Approximating the Smallest Grammar: Kolmogorov Complexity in Natural Models

Moses Charikar* Princeton University

Eric Lehman† MIT

Ding Liu* Princeton University

Rina Panigrahy‡ Cisco Systems

Manoj Prabhakaran* Princeton University

April Rasala† MIT

Amit Sahai* Princeton University

abhi shelat† MIT

Abstract

We consider the problem of finding the smallest context-free grammar that generates exactly one given string of length n. The size of this grammar is of theoretical interest as an efficiently computable variant of Kolmogorov complexity. The problem is of practical importance in areas such as data compression and pattern extraction.

The smallest grammar is known to be hard to approximate to within a constant factor, and an $o(\log n/\log\log n)$ approximation would require progress on a long-standing algebraic problem [10]. Previously, the best proved approximation ratio was $O(n^{1/2})$ for the BISECTION algorithm [8]. Our main result is an exponential improvement of this ratio; we give an $O(\log(n/g^*))$ approximation algorithm, where g^* is the size of the smallest grammar.

We then consider other computable variants of Kolomogorov complexity. In particular we give an $O(\log^2 n)$ approximation for the smallest non-deterministic finite automaton with *advice* that produces a given string. We also apply our techniques to "advice-grammars" and "edit-grammars", two other natural models of string complexity.

1. Introduction

This paper addresses the *smallest grammar problem*; namely, what is the smallest context-free grammar that generates exactly one given string σ? For example, the smallest context-free grammar generating the string:

$$\sigma = \underline{\texttt{a rose is a rose is a rose}}$$

is as follows:

$$\begin{array}{rcl} S & \rightarrow & BBA \\ A & \rightarrow & \underline{\texttt{a rose}} \\ B & \rightarrow & A\underline{\texttt{ is }} \end{array}$$

*e-mail: {moses,dingliu,mp,sahai}@cs.princeton.edu

†e-mail: {e_lehman,arasala,abhi}@mit.edu

‡e-mail: rinap@cisco.com

STOC'02, May 19-21, 2002, Montreal, Quebec, Canada.

The size of a grammar is defined to be the total number of symbols on the right sides of all rules. The smallest grammar is known to be hard to approximate within a small constant factor. Further, even for a very restricted class of strings, approximating the smallest grammar within a factor of $o\left(\frac{\log n}{\log\log n}\right)$ would require progress on a well-studied algebraic problem [10]. Previously, the best proven approximation ratio was $O(n^{1/2})$ for the BISECTION algorithm [8]. Our main result is an exponential improvement of this ratio.

Several rich lines of research connect this elegant problem to fields such as Kolmogorov complexity, data compression, pattern identification, and approximation algorithms for hierarchical problems.

The size of the smallest context-free grammar generating a given string is a natural, but more tractable variant of Kolmogorov complexity. (The Kolmogorov complexity of a string is the description length of the smallest Turing machine that outputs that string.) The Turing machine model for representing strings is too powerful to be exploited effectively; in general, the Kolmogorov complexity of a string is uncomputable. However, weakening the model from Turing machines to context-free grammars reduces the complexity of the problem from the realm of undecidability to mere intractability. How well the smallest grammar can be approximated remains an open question. In this work we make significant progress toward an answer: our main result is that the "grammar complexity" of a string is $O(\log(n/g^*))$-approximable in polynomial time, where g^* is the size of the smallest grammar.

This perspective on the smallest grammar problem suggests a general direction of inquiry: can the complexity of a string be determined or approximated with respect to other natural models? Along this line, we consider the problem of optimally representing an input string by a non-deterministic finite automaton guided by an advice string. We show that the optimal representation of a string is $O(\log^2 n)$-approximable in this model. Then we consider context-free grammars with multiple productions per nonterminal, again guided by an advice string. This modified grammar model turns out, unexpectedly, to be essentially equivalent to the original.

The smallest grammar problem is also important in the area of data compression. Instead of storing a long string, one can store a small grammar that generates it, provided such a grammar can be found. This line of thought has led to a flurry of research [8, 12, 16, 7, 1, 9] in the data compression community. Kieffer and Yang show that a good solution to the grammar problem leads to a good universal compression algorithm for finite Markov sources [7]. Empirical results indicate that grammar-based data compression is competitive with other techniques in practice [8, 12]. Surprisingly, however, the most prominent algorithms for the smallest grammar problem have been shown to exploit the grammar model poorly [10]. In particular, the SEQUITUR [12], BISECTION [8], LZ78 [16], and SEQUENTIAL [14] algorithms all have approximation ratios that are $\Omega(n^{1/3})$. In this paper, we achieve a logarithmic ratio.

The smallest grammar problem is also relevant to identifying important patterns in a string, since such patterns naturally correspond to nonterminals in a compact grammar. In fact, an original motivation for the problem was to identify regularities in DNA sequences [11]. Since then, smallest grammar algorithms have been used to highlight patterns in musical scores and uncover properties of language from example texts [4]. All this is possible because a string represented by a context-free grammar remains relatively comprehensible. This comprehensibility is an important attraction of grammar-based compression relative to otherwise competitive compression schemes. For example, the best pattern matching algorithm that operates on a string compressed as a grammar is asymptotically faster than the equivalent for LZ77 [6].

Finally, work on the smallest grammar problem extends the study of approximation algorithms to hierarchical objects, such as grammars, as opposed to "flat" objects, such as graphs, CNF-formulas, etc. This is a significant shift, since many real-world problems have a hierarchical nature, but standard approximation techniques such as linear and semi-definite programming are not easily applied to this new domain.

The remainder of this paper is organized as follows. Section 2 defines terms and notation. Section 3 contains our main result, an $O(\log(n/g^*))$ approximation algorithm for the smallest grammar problem. Section 4 follows up the Kolmogorov complexity connection and includes the results on the complexity measure in other models. We conclude with some open problems and new directions.

2. Definitions

A *grammar* $\mathcal{G}$ is a 4-tuple $(\Sigma, \Gamma, S, \Delta)$, where Σ is a set of *terminals*, Γ is a set of *nonterminals*, $S \in \Gamma$ is the *start symbol*, and Δ is a set of *rules* of the form $T \to (\Sigma \cup \Gamma)^*$. A *symbol* may be either a terminal or nonterminal. The size of a grammar is defined to be the total number of symbols on the right sides of all rules. Note that a grammar of size g can be readily encoded using $O(g \log g)$ bits, and so our measure is close to this natural alternative.[1]

Since $\mathcal{G}$ generates exactly one finite string, there is exactly one rule in Δ defining each nonterminal in Γ. Furthermore, $\mathcal{G}$ is acyclic; that is, there exists an ordering of the nonterminals in Γ such that each nonterminal precedes all nonterminals in its definition. The *expansion* of a string of symbols η is obtained by replacing each nonterminal in η by its definition until only terminals remain. The expansion of η is denoted $\langle\eta\rangle$, and the length of $\langle\eta\rangle$ is denoted $[\eta]$.

