries of u_i are substrings of x of length 2^{5-i}. The entries of u_i are obtained by partitioning x into nonoverlapping blocks of length 2^{5-i}, and, of these blocks, keeping only the distinct blocks as entries of u_i, in order of first appearance from left to right in the partitioning. The sequence $S_i(x)$ is then obtained from the sequence u_i by bisecting each entry of u_i. (In other words, simultaneously replace each entry of u_i with the two substrings that form the left and right halves of that entry.) This gives us

$$\begin{aligned} S_0(x) &= (0010010010000001,\ 0010000010010010) \\ S_1(x) &= (00100100,\ 10000001,\ 00100000,\ 10010010) \\ S_2(x) &= (0010,\ 0100,\ 1000,\ 0001,\ 0010,\ 0000,\ 1001,\ 0010) \\ S_3(x) &= (00,\ 10,\ 01,\ 00,\ 10,\ 00,\ 00,\ 01,\ 00,\ 00,\ 10,\ 01) \\ S_4(x) &= (0,\ 0,\ 1,\ 0,\ 0,\ 1). \end{aligned}$$

We then "tokenize" each $S_i(x)$ $(i = 0, 1, 2, 3)$, obtaining the multilevel representation $(T_0, T_1, T_2, T_3, T_4)$ in which

$$\begin{aligned} T_0 &= (t_0,\ t_1) \\ T_1 &= (t_0,\ t_1,\ t_2,\ t_3) \\ T_2 &= (t_0,\ t_1,\ t_2,\ t_3,\ t_0,\ t_4,\ t_5,\ t_0) \\ T_3 &= (t_0,\ t_1,\ t_2,\ t_0,\ t_1,\ t_0,\ t_0,\ t_2,\ t_0,\ t_0,\ t_1,\ t_2) \\ T_4 &= (0,\ 0,\ 1,\ 0,\ 0,\ 1). \end{aligned}$$

For $0 \le i \le 3$, we tokenized to obtain T_i from $S_i(x)$ by scanning $S_i(x)$ from left to right, assigning to the first entry the token t_0, assigning to each succeeding entry not matching a previously scanned pattern the first token from the list $t_0, t_1, t_2, \cdots$ not used previously, and assigning to each entry matching a previously scanned pattern the same token that was used previously for that pattern.

The multilevel representation $(T_0, T_1, T_2, T_3, T_4)$ allows reconstruction of x via four rounds of parallel substitutions, as follows. In the first round of parallel substitutions, replace each t_0 in T_0 by (t_0, t_1) (the first two entries of T_1) and replace each t_1 in T_0 by (t_2, t_3) (the next two entries of T_1). The resulting token sequence is

$$x^{(1)} = (t_0,\ t_1,\ t_2,\ t_3).$$

In the second round of parallel substitutions, make the substitutions

$$\begin{aligned} t_0 &\to (t_0,\ t_1) \\ t_1 &\to (t_2,\ t_3) \\ t_2 &\to (t_0,\ t_4) \\ t_3 &\to (t_5,\ t_0) \end{aligned}$$

for the entries of $x^{(1)}$, with the right sides of the substitutions taken as the entries of T_2, two by two. This yields the token sequence

$$x^{(2)} = (t_0,\ t_1,\ t_2,\ t_3,\ t_0,\ t_4,\ t_5,\ t_0).$$

In the third round of parallel substitutions, make the substitutions

$$\begin{aligned} t_0 &\to (t_0,\ t_1) \\ t_1 &\to (t_2,\ t_0) \\ t_2 &\to (t_1,\ t_0) \\ t_3 &\to (t_0,\ t_2) \\ t_4 &\to (t_0,\ t_0) \\ t_5 &\to (t_1,\ t_2) \end{aligned}$$

for the entries of $x^{(2)}$, where the right sides of the substitutions come from the entries of T_3, two by two. This yields the token sequence

$$x^{(3)} = (t_0, t_1, t_2, t_0, t_1, t_0, t_0, t_2, t_0, t_1, t_0, t_0, t_1, t_2, t_0, t_1).$$

In the fourth and last round of parallel substitutions, a sequence $x^{(4)}$ is obtained from $x^{(3)}$, by making the following substitutions for the entries of $x^{(3)}$:

$$t_0 \rightarrow (0, 0)$$
$$t_1 \rightarrow (1, 0)$$
$$t_2 \rightarrow (0, 1).$$

The right sides come from the entries of T_4. The reader can see that $x = x^{(4)}$.

We discuss what shall be accomplished in this paper concerning the MPM data compression code. In Section II, we lay out the particulars concerning the encoding part and the decoding part of the MPM coding procedure. In Section III, we investigate the order of growth of the total number of entries which appear in the sequences forming the multilevel representation $(T_0, T_1, \cdots, T_I)$ of a data string of length n; in particular, we show that taking $I = O(\log \log n)$ makes this order of growth $O(n/\log n)$. In Section IV, we apply the Section III order of growth result to perform a redundancy analysis for the MPM code. The key result of the paper (Theorem 4) asserts that the maximal redundancy/sample of the MPM code relative to any class of finite-state sources of uniformly bounded complexity is $O(1/\log n)$ as a function of the data length n, a better redundancy result than has currently been established for the Lempel–Ziv code [17] (whose maximal redundancy/sample is only known to be $O(\log \log n/\log n)$ [14]). The MPM code is a universal code in the sense that it optimally encodes any stationary source (Theorem 5). In Section IV, we also present some results of compression experiments in which the MPM code was used to losslessly compress binary images—the results show that the MPM code is competitive with JBIG on large binary images. The paper concludes with a complexity analysis of the MPM code; it is shown (Theorem 6) that the MPM code, like the Lempel–Ziv code, is of linear time and storage complexity as a function of the data length.

To our knowledge, the MPM code is the first pattern matching based universal lossless data compression code for which both of the following have been established:

a) linearity in time and space complexity as a function of data length n;
b) $O(1/\log n)$ maximal redundancy/sample behavior.

II. The Algorithm MPM (r, I)

Let A denote a finite alphabet containing at least two symbols, fixed for the rest of this paper. The terminology A-string shall refer to any string of finite length whose entries are selected from A (excluding the empty string). We shall be using the multilevel pattern matching method outlined in Section I to compress and decompress A-strings. As indicated in Section I, there are two parameters r and I that are preselected in order to compress an A-string by the multilevel pattern-matching method. The parameter r is an integer that must satisfy $r \geq 2$. The parameter I is a nonnegative integer.

We fix the parameters r and I throughout this section. Consider the data compression method, which we call algorithm MPM(r, I), in which each A-string x of length at least r^I is compressed and decompressed in four phases, performed in the following order.

1) **Multilevel Decomposition Phase:** The sequences $S_0(x), S_1(x), \cdots, S_I(x)$ are formed. Each $S_i(x)$ consists of nonoverlapping (but not necessarily contiguous) substrings of x of length r^{I-i}.
2) **Tokenization Phase:** The output of the tokenization phase is the multilevel representation $(T_0, T_1, \cdots, T_I)$ of the data string x, in which each sequence $S_i(x)$ from the multilevel decomposition phase is "tokenized" to form the sequence T_i.
3) **Encoding/Decoding Phase:** The entries T_i of the multilevel representation $(T_0, T_1, \cdots, T_I)$ are separately encoded and decoded.
4) **Reconstruction Phase:** Parallel substitutions are used to reconstruct x from the multilevel representation.

In this section, we make precise the workings of the algorithm MPM(r, I) by explaining the preceding four phases in detail. The eventual MPM code that shall be spelled out in Section IV employs a certain choice of I as a function of the length of the A-string to be compressed. (In other words, in Section IV, we shall specify a sequence of nonnegative integers $\{I_n\}$ such that if an A-string x is of length n, then x will be compressed/decompressed with the algorithm MPM(r, I_n).) The choice of the parameter I as a function of the data length n cannot be made now, since at present we do not know how to optimize the choice of I. By holding I fixed in this section and in Section III, we shall be able to make an analysis of the algorithm MPM(r, I) that will enable us to cleverly choose I as a function of n. We shall see that $I \sim \log \log n$ is the best choice.

Before describing the algorithm MPM(r, I), we introduce some notation and terminology that shall be in effect throughout this paper. If D is a nonempty set, we let D^+ denote the set of all strings of finite length over the alphabet D, excluding the empty string. (As a special case, A^+ is the set of all A-strings.) Sometimes, for convenience, we shall want to append an empty string to the set D^+; letting λ denote the empty string, D^* denotes the set $D^+ \cup \{\lambda\}$. For $n = 1, 2, \cdots$, D^n shall denote the set of all strings in D^+ of length n. For each x in D^*, let $|x|$ denote the length of x. If $x^{(1)}, x^{(2)}, \cdots, x^{(k)}$ are strings in D^*, let $x^{(1)}x^{(2)} \cdots x^{(k)}$ denote the string in D^* obtained by concatenating together the strings $x^{(1)}, x^{(2)}, \cdots, x^{(k)}$ from left to right. If S is a finite set, then $|S|$ shall denote the cardinality of S. All logarithms written "log" without any subscript shall denote logarithms to the base two; if we use a base for the logarithm other than base two, that base shall be denoted by means of a subscript.

We let $\mathcal{S}(A)$ denote the set of all sequences of finite length whose entries come from A^+ (including the empty sequence).

We have to be very careful with our notation so that we do not confuse the A-strings in A^+ with members of $\mathcal{S}(A)$. Letting $A = \{a_0, a_1, \cdots, a_{|A|-1}\}$, we shall take the A-strings to be all the expressions of the form $a_{i_1} a_{i_2} \cdots a_{i_n}$, where $n = 1, 2, \cdots$, and $i_1, i_2, \cdots, i_n \in \{0, 1, \cdots, |A|-1\}$. (Notice that in these expressions $a_{i_1} a_{i_2} \cdots a_{i_n}$ we do not separate the entries by commas or place parentheses around the expressions.) An element of $\mathcal{S}(A)$ shall be written $(u_1, u_2, \cdots, u_k)$, where each entry u_j is an A-string and therefore an expression of the form $a_{i_1} a_{i_2} \cdots a_{i_n}$. If $S_1, S_2, \cdots, S_k$ are sequences in $\mathcal{S}(A)$, then $S_1 \cup S_2 \cup \cdots \cup S_k$ is the sequence in $\mathcal{S}(A)$ in which we first write down the entries of S_1, then the entries of S_2, then the entries of S_3, etc.

Example 2: Let $A = \{0, 1\}$. Then 0010 is a member of A^+, whereas (0010), (0, 0, 1, 0) and (00, 10) are members of $\mathcal{S}(A)$. If $S_1 = (0010)$ and $S_2 = (00, 10)$, then

$$S_1 \cup S_2 = (0010, 00, 10).$$

A. Multilevel Decomposition Phase

We fix throughout this subsection the pair of integers (r, I) in which $r \geq 2$ and $I \geq 0$. The goal of this subsection is to explain the workings of the multilevel decomposition phase of the algorithm MPM(r, I). Let x be an A-string of length at least r^I, and we operate on this string with the algorithm MPM(r, I). The end product of the multilevel decomposition phase is then a set of sequences $\{S_0(x), S_1(x), \cdots, S_I(x)\}$ from $\mathcal{S}(A)$, with each sequence $S_i(x)$ consisting of certain carefully selected substrings of x of length r^{I-i}.

The sequences $S_0(x), S_1(x), \cdots, S_I(x)$ are generated recursively. This shall be accomplished using three "string processing functions" Ψ, Φ, and Π. We proceed to define each of these functions, followed by the definition of the $\{S_i(x)\}$.

The Function Ψ*:* This function is a mapping from the set $\{x \in A^+ : |x| \geq r^I\}$ into the set $\mathcal{S}(A) \times A^*$. Let x be an A-string of length at least r^I. Let $k = \lfloor |x|/r^I \rfloor$. Let $x^{(1)}, x^{(2)}, \cdots, x^{(k)}, x^{(k+1)}$ be the unique strings in A^* such that

$$x = x^{(1)} x^{(2)} \cdots x^{(k)} x^{(k+1)}$$

and such that each of the strings except $x^{(k+1)}$ is of length r^I. Then

$$\Psi(x) \triangleq \left(\left(x^{(1)}, x^{(2)}, \cdots, x^{(k)} \right), x^{(k+1)} \right).$$

The Function Φ*:* Let $\mathcal{S}_r(A)$ be the subset of $\mathcal{S}(A)$ such that a sequence $u \in \mathcal{S}(A)$ is a member of $\mathcal{S}_r(A)$ if and only if all of the A-strings which are entries of u are of the same length, where this common length can depend on u and is of the form r^k for some $k > 0$. The function Φ is a mapping from $\mathcal{S}_r(A) \times A^*$ into $\mathcal{S}(A) \times A^*$. Let $u = (u_1, \cdots, u_m)$ be a sequence in $\mathcal{S}_r(A)$ and let $v \in A^*$. We define $\Phi(u, v)$ as follows. First, identify the distinct entries of u, which we label as $u_{i_1}, u_{i_2}, \cdots, u_{i_q}$, where

$$q = |\{u_1, u_2, \cdots, u_m\}| \tag{2.1}$$

$$1 \leq i_1 < i_2 < \cdots < i_q \leq m \tag{2.2}$$

and

$$i_j = \min\{1 \leq i \leq m : u_i = u_{i_j}\}, \qquad j = 1, 2, \cdots, q. \tag{2.3}$$

Let each entry of u have length r^k. For each $j = 1, 2, \cdots, q$, let $w^{(j)} = (w_{j,1}, w_{j,2}, \cdots, w_{j,r})$ be the sequence in $\mathcal{S}(A)$ such that

- $u_{i_j} = w_{j,1} w_{j,2} \cdots w_{j,r}$ and
- $w_{j,1}, w_{j,2}, \cdots, w_{j,r}$ are all of length r^{k-1}.

