Approximation Algorithms for Grammar-Based Data Compression

by

Eric Lehman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2002

Author ....

Department of Electrical Engineering and Computer Science

February 1, 2002

Certified by..

Madhu Sudan

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by ....

Arthur C. Smith

Chairman, Department Committee on Graduate Students

Approximation Algorithms for Grammar-Based Data Compression

by

Eric Lehman

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2002, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis considers the *smallest grammar problem*: find the smallest context-free grammar that generates exactly one given string. We show that this problem is intractable, and so our objective is to find approximation algorithms. This simple question is connected to many areas of research. Most importantly, there is a link to data compression; instead of storing a long string, one can store a small grammar that generates it. A small grammar for a string also naturally brings out underlying patterns, a fact that is useful, for example, in DNA analysis. Moreover, the size of the smallest context-free grammar generating a string can be regarded as a computable relaxation of Kolmogorov complexity. Finally, work on the smallest grammar problem qualitatively extends the study of approximation algorithms to hierarchically-structured objects. In this thesis, we establish hardness results, evaluate several previously proposed algorithms, and then present new procedures with much stronger approximation guarantees.

Thesis Supervisor: Madhu Sudan
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis is the product of collaboration with Abhi Shelat and April Rasala at MIT as well as Amit Sahai, Moses Charikar, Manoj Prabhakaran, and Ding Liu at Princeton. The problem addressed here was proposed by Yevgeniy Dodis and Amit Sahai. My readers, Piotr Indyk and Dan Spielman, provided helpful, low-key advice. During the final writeup, April Rasala suggested dozens of fixes and improvements and even agreed to marry me to keep my spirits up. The entire project was deftly overseen by my advisor, Madhu Sudan. During my long haul through graduate school, Be Blackburn made the lab a welcoming place for me, as she has for countless others. Many professors offered support and advice at key times, including Tom Leighton, Albert Meyer, Charles Leiserson, and Michel Goemans. My mother taught me to count. My warmest thanks to them all.

In memory of Danny Lewin.

May 14, 1970 - September 11, 2001

Contents

1 Introduction **11**

1.1 Motivations 12

1.1.1 Data Compression 12

1.1.2 Complexity 13

1.1.3 Pattern Recognition 13

1.1.4 Hierarchical Approximation 14

1.2 Previous Work 14

1.3 Summary of Our Contributions 17

1.4 Organization 18

2 Preliminaries **19**

2.1 Definitions and Notation 19

2.2 Basic Observations 22

3 Hardness **27**

3.1 NP-Hardness 27

3.2 Hardness via Addition Chains 30

3.3 Background on Addition Chains 34

3.4 An Observation on Hardness 36

4 Analysis of Previous Algorithms **37**

4.1 Compression Versus Approximation 37

4.2 LZ78 39

4.3 BISECTION 45
4.4 SEQUENTIAL 47
4.5 Global Algorithms 57
4.5.1 LONGEST MATCH 63
4.5.2 GREEDY 69
4.5.3 RE-PAIR 72

5 New Algorithms **77**

5.1 An $O(\log^3 n)$ Approximation Algorithm 77
5.1.1 Preliminaries 77
5.1.2 The Algorithm 80
5.1.3 Aside: The Substring Problem 83
5.2 An $O(\log n/m^*)$-Approximation Algorithm 84
5.2.1 An LZ77 Variant 84
5.2.2 Balanced Binary Grammars 86
5.2.3 The Algorithm 99
5.3 Grammar-Based Compression versus LZ77 102

6 Future Directions **105**

Chapter 1

Introduction

This thesis 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:

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

is as follows:

$$\begin{aligned} S &\rightarrow BBA \\ A &\rightarrow \underline{\text{a rose}} \\ B &\rightarrow A\underline{\text{ is }} \end{aligned}$$

The size of a grammar is defined to be the total number of symbols on the right sides of all rules. Thus, in the example, the grammar has size 14. The decision version of this problem is NP-complete, so our objective is to find an approximation algorithm; that is, an algorithm which finds a grammar that generates the given string and is not much larger than the smallest such grammar.

1.1 Motivations

This elegant problem has considerable interest in its own right. It is a natural question about a standard object in theoretical computer science (context-free grammars) that can be posed in a single sentence. But, perhaps by virtue of this simplicity, there are also interesting connections to diverse areas: data compression, Kolmogorov complexity, pattern identification, and approximation algorithms for hierarchical problems. These connections are discussed below.

1.1.1 Data Compression

The smallest grammar problem has attracted interest particularly in the data compression community. The connection is direct: instead of storing a long string, one can store a small grammar that generates it, provided one can be found. The original string can be easily reconstructed from the grammar when needed. Many data compression procedures use this idea, and therefore amount to approximation algorithms for the smallest grammar problem [40, 37, 18, 16, 17, 27, 24, 23, 10]. Most of these procedures are analyzed in detail in Chapter 4.

Empirical results indicate that the grammar-based approach to compression is competitive with other techniques in practice [16, 10, 27, 24, 3, 1], and some grammar-based compressors are known to be asymptotically optimal on input strings generated by finite-state sources. But in Chapter 4 we show that, surprisingly, many of the best-known compressors of this type can fail dramatically; that is, there exist input strings generated by small grammars for which these compressors produce large grammars. Consequently, they turn out not to be very effective approximation algorithms for the smallest grammar problem.

There are compelling reasons why a good compressor might not be a good approximation algorithm and vice-versa. We delve into this issue in Section 4.1.

1.1.2 Complexity

The size of the smallest context-free grammar generating a given string is a natural, but more tractable variant of Kolmogorov complexity [20]. The Kolmogorov complexity of a string σ is the length of the shortest pair $\langle M, x\rangle$ where M is a Turing machine description, x is a string, and M outputs σ on input x. This 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 string representation model from Turing machines to context-free grammars reduces the complexity of the problem from the realm of undecidability to mere intractability. Moreover, we show that one can efficiently approximate the "grammar complexity" of a string. This perspective is not new. Indeed, the well-known compression algorithms due to Lempel and Ziv [39, 40] were an outgrowth of their earlier efforts to find a tractable alternative to Kolmogorov complexity [25]. However, one might argue that the models they adopted for string representation are considerably less natural than context-free grammars.

1.1.3 Pattern Recognition

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 and continuing motivation for work on the problem was to identify regularities in DNA sequences [27, 23]. (Interestingly, [23] espouses the goal of determining the entropy of DNA. This amounts to upper bounding the Kolmogorov complexity of a human being.) In addition, smallest grammar algorithms have been used to highlight patterns in musical scores [29] and uncover properties of language from example texts [10]. 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 the well-known LZ77 compression format [14].

1.1.4 Hierarchical Approximation

Finally, work on the smallest grammar problem qualitatively extends the study of approximation algorithms. In particular, we shift from problems on "flat" objects (such as graphs, CNF formulas, bins and weights, etc.) to a hierarchical object, context-free grammars. This is a significant shift. Many real-world problems such as circuit design and image compression have a hierarchical nature. (We describe some of these in more detail in Chapter 6.) But standard approximation techniques such as linear and semidefinite programming are not easily transferred to this new domain.

