Universal Lossless Data Compression With Side Information by Using a Conditional MPM Grammar Transform

En-hui Yang, Senior Member, IEEE, Alexei Kaltchenko, Student Member, IEEE, and John C. Kieffer, Fellow, IEEE

Abstract—A grammar transform is a transformation that converts any data sequence to be compressed into a grammar from which the original data sequence can be fully reconstructed. In a grammar-based code, a data sequence is first converted into a grammar by a grammar transform and then losslessly encoded. Among several recently proposed grammar transforms, the multilevel pattern matching (MPM) grammar transform is the most powerful. In this paper, the MPM grammar transform is first extended to the case of side information known to both the encoder and decoder, yielding a conditional MPM (CMPM) grammar transform. A new simple linear-time and space complexity algorithm is then proposed to implement the MPM and CMPM grammar transforms. Based on the CMPM grammar transform, a universal lossless data compression algorithm with side information is developed, which can achieve asymptotically the conditional entropy rate of any stationary, ergodic source pair. It is shown that the algorithm’s worst case redundancy/sample against the k-context conditional empirical entropy among all individual sequences of length n is upper-bounded by $c(1/\log n)$, where $c$ is a constant. The proposed algorithm with side information is the first in the coming family of conditional grammar-based codes, whose expected high efficiency is due to the efficiency of the corresponding unconditional codes.

Index Terms—Arithmetic coding, coding with side information, entropy, grammar-based source codes, multilevel pattern matching (MPM), redundancy, string matching, universal data compression.

I. INTRODUCTION

Consider the typical data compression diagram shown in Fig. 1 with side information known to both the encoder and decoder. The sequences $x^n$ and $y^n$, where $n$ is the length of the sequences, are regarded as a sequence to be compressed and side information, respectively. Since $y^n$ is assumed known to both the encoder and decoder, the encoder can use its knowledge about $y^n$ to conditionally encode $x^n$ into a binary codeword $B(x^n|y^n)$. Upon receiving $B(x^n|y^n)$, the decoder can fully recover the original sequence $x^n$ with the help of its knowledge about $y^n$.

From information theory, the advantage of using side information, if any, for data compression is obvious; one can considerably reduce the compression rate if the side information $y^n$ is highly correlated with the original sequence $x^n$.

In many applications, side information known to both the encoder and decoder is naturally present. For instance, in video compression, one can use previous frames as side information for the current frame; in lossless resolution, scalable progressive image coding (or multiresolution image coding), one may use low-resolution images as side information for high-resolution images [6]; and in bioinformatics, especially in DNA sequence analysis [4], [9], it is desirable to estimate information distance between two DNA sequences (in this case, either DNA sequence can be regarded as side information for the other sequence). The challenge is often how to utilize efficiently the available side information. This is the issue that will be addressed in this paper and its companions.

Previously, the side-information problem has been theoretically studied for memoryless source pairs [16] and for individual sequences [23], [24]. There has also been an attempt [15] to create a conditional Lempel–Ziv algorithm; nonetheless, neither a practical algorithm nor a complete compression performance analysis has been developed.

Within the design framework of grammar-based codes [7], [18], [20], [21], in this paper, we extend the Multilevel Pattern Matching (MPM) grammar transform [8] to the case of side information known to both the encoder and decoder, yielding a conditional MPM (CMPM) grammar transform. We then propose a new simple algorithm for implementing the MPM and CMPM grammar transforms. Our algorithm implementation has linear storage and time complexities for MPM grammars with any number of levels while the original MPM transform is only proven to have linear complexity when the number of levels does not exceed $\lfloor \log n \rfloor$. Based on the CMPM grammar transform, we develop a universal lossless data compression algorithm with side information, which will be called a CMPM code and can achieve asymptotically the conditional entropy rate of any stationary, ergodic source pair. It is also shown that the algorithm’s worst case redundancy/sample against the k-context conditional empirical entropy among all individual sequences of length $n$ is upper-bounded by $c(1/\log n)$, where $c$ is a constant.

Manuscript received June 6, 2000; revised March 12, 2001. This work was supported in part by the Natural Science and Engineering Research Council of Canada under Grant RGPIN203035-98, by the Communications and Information Technology Ontario, Canada, by the Premier’s Research Excellence Awards of Ontario, and by the National Sciences Foundation under Grants NCR-9627965 and CCC-9902081.

E.-h. Yang and A. Kaltchenko are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: ehyang@ece.uwaterloo.ca; akaltche@ece.uwaterloo.ca).

J. C. Kieffer is with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA (e-mail: kieffer@ece.umn.edu).

Communicated by M. Weinberger, Associate Editor for Source Coding. Published by the IEEE Computer Society Press for IEEE. (0018–9448/01$10.00 © 2001 IEEE
The MPM grammar transform and its corresponding MPM code are a special case of general grammar transforms and grammar-based codes. Another interesting grammar transform is the so-called Yang–Kieffer greedy grammar transform [18], [20], based on which several efficient universal lossless compression algorithms have been proposed. These algorithms now represent some of the best known lossless compression algorithms. In forthcoming papers, we will extend the Yang–Kieffer greedy grammar transform and its corresponding algorithms to the case of side information and provide a unified theoretic framework for general conditional grammar-based codes.

This paper is organized as follows. In Section II, we review the structure of the (unconditional) MPM code and its performance. In Section III, we present a simple linear complexity algorithm for implementing the MPM grammar transform. This algorithm is extended in Section IV to construct our CMPM grammar transform and CMPM code; in that section, we also analyze the performance of the CMPM code. Finally, in Section V, we present our simulation results.

II. REVIEW OF THE MPM CODE

As alluded to in the last section, the MPM grammar-based code presented in [8] is a special case of general grammar-based codes [7]. Following [20], a general grammar-based code has a structure shown in Fig. 2. Let $\mathcal{A}$ = \{a$_1$, a$_2$, ..., a$_n$\} be the input data sequence alphabet. For any positive integer n, $\mathcal{A}^n$ denotes the set of all sequences of length n from $\mathcal{A}$. A sequence from $\mathcal{A}$ is sometimes called an A-sequence. The original data sequence $x^n = x_1 x_2 \cdots x_n \in \mathcal{A}^n$ is first transformed into a context-free grammar $G_{x^n}$ from which $x^n$ can be fully recovered. The grammar $G_{x^n}$ also gives rise to a nonoverlapping, variable-length parsing of $x^n$. The sequence $x^n$ is then compressed by using a zero-order adaptive arithmetic code to compress either $G_{x^n}$ or the corresponding sequence of parsed phrases. To get an appropriate $G_{x^n}$, string matching is often used in some manner. If $G_{x^n}$ is obtained by performing multilevel pattern matching, then one gets the MPM grammar transform and the MPM code.

A. MPM($r, I$) Grammar Transform

Let us now briefly review the MPM grammar transform. Let $r$ and $I$ be two positive integers that control how multilevel pattern matching is performed; as a result, the corresponding grammar transform is called the MPM($r, I$) grammar transform. To get the transformed grammar $G_{x^n}$ or its equivalent form given by the MPM($r, I$) grammar transform, we perform multilevel pattern matching. First, we partition the string $x_1 x_2 \cdots x_n$ from left to right into nonoverlapping substrings of the lengths $n_i$, $i = I, I - 1, \ldots, 0$, where the lengths are obtained from the r-ary expansion of the integer n. We denote these substrings by $x^{(n)i}$. Thus, the concatenation of $x^{(n)i}$, $x^{(n)i-1}$, ..., $x^{(n)0}$ gives the original string $x_1 x_2 \cdots x_n$. Let $(h_{00} h_{01} \cdots h_{I0})_r$ be the r-ary expansion of n. Then

$$
\begin{align*}
n_I &\triangleq (h_{00} h_{01} \cdots h_{I0} 0 \cdots 0)_r, \\
n_{I-1} &\triangleq (h_{I-1} 0 \cdots 0)_r, \\
&\vdots \\
n_1 &\triangleq (h_1 0)_r, \\
n_0 &\triangleq (h_0)_r,
\end{align*}
$$

where, for each $j$, $h_j$ is an integer between 0 and $r - 1$ inclusively, and some $n_i$ may be zero. For example, for $n = 31$, $I = 3$, and $r = 2$, we have

$$
\begin{align*}
n_3 &= 24, \\
n_2 &= 4, \\
n_1 &= 2, \\
n_0 &= 1.
\end{align*}
$$

At each level $i = I, \ldots, 1$, processing\(^1\) takes place in which a sequence of subblocks of $x^n$ of length $r^{i-1}$ is identified and

\(^1\)For brevity, we will occasionally say: "level i is processed (created, updated, etc.)" meaning that appropriate sequences of blocks, labels, and/or tokens at level i are processed.
then represented by a sequence of labels. To perform the processing at the top level $I$, we partition the sequence $x^{(r_I)}$ into blocks of $A$-symbols of the length $r^I$. From left to right, we visit every block and label the first appearance of each distinct block $\alpha$ by the symbol “$s$.” If the same block $\alpha$ appears again, label it by an integer which points to this block’s first appearance. That is, all identical blocks $\alpha$, except the leftmost one, will be labeled by the same integer, which is just the number of distinct blocks up to the first appearance of the block $\alpha$ inclusively. This natural ordering will provide the unique correspondence between blocks denoted by “$s$” and those denoted by integers.

Suppose processing has taken place at levels $I, I-1, \ldots, i$ where $i > 1$. To perform the processing at the next level $i - 1$, do the following.

Step 1: Partition every $A$-block from the level $i$ into $r$ subblocks of length $r^{i-1}$.

Step 2: Partition the sequence $x^{(r_{i-1})}$ into blocks of length $r^{i-1}$. Form a sequence of blocks of length $r^{i-1}$ by listing in the indicated order all the subblocks of length $r^{i-1}$ constructed in Step 1 and all the blocks of length $r^{i-1}$ arising from the partition of $x^{(r_{i-1})}$. This produces the sequence of blocks of $A$-symbols at level $i - 1$.

Step 3: Go through the sequence of blocks at level $i - 1$ from left to right. Mark every first-appearing distinct block with the symbol “$s$.” If the same block appears again, replace it with an integer which points to this block’s first appearance using the natural ordering described earlier.

Repeat the above procedure up through the processing of the last level, level 0. Fig. 3 illustrates the above procedure for $I = 3$, $r = 2$, $n = 31$, and $x^n = a b a b b a b b b a b b b a b b b a b b b a a b b a b b a a a a b b a b a b$. 

Note that the transformed MPM $x^{r_I}$ grammar can be naturally represented by an $r$-ary tree.

Following [8], we denote,\textsuperscript{2} for every level $i$, the sequence of block labels consisting of integers and the symbol “$s$” by $T_i$. We call each block label a “token,” and we call $T_i$ the “token sequence” corresponding to the level $i$. The sequence $(T_I, T_{I-1}, \ldots, T_0)$, called the multilevel representation of $x^n$ in [8], represents the grammar $G_{2^n}$. For our example in Fig. 3, we have

$$x^{(r_3)} = abababababababababababababaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaabobaab
We get

\[
\text{Level 0: } (a, b, b, b, a, a, b, a, b)
\]

and the corresponding token sequence is

\[
T_0 = abbbabab.
\]  

Combining (II.1)–(II.4), we obtain the multilevel representation of \(x^n\)

\[
T_3 = ss2 \quad T_2 = ss2ss \quad T_1 = s1s12s1s3 \quad T_0 = abbbabab.
\]

### B. Arithmetic Encoder

In any grammar-based code, the original sequence \(x^n\) is compressed indirectly by compressing the transformed grammar \(G_{x^n}\). In the case of the MPM code, we separately encode each token sequence \(T_i\) in the multilevel representation \((T_1, T_2, \ldots, T_0)\) via the multilevel arithmetic coding algorithm proposed in [19]. Before encoding, since every token sequence begins with the symbol “s” (except the bottom one), we simply remove the first symbol “s” from each token sequence. For the levels \(i = 1, \ldots, I\), let

\[
\nu^{(i)} = \nu^{(i)}_1 \nu^{(i)}_2 \cdots \nu^{(i)}_k
\]

be the token sequence \(T_i\) with the first symbol “s” removed, where

\[
\nu^{(i)}_k \in \{s\} \cup \{1, 2, 3, \ldots, f_i^s\}
\]

\(l_i\) denotes the length \(|\nu^{(i)}_k|\), and \(f_i^s\) denotes the number of times the symbol “s” appears in \(T_i\). Let’s agree that \(\nu^{(0)} = \nu^{(0)}_1 \nu^{(0)}_2 \cdots \nu^{(0)}_k\). For our example in Fig. 3, we have

\[
\begin{align*}
\nu^{(3)} &= s2 \\
\nu^{(2)} &= s2ss \\
\nu^{(1)} &= 1s12s1s3 \\
\nu^{(0)} &= abbbabab.
\end{align*}
\]

The arithmetic encoder sequentially and independently encodes the sequences \(\nu^{(i)}, \nu^{(i-1)}, \ldots, \nu^{(0)}\), restarting at the beginning of each \(\nu^{(i)}\), and concatenates the binary codewords for each sequence in the indicated order into one single binary codeword \(B(G_{x^n})\). Each of the sequences \(\nu^{(i)}, \nu^{(i-1)}\) is encoded as follows. We associate each symbol

\[
\beta \in \{1, 2, 3, \ldots, f_i^s\} \cup \{s\}
\]

with a counter \(c(\beta)\). Initially, \(c(\beta)\) is set to 1 if \(\beta \in \{s, 1\}\) and 0 otherwise. The initial alphabet used by the arithmetic code is \(\{s, 1\}\). Encode each symbol \(\beta\) in the sequence \(\nu^{(i)}\) and update the related counters according to the following steps.