Let $v(2)$ be the string in A^*, and let $v(1) = (v^1, v^2, \cdots, v^s)$ be the sequence in $\mathcal{S}(A)$ such that

- $s = \lfloor |v|/r^{k-1} \rfloor$
- $v = v^1 v^2 \cdots v^s v(2)$
- $v^1, v^2, \cdots, v^s$ are all of length r^{k-1}.

(If $|v| < r^{k-1}$, $v(1)$ is taken to be the empty sequence in $\mathcal{S}(A)$.) Then

$$\Phi(u, v) \triangleq \left(w^{(1)} \cup w^{(2)} \cup \cdots \cup w^{(q)} \cup v(1), v(2) \right).$$

The Function Π*:* This function is the projection mapping from $\mathcal{S}(A) \times A^*$ onto $\mathcal{S}(A)$. Therefore,

$$\Pi(u, v) = u, \qquad u \in \mathcal{S}(A), \quad v \in A^*.$$

Definition of the Sequences $\{S_i(x): 0 \leq i \leq I\}$*:* Let x be any A-string of length at least r^I. Let

$$(u_0, v_0) = \Psi(x)$$
$$(u_i, v_i) = \Phi(u_{i-1}, v_{i-1}), \qquad 0 < i \leq I.$$

Then

$$S_i(x) \triangleq \Pi(u_i, v_i), \qquad 0 \leq i \leq I.$$

Example 3: Consider the string

$$x = 0000000100000001000 1011 \tag{2.4}$$

and we suppose that $r = 2$, $I = 4$. We recursively compute

$$(u_0, v_0) = ((0000000100000001), 0001011)$$
$$(u_1, v_1) = ((00000001, 00000001), 0001011)$$
$$(u_2, v_2) = ((0000, 0001, 0001), 011)$$
$$(u_3, v_3) = ((00, 00, 00, 01, 01), 1)$$
$$(u_4, v_4) = ((0, 0, 0, 1, 1), \lambda).$$

Projecting down onto the first coordinate

$$S_0(x) = (0000000100000001)$$
$$S_1(x) = (00000001, 00000001)$$
$$S_2(x) = (0000, 0001, 0001)$$
$$S_3(x) = (00, 00, 00, 01, 01)$$
$$S_4(x) = (0, 0, 0, 1, 1).$$

B. Tokenization Phase

Throughout this subsection, we fix $r \geq 2$, and a nonnegative integer I. The purpose of this subsection is to describe the tokenization phase of the algorithm MPM(r, I) applied to any A-string x of length at least r^I, in which each sequence $S_i(x)$, $0 \leq i \leq I$, generated in the multilevel decomposition phase, is converted into a certain sequence T_i of the same length as $S_i(x)$.

We define a "token alphabet"

$$\mathcal{T} \triangleq \{t_0, t_1, t_2, \cdots\}$$

whose entries are abstract symbols called "tokens." The symbols t_i in the token alphabet $\mathcal{T}$ are distinct, and, in addition, we assume that none of them is a member of the data alphabet A.

According to our definitions at the beginning of Section II, $\mathcal{T}^+$ denotes the set of all sequences of finite length whose entries are selected from the token alphabet $\mathcal{T}$. We use the same notational convention in writing the elements of $\mathcal{T}^+$ that we do in writing the elements of $\mathcal{S}(A)$, namely, the entries of a sequence in $\mathcal{T}^+$ are separated by commas, with the entire sequence enclosed in parentheses. For example, $(t_0, t_1, t_0, t_2, t_1)$ is a member of the set $\mathcal{T}^+$.

Let γ be the following natural injection mapping from A^+ into $\mathcal{S}(A)$:

$$\gamma(x_1x_2\cdots x_n) \triangleq (x_1, x_2, \cdots, x_n), \qquad x_1x_2\cdots x_n \in A^+.$$

The tokenization phase shall employ a mapping

$$t: \mathcal{S}_r(A) \cup \gamma(A^+) \to \mathcal{T}^+ \cup \gamma(A^+)$$

which we shall call the *tokenization map*. If $u \in \gamma(A^+)$, we define $t(u) = u$. Now we suppose that $u = (u_1, u_2, \cdots, u_m)$ is a sequence in $\mathcal{S}_r(A)$. In this case, $t(u)$ will be a member of $\mathcal{T}^+$. We describe how $t(u)$ is formed. First, identify the distinct entries of u, which we label as $u_{i_1}, u_{i_2}, \cdots, u_{i_q}$, where (2.1)–(2.3) hold. Writing

$$v_j = u_{i_{j+1}}, \qquad j = 0, 1, \cdots, q-1$$

we can then rewrite u as

$$u = (v_{s_1}, v_{s_2}, \cdots, v_{s_m})$$

where $s_1, s_2, \cdots, s_m$ belong to $\{0, 1, \cdots, q-1\}$. Define

$$t(u) \triangleq (t_{s_1}, t_{s_2}, \cdots, t_{s_m}).$$

Definition of Sequences $\{T_i\}$*:* Let x be an A-string of length at least r^I. Application of the algorithm MPM(r, I) to x yields the sequences $\{S_i(x): 0 \le i \le I\}$ in the multilevel decomposition phase. We define

$$T_i = t(S_i(x)), \qquad i = 0, 1, 2, \cdots, I.$$

The sequence $(T_0, T_1, \cdots, T_I)$ is the *multilevel representation* of x generated by the algorithm MPM(r, I).

Example 4: We compute the multilevel representation of the data string in Example 3, where $r = 2$ and $I = 4$

$$T_0 = t((0000000100000001)) = (t_0) \tag{2.5}$$
$$T_1 = t((00000001, 00000001)) = (t_0, t_0) \tag{2.6}$$
$$T_2 = t((0000, 0001, 0001)) = (t_0, t_1, t_1) \tag{2.7}$$
$$T_3 = t((00, 00, 00, 01, 01)) = (t_0, t_0, t_0, t_1, t_1) \tag{2.8}$$
$$T_4 = t((0, 0, 0, 1, 1)) = (0, 0, 0, 1, 1). \tag{2.9}$$

C. Reconstruction Phase

The purpose of this subsection is to specify how the algorithm MPM(r, I) reconstructs an A-string from its multilevel representation. Let $\mathcal{T}^{(2)}$ denote the set of all pairs (u, v) such that

- $u \in \mathcal{T}^+$, $v \in \mathcal{T}^+ \cup \gamma(A^+)$.
- If $u = (s_1, \cdots, s_k)$, and j is the cardinality of the set $\{s_1, s_2, \cdots, s_k\}$, then the length of v is at least rj.

Example 5: Suppose $r = 2$. Let

$$u = (t_0, t_1, t_0, t_2, t_1, t_2) \tag{2.10}$$
$$v = (t_0, t_0, t_1, t_0, t_2, t_1, t_0). \tag{2.11}$$

The number of distinct entries of u is 3. The length of v (which is 7) is at least as big as $3r = 6$. Therefore, the pair (u, v) belongs to $\mathcal{T}^{(2)}$.

Definition: We define a mapping $P: \mathcal{T}^{(2)} \to \mathcal{T}^+ \cup A^+$ which we call the *parallel substitution map*. Let $(u, v) \in \mathcal{T}^{(2)}$. Let the distinct entries of u be the following symbols in $\mathcal{T}$:

$$t_{i_1}, t_{i_2}, \cdots, t_{i_j} \tag{2.12}$$

where we have ordered the list in (2.12) so that

$$i_1 < i_2 < \cdots < i_j.$$

Let $v^{(1)}, v^{(2)}, \cdots, v^{(j)}, v^{(j+1)}$ be the unique sequences such that

- $v^{(1)}, v^{(2)}, \cdots, v^{(j)}$ are the sequences in $\mathcal{T}^+ \cup \gamma(A^+)$ of length r, which, when concatenated together from left to right in the indicated order, yield a prefix of v.
- If k is the length of v, then $v^{(j+1)}$ is the suffix of v of length $k - jr$. (Note: $v^{(j+1)}$ is taken to be the empty sequence if $k = jr$.)

The string $P(u, v) \in \mathcal{T}^+ \cup A^+$ is formed via the following two steps:

Step 1: Write down below each entry of

$$u = (u_1, u_2, \cdots, u_m)$$

the corresponding member of the set

$$\{v^{(1)}, v^{(2)}, \cdots, v^{(j)}\}$$

according to the substitutions

$$t_{i_q} \to v^{(q)}, \qquad q = 1, 2, \cdots, j.$$

Let $w_1, w_2, \cdots, w_m$ be the resulting list of members of $\mathcal{T}^+ \cup \gamma(A^+)$.

Step 2: Concatenate together the members of the list

$$w_1, w_2, \cdots, w_m, v^{(j+1)}$$

from left to right, thereby obtaining a sequence σ in $\mathcal{T}^+ \cup \gamma(A^+)$. Then

$$P(u, v) \triangleq \begin{cases} \sigma, & \sigma \in \mathcal{T}^+ \\ \gamma^{-1}(\sigma), & \sigma \in \gamma(A^+). \end{cases}$$

Example 6: We consider again the strings u and v given in (2.10) and (2.11). We suppose that $r = 2$. Then, the substitutions that are to be used in computing $P(u, v)$ are

$$t_0 \to (t_0, t_0)$$
$$t_1 \to (t_1, t_0)$$
$$t_2 \to (t_2, t_1).$$

Using these substitutions on the six entries of u, we obtain the six sequences

$$(t_0, t_0), (t_1, t_0), (t_0, t_0), (t_2, t_1), (t_1, t_0), (t_2, t_1).$$

These sequences are concatenated and (t_0), the suffix of v that was not used in forming the above substitutions, is also concatenated to the right end. We conclude that

$$P(u, v) = (t_0, t_0, t_1, t_0, t_0, t_0, t_2, t_1, t_1, t_0, t_2, t_1, t_0).$$

Formation of x from its multilevel representation. Let x be an A-string of length at least r^I to which the algo-

rithm MPM(r, I) has assigned the multilevel representation $(T_0, T_1, \cdots, T_I)$. A comparison of the definitions of the mappings Φ and t with the definition of the mapping P indicates that the operator P inverts the effect of the mappings Φ and t. Since Φ and t are used to determine the $\{T_i\}$ from x, this means that x can be reconstructed from the $\{T_i\}$ using P. Here is the algorithm via which this reconstruction is accomplished:

ALGORITHM: Given $(T_0, T_1, \cdots, T_I)$, generate $t^{(0)}$, $t^{(1)}, \cdots, t^{(I)}$ recursively by

$$t^{(0)} = T_0 \tag{2.13}$$

$$t^{(i)} = P(t^{(i-1)}, T_i), \qquad 0 < i \le I. \tag{2.14}$$

Then

$$x = t^{(I)}.$$

Example 7: Assume that $r = 2$, and that the alphabet is $A = \{0, 1\}$. We consider the multilevel description $\{T_0, T_1, T_2, T_3, T_4\}$ given in (2.5)–(2.9), for a certain string $x \in A^+$. Applying (2.13) and (2.14), we obtain

$$\begin{aligned} t^{(0)} &= T_0 = (t_0) \\ t^{(1)} &= P\left(t^{(0)}, T_1\right) = (t_0, t_0) \\ t^{(2)} &= P\left(t^{(1)}, T_2\right) = (t_0, t_1, t_0, t_1, t_1) \\ t^{(3)} &= P\left(t^{(2)}, T_3\right) = (t_0, t_0, t_0, t_1, t_0, t_0, t_0, t_1, t_0, t_1, t_1) \\ t^{(4)} &= P\left(t^{(3)}, T_4\right) = 0000000100000001000101\,1 = x. \end{aligned}$$

D. Encoding/Decoding Phase

It is the purpose of this subsection to describe the encoding/decoding phase of the algorithm MPM(r, I). Let x be an A-string of length at least r^I. Let $(T_0, T_1, \cdots, T_I)$ be the multilevel representation of x obtained by applying the algorithm MPM(r, I) to x. In the encoding/decoding phase, an encoder encodes the multilevel representation $(T_0, T_1, \cdots, T_I)$ of x into a binary string that is transmitted to the decoder; the decoder then decodes this binary string back into the multilevel representation.

In the encoding/decoding phase, three encoder/decoder pairs (E_1, D_1), (E_2, D_2), (E_3, D_3) are employed. The encoder/decoder pair (E_1, D_1) is used first, and allows for the communication of $|x|$ to the decoder; this step is necessary because the value of $|x|$ is needed by the decoder for the rest of the encoding/decoding phase. After the encoder/decoder pair (E_1, D_1) has been used to communicate to the decoder the value of $|x|$, the encoder/decoder pair (E_2, D_2) is used to communicate to the decoder each sequence $T_i \in \{T_0, T_1, \cdots, T_{I-1}\}$. Finally, the encoder/decoder pair (E_3, D_3) is used to communicate the sequence T_I to the decoder.