In Chapter 5, we present two distinct solutions to the smallest grammar problem based on completely different techniques. What elements of these techniques can be transferred to other hierarchical optimization problems remains to be seen. Beyond this, in Section 4.5 we formulate a class of *global algorithms* for the smallest grammar problem. The idea underlying global algorithms is simple enough that one can readily devise analogous algorithms for other hierarchical problems. We expend considerable effort analyzing global algorithms for the smallest grammar problem, but unfortunately our results remain far from complete. The need for a deeper understanding here is perhaps the most interesting open problem radiating from this work.

1.2 Previous Work

The are many algorithms that implicitly or explicitly attempt to solve the smallest grammar problem. Most arose in the data compression community, but some were generated in fields as diverse as linguistics, computational biology, and circuit design.

The smallest grammar problem was articulated explicitly by two groups of authors at about the same time. Nevill-Manning and Witten stated the problem and proposed the SEQUITUR algorithm as a solution [27, 29]. Their main focus was on extracting patterns from DNA sequences, musical scores, and even the Church of Latter-Day

Saints genealogical database, although they evaluated SEQUITUR as a compression algorithm as well.

The other group, consisting of Kieffer, Yang, Nelson, and Cosman, approached the smallest grammar problem from a traditional data compression perspective [17, 16, 18]. They offered a host of algorithms including BISECTION, MPM, and LONGEST MATCH. Furthermore, they gave an algorithm, which we refer to as SEQUENTIAL, in the same spirit as SEQUITUR, but with significant defects removed. All of these algorithms are described and analyzed in Chapter 4. Interestingly, on inputs with power-of-two lengths, the BISECTION algorithm of Nelson, Kieffer, and Cosman [26] gives essentially the same representation as a binary decision diagram (BDD) [9]. BDDs have been used widely in digital circuit analysis since the 1980's and also recently exploited for more general compression tasks [15, 22].

While these two lines of research led to the first clear articulation of the smallest grammar problem, its roots go back to much earlier work in the 1970's. In particular, Lempel and Ziv approached the problem from the direction of Kolmogorov complexity [25]. Over time, however, their work evolved toward data compression, beginning with a seminal paper [39] proposing the LZ77 compression algorithm. This procedure does *not* represent a string by a grammar, but rather with a different structure. Nevertheless, we show in Chapter 5 that LZ77 is deeply entwined with grammar-based compression. Lempel and Ziv soon produced another algorithm, LZ78, which did implicitly represent a string with a grammar [40]. We describe and analyze LZ78 in detail in Chapter 4. In 1984, Welch increased the efficiency of LZ78 with a new procedure, now known as LZW [37]. In practice, LZW is much preferred over LZ78, but for our purposes the difference is small.

Also in the 1970's, Storer and Szymanski were exploring a wide range of "macro-based" compression schemes [33, 35, 34]. They defined a collection of attributes that such a compressor might have, such as "recursive", "restricted", "overlapping", etc. Each combination of these adjectives described a different scheme, many of which they considered in detail. Apparently, none of these was precisely the context-free grammar model we employ here, but many of them were very close. So much so that our proof in Chapter 3 that the smallest-grammar problem is NP-hard is closely based on Storer and Szymanski's arguments about macro-based compression.

Recently, the smallest grammar problem has received increasing interest in a broad range of communities. For example, de Marcken's thesis [10] investigated whether the structure of the smallest grammar generating a large, given body of English text could lead to insight about the structure of the language itself. Lanctot, Li, and Yang [23] proposed using the LONGEST MATCH algorithm for the smallest grammar problem to estimate the entropy of DNA sequences. Apostolico and Lonardi [1, 2, 3] suggested a scheme that we call GREEDY and applied it to the same problem. Larsson and Moffat proposed RE-PAIR [24] as a general, grammar-based algorithm. (This scheme was partly anticipated by Gage's byte-pair encoding algorithm [13].) Most of these procedures are described and analyzed in Chapter 4.

Beyond the design of new algorithms for the smallest grammar problem, there has been an effort to develop algorithms that manipulate strings while still in compressed form. For example, Kida [14] and Shibata, et al. [32] have proposed pattern matching algorithms that run in time related not to the length of the searched string, but rather to the size of the grammar representing it. The good performance of such algorithms is emerging as a significant advantage of grammar-based compression over other compression techniques such as LZ77.

Previous work on monomial evaluation also has major implications for the smallest grammar problem. However, we defer a review of key works in that area until Section 3.3, after we have established the link between these problems.

In summary, the smallest grammar problem has been considered by many authors in many disciplines for many reasons over a span of decades. Frequently, it appears that one group of authors is unaware of very similar work by another group in a different field, limiting the cross-fertilization and advancement of ideas. Given this level of interest, it is remarkable that the problem has not attracted greater attention in the general algorithms community.

1.3 Summary of Our Contributions

This thesis makes four main contributions, which are enumerated below. Throughout, n denotes the length of an input string, and m^* denotes the size of the smallest grammar generating that one string.

1. We show that the smallest grammar generating a given string is hard to approximate to within a small constant factor. Furthermore, we show that an $o(\log n / \log\log n)$ approximation would require progress on a well-studied problem in computational algebra.

2. We bound approximation ratios for several of the best-known grammar-based compression algorithms. These results are summarized below:

	Approximation Ratio	
Algorithm	Upper Bound	Lower Bound
LZ78	$O((n/\log n)^{2/3})$	$\Omega(n^{2/3}/\log n)$
Bisection	$O((n/\log n)^{1/2})$	$\Omega(n^{1/2}/\log n)$
Sequential	$O((n/\log n)^{3/4})$	$\Omega(n^{1/3})$
Longest Match	$O((n/\log n)^{2/3})$	$\Omega(\log\log n)$
Greedy	$O((n/\log n)^{2/3})$	$> 1.37\ldots$
Re-Pair	$O((n/\log n)^{2/3})$	$\Omega(\sqrt{\log n})$

   The bounds for LZ78 hold for some variants, including LZW. Results for MPM mirror those for Bisection. The lower bound for Sequential extends to Sequitur.

3. We give new algorithms for the smallest grammar problem with exponentially better approximation ratios. First, we give a simple $O(\log^3 n)$ approximation. Then we provide a more complex $O(\log(n/m^*))$ approximation based on an entirely different approach.

4. We bring to light an intricate connection between grammar-based compression and the well-known LZ77 compression scheme.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 contains definitions and notational conventions, together with some basic lemmas. In Chapter 3, we establish the hardness of the smallest grammar problem in two different and complementary senses. Then, in Chapter 4, we analyze the most widely known algorithms for the smallest grammar problem. Following this, we propose new algorithms in Chapter 5 with approximation ratios that are exponentially better. The thesis concludes with Chapter 6, which discusses some of the many interesting lines of research radiating from this problem.

Chapter 2

Preliminaries

This chapter introduces terminology and notation used throughout. We then prove some basic lemmas about grammars to make the model more familiar and for use in later chapters.

2.1 Definitions and Notation

A *grammar* G is a 4-tuple $(\Sigma, \Gamma, S, \Delta)$. Here Σ is a finite alphabet whose elements are called *terminals*, Γ is a disjoint set whose elements are called *nonterminals*, and $S \in \Gamma$ is a special nonterminal called the *start symbol*. All other nonterminals are called *secondary*. In general, the word *symbol* refers to any terminal or nonterminal. The last component of a grammar, denoted Δ, is a set of *rules* of the form $T \to \alpha$, where $T \in \Gamma$ is a nonterminal and $\alpha \in (\Sigma \cup \Gamma)^*$ is a string of symbols referred to as the *definition* of T.