Step 1: Encode \(\beta\) by using the probability

\[
\frac{c(\beta)}{\sum_{\alpha \in A} c(\alpha)}
\]

where the summation \(\sum_{\alpha}\) is taken over \(\{s, 1\} \cup \{2, 3, \ldots, j + 1\}\), and \(j\) is the number of times that \(s\) occurs before the position of this \(\beta\). Note that the alphabet used at this point by the arithmetic code is \(\{s, 1\} \cup \{2, 3, \ldots, j + 1\}\).

Step 2: Increase the counter \(c(\beta)\) by 1.

Step 3: If \(\beta = s\), increase the counter \(c(j + 2)\) from 0 to 1, where \(j\) is defined in Step 1.

Repeat the above procedure until the entire sequence \(\nu^{(i)}\) is encoded.

To encode the bottom level sequence \(\nu^{(0)}\), we use the following two-steps adaptive arithmetic coding.

Step 1: Encode \(\beta\) by using the probability

\[
\frac{c(\beta)}{\sum_{\alpha \in A} c(\alpha)}
\]

Step 2: Increase the counter \(c(\beta)\) by 1.

Repeat the above procedure until the entire sequence \(\nu^{(0)}\) is encoded.

For our example given in Fig. 3, the product of the probabilities used in the arithmetic coding process of the entire grammar is

\[
\begin{bmatrix}
\nu^{(3)} \\
\nu^{(2)} \\
\nu^{(1)} \\
\nu^{(0)}
\end{bmatrix} = \begin{bmatrix}
\frac{11}{44} \\
\frac{11}{22} \\
\frac{11}{22} \\
\frac{11}{44}
\end{bmatrix} \cdot \begin{bmatrix}
\frac{1}{4} \\
\frac{1}{2} \\
\frac{1}{2} \\
\frac{1}{4}
\end{bmatrix} \cdot \begin{bmatrix}
\frac{1}{2} \\
\frac{1}{2} \\
\frac{1}{2} \\
\frac{1}{2}
\end{bmatrix}.
\]

**Remark 1:** When decoding the binary codeword \(B(G_{x^n})\), the arithmetic decoder must know the length of each sequence \(\nu^{(i)}\) before it starts processing the part of \(B(G_{x^n})\) corresponding to \(\nu^{(i)}\). To satisfy this requirement, the decoder sequentially computes all the lengths \(|\nu^{(i)}|\) = \(T_i - 1\) as follows. Assume that the length \(n\) of the original sequence \(x^n\) is known to the decoder. Then, the length of the top level token sequence is given by

\[
T_1 = \left\lfloor \frac{n}{4^3} \right\rfloor \equiv \left\lfloor \frac{n}{4^3} \right\rfloor.
\]

Once the sequence \(\nu^{(i)}\) is decoded, the length of the next sequence \(\nu^{(i-1)}\) is calculated according to the following formula:

\[
|\nu^{(i-1)}| + 1 = |T_{i-1}| = r f_i^s + \left\lfloor \frac{n_{i-1}}{r f_i^s} \right\rfloor \
\equiv r f_i^s + \left\lfloor \frac{n_{i-1}}{r f_i^s} \right\rfloor - r \left\lfloor \frac{n_{i-1}}{r f_i^s} \right\rfloor, \quad i = I, \ldots, 1.
\]

### C. Bounds on the Size of the Multilevel Grammar Representation

In this section, we make references to some results from [8], which will be used for complexity and compression performance analysis. Let us agree that, unless written explicitly, all logarithm bases throughout the paper are assumed to be 2. From
the description of the MPM code, there are simple relations between \( f_{i+1}^s \) and \( l_i \)

\[
\sum_{i=0}^{l-1} l_i - I(r-2) \leq r \sum_{i=1}^{l-1} f_i^s \leq \sum_{i=0}^{l-1} l_i + I.
\]  

(II.8)

Also, the following bound is proved in [8]:

\[
\sum_{i=0}^{l} l_i - I[I] \leq 2n^r - I + 2I + 2I[A]^r + 4r[A]^2
\]

\[
+ 8r^2 \log |A| \left( \frac{n}{\log n} \right). 
\]

(II.9)

Note that \( I \) cannot exceed \( \lfloor \log_2 n \rfloor \). To minimize the order on \( n \) of the expression given by the right-hand side of (II.9), the integer parameter \( I \) must be chosen to be at least of the order \( n \). For \( \log |A| \leq I \leq \lfloor \log_2 n \rfloor \), we obtain the following bound:

\[
\sum_{i=0}^{l} l_i - I[I] \leq (2n^r \log |A| + 2) \left( \frac{n}{\log n} \right)
\]

\[
+ 2r \log n + 2r[A]^r + 4r[A]^2.
\]

(II.10)

For sufficiently large \( n \), we simplify this bound as follows:

\[
\sum_{i=0}^{l} l_i - I[I] < C_2 \left( \frac{n}{\log n} \right)
\]

(II.11)

where \( C_2 = 10r^2 \log |A| \).

\[\text{D. Computational Complexity of the Arithmetic Coding}\]

Using the results from the previous section, we are ready to upper-bound the computational complexity of the arithmetic encoding and decoding part of the MPM code. Recall that the time, required for the “classical” adaptive arithmetic coding algorithm [17] to encode a sequence of length \( n \) drawn from an alphabet \( A \) is \( O(n|A|) \). Consequently, in view of (II.8) and (II.11), the time, required for that algorithm to encode the token sequences \( \psi^{(i)} \), \( \psi^{(I-1)} \), \ldots, \( \psi^{(0)} \), will be

\[
O \left( \sum_{i=1}^{I} l_i f_i^s + l_0 |A| \right) = O \left( \frac{n^2}{\log^2 n} \right).
\]

Another problem associated with the classical adaptive arithmetic coding algorithm is that it cannot handle a dynamic alphabet which grows without bound. To get around these two problems, we instead apply the multilevel arithmetic coding algorithm, proposed in [19], to encode the token sequences. One of the advantages of the multilevel arithmetic coding algorithm over the classical adaptive arithmetic coding algorithm is that the time required for the multilevel arithmetic coding algorithm to encode a sequence of length \( n \) drawn from an alphabet \( A \)

is \( O(n \log |A|) \), and that the storage required is \( O(|A|) \). In our case, for each level \( i = I, \ldots, 0 \) we have

\[
m = l_i
\]

\[
|A| = \begin{cases} f_i^s + 1, & i \neq 0 \\ f_i^s + l_0 \log |A|, & i = 0. \end{cases}
\]

Consequently, the arithmetic encoding and decoding part of the MPM code has the \( O(\max_{1 \leq i \leq I} f_i^s) \)-space and the \( O(\sum_{i=1}^{I} l_i \log f_i^s + l_0 \log |A|) \)-time complexities. It is easy to see that

\[
\sum_{i=1}^{I} l_i \log f_i^s + l_0 \log |A| \leq \log \left( \sum_{i=1}^{I} f_i^s + |A| \right) \sum_{i=1}^{I} l_i,
\]

(II.12)

In view of the relations (II.8), (II.11), and (II.12), we obtain

\[
\sum_{i=1}^{I} l_i \log f_i^s + l_0 \log |A| < 10r \log |A| \cdot \frac{n}{\log n}
\]

\[
\sum_{i=1}^{I} l_i \log f_i^s + l_0 \log |A| < 10 \log |A| \cdot n.
\]

Remark 2: It should be pointed out that when we measure the time and space complexity of the arithmetic coding part of the MPM code, i.e., the multilevel arithmetic coding, a standard random-access machine (RAM) model with uniform cost criterion [1], [2] is assumed. This assumption is usually made in almost all papers addressing the implementation of arithmetic coding such as [11], [17], and [19], and is certainly valid if only finite precision implementation is concerned. To the best of our knowledge, the only exception so far is the papers [13], [14], where the logarithmic cost criterion, i.e., bitwise computation, is used, and where the tradeoff between redundancy and bitwise complexity is addressed.

\[\text{E. Compression Rate Related to the Grammar Entropy}\]

Let \( \psi^{(i)} \) be the sequence obtained from \( \psi^{(0)} \) by removing all symbols “s.” The unnormalized empirical entropy of \( \psi^{(i)} \) is defined as

\[
H \left( \psi^{(i)} \right) = \sum_{\alpha \in \{1, \ldots, f_i^s\}} n(\alpha) \log \frac{|\psi^{(i)}|}{n(\alpha)}
\]

where \( n(\alpha) \) denotes the number of occurrences of \( \alpha \) in \( \psi^{(i)} \), and \( |\psi^{(i)}| \) denotes the length of \( \psi^{(i)} \). A similar definition also applies to any other sequences.

The probability used to encode the symbol \( \psi^{(i)}_k \) in the sequence \( \psi^{(i)}_1 \cdots \psi^{(i)}_k \) is

\[
\frac{C \left( \psi^{(i)}_k \right)}{k + c^{(i)}(s) + 1}
\]

for \( 1 \leq i \leq I \) and

\[
\frac{C \left( \psi^{(0)}_k \right)}{k + |A| - 1}
\]
for \( i = 0 \), where \( C(v^{(0)}_k) \) is the number of occurrences of \( v^{(0)}_k \) in the prefix \( v^{(0)}_1 \cdots v^{(0)}_{k-1} \), and \( e^{(i)}(s) \) is the number of occurrences of \( \text{“} s \text{”} \) in \( v^{(i)}_1 \cdots v^{(i)}_{k-1} \). Assume that exact arithmetic is used in the arithmetic coding. Then the number of bits needed to encode \( v^{(i)}_k \) is

\[
\log \left( \frac{k + e^{(i)}(s) + 1}{C(v^{(i)}_k)} \right)
\]

for \( 1 \leq i \leq I \) and

\[
\log \left( \frac{k + |A|}{C(v^{(i)}_k)} \right)
\]

for \( i = 0 \). Since \( e^{(i)}(s) \leq f^{(i)}_s - 1 \), we have

\[
\log \left( \frac{k + e^{(i)}(s) + 1}{C(v^{(i)}_k)} \right) \leq \log \left( \frac{k + f^{(i)}_s}{C(v^{(i)}_k)} \right)
\]

for \( 1 \leq i \leq I \). Then, the number of bits \( B_i \) needed to encode the entire level \( i, 1 \leq i \leq I \), is

\[
B_i \leq \log \left( \frac{\prod_{k=1}^i k + f^{(i)}_s}{\prod_{\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}} \left( e^{(i)}_\beta ! \right)} \right) + \log \left( \frac{\prod_{\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}} \left( e^{(i)}_\beta ! \right)}{\prod_{\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}} \left( e^{(i)}_\beta ! \right)} \right)
\]

\[
\leq \log \left( \frac{\prod_{k=1}^i k + f^{(i)}_s}{\prod_{\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}} \left( e^{(i)}_\beta ! \right)} \right) + \log \left( \frac{\prod_{\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}} \left( e^{(i)}_\beta ! \right)}{\prod_{\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}} \left( e^{(i)}_\beta ! \right)} \right)
\]

\[
\leq H\left( v^{(i)} \right) + l_i + f^{(i)}_s
\]

\[
\leq 2l_i + f^{(i)}_s + H\left( v^{(i)} \right).
\]

In the above, \( e^{(i)}_\beta \) denotes, for each

\[
\beta \in \{s\} \cup \{A\} \cup \{1, 2, \ldots, f^{(i)}_s\}
\]

the number of occurrences of \( \beta \) in \( v^{(i)} \). The inequality1 is due to the inequality on the size of a type class; the equality 2 follows from the identity

\[
H(X, Y) = H(X) + H(Y|X)
\]

for any random variables \( X \) and \( Y \); and the final inequality is attributable to the fact that \( H\left( \frac{e^{(i)}_\beta}{l_i}, \frac{l_i - e^{(i)}_\beta}{l_i} \right) \leq 1 \).