We now give precise descriptions of the encoder/decoder pairs (E_1, D_1), (E_2, D_2), (E_3, D_3).

1) Encoder/Decoder Pair (E_1, D_1)*:* Each positive integer n has a unique binary expansion $(b_1, b_2, \cdots, b_k)$ in which $b_1 = 1$ and

$$n = 2^{k-1} + \sum_{i=2}^{k} b_i 2^{k-i}.$$

We shall write

$$n \sim (b_1, b_2, \cdots, b_k)$$

to denote that $(b_1, b_2, \cdots, b_k)$ is the unique binary expansion of the positive integer n. The encoder E_1 is taken to be the one-to-one mapping from the set $\{1, 2, \cdots\}$ into the set of binary strings $\{0, 1\}^+$ such that, if $n \sim (b_1, b_2, \cdots, b_k)$, then

$$E_1(n) = (b_1, b_1, b_2, b_2, \cdots, b_{k-1}, b_{k-1}, b_k, 1 - b_k).$$

Note that the set

$$\{E_1(n): n = 1, 2, \cdots\} \tag{2.15}$$

is a *prefix set*, meaning that any infinite or finite binary string can have at most one member of the set (2.15) as a prefix. The decoder D_1 is the unique mapping from the set

$$\{u \in \{0, 1\}^+: u \text{ has a prefix in the set (2.15)}\}$$

onto the set $\{1, 2, 3, \cdots\}$ in which

$$n = D_1(u)$$

whenever u is a binary string with prefix $E_1(n)$.

For later use, it is not hard to see that

$$|E_1(n)| = 2\lceil \log(n+1) \rceil, \qquad n \ge 1. \tag{2.16}$$

Remark: To obtain good compression performance for the MPM code that we shall define in Section IV, one needs to require of E_1 only that

$$|E_1(n)| = O(\log n). \tag{2.17}$$

The particular encoder E_1 defined above satisfies (2.17), but many other encoders for the integers satisfying (2.17) have been studied. Elias [4] gives examples of encoders $\tilde{E}_1$ for the integers satisfying the property that

$$|\tilde{E}_1(n)| = \log n + o(\log n)$$

which is a stronger property than (2.17). We chose E_1 above rather than one of the Elias encoders because E_1 is simpler and allows us to obtain specific compression bounds later on in the paper.

The encoder/decoder pair (E_1, D_1) is now determined. The encoder will transmit the binary string $E_1(|x|)$ to the decoder, who will then use the decoding function D_1 to learn what $|x|$ is.

2) Encoder/Decoder Pair (E_2, D_2)*:* Recall from Section II-B the tokenization map t defined on $\mathcal{S}_r(A) \cup \gamma(A^+)$. Let u be any sequence in $t(\mathcal{S}_r(A))$. Let m be the number of distinct terms in u. Then, each term of u belongs to the set $\{t_0, t_1, \cdots, t_{m-1}\}$. Define $\mathcal{T}_u$ to be the set

$$\mathcal{T}_u \triangleq \{t_0, t_1, \cdots, t_{m-1}, t_m\}.$$

The set $\mathcal{T}_u$ has the following property that is useful for adaptive arithmetic coding:

Property: If u' is any sequence in $t(\mathcal{S}_r(A))$ having u as a proper prefix, then the entry of u' in position $|u| + 1$ belongs to the set $\mathcal{T}_u$.

For each $0 \le i \le m-1$, let $\eta(t_i|u)$ denote the number of times that t_i appears as an entry of u. Let $p(\cdot|u)$ denote the following probability distribution on $\mathcal{T}_u$:

$$p(t_i|u) = \begin{cases} \dfrac{\eta(t_i|u)}{|u|+m}, & i = 0, 1, \cdots, m-1 \\ \dfrac{m}{|u|+m}, & i = m. \end{cases} \tag{2.18}$$

Notice that each of the probabilities in this definition is positive. This fact, combined with the above Property, allows us to apply the theory in [2, Sec. 5.10] to conclude the existence of an adaptive arithmetic encoder/decoder pair (E_2, D_2) such that

i) E_2 is a mapping from $t(\mathcal{S}_r(A))$ into $\{0, 1\}^*$.

ii) If $u \in t(\mathcal{S}_r(A))$ is of length one, then $E_2(u)$ is the empty string in $\{0, 1\}^*$. (In this case, u must be the sequence (t_0).)

iii) If $u \in t(\mathcal{S}_r(A))$ is of length greater than one, then

$$|E_2(u)| = 1 + \left\lceil \sum_{i=1}^{|u|-1} -\log p(u_{i+1}|u^i) \right\rceil \tag{2.19}$$

where u^i denotes the prefix of u of length i, and u_{i+1} denotes the $(i+1)$st entry of u.

iv) D_2 is a mapping from $\{1, 2, \cdots\} \times \{0, 1\}^+$ into $t(\mathcal{S}_r(A))$ such that

$$u = D_2(|u|, B)$$

whenever $u \in t(\mathcal{S}_r(A))$ and B is any string in $\{0, 1\}^+$ having $E_2(u)$ as a prefix.

The reader interested in the practical implementation details of the adaptive arithmetic encoder/decoder (E_2, D_2) may consult the paper [11].

3) Encoder/Decoder Pair (E_3, D_3): The encoder/decoder pair (E_3, D_3) is easy to describe. The symbols appearing in T_I belong to the set A. Each symbol in A can be encoded using a binary string of length $\lceil \log |A| \rceil$. Therefore, T_I can be encoded symbol by symbol using a binary string of length $|T_I|\lceil \log |A| \rceil$.

4) Encoder/Decoder Implementation: We are now ready to describe the mechanics of the overall encoder/decoder scheme. The encoder output is the binary string B^* defined by

$$B^* \triangleq \begin{cases} E_1(|x|)E_3(T_0), & I = 0 \\ E_1(|x|)E_2(T_0)E_2(T_1) \\ \quad \cdots E_2(T_{I-1})E_3(T_I), & I > 0. \end{cases} \tag{2.20}$$

The decoder structure is a little more complicated. We need the following two definitions in order to specify the decoder structure:

Definition 1: If U and V are strings such that U is a prefix of V, define $V - U$ to be the suffix of V that remains after U is removed from the beginning of V.

Definition 2: If $u = (u_1, u_2, \cdots, u_j)$ belongs to the set $t(\mathcal{S}_r(A))$, define $\langle u \rangle$ to be the cardinality of the set $\{u_1, u_2, \cdots, u_j\}$.

If $I = 0$, the decoder determines T_0 from B^* by means of the equations

$$|x| = D_1(B^*)$$
$$T_0 = D_3(B^* - E_1(|x|)).$$

If $I > 0$, let $B_0, B_1, \cdots, B_I$ be the binary strings

$$B_i = \begin{cases} E_2(T_i) \cdots E_2(T_{I-1})E_3(T_I), & 0 \le i \le I-1 \\ E_3(T_I), & i = I. \end{cases}$$

The decoder then recursively determines $T_0, T_1, \cdots, T_I$ from B^* according to the equations

$$|x| = D_1(B^*) \tag{2.21}$$

$$B_i = \begin{cases} B^* - E_1(|x|), & i = 0 \\ B_{i-1} - E_2(T_{i-1}), & 1 \le i \le I \end{cases} \tag{2.22}$$

$$|T_i| = \begin{cases} \lfloor |x|/r^I \rfloor, & i = 0 \\ r\langle T_{i-1} \rangle \\ \quad + \left\lfloor \dfrac{|x| - r^{I-i+1}\lfloor |x|/r^{I-i+1} \rfloor}{r^{I-i}} \right\rfloor, & 1 \le i \le I-1 \end{cases} \tag{2.23}$$

$$T_i = \begin{cases} D_2(|T_i|, B_i), & 0 \le i \le I-1 \\ D_3(B_I), & i = I. \end{cases} \tag{2.24}$$

The order in which these recursions are performed may not be clear to the reader. We point out that (B_{i-1}, T_{i-1}) determines (B_i, T_i) as follows. First, (B_{i-1}, T_{i-1}) is used to compute B_i via (2.22). Then, $|T_i|$ is determined from T_{i-1} via (2.23). Finally, $|T_i|$ and B_i are jointly used to compute T_i via (2.24).

III. Counting Tokens

As in the previous section, the integer $r \ge 2$ and the nonnegative integer I are fixed throughout this section. Let $(T_0, T_1, \cdots, T_I)$ be the multilevel representation of an A-string x of length $\ge r^I$, generated by the algorithm MPM(r, I). In order to judge the performance of the algorithm MPM(r, I), we shall need to obtain a bound on

$$|T_0| + |T_1| + \cdots + |T_I|.$$

The purpose of this section is to prove the following theorem, which gives us the desired bound.

Theorem 1: Let $n \ge \max(2, r^I)$. Define $C(r, I, |A|, n)$ to be the constant

$$C(r, I, |A|, n) \triangleq 2nr^{-I} + 2Ir + 2r|A|^r + 4r|A|^5 + 8 \log |A| r^2 \left(\frac{n}{\log n} \right).$$

Let x be any A-string of length n and let, $(T_0, T_1, \cdots, T_I)$ be the multilevel representation of x generated by the algorithm MPM (r, I). Then

$$|T_0| + |T_1| + \cdots + |T_I| \le C(r, I, |A|, n). \tag{3.25}$$

We lay some groundwork that shall be necessary for proving Theorem 1. We define an r-set to be any subset J of $\{1, 2, 3, \cdots\}$ such that

- J consists of consecutive integers; i.e., J is of the form
$$J = \{a, a+1, a+2, \cdots, b-1, b\}.$$
- The smallest integer a in J and the largest integer b in J satisfy
$$a = jr^i + 1$$
$$b = (j+1)r^i$$
for integers $i, j \ge 0$.

Given any two distinct r-sets J_1 and J_2, exactly one of the following statements must be true:

- J_1 and J_2 are disjoint or
- one of the two sets J_1, J_2 is properly contained in the other.

Also, it can be seen that the following two properties are true:

- Given any r-set J, there is a unique r-set J' with $r|J|$ elements that properly contains J. We shall call J' the *father* of J and shall denote J' by $\mathrm{fa}(J)$.
- Given any r-set J containing more than one element, there are exactly r r-sets $J_1, J_2, \cdots, J_r$ of size $|J|/r$ which are contained in J (i.e., have J as their father). We shall call these r-sets $J_1, J_2, \cdots, J_r$ the *children* of J.

Let n be an integer satisfying $n \geq r^I$. We define three families of r-sets $\mathcal{J}(r, I|n)$, $\mathcal{J}_0(r, I|n)$, and $\mathcal{J}_1(r, I|n)$ as follows:

$$\mathcal{J}(r, I|n) \triangleq \{J \text{ is an } r\text{-set: } |J| \leq r^I, J \subset \{1, 2, \cdots, n\}\}$$
$$\mathcal{J}_0(r, I|n) \triangleq \{J \in \mathcal{J}(r, I|n)\text{: } \mathrm{fa}(J) \notin \mathcal{J}(r, I|n)\}$$
$$\mathcal{J}_1(r, I|n) \triangleq \{J \in \mathcal{J}(r, I|n)\text{: } \mathrm{fa}(J) \in \mathcal{J}(r, I|n)\}.$$

Example 8: We denote an r-set $\{a, a+1, \cdots, b\}$ by $[a, b]$, where a is the smallest of the elements of the r-set, and b is the largest of the elements. Let $r = 2$, $I = 4$, and $n = 23$. The r-sets comprising $\mathcal{J}_0(r, I|n)$ are

$$\begin{aligned}&[1, 16]\\&[17, 20]\\&[21, 22]\\&[23, 23].\end{aligned} \tag{3.26}$$

The r-sets comprising $\mathcal{J}_1(r, I|n)$ are

$$\begin{aligned}&[1, 8], [9, 16]\\&[1, 4], [5, 8], [9, 12], [13, 16]\\&[1, 2], [3, 4], [5, 6], [7, 8], \cdots, [15, 16], [17, 18], [19, 20]\\&[1, 1], [2, 2], \cdots, [20, 20], [21, 21], [22, 22].\end{aligned} \tag{3.27}$$

Let $x = x_1x_2\cdots x_n$ be an A-string of length $n \geq r^I$. If $J = [a, b]$ is any r-set which is a subset of $\{1, 2, \cdots, n\}$, then

- we let $x(J)$ denote the substring $x_ax_{a+1}\cdots x_b$ of x;
- we say that J is *x-redundant* if there is an r-set J' further to the left of J on the real line such that $x(J') = x(J)$;
- we say that J is *x-innovative* if J is not x-redundant.

We define $\mathcal{J}(r, I|x)$ to be the following collection of r-sets:

$$\mathcal{J}(r, I|x) \triangleq \mathcal{J}_0(r, I|n) \cup \{J \in \mathcal{J}_1(r, I|n) : \mathrm{fa}(J) \text{ is } x\text{-innovative}\}.$$

Each r-set in $\mathcal{J}(r, I|x)$ either has no children in $\mathcal{J}(r, I|x)$, or else it has exactly r children in $\mathcal{J}(r, I|x)$; for later use, we term the *leaves* of $\mathcal{J}(r, I|x)$ to be those r-sets in $\mathcal{J}(r, I|x)$ that have no children in $\mathcal{J}(r, I|x)$.