An example grammar is shown below. In this case, the set of terminals Σ is $\{x, y, z\}$, the set of nonterminals Γ is $\{S, A, B, C\}$, the start symbol is S, and Δ comprises four rules.

$$
\begin{aligned}
S &\rightarrow xAAyAzxBC \\
A &\rightarrow BBxCyB \\
B &\rightarrow xCyC \\
C &\rightarrow zzz
\end{aligned}
$$

The *left side of a rule* $T \rightarrow \alpha$ is the symbol T, and the *right side of the rule* is the string α. Similarly, the *left side of a grammar* consists of all nonterminals on the left sides of rules, and the *right side of a grammar* consists of all strings on the right sides of rules.

In the grammars we consider, there is exactly one rule $T \rightarrow \alpha$ in Δ for each nonterminal $T \in \Gamma$. Furthermore, all grammars are acyclic; that is, there exists an ordering of the nonterminals Γ such that each nonterminal precedes all the nonterminals in its definition. These properties guarantee that a grammar generates exactly one finite-length string.

A grammar naturally defines an *expansion* function of the form $(\Sigma \cup \Gamma)^* \mapsto \Sigma^*$. The expansion of a string is obtained by iteratively replacing each nonterminal by its definition until only terminals remain. We denote the expansion of a string α by $\langle \alpha \rangle$, and the length of the expansion of a string by $[\alpha]$; that is, $[\alpha] = |\langle \alpha \rangle|$. (In contrast, $|\alpha|$ denotes the length of the string α in the traditional sense; that is, the number of symbols in the string.) For the example grammar above, we have:

$$
\begin{aligned}
\langle C \rangle &= zzz \\
\langle CyB \rangle &= zzzyxzzzyzzz \\
[CyB] &= 12 \\
\langle S \rangle &= xxzzzyzzzxzzzyzzzxzzyxzzzyzzzxzzzyzzzxzzzyzzzxzzy \\
& xzzzyzzzyxzzzyzzzxzzzyzzzxzzyxzzzyzzzzxxzzzyzzzzz
\end{aligned}
$$

Note that the string generated by a grammar is the expansion of its start symbol, $\langle S \rangle$.

The *size* of a grammar G is the total number of symbols in all definitions:

$$|G| \;=\; \sum_{T \rightarrow \alpha \;\in\; \Delta} |\alpha|$$

The example grammar has size $9 + 6 + 4 + 3 = 22$.

We use several notational conventions to compactly express strings. The symbol $\mid$ represents a terminal that appears only once in a string. (For this reason, we refer to $\mid$ as a *unique symbol.*) When $\mid$ is used several times in the same string, each appearance represents a different symbol. For example, $a \mid bb \mid cc$ contains five distinct symbols and seven symbols in total. Product notation is used to express concatenation, and parentheses are used for grouping. For example:

$$\begin{aligned} (ab)^5 &= ababababab \\ \prod_{i=1}^{3} ab^i \mid &= ab \mid abb \mid abbb \mid \end{aligned}$$

The input to the smallest grammar problem is never specified using such shorthand; we use it only for clarity of exposition in proofs, counterexamples, etc.

Finally, we observe the following variable-naming conventions throughout: terminals are lowercase letters or digits, nonterminals are uppercase letters, and strings of symbols are lowercase Greek. In particular, σ denotes the input to a compression algorithm, and n denotes its length; that is, $n = |\sigma|$. The size of a particular grammar for σ is m, and the size of the smallest grammar is m^*.

2.2 Basic Observations

In this section, we give some easy lemmas that highlight basic points about the grammar model of compression. We begin with absolute lower and upper bounds on the size of the smallest grammar generating a string of length n. In proofs here and elsewhere, we ignore the possibility of degeneracies where they raise no substantive issues, e.g. a nonterminal with an empty definition or a secondary nonterminal that never appears in a definition.

Lemma 1 *The smallest grammar for a string of length n has size $\Omega(\log n)$.*

Proof. Let G be an arbitrary grammar of size m. We show that G generates a string of length $O(3^{m/3})$, which implies the claim.

Define a sequence of nonterminals as follows. Let T_1 be the start symbol of grammar G. Let T_{i+1} be the nonterminal in the definition of T_i that has the longest expansion. (Break ties arbitrarily.) When a nonterminal defined only in terms of terminals is reached, the sequence ends. Note that the nonterminals in this sequence are distinct, since the grammar is acyclic.

Suppose that T_i has a definition of length k. Then the length of the expansion of T_i is upper bounded by k times the length of the expansion of T_{i+1}. By an inductive argument, the length of the expansion of T_1 is at most the product of the lengths of the definitions of all the nonterminals T_i. However, the sum of the lengths of all these definitions is at most m, and it is well known that a set of positive integers with sum at most m has product at most $3^{\lceil m/3 \rceil}$. Thus the length of the string generated by G is $O(3^{m/3})$ as claimed. $\square$

Lemma 2 *Every string of length n over an alphabet of size b has a grammar of size $O(n/\log_b n)$.*

Proof. Let σ be a string of length n over an alphabet of size b. Partition σ into segments of length $k = \log_b n - 2\log_b \log_b n$. Define a nonterminal for every possible string of k terminals. Then define the start symbol to be the sequence of $\lfloor n/k \rfloor$ of these nonterminals followed by at most $k-1$ terminals that altogether expand to σ. The total size of this grammar for σ is at most:

$$\begin{aligned}
b^k \cdot k + \frac{n}{k} + k &= \frac{n}{\log_b^2 n} \cdot (\log_b n - 2\log_b \log_b n) \\
&\quad + \frac{n}{\log_b n - 2\log_b \log_b n} \\
&\quad + \log_b n - 2\log_b \log_b n \\
&= O\left(\frac{n}{\log_b n}\right)
\end{aligned}$$

In the initial expression, the first term counts the cost of defining all secondary nonterminals, the second term counts nonterminals in the start rule, and the third counts terminals in the start rule. □

The preceding lemma might appear counterintuitive. A standard observation is that not all strings are compressible, and yet we have shown that every binary string of length n is generated by a grammar of size $O(n/\log n)$. The explanation, of course, is that the grammar employs a larger alphabet. Nevertheless, this makes the point that grammar size is an imperfect measure of compression performance. This issue is discussed further in Section 4.1.

Next we show that certain highly structured strings are generated by small grammars. We shall often make reference to this lemma in lower bound arguments.

Lemma 3 *Let α be the string generated by grammar G_α, and let β be the string generated by grammar G_β. Then:*

1. *There exists a grammar of size $|G_\alpha| + |G_\beta| + 2$ that generates the string $\alpha\beta$.*
2. *There exists a grammar of size $|G_\alpha| + O(\log k)$ that generates the string α^k.*

Proof. To establish (1), create a grammar containing all rules in G_α, all rules in G_β, and the start rule $S \to S_\alpha S_\beta$ where S_α is the start symbol of G_α and S_β is the start symbol of G_β.