Analogously, the number of bits needed to encode the bottom level \( i = 0 \) is upper-bounded by

\[
B_0 \leq H\left( v^{(0)} \right) + \log \left( \frac{l_0 + |A| - 1}{|A| - 1} \right) \leq l_0 + |A| + H\left( v^{(0)} \right).
\]

To simplify our notation, let us agree that \( v^{(0)} \), \( v^{(0)} \), \( v^{(0)} \), \( v^{(0)} \), \( v^{(0)} \), \( v^{(0)} \) are equal to \( v^{(0)} \). Then, from (II.13) and (II.14)

\[
\sum_{i=0}^I B_i \leq \sum_{i=0}^I H\left( v^{(i)} \right) + 2 \cdot \sum_{i=0}^I l_i + \sum_{i=1}^I f^{(i)}_s - l_0 - |A|.
\]

Definition II.1: The entropy of an MPM grammar is defined as

\[
H_{M}(x^n) \triangleq \sum_{i=0}^I H\left( v^{(i)} \right).
\]

In terms of Definition II.1, therefore, we have proved that

\[
\sum_{i=0}^I B_i \leq H_{M}(x^n) + 2 \cdot \sum_{i=0}^I l_i + \sum_{i=1}^I f^{(i)}_s - l_0 - |A|.
\]

To go further, we have to upper-bound the two sums in the right-hand side of (II.15) in terms of the length \( n \) of the original sequence \( x^n \).

F. Redundancy of the MPM Code

Following the approach from [8], [7], [20], we compare the compression performance of the algorithm with that of the best arithmetic coding algorithm with \( k \) contexts operating letter by letter (not phrase by phrase).

Let \( S_k \) be a finite set of \( k \) elements; each element \( s \in S_k \) is regarded as an abstract context. Let \( p : S_k \times (A \times S_k) \rightarrow [0, 1] \) be a transitional probability function from \( S_k \) to \((A \times S_k)\), i.e., \( p \) satisfies

\[
\sum_{s \in S_k, \beta \in A} p(s, \beta | s') = 1, \quad s' \in S_k.
\]

Random transitions between contexts are allowed. For any sequence \( x^n = x_1 x_2 \cdots x_n \) from the finite source alphabet \( A \), the compression rate in bits per letter resulting for the arithmetic coding algorithm with transition probability \( p \) is given by

\[
\frac{1}{n} \log \left( \sum_{s_1, s_2, \ldots, s_n \in S_k} \prod_{i=1}^n p(s_i, x_i | s_{i-1}) \right),
\]

where \( s_0 \) is the initial context. The \( k \)-context empirical entropy of \( x^n \), \( H_k(x^n) \), defined in [20] and given below, gives the smallest compression rate among all arithmetic coding algorithms with \( k \) contexts operating letter by letter

\[
H_k(x^n) \triangleq -\frac{1}{n} \log \left( \max_{s_1, s_2, \ldots, s_n \in S_k} \sum_{s_1 \in S_k} \prod_{i=1}^n p(s_i, x_i | s_{i-1}) \right).
\]

(II.16)
We want to compare the compression rate of the MPM code with $H^k(x^n)$ for any $k$ and any $x^n = x_1x_2\ldots x_n$.

Fix $x^n = x_1x_2\ldots x_n$. Let $p$ be a transitional probability function which maximizes (II.16); $p$ of course, depends on $x^n$. For any $A$-string $z^q = z_1\ldots z_q \in A^q$ and any $q \geq 1$, define

$$\tau(z^q) = \max_{s_i \in \Sigma_k} \left( \sum_{i=1}^{q} p(s_i, z_i|s_{i-1}) \right).$$

We note that for every $A$-string $z^q \in A^q$ and every parsing of $z^q$ into phrases $v_1, v_2, \ldots, v_j$, the following log-subadditivity relation holds:

$$\tau(z^q) \leq \tau(v_1) \cdot \tau(v_2) \cdots \tau(v_j).$$

It follows from the definition of the function $\tau(\cdot)$ that

$$1 \leq \sum_{z^q \in A^q} \tau(z^q) \leq k.$$ 

For each $q \geq 1$, we normalize $\tau(\cdot)$ over $A^q$ so that

$$p^k(z^q) = \frac{Q_k}{k} \cdot \tau(z^q), \quad z^q \in A^q$$

is a probability distribution over $A^q$ satisfying

$$\sum_{z^q \in A^q} p^k(z^q) = 1.$$ 

From (II.18)–(II.20), it is easy to see that the constant $Q_k$ depends on $q$ and satisfies $1 \leq Q_k \leq k$.

Recall [7], [20] that a grammar transform induces a so-called partition sequence $\psi^m = \psi_1\psi_2\ldots\psi_m$. The following recursive procedure takes the root node of the corresponding tree representation of the transformed grammar (see the example depicted in Fig. 3) as an input parameter and creates the partition sequence.

Postorder(Node)
Begin
if the Node has no children, then append this Node’s token to the output list
else
Postorder(Node’s 1st child);
Postorder(Node’s 2d child);
............................;
Postorder(Node’s last child);
End

In our example in Fig. 3, $\psi^m = ab_1b_2b_1b_2\ldots a_1a_2a_1b_3\psi_b$, where a subscript of each integer variable indicates the grammar level $i$ for this variable.

Since each token represents some substring of $x^n$, the partition sequence $\psi^m$ induces a parsing of $x^n$ if each symbol in $\psi^m$ is replaced with the corresponding substring of $x^n$. The concatenation of the phrases in this parsing is $x^n$. Clearly, the partition sequence will not contain the symbol “$\psi$.” Consequently, the partition sequence $\psi^m$ is a permutation of the sequence obtained by concatenating $\psi_i^{(1)}, \psi_i^{(1-1)}, \ldots, \psi_i^{(0)}$. Let us agree, that when used as an argument in the functions $\tau(\cdot), p^k(\cdot), p^k(\cdot)$, the record $\psi_j^{(i)}$ denotes the $A$-sequence represented by the token $\psi_j^{(i)}$. Then, due to the log-subadditivity relation

$$H^k(x^n) = -\frac{1}{n} \sum_{i=0}^{l} \log \tau(\psi_j^{(i)}) \geq -\frac{1}{n} \sum_{i=0}^{l} \sum_{j=1}^{|\psi_j^{(i)}|} \log \tau(\psi_j^{(i)}),$$

and

$$-\sum_{j=1}^{l} \log \tau(\psi_j^{(i)})$$

In view of the information inequality [3, Theorem 2.6.3, p. 26]

$$H(\psi_j^{(i)}) \equiv \min_{\hat{p}} \sum_{j=1}^{l} -\log \hat{p}(\psi_j^{(i)})$$

where the minimum is over all probability distributions $\hat{p}$ on $A^n$. Combining (II.21)–(II.23) together, we get

$$H^k(x^n) \geq \frac{1}{n} \sum_{i=0}^{l} \left( H(\psi_j^{(i)}) - |\psi_j^{(i)}| \log k \right)$$

$$= \frac{1}{n} H(\psi) - \frac{1}{n} \cdot \log k \cdot \sum_{i=0}^{l} |\psi_j^{(i)}|.\quad (II.24)$$

Using the inequalities (II.15), (II.7), and (II.8), we obtain

$$\frac{1}{n} \sum_{i=0}^{l} B_i = H^k(x^n)$$

\begin{align*}
\leq & \frac{1}{n} \left[ 2 \cdot \sum_{i=1}^{l} l_i + \sum_{i=1}^{l} f_i^k + \log k \cdot \sum_{i=0}^{l} |\psi_j^{(i)}| + |A| - l_0 \right] \\
< & \frac{1}{n} \left[ 2 + \log k \cdot k \right] \sum_{i=0}^{l} l_i + l_i + |A|.\quad (II.25)
\end{align*}
By substituting the bound for $\sum_{i=0}^{I} I_i$ from (II.11) into (II.25) and recalling that $I \leq \left\lfloor \log_2 n \right\rfloor$, we get

$$\frac{1}{n} \sum_{i=0}^{I} B_i - H^k(x^n) < \frac{1}{n} \left[ 10r(2r + r \log k + 1) \log |A| + \frac{n}{\log n} + \frac{\log_2 n}{r} + |A| \right].$$

(II.26)

Finally, we find the following upper bound on the worst case redundancy (per sample) of the algorithm against the k-context empirical entropy defined in [20]:

$$\max_{x^n \in A^n} \left[ \frac{1}{n} \sum_{i=0}^{I} B_i - H^k(x^n) \right] < C \frac{1}{\log n}$$

for large $n$, where $C = 10r(2r + r \log k + 2) \log |A|$.

Remark 3: The notion of the worst case redundancy of a compression algorithm is rather a strong one. As shown in [7], [8], the worst case redundancy bound also implies the same-order bound for several different notions of redundancy. In particular, an MPM code has the $O(1/\log n)$ maximal individual redundancy relative to the class of all alphabet $A$ finite-state information sources. For comparison, although the same $O(1/\log n)$ bound has been obtained for the average [10] redundancy of the Lempel–Ziv LZ 78 code [25], its maximal individual redundancy is only known [12] to be $O(\log n/\log \log n)$. The average redundancy of the MPM code, which is obviously no worse than $O(1/\log n)$, still remains the open problem.

III. A NEW ALGORITHM FOR IMPLEMENTING THE MPM$(r, I)$ GRAMMAR TRANSFORM

In Section II-A, we have described how to get the transformed grammar $G_{en}$ given by the MPM grammar transform from top to bottom. In this section, we present a linear-time algorithm which constructs the transformed grammar $G_{en}$ from bottom to top. We name this new algorithm a “Bottom-to-Top algorithm” in order to distinguish it from the original one, described in [8]. A nice feature of the new algorithm is that no string comparison is made explicitly. In the Bottom-to-Top algorithm, every level is processed in two stages: Creating stage and Updating stage. When the $i$th level is being updated, the $(i+1)$th level is being created. That is, this is a “sliding-window” algorithm that processes two adjacent levels at a time while moving up.

Let the token sequence $T_0$ at the level $i = 0$ be the original string $x_1x_2\cdots x_n$. So we say that the token sequence $T_0$ has been just created, but not updated yet. Suppose, more generally, the token sequence $T_i$ has been created, but not updated yet. The token sequence $T_{i+1}$ is created and the token sequence $T_i$ is updated as follows.

Creating $T_{i+1}$: From left to right, we partition the sequence $T_i$ into blocks of length $r$, so we will have $[|T_i|/r]$ such blocks. Each block will be replaced with a token from an abstract token alphabet, so that distinct (when treated as supersymbols) blocks are assigned distinct tokens. In practice, we use the set of natural numbers $\{1, 2, 3, \ldots \}$ as the token alphabet. All newly created tokens will form the token sequence $T_{i+1}$, corresponding to the next, $i+1$, level. For later use, we note that in the corresponding tree representation of the grammar, the node corresponding to a token $\alpha$ in the sequence $T_{i+1}$ will be a parent of the nodes corresponding to those $T_i$-sequence tokens that form the block replaced by $\alpha$.

Updating $T_i$: Traverse the sequence $T_i$ from left to right. Those tokens in the sequence $T_i$, whose parental tokens in the sequence $T_{i+1}$ have already appeared, are deleted (in reality they will not be deleted, but replaced with the integer “0”). While traversing the sequence $T_i$, we rename the token alphabet according to the natural ordering, defined in Section II. Finally, every first (the most left) appearance of a distinct token is replaced with the symbol “0.”

Note 1: When having created the level $i$, the algorithm has actually built the MPM$(r, i)$ grammar. Therefore, we say that to get the MPM$(r, I)$ grammar for any given $x^n$ and $r$, the Bottom-to-Top algorithm sequentially creates the grammars MPM$(r, 1)$, MPM$(r, 2)$, ..., MPM$(r, I - 1)$, and for any natural numbers $i$ and $k$, the grammar MPM$(r, i + k)$ is the extension of the grammar MPM$(r, i)$ in the sense that they have identical token sequences $T_0$, ..., $T_{i-1}$.

Remark 4: The described implementation of MPM$(r, I)$ Grammar Transform is “off-line.” That is, the entire sequence has to be observed before it has been processed. However, MPM$(r, I)$ Grammar Transform can also be implemented [19] sequentially, with linear computational complexity. Based on the sequentially implemented MPM$(r, I)$ Grammar Transform, a sequential MPM code will have, similarly to the 1978 Lempel–Ziv code, a $O(\log n)$ delay between the encoder input and the decoder output.

A. Actual Implementation of the Bottom-to-Top Algorithm

In actual implementation, every token sequence $T_i$ will be represented by an array $T(i)[1 \cdots [n/r^2]]$. To avoid explicit string comparison, both the creation process and updating process are helped with some lists $L(i)$ that are defined for every distinct token symbol $\alpha$ in the token sequence $T_i$ as follows:

$L(i) = \{\alpha\} \cup \{j; T(i)[j] = \alpha\}$.