The set $\mathcal{J}(r, I|x)$ is important for the following reason. Suppose we apply the algorithm MPM(r, I) to an A-string x, and let $S_0(x), S_1(x), \cdots, S_I(x)$ be the sequences of substrings of x arising from the multilevel decomposition phase of the algorithm MPM(r, I). From the manner in which these sequences were defined earlier in the paper, the following lemma, presented without proof, is clear.

Lemma 1: Let x be any A-string of length at least r^I. Let $0 \leq i \leq I$, and let $K_1, K_2, \cdots, K_m$ be the members of $\mathcal{J}(r, I|x)$ of cardinality r^{I-i}, ordered according to their left-to-right appearances as subsets of the real line. Then

$$S_i(x) = (x(K_1), x(K_2), \cdots, x(K_m)). \tag{3.28}$$

Example 8 (Continued): As before, we take $r = 2$, $I = 4$, $n = 23$. Let $A = \{0, 1\}$, and let x be the A-string of length 23 given by (2.4). Strike out from $\mathcal{J}_1(r, I|n)$ in (3.27) all r-sets whose fathers are x-redundant. This leaves us with

$$\begin{aligned}&[1, 8], [9, 16]\\&[1, 4], [5, 8]\\&[1, 2], [3, 4], [5, 6], [7, 8]\\&[1, 1], [2, 2], [7, 7], [8, 8].\end{aligned} \tag{3.29}$$

For example, [9, 12] and [17, 18] were eliminated because their fathers are, respectively, [9, 16] and [17, 20] and

$$\begin{aligned}x([9, 16]) &= x([1, 8]) = 00000001\\ x([17, 20]) &= x([5, 8]) = 0001.\end{aligned}$$

Combining the r-sets (3.26) with the r-sets (3.29), we obtain the following members of $\mathcal{J}(r, I|x)$:

$$\begin{aligned}&[1, 16]\\&[1, 8], [9, 16]\\&[1, 4], [5, 8], [17, 20]\\&[1, 2], [3, 4], [5, 6], [7, 8], [21, 22]\\&[1, 1], [2, 2], [7, 7], [8, 8], [23, 23].\end{aligned}$$

Applying Lemma 1, we have

$$\begin{aligned}S_0(x) &= (x([1, 16])) = (0000000100000001)\\
S_1(x) &= (x([1, 8]), x([9, 16])) = (00000001, 00000001)\\
S_2(x) &= (x([1, 4]), x([5, 8]), x([17, 20]))\\
&= (0000, 0001, 0001)\\
S_3(x) &= (x([1, 2]), x([3, 4]), x([5, 6]), x([7, 8]), x([21, 22]))\\
&= (00, 00, 00, 01, 01)\\
S_4(x) &= (x([1, 1]), x([2, 2]), x([7, 7]), x([8, 8]), x([23, 23]))\\
&= (0, 0, 0, 1, 1).\end{aligned}$$

This confirms the results obtained in Example 3.

Letting $(T_0, T_1, \cdots, T_I)$ be the multilevel representation of x which results from the application of the algorithm MPM(r, I) to x, Lemma 1 tells us that

$$|T_0| + |T_1| + \cdots + |T_I| = |\mathcal{J}(r, I|x)|. \tag{3.30}$$

In view of (3.30), we see that Theorem 1 can be established by bounding the cardinality of the set $\mathcal{J}(r, I|x)$. We shall be able

to do this by first bounding the cardinality of the set of leaves of $\mathcal{J}(r, I|x)$.

Lemma 2: Let x be an A-string of length at least r^I, and let $N(\mathcal{J}(r, I|x))$ be the number of leaves of $\mathcal{J}(r, I|x)$. Then

$$|\mathcal{J}(r, I|x)| \leq 2N(\mathcal{J}(r, I|x)). \tag{3.31}$$

Proof: Let M be the integer obtained by counting all the children of all the elements of $\mathcal{J}(r, I|x)$ which are not leaves. Since every element of $\mathcal{J}(r, I|x)$ which is not a leaf has exactly r children, we have

$$M = r[|\mathcal{J}(r, I|x)| - N(\mathcal{J}(r, I|x))]. \tag{3.32}$$

All of the children counted to obtain M belong to $\mathcal{J}(r, I|x)$ but do not belong to $\mathcal{J}_0(r, I|x|)$; on the other hand, every element of $\mathcal{J}(r, I|x)$ which is not in $\mathcal{J}_0(r, I|x|)$ is one of the children of some member of $\mathcal{J}(r, I|x)$, and therefore entered into the computation of M. This gives us the equation

$$M = |\mathcal{J}(r, I|x)| - |\mathcal{J}_0(r, I|x|)|. \tag{3.33}$$

Equating the right sides of (3.32) and (3.33), one readily obtains (3.31).

Proof of Theorem 1: Let $n \geq \max(2, r^I)$, and let x be a fixed (but arbitrary) A-string of length n. In view of (3.30), relationship (3.25) of Theorem 1 is valid provided we can show that

$$|\mathcal{J}(r, I|x)| \leq C(r, I, |A|, n). \tag{3.34}$$

In order to establish (3.34), we first show that

$$|\mathcal{J}_0(r, I|n)| \leq nr^{-I} + rI. \tag{3.35}$$

Let $J \in \mathcal{J}_0(r, I|n)$. Then one of the conditions i), ii) below must hold:

i) $|J| = r^I$ or

ii) $|J| < r^I$, and $\mathrm{fa}\,(J)$ contains the integer $n + 1$.

If J satisfies condition i), there are at most nr^{-I} possibilities for J, as there are no more than nr^{-I} r-sets of cardinality r^I contained in $\{1, 2, \cdots, n\}$. If J satisfies ii), there are at most I possibilities for $\mathrm{fa}\,(J)$ (the cardinality of $\mathrm{fa}\,(J)$ must be one of the I numbers $r, r^2, \cdots, r^I$); since each of these possibilities for the father of J has r children, there are at most rI possibilities for J under condition ii). We conclude that (3.35) holds.

Let $\mathcal{J}^*(r, I|x)$ be the set of leaves of $\mathcal{J}(r, I|x)$. Define $\mathcal{J}^1$ and $\mathcal{J}^2$ to be the following subsets of $\mathcal{J}_1(r, I|n)$:

$$\mathcal{J}^1 \triangleq \{J \in \mathcal{J}_1(r, I|n) : \mathrm{fa}\,(J) \text{ is } x\text{-innovative}, \ J \text{ is } x\text{-redundant}\}$$
$$\mathcal{J}^2 \triangleq \{J \in \mathcal{J}_1(r, I|n) : |J| = 1, \ \mathrm{fa}\,(J) \text{ is } x\text{-innovative}\}. \tag{3.36}$$

Observe that

$$\mathcal{J}^*(r, I|x) \subset \mathcal{J}_0(r, I|n) \cup \mathcal{J}^1 \cup \mathcal{J}^2. \tag{3.37}$$

It is clear that

$$|\mathcal{J}^2| \leq r|A|^r. \tag{3.38}$$

In the Appendix, we show that

$$|\mathcal{J}^1| \leq 2r|A|^5 + 4 \log |A| r^2 \left(\frac{n}{\log n}\right). \tag{3.39}$$

Applying (3.35), (3.39), and (3.38) to (3.37), we conclude that

$$\begin{aligned} |\mathcal{J}^*(r, I|x)| &\leq nr^{-I} + Ir + r|A|^r + 2r|A|^5 \\ &\quad + 4 \log |A| r^2 \left(\frac{n}{\log n}\right) \\ &= C(r, I, |A|, n)/2 \end{aligned}$$

from which (3.34) follows via an application of Lemma 2.

We conclude this section by presenting Lemma 3 below, which shall be needed later on. Lemma 3 and subsequent parts of the paper employ the following notation: if $u = (u_1, u_2, \cdots, u_k)$ is any sequence (over any alphabet), we let $\tilde{u}$ denote the sequence obtained from u by striking from u each entry u_i which is not equal to any entry of u further to the left of u_i. (It could be that $\tilde{u}$ is an empty sequence.)

Lemma 3: Let x be an A-string of length at least r^I. Let $J_1, J_2, \cdots, J_k$ be the ordering of the leaves of $\mathcal{J}(r, I|x)$ according to the left-to-right ordering of these r-sets as subsets of the real line.

a) The string x is the concatenation of the strings $x(J_1), x(J_2), \cdots, x(J_k)$.

b) Let $0 \leq i < I$, and let $S_i = S_i(x)$ be the sequence of substrings of x of length r^{I-i} defined in Section II. Then $\tilde{S}_i$ is the subsequence of $(x(J_1), \cdots, x(J_k))$ consisting of all entries $x(J_{k'})$ of this sequence for which $J_{k'}$ has cardinality r^{I-i}.

Lemma 3 is proved in the Appendix.

IV. Compression Performance

In this section, we shall first establish a bound telling us how well the algorithm MPM(r, I) compresses A-strings of length at least r^I. The bound (Theorem 2) is an entropy bound, showing that the MPM(r, I) codeword length can be bounded above by an expression involving empirical entropy. We then turn our attention to the redundancy performance of the MPM lossless data compression code. The MPM code is a universal code, formally defined in Section IV-B, built by letting the parameter I in the algorithm MPM(r, I) vary appropriately with the data length. Redundancy is a figure of merit commonly used in source coding theory to evaluate lossless data compression codes. In particular, the redundancy of the MPM data compression code is a measure of the deviation of MPM code compression performance from the optimal performance achievable for some class of lossless codes (a precise definition of redundancy shall be given later on). We shall obtain specific redundancy bounds for the MPM data compression code (Theorems 3 and 4), which can be regarded as the key contributions of this paper.

A. Entropy Bound

In this subsection, the integers $r \geq 2$ and $I \geq 0$ are fixed. Let $\tilde{L}(x|r, I)$ be the length of the binary codeword assigned by the algorithm MPM(r, I) to an A-string x of length at least r^I.

We shall obtain an upper bound on $\tilde{L}(x|r, I)$ which involves the finite-context empirical entropies of the string x. We begin by explaining what these entropies are. Let s be a fixed positive integer. Let $\mathcal{I}(s)$ be the set $\{1, 2, \cdots, s\}$. Let $\mathcal{P}_s(A)$ be the family of all functions p from $\mathcal{I}(s) \times A \times \mathcal{I}(s)$ into $[0, 1]$ such that

- for each $(u, a, u') \in \mathcal{I}(s) \times A \times \mathcal{I}(s)$, the value of the function p at (u, a, u') is denoted $p(u, a|u')$;
- for each $u' \in \mathcal{I}(s)$
$$\sum_{(u, a) \in \mathcal{I}(s) \times A} p(u, a|u') = 1.$$

Definition: Let $x = x_1 x_2 \cdots x_n$ be any A-string. The s-context unnormalized empirical entropy of x is the nonnegative real number $H^s(x)$ defined by

$$H^s(x) \triangleq \inf_{p \in \mathcal{P}_s(A)} \min_{u_0 \in \mathcal{I}(s)} - \log \left[\sum_{u_1, u_2, \cdots, u_n \in \mathcal{I}(s)} \prod_{i=1}^{n} p(u_i, x_i | u_{i-1}) \right].$$

The quantities $\{H^s(x): s = 1, 2, \cdots\}$ are the finite-context empirical entropies of x. The following theorem bounds the codeword length assigned to a data string by the algorithm MPM(r, I) in terms of the finite-context empirical entropies of the data string.

Theorem 2: Let s be any positive integer. Then, for any integer $n \geq \max(2, r^I)$, and any A-string x of length n

$$\tilde{L}(x|r, I) \leq H^s(x) + 2\lceil \log(n+1) \rceil + 2r|A|^{r+1} + (2 + \log s) C(r, I, |A|, n). \quad (4.40)$$

In order to prove Theorem 2, we shall employ another concept of entropy called zeroth-order entropy. If $u = (u_1, u_2, \cdots, u_k)$ is any nonempty sequence of finite length (over any alphabet), we define the *zeroth-order entropy* of u to be the nonnegative real number

$$H_0(u) \triangleq \sum_{i=1}^{k} - \log \frac{\eta(u_i|u)}{k}$$

where $\eta(a|u)$ denotes the number of times that the symbol a appears as an entry in the sequence u. If u is an empty sequence, in our later work it is convenient for us to define the zeroth-order entropy $H_0(u)$ of u to be equal to zero.

We state two properties of zeroth-order entropy which are needed in proving Theorem 2. The simple proofs of these properties are omitted.