[1] More sophisticated ways to encode a grammar [7, 14] relate the encoding size in bits to the above definition of the size of a grammar in a closer manner, and give theoretical justification to this definition.

Throughout, symbols (terminals and nonterminals) are uppercase, and strings of symbols are lowercase Greek. We use σ for the input string, n for its length, g for the size of a grammar that generates σ, and g^* for the size of the smallest such grammar.

$|\mathcal{G}|$ denotes the size of grammar minus number of nonterminals. Without loss of generalily we do not allow unary rules of the form $A \to B$; then $|\mathcal{G}|$ is at least half the size of $\mathcal{G}$.

3. An $O(\log(n/g^*))$ Approximation Algorithm

We now give an $O(\log(n/g^*))$-approximation algorithm for the smallest grammar problem. The description is divided into three sections. First, we introduce a variant of the well-known LZ77 compression scheme. This serves two purposes: it gives a lower bound on the size of the smallest grammar for a string and is the starting point for our construction of a small grammar. Second, we introduce balanced binary grammars, a class of well-behaved grammars that our procedure employs. We also introduce three basic operations on balanced binary grammars. Finally, we present the main algorithm, which translates a string compressed using our LZ77 variant into a grammar at most $O(\log(n/g^*))$ times larger than the smallest.

3.1 LZ77

Our algorithm for approximating the smallest grammar employs a variant of the LZ77 compression scheme [15]. In this variant, a string σ is represented by a list of pairs $(p_1, l_1), \ldots, (p_s, l_s)$. Each pair (p_i, l_i) represents a string, and the concatenation of these strings is σ. In particular, if $p_i = 0$, then the pair represents the string l_i, which is a single terminal. If $p_i \neq 0$, then the pair represents a portion of the prefix of σ that is represented by the preceding $i-1$ pairs; namely, the l_i terminals beginning at position p_i in σ. Note that, unlike in the standard LZ77, we require that the substring referred to by (p_i, l_i) be fully contained within the prefix of σ generated by the previous pairs. The *size* of the LZ77 representation of a string is s, the number of pairs employed. The smallest LZ77 representation can be easily found using a greedy algorithm in polynomial time. More efficient algorithms are possible; for example, [5] gives an $O(n \log n)$ time algorithm for our variant of LZ77.

As observed in [12], a grammar $\mathcal{G}$ for a string can be converted in a simple way to an LZ77 representation. It is not hard to see that this transformation gives an LZ77 list of at most $|\mathcal{G}| + 1$ pairs. Thus we have:

LEMMA 1. *The size of the smallest grammar for a string is lower bounded by the size of the smallest* LZ77 *representation for that string.*

Our $O(\log(n/g^*))$-approximation algorithm essentially inverts this process, mapping an LZ77 sequence to a grammar. This direction is much more involved, because the strings represented by pairs can arbitrarily overlap each other whereas the rules in a grammar have less flexibility. Hence, one of our main technical contributions is an efficient procedure to perform this conversion.

3.2 Balanced Binary Grammars

Our approximation algorithm works exclusively with a restricted class of well-behaved grammars referred to as *balanced binary grammars.* A *binary rule* is a grammar rule with exactly two symbols on the right side. A *binary grammar* is a grammar in which every rule is binary. Two strings of symbols, β and γ, are *α-balanced* if

$$\frac{\alpha}{1-\alpha} \leq \frac{[\beta]}{[\gamma]} \leq \frac{1-\alpha}{\alpha}$$

for some constant α between 0 and $\frac{1}{2}$. Intuitively, α-balanced means "about the same length". Note that inverting the fraction $\frac{[\beta]}{[\gamma]}$ gives an equivalent condition. The condition above is also equivalent to saying that the length of the expansion of each string ($[\beta]$ and $[\gamma]$) is between an α and a $1-\alpha$ fraction of the length of the combined expansion ($[\beta\gamma]$). An *α-balanced rule* is a binary rule in which the two symbols on the right are α-balanced. An *α-balanced grammar* is a binary grammar in which every rule is α-balanced. For brevity, we usually shorten "α-balanced" to simply "balanced".

If $\mathcal{G}$ is a binary grammar, $|\mathcal{G}| = \frac{\text{size of } \mathcal{G}}{2}$ = number of non-terminals in $\mathcal{G}$. So in the rest of this section we shall consider the number of non-terminals of a binary grammar instead of its size.

The remainder of this section defines three basic operations on balanced binary grammars. Each operation adds a small number of rules to an existing balanced grammar to produce a new balanced grammar that has a nonterminal with specified properties. These operations are summarized as follows.

`AddPair`: Produces a balanced grammar containing a nonterminal with expansion $\langle XY \rangle$ from a balanced grammar containing symbols X and Y. The number of rules added to the original grammar is $O\left(1 + \left|\log \frac{[X]}{[Y]}\right|\right)$.

`AddSequence`: Produces a balanced grammar containing a nonterminal with expansion $\langle X_1 \ldots X_t \rangle$ from a balanced grammar containing symbols $X_1 \ldots X_t$. The number of rules added is $O\left(t\left(1 + \log \frac{[X_1 \ldots X_t]}{t}\right)\right)$.

`AddSubstring`: Produces a balanced grammar containing a nonterminal with expansion β from a balanced grammar containing a nonterminal with β as a substring of its expansion. Adds $O(\log |\beta|)$ new rules.

For these operations to work correctly, we require that α be selected from the limited range $0 < \alpha \leq 1 - \frac{1}{2}\sqrt{2}$, which is about 0.293. These three operations are detailed below.

3.2.1 The AddPair Operation

We are given a balanced grammar with symbols X and Y and want to create a balanced grammar containing a nonterminal with expansion $\langle XY \rangle$. Suppose that $[X] \leq [Y]$; the other case is symmetric.

First, we decompose Y into a string of symbols as follows. Initially, this string consists of the symbol Y itself. Thereafter, while the first symbol in the string is not in balance with X, we replace it by its definition. A routine calculation, which we omit, shows that balance is eventually achieved. At this point, we have a string of symbols $Y_1 \ldots Y_t$ with expansion $\langle Y \rangle$ such that Y_1 is in balance with X. Furthermore, note that $Y_1 \ldots Y_i$ is in balance with Y_{i+1} for all $1 \leq i < t$ by construction.

Now we create new rules. Initially, we create a new rule $Z_1 \rightarrow XY_1$ and declare this to be the *active rule*. The remainder of the operation is proceeds in steps. At the start of the i-th step, the active rule has the form $Z_i \rightarrow A_i B_i$, and the following three invariants hold:

1. $\langle Z_i \rangle = \langle XY_1 \ldots Y_i \rangle$
2. $\langle B_i \rangle$ is a substring of $\langle Y_1 \ldots Y_i \rangle$.
3. All rules in the grammar are balanced, including the active rule.

The relationships between strings implied by the first two invariants are indicated in the following diagram:

$$\overbrace{\underbrace{X\ Y_1\ Y_2\ \ldots}_{A_i}\ \underbrace{\ldots\ Y_{i-1}\ Y_i}_{B_i}}^{Z_i}\ Y_{i+1}\ \ldots\ Y_t$$

After t steps, the active rule defines a nonterminal Z_t with expansion $\langle XY_1 \ldots Y_t \rangle = \langle XY \rangle$ as desired.