All the lists $L(i)$ will be constructed so that every token symbol $\alpha$ will belong to the set of integers $\{1, \ldots, f_i \}$. Therefore, we simply write

$L(i) = \{m\} \cup \{j; T(i)[j] = m\}$, \hspace{0.5cm} \text{where} \hspace{0.5cm} m = 1, \ldots, f_i$.

Recall that in Section II-B we denoted the number of all distinct symbols in the token sequence $T_i$ by $f_i$. According to its definition, the list $L(i)$ will consist of the integer $m$ at the first
position and the indexes of all those array $T^{(i)}[1 \cdots \lceil n/r^i \rceil]$'s elements which are equal to $m$. Let $S^{(i)}$ be the set

$$S^{(i)} = \{ L_1^{(i)}, L_2^{(i)}, \ldots, L_{|A|}^{(i)} \}.$$ 

Clearly, the set $S^{(i)}$ uniquely specifies the array $T^{(i)}[\cdots]$. Consequently, if we can construct $S^{(i)}$, then we will be able to obtain the array $T^{(i)}[\cdots]$. 

**Bottom level creation ($i = 0$):** Assume there is a mapping $\alpha_j \rightarrow j$, $j = 1 \cdots |A|$, of the alphabet $A$ to the set of integers $\{1, 2, \ldots, |A|\}$. Denote this mapping by int$(\alpha_j)$. The algorithm initializes the array $T^{(0)}[1 \cdots n]$ and fills it with integers as follows:

$$T^{(0)}[1 \cdots n] := \{ \text{int}(x_1), \ldots, \text{int}(x_n) \}.$$ 

Originally, the lists in the set $S^{(0)}$ are initialized as follows:

$$L_1^{(0)} = \{ 1 \}$$

$$(i = 1 \cdots |A|, i \neq 1).$$

The following piece of pseudocode appends the indexes of the elements of the array $T^{(0)}[\cdots]$ to the lists $L_1^{(0)}, L_2^{(0)}, \ldots, L_{|A|}^{(0)}$:

```
Begin
Create the array Table[1 \cdots |A|] of pointers to the lists $L_1^{(0)}, \ldots, L_{|A|}^{(0)}$;
For every $|A|$, $j = 1 \cdots n$;
append index $j$ to the list pointed by the pointer Table[T^{(0)}[j]];
End.
```

Having created $T^{(0)}[1 \cdots n]$ and $S^{(0)}$, we say that the level 0 has been created, but not yet updated. 

**Creating the level $i + 1$ and updating the level $i$ ($i = 0, \ldots, i-1$):** Now suppose that the levels $0, \ldots, i-1$ have been created and updated and that the level $i$ has been created, but not yet updated. That is, the set $S^{(i)} = \{ L_1^{(i)}, L_2^{(i)}, \ldots, L_{|A|}^{(i)} \}$ and the array $T^{(i)}[1 \cdots \lceil n/r^i \rceil]$ both have been just created. To simplify our discussion, let us first look at the case of $r = 2$. See remarks below for the general case. Then, the pseudocode shown below creates the level $i + 1$ and updates the $i$th level for the case of $r = 2$, for which the corresponding MPM grammar transform is called a bisection grammar transform. As the set $S^{(i+1)}$ is being created, its size is growing from zero to $f_i$ and the current size is being stored in the variable $s(i + 1)$.

```
Begin
1 Set $s(i + 1) := 0$;
2 Initialize the array $T^{(i+1)}[1 \cdots \lceil n/2^{i+1} \rceil]$ of integers;
3 initialize the array $INDICATOR[1 \cdots f_i]$ of pointers to Null;
4 For $m = 1$ to $f_i$ do
skip the first element of the list $L_m^{(i)}$;
5 traverse it from left to right and do
6 For every odd index $j \leq \lceil n/2 \rceil$ in the list $L_m^{(i)}$
7 if $INDICATOR[T^{(i)}[j + 1]] = Nul$, then
8 $s(i + 1) := s(i + 1) + 1$; create the new
9 list $L_{i+1}^{(i+1)} = \{ s(i + 1), (j + 1)/2 \}$
10 and append it to the set $S^{(i+1)}$;
11 make the pointer $INDICATOR[T^{(i)}[j + 1]]$
point to this list $L_{i+1}^{(i+1)}$;
12 $T^{(i+1)}[(j + 1)/2] := s(i + 1)$ (later this element will be replaced by the symbol “*”);
else
13 append the index $(j + 1)/2$ to the list pointed by the pointer
$INDICATOR[T^{(i)}[j + 1]]$ and
14 replace the array’s element $T^{(i+1)}[(j + 1)/2]$ by the first member of this list;
15 $T^{(i)}[j] := 0$ and $T^{(i)}[j+1] := 0$ (i.e., delete elements $T^{(i)}[j]$ and $T^{(i)}[j+1]$);
skip the first element of the list $L_m^{(i)}$;
16 traverse it from left to right and do
17 For every odd index $j \leq \lceil n/2 \rceil$ in the list $L_m^{(i)}$,
18 $INDICATOR[T^{(i)}[j + 1]] := Null$;
19 Destroy the array $INDICATOR[1 \cdots f_i]$ and release the memory allocated for it.
20 Destroy the set $S^{(i)}$ and all the lists $L_1^{(i)}, \ldots, L_{|A|}^{(i)}$, and release the corresponding allocated memory.
21 Traverse the array $T^{(i)}[\cdot \cdot]$, skip all zero elements, rename the token alphabet according to the natural ordering, and replace the first appearance of each distinct token by the symbol “*”. This modified array $T^{(i)}[\cdot \cdot]$ is the algorithm output at the level $i$.
End.
```

Repeat the above procedure for $i = 0, \ldots, I-1$. Since the last value of $i$, passed to this procedure, is $I-1$, the level $I$ will not be updated, but just created and renamed. 

**Remark 5:** The renaming of the token symbols at level $i$ is being done while this level is being traversed from left to right.
Consequently, the time required to rename the tokens at level \(i\) is linear on the length \(|T_i|\) of level \(i\).

**Remark 6:** Just to avoid confusion, we have not mentioned earlier that the set \(S^{(i)}\) itself can be implemented as the array of pointers.

**Remark 7:** We call the reader’s attention to the fact that although every block of tokens can be associated with a supersymbol, we do not, however, explicitly compare token strings when determining identical supersymbols. Such explicit string comparison would significantly increase the computational complexity. Instead of comparing blocks of tokens as strings, we sequentially compare single tokens forming those blocks. Moreover, in our actual linked-list implementation there is no explicit integer multiplication, division, and summation except incrementing by 1, and integer bitwise shift by 1 (division or multiplication by two).

**Remark 8:** The pseudocode for \(\mathcal{C}\) can be easily modified to handle an arbitrary \(r\). We just need to convert the \(r\)-ary tree into binary \((r = 2)\) tree by inserting some intermediate levels as shown in Fig. 4. One may see that the number of added intermediate binary nodes is no more than the number of the nodes (tokens) at the level \(i\). This property will be used later to establish the linear complexity of the Bottom-to-Top algorithm for an arbitrary \(r\).

**Example 1:** Let the string \(x^n\) be the same as that shown in Fig. 3. The Bottom-to-Top algorithm sequentially builds MPM(2, 1), MPM(2, 2), and MPM(2, 3) grammars, where the final MPM(2, 3) grammar is the same as the grammar shown in Fig. 3. The token sequences representing the grammars are denoted by \(T_i\). For the illustrative purpose, each first-appearing distinct token in \(T_i\) is not replaced by the symbol \(s\) but just underlined. The algorithm begins with the MPM(2, 0) grammar shown in Fig. 5, which consists of only one token sequence \(T_0 = x^n\). First, the algorithm produces the MPM(2, 1) grammar by creating the sequence \(T_1\) and updating the sequence \(T_0\). Below, we illustrate the process of creating the level 1 and updating the level 0 in details; all other levels are processed similarly. To create \(T_1\) and update \(T_0\), the algorithm does three passes. Let us think of the sequences \(T_0\) and \(T_1\) as arrays \(T_0[1 \cdots n]\) and \(T_1[1 \cdots \lfloor n/2 \rfloor]\), respectively. Recall that for each even number \(j\) between 1 and \(n\), token \(T_1[j/2]\) is called a “parent” of tokens \(T_0[j]\) and \(T_0[j - 1]\). At the beginning, the counter \(s(1)\) is set to zero and the elements of \(T_1\) are not yet defined.

**Pass 1:** From left to right, the algorithm visits those even elements of \(T_0\) (accentuated by check \(\checkmark\) in Fig. 6), whose immediate left neighbor is the symbol \(a\). If a token \(\alpha\) being visited has already appeared earlier, the parent of this token is set equal to the parent of that token which is the first appearance of \(\alpha\). Otherwise, the counter \(s(1)\) is incremented by 1 and the parent of the token being visited is set to \(s(1)\) and underlined. The resulted grammar is shown in Fig. 6.

**Pass 2:** From left to right, the algorithm visits those even elements of \(T_0\) (accentuated by check \(\checkmark\) in Fig. 7) whose immediate left neighbor is the symbol \(b\). If a token \(\alpha\) being visited has already appeared earlier, the parent of this token is set equal to the parent of that token which is the first appearance of \(\alpha\). Otherwise, the counter \(s(1)\) is incremented by 1 and the parent of the token being visited is set to \(s(1)\) and underlined. The resulted grammar is shown in Fig. 7.

**Pass 3:** From left to right, the algorithm traverses \(T_1\) and re-names tokens to preserve the natural ordering. For this particular sequence \(T_1\), the algorithm switches tokens 2 and 3. At the same time, the algorithm sets to zero (i.e., deletes) those tokens at the level 0 whose parents are not underlined. The resulted grammar is shown in Fig. 8.

In a similar manner, by creating the sequence \(\hat{T}_2\) and updating the sequence \(\hat{T}_1\), the algorithm produces the MPM(2, 2) grammar shown in Fig. 9. Analogously, the algorithm produces the MPM(2, 3) grammar, shown in Fig. 10, by creating the sequence \(\hat{T}_3\) and updating the sequence \(\hat{T}_2\). By replacing the un-
derlined symbols in $\hat{T}_1, \ldots, \hat{T}_3$ with the symbol $s$, we get the resulting output grammar

$$
T_3 = ss2 \\
T_2 = ss2ss \\
T_1 = s1s12s1s3 \\
T_0 = abbbbaabab.
$$

Before encoding the grammar sequences (III.1), the algorithm removes the leftmost symbol $s$ in $\hat{T}_1, \ldots, \hat{T}_3$. The resulting sequences $\psi(3), \ldots, \psi(0)$

$$
\psi(3) = s2 \\
\psi(2) = s2ss \\
\psi(1) = 1s1s12s1s3 \\
\psi(0) = \hat{T}_0 = abbbbaabab.
$$

are then encoded by the arithmetic encoder.

B. The Complexity of the Bottom-to-Top Algorithm

Following [2], [1], unless specified otherwise, a RAM model with uniform cost criterion, on which most practical computers are based, is assumed when we measure the time and space complexity of our algorithms. This assumption implies that each basic operation in our pseudocode, such as node creation, deletion, and insertion as well as node data assigning and reading, takes a constant amount of time. This viewpoint reflects [2] how the pseudocode would be implemented on most actual computers. It also should be noted that, as we discussed in Section III-A, in the Bottom-to-Top algorithm, we use only the following explicit arithmetic operations on integers: assignment, comparison, incrementing by 1, and integer bitwise shift by 1 (division or multiplication by two).

To estimate the time complexity of the Bottom-to-Top algorithm, we introduce the following notation:

- $c_L$ is the execution time of node creation, deletion, insertion, and visiting (when traversing a list);
- $c_A$ is the time required to read or write an array element;
- $c_{cmp}$ is the time required to compare two integers;
- $c_{inc}$ is the time required to increment an integer by 1;
- $c_{shift}$ is the execution time of integer bitwise shift by 1.

We consider two cases:

1) Case $r = 2$ (Bisection): The execution time of the bottom level creation is the sum of the amounts of time required to create the following structures:

- $T(0)[1 \cdots n] \rightarrow (c_A + c_{inc})n$
- $T[1 \cdots |A|] \rightarrow (c_A + c_{inc})|A|$
- $L_{1}^{(0)}, \ldots, L_{|A|}^{(0)} \rightarrow c_L(n + |A|) + (2c_A + c_{inc})n$
- $S^{(0)} \rightarrow c_L |A|$

Thus, the time required to create the bottom level is

$$(2c_L + c_A + c_{inc})|A| + (c_L + 3c_A + 2c_{inc})n$$

To calculate the time required to create level $i + 1$ and update level $i$, we sum the running times for the corresponding pseudocode lines which are listed in Table I. We get

$$
3c_A + (3c_A + 2c_L + c_{inc})f_r^* \\
+ (2c_{shift} + 5c_{cmp} + 4c_A + 2c_L + 2c_{inc} \\
+ \max\{c_{shift} + c_{inc} + 4c_L + 3c_A, c_{shift} + 2c_L + 5c_A\}) \\
\cdot \sum_{n=1}^{f_r} |L_{in}^{(0)}| + (3.5c_A + c_{cmp} + c_{inc}) \frac{n^2}{2} \\
< (3c_{shift} + 5c_{inc} + 6c_{cmp} + 15.5c_A + 8c_L) \frac{n^2}{2} \\
\text{With the exception of branching if/else, where we take the maximum.}$$

$^7$In complexity analysis, a RAM model with logarithmic cost criterion is also used sometimes; see [2], [1] for discussions regarding the difference and appropriateness of these two cost criteria.