Property 1: Let $u = (u_1, u_2, \cdots, u_k)$ and $v = (\phi(u_1), \phi(u_2), \cdots, \phi(u_k))$ be sequences, where ϕ is a one-to-one mapping. Then

$$H_0(u) = H_0(v).$$

Property 2: Let $u = (u_1, u_2, \cdots, u_k)$ be any sequence, and let q be any probability distribution on $\{u_1, u_2, \cdots, u_k\}$. Then

$$H_0(u) \leq \sum_{i=1}^{k} - \log q(u_i).$$

Proof of Theorem 2: Let $u = (u_1, u_2, \cdots, u_k)$ be a string in $t(\mathcal{S}_r(A))$ of length $k > 1$. The key to the proof is an examination of the codeword length $|E_2(u)|$. Let j be the number of distinct entries of u; then

$$\{t_0, t_1, \cdots, t_{j-1}\} = \{u_1, u_2, \cdots, u_k\}.$$

From (2.18) and (2.19) it can be seen that (see (4.41) at the bottom of this page). Notice that

$$\log \left[\frac{(k-1)!}{(j-1)!\,(k-j)!} \right] \leq k - 1. \quad (4.42)$$

Also, by [3, Lemma 2.3]

$$\log \left[\frac{(k-j)!}{(\eta(t_0|u)-1)!(\eta(t_1|u)-1)! \cdots (\eta(t_{j-1}|u)-1)!} \right] \leq H_0(\tilde{u}). \quad (4.43)$$

Applying (4.42) and (4.43) to (4.41), we obtain

$$|E_2(u)| \leq 2|u| + H_0(\tilde{u}). \quad (4.44)$$

Let $n \geq \max(2, r^I)$, and let x be an A-string of length n. Let $(T_0, T_1, \cdots, T_I)$ be the multilevel representation of x generated by applying the algorithm MPM(r, I) to the string x. Referring to (2.20), (4.44), and (2.16), we see that

$$\tilde{L}(x|r, I) \leq 2\lceil \log(n+1) \rceil + \sum_{i=1}^{I-1} \{2|T_i| + H_0(\tilde{T}_i)\} + |E_3(T_I)|. \quad (4.45)$$

We have

$$|E_3(T_I)| = |T_I| \lceil \log |A| \rceil \leq (r|A|^r + r - 1) \lceil \log |A| \rceil \leq 2r|A|^{r+1}. \quad (4.46)$$

For each $0 \leq i < I$, let $S_i = S_i(x)$ be the sequence of substrings of x of length r^{I-i} defined in Section II. By Property 1 of zeroth-order entropy

$$H(\tilde{S}_i) = H(\tilde{T}_i), \qquad 0 \leq i < I. \quad (4.47)$$

From (4.47), (4.46), (4.45), and Theorem 1, we conclude that

$$\tilde{L}(x|r, I) \leq \left\{ \sum_{i=1}^{I-1} H_0(\tilde{S}_i) \right\} + 2\lceil \log(n+1) \rceil + 2r|A|^{r+1} + 2C(r, I, |A|, n). \quad (4.48)$$

$$|E_2(u)| \leq k + 1 + \log \left[\frac{(k-1)!}{(j-1)!(\eta(t_0|u)-1)!(\eta(t_1|u)-1)! \cdots (\eta(t_{j-1}|u)-1)!} \right]. \quad (4.41)$$

Pick $p \in \mathcal{P}_s(A)$ such that

$$H^s(x) = \min_{u_0 \in \mathcal{I}(s)} -\log \left[\sum_{u_1, u_2, \cdots, u_n \in \mathcal{I}(s)} \prod_{i=1}^{n} p(u_i, x_i | u_{i-1}) \right].$$

For each string $y = y_1 y_2 \cdots y_k$ in A^+, define

$$\tau(y) \triangleq \max_{u_0 \in \mathcal{I}(s)} \sum_{u_1, u_2, \cdots, u_k \in \mathcal{I}(s)} \prod_{i=1}^{k} p(u_i, y_i | u_{i-1}).$$

For each positive integer m, there is a positive constant $\Sigma_m \le s$ and a probability distribution q_m on A^m such that

$$q_m(y) = \frac{\tau(y)}{\Sigma_m}, \qquad y \in A^m.$$

Suppose that $y \in A^+$, and that $y^1, y^2, \cdots, y^J$ are substrings of y of possibly varying lengths which, when concatenated together, yield y. It is not hard to see that

$$\tau(y) \le \tau\left(y^1\right) \tau\left(y^2\right) \cdots \tau\left(y^J\right). \tag{4.49}$$

By Lemma 3, we may find substrings $y^1, y^2, \cdots, y^J$ of x such that

- x is obtained when $y^1, y^2, \cdots, y^J$ are concatenated together in the given order.
- For each $i = 0, 1, \cdots, I-1$, the sequence $\tilde{S}_i$ is the sequence whose entries are the members of the list $y^1, y^2, \cdots, y^J$ of length r^{I-i}, in the order of their appearance in this list. (If there are no such entries, then $\tilde{S}_i$ is the empty sequence.)

From Property 2 of zeroth-order entropy

$$\begin{aligned} H_0(\tilde{S}_i) &\le \sum_{\{j:\, |y^j| = r^{I-i}\}} -\log q_{r^{I-i}}\left(y^j\right) \\ &\le |\tilde{S}_i| \log s + \sum_{\{j:\, |y^j| = r^{I-i}\}} -\log \tau\left(y^j\right), \\ &\qquad 0 \le i \le I-1. \end{aligned} \tag{4.50}$$

Summing over i in (4.50) and using (4.49) as well as Theorem 1

$$\begin{aligned} \sum_{i=0}^{I-1} H_0(\tilde{S}_i) &\le C(r, I, |A|, n) \log s + \sum_{j=1}^{J} -\log \tau\left(y^j\right) \\ &\le C(r, I, |A|, n) \log s - \log \tau(x). \end{aligned} \tag{4.51}$$

Applying (4.51) to (4.48), and using the fact that $-\log \tau(x) = H^s(x)$, we obtain (4.40).

B. Redundancy Bounds

In this subsection, we shall make precise some different notions of redundancy for a lossless data compression code, and shall establish some redundancy bounds for the MPM data compression code (defined below).

Let us first give a formal definition of the concept of lossless data compression code. Let n be a positive integer. A lossless *encoder–decoder pair* on A^n is a pair of mappings (ϕ_n, δ_n) such that

- ϕ_n, the *encoder* of the encoder–decoder pair, is a one-to-one mapping from A^n into $\{0, 1\}^+$.
- δ_n, the *decoder* of the encoder–decoder pair, is the inverse mapping for ϕ_n (i.e., the mapping δ_n from $\phi_n(A^n)$ into A^n such that $\delta_n(u)$ is the unique x such that $\phi_n(x) = u$).

Notice that the decoder half of an encoder–decoder pair is uniquely determined once the encoder half has been specified. Suppose that for some fixed positive integer n_0, a lossless encoder–decoder pair (ϕ_n, δ_n) has been specified on A^n for each $n \ge n_0$. The family $\{(\phi_n, \delta_n)\colon n \ge n_0\}$ of encoder–decoder pairs is called an *alphabet A lossless data compression code*.

We are now going to formally define the notion of the MPM data compression code. If $r \ge 2$ and $I \ge 1$ are fixed integers, we have discussed the algorithm MPM(r, I) for losslessly compressing every A-string of length $\ge r^I$. We remove the dependence on the code parameter I by requiring that I vary with the data length in a prescribed manner. For each $n \ge r^r$, we define I_n to be the positive integer

$$I_n \triangleq \lfloor \log_r \log_r n \rfloor. \tag{4.52}$$

Fix the integer $r \ge 2$. For each $n \ge r^r$, let $\tilde{\phi}_n^r$ be the one-to-one mapping from A^n into $\{0, 1\}^+$ such that, for $x \in A^n$, $\tilde{\phi}_n^r(x) \in \{0, 1\}^+$ is the codeword (2.20) assigned to x by the algorithm MPM(r, I_n) (this codeword is defined because $n \ge r^{I_n}$). Let $\tilde{\delta}_n^r$ be the inverse mapping corresponding to $\tilde{\phi}_n^r$. We define the *MPM data compression code* to be the family of encoder–decoder pairs $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r) : n \ge r^r\}$.

Since we have now removed dependence on the parameter I in the MPM code, we shall use a different notation for codeword length that does not involve I. If $x \in A^n$ and $n \ge r^r$, we let $\tilde{L}_n(x|r)$ denote the length of the codeword assigned to x by the MPM data compression code. In terms of our earlier notation, this means that

$$\tilde{L}_n(x|r) = \tilde{L}(x|r, I_n) = |\tilde{\phi}_n^r(x)|.$$

There are two scenarios in which we shall want to perform redundancy measurements for the MPM data compression code.

Scenario i): In this scenario, called *redundancy relative to a class of codes*, a class of lossless data compression codes is given, and one measures redundancy as the difference between the MPM codeword length and the optimum codeword length achievable via codes in the given class of codes.

Scenario ii): In this scenario, called *redundancy relative to a class of sources*, a class of information sources is given, and one measures redundancy as the difference between the MPM codeword length and the optimum self-information over the given class of sources.

In both redundancy relative to a class of codes (Theorem 3) and redundancy relative to a class of sources (Theorem 4), we examine the behavior of the growth of redundancy as a function of the data length. Results of compression experiments on images are presented at the end of Section IV.

1) Redundancy Relative to a Class of Codes: Throughout this subsection, the integer $r \ge 2$ is fixed. Let s be an arbitrary

positive integer. It is our purpose here to investigate the redundancy of the MPM code $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r)\}$ relative to the class of all s-state arithmetic codes on the alphabet A. Each such s-state arithmetic code is characterized by a triple (u_0, f, p) in which

a) $u_0 \in \mathcal{I}(s)$;
b) f is a function from $\mathcal{I}(s) \times A$ into $\mathcal{I}(s)$;
c) p is a function from $\mathcal{I}(s) \times A$ into the interval of real numbers $\{x: 0 < x \le 1\}$;
d) for each $u \in \mathcal{I}(s)$

$$\sum_{a \in A} p(a|u) = 1$$

where, by convention, we write $p(a|u)$ to denote the value of the function p at (u, a).

For each $n \ge 1$, the s-state arithmetic code induced by the triple (u_0, f, p) encodes each string $x = x_1 x_2 \cdots x_n$ in A^n into a binary codeword of length

$$L_n(x|u_0, f, p) \triangleq 1 + \left\lceil \sum_{i=1}^{n} -\log p(x_i|u_i) \right\rceil$$

where $u_1, u_2, \cdots, u_n$ are generated according to the formula

$$u_i = f(u_{i-1}, x_i), \qquad i = 1, \cdots, n.$$

The optimum codeword length arising from the use of s-state arithmetic codes to encode $x \in A^n$ is the quantity $L_n(x|s)$ defined by

$$L_n(x|s) \triangleq \inf_{(u_0, f, p)} L_n(x|u_0, f, p)$$

where the infimum is over all triples (u_0, f, p) satisfying the criteria a)–d) above. We let $\mathcal{C}_s(A)$ denote the class of all s-state arithmetic codes.

Theorem 3: Let s be an arbitrary positive integer. Then, we have the following redundancy bound for the MPM data compression code:

$$\max_{x \in A^n} n^{-1} \left\{ \tilde{L}_n(x|r) - L_n(x|s) \right\} \le (2 + \log s) D(r, |A|) \left(\frac{1}{\log n} \right) \qquad \forall\, n \ge r^r \quad (4.53)$$

where $D(r, |A|)$ is the positive constant

$$D(r, |A|) \triangleq 4 + 4r + 2r \log r + 3r|A|^{r+1} + 4r|A|^5 + 8r^2 \log |A|.$$

Discussion. The left-hand side of (4.53) is the *maximal redundancy/sample* of the MPM data compression code relative to the class of codes $\mathcal{C}_s(A)$. Theorem 3 tells us that the maximal redundancy/sample is $O(1/\log n)$ as a function of the data length n. To illustrate, for $r = 2$ and $|A| = 2$, we conclude from Theorem 3 that the maximal redundancy/sample is no larger than $352(2 + \log s)/\log n$. (Note: The "352" in this bound is not the best possible—it is an open problem to determine what is the smallest positive constant that will work in place of "352.")

Proof: Since $H^s(x) \le L_n(x|s)$ for an A-string x of length n, we see from Theorem 2 that the left-hand side of (4.53) is upper-bounded by the sum of the following two expressions:

$$\begin{aligned} E_1 &= 2n^{-1}\lceil \log(n+1) \rceil + 2r^{-I_n}(2 + \log s) + 2n^{-1} r I_n (2 + \log s) \\ E_2 &= n^{-1}[3r|A|^{r+1} + 4r|A|^5](2 + \log s) + \left(\frac{8 \log |A| r^2 (2 + \log s)}{\log n} \right). \end{aligned}$$

It is clear how the terms in E_2 enter into terms of $D(r, |A|)$. To complete the proof, we bound the terms of E_1 as follows:

$$\begin{aligned} 2n^{-1}\lceil \log(n+1) \rceil &\le n^{-1}[4 + 2 \log n] \\ &\le \frac{4(2 + \log s)}{\log n} \end{aligned} \quad (4.54)$$

$$\begin{aligned} 2r^{-I_n}(2 + \log s) &\le 2r^{1 - \log_r \log_r n}(2 + \log s) \\ &\le \frac{2r \log r (2 + \log s)}{\log n} \\ 2n^{-1} r I_n (2 + \log s) &\le 2n^{-1} r \log n (2 + \log s) \\ &\le \frac{4r(2 + \log s)}{\log n}. \end{aligned} \quad (4.55)$$

To establish inequalities (4.54) and (4.55), we used the fact that

$$\frac{\log n}{n} \le \frac{2}{\log n}, \qquad n \ge 2.$$

Remark: The code classes $\{\mathcal{C}_s(A): s \ge 1\}$ are not the only classes of codes to which one might want to compare the MPM code. One could also, for each fixed positive integer m, consider the class $\mathcal{C}^m(A)$ of all lossless block to variable-length codes on the alphabet A which have block length m. However, for a sufficiently large s, the class of codes $\mathcal{C}_s(A)$ outperforms the class of codes $\mathcal{C}^m(A)$. Theorem 3 therefore automatically extends to the class of codes $\mathcal{C}^m(A)$. In similar fashion, one can extend Theorem 3 to other classes of codes which are outperformed by one of the classes $\mathcal{C}_s(A)$, $s \ge 1$.