The invariants stated above imply some inequalities that are needed later to show that the grammar remains in balance. Since $Y_1 \ldots Y_i$ is in balance with Y_{i+1}, we have:

$$\frac{\alpha}{1-\alpha} \leq \frac{[Y_{i+1}]}{[Y_1 \ldots Y_i]} \leq \frac{1-\alpha}{\alpha}$$

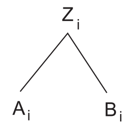
Since $\langle B_i \rangle$ is a substring of $\langle Y_1 \ldots Y_i \rangle$ by invariant 2, we can conclude:

$$\frac{\alpha}{1-\alpha} \leq \frac{[Y_{i+1}]}{[B_i]} \tag{1}$$

On the other hand, since $\langle Z_i \rangle$ is a superstring of $\langle Y_1 \ldots Y_i \rangle$ by invariant 1, we can conclude:

$$\frac{[Y_{i+1}]}{[Z_i]} \leq \frac{1-\alpha}{\alpha} \tag{2}$$

Below we describe how during a step a new active rule is created. For clarity, we supplement the text with diagrams. In these diagrams, a rule $Z_i \rightarrow A_i B_i$ is indicated with a wedge:

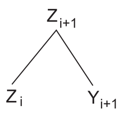


Preexisting rules are indicated with broken lines, and new rules with dark lines.

At the start of the i-th step, the active rule is $Z_i \rightarrow A_i B_i$. Our goal is to create a new active rule that defines Z_{i+1} while maintaining the three invariants. There are three cases to consider.

<u>Case 1</u>: If Z_i and Y_{i+1} are in balance, then we create a new rule:

$$Z_{i+1} \rightarrow Z_i Y_{i+1}$$



This becomes the active rule. It is easy to check that the three invariants are maintained.

If case 1 is bypassed, then Z_i and Y_{i+1} are not in balance; that is, the following assertion does not hold:

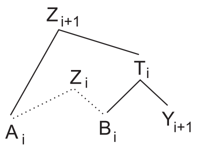
$$\frac{\alpha}{1-\alpha} \leq \frac{[Y_{i+1}]}{[Z_i]} \leq \frac{1-\alpha}{\alpha}$$

Since the right inequality is (2), the left inequality must be violated. Thus, hereafter we can assume:

$$\frac{\alpha}{1-\alpha} > \frac{[Y_{i+1}]}{[Z_i]} \tag{3}$$

<u>Case 2</u>: Otherwise, if A_i is in balance with B_iY_{i+1}, then we create two new rules:

$$\begin{aligned} Z_{i+1} &\rightarrow A_iT_i \\ T_i &\rightarrow B_iY_{i+1} \end{aligned}$$



The first of these becomes the active rule. It is easy to check that the first two invariants are maintained. But the third, which asserts that all new rules are balanced, requires some work. The rule $Z_{i+1} \rightarrow A_iT_i$ is balanced by the case assumption. What remains is to show that the rule $T_i \rightarrow B_iY_{i+1}$ is balanced; that is, we must show:

$$\frac{\alpha}{1-\alpha} \leq \frac{[Y_{i+1}]}{[B_i]} \leq \frac{1-\alpha}{\alpha}$$

The left inequality is (1). For the right inequality, begin with (3):

$$\begin{aligned} [Y_{i+1}] &< \frac{\alpha}{1-\alpha}[Z_i] \\ &= \frac{\alpha}{1-\alpha}([A_i]+[B_i]) \\ &\leq \frac{\alpha}{1-\alpha}\left(\frac{1-\alpha}{\alpha}[B_i]+[B_i]\right) \\ &\leq \frac{1-\alpha}{\alpha}[B_i] \end{aligned}$$

The equality follows from the definition of Z_i by the rule $Z_i \rightarrow A_iB_i$. The subsequent inequality uses the fact that this rule is balanced, according to invariant 3. The last inequality uses only algebra and holds for all $\alpha \leq 0.381$.

If case 2 is bypassed then A_i and B_iY_{i+1} are not in balance; that is, the following assertion is false:

$$\frac{\alpha}{1-\alpha} \leq \frac{[A_i]}{[B_iY_{i+1}]} \leq \frac{1-\alpha}{\alpha}$$

Since A_i is in balance with B_i alone by invariant 3, the right inequality holds. Therefore, the left inequality must not; hereafter, we can assume:

$$\frac{\alpha}{1-\alpha} > \frac{[A_i]}{[B_iY_{i+1}]} \tag{4}$$

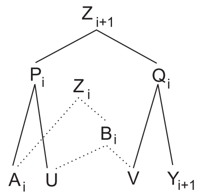
Combining inequalities (3) and (4), one can use algebraic manipulation to establish the following bounds, which hold hereafter:

$$\begin{aligned} \frac{[A_i]}{[B_i]} &\leq \frac{\alpha}{1-2\alpha} && (5) \\ \frac{[Y_{i+1}]}{[B_i]} &\leq \frac{\alpha}{1-2\alpha} && (6) \end{aligned}$$

Then $[B_i] > 1$, as say $[A_i] \geq 1 > \frac{\alpha}{1-2\alpha}$; i.e., B_i is not a terminal. So let B_i be defined by the rule $B_i \rightarrow UV$.

<u>Case 3</u>: If we bypass cases 1 and 2, we create three new rules:

$$\begin{aligned} Z_{i+1} &\rightarrow P_iQ_i \\ P_i &\rightarrow A_iU \\ Q_i &\rightarrow VY_{i+1} \end{aligned}$$



The first of these becomes the active rule. We must check that all of the new rules are in balance. We begin with $P_i \rightarrow A_iU$. In one direction, we have:

$$\begin{aligned} \frac{[A_i]}{[U]} &\geq \frac{[A_i]}{[B_i]} \\ &\geq \frac{\alpha}{1-\alpha} \end{aligned}$$

The first inequality follows as $[B_i] = [UV] > [U]$. The second inequality uses the fact that A_i and B_i are in balance. In the other direction, we have:

$$\begin{aligned} \frac{[A_i]}{[U]} &\leq \frac{[A_i]}{\alpha[B_i]} \\ &\leq \frac{1}{1-2\alpha} \\ &\leq \frac{1-\alpha}{\alpha} \end{aligned}$$

The first inequality uses the fact that $B_i \rightarrow UV$ is balanced, and the second follows from (5). The last inequality holds for all $\alpha < 0.293$.

Next, we show that $Q_i \rightarrow VY_{i+1}$ is balanced. In one direction, we have:

$$\begin{aligned} \frac{[Y_{i+1}]}{[V]} &\leq \frac{[Y_{i+1}]}{\alpha[B_i]} \\ &\leq \frac{1}{1-2\alpha} \\ &\leq \frac{1-\alpha}{\alpha} \end{aligned}$$

The first inequality uses the fact that $B_i \rightarrow UV$ is balanced, the second uses (6), and the third holds for all $\alpha < 0.293$. In the other direction, we have:

$$\begin{aligned} \frac{[Y_{i+1}]}{[V]} &\geq \frac{[Y_{i+1}]}{[B_i]} \\ &\geq \frac{\alpha}{1-\alpha} \end{aligned}$$

The first inequality follows as $[B_i] > [V]$, and the second is (1).