$^8$It is possible, however, to eliminate integer arithmetic in our pseudocode almost entirely by introducing auxiliary linked-list structures while keeping linear computational complexity.
where \( |L_{n}^{(i)}| \) denotes the size of a linked list \( L_{n}^{(i)} \). Consequently, we obtain the following upper bound on the Bottom-to-Top algorithm time complexity for \( r = 2 \):

\[
2c_{L} + c_{A} + c_{inc} \cdot |A| + (c_{L} + 3c_{A} + 2c_{inc})n + (3c_{shift} + 5c_{inc} + 6c_{cmp} + 15.5c_{A} + 8c_{L}) \sum_{i=0}^{L-1} \frac{n}{2^i} < 2c_{L} + c_{A} + c_{inc} \cdot |A| + (6c_{shift} + 12c_{inc} + 12c_{cmp} + 3c_{A} + 17c_{L})n.
\]

Similarly, the space complexity is bounded by \( C_{1} \cdot |A| + C_{2}n \), where \( C_{1} \) and \( C_{2} \) are constants.

2) Arbitrary \( r \): As we already mentioned, the Bottom-to-Top algorithm with an arbitrary \( r \) is reduced to the bisection algorithm by converting the \( r \)-ary tree to the binary tree. The number of added intermediate binary nodes between levels \( i \) and \( i + 1 \) is still bounded by the length of \( T_{i}^{(r)} \), which has not been created, but not yet updated. Therefore, the time complexity of the entire grammar transform is bounded by

\[
C_{\text{begin}} \cdot |A| + \sum_{i=0}^{L-1} \frac{Cn}{2^i} < C_{\text{begin}} \cdot |A| + 2Cn,
\]

The space complexity is also linear.

IV. THE CONDITIONAL MPM GRAMMAR TRANSFORM AND THE CONDITIONAL MPM CODE

Having defined the [unconditional] MPM(\( r, I \)) grammar transform, we now extend it to the conditional transform, called CMM(\( r, I \)). The input to the conditional grammar transform is the sequence of pairs

\[
\left\{ (x_{1}, y_{1}), (x_{2}, y_{2}), \ldots, (x_{n}, y_{n}) \right\}
\]

from a joint alphabet \( \mathcal{A} \times \mathcal{A}_{y} \), where \( \mathcal{A}_{y} \) is the alphabet from which the side information sequence \( y^{n} \) is drawn. To help the reader understand what the transformed CMM(\( r, I \)) grammar is, we first explain how \( G_{x_{1} \cdots x_{n} | y^{n}} \) can be constructed from top to bottom as we did in Section II-A for the unconditional grammar. First, we partition the sequences \( x_{1}x_{2} \cdots x_{n} \) and \( y_{1}y_{2} \cdots y_{n} \) from left to right into nonoverlapping subsequences of the lengths \( n_{i} \), \( i = 1, I - 1, \ldots, 0 \), where the lengths are obtained from the \( r \)-ary expansion of the integer \( n \) as shown in Section II-A. We denote these subsequences by \( x^{(n_{i})} \) and \( y^{(n_{i})} \), respectively. Accordingly, the concatenation of \( x^{(n_{i})}, x^{(n_{i-1})}, \ldots, x^{(n_{0})} \) gives the sequence \( x_{1}x_{2} \cdots x_{n} \) and the concatenation of \( y^{(n_{i})}, y^{(n_{i-1})}, \ldots, y^{(n_{0})} \) gives the sequence \( y_{1}y_{2} \cdots y_{n} \). The CMM(\( r, I \)) transform generates the top level \( I \) of the conditional grammar \( G_{x_{1} \cdots x_{n} | y^{n}} \) according to the following three steps.

Step 1: Partition \( x^{(n_{i})} \) into blocks of \( \mathcal{A} \)-symbols of length \( r^{I} \). Denote these blocks by variables \( \tilde{x}_{1}^{(I)}, \ldots, \tilde{x}_{\tilde{r}_{1}^{(r)}}^{(I)} \) and the resulting sequence \( \tilde{x}_{1}^{(I)} \cdots \tilde{x}_{\tilde{r}_{1}^{(r)}}^{(I)} \) by \( \tilde{x}_{\tilde{r}_{1}^{(r)}}^{(l)} \). Analogously, partition \( y^{(n_{i})} \) into blocks of \( \mathcal{A}_{y} \)-symbols of length \( r^{I} \), and denote these blocks by variables \( \tilde{y}_{1}^{(I)}, \ldots, \tilde{y}_{\tilde{r}_{1}^{(r)}}^{(I)} \), and the resulting sequence \( \tilde{y}_{1}^{(I)} \cdots \tilde{y}_{\tilde{r}_{1}^{(r)}}^{(I)} \) by \( \tilde{y}_{\tilde{r}_{1}^{(r)}}^{(I)} \). For brevity, we will call a block of \( \mathcal{A} \)-symbols an “\( \mathcal{A} \)-block” and a block of \( \mathcal{A}_{y} \)-symbols an “\( \mathcal{A}_{y} \)-block.”

Step 2: Visit every \( \mathcal{A}_{y} \)-block in the sequence \( \tilde{y}_{\tilde{r}_{1}^{(r)}}^{(I)} \) from left to right, and label all identical \( \mathcal{A}_{y} \)-blocks with the same integers and all distinct \( \mathcal{A}_{y} \)-blocks with distinct integers in increasing order, starting with 1. Denote each label, or a \( \mathcal{A}_{y} \)-token, by a \( \mathcal{A}_{y} \)-token. For every distinct \( \mathcal{A}_{y} \)-token \( \mathcal{A}_{y} \), let \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \) denote the subsequence \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \). We call this subsequence a conditional subsequence of \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \) since \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \) can be regarded as the sequence \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \) conditioned on the \( \mathcal{A}_{y} \)-token \( \mathcal{A}_{y} \). All conditional subsequences of \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \) are processed independently from each other at Step 3 below.

Step 3: For each distinct \( \mathcal{A}_{y} \)-token \( \mathcal{A}_{y} \), visit every \( \mathcal{A} \)-block in the conditional subsequence \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \mathcal{A}_{y} \) from left to right and label the first appearance of each distinct \( \mathcal{A} \)-block \( \mathcal{A}_{y} \) in this subsequence by a special symbol “\( s \).” If the same \( \mathcal{A} \)-block \( \mathcal{A}_{y} \) appears in \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \mathcal{A}_{y} \) again, label it by an integer so that all identical \( \mathcal{A} \)-blocks \( \mathcal{A}_{y} \) in \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \mathcal{A}_{y} \) up to the first appearance of the \( \mathcal{A} \)-block \( \mathcal{A}_{y} \) are labeled by the same integer, which is just the number of distinct \( \mathcal{A} \)-blocks \( \mathcal{A}_{y} \) in the \( \mathcal{A} \)-block \( \mathcal{A}_{y} \) up to the first appearance of the \( \mathcal{A} \)-block \( \mathcal{A}_{y} \) inclusively. Denote each label corresponding to an \( \mathcal{A} \)-block \( \mathcal{A}_{y} \) by \( \tilde{y}_{\mathcal{A}_{y}}^{(I)} \).

Apply similar notation to levels \( I - 1, \ldots, 1 \). The CMM(\( r, I \)) transform generates the levels \( I - 1 \) through 2 by repeating the following four steps for each level \( i \).

Step 1: For every \( j \) such that \( \tilde{y}_{\mathcal{A}_{y}}^{(i+1)} = \mathcal{A}_{y} \), partition the \( \mathcal{A} \)-block \( \tilde{y}_{\mathcal{A}_{y}}^{(i+1)} \mathcal{A}_{y} \) and the \( \mathcal{A}_{y} \)-block \( \tilde{y}_{\mathcal{A}_{y}}^{(i+1)} \) into \( r \) subblocks of length \( r^{i} \).
Step 2: Partition the sequence $x^{(n_i)}$ into $A$-blocks of length $r^i$ and concatenate these $A$-blocks to the $A$-sub-blocks of length $r_i$ constructed at Step 1 above. We denote this new concatenated sequence of $A$-blocks at the level $i$ by $\bar{x}^{(i)}$. Similarly, partition the sequence $y^{(n_i)}$ into $A_y$-blocks of length $r^i$ and concatenate these $A_y$-blocks to the $A_y$-sub-blocks of length $r_i$ constructed at Step 1 above. We denote this concatenated sequence of $A_y$-blocks at the level $i$ by $\bar{y}^{(i)}$.

Step 3: Visit every $A_y$-block in the sequence $\bar{y}^{(i)}$ from left to right, and label all identical $A_y$-blocks with the same integers and all distinct $A_y$-blocks with distinct integers in increasing order, starting with 1. Denote each label, or a $y$-token, corresponding to an $A_y$-block by $t^{(i)}$. For every distinct $y$-token $\gamma$, let $\bar{y}^{(i)}[\gamma]$ denote the subsequence $\{t^{(i)} : t^{(i)} = \gamma\}$. We call this subsequence the conditional subsequence corresponding to $\gamma$. All conditional subsequences of $\bar{y}^{(i)}$ are processed independently from each other at Step 4 below.

Step 4: For each distinct $y$-token $\gamma$, visit every $A_y$-block in the conditional subsequence $\bar{y}^{(i)}[\gamma]$ from left to right and label the first appearance of each distinct $A_y$-block $\alpha$ in this subsequence by a special symbol "$s$". If the same $A_y$-block $\alpha$ appears in $\bar{y}^{(i)}[\gamma]$ again, label it by an integer so that all identical $A_y$-blocks $\alpha$ in $\bar{y}^{(i)}[\gamma]$, except for the most left one, will be labeled by the same integer, which is just the number of distinct $A_y$-blocks in $\bar{y}^{(i)}[\gamma]$ up to the first appearance of the $A_y$-block $\alpha$ inclusively. For each $A_y$-block $\bar{y}^{(i)}_j$, denote its label by $t_j^{(i)}$.

For level 1, we perform only Steps 1 and 2 described above, and instead of performing Steps 3 and 4, we let $x^{(0)}$ and $y^{(0)}$ be $x$ and $y$, respectively, where in $x^{(0)}$ each $A$-block $x^{(0)}_j$ consists of only one $A$-symbol, and in $y^{(0)}$ each $A_y$-block $y^{(0)}_j$ consists of only one $A_y$-symbol. Thus, in the CMPM grammar, every level $i$ ($i = 0, \ldots, I$) is represented by the sequence of token pairs denoted by $T_i$.

$$T_i = \left\{ (t_1^{(i)}, t_2^{(i)}), (t_2^{(i)}, t_3^{(i)}), \ldots, (t_{|T_i|}^{(i)}, T_{|T_i|}) \right\}$$

Fig. 11 illustrates the above procedure for $I = 3, r = 2, n = 31, x^{(n)} = ababbaababababababababaababbaababaababbaababbaababbaabababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababbaababfa

Fig. 11. Conditional grammar $G_{x \| y^n}$; $n = 31; I = 3; r = 2$. 

A. Conditional Bottom-to-Top Algorithm

In the previous subsection, we briefly explained how the CMPM grammar can be constructed from top to bottom. Now we present our new conditional Bottom-to-Top algorithm which is an extension of the Bottom-to-Top algorithm described in Section III. Similarly to the unconditional algorithm, the conditional Bottom-to-Top algorithm processes two adjacent levels at a time while moving up. That is, when the $i$th level is being updated, the $(i + 1)$th level is being created. The algorithm will create and use the following structures.
Arrays $T_x^{(i)}[1 \cdots [n/r^i]]$, $i = 0, \ldots, I$, corresponding to the token sequences of the unconditional transform of $x^n$, are an auxiliary structure used in intermediate steps of our algorithm.

Arrays $T_y^{(i)}[1 \cdots [n/r^i]]$ and $T_y^{(i)}[1 \cdots [n/r^i]]$ are the output of our conditional grammar transform and represent the conditional CPMF($r, I$) grammar being built.

Lists $L_{x,m}^{(i)}$ and $L_{y,m}^{(i)}$ of the arrays’ indexes are defined by the following expressions:

\[
L_{x,m}^{(i)} = \{m\} \cup \{j : T_x^{(i)}[j] = m\}, \quad m = 1, \ldots, f_x^{(i)},
\]

\[
L_{y,m}^{(i)} = \{m\} \cup \{j : T_y^{(i)}[j] = m\}, \quad m = 1, \ldots, f_y^{(i)},
\]

where $f_x^{(i)}$ is the number of distinct integers in the array $T_x^{(i)}[1 \cdots [n/r^i]]$ and $f_y^{(i)}$ is the number of distinct integers in the array $T_y^{(i)}[1 \cdots [n/r^i]]$.

Lists $S_x^{(i)}$ and $S_y^{(i)}$ of the lists are defined as follows:

\[
S_x^{(i)} = \{L_{x,1}^{(i)}, \ldots, L_{x,f_x^{(i)}}^{(i)}\}
\]

\[
S_y^{(i)} = \{L_{y,1}^{(i)}, \ldots, L_{y,f_y^{(i)}}^{(i)}\}
\]

**Bottom level creation ($i = 0$):** We denote a mapping from the alphabet $A$ to the set of integers $\{1, \ldots, |A|\}$ by $\text{int}_A(x_j)$ and a mapping from the alphabet $A_y$ to the set of integers $\{1, \ldots, |A_y|\}$ by $\text{int}_{A_y}(y_j)$. The conditional algorithm creates the arrays

\[
T_x^{(0)}[1 \cdots n] = \{\text{int}_A(x_1), \ldots, \text{int}_A(x_n)\}
\]

\[
T_y^{(0)}[1 \cdots n] = \{\text{int}_{A_y}(y_1), \ldots, \text{int}_{A_y}(y_n)\}
\]

and the sets of lists

\[
S_x^{(0)} = \{L_{x,1}^{(0)}, \ldots, L_{x,|A|}^{(0)}\}
\]

\[
S_y^{(0)} = \{L_{y,1}^{(0)}, \ldots, L_{y,|A_y|}^{(0)}\}
\]

in the same way as does the unconditional algorithm, with only distinction that now $T_x^{(i)}$ does not need to be updated. The lists $L_x^{(0)}$ and $L_y^{(0)}$ are defined as follows:

\[
L_{x,m}^{(0)} = \{m\} \cup \{j : T_x^{(0)}[j] = m\}, \quad m = 1, \ldots, |A|
\]

\[
L_{y,m}^{(0)} = \{m\} \cup \{j : T_y^{(0)}[j] = m\}, \quad m = 1, \ldots, |A_y|
\]

are created with the help of auxiliary arrays $\text{Table}_x[1 \cdots |A|]$ and $\text{Table}_y[1 \cdots |A_y|]$.

Creating the level $i + 1$ and updating the level $i$ ($i = 0, \ldots, I - 1$):

Suppose the level $i$ has just been created. That is, the sets of lists $S_x^{(i)}$ and $S_y^{(i)}$ and the array $T_x^{(i)}[1 \cdots [n/r^i]]$ have been created, but not yet updated. The algorithm will create $S_x^{(i+1)}$, $T_x^{(i+1)}[1 \cdots [n/r^{i+1}]]$, $S_y^{(i+1)}$, $T_y^{(i+1)}[1 \cdots [n/r^{i+1}]]$, and $T_y^{(i+1)}[1 \cdots [n/r^{i+1}]]$, and will update $T_y^{(i)}[1 \cdots [n/r^i]]$ and $T_y^{(i)}[1 \cdots [n/r^i]]$ according to the following steps.

**Step 1:** Use the set of lists $S_x^{(i)}$ and the array $T_x^{(i)}[1 \cdots [n/r^i]]$ to create the set of lists $S_x^{(i+1)}$ and the array $T_x^{(i+1)}[1 \cdots [n/r^{i+1}]]$.

Then, destroy the set $S_y^{(i)}$ and the array $T_y^{(i)}[1 \cdots [n/r^i]]$. The piece of pseudocode below implements Step 1 for the case of the bisection grammar ($r = 2$).

Begin
1 Set $s(i + 1) := 0$;
2 Initialize the array $T_x^{(i+1)}[1 \cdots [n/2^{i+1}]]$ of integers;
3 initialize the array $INDICATOR[1 \cdots f_x^{(i)}]$ of pointers to null;
4 For m = 1 to $f_x^{(i)}$ do
5 skip the first element of the list $L_{x,m}^{(i)}$;
6 For every odd index $j \leq [n/2^i]$ in the list $L_{x,m}^{(i)}$,
7 if $INDICATOR[R[T_x^{(i)}[j]] = \text{null}$, then
8 $s(i + 1) := s(i + 1) + 1$;
9 Create the new list $L_{x,a(i+1)}^{(i)} = \{s(i + 1); (j + 1)/2\}$
10 and append it to the set $S_x^{(i+1)}$;
11 make the pointer $INDICATOR[R[T_x^{(i)}[j]] point to this list $L_{x,a(i+1)}^{(i)}$;
12 $T_x^{(i+1)}[(j + 1)/2] := s(i + 1)$;
else
13 append the index $(j + 1)/2$ to the list pointed by the pointer $INDICATOR[R[T_x^{(i)}[j + 1]]$ and
14 replace the array’s element $T_x^{(i+1)}[(j + 1)/2]$ by the first member of this list;
15 traverse it from left to right and do
16 For every odd index $j \leq [n/2^i]$ in the list $L_{x,m}^{(i)}$,
17 $INDICATOR[R[T_x^{(i)}[j + 1]] := \text{null}$;
18 Destroy the array $INDICATOR[1 \cdots f_x^{(i)}]$ and release the corresponding allocated memory.
19 Destroy the set $S_y^{(i)}$, all the lists $L_{x,1}^{(i)}, \ldots, L_{x,|A|}$,
20 and the array $T_y^{(i)}[1 \cdots [n/r^i]]$ and, release the corresponding allocated memory.
End.

**Step 2:** Use the set of lists $S_y^{(i)}$ and the array $T_y^{(i)}[1 \cdots [n/r^i]]$ to create the set of lists $S_y^{(i+1)}$ and array $T_y^{(i+1)}[1 \cdots [n/r^{i+1}]]$.

---

The notations $T_x^{(i)}[1 \cdots [n/r^i]]$ and $T_y^{(i)}[1 \cdots [n/r^i]]$ are used here abusively to denote both the unupdated token sequence and the updated token sequence.
and then, destroy the set $S_y^{(i)}$ in the exact same way as in Step 1.

Step 3: Use the set of lists $S_y^{(i+1)}$ and the array $T_x^{(i+1)}[1 \ldots [n/r^{i+1}]]$ to create the array $T_x^{[i+1]}[1 \ldots [n/r^{i+1}]]$ and update the array $T_x^{[i]}[1 \ldots [n/r^i]]$ according to the following piece of pseudocode.

Begin
Initialize the array $T_x^{[i+1]}[1 \ldots [n/2^{i+1}]]$ of integers;
initialize the array $INDICATOR[1 \ldots f_x, +, 1]$ of integers to all zeros;
For $m = 1$ to $f_x, +, 1$ do
set $s(i + 1) := 0$;
skip the first element of the list $L_{y, m}^{(i+1)}$,
traverse it from left to right and do
For every index $j$ in the list $L_{y, m}^{(i+1)}$, if $INDICATOR[T_x^{[i+1]}[j]] = 0$, then
$s(i + 1) := s(i + 1) + 1$;
$INDICATOR[T_x^{[i+1]}[j]] := s(i + 1)$;
else
$T_x^{[i+1]}[j] := INDICATOR[T_x^{[i+1]}[j]]$;
$T_x^{[i]}[2j - 1] := 0$ and $T_x^{[i]}[2j] := 0$;
$T_x^{[i]}[2j - 1] := 0$ and $T_x^{[i]}[2j] := 0$ (i.e.,
delete tokens $T_x^{[i]}[2j - 1], T_x^{[i]}[2j], T_x^{[i]}[2j - 1],$
and $T_x^{[i]}[2j]$);
traverse the list $L_{y, m}^{(i+1)}$ from left to right and do
For every index $j$ in the list $L_{y, m}^{(i+1)}$, $INDICATOR[T_x^{[i+1]}[j]] := 0$;
Destroy the array $INDICATOR[1 \ldots f_x, +, 1]$ and release the corresponding allocated memory.
End

The arrays $T_x^{[i]}[1 \ldots [n/r^i]]$ and $T_y^{[i]}[1 \ldots [n/r^i]]$ are the output of the grammar transform for level $i$.

Remark 9: Although not natural, the alphabet ordering in every array $T_y^{[i]}[1 \ldots [n/r^i]]$ is unique in the sense that the decoder will construct exactly the same arrays $T_y^{[i]}[1 \ldots [n/r^i]]$ as does the encoder. Therefore, we do not have to rename the integers in the array $T_y^{[i]}[1 \ldots [n/r^i]]$ according to the natural ordering. On the other hand, since the indices in each list $L_{y, m}^{(i)}$ are placed (at Step 2) in the strictly increasing order, Step 3 of the algorithm implicitly provides the conditional natural alphabet ordering of the array $T_x^{[i]}[1 \ldots [n/r^i]]$.

Remark 10: The described implementation of CMPM Transform is “off-line.” Similarly to its unconditional prototype, CMPM Transform itself can be implemented sequentially with linear computational complexity.

Recall that the sequences of token pairs $T_i (i = 0, \ldots, I)$ can be obtained from the sequence of pairs

$$
\left( T_x^{(i)}[1] \right) \left( T_x^{(i)}[2] \right) \cdots \left( T_x^{(i)}[[n/r^i]] \right)
$$

by deleting pairs of the form $\left( \alpha \right)$. For example, for the conditional grammar $G_{x, |y,n}$ shown in Fig. 5, the conditional Bottom-to-Top algorithm will produce the following $T_i$: $T_3 = \left( s \begin{array} {c} 1 \\ 2 \\ 3 \end{array} \right)$ $T_2 = \left( s \begin{array} {c} 1 \\ 2 \\ 2 \\ 3 \\ 4 \end{array} \right)$ $T_1 = \left( s \begin{array} {c} 1 \\ 1 \\ 1 \\ 1 \\ 3 \\ 2 \\ 3 \\ 3 \\ 3 \end{array} \right)$ $T_0 = \left( a \begin{array} {c} a \\ b \\ b \\ b \\ c \\ c \\ d \\ d \\ c \\ c \\ c \\ c \\ c \end{array} \right)$ (IV.1)

B. Complexity of Conditional Bottom-to-Top Algorithm

To estimate the computational complexity of the conditional Bottom-to-Top algorithm, we follow the approach we took for the unconditional algorithm analysis and use the same notation for the time costs. First, we consider the case $r = 2$ (bisection). The execution time of the bottom level creation is the sum of the amounts of time required to create the following structures:

- $T_x^{[0]}[1 \ldots [n]], T_y^{[0]}[1 \ldots [n]],$ and $T_y^{[0]}[1 \ldots [n]] \rightarrow \left( 3c_A + c_{inc} \right) n$
- $T_x^{[0]}[1 \ldots [A]], T_y^{[0]}[1 \ldots [A]] \rightarrow \left( c_A + c_{inc} \right) A$
- $L_{x, 1}^{(0)}, L_{x, 2}^{(0)}, L_{y, 1}^{(0)}, L_{y, 2}^{(0)} \rightarrow c_L (n + |A|) + (2c_A + c_{inc}) n$
- $L_{y, 1}^{(0)}, L_{y, 2}^{(0)} \rightarrow c_L (n + |A|) + (2c_A + c_{inc}) n$
- $S_{y, 0}^{(0)} \rightarrow c_L |A|$
- $S_{y, 0}^{(0)} \rightarrow c_L |A|$

Thus, the time required to create the bottom level is

$$(2c_L + c_A + c_{inc})(|A| + |A|) + (2c_L + 7c_A + 3c_{inc}) n$$

To calculate the execution time of the pseudocode at Step 1 of creating level $i + 1$ and updating level $i$, we sum all the time entries in Table II. We obtain

$$4c_A + c_{inc} n \left( v_{x, i} \right) + \left( c_A + c_{inc} + 2c_L \right) f_x^{s, i}$$

+ $\left( 3c_L + 2c_{concat} + 4c_A + 2c_{inc} \right)$

+ $\left( 3c_L + 2c_{concat} + 4c_A + 2c_{inc} \right)$

+ $\left( 5c_{concat} + 4c_{concat} + 4c_A + 2c_{inc} \right)$

+ $\left( 2c_L + 3c_A \right)$

* $\sum_{m=1}^{f_x^{s, i}} |L_{y, m}^{(i)}|$

$$\left< 9c_L + 3c_{concat} + 8.5c_A + 4c_{inc} \right> \frac{n}{2}$$
In a very similar manner we can see that the execution time of the pseudocode at Steps 2 and 3 of creating level $i + 1$ and updating level $i$ is upper-bounded by the expression

$$C_{time}(c_L + c_{shft} + c_{cmpg} + c_A + c_{inc}) \frac{n}{2^r},$$

where $C_{time}$ is a constant. As a result, we obtain the following upper bound on the conditional Bottom-to-Top algorithm time complexity for $r = 2$:

$$C(c_L + c_{shft} + c_{cmpg} + c_A + c_{inc})(|A| + |A_0| + n),$$

where $C$ is a constant. It is easy to see that the storage complexity of the conditional Bottom-to-Top algorithm is linear with respect to the sum $(|A| + |A_0| + n)$.

In a manner similar to that discussed in Remark 8, the conditional Bottom-to-Top algorithm with an arbitrary $r$ can be reduced to the conditional bisection ($r = 2$) algorithm by converting the $r$-ary tree to the binary tree. Therefore, the time and storage complexities of the conditional Bottom-to-Top algorithm with an arbitrary $r$ are also linear with respect to the sum $(|A| + |A_0| + n)$.

### C. Arithmetic Encoder

Let $T_i'$ ($i = 0, \ldots, I$) be a sequence obtained from $T_i$ by removing the first (the most left) appearance of the pair $(s, s)$ for every distinct token $s$. Continuing our example, for $T_i$ shown in (IV.1) we will have the following $T_i'$:

$$T_0' = \left( \begin{array}{c} a \\ b \\ c \\ d \\ b \\ c \\ d \\ a \\ (c + d + e) \end{array} \right)$$

Since every conditional subsequence $q^{(i)}_j$ begins with the symbol “s,” $T_i'$ can easily be recovered from $T_i'$. Therefore, the arithmetic encoder encodes $T_i'$ instead of $T_i$. Let us denote the token sequences, constituting the token pairs in $T_i'$, as follows:

$$\left\{ \left( s, q^{(i)}_j \right) \right\} \triangleq T_i'$$

$$q^{(i)}_j \triangleq q^{(i)}_j \left( q^{(i-1)} \right)$$

Let $A^{(i)}_U$ be the set of all the distinct tokens in $\{ q^{(i)}_1, q^{(i)}_2, \ldots, q^{(i)}_n \}$. Since the conditional grammar can be transformed back into the original string $x^i$ only in the top-to-bottom order, the arithmetic encoder encodes the grammar in this, top-to-bottom, order. Thus, the sequences $q^{(i)}_1, q^{(i-1)}_2, \ldots, q^{(i)}_n$ are encoded in the indicated order by a conditional arithmetic encoder. Specifically, the arithmetic encoder encodes every $x$-token $q^{(i)}_j$ in the sequence $q^{(i)}_j$, conditioning on $q^{(i)}_j$; therefore, the conditional subsequences $q^{(i)}_j\gamma$ are encoded independently for every distinct token $\gamma \in A^{(i)}_U$. The encoder restarts at the beginning of each next sequence $q^{(i)}_k$, and concatenates the binary codewords for each $q^{(i)}_k$ in the indicated order into one single binary codeword $B(G_{x^i}q^{(i)}_k)$. For $i = I, \ldots, 1$, the $x$-token sequence $q^{(i)}_k$ is encoded as follows. We associate each distinct pair of symbols $(s, s)$ with a counter $c_{\gamma}(x)$, where $\gamma \in A^{(i)}_U$ and $x$ belongs to the set of all distinct $x$-token symbols in the conditional subsequence $q^{(i)}_j\gamma$. Initially, $c_{\gamma}(x)$ is set to 1 if $x \in \{ s, 1 \}$ and 0 otherwise. The initial alphabet used by the arithmetic code is $\{ s, 1 \}$. Denoting $q^{(i)}_k$ by $q^{(i)}_j\gamma$, we encode each token $q^{(i)}_k$ in the sequence $q^{(i)}_k\gamma$ and update the related counters according to the following steps.

**Step 1:** Encode $q^{(i)}_k\gamma$ by using the probability

$$c_{\gamma}(x) / \sum_{\alpha} c_{\gamma}(x)$$

where the summation $\sum_{\alpha}$ is taken over $\{ s, 1 \} \cup \{ 2, 3, \ldots, j + 1 \}$, and $j$ is the number of times that the pair $(s, s)$ has occurred before the position $i$. Note that the alphabet used at this point by the arithmetic code is $\{ s, 1 \} \cup \{ 2, 3, \ldots, j + 1 \}$.

**Step 2:** Increase the counter $c_{\gamma}(q^{(i)}_k\gamma)$ by 1.

**Step 3:** If $q^{(i)}_k = s$, increase the counter $c_{\gamma}(j + 2)$ from 0 to 1, where $j$ is defined in Step 1.

Repeat the above procedure until the entire sequence $q^{(i)}_k$ is encoded.

To encode the bottom level array $q^{(0)}$, we use the following two-steps conditional adaptive arithmetic coding with the fixed alphabet $A$.

11The initial alphabet includes the symbol “1” due to the fact of removing the first appearance of the symbol $s$ from each conditional subsequence of tokens.
Step 1: Denoting $t_k^{(0)}$ by $\gamma \in A_y$, encode $v_k^{(0)}$ by using the probability

$$c_\gamma \left( v_k^{(0)} \right) \sum_{\alpha \in A} c_\alpha (\alpha)$$

Step 2: Increase the counter $c_\gamma (t_k^{(0)})$ by 1.

Repeat the above procedure until the entire sequence $v^{(0)}$ is encoded.

Example 2: The product of the probabilities used in the arithmetic coding of the entire conditional grammar represented by token sequences (IV.2) is given by

$$p = \prod_{i=1}^n p_i^{(i)} = \prod_{i=1}^n \left( \prod_{\gamma \in A_y} c_\gamma \left( v_{i,j}^{(i)} \right) \right)$$

Remark 11: To decode the binary codeword $B(G^{(y)} | y^n)$ the decoder must know the side information sequence $y^n$ which in turn implies the knowledge of the length $n$ of the sequence. Therefore, the decoder will be able to compute digits $h_j$ in the $r$-ary expansion of $n$.

Remark 12: Similarly to that of a MPM code, the arithmetic coding part of a CMPM code has linear computational complexity. This is achieved by using the multilevel arithmetic coding algorithm [19] to encode the conditional token subsequences.

D. Compression Rate Related to the Empirical Entropy of the Conditional Grammar

Let $f_i^s$ denote the number of the distinct pairs at $T_i$. Let $l_i \triangleq |T_i|$.

Let $T_i'' \triangleq \left\{ \left( v_{i,j}^{(i)} \right) : v_{i,j}^{(i)} \neq \emptyset \right\}$

that is, $T_i''$ is the subsequence of $T_i$ obtained by removing pairs of the form $(e,0)$. Let

$$\eta_i^{(i)} \triangleq \eta_i^{(i)} \cdots \eta_i^{(i)}|T_i''$$

Let us agree that $\eta_i^{(0)} \triangleq \emptyset$ and $\eta_i^{(0)} \triangleq \emptyset$.

Definition IV.1: We define the conditional entropy of the CMPM grammar $G^{(y)} | y^n$ as follows:

$$H_G(x^n | y^n) \triangleq \sum_{i=0}^{l_i} H(\eta_i^{(i)} | \eta_i^{(i)})$$

where $H(\eta_i^{(i)} | \eta_i^{(i)})$ is the unnormalized conditional empirical entropy of the sequence $\eta_i^{(i)}$ given $\eta_i^{(i)}$.

Lemma IV.1: Let $\sum_{i=0}^{l_i} B_i$ be the size of the output binary codeword for the input sequence $x^n$. Then

$$\sum_{i=0}^{l_i} B_i \leq H_G(x^n | y^n) + \sum_{i=0}^{l_i} l_i + \sum_{i=1}^{f_i^s} l_i + 2 |A_y| |A|$$

Proof: If we fix a symbol $\alpha \in A_y^{(y)}$, then, as it was shown in (II.13) and (II.14), the number of bits required to encode the conditional subsequence $v_i^{(i)} | \alpha$ is bounded as follows:

$$B_i | \alpha \leq 2 l_i + f_i^s + H(\eta_i^{(i)} | \alpha)$$

$$B_i | \alpha \leq l_i + H(\mathbf{w}^{(i)} | \alpha) + |A_y||A|$$

Analogously,

$$B_0 = \sum_{\alpha \in A_y} B_0 | \alpha \leq l_0 + H(\mathbf{w}^{(0)} | \emptyset) + |A_y||A|$$

Finally, the total number of bits in the output binary codeword is bounded as follows:

$$\sum_{i=0}^{l_i} B_i \leq \sum_{i=0}^{l_i} H(\eta_i^{(i)} | \eta_i^{(i)}) + \sum_{i=0}^{l_i} l_i + \sum_{i=1}^{l_i} f_i^s - l_i + |A_y||A|$$

$$= H_G(x^n | y^n) + 2 \sum_{i=0}^{l_i} l_i + \sum_{i=1}^{l_i} f_i^s - l_i + |A_y||A|$$

E. Redundancy of the CMPM Code

Here we extend the finite-state arithmetic encoder model, which we used in Section II-F, by conditioning on side information sequences. Let

$$G^{(2N+1)} \times S_h \times (A \times S_h) \rightarrow [0, 1]$$

be a transitional conditional probability function satisfying

$$\sum_{s \in S_h, \alpha \in A} p(s, x \mid \alpha, y^{2N+1}) = 1$$

for any $\alpha \in S_h$ and $y^{2N+1} \in A_y^{2N+1}$. Random transitions between contexts from the set $S_h$ are allowed. For any sequences $x^n \in A^n$ and $y^n \in A^n$, the compression rate in bits per letter resulting from the arithmetic encoder with transition probability $p$ is given by

$$-\frac{1}{n} \log \left( \prod_{s_1, s_2, \ldots, s_n \in S_h} \prod_{i=1}^{n} p(s_i, x_i | s_{i-1}, y_{i-N}^{i+N}) \right)$$

where $s_0$ is the initial context, and $y_{i-N}^{i+N} \in A_y^N$ and $y_{i+1}^{i+1} \in A_y^N$ are some prescribed initial and final segments of the side information sequence. For the rest of the paper, we assume $N < n$.

Remark 13: The conditional arithmetic encoder, mentioned above, is a finite-state machine which encodes each symbol $x_i$ conditionally based on the context $s_{i-N}$ and the side information subsequence $y_{i-N} \cdots y_{i+N}$. Thus, the parameter $N$ defines the sliding window of size $2N + 1$.

The following definition extends the notion of the $k$-context [unconditional] empirical entropy to the $k$-context conditional
empirical entropy $H_N^k(x^n|y^n)$ with a sliding window of size $2N + 1$.

**Definition IV.2:**

$$H_N^k(x^n|y^n) \overset{\triangle}{=} - \frac{1}{n} \log \left( \max_{p} \max_{s_{0} \in S_k} \max_{y_{N+N}^{+}} \sum_{i=1}^{n} p(s_i, x_i|s_{i-1}, y_{i+N}^{+}) \right) \quad \text{(IV.4)}$$

where the maximization for $p$ is taken over all transitional conditional probability functions $p$ from $(A_y^{2N+1} \times S_k) \times (A \times S_k) \rightarrow [0, 1]$.

**Definition IV.3:** We define the worst case redundancy (per sample) $R_{n, k, N}$ of the algorithm against the $k$-context conditional empirical entropy $H_N^k(x^n|y^n)$ as

$$R_{n, k, N} \overset{\triangle}{=} \max_{\mu \in \mathcal{A}_y^n} \left[ \frac{1}{n} \sum_{i=0}^{n-1} B_k - H_N^k(x^n|y^n) \right]. \quad \text{(IV.5)}$$

**Theorem IV.1:**

$$R_{n, k, N} < C_{\text{CMIM}} \cdot \frac{1}{\log n} \quad \text{(IV.5)}$$

where $C_{\text{CMIM}} = 10(r \log k + 2r \log |A_y| + 2r + 2) \log |A_y|$. $p$ of course, depends on $x_{1}2 \cdots x_{n}$ and $y_{1}y_{2} \cdots y_{n}$. Let $p$ be a transitional probability function which maximizes (IV.4); $\tilde{p}$, of course, depends on $x_{1}2 \cdots x_{n}$ and $y_{1}y_{2} \cdots y_{n}$. For any $x_{1}, \ldots, x_{q} \in A$, and $y_{1}, \ldots, y_{q} \in A_y$, define

$$\tau(x_{1}, \ldots, x_{q}|y_{1}, \ldots, y_{q}) \overset{\triangle}{=} \max_{s_{0} \in S_k} \max_{y_{N+N}^{+}} \left( \frac{1}{n} \sum_{i=1}^{q} p(s_i, x_i|s_{i-1}, y_{i+N}^{+}) \right) \quad \text{(IV.6)}$$

Consequently,

$$1 \leq \sum_{(x_{1}, \ldots, x_{q}) \in A^n} \tau(x_{1}, \ldots, x_{q}|y_{1}, \ldots, y_{q}) \leq k \cdot |A_y|^{2N}.$$ 

For each $q \geq 1$, we normalize $\tau(\cdot)$ so that

$$\tilde{p}(x_{1}, \ldots, x_{q}|y_{1}, \ldots, y_{q}) \overset{\triangle}{=} \frac{Q_{k}}{k \cdot |A_y|^{2N}} \tau(x_{1}, x_{2}, \ldots, x_{q}|y_{1}, \ldots, y_{q})$$

is a probability distribution over $A$ for any $y_{1}, \ldots, y_{q} \in A_y$, i.e.,

$$\sum_{(x_{1}, \ldots, x_{q}) \in A^n} \tilde{p}(x_{1}, \ldots, x_{q}|y_{1}, \ldots, y_{q}) = 1.$$ 

Generally speaking, $Q_{k}$ depends on $y_{1}, \ldots, y_{q}$. Nonetheless, it is easy to see that $Q_{k}$ satisfies $1 \leq Q_{k} \leq k \cdot |A_y|^{2N}$.

Now we are ready to extend our reasoning in (II.21)–(II.25) to the conditional case. Let us agree, that when used as arguments of the functions $\tau(\cdot), p^*(\cdot)$, and $\tilde{p}(\cdot)$, the records $\nu^{(i)}_j$ and $\nu^*(i)_j$ denote the $A$-sequence represented by the token $y_{j}^{(i)}$ and the $A_y$-sequence represented by the token $y_{j}^{(i)}$ respectively. Then

$$H_N^k(x^n|y^n) \equiv - \frac{1}{n} \log \tau(x^n|y^n) \geq - \frac{1}{n} \sum_{i=0}^{n} \sum_{j=1}^{I} \log \tau \left( \nu^{(i)}_j | \nu^*(i)_j \right) \quad \text{(IV.7)}$$

and

$$|\nu^{(i)}_j| - \sum_{j=1}^{I} \log \tau \left( \nu^{(i)}_j | \nu^*(i)_j \right)$$

$$= - \sum_{j=1}^{I} \log \left[ \frac{k \cdot |A_y|^{2N}}{Q_k} \right] p^* \left( \nu^{(i)}_j | \nu^*(i)_j \right)$$

$$= \sum_{j=1}^{I} - \log \tilde{p} \left( \nu^{(i)}_j | \nu^*(i)_j \right)$$

$$\geq \sum_{j=1}^{I} - \log \tilde{p} \left( \nu^{(i)}_j | \nu^*(i)_j \right)$$

where the minimum is over all transitional conditional probability distributions $\tilde{p}(\cdot)$ on $A^n \times A_y^n$, and function $p^*(\cdot)$ can be viewed as one of such distributions. Combining (IV.7)–(IV.9) together, we get

$$nH_N^k(x^n|y^n) \geq \sum_{i=0}^{I} H \left( \nu^{(i)}_j | \nu^*(i)_j \right) - (\log k + 2N \log |A_y|) \sum_{i=0}^{I} l_{i}$$

$$= H_C(x^n|y^n) - (\log k + 2N \log |A_y|) \sum_{i=0}^{I} l_{i} \quad \text{(IV.10)}$$

The information inequality [3, Theorem 2.6.3, p. 26] together with the definition of traditional conditional entropy

$$H(X|Y) \overset{\triangle}{=} \sum_{y} \Pr\{Y = y\} H(X|Y = y)$$

(see also [3, pp. 23, 27]) implies

$$H \left( \nu^{(i)}_j | \nu^*(i)_j \right) \equiv \min_{\tilde{p}} \sum_{j=1}^{I} - \log \tilde{p} \left( \nu^{(i)}_j | \nu^*(i)_j \right)$$

$$\leq \sum_{j=1}^{I} - \log \tilde{p} \left( \nu^{(i)}_j | \nu^*(i)_j \right)$$

where the minimum is over all conditional probability distributions $\tilde{p}$ on $A^n \times A_y^n$, and function $p^*(\cdot)$ can be viewed as one of such distributions. Combining (IV.7)–(IV.9) together, we get

$$nH_N^k(x^n|y^n) \geq \sum_{i=0}^{I} H \left( \nu^{(i)}_j | \nu^*(i)_j \right) - (\log k + 2N \log |A_y|) \sum_{i=0}^{I} l_{i}$$

$$= H_C(x^n|y^n) - (\log k + 2N \log |A_y|) \sum_{i=0}^{I} l_{i} \quad \text{(IV.10)}$$
Combining inequality (IV.10) with Lemma IV.1 and utilizing relations (II.7) and (II.8), we get

\[
\sum_{i=0}^{I} B_i - nH_N^K(x^n|y^n) \\
\leq 2 \sum_{i=0}^{I} l_i + \sum_{i=0}^{I} f_i^q + (\log k + 2N \log |A_y|) \sum_{i=0}^{I} l_i + \log n + |A||A_y| \leq (\log k + 2N \log |A_y| + 2 + 1) \sum_{i=0}^{I} l_i + \frac{1}{r} + |A||A_y|.
\]  

(IV.11)

Since the numbers \(l_i\) and \(f_i^q\) for our conditional grammar are identical to those for the unconditional grammar for the joint sequence \(\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}\), we have, for \(\log \log n \leq I \leq \log n\)

\[
\sum_{i=0}^{I} l_i \leq C_{\text{CPM}}^{\text{C-MPL}} \frac{n}{\log n}.
\]  

(IV.12)

where \(C_{\text{CPM}}^{\text{C-MPL}} = 10r^2 \log |A||A_y|\). Substituting (IV.12) into (IV.11) yields

\[
\sum_{i=0}^{I} B_i - nH_N^K(x^n|y^n) \\
< 10r^2 \log k + 2N \log |A_y| + 2 + 1 \cdot \left( \frac{n}{\log n} \right) \log |A||A_y| + \frac{\log n}{r} + |A||A_y|.
\]  

(IV.13)

Finally, we obtain the value of the constant \(C_{\text{CPM}}\) from the right-hand side of the above expression for sufficiently large \(n\)

\[
C_{\text{CPM}} = 10r^2 \log k + 2N \log |A_y| + 2 + 2 \log |A||A_y|.
\]

Corollary 1: For any stationary, ergodic source pair \(XY = \{X_iY_i\}_{i=1}^{\infty}\) with alphabet \(A \times Y\)

\[
\frac{1}{n} \sum_{i=0}^{I} B_i \rightarrow H_\infty(X|Y)
\]

with probability one as \(n \rightarrow \infty\), where \(H_\infty(X|Y)\) denotes the conditional entropy rate of \(\{X_i\}\) given \(\{Y_i\}\) defined as

\[
\lim_{n \rightarrow \infty} \frac{1}{n} H(X_1, \ldots, X_n|Y_1, \ldots, Y_n).
\]

Proof: Let \(\lambda = |A|^r\). For any \(A\)-sequence \(u\) with length \(|u| < n\) and any \(w \in A_y^{N+1}\), let \(f(u, w|x^n|y^n)\) be the frequency of a special conglomerate \((u, w)\) in the joint sequence

\[
\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \in A^n \times A_y^n
\]

computed in a cyclic manner as follows:

\[
(u, w|x^n|y^n) \\
= \left\{ 1 \leq i \leq n; x_i = x_{i+N+1} \text{ and } y_i = y_{i+N} \right\} w
\]

where as convention, \(x_i = x_j\) and \(y_i = y_j\) whenever \(i = j \mod n\). We define the \(q\)-th order conditional empirical entropy with sliding window of size \(2N + 1\)

\[
H_{q,N} \triangleq - \sum_{u \in A_q^{N+1}} \sum_{w \in A} \sum_{\alpha \in A} \left( \log \left( \frac{f(u, w|x^n|y^n)}{f(u, w|x^n|y^n)} \right) \right)
\]

where \(X^n = X_1 \cdots X_n\) and \(Y^n = Y_1 \cdots Y_n\). Since we can always define a context in (IV.4) at time \(i\) as the most recent past \(q\) characters \(x_{i-q} \cdots x_{i-q+i}\), there always exists a probability function \(p\) such that

\[
\frac{1}{n} \log \left( \max_{s_i \in S_k} \max_{y_{i-N}^{i-N}} s_i \frac{1}{n} \sum \prod_{i=1}^{n} p(s_i, x_i) s_{i+1}^{i+N} \right) \leq H_{q,N}.
\]  

(IV.14)

Therefore,

\[
H_N^K(x^n|y^n) \leq H_{q,N}.
\]

Thus, from Theorem IV.1

\[
\frac{1}{n} \sum_{i=0}^{I} B_i \leq H_{q,N} + C \frac{1}{\log n}.
\]

Letting \(n \rightarrow \infty\) and applying the ergodic theorem, we get

\[
\lim_{n \rightarrow \infty} H_{q,N} = H(X_i|X_{i-q} \cdots X_{i-q+i} \cdots X_{i-1} Y_{i+N}^{i+N})
\]

with probability one and

\[
\limsup_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=0}^{I} B_i \right) \leq H(X_i|X_{i-q} \cdots X_{i-q+i} \cdots X_{i-1} Y_{i+N}^{i+N})
\]

with probability one. Letting \(q \rightarrow \infty\) and \(N \rightarrow \infty\) in the above inequality yields

\[
\lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=0}^{I} B_i \right) \leq H_\infty(X|Y)
\]

(IV.15)

On the other hand, from source coding theory [5], [22], we have

\[
\limsup \left( \frac{1}{n} \sum_{i=0}^{I} B_i \right) \geq H_\infty(X|Y)
\]

with probability one,

\[
\liminf \left( \frac{1}{n} \sum_{i=0}^{I} B_i \right) \geq H_\infty(X|Y)
\]  

(IV.16)

Inequalities (IV.15) and (IV.16) together imply

\[
\frac{1}{n} \sum_{i=0}^{I} B_i \rightarrow H_\infty(X|Y)
\]

with probability one.
V. Simulation Results

In this section, we present simulation results on random binary sequences. We demonstrate the asymptotical convergence of the compression rate and the normalized grammar entropy \( \frac{1}{n} H_G(x^n | y^n) \) to the conditional entropy rate \( H_{\infty}(X | Y) \) for so-called compound first-order Markov binary source pairs \( XY = \{X_i Y_i\}_{i=1}^{\infty} \). In such a pair, the sequence \( \{X_i\} \) is obtained by transmitting a first-order Markov source \( \{Y_i\} \) via a memoryless binary-symmetrical channel. We generate \( \{Y_i\} \) and \( \{X_i\} \) as follows:

\[
Y_i = Y_{i-1} \oplus V_i
\]

\[
X_i = Y_i \oplus W_i
\]

where \( \{V_i\} \) is independent and identically distributed (i.i.d.) with the probability of symbol 1 being \( q \) and \( \{W_i\} \) is i.i.d. with the probability of symbol 1 being \( p \), and \( \{V_i\} \) and \( \{W_i\} \) are also jointly independent. The notation \( \oplus \) stands for modulo-2 addition, and the initial distributions of \( \{Y_i\} \) is uniform. Thus, the transition matrix of the Markov source \( \{Y_i\} \) is

\[
\begin{bmatrix}
1 - q & q \\
q & 1 - q
\end{bmatrix}
\]

and the conditional entropy rate \( H_{\infty}(X | Y) \) is equal to the binary entropy function \( H(p) \) as shown in the following:

\[
H(X_1, \ldots, X_n | V_1, \ldots, V_n)
= H(Y_1 \oplus W_1, \ldots, Y_n \oplus W_n | Y_1, \ldots, Y_n)
= H(W_1, \ldots, W_n | Y_1, \ldots, Y_n)
= H(W_1, \ldots, W_n) = nH(W) = nH(p).
\]

Tables III–VI list simulation results for source pairs with different generating parameters \( p \) and \( q \). Parameter \( I \) (the number of the grammar levels) is set to \( \lceil \log \log n \rceil \). Recall that normalized grammar entropy is denoted by \( \frac{1}{n} H_G(x^n | y^n) \). Entropy and rates are expressed in terms of bits per letter.

For all the tables, as the length \( n \) of the sequences increases, we can clearly see the following two trends:

1) the compression rate decreases but is always above the conditional entropy rate \( H_{\infty}(X | Y) \) of a source pair;
2) the normalized grammar entropy \( \frac{1}{n} H_G(x^n | y^n) \) increases but is always below \( H_{\infty}(X | Y) \).

The first trend simply illustrates that the compression rate approaches from above the conditional entropy rate. Similarly, the second trend indicates the convergence of \( \frac{1}{n} H_G(x^n | y^n) \) to \( H_{\infty}(X | Y) \) and is due to the fact that pairs \( (e^n) \) are ignored in the calculation of the grammar entropy \( H_G(x^n | y^n) \).

REFERENCES