2) Redundancy Relative to a Class of Sources: Throughout this subsection, the integer $r \ge 2$ is fixed. It is our purpose to investigate the redundancy of the MPM code $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r)\}$ relative to classes of finite-state information sources (defined below).

Let A^∞ be the set of all infinite sequences $(x_1, x_2, x_3, \cdots)$ in which each entry x_i is chosen from the alphabet A. For each string $x = x_1 x_2 \cdots x_n \in A^+$, let $[x]$ denote the set of all sequences in A^∞ whose first n terms are $x_1, x_2, \cdots, x_n$. The set A^∞ becomes a measurable space if we endow this set with the sigma field spanned by the sets $\{[x]: x \in A^+\}$. A probability measure μ on A^∞ shall be called an *alphabet A information source* or simply an *alphabet A source*. Let $\Lambda_s(A)$ denote the class of all alphabet A sources μ for which there exists $p \in \mathcal{P}_s(A)$ and $u_0 \in \mathcal{I}(s)$ such that, for every $x = x_1 x_2 \cdots x_n \in A^+$

$$\mu([x]) = \sum_{u_1, \cdots, u_n \in \mathcal{I}(s)} \prod_{i=1}^{n} p(u_i, x_i | u_{i-1}).$$

The sources in the class $\Lambda_s(A)$ are called *finite-state information sources with s states*.

Definition: If μ is an alphabet A information source and $x \in A^+$, then the *self-information of x with respect to the source μ* is defined by

$$I_\mu(x) \stackrel{\Delta}{=} -\log \mu([x]).$$

Theorem 4: Let s be an arbitrary positive integer. Then, we have the following redundancy bound for the MPM data compression code:

$$\max_{x\in A^n} \sup_{\mu\in\Lambda_s(A)} n^{-1}\{\tilde{L}_n(x|r) - I_\mu(x)\} \le (2+\log s)D(r, |A|)\left(\frac{1}{\log n}\right), \qquad \forall n \ge r^r. \quad (4.56)$$

Proof: The minimum of $I_\mu(x)$ over $\mu \in \Lambda_s(A)$ is $H^s(x)$. Therefore, Theorem 4 follows from Theorem 2 in the same way that Theorem 3 followed from Theorem 2.

Discussion. Let Λ be any class of alphabet A information sources. The quantity

$$\max_{x\in A^n} \sup_{\mu\in\Lambda} n^{-1}\left\{\tilde{L}_n(x|r) - I_\mu(x)\right\} \quad (4.57)$$

is called the *maximal redundancy/sample* of the MPM data compression code relative to the class of sources Λ. Theorem 4 tells us that the maximal redundancy/sample of the MPM data compression code is $O(1/\log n)$, relative to each class of sources $\Lambda_s(A)$. Theorem 4 is of interest because the 1978 Lempel–Ziv data compression code [17] is known to have maximal redundancy/sample $O(\log\log n/\log n)$ relative to each class $\Lambda_s(A)$ [14], but it is not known whether its maximal redundancy/sample is $O(1/\log n)$. The MPM data compression code and the Lempel–Ziv data compression code are similar in structure in that they both use pattern matching to represent a data string as a string of pointers used for compression; it is therefore natural to make comparisons between these two compression algorithms.

Remark: Earlier, we defined the MPM code by requiring that the parameter I in the algorithm MPM(r, I) be dependent upon the data length n according to the formula $I = \lfloor \log_r \log_r n \rfloor$. Instead, suppose one weakens the definition of the MPM code to require only that I be a function I_n of n in which

$$C \log\log n \le I_n \le \log_r n \quad (4.58)$$

for n sufficiently large, where C is some positive constant. Then, it can be shown that Theorems 3 and 4 can be extended to yield $O(1/\log n)$ redundancy/sample for this more general notion of MPM code. This fact gives more flexibility to those who may want to implement an MPM code so that good redundancy performance is assured. On the other hand, if one implements the MPM code by selecting I_n to have a slower order of growth than $\log\log n$, the maximal redundancy/sample in Theorems 3 and 4 can decay to zero more slowly than the sequence $1/\log n$ decays to zero; consequently, choosing I_n to have an order of growth at least as fast as the sequence $\log\log n$ is what one should strive for in implementation of an MPM code.

3) Other Redundancy Notions: We discuss the performance of the MPM data compression code with respect to some weaker notions of redundancy than the notion of redundancy used in Theorem 4. Let μ be a fixed alphabet A information source. The quantity

$$\max_{x\in A^n} n^{-1}\left\{\tilde{L}_n(x|r) - I_\mu(x)\right\} \quad (4.59)$$

is called the maximal redundancy/sample of the MPM code relative to the source μ. Theorem 4 automatically implies that the MPM code has maximal/redundancy $O(1/\log n)$ relative to each individual source in the union of the classes of sources $\{\Lambda_s(A): s \ge 1\}$. Savari [15] proved $O(1/\log n)$ behavior of the maximal redundancy of the 1978 Lempel–Ziv code relative to each individual unifilar finite-order Markov source.

Again, let μ be a fixed alphabet A information source. The quantity

$$n^{-1}\int_{A^n}\left\{\tilde{L}_n(x|r) - I_\mu(x)\right\} d\mu(x) \quad (4.60)$$

is called the average redundancy/sample of the MPM code relative to the source μ. The quantity (4.60) is clearly less than or equal to the quantity (4.59). Therefore, Theorem 4 also automatically implies that the MPM code has average redundancy $O(1/\log n)$ relative to each individual source in the union of the classes of sources $\{\Lambda_s(A): s \ge 1\}$. Let $\Lambda_0(\{0, 1\})$ denote the family of all binary memoryless information sources. Louchard and Szpankowski [10] have shown that for all but countably many $\mu \in \Lambda_0(\{0, 1\})$, the average redundancy/sample of the 1978 Lempel–Ziv code relative to μ has an asymptotic expansion of the form

$$\frac{A(\mu)}{\log n} + o\left(\frac{1}{\log n}\right)$$

where $A(\mu)$ is a certain constant depending on μ. It is an open problem whether an analogous result holds for the MPM data compression code.

4) Universal Coding: An alphabet A information source μ is said to be *stationary* if there is a stationary alphabet A stochastic process $(X_1, X_2, X_3, \cdots)$ such that

$$\mu([x]) = \Pr\{X_1 = x_1, X_2 = x_2, \cdots, X_n = x_n\}, \qquad x = x_1x_2\cdots x_n \in A^+. \quad (4.61)$$

The *entropy rate* of a stationary alphabet A source μ is the number $H^*(\mu)$ defined by

$$H^*(\mu) \stackrel{\Delta}{=} \lim_{n\to\infty} n^{-1} \sum_{x\in A^n} -\mu([x]) \log \mu([x]).$$

Theorem 5: Let μ be any stationary alphabet A information source. Then

$$\lim_{n\to\infty} n^{-1}\int_{A^n} \tilde{L}_n(x|r)\, d\mu(x) = H^*(\mu).$$

Discussion. Let $\Lambda_{\rm sta}(A)$ be the class of all alphabet A stationary sources. It is well known that any lossless alphabet A data compression code $\{(\phi_n, \delta_n)\}$ satisfies

$$\lim_{n\to\infty} n^{-1} \int_{A^n} |\phi_n(x)|\, d\mu(x) \geq H^*(\mu), \qquad \forall\, \mu \in \Lambda_{\rm sta}(A). \tag{4.62}$$

Therefore, the codes which perform best on the class of stationary sources $\Lambda_{\rm sta}(A)$ are the ones for which equality holds in (4.62). In the literature, these codes have been given a special name. An alphabet A code $\{(\phi_n, \delta_n)\}$ satisfying

$$\lim_{n\to\infty} n^{-1} \int_{A^n} |\phi_n(x)|\, d\mu(x) = H^*(\mu), \qquad \forall\, \mu \in \Lambda_{\rm sta}(A)$$

is said to be *universal*. Theorem 5 tells us that the MPM data compression code is universal. The Lempel–Ziv code is also universal [17].

Theorem 5 follows from Theorem 4, if one uses standard approximations of stationary sources via finite-state sources [5]. Or, one can use the facts that i) the MPM data compression code belongs to a class of lossless data compression codes called asymptotically compact grammar-based codes [6]; and ii) every asymptotically compact grammar-based code is universal [6].

5) Compression Experiments: A version of the MPM code called the quadrisection code (QUAD, for short) has been developed for lossless image compression [8], [13], [9]. First, one scans a $2^n \times 2^n$ image M to obtain a one–dimensional (1-D) string x_M of length 4^n, so that for each $1 < j < n$, the $2^j \times 2^j$ subblocks of M arising from a partitioning of M correspond to disjoint substrings of length 4^j of the 1-D string x_M. For example, the pixels of an 8×8 image are scanned in the following order:

$$\begin{array}{cccccccc}
1 & 2 & 5 & 6 & 17 & 18 & 21 & 22\\
3 & 4 & 7 & 8 & 19 & 20 & 23 & 24\\
9 & 10 & 13 & 14 & 25 & 26 & 29 & 30\\
11 & 12 & 15 & 16 & 27 & 28 & 31 & 32\\
33 & 34 & 37 & 38 & 49 & 50 & 53 & 54\\
35 & 36 & 39 & 40 & 51 & 52 & 55 & 56\\
41 & 42 & 45 & 46 & 57 & 58 & 61 & 62\\
43 & 44 & 47 & 48 & 59 & 60 & 63 & 64.
\end{array}$$

Such a scanning is called a *quadrisection scanning*. One then applies the MPM code to the 1-D string x_M with $r = 4$. In the recursive generation of the multilevel representation of x_M, substrings of x_M of length a power of four are partitioned into four substrings of equal length, which corresponds to a partitioning of a corresponding subblock of M into four subblocks of equal size, namely, the NW, NE, SW, and SE corners of M.

The quadrisection code compresses binary images particularly well. In Table I, we report the results of compression experiments on binary 512×512 archival images. Each image was originally a 256-level image, from which the principal bit plane was extracted as input to QUAD. As can be seen from the table, QUAD compression is competitive with JBIG, one of the best binary image compressors. The gap between QUAD and JBIG lessens as one increases the size of the image beyond 512×512, confirming the theory developed in this paper that tells us that QUAD should compress high-resolution binary images well. These compression experiments were run on a SUN SparcStation using a C program. The running time for one compression run on a 512×512 binary image was reasonably fast (no more than about 4 s)—improvements in running time are no doubt possible, since our programs were not optimized for speed.

TABLE I
COMPRESSION RESULTS IN BITS/PIXEL FOR 512×512 BINARY IMAGES

IMAGE	QUAD	JBIG
lena	0.2043	0.1633
airplane	0.1485	0.1154
baboon	0.5814	0.5276
barbara	0.3042	0.2513
peppers	0.1722	0.1307
sailboat	0.2004	0.1553
splash	0.0706	0.0589
tiffany	0.0242	0.0264

TABLE II
MULTILEVEL DECOMPOSITION OF 256×256 BINARY LENA IMAGE

size of block	distinct blocks
256 × 256	1
128 × 128	4
64 × 64	16
32 × 32	63
16 × 16	207
8 × 8	496
4 × 4	639
2 × 2	16

The quadrisection code can easily be modified for progressive resolution scalable binary image compression [9] by transmitting additional information to the decoder. For each distinct image subblock of size bigger than 1×1 generated in the multilevel decomposition phase, a single bit can be used to tell the decoder what binary level to use for that subblock. To illustrate, we refer the reader to Table II, which gives the number of distinct subblocks of each size in the principal bit plane of the 256×256 Lena image (as reported in [8]). The total number of distinct blocks of all sizes is 1442. Transmitting an extra 1442 bits only increases the compression rate in bits per pixel by $1442/65536 = 0.0220$.

The MPM code, modified for progressive data reconstruction, will still have an $O(1/\log n)$ maximal redundancy bound. This is because the number of additional code bits transmitted to the decoder will be proportional to the number of distinct substrings of all orders that are represented in the multilevel representation. By Theorem 1, this number is $O(n/\log n)$; dividing by n, the increase in compression rate is only $O(1/\log n)$.

V. COMPLEXITY

Fix $r \geq 2$ throughout this section. In the preceding section, we showed that the maximal redundancy/sample of the MPM

code $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r)\}$ is $O(1/\log n)$. In this section, we address the question of the complexity of the MPM code. We shall show (Theorem 6) that the MPM code is linear in both time complexity and space complexity, as a function of the data length. We shall prove our linear complexity result by using a special-purpose machine on which the multilevel representation of a data string of length n is computed via $O(n)$ computational cycles of the machine, while using $O(n)$ storage cells in the working space of the machine.

We put forth some background material needed to accomplish the goals of this section. Let the alphabet A be denoted as $A = \{a_0, a_1, \cdots, a_{|A|-1}\}$. We let $\tau(A)$ be the infinite tree characterized as follows.