For (2), begin with the grammar G_α, and call the start symbol A_1. We extend this grammar by defining nonterminals A_i with expansion α^i for various i. The start rule of the new grammar is A_k. If k is even (say, $k = 2j$), define $A_k \to A_j A_j$ and define A_j recursively. If k is odd (say, $k = 2j + 1$), define $A_k \to A_j A_j A_1$ and again define A_j recursively. When $k = 1$, we are done. With each recursive call, the nonterminal subscript drops by a factor of at least two and at most three symbols are added to the grammar. Therefore, the total grammar size is $|G_\alpha| + O(\log k)$. □

The following example illustrates the usefulness of Lemma 3. Suppose that we want to show that there exists a small grammar for the string:

$$\sigma_k \;=\; a^{k(k+1)/2}\,(ba^k)^{(k+1)^2}$$

For the duration of this example, let $\{\tau\}$ denote the size of the smallest grammar that generates the string τ. By part 1 of the lemma, we have:

$$\{\sigma_k\} \;\leq\; \left\{a^{k(k+1)/2}\right\} + \left\{(ba^k)^{(k+1)^2}\right\} + 2$$

Applying part 2 of the lemma to each braced expression on the right gives:

$$\begin{aligned}\{\sigma_k\} &\leq \{a\} + O\left(\log k(k+1)/2\right) + \left\{ba^k\right\} + O\left(\log(k+1)^2\right) + 2 \\ &= \left\{ba^k\right\} + O(\log k)\end{aligned}$$

Applying part 1 and then part 2 to the remaining braced expression gives:

$$
\begin{aligned}
\{\sigma_k\} &\leq \{b\} + \left\{a^k\right\} + 2 + O(\log k) \\
&= \left\{a^k\right\} + O(\log k) \\
&\leq \{a\} + O(\log k) + O(\log k) \\
&= O(\log k)
\end{aligned}
$$

The following lemma is used extensively in our analysis of previously-proposed algorithms. Roughly, it upper bounds the complexity of a string generated by a small grammar.

Lemma 4 *If a string σ is generated by a grammar of size m, then σ contains at most mk distinct substrings of length k.*

Proof. Let G be a grammar for σ of size m. For each rule $T \to \alpha$ in G, we upper bound the number of length-k substrings of $\langle T \rangle$ that are not substrings of the expansion of a nonterminal in α. Each such substring either begins at a terminal in α, or else begins with between 1 and $k-1$ terminals from the expansion of a nonterminal in α. Therefore, the number of such strings is at most $|\alpha| \cdot k$. Summing over all rules in the grammar gives the upper bound mk.

All that remains is to show that all substrings are accounted for in this calculation. To that end, let τ be an arbitrary length-k substring of σ. Find the rule $T \to \alpha$ such that τ is a substring of $\langle T \rangle$, and $\langle T \rangle$ is as short as possible. Thus, τ is a substring of $\langle T \rangle$ and is not a substring of the expansion of a nonterminal in α. Therefore, τ was indeed accounted for above. $\square$

Chapter 3

Hardness

We establish the hardness of the smallest grammar problem in two ways. First, we show that approximating the size of the smallest grammar to within a small constant factor is NP-hard. Second, we show that approximating the size to within $o(\log n/\log\log n)$ would require progress on an apparently difficult computational algebra problem. These two hardness arguments are curiously complementary, as we discuss in Section 3.4.

3.1 NP-Hardness

Theorem 5 *There is no polynomial-time algorithm for the smallest grammar problem with approximation ratio less than* $8569/8568$ *unless* $P = NP$.

Proof. We use a reduction from a restricted form of vertex cover based closely on arguments by Storer and Szymanski [35, 34]. Let $H = (V, E)$ be a graph with maximum degree three and $|E| \geq |V|$. We can map the graph H to a string σ over an alphabet that includes a distinct terminal (denoted v_i) corresponding to each vertex $v_i \in V$ as follows:

$$\sigma = \prod_{v_i \in V} (\#v_i \mid v_i\# \mid)^2 \qquad \prod_{v_i \in V} (\#v_i\# \mid) \qquad \prod_{(v_i,v_j) \in E} (\#v_i\#v_j\# \mid)$$

We will show that the smallest grammar for σ has size $15|V| + 3|E| + k$, where k is the size of the minimum vertex cover for H. However, the size of the minimum cover for this family of graphs is known to be hard to approximate below a ratio of 145/144 unless P = NP [4]. Therefore, it is equally hard to approximate the size of the smallest grammar for σ below the ratio:

$$\rho = \frac{15|V| + 3|E| + \frac{145}{144}k}{15|V| + 3|E| + k}$$

Since all vertices in H have degree at most three, $|E| \leq \frac{3}{2}|V|$. Furthermore, each vertex can cover at most three edges, and so we get that k, the size of the minimum vertex cover, is at least $\frac{1}{3}|E| \geq \frac{1}{3}|V|$. The expression above is minimized when $|E|$ is large and k is small. Therefore, we get the lower bound:

$$\begin{aligned}\rho &\geq \frac{15|V| + 3 \cdot \frac{3}{2}|V| + \frac{145}{144}(\frac{1}{3}|V|)}{15|V| + 3 \cdot \frac{3}{2}|V| + (\frac{1}{3}|V|)} \\ &= \frac{8569}{8568}\end{aligned}$$

All that remains is to verify that the smallest grammar for σ has size $15|V| + 3|E| + k$. To accomplish this, the main effort is directed toward showing that the smallest grammar for σ must have a very particular structure. Let G be an arbitrary grammar that generates σ. Suppose that there exists a nonterminal with an expansion of some form other than $\#v_i$, $v_i\#$, or $\#v_i\#$. Then that nonterminal either appears at most once in G or else expands to a single character, since no other substring of two or more characters appears multiple times in σ. Replacing each occurrence of this nonterminal by its definition and deleting its defining rule can only decrease the size of G. Thus, in searching for the smallest grammar for σ, we need only consider grammars in which every nonterminal has an expansion of the form $\#v_i$, $v_i\#$, or $\#v_i\#$.

Next, suppose grammar G does not contain a nonterminal with expansion $\#v_i$. Then this string must appear at least twice in the start rule, since the two occurrences generated by the first product term can not be written another way. Adding a nonterminal with expansion $\#v_i$ costs two symbols, but also saves at least two symbols, and consequently gives a grammar no larger than G. Similar reasoning applies for strings of the form $v_i\#$. Thus, we need only consider grammars in which there are nonterminals with expansions $\#v_i$ and $v_i\#$ for all vertices v_i in the graph H.

Finally, let $C \subseteq V$ denote the set of vertices v_i such that G contains a rule for the substring $\#v_i\#$. Now suppose that C is not a vertex cover for H. Then there exists an edge $(v_i, v_j) \in E$ such that G does not contain rules for either $\#v_i\#$ or $\#v_j\#$. As a result, the occurrences of these strings generated by the second product term must both be represented by at least two symbols in the start rule of G. Furthermore, the string $\#v_i\#v_j\#$ generated by the third product term must be represented by at least three symbols. However, defining a nonterminal with expansion $\#v_i\#$ costs two symbols (since there is already a nonterminal with expansion $\#v_i$), but saves at least two symbols as well, giving a grammar no larger than before. Therefore, we need only consider grammars such that the corresponding set of vertices C is a vertex cover.