Finally, we must check that $Z_{i+1} \to P_i Q_i$ is in balance. In one direction, we have:

$$\begin{aligned}
\frac{[P_i]}{[Q_i]} &= \frac{[A_i U]}{[V Y_{i+1}]} \\
&\leq \frac{[A_i] + (1-\alpha)[B_i]}{\alpha[B_i] + [Y_{i+1}]} \\
&= \frac{\frac{[A_i]}{[B_i]} + (1-\alpha)}{\alpha + \frac{[Y_{i+1}]}{[B_i]}} \\
&\leq \frac{\frac{\alpha}{1-2\alpha} + (1-\alpha)}{\alpha + \frac{\alpha}{1-\alpha}} \\
&\leq \frac{1-\alpha}{\alpha}
\end{aligned}$$

The equality follows from the definitions of P_i and Q_i. The first inequality uses the fact that the rule $B_i \to UV$ is balanced. The subsequent equality follows by dividing the top and bottom by $[B_i]$. In the next step, we use (5) on the top, and (1) on the bottom. The final inequality holds for all $\alpha \leq \frac{1}{3}$. In the other direction, we have:

$$\begin{aligned}
\frac{[P_i]}{[Q_i]} &= \frac{[A_i U]}{[V Y_{i+1}]} \\
&\geq \frac{[A_i] + \alpha[B_i]}{(1-\alpha)[B_i] + [Y_{i+1}]} \\
&= \frac{\frac{[A_i]}{[B_i]} + \alpha}{(1-\alpha) + \frac{[Y_{i+1}]}{[B_i]}} \\
&\geq \frac{\frac{\alpha}{1-\alpha} + \alpha}{(1-\alpha) + \frac{\alpha}{1-2\alpha}} \\
&\geq \frac{\alpha}{1-\alpha}
\end{aligned}$$

As before, the first inequality uses the definitions of P_i and Q_i. Then we use the fact that $B_i \to UV$ is balanced. We obtain the second equality by dividing the top and bottom by $[B_i]$. The subsequent inequality uses the fact that A_i and B_i are in balance on the top and (6) on the bottom. The final inequality holds for all $\alpha \leq \frac{1}{3}$.

All that remains is to upper bound the number of rules created during the AddPair operation. At most three rules are added in each of the t steps. Therefore, it suffices to upper bound t. Recall that t is determined when Y is decomposed into a string of symbols: each time we expanded the first symbol in the string, the length of the expansion of the first symbol decreases by a factor of at least $1-\alpha$. When the first symbol is in balance with X, the process stops. Therefore, the number of steps is $O(\log([Y]/[X]))$. Since the string initially contains one symbol, t is $O(1+\log([Y]/[X]))$. Therefore, the number of new rules is $O\left(1 + \log([Y]/[X])\right)$. The case of $[X] > [Y]$ is symmetric, and the common bound for the two cases can be written as $O\left(1 + \left|\log \frac{[X]}{[Y]}\right|\right)$.

3.2.2 The AddSequence Operation

The AddSequence operation is a generalization of AddPair. We are given a balanced grammar with nonterminals $X_1 \ldots X_t$ and want to create a balanced grammar containing a nonterminal with expansion $\langle X_1 \ldots X_t \rangle$.

But we solve this problem in a much simpler manner, using the AddPair operation repeatedly. Consider placing the X_i at the leaves of a balanced binary tree. (To simplify the analysis, assume that t is a power of two.) We create a nonterminal for each internal node by combining the nonterminals at the child nodes using AddPair. Recall that the number of rules that AddPair creates when combining nonterminals X and Y is:

$$O\left(1 + \left|\log \frac{[X]}{[Y]}\right|\right) \quad = \quad O\left(1 + \log[X] + \log[Y]\right)$$

Let c denote the hidden constant on the right, and let s equal $[X_1 \ldots X_t]$. Creating all the nonterminals on the bottom level of the tree generates at most

$$ct/2 + c\sum_{i=1}^{t} \log[X_i] \quad \leq \quad ct/2 + ct \log \frac{s}{t}$$

rules. (The inequality follows from the concavity of log.) Similarly, the number of rules created on the second level of the tree is at most $ct/4 + c(t/2)\log\frac{s}{t/2}$, because we pair $t/2$ nonterminals, but the sum of their expansion lengths is still s. In general, on the i-th level, we create at most

$$ct/2^{i+1} + c(t/2^i)\log\frac{s}{t/2^i} \quad = \quad c(t/2^i)\log\frac{s}{t} + ct(i+\frac{1}{2})/2^i$$

new rules. Summing i from 0 to $\log t$, we find that the total number of rules created is at most

$$\sum_{i=0}^{\log t} c(t/2^i)\log\frac{s}{t} + ct(i+\frac{1}{2})/2^i \quad = \quad O\left(t\left(1 + \log\frac{s}{t}\right)\right)$$

where $s = [X_1 \ldots X_t]$.

3.2.3 The AddSubstring Operation

We are given a balanced grammar containing a nonterminal with β as a substring of its expansion. We want to create a balanced grammar containing a nonterminal with expansion exactly β.

Let T be the nonterminal with the shortest expansion such that its expansion contains β as a substring. Let $T \to XY$ be its definition. Then we can write $\beta = \beta_p \beta_s$, where the prefix β_p lies in $\langle X \rangle$ and the suffix β_s lies in $\langle Y \rangle$. (Confusingly, β_p is actually a suffix of $\langle X \rangle$, and β_s is a prefix of $\langle Y \rangle$.) We generate a nonterminal that expands to the prefix β_p, another that expands to the suffix β_s, and then merge the two with AddPair. The last step generates only $O(\log|\beta|)$ new rules. So all that remains is to generate a nonterminal that expands to the prefix, β_p; the suffix is handled symmetrically. This task is divided into two phases.

In the first phase, we find a sequence of nonterminals $X_1 \ldots X_t$ with expansion equal to β_p. To do this, we begin with an empty sequence and employ a recursive procedure. At each step, we have a *remaining suffix* (initially β_p) of some *current nonterminal* (initially X). During each step, we consider the definition of the current nonterminal, say $X \to AB$. There are two cases:

1. If the remaining suffix wholly contains $\langle B \rangle$, then we prepend B to the nonterminal sequence. The remaining suffix becomes the portion of the old suffix that overlaps $\langle A \rangle$, and the current nonterminal becomes A.

2. Otherwise, we keep the same remaining suffix, but the current nonterminal becomes B.

A nonterminal is only added to the sequence in case 1. But in that case, the length of the desired suffix is scaled down by at least a factor $1-\alpha$. Therefore the length of the resulting nonterminal sequence is $t = O(\log|\beta|)$.

This construction implies the following inequality, which we will need later:

$$\frac{[X_1 \dots X_i]}{[X_{i+1}]} \quad \leq \quad \frac{1-\alpha}{\alpha} \tag{7}$$

This inequality holds because $\langle X_1 \dots X_i \rangle$ is a suffix of the expansion of a nonterminal in balance with X_{i+1}. Consequently, $X_1 \dots X_i$ is not too long to be in balance with X_{i+1}.

In the second phase, we merge the nonterminals in the sequence $X_1 \dots X_t$ to obtain the nonterminal with expansion β_p. The process goes from left to right. Initially, we set $R_1 = X_1$. Thereafter, at the start of the i-th step, we have a nonterminal R_i with expansion $\langle X_1 \dots X_i \rangle$ and seek to merge in nonterminal X_{i+1}. There are two cases, distinguished by whether or not the following inequality holds:

$$\frac{\alpha}{1-\alpha} \quad \leq \quad \frac{[R_i]}{[X_{i+1}]}$$

- If so, then R_i and X_{i+1} are in balance. (Inequality (7) supplies the needed upper bound on $[R_i]/[X_{i+1}]$.) Therefore, we add the rule $R_{i+1} \rightarrow R_i X_{i+1}$.
- If not, then R_i is too small to be in balance with X_{i+1}. We use AddPair to merge the two, which generates $O(1 + \log([X_{i+1}]/[R_i]))$ new rules. Since $[X_{i+1}] \leq [R_{i+1}]$, the number of new rules is $O(1 + \log([R_{i+1}]/[R_i]))$.

Summing the number of rules created in this process gives:

$$\begin{aligned} \sum_{i=1}^{t} O\left(1 + \log \frac{[R_{i+1}]}{[R_i]}\right) &= O(t + \log[R_t]) \\ &= O(\log|\beta|) \end{aligned}$$

The second bound follows from the fact, observed previously, that $t = O(\log|\beta|)$ and from the fact that $\langle R_t \rangle = \beta_p$. Generating a nonterminal for the suffix β_s requires $O(\log|\beta|)$ rules as well. Therefore, the total number of new rules is $O(\log|\beta|)$ as claimed.

3.3 The Algorithm

We now combine all the tools of the preceding two Sections to obtain an $O(\log(n/g^*))$-approximation algorithm for the smallest grammar problem.

First, we apply the LZ77 variant described in Section 3.1. This gives a sequence of pairs $(p_1, l_1) \dots (p_s, l_s)$. By Lemma 1, the length of this sequence is a lower bound on the size of the smallest grammar for σ; that is, $s \leq g^*$. Now we employ the tools of Section 3.2 to translate this sequence to a grammar. We work through the sequence from left to right, building up a balanced binary grammar as we go. Throughout, we maintain an *active list* of grammar symbols.

Initially, the active list is l_1, which must be a single character since $p_1 = 0$ necessarily. In general, at the beginning of the i-th step, the expansion of the active list is the string represented by $(p_1, l_1) \dots (p_i, l_i)$. Our goal for the step is to augment the grammar and alter the active list so that the expansion of the symbols in the active list is the string represented by $(p_1, l_1) \dots (p_{i+1}, l_{i+1})$.

If $p_{i+1} = 0$, we can accomplish this goal by simply appending l_{i+1} to the active list. If $p_{i+1} \neq 0$, then it specifies a substring β_i of the expansion of the active list. If β_i is contained in the expansion of a single symbol in the active list, then we use AddSubstring to create a nonterminal with expansion β_i using $O(\log|\beta_i|)$ rules. This nonterminal is then appended to the active list.

On the other hand, if β_i is not contained in the expansion of a single symbol in the active list, then it is the concatenation of a suffix of $\langle X \rangle$, all of $\langle A_1 \dots A_{t_i} \rangle$, and a prefix of $\langle Y \rangle$, where $X A_1 \dots A_{t_i} Y$ are consecutive symbols in the active list. We then perform the following operations:

1. Construct a nonterminal M with expansion $\langle A_1 \dots A_{t_i} \rangle$ using AddSequence. The number of new rules produced is $O(t_i(1 + \log(|\beta_i|/t_i)))$.
2. Replace $A_1 \dots A_{t_i}$ in the active list by the single symbol M.
3. Construct a nonterminal X' with expansion equal to the prefix of β_i in $\langle X \rangle$ using AddSubstring. Similarly, construct a nonterminal Y' with expansion equal to the suffix of β_i in $\langle Y \rangle$ using AddSubstring. This produces $O(\log|\beta_i|)$ new rules in total.
4. Create a nonterminal N with expansion $\langle X' M Y' \rangle$ using AddSequence on X', M, and Y'. This creates $O(\log|\beta_i|)$ new rules. Append N to the end of the active list.

Thus, in total, we add $O(t_i + t_i \log(|\beta_i|/t_i) + \log|\beta_i|)$ new rules during each step. The total number of rules created is:

$$\begin{aligned} & O\left(\sum_{i=1}^{s} t_i + t_i \log(|\beta_i|/t_i) + \log|\beta_i|\right) \\ &= O\left(\sum_{i=1}^{s} t_i + \sum_{i=1}^{s} t_i \log(|\beta_i|/t_i) + \sum_{i=1}^{s} \log|\beta_i|\right) \end{aligned}$$

The first sum is upper bounded by the total number of symbols inserted into the active list. This is at most two per step (M and N), which implies $\sum_{i=1}^{s} t_i \leq 2s$.

To upper bound the second sum, we use the concavity inequality:

$$\sum_{i=1}^{s} a_i \log b_i \quad \leq \quad \left(\sum_{i=1}^{s} a_i\right) \log\left(\frac{\sum_{i=1}^{s} a_i b_i}{\sum_{i=1}^{s} a_i}\right)$$

and set $a_i = t_i$, $b_i = |\beta_i|/t_i$ to give:

$$\begin{aligned} \sum_{i=1}^{s} t_i \log \frac{|\beta_i|}{t_i} &\leq \left(\sum_{i=1}^{s} t_i\right) \log\left(\frac{\sum_{i=1}^{s} |\beta_i|}{\sum_{i=1}^{s} t_i}\right) \\ &= O\left(s \log\left(\frac{n}{s}\right)\right) \end{aligned}$$

The latter inequality uses the fact that $\sum_{i=1}^{s} |\beta_i| \leq n$ and that $\sum_{i=1}^{s} t_i \leq 2s$. Note that the function $x \log(n/x)$ is increasing for x up to n/e, and so this inequality holds only if $2s \leq n/e$. This condition is violated only when input string (length n) turns out to be only a small factor ($2e$) longer than the LZ77 sequence (length s). If we detect this special case, then we can output the trivial grammar $S \to \sigma$ and achieve a constant approximation ratio.

By concavity again, the third sum is upper bounded by $s \log \frac{\sum |\beta_i|}{s} \leq s \log \frac{n}{s}$ and thus total grammar size is at most:

$$O\left(s \log \frac{n}{s}\right) \quad = \quad O\left(g^* \log \frac{n}{g^*}\right)$$

where we use the inequality $s \leq g^*$ and, again, the observation that $x \log(n/x)$ is increasing for $x < n/e$. The preceeding analysis establishes the following theorem.

THEOREM 1. *The optimum grammar for a string can be approximated efficiently within a factor of $O(\log(n/g^*))$, where n is the length of the string and g^* is the size of the smallest grammar.*

Remark: *Universal source codes* are of great interest in information theory. Grammar-based codes have been shown to be universal provided there is a *good* grammar transform and assuming a finite alphabet [7, 14]. [7] defines a grammar-transform to be *asymptotically compact* if for every string σ, it produces a grammar $\mathcal{G}_\sigma$ such that $|\mathcal{G}_\sigma|/|\sigma| = o(1)$, as $|\sigma|$ goes to infinity. Though not the motivation for our work, we note that our algorithm is asymptotically compact. To see this, note that for a finite alphabet, the number of pairs in the LZ77 representation of σ is $g = O(\frac{n}{\log n})$, where $n = |\sigma|$ (e.g. Lemma 12.10.1, [3]). So $g \log(n/g) = O(n \frac{\log \log n}{\log n})$ and hence $|\mathcal{G}_\sigma|/n = O(\frac{\log \log n}{\log n}) = o(1)$. It then follows from Theorem 7 in [7] that there is a source code based on our grammar transform which is universal with a *maximal pointwise redundancy* of $O(\frac{(\log \log n)^2}{\log n})$.[2]

4. Kolmogorov complexity in related models

The complexity of a string is a natural question of theoretical importance. Kolmogorov complexity is an elegant way to define this notion. It is the description length of the smallest Universal Turing machine program which prints the string. But it is well-known that this is an uncomputable measure of complexity. Thus we are led to the idea of defining a *computable* measure of the complexity of strings. A natural approach to defining such a measure is to restrict the computation model under which the string is generated.

We take the most natural candidate restrictions: instead of Turing machines we restrict ourselves to non-deterministic finite automata and context-free grammars. We make precise the notion of generating strings under such a model later. Then we ask the question of whether the string complexity under these models can be efficiently estimated. Although the problems are known to be NP-hard, we show that efficient algorithms with good approximation ratios exist.

The grammar model, which forms the focus of this work, turns out to be closely related to both the models mentioned above. The problem cast in the NFA model reduces to a two-level grammar model, for which we develop an $O(\log^2 n)$ approximation algorithm. In the grammar model above we used a CFG which generates a language containing a single string to define the complexity of the string. Another natural possibility, which we call the advice-grammar model, encodes an input string with a CFG (which can potentially generate an infinite number of strings) and an advice string. The advice string specifies how the production rules of the CFG should be applied such that only the specified string is produced. We show that the complexities in these two models are equivalent within a constant factor. Hence our main result on grammar approximation implies that we can approximate the CFG-based complexity of a string (either definition) within a logarithmic factor.

4.1 NFA-complexity of a string

A Non-deterministic Finite Automaton (NFA) is specified by a set of states and a transition function between the states. To associate an NFA with a string we consider it as a Mealy machine, which outputs a *string* on each transition. An NFA is said to *generate* the string that it outputs during an entire sequence of transitions from the start state to a final state. Thus an NFA can generate many strings, and we have to provide an *advice* to specify which transition to take at each point when multiple transitions are possible. To measure the complexity of the input string, we count both the NFA cost, which is the total length of the strings appearing on the transition edges, and the advice cost which is simply the number of advices required. Having fixed this measure of complexity, it is easy to see that without loss of generality we can have just one state (apart from a final state) and all the transitions are from this state to itself.[3] This is identical to a *two-level* restriction of our grammar model- with one rule (for the start symbol) corresponding to the advice string, and one rule for each transition.

In a *two-level grammar*, only the definition of the start symbol may contain non-terminals; the definitions of all other non-terminals contain only terminals. The problem is to find the smallest two-level grammar for a string. This problem resembles the optimal dictionary problem in which the goal is to find the best possible dictionary of codewords used to compress the input string. Storer [13] shows that this problem is NP-hard. However, several practical algorithms have been developed to address this problem. In this section, we outline an algorithm with a $O(\log^2 n)$ approximation ratio.

THEOREM 2. *The 2-level grammar compression problem is $O(\log^2 n)$-approximable.*

PROOF. The main idea is to recast the two-level grammar compression problem as a disjoint set cover problem. Given a string σ, define the ground set to consist of the n positions in the input string σ such that each position of σ contributes a distinct element. For every collection of non-overlapping instances of a substring η of σ, define a covering set consisting of all the positions contained in those instances. For

[2]In fact by Theorem 8 in [7], by modifying the transform to give an *irreducible* grammar a better redundancy of $O(\frac{\log \log n}{\log n})$ can be achieved. One way to modify the algorithm is to post-process the grammar in a straight-forward way by applying the Reduction Rules in [7] (Section VI). This can be done in polynomial time.

[3]Note that this is possible only because we charge unit cost for each advice, which is unjustified if we consider the bit representation of the advice. But this affects the complexity measure by at most a factor of logarithm of the number of different transitions of the NFA.

example, if the string abc appears k times without overlap, then we generate $2^k - 1$ covering sets for this substring. The cost of a covering set with string η and some t non-overlapping instances is the length of η plus t. In addition, for each character in σ, we define a covering set with cost one that contains only that character. It is not hard to see that this reduces the smallest two-level grammar problem to the problem of covering the ground set with a minimum cost collection of *disjoint* covering sets.

If we relax the condition that the cover consist of disjoint sets, then the cost of the optimum cover can only decrease. Furthermore, we can obtain an overlapping cover that has cost at most $1+\ln n$ times the optimum using the well-known greedy algorithm for set-cover [2].

This overlapping cover is converted to a two-level grammar as follows. First we modify the cover so that no instance of a string in any covering set is redundant for achieving the cover, by simply removing the redundant instances systematically. This ensures that a disjoint covering of σ can be obtained from the instances in the covering sets by taking at most one substring of each instance. Now consider a covering set (in the modified cover) consisting of instances of a substring η. We define a non-terminal for η, non-terminals for the halves of η, quarters of η and so forth until η is partitioned into single characters. The total length of all these definitions generated by η is at most $\log n$ times the length of η. Now any substring of η is expressible using a sequence of at most $\log n$ of these non-terminals. In particular, for each instance of η, we can express the portion of that instance not covered by previously-selected sets. Thus for each instance we add at most $\log n$ symbols to the rule of the start-symbol (and to the size of the grammar). Since we incur an approximation factor of $\log n$ in approximating the cover, the resulting grammar is at most $O(\log^2 n)$ larger than the optimum.

All that remains is to describe how the greedy set cover algorithm is applied. Potentially, there could be exponentially many covering sets. To overcome this, at each step we consider every distinct substring η of σ. There are at most n^2 substrings. For each i from 1 to $|\sigma|/|\eta|$, we use dynamic programming to find the collection of i non-overlapping instances of η that cover the greatest number of previously uncovered characters in the ground set. Thus, for each η and i we narrow the search to one candidate covering set. From among the candidate covering sets, we select the one which maximizes the number of new elements covered divided by $(|\eta| + i)$. $\square$

4.2 Advice-Grammar

Instead of restricting the grammar to produce exactly one string one could think of a general CFG and a program or advice-string for expanding the CFG such that exactly one string is produced. In a general CFG a single non-terminal may have many production rules and cyclic dependencies are also allowed. With at most a constant factor increase in the size of the CFG, we restrict the non-terminals to be of one of the two kinds: concatenation non-terminals, with exactly one binary production, or choice non-terminals with rules of the form $A \to X_0|X_1|\ldots|X_{t_A}$, where the X_i are any symbols. We call such a CFG a *canonical CFG*.

Advice-grammar $\mathcal{C}^{\mathcal{A}}$ consists of a CFG $\mathcal{C}$ and an *advice-string* of integers, $\mathcal{A}$. To expand the advice-grammar, one starts off expanding the CFG depth-first, but on encountering a choice non-terminal the *next* integer from the advice-string is read and used to choose the production to be used. We define the complexity of the advice-grammar as $|\mathcal{C}^{\mathcal{A}}| = |\mathcal{C}| + |\mathcal{A}|$, where $|\mathcal{C}|$ is the number of non-terminals in $\mathcal{C}$ and $|\mathcal{A}|$ is the length of the advice-string $\mathcal{A}$.

Remarkably, as we show now, this seemingly more powerful model is not more than a constant factor more efficient than the grammar we have been considering.

Define the partial expansion of a concatenation non-terminal as an expansion till terminals or choice non-terminals. Note that there cannot be any production loop without a choice non-terminal involved, and so the partial expansion gives rise to a finite tree, called the *partial expansion tree* (PET for short), with terminals and choice non-terminals as the leaves (the latter will be referred to as the choice leaves of the PET). Two non-terminals are said to be equivalent if they have the same partial expansion. A canonical CFG $\mathcal{G}'$ is a *transform* of a canonical CFG $\mathcal{G}$ if they have the same terminals and choice non-terminals and for every concatenation non-terminal in $\mathcal{G}$ there is an equivalent non-terminal in $\mathcal{G}'$, and the choice rules in $\mathcal{G}'$ are derived from that in $\mathcal{G}$ by replacing any concatenation non-terminal by its equivalent in $\mathcal{G}'$.

LEMMA 2. *Suppose $\mathcal{C}$ is a canonical CFG with $|\mathcal{C}| = g$. Then we can construct a transform of $\mathcal{C}$, $\mathcal{C}'$ such that $|\mathcal{C}'| = O(g)$ and every concatenation non-terminal A in $\mathcal{C}'$ satisfies the following:*

- *If more than one choice non-terminal appears in the partial expansion of A, then $A \to XY$ where both X and Y have choice non-terminals in their partial expansions.*
- *If A has only one choice non-terminal in its partial expansion, then the corresponding choice leaf appears in the PET of A at a depth of at most 2.*

Then, in A's PET, the number of nodes in the spanning subtree containing the root and all the choice leaves in it is at most a constant times the number of choice leaves.

In the appendix we prove the above Lemma and use it to prove the following:

THEOREM 3. *Suppose $\mathcal{C}^{\mathcal{A}}$ is an advice-grammar, with $|\mathcal{C}| = g$ and $|\mathcal{A}| = a$. Then there exists a grammar $\mathcal{G}$ with $|\mathcal{G}| = O(g + a)$, producing the same string as $\mathcal{C}^{\mathcal{A}}$.*

4.3 Edit Grammars

The fact that both the natural variants of CFGs are equivalent suggests the robustness of grammar based string complexity. We further explore the robustness of the model, by allowing *edit-rules* into the grammar. We show that this extended model can affect the complexity of the string by at most a logarithmic factor. Further our algorithm gives a logarithmic approximation under this model too.

An *edit-rule* is a rule of the form $A \to X[editop]$, where X is a symbol (which might in turn be a non-terminal defined by an edit-rule), and *editop* is a single edit operation; we restrict ourselves to the following three kinds of edit operations: insert, delete and replace a character. The edit operation specifies the position in $\langle X \rangle$ where the edit is to be performed, and in case of insert and replace, the character to be introduced.[4] The semantics of the edit operation is the natural one: $\langle A \rangle$ is obtained by performing *editop* on $\langle X \rangle$. An edit rule incurs unit cost. An edit-grammar is called binary or balanced depending on the non-edit (concatenation) rules in it.

We show that edit-grammars are well-approximated by usual grammars. For this we introduce a representation of the string called *edit-LZ77*. In the edit-LZ77 representation of a string, instead of pairs as in LZ77, we have triplets $(p_i, l_i, [editlist])$. *editlist* consists of a list of primitive operations (insert/delete/replace) to be applied to the sub-string generated by the (p_i, l_i) part of the triplet, one after the other. Each primitive operation will point to a location in the sub-string (which might have already been modified by the preceding operations in the list) and prescribe an insert, delete or replace operation. The *edit-cost* of a triplet is the number of edit operations in the triplet's list.

Along the lines of Lemma 1 we get the following lemma.

LEMMA 3. *If $\mathcal{G}$ is a binary edit-grammar with g concatenation rules and k edit rules, then there is an edit-LZ77 representation $\mathcal{L}$ of the string generated by $\mathcal{G}$, with at most $g+1$ triplets and a total edit cost of k.*

We observe that edit-LZ77 is approximated within a constant factor by the usual LZ77.

LEMMA 4. *If $\mathcal{L}$ is an edit-LZ77 list for a string σ, with t triplets and a total edit cost of k, then there is an LZ77 list $\mathcal{L}'$ for σ with at most $t+2k$ pairs.*

PROOF. We can replace a triplet $(p_i, l_i, [editlist])$ where *editlist* has k_i operations, by at most $1+2k_i$ pairs: at most $1+k_i$ pairs to refer to the sub-strings into which the k_i edit operations split the sub-string generated by the triplet, and at most k_i pairs to represent the inserts and replaces. ■

From the above two lemmas and the procedure from Section 3.3 we conclude the following.

THEOREM 4. *The optimum edit-grammar for a string can be approximated efficiently within a factor of $O(\log n)$ by a grammar, where n is the length of the string.*

5. Future Work

The smallest grammar problem has a theoretically interesting connection to Kolmogorov complexity, practical relevance in the areas of data compression and pattern extraction, and a hierarchical structure that is evident in other real-world problems.

One line of research leading from this problem is the study of string complexity with respect to other natural models. For example, the grammar model could be extended to allow a non-terminal to take a parameter. One could then write a rule such as $T(P) \to PP$, and write the string $xxyzyz$ as $T(x)T(yz)$. Presumably as model power increases, approximability decays to uncomputability.

[4]It is sometimes reasonable to define other edit operations. For instance one could allow a prefix/suffix operation. In this case the resulting edit-grammar becomes as powerful as LZ77 representation, within a constant factor. Or, one could allow insert/replace to introduce arbitrary symbols instead of terminals, which doesn't change the model significantly.