- There is a unique root of the tree $\tau(A)$, which shall be denoted v^*.
- Each vertex v of the tree $\tau(A)$ has exactly $|A|$ edges emanating from it; these edges terminate at vertices which are called the children of v. The children of v shall be referred to as child a_0, child $a_1, \cdots$, child $a_{|A|-1}$.
- For each vertex $v \neq v^*$ of the tree $\tau(A)$, there is a unique vertex v_f of $\tau(A)$ such that v is a child of v_f. The vertex v_f shall be called the father of v.
- For each vertex $v \neq v^*$ of the tree $\tau(A)$, there is a unique positive integer n and unique vertices $v_0, v_1, \cdots, v_n$ such that $v_0 = v^*$, $v_n = v$, and v_{i-1} is the father of v_i for $1 \leq i \leq n$.

For each string $x = x_1 x_2 \cdots x_n \in A^+$, there is a unique path $\pi(x)$ in $\tau(A)$ consisting of $n+1$ vertices $v_0, v_1, \cdots, v_n$ of $\tau(A)$ such that

- $v_0 = v^*$.
- For each $1 \leq i \leq n$, vertex v_i is child x_i of vertex v_{i-1}.

Definition: Let $V(\tau(A))$ denote the set of vertices of the tree $\tau(A)$. Let b be a symbol which is not a member of the set of tokens $\{t_0, t_1, t_2, \cdots\}$ (b will serve as a "blank symbol"). A *labeling* of the tree $\tau(A)$ is any mapping α from $V(\tau(A))$ into the set $\{b, t_0, t_1, t_2, \cdots\}$ in which

- $\alpha(v^*) \neq b$.
- $\alpha(v) \neq b$ for only finitely many vertices v.
- If $v \neq v^*$, then $\alpha(v_f) \neq b$ whenever $\alpha(v) \neq b$.

We let α^* denote the special labeling of $\tau(A)$ in which

$$\alpha^*(v) = \begin{cases} t_0, & v = v^* \\ b, & v \neq v^*. \end{cases}$$

We describe a machine M on which our calculations shall be performed. A *configuration* of the machine M is defined to be a pair (v, α) in which $v \in V(\tau(A))$ and α is a labeling of $\tau(A)$. In one computational cycle of the machine M, the machine moves from one configuration to another. Suppose (v, α) is the configuration of the machine M at the beginning of a computational cycle. The control head of the machine views the content of a storage cell C_v located at vertex v. If $\alpha(v) \neq b$, then the content of the cell C_v is the token $\alpha(v) \in \{t_0, t_1, \cdots\}$; otherwise, C_v is empty. During execution of the computational cycle, the content of C_v might be changed, or the content of C_v might remain unchanged while the control head repositions itself at a vertex adjacent to v (meaning the vertex v_f or one of the children of v). The movement of the machine M during the computational cycle is accomplished by means of an *instruction*. We specify the set of instructions for the machine M as follows:

i) Let (v, α) be a configuration in which $v \neq v^*$. Let α_1 be the labeling of $\tau(A)$ in which

$$\alpha_1(u) = \begin{cases} \alpha(u), & u \in V(\tau(A)),\, u \neq v \\ \alpha(v_f), & u = v. \end{cases}$$

The instruction

"Move from configuration (v, α) to configuration (v, α_1)."

belongs to the instruction set for the machine M. An instruction of this type shall be called a Type i) instruction.

ii) Let (v^*, α) be a configuration in which $\alpha(v^*) = t_i$. Let α_2 be the labeling of $\tau(A)$ in which

$$\alpha_2(u) = \begin{cases} \alpha(u), & u \in V(\tau(A)),\, u \neq v^* \\ t_{i+1}, & u = v^*. \end{cases}$$

The instruction

"Move from configuration (v^*, α) to configuration (v^*, α_2)."

belongs to the instruction set for the machine M. An instruction of this type shall be called a Type ii) instruction.

iii) Let (v, α) be any configuration, and let v_c be any child of v. The instruction

"Move from configuration (v, α) to configuration (v_c, α)."

belongs to the instruction set for the machine M. An instruction of this type shall be called a Type iii) instruction.

iv) Let (v, α) be any configuration in which $v \neq v^*$. The instruction

"Move from configuration (v, α) to configuration (v_f, α)."

belongs to the instruction set for the machine M. An instruction of this type shall be called a Type iv) instruction.

v) The instruction set for the machine M includes only those instructions specified in i)–iv) above.

We describe now a program for computing the tokenization function t of Section II-B on the machine M. The program accepts as input any positive integer k and any k strings $x^{(1)}$, $x^{(2)}, \cdots, x^{(k)}$ in A^+ for which

$$(x^{(1)}, x^{(2)}, \cdots, x^{(k)}) \in \mathcal{S}_r(A).$$

The output of the program is the sequence $t((x^{(1)}, x^{(2)}, \cdots, x^{(k)}))$.

PROGRAM:

1) Read k from machine input tape.
2) Let $i = 1$. Let $\alpha = \alpha^*$.
3) If $i > k$, halt PROGRAM. Otherwise, continue.
4) Read $x^{(i)}$ from the input tape of the machine M.
5) Let $v_1, v_2, \cdots, v_m$ be the nonroot vertices along the path $\pi(x^{(i)})$. Execute machine instructions of Type iii) to move from configuration (v^*, α) to the configuration

(v_m, α) in m computational cycles. (Note that by definition of $\mathcal{S}_r(A)$, we have $m > 1$.)

6) If $\alpha(v_m) \neq b$, go to Line 7). Otherwise, go to Line 8).
7) Print $\alpha(v_m)$ on the machine output tape. Execute machine instructions of Type iv) to move back to the configuration (v^*, α) in m computational cycles. Increase i by one. Go to Line 3).
8) Move to configuration (v_1, α) in $m-1$ computational cycles by executing instructions of Type iv). Move to configuration (v_1, α_1) in one computational cycle by executing an instruction of Type i). Print $\alpha_1(v_1)$ on the machine output tape.
9) Move to configuration (v^*, α_1) in one computational cycle by executing an instruction of Type iv). Move to configuration (v^*, α_2) in one computational cycle by executing an instruction of Type ii).
10) Move to configuration (v_2, α_2) in two computational cycles by executing instructions of Type iii).
11) Move to configuration (v_2, α_3) in one computational cycle by executing an instruction of Type i).
12) Move to configuration (v_m, α_4) in $2(m-2)$ computational cycles by alternately performing instructions of Type iii) and i). (If $m = 2$, nothing needs to be done.)
13) Move to configuration (v^*, α_4) in m computational cycles by performing instructions of Type iv). Update α by setting it equal to α_4, increase i by one, and go to Line 3).

The following lemma is clear from an examination of the PROGRAM.

Lemma 4: Let $(x^{(1)}, x^{(2)}, \cdots, x^{(k)}) \in \mathcal{S}_r(A)$ and let m be the common length of the strings $x^{(1)}, x^{(2)}, \cdots, x^{(k)}$. Then, $t((x^{(1)}, x^{(2)}, \cdots, x^{(k)}))$ can be computed on M in at most $k(5m+1)$ computational cycles.

The computation of the multilevel representation of a data string via the MPM code is the most computationally intense part of the overall MPM code operation. The following lemma addresses the time complexity of this particular task. We adopt the usual convention of measuring the time complexity of a computational task in terms of the number of computational cycles needed to accomplish the task.

Lemma 5: There is a positive constant $C_1(r, |A|)$ such that for every integer $n \geq r^r$, and every string $x \in A^n$, the multilevel representation $(T_0, T_1, \cdots, T_{I_n})$ of x generated by the MPM code $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r): n \geq r^r\}$ can be computed on the machine M via a calculation involving no more than $C_1(r, |A|)n$ computational cycles.

Proof: Parsing off nonoverlapping blocks of length r^{I_n} from x, one obtains

$$S_0(x) = \left(u^1, u^2, \cdots, u^{|T_0|}\right). \tag{5.63}$$

According to Lemma 4, the PROGRAM computes T_0 in at most $|T_0|(5r^{I_n}+1)$ computational cycles. The PROGRAM, during the course of these cycles, can also identify which entries of $S_0(x)$ are new, since a block u^j in $S_0(x)$ has not been seen before if and only if Line 7) of the PROGRAM is not executed while processing u^j. Therefore, as a by-product of the PROGRAM run on the machine M, one obtains $S_1(x)$ (partition each of the new u^j's into r substrings each, and append at most $r-1$ substrings of x that appear in x after $u^{|T_0|}$). Applying the PROGRAM to the entries of $S_1(x)$, we obtain T_1 in at most $|T_1|(5r^{I_n-1}+1)$ computational cycles, and we also obtain $S_2(x)$ as a by-product. Continuing in this way, one sees that $(T_0, T_1, \cdots, T_{I_n})$ is computed by using no more than

$$\sum_{i=0}^{I_n-1} \left(5r^{I_n-i}+1\right)|T_i| \tag{5.64}$$

computational cycles. (Recall from Section II-B that $T_{I_n} = S_{I_n}(x)$, so that T_{I_n-1} and T_{I_n} are both generated during the last $(5r+1)|T_{I_n-1}|$ computational cycles of the machine M.) Using the fact that $r^{I_n} \leq \log n$, we can upper-bound the expression in (5.64) by

$$(5 \log n + 1)(|T_0| + |T_1| + \cdots + |T_{I_n}|).$$

Applying Theorem 1 to this expression yields the desired conclusion.

Now we turn our attention to the problem of quantifying the storage complexity of the computational task of computing the multilevel representation of a data string via the MPM code. Suppose that the machine M goes through the configurations

$$(v^1, \alpha^1), (v^2, \alpha^2), \cdots, \left(v^Q, \alpha^Q\right) \tag{5.65}$$

during the course of a computation. The number of storage cells that are used to store information in the working space of the calculation is the cardinality of the set

$$\bigcup_{1 \leq q \leq Q} \{v: \alpha^q(v) \neq b\}. \tag{5.66}$$

This number shall be our measure of the storage complexity. (We have followed the customary practice of measuring the storage complexity of a computation as the number of storage cells that are employed in the working space of the calculation—this means that we exclude storage space employed to store input data or output data.)

Lemma 6: There is a positive constant $C_2(r, |A|)$ such that for every integer $n \geq r^r$, and every string $x \in A^n$, the multilevel representation $(T_0, T_1, \cdots, T_{I_n})$ of x generated by the MPM code $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r)\}$ can be computed on the machine M using no more than $C_2(r, |A|)n$ storage cells to store data in the working space of the calculation.

Proof: Parsing off nonoverlapping blocks of length r^{I_n} from $x \in A^n$, one obtains (5.63). Let (5.65) be the sequence of configurations of M that result in computing T_0 from $S_0(x)$ using the PROGRAM. A study of the PROGRAM reveals that once a token in $\{t_0, t_1, \cdots\}$ is assigned as a label to a vertex of $\tau(A)$ during some computational cycle of M, then that vertex will carry a token as label throughout the remaining cycles (not necessarily the same token the whole way). In other words, referring to (5.65), if $\alpha^q(v) \neq b$ for some $1 \leq q < Q$, then $\alpha^Q(v) \neq b$. Consequently, the number of storage cells employed during the course of the computation can be simplified from the cardinality of the set (5.66) to the cardinality of the set $\{v: \alpha^Q(v) \neq b\}$. This cardinality can certainly be no bigger than $|T_0|r^{I_n}$, the total of the lengths of the strings $u^1, u^2, \cdots, u^{|T_0|}$ (which is the same as the number of entries

from A on the machine M's input tape). As discussed during the proof of Lemma 5, as a by-product of the computation of $t(S_0(x))$ on the machine M, those entries u^j of $S_0(x)$ in (5.63) are identified which are "new." Those entries are kept on M's input tape, and the remaining entries of $S_0(x)$ are erased from the input tape. Some additional entries are entered on the input tape to represent substrings of x of length r^{I_n-1} appearing after u^J. The input tape now contains $|T_1|r^{I_n-1}$ terms from A, and this information is used to compute T_1 on M using the PROGRAM. By the same argument used above in discussing the computation of T_0, one argues that, in computing T_1, no more than $|T_1|r^{I_n-1}$ storage cells are used in the working space of the machine M (i.e., the number of entries on the input tape). Repeating this argument, one concludes that for each $0 \le i < I_n$, the number of storage cells used in the working space of M to compute T_i is at most $|T_i|r^{I_n-i}$. Moreover, no storage cells are used to compute T_{I_n} other than those used to compute T_{I_n-1} (these two token strings are computed by M together, as pointed out in the proof of Lemma 5). Summing, the total number of storage cells used in the computation of $(T_0, \cdots, T_{I_n})$ can be no bigger than

$$\sum_{i=0}^{I_n-1} |T_i|r^{I_n-i}. \tag{5.67}$$

Using the bound $r^{I_n} \le \log n$, we bound (5.67) as

$$\log n(|T_0| + |T_1| + \cdots + |T_{I_n}|)$$

which, appealing to Theorem 1, gives us our result.

Here is the main result of this section. Since the computational task of computing the multilevel representation of a data string is the most expensive part of the MPM code both in terms of computation time and storage requirements, the result follows from Lemmas 5 and 6.

Theorem 6: The MPM code $\{(\tilde{\phi}_n^r, \tilde{\delta}_n^r) : n \ge r^r\}$ has time complexity $O(n)$ and storage complexity $O(n)$.