The size of a grammar with the structure described above is $8|V|$ for the first section of the start rule, plus $3|V| - |C|$ for the second section, plus $3|E|$ for the third section, plus $4|V|$ for rules for strings of the form $\#v_i$ and $v_i\#$, plus $2|C|$ for rules for strings of the form $\#v_i\#$, which gives $15|V| + 3|E| + |C|$. This quantity is minimized when C is a minimum vertex cover. In that case, the size of the grammar is $15|V| + 3|E| + k$ as claimed. $\square$

It is not known whether the smallest grammar problem is NP-hard when restricted to input strings over a binary alphabet. Then again, no exact algorithm is known even for input strings over a *unary* alphabet despite substantial efforts by many researchers. The precise state of affairs is described in Section 3.3.

3.2 Hardness via Addition Chains

This section demonstrates the hardness of the smallest grammar problem in an alternative sense: a procedure with an approximation ratio $o(\log n/\log\log n)$ would imply progress on an apparently difficult algebraic problem in a well-studied area.

Consider the following problem. Let $k_1, k_2, \ldots, k_p$ be positive integers. How many multiplications are required to compute $x^{k_1}, x^{k_2}, \ldots, x^{k_p}$, where x is a real number? For example, we could compute x^9 and x^{23} using seven multiplications as follows:

$$
\begin{array}{rclcrcl}
x^2 & = & x \cdot x & \qquad & x^{18} & = & x^9 \cdot x^9 \\
x^4 & = & x^2 \cdot x^2 & & x^{22} & = & x^{18} \cdot x^4 \\
x^8 & = & x^4 \cdot x^4 & & x^{23} & = & x^{22} \cdot x \\
x^9 & = & x^8 \cdot x & & & &
\end{array}
$$

This problem has a convenient, alternative formulation. An *addition chain* is an increasing sequence of positive integers starting with 1 and with the property that every other term is the sum of two (not necessarily distinct) predecessors. The connection between addition chains and computing powers is straightforward: the terms in the chain indicate the powers to be computed. For example, $1, 2, 4, 8, 9, 18, 22, 23$ is an addition chain corresponding to the algorithm given above for computing x^9 and x^{23} using seven multiplications. In general, the number of multiplications required to compute $x^{k_1}, x^{k_2}, \ldots, x^{k_p}$ is one less than the shortest addition chain containing all of $k_1, k_2, \ldots, k_p$.

Surprisingly, the power evaluation and addition chain problems are closely related to the smallest grammar problem. In particular, the problem of computing, say, x^9 and x^{23} using the fewest multiplications is closely tied to the problem of finding the smallest grammar for the string $\sigma = x^9 \mid x^{23}$. Roughly speaking, a grammar for σ can be regarded as an algorithm for computing x^9 and x^{23} and vice versa. The following theorem makes these mappings precise.

Theorem 6 *Let $T = \{k_1, \ldots k_p\}$ be a set of distinct positive integers, and define the string σ as follows:*

$$\sigma = x^{k_1} \mid x^{k_2} \mid \ldots \mid x^{k_p}.$$

Then the following relationship holds, where l^* is the length of the shortest addition chain containing T and m^* is the size of the smallest grammar for the string σ:

$$l^* \leq m^* \leq 4l^*.$$

Proof. We translate the grammar of size m^* for string σ into an addition chain containing T with length at most m^*. This will establish the left inequality, $l^* \leq m^*$. For clarity, we accompany the description of the procedure with an example and some intuition. Let T be the set $\{9, 23\}$. Then $\sigma = x^9 \mid x^{23}$. The smallest grammar for this string has size $m^* = 13$:

$$\begin{aligned} S &\to A \mid AABxx \\ A &\to BBB \\ B &\to xxx \end{aligned}$$

We begin converting the grammar to an addition chain by ordering the rules so that their expansions increase in length. Then we underline symbols in the grammar according to the following two rules:

1. The first symbol in the first rule is underlined.

2. Every symbol preceded by a nonterminal or an x is underlined.

Thus, in the example, we would underline as follows:

$$\begin{aligned} B &\rightarrow \underline{xxx} \\ A &\rightarrow B\underline{BB} \\ S &\rightarrow A \mid A\underline{ABxx} \end{aligned}$$

Each underlined symbol generates one term in the addition chain. Starting at the underlined symbol, work leftward until the start of the definition or a unique symbol is reached. This defines a substring ending with the underlined symbol. The length of the expansion of this substring is a term in the addition chain. In the example, we would obtain the substrings:

$$x, xx, xxx, BB, BBB, AA, AAB, AABx, AABxx$$

and hence the addition chain:

$$1, 2, 3, 6, 9, 18, 21, 22, 23$$

Intuitively, the terms in the addition chain produced above are the lengths of the expansions of the secondary nonterminals in the grammar. But these alone do not quite suffice. To see why, observe that the length of the expansion of a nonterminal is the sum of the expansion lengths of the symbols in its definition. For example, the rule $T \rightarrow ABC$ implies that $[T] = [A] + [B] + [C]$. If we ensure that the addition chain contains $[A]$, $[B]$, and $[C]$, then we still can not immediately add $[T]$. This is because $[T]$ is the sum of three preceding terms, instead of two. Thus, we must also include, say, the term $[AB]$, which is itself the sum of $[A]$ and $[B]$. Now $[T]$ is expressible as the sum of two preceding terms, $[AB]$ and $[C]$, and so we have a valid addition chain. The creation of such extra terms is what the elaborate underlining procedure accomplishes. In this light, is easy to verify that the construction detailed above gives an addition chain of length at most m^* that contains T.

All that remains is to establish the second inequality, $m^* \leq 4l^*$. We do this by translating an addition chain of length l into a grammar for the string σ of size at most $4l$. As before, we carry along an example. Let $T = \{9, 23\}$. The shortest addition chain containing T has length $l = 7$:

$$1, 2, 4, 5, 9, 18, 23$$

We associate the symbol x with the initial 1 and a distinct nonterminal with each subsequent term. Each nonterminal is defined in terms of the symbols associated with two preceding terms, just as each term in the addition sequence is the sum of two predecessors. The start rule consists of the nonterminals corresponding to the terms in T, separated by uniques. In the example, this gives the following grammar:

$$\begin{array}{lclclcl} T_2 & \rightarrow & xx & \qquad & T_4 & \rightarrow & T_2T_2 \\ T_5 & \rightarrow & T_4x & & T_9 & \rightarrow & T_5T_4 \\ T_{18} & \rightarrow & T_9T_9 & & T_{23} & \rightarrow & T_{18}T_5 \end{array}$$
$$S \rightarrow T_9 \mid T_{23}$$

The start rule has length $2|T| - 1 \leq 2l^*$, and the $l^* - 1$ secondary rules each have exactly two symbols on the right. Thus, the total size of the grammar is at most $4l^*$. □

The implication of the preceding theorem is that, up to a constant factor, approximating the smallest grammar for a string is at least as hard as approximating the shortest addition chain containing a specified set of numbers.

3.3 Background on Addition Chains

Addition chains have been studied extensively for decades. There is a survey in Knuth [19] and a less comprehensive, but more recent survey due to Thurber [36].

The classical *addition chain problem* is to find the shortest addition chain containing a single, specified integer n. This is closely allied with the problem of finding the smallest grammar for the unary string x^n. All major bounds and techniques carry over.

The i-th term of an addition chain can be no larger than 2^{i-1}. Consequently, the shortest addition chain for x^n must have length at least $\log_2 n$. There is a simple algorithm with approximation ratio 2. Begin with a chain containing only n. Then repeat the following step recursively. If n is even, prepend $n/2$ to the chain, and set $n = n/2$. If n is odd, prepend $(n-1)/2$ and $n-1$ to the chain, and set $n = (n-1)/2$. When n reaches 1, stop. With each recursive call, n decreases by a factor of two and at most two terms are added to the chain. Therefore, the length of the resulting addition chain is at most $2\log_2 n$, giving a 2-approximation as claimed.

A somewhat more subtle algorithm known as the *M-ary method* gives a $1 + O(1/\log\log n)$ approximation. (This is apparently folklore.) One writes n in a base M, which is a power of 2:

$$n \quad = \quad d_0 M^k + d_1 M^{k-1} + d_2 M^{k-2} + \ldots + d_{k-1} M + d_k$$

The addition chain begins $1, 2, 3, \ldots, M-1$. Then one puts d_0, doubles it $\log M$ times, adds d_1 to the result, doubles that $\log M$ times, adds d_2 to the result, etc. The total length of the addition chain produced is at most:

$$(M-1) + \log n + \frac{\log n}{\log M} \quad = \quad \log n + O(\log n / \log\log n)$$

In the expression on the left, the first term counts the first $M-1$ terms of the addition chain, the second counts the doublings, and the third counts the increments of d_i. The equality follows by substituting $M = \log n / \log\log n$.

The M-ary method is very nearly the best possible. Erdös [12] showed that, in a certain sense, the shortest addition chain containing n has length at least $\log n + \log n / \log\log n$ for almost all n.

Nevertheless, an exact solution to the addition chain problem remains strangely elusive. The M-ary method runs in time polylog(n) and gives a $1 + o(1)$ approximation. However, even if exponentially more time is allowed, poly(n), no exact algorithm (and apparently even no better approximation algorithm) is known. A determined attempt by Bleichenbacher [5] yielded an exact algorithm with an empirically observed running time of $n^{f(n)}$ for some slow-growing function f. In particular, he observed that $f(n)$ was approximately 2 for small n and about 2.3 for $n = 500,000$.

The *general addition chain problem* is to find the shortest addition chain containing a specified set of integers $k_1, \ldots, k_p$. Pursuing the obvious approach to the classical addition chain problem described above leads naturally to this one. Suppose that we were trying to find the shortest addition chain containing a single integer n. We could do this by finding the shortest addition chain containing two integers k_1 and k_2 that sum to n and then appending n to the end of that chain. This, however, requires a solution to the general addition chain problem.

The general addition chain problem is known to be NP-hard if the integers k_i are given in binary [11]. There is an easy $O(\log n)$ approximation algorithm, where $n = \sum k_i$. First, generate all powers of two less than or equal to the maximum of the input integers k_i. Then form each k_i independently by summing a subset of these powers corresponding to 1's in the binary representation of k_i. In 1976, Yao [38] pointed out that the second step could be tweaked in the spirit of the M-ary method. Specifically, he groups the bits of k_i into blocks of size $\log\log k_i - 2\log\log\log k_i$ and tackles all blocks with the same bit pattern at the same time. This improves the approximation ratio slightly to $O(\log n / \log\log n)$.

Yao's method retains a frustrating aspect of the naive algorithm: there is no attempt to exploit special relationships between the integers k_i; each one is treated independently. This can lead to rather suboptimal results. For example, suppose $k_i = 3^i$ for $i = 1$ to p. Then there exists a short addition chain containing all of the k_i: $\underline{1}$, 2, $\underline{3}$, 6, $\underline{9}$, 18, $\underline{27}$, 54, $\underline{81}$, ... But Yao's algorithm thrashes about, effectively trying to represent powers of three in base two.

Remarkably, even if the k_i are written in unary, apparently no polynomial time algorithm with a better approximation ratio than Yao's is known. However, since Theorem 6 links addition chains and small grammars, finding an approximation algorithm for the smallest grammar problem with ratio $o(\log n / \log\log n)$ would require improving upon Yao's method.

3.4 An Observation on Hardness

We have seen that the smallest grammar problem is hard to approximate via reductions from two very different problems. Interestingly, there is also a marked difference on the other end of these reductions; that is, in the types of strings generated.

Specifically, Theorem 5 maps graphs to strings with large alphabets and few repeated substrings. As a consequence, there is minimal potential for the use of hierarchy when representing such strings with grammars. Thus, we show the NP-completeness of the smallest grammar problem by analyzing a space of input strings that specifically dodges the most interesting aspect of the problem: hierarchy.

On the other hand, Theorem 6 maps addition chain problems to strings over a unary alphabet (plus unique symbols). The potential for use of hierarchy in representing such strings is enormous; in fact, the whole challenge now is to construct an intricate hierarchy of rules, each defined in terms of the others. Thus, this reduction more effectively captures the most notable aspect of the smallest grammar problem.

Taken together, these two reductions show that the smallest grammar problem is hard in both a "combinatorial packing" sense and a seemingly orthogonal "hierarchical structuring" sense.

Chapter 4

Analysis of Previous Algorithms

In this chapter, we establish upper and lower bounds on the approximation ratios of six previously proposed algorithms for the smallest grammar problem: LZ78, Bisection, Sequential, Longest Match, Greedy, and Re-Pair. In addition, we discuss some closely-related algorithms: LZW, MPM, and Sequitur. The results in this chapter are summarized in the table below, which also appeared in the introduction.

	Approximation Ratio	
Algorithm	Upper Bound	Lower Bound
LZ78	$O((n/\log n)^{2/3})$	$\Omega(n^{2/3}/\log n)$
Bisection	$O((n/\log n)^{1/2})$	$\Omega(n^{1/2}/\log n)$
Sequential	$O((n/\log n)^{3/4})$	$\Omega(n^{1/3})$
Longest Match	$O((n/\log n)^{2/3})$	$\Omega(\log\log n)$
Greedy	$O((n/\log n)^{2/3})$	$> 1.37\ldots$
Re-Pair	$O((n/\log n)^{2/3})$	$\Omega(\sqrt{\log n})$

4.1 Compression Versus Approximation

We regard the algorithms in this chapter as approximation algorithms for the smallest grammar problem. However, most were originally designed as compression algorithms. Generally speaking, a good grammar-based compression algorithm should seek the smallest possible grammar generating the input string. But there do exist significant disconnects between our theoretical study of the smallest grammar problem and practical data compression. We highlight three of these below.

Most obviously, our optimization criteria is grammar size, whereas the optimization criteria in data compression is the length of the compressed string in bits. There is some relationship between the two measures, of course. At the very least, a grammar of size m can be represented by a string of at most $m \log m$ bits by assigning each distinct symbol a unique $(\log m)$-bit representation. Such a $\log m$ factor is small by the standards of our worst-case theoretical analyses, but enormous by practical data compression standards.

Perhaps more importantly, data compression algorithms are typically designed with an eye toward *universality* (asymptotically optimal compression of strings generated by a finite-state source) and *low redundancy* (fast convergence to that optimum). Informally, strings generated by a finite-state source have high entropy; that is, they are compressible by only a constant factor. Thus, the main focus in the design of a data compressor is on high entropy strings. In fact, Kosaraju and Manzini [21] point out that universality and redundancy are not meaningful measures of a compressor's performance on low entropy strings. Consequently, performance on low-entropy strings is typically neglected completely.

In contrast, we focus here on the approximation ratio of algorithms for the smallest grammar problem. Now high-entropy strings are of little interest. If the smallest grammar for an input string of length n has size, say, $n/\log n$, then any compressor can approximate the smallest grammar to within a $\log n$ factor. However, low-entropy strings present a serious challenge. If an input string is generated by a grammar of size, say, $n^{1/3}$, then a carelessly designed algorithm could exhibit an approximation ratio as bad as $n^{2/3}$. Thus, grammar-based data compressors and approximation algorithms can both be regarded as approaches to the smallest grammar problem, but they target different ranges of inputs.

Finally, practical data compression raises many considerations given scant attention here. For example, the running time should be linear in the length of the input string. Even an $O(n \log n)$ running time is justifiable only in restricted contexts such as long-term archiving, where the initial compression cost is small next to the storage costs over time. One can not even neglect constants in the running time that are hidden by asymptotic notation. Ideally, a compressor should also be on-line; that is, a single left-to-right pass through the input string should suffice. Space consumption throughout this pass should, preferably, be a function of the size of the compressed string, not the size of the string being compressed.

As a result of these disconnects, one must take the results in the remainder of this chapter with a caveat: while we show that many grammar-based data compression algorithms exhibit mediocre approximation ratios, the designers of these algorithms are concerned with slightly different measures, different inputs, and many practical issues that we ignore.

4.2 LZ78

The well-known LZ78 compression scheme was described by Lempel and Ziv [40]. While not originally described in terms of grammars, the algorithm has a natural interpretation within our framework.

The Procedure

In traditional terms, LZ78 represents a string σ by a sequence of pairs. For example, one possible sequence is as follows:

$$(0, a)\ (1, b)\ (0, b)\ (2, a)\ (3, a)\ (2, b)\ (4, a)$$

Each pair expands to a substring of σ, and the concatenation of all their expansions is the whole of σ. Each pair is of the form (i, c), where i is an integer and c is a symbol in σ. If i is zero, then the expansion of the pair is simply c. Otherwise, the expansion is equal to the expansion of the i-th pair followed by the symbol c. The expansions of the pairs in the example sequence are as follows:

$$\begin{array}{rclcrcl} (0,a) & = & a & \quad & (3,a) & = & ba \\ (1,b) & = & ab & & (2,b) & = & abb \\ (0,b) & = & b & & (4,a) & = & abaa \\ (2,a) & = & aba & & & & \end{array}$$

Consequently, the entire sequence represents the string $a\ ab\ b\ aba\ ba\ abb\ abaa$, where spaces are added for clarity.

The sequence-of-pairs representation of a string is generated by LZ78 in a single left-to-right pass as follows. Begin with an empty sequence of pairs. At each step, find the shortest, nonempty prefix of the unprocessed portion of the string that is not the expansion of a pair already in the sequence. There are two cases:

1. If this prefix consists of a single symbol c, then append the pair $(0, c)$ to the sequence.

2. Otherwise, this prefix must be of the form αc, where α is the expansion of some pair already in the sequence (say, the i-th one) and c is a symbol. In this case, append the pair (i, c) to the sequence.

For example, consider the earlier string, $aabbababaabbabaa$. We begin with an empty sequence of pairs, and then take the shortest, nonempty prefix that is not the expansion of an existing pair. This prefix is the single symbol a. This is case (1), and so we append the pair $(0, a)$. The shortest, nonempty prefix of the remainder which is not the expansion of an existing pair is now ab. This is case (2); ab consists of the expansion of the first pair followed by the symbol b, and so we append the pair $(1, b)$. The procedure continues in this way until the entire input string is translated to a sequence of pairs.

LZ78 in Grammar Terms

An LZ78 pair sequence maps naturally to a grammar. Associate a nonterminal T with each pair (i, c). If i is zero, define the nonterminal by $T \to c$. Otherwise, define the nonterminal to be $T \to Uc$, where U is the nonterminal associated with the i-th pair. The right side of the start rule contains all the nonterminals associated with pairs. For example, the grammar associated with the example sequence is as follows:

$$
\begin{array}{rclcrcl}
S & \to & X_1X_2X_3X_4X_5X_6X_7 & & & & \\
X_1 & \to & a & & X_5 & \to & X_3a \\
X_2 & \to & X_1b & & X_6 & \to & X_2b \\
X_3 & \to & b & & X_7 & \to & X_4a \\
X_4 & \to & X_2a & & & &
\end{array}
$$

Given this easy mapping, hereafter we simply regard the output of LZ78 as a grammar rather than as a sequence of pairs.

Note that the grammars produced by LZ78 are of a very restricted form. In particular, the right side of each rule contains at most two symbols and at most one nonterminal. Subject to these restrictions, the smallest grammar for even the string x^n has size $\Omega(\sqrt{n})$. On the other hand, grammars with such a regular form can be encoded in bits more efficiently.

Analysis

The next two theorems provide nearly-matching upper and lower bounds on the approximation ratio of LZ78 when it is regarded as an approximation algorithm for the smallest grammar problem.

Theorem 7 *The approximation ratio of* LZ78 *is* $\Omega(n^{2/3}/\log n)$.

Proof. The lower bound follows by analyzing the behavior of LZ78 on input strings of the form

$$\sigma_k = a^{k(k+1)/2} \ (ba^k)^{(k+1)^2}$$

where $k > 0$. The length of this string is $n = \Theta(k^3)$. Repeated application of Lemma 3 implies that there exists a grammar for σ_k of size $O(\log k) = O(\log n)$. (In fact, this is the example string analyzed immediately after the statement of Lemma 3.)

The string σ_k is processed by LZ78 in two stages. During the first, the $k(k+1)/2$ leading a's are consumed and nonterminals with expansions $a, aa, aaa, \dots, a^k$ are created. During the second stage, the remainder of the string is consumed and a nonterminal with expansion $a^i b a^j$ is created for all i and j between 0 and k. For example, σ_4 is represented by nonterminals with expansions as indicated below:

a	aa	aaa	$aaaa$	
b	$aaaab$ $aaaaba$	$aaab$ $aaaabaa$	aab $aaaabaaa$	ab $aaaabaaaa$
ba	$aaaba$ $aaabaa$	$aaba$ $aaabaaa$	aba $aaabaaaa$	
baa	$aabaa$ $aabaaa$	$abaa$ $aabaaaa$		
$baaa$	$abaaa$ $abaaaa$			
$baaaa$				

The pattern evident above can be shown to occur in general by a routine induction. As a result, the grammar produced by LZ78 has size $\Omega(k^2) = \Omega(n^{2/3})$. Dividing by our upper bound on the size of the smallest grammar proves the claim. $\square$

Theorem 8 *The approximation ratio of* LZ78 *is* $O\left((n/\log n)^{2/3}\right)$.

Proof. Suppose that the input to LZ78 is a string σ of length n, and that the smallest grammar generating σ has size m^*. Let $S \to X_1 \dots X_p$ be the start rule generated by LZ78. Note that the size of the entire grammar is at most $3p$, since each nonterminal X_i is used once in the start rule and is defined by a rule with at most two symbols on the right side. Therefore, to upper bound the size of the LZ78 grammar, it suffices to upper bound p, the number of nonterminals in the start rule.

To that end, list the nonterminals X_i in order of increasing expansion length. Group the first m^* of these nonterminals, the next $2m^*$, the next $3m^*$, and so forth. Let g be the number of complete groups of nonterminals that can be formed in this way. By this definition of g, we have:

$$m^* + 2m^* + \ldots + (g+1)m^* \quad > \quad p$$

And so $p = O(g^2 m^*)$.

On the other hand, the definition of LZ78 implies that the X_i expand to distinct substrings of σ. But Lemma 4 states that σ contains at most $m^* k$ distinct substrings of length k. It follows that each nonterminal in the j-th group must expand to a string of length at least j, since the number of distinct, shorter substrings of σ is upper bounded by the number of nonterminals in preceding groups. Therefore, we have:

$$\begin{aligned} n &= [X_1] + \ldots + [X_p] \\ &\geq m^* + 2^2 m^* + 3^2 m^* + \ldots + g^2 m^* \end{aligned}$$

This inequality implies $g = O\left((n/m^*)^{1/3}\right)$. Substituting this bound on g into the upper bound on p obtained previously gives:

$$\begin{aligned} p &= O\left(\left(\frac{n}{m^*}\right)^{2/3} m^*\right) \\ &= O\left(\left(\frac{n}{\log n}\right)^{2/3} m^*\right) \end{aligned}$$

The second equality follows from Lemma 1, which says that the smallest grammar for a string of length n has size $\Omega(\log n)$. $\square$

LZW

Some practical improvements on LZ78 are embodied in a later algorithm, LZW [37]. The grammars implicitly generated by the two procedures are not substantively different, but LZW is more widely used in practice. For example, it is used to encode images in the the popular `gif` format. Interestingly, the bad strings introduced in Theorem 7 have a natural graphical interpretation. Here is σ_4 written in a 15×9 grid pattern:

$$
\begin{array}{c}
aaaaaaaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa \\
baaaabaaaabaaaa
\end{array}
$$

Thus, an image with colors in this simple vertical stripe pattern yields a worst-case string in terms of approximation ratio. This effect can be observed in practice on even small examples. For example, a 68×68 image consisting of four horizontal lines spaced 16 pixels apart is stored by Corel PhotoPaint, a commercial graphics program, in a 933 byte file. However, if the image is simply rotated ninety degrees to create vertical lines instead, the stored file grows to 1142 bytes, an increase of more than 20%.

4.3 Bisection

The Bisection algorithm was proposed by Kieffer, Yang, Nelson, and Cosman [18, 26]. For binary input strings of length 2^k, the same technique was employed much earlier in binary decision diagrams (BDDs), a data structure used to compactly represent and easily manipulate boolean functions. Indeed, BDDs have themselves been proposed as a compression tool more than once.

The Procedure

Bisection works on an input string σ as follows. Select the largest integer j such that $2^j < |\sigma|$. Partition σ into two substrings with lengths 2^j and $|\sigma| - 2^j$. Repeat this partitioning process recursively on each substring produced that has length greater than one. Afterward, create a nonterminal for every distinct string of length greater than one generated during this process. Each such nonterminal can then be defined by a rule with exactly two symbols on the right.

For example, consider the string $\sigma = 1110111010011$. We recursively partition and associate a nonterminal with each distinct substring generated as shown below:

$$\underbrace{1110111010011}_{S} \qquad\qquad S \to T_1 T_2$$

$$\underbrace{11101110}_{T_1} \quad \underbrace{10011}_{T_2} \qquad\qquad T_1 \to U_1 U_1 \quad T_2 \to U_2 1$$

$$\underbrace{1110}_{U_1} \quad 1110 \quad \underbrace{1001}_{U_2} \quad 1 \qquad\qquad U_1 \to V_1 V_2 \quad U_2 \to V_2 V_3$$

$$\underbrace{11}_{V_1} \quad \underbrace{10}_{V_2} \quad 11 \quad 10 \quad 10 \quad \underbrace{01}_{V_3} \quad 1 \qquad V_1 \to 11 \quad V_2 \to 10 \quad V_3 \to 01$$

Analysis

The following two theorems give nearly-matching lower and upper bounds on the approximation ratio of Bisection

Theorem 9 *The approximation ratio of* Bisection *is* $\Omega(\sqrt{n}/\log n)$.

Proof. We analyze the behavior of BISECTION on input strings of the form

$$\sigma_k \quad = \quad a(b^{2^k}a)^{2^k-1}$$

where $k > 0$. This string has length $n = 2^{2k}$. After k bisections, σ_k is partitioned into 2^k distinct substrings of length 2^k. In particular, each contains a single a, which appears in the i-th position in the i-th substring. For example, bisecting σ_2 twice gives four distinct strings:

$$abbb\ babb\ bbab\ bbba$$

A routine induction argument shows that this pattern holds in general for σ_k. Since each distinct substring generates a nonterminal, BISECTION produces a grammar of size $\Omega(2^k) = \Omega(\sqrt{n})$ on input σ_k.

On the other hand, Lemma 3 implies that there exists a grammar for σ_k of size $O(k) = O(\log n)$. The approximation ratio of $\Omega(\sqrt{n}/\log n)$ follows. □

Theorem 10 *The approximation ratio of* BISECTION *is* $O(\sqrt{n/\log n})$.

Proof. Suppose that the input to BISECTION is a string σ of length n, and that the smallest grammar generating σ has size m^*. Let j be the largest integer such that $2^j \leq n$. Note that the size of the BISECTION grammar for σ is at most twice the number of distinct substrings generated during the recursive partitioning process. Thus, it suffices to upper bound the latter quantity.

At most one string at each level of the recursion has a length that is not a power of two; therefore, there are at most j strings with irregular lengths. All remaining strings have length 2^i for some i between 1 and j. We can upper bound the number of these in two ways. On one hand, BISECTION creates at most one string of length 2^j, at most two of length 2^{j-1}, at most four of length 2^{j-2}, etc. On the other hand, Lemma 4 says that σ contains at most $2^i m^*$ distinct substrings of length 2^i. The first observation gives a good upper bound on the number of distinct long strings generated by the recursive partitioning process, and the second is tighter for short strings. Putting this all together, the size of the BISECTION grammar is at most:

$$2 \cdot \left(j + \sum_{i=1}^{\frac{1}{2}(j-\log j)} m^* 2^i + \sum_{i=\frac{1}{2}(j-\log j)}^{j} 2^{j-i} \right)$$
$$= O(\log n) + O\left(m^* \sqrt{\frac{n}{\log n}} \right) + O\left(\sqrt{n \log n} \right)$$
$$= O\left(m^* \sqrt{\frac{n}{\log n}} \right)$$

In the second equation, we use the fact that $m^* = \Omega(\log n)$ by Lemma 1. $\square$

MPM

BISECTION was generalized to an algorithm called MPM [18], which permits a string to be split more than two ways during the recursive partitioning process and allows that process to terminate early. For reasonable parameters, performance bounds are the same as for BISECTION.

4.4 SEQUENTIAL

Nevill-Manning and Witten introduced the SEQUITUR algorithm [27, 29]. Kieffer and Yang subsequently offered a similar, but improved algorithm that we refer to here as SEQUENTIAL [16].

The Procedure

SEQUENTIAL works as follows. Begin with an empty grammar and make a single left-to-right pass through the input string. At each step, find the longest prefix of the unprocessed