The $O(\log \frac{n}{g})$-approximation algorithm presented here runs in near-linear time and hence can be used in a potentially practical grammar-based compression algorithm. Empirical study is required to determine its effectiveness compared to other compressors. On the theoretical front, it would be interesting to explore new probabilistic models of sources for which it performs well.

Also, while an $o\left(\frac{\log n}{\log \log n}\right)$ approximation algorithm for the smallest grammar problem would require progress on the well-studied addition chain problem, it is only known that the optimal grammar cannot be approximated to within a small constant unless P = NP. Thus, nailing down the actual hardness of the smallest grammar problem remains an intriguing open problem.

Finally, many other important real-world problems share the hierarchical structure of the smallest grammar problem. For example, suppose one must design a digital circuit that computes the logical AND of various, specified subsets of the input signals. How many two-input AND gates suffice? This amounts to a variant of the smallest grammar problem where non-terminals represent sets rather than strings. We know of neither an approximation algorithm nor hardness of approximation result for this natural question.

Acknowledgments

We sincerely thank Yevgeniy Dodis, Martin Farach-Colton, Michael Mitzenmacher, Madhu Sudan and the anonymous reviewers.

6. REFERENCES

[1] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In J. A. Storer and M. Cohn, editors, *Data Compression Conference*, pages 119–128, Snowbird, Utah, 1998.

[2] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[3] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, New York, NY, USA, 1991.

[4] C. de Marcken. *Unsupervised Language Acquisition*. PhD thesis, Massachusetts Institute of Technology, 1996.

[5] M. Farach-Colton. Personal communication.

[6] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *SPIRE/CRIWG*, pages 89–96, 1999.

[7] J. C. Kieffer and E. hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[8] J. C. Kieffer, E. hui Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(4):1227–1245, 2000.

[9] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305, 1999.

[10] E. Lehman and A. Shelat. Approximations algorithms for grammar-based compression. In *Thirteenth Annual*

Symposium on Discrete Algorithms (SODA'02), 2002.

[11] C. Nevill-Manning. *Inferring Sequential Structure.* PhD thesis, University of Waikato, 1996.

[12] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[13] J. A. Storer. *Data Compression: Methods and Complexity Issues.* PhD thesis, Princeton University, 1979.

[14] E. h. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform–part one: Without context models. *IEEE Transactions on Information Theory*, 46(3):755–777, 2000.

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

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

Appendix

A.1 Proof of Theorem 3

First we prove Lemma 2.

PROOF of LEMMA 2. Non-terminals whose partial expansions contain only terminals are copied as such into $\mathcal{C}'$. We describe how to introduce equivalent non-terminals in $\mathcal{C}'$ for each of the remaining concatenation non-terminal in $\mathcal{C}$ by adding at most a constant number of (concatenation) rules. We *process* the remaining concatenation non-terminals in $\mathcal{C}$ (which contain at least one choice non-terminal in their partial expansion) in increasing order of the size of (number of symbols in) their partial expansions. For all non-terminals A that we process we produce an equivalent non-terminal A' in $\mathcal{C}'$ ensuring the following (in addition to the condition in the lemma): *In the PET of A', the depth of both the leftmost choice leaf and the rightmost choice leaf is at most 3.*

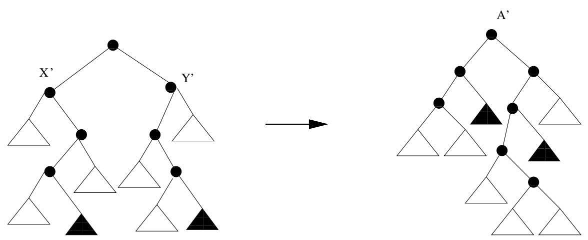


Figure 1: *Making a new rule in the proof of Lemma 2. The shaded sub-trees are the choice leaves of interest.*

We proceed inductively. Consider processing $A \to XY$. If neither X nor Y had to be processed before, we just set $A' = A$ and add the rule $A' \to XY$ (this is the base case). If X (resp Y) is a concatenation non-terminal in $\mathcal{C}'$ we would have already processed it and added an equivalent non-terminal X' (resp Y') to $\mathcal{C}'$. Consider simply making a concatenation rule $X'Y'$. This may have the (at most) two choice leaves of interest- the leftmost and the rightmost ones- at a depth of at most four in the resulting PET. Figure 1 shows a general scenario, with a maximum of eight possible sub-trees, two of which are the leaves of interest. We can arrange the (at most) eight sub-trees into a tree such that the leaves are at depth three or higher, and they fall on either side of the root. This can be done by making at most 7 new rules, the one at the root being A'. If there is only one leaf of interest, it can be pushed up to depth 2. When we finish processing the concatenation non-terminals in $\mathcal{C}$, we complete the construction of $\mathcal{C}'$ by adding rules for each of the choice non-terminals from $\mathcal{C}$ (replacing any concatenation non-terminal appearing in the RHS by equivalent non-terminals). ■

PROOF of THEOREM 3. By Lemma 2 we assume w.l.o.g that $\mathcal{C}$ satisfies the condition in the lemma, with $|\mathcal{C}| = O(g)$. A symbol is called choice-free if its partial expansion has only terminals. First we add all rules for choice-free non-terminals in $\mathcal{C}$ to $\mathcal{G}$. There are $O(g)$ such rules. Figure A.1 gives a recursive algorithm to add other rules to $\mathcal{G}$; we call this algorithm on the start symbol of $\mathcal{C}$.

Procedure DE-ADVICE(X)
{ i is a *global* variable, initialized to 0}
if the symbol X is choice-free **then**
  return X; { No new rule is produced}
else if X is a choice non-terminal **then**
  Let $X \to X_0|X_1|\dots|X_{t_X}$
  $b = \mathcal{A}_i$
  $i = i + 1$
  return DE-ADVICE(X_b) { No new rule is produced}
else
  Let $X \to YZ$
  $M :=$ DE-ADVICE(Y)
  $P :=$ DE-ADVICE(Z)
  Produce new rule $N \to MP$ {a new non-terminal is created here}
  return N

It is easy to see that in the resulting grammar $\mathcal{G}$, the non-terminal returned by the top-level call to DE-ADVICE expands to the string produced by $\mathcal{C}^{\mathcal{A}}$. We claim that the number of rules added to $\mathcal{G}$ by this call is $O(a)$. Note that new rules are produced at nodes in $\mathcal{C}$ which have at least one choice leaf below it in the PET of the last choice non-terminal occurring in the recursion; that is, the nodes at which new rules are produced are exactly the ones in the spanning sub-tree of a PET which spans the root of the PET and the choice leaves. At each choice leaf in each occurrence of a PET in the recursion, an entry of the advice string is consumed, and hence the total number of choice points is a. So by Lemma 2 the number of rules added by DE-ADVICE is $O(a)$. So the total number of rules in $\mathcal{G}$ is $O(g + a)$. ■