Remark: At the end of Section IV-B2, we remarked upon an extension of the MPM code in which it is required that the parameter I in the algorithm MPM(r, I) be dependent upon the data length n according to the inequality (4.58). We remarked that this extended MPM code still yields $O(1/\log n)$ maximal redundancy/sample. It is an open question (suggested by Theorem 6) whether this extended MPM code is of linear time and space complexity.

VI. Conclusions

As a consequence of this paper, we have isolated a data compression code, the MPM code, which, like the Lempel–Ziv code, is of linear time and space complexity, but for which we can prove a better redundancy bound. Let us try to give a reason for this state of affairs. The MPM code and the Lempel–Ziv code are "pure pattern matching" codes in the sense that they do not directly compress the data, but instead form essentially all of their encoder output by compressing pointers pointing between matching patterns in the data. However, the MPM code does its pattern matching on multiple levels, giving the MPM code a hierarchical structure that the Lempel–Ziv code does not possess. This insight seems to suggest that a properly designed hierarchical pattern matching-based data compression code can outperform a nonhierarchical pattern matching-based data compression code.

We conclude with some historical remarks. The first instance of the MPM code was developed for $r = 2$, strictly for data of length a power of two, and named the *bisection algorithm* [12]. However, the implementation of the bisection algorithm given in [12] was computationally inefficient. Subsequent work led to the present efficient implementation of the MPM code (announced in [7]). There are some points of resemblance between the bisection algorithm and the N-gram algorithm of Bugajski and Russo [1], [16]. These are as follows:

- both algorithms are hierarchical in nature;
- both algorithms employ multilevel dictionaries, where each dictionary consists of strings of length a fixed power of two, the length varying from level to level;
- both algorithms generate a parsing of the data into variable-length substrings, each substring of length a power of two.

On the other hand, there are some differences between the two algorithms:

- the algorithms generate their dictionaries differently, and employ them differently;
- the bisection algorithm is universal in the sense of Section IV-B4, whereas the N-gram algorithm has not yet been proved to be universal.

Appendix

We first prove (3.39), needed in the proof of Theorem 1. We then establish Lemma 3, needed in the proof of Theorem 2.

Proof of (3.39): Recall from the proof of Theorem 1 that $n \ge \max(2, r^I)$, and that x is an A-string of length n. Let q be the cardinality of the set $\mathcal{J}^1$ defined by (3.36), and let $J_1, J_2, \cdots, J_q$ be the r-sets comprising $\mathcal{J}^1$. Let $K_1, K_2, \cdots, K_q$ be the fathers of $J_1, J_2, \cdots, J_q$, respectively. Some of the sets $K_1, K_2, \cdots, K_q$ might coincide. However, for each $1 \le i \le q$, there are at most r integers $1 \le u \le q$ such that K_u is the father of J_i. Therefore, we may pick an integer $m \ge q/r$ and integers $1 \le i_1 < i_2 < i_3 < \cdots < i_m \le q$ such that $K_{i_1}, K_{i_2}, \cdots, K_{i_m}$ are distinct. By definition of $\mathcal{J}^1$, the r-sets $K_{i_1}, K_{i_2}, \cdots, K_{i_m}$ are all x-innovative, whence the substrings $x(K_{i_1}), x(K_{i_2}), \cdots, x(K_{i_m})$ are distinct. Notice that

$$\sum_{k=1}^{m} |x(K_{i_k})| \le r[|J_1| + |J_2| + \cdots + |J_q|]. \tag{A.68}$$

Any r-set in $\mathcal{J}^1$ is x-redundant, but its father is not x-redundant. Therefore, if $J \in \mathcal{J}^1$ properly contains $J' \in \mathcal{J}^1$, then J must also properly contain $\mathrm{fa}\,(J')$, which contradicts the fact that any r-set contained in an x-redundant r-set is also x-redundant. We conclude that the r-sets in $\mathcal{J}^1$ are pairwise disjoint, whence the right side of (A.68) is at most rn. This gives us

$$\sum_{k=1}^{m} |x(K_{i_k})| \le rn$$

which implies that the sum of the first m terms of the sequence consisting of $|A|$ ones, followed by $|A|^2$ twos, $|A|^3$ threes, etc., can be no bigger than rn. Suppose

$$m \geq |A| + |A|^2 + |A|^3 + |A|^4 + |A|^5. \tag{A.69}$$

Then

$$\sum_{i=1}^{j} i|A|^i \leq rn \tag{A.70}$$

where j is the integer ≥ 5 satisfying

$$|A|+|A|^2+\cdots+|A|^j \leq m < |A|+|A|^2+\cdots+|A|^{j+1}. \tag{A.71}$$

Doing the sum on the left side of (A.70), we see that

$$\frac{|A| + |A|^{j+1}[j(|A|-1)-1]}{|A|-1} \leq rn$$

which implies

$$|A|^{j+1}[j-1] \leq rn. \tag{A.72}$$

Notice that the left side of (A.72) is ≥ 4. For $4 \leq u \leq v$, one has $u/\log u \leq v/\log v$, and so, letting u, v be the left and right sides of (A.72), respectively, we conclude that

$$\frac{|A|^{j+1}}{\left(\frac{j+1}{j-1}\right)\log|A| + \frac{\log(j-1)}{j-1}} \leq \frac{rn}{\log(rn)}. \tag{A.73}$$

The previous inequality implies that

$$\frac{|A|^{j+1}}{(3/2)\log|A| + (1/2)} \leq r\left(\frac{n}{\log n}\right) \tag{A.74}$$

because $\log(j-1)/(j-1) \leq 1/2$ for $j \geq 5$. Summing the right side of (A.71), we obtain

$$m < |A|\,\frac{|A|^{j+1}-1}{|A|-1}$$

which implies

$$|A|^{j+1} > m/2.$$

Applying this to (A.74), we get

$$m \leq (3\log|A|+1)r\left(\frac{n}{\log n}\right) \leq 4\log|A|\left(\frac{n}{\log n}\right).$$

We demonstrated this under that assumption that (A.69) holds. If (A.69) does not hold, then

$$m \leq 2|A|^5.$$

Therefore, whether (A.69) holds or not

$$q/r \leq m \leq 2|A|^5 + 4\log|A|r\left(\frac{n}{\log n}\right)$$

which completes our proof of (3.39).

Proof of Part a) of Lemma 3: Let x be an A-string of length at least r^I, and let n be the length of x. Let us call a family of r-sets $\mathcal{J}$ a $\mathcal{J}(r, I|x)$-tree if all of the following properties hold for $\mathcal{J}$:

- $\mathcal{J}_0(r, I|n) \subset \mathcal{J} \subset \mathcal{J}(r, I|x)$.
- If $J \in \mathcal{J}$ and $|J| > 1$, then either all of the children of J belong to $\mathcal{J}$ or none of them do.
- If $J \in \mathcal{J}$ and $J \notin \mathcal{J}_0(r, I|n)$, then $\mathrm{fa}(J) \in \mathcal{J}$.

It is easy to check that $\mathcal{J}_0(r, I|n)$ and $\mathcal{J}(r, I|x)$ are each $\mathcal{J}(r, I|x)$-trees. We establish part a) of Lemma 3 by mathematical induction. Our induction hypothesis is the statement that for any $\mathcal{J}(r, I|x)$-tree $\mathcal{J}$, the string x is the concatenation of the strings $x(L_1), x(L_2), \cdots, x(L_s)$, where $L_1, L_2, \cdots, L_s$ are the leaves of $\mathcal{J}$ (the members of $\mathcal{J}$ not properly containing any member of $\mathcal{J}$), ordered according to their left-to-right appearances as subsets of the real line. The induction hypothesis can be seen to be true for $\mathcal{J} = \mathcal{J}_0(r, I|n)$. Suppose $\mathcal{J}_1$ and $\mathcal{J}_2$ are two $\mathcal{J}(r, I|x)$-trees such that

- $\mathcal{J}_2$ is obtained by appending to $\mathcal{J}_1$ all of the children of a leaf of $\mathcal{J}_1$.

Then, we shall say that $\mathcal{J}_2$ is a *simple extension* of $\mathcal{J}_1$. It is easy to see that if the induction hypothesis holds for $\mathcal{J}$, then it must also hold for any simple extension of $\mathcal{J}$. Starting from $\mathcal{J}_0(r, I|n)$, one can perform finitely many simple extensions to arrive at any $\mathcal{J}(r, I|x)$-tree whatsoever. We conclude that the induction hypothesis must hold for every $\mathcal{J}(r, I|x)$-tree $\mathcal{J}$. Part a) of Lemma 3 now follows by taking $\mathcal{J} = \mathcal{J}(r, I|x)$ in the induction hypothesis.

Proof of Part b) of Lemma 3: Again, let x be an A-string of length at least r^I. Fix $0 \leq i < I$. As in Lemma 1, let $(K_1, \cdots, K_m)$ be the sequence of r-sets in $\mathcal{J}(r, I|x)$ of cardinality r^{I-i}, ordered according to their left-to-right appearance on the real line. Let $(K_{i_1}, K_{i_2}, \cdots, K_{i_s})$ be the subsequence of $(K_1, \cdots, K_m)$ whose entries are the leaves of $\mathcal{J}(r, I|x)$ of cardinality r^{I-i}. This is precisely the subsequence of $(K_1, \cdots, K_m)$ formed by the x-redundant entries of $(K_1, \cdots, K_m)$; that is, $j \in \{i_1, i_2 \cdots, i_s\}$ if and only if $x(K_j)$ coincides with $x(K_i)$ for some $i < j$. Since, by Lemma 1, $(x(K_1), \cdots, x(K_m))$ is the sequence $S_i(x)$, it follows by the definition of how $\tilde{S}_i(x)$ is obtained from $S_i(x)$ that $\tilde{S}_i(x)$ must be equal to $(x(K_{i_1}), \cdots, x(K_{i_s}))$, which gives us part b) of Lemma 3.

ACKNOWLEDGMENT

J. C. Kieffer would like to thank S. Yakowitz and T. Park at the University of Arizona and J. Massey at the Swiss Federal Institute of Technology (Zürich, Switzerland), during his stays at these institutions as part of his 1996–1997 sabbatical. The results reported in Table I were compiled by R. Stites at the University of Minnesota. The authors would all like to thank D. Neuhoff for bringing the N-gram text compression algorithm to their attention.

REFERENCES

[1] J. Bugajski and J. Russo, “Data compression with pipeline processors having separate memories,” U.S. Patent 5245337, Sept. 1993.

[2] T. Cover and J. Thomas, *Elements of Information Theory*. New York: Wiley, 1991.

[3] I. Csiszár and J. Körner, *Information Theory, Coding Theorems for Discrete Memoryless Systems*. Budapest, Hungary: Akadémiai Kaidó, 1981.

[4] P. Elias, “Universal codeword sets and representations of the integers,” *IEEE Trans. Inform. Theory*, vol. IT-21, pp. 194–203, Mar. 1975.

[5] C. Hobby and N. Ylvisaker, “Some structure theorems for stationary probability measures on finite state sequences,” *Ann. Math. Statist.*, vol. 35, pp. 550–556, 1964.

[6] J. Kieffer and E. Yang, “Grammar based codes: A new class of universal lossless source codes,” *IEEE Trans. Inform. Theory*, vol. 46, pp. 737–754, May 2000.

[7] ——, “Redundancy of MPM data compression system,” in *Proc. 1998 IEEE Int. Symp. Information Theory*, Cambridge, MA, p. 136.

[8] J. Kieffer, G. Nelson, and E. Yang, “Tutorial on the quadrisection method and related methods for lossless data compression,” Dept. Elec. Comput. Eng., Univ. Minn., Tech. Rep., 1996.

[9] J. Kieffer, T. Park, Y. Xu, and S. Yakowitz, “Progressive lossless image coding via self-referential partitions,” in *Proc. 1998 Intl. Conf. Image Processing*, vol. 1, Chicago, IL, pp. 498–502.

[10] G. Louchard and W. Szpankowski, “On the average redundancy rate of the Lempel–Ziv code,” *IEEE Trans. Inform. Theory*, vol. 43, pp. 2–8, Jan. 1997.

[11] A. Moffat, R. Neal, and I. Witten, “Arithmetic coding revisited,” in *Proc. 1995 Data Compression Conf.*, Snowbird, UT, pp. 202–211.

[12] G. Nelson, J. Kieffer, and P. Cosman, “An interesting hierarchical lossless data compression algorithm,” in *Proc. 1995 IEEE Information Theory Soc. Workshop*, Rydzyna, Poland.

[13] T. Park, “Computational considerations in quadrisection image compression,” Honors thesis, Dept. Syst. Indust. Eng., Univ. Ariz., 1997.

[14] E. Plotnik, M. Weinberger, and J. Ziv, “Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel–Ziv algorithm,” *IEEE Trans. Inform. Theory*, vol. 38, pp. 66–72, Jan. 1992.

[15] S. Savari, “Redundancy of the Lempel–Ziv incremental parsing rule,” *IEEE Trans. Inform. Theory*, vol. 43, pp. 9–21, Jan. 1997.

[16] C. Teng and D. Neuhoff, “An improved hierarchical lossless text compression algorithm,” *Proc. 1995 IEEE Data Compression Conf.*, pp. 292–301.

[17] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530–536, Sept. 1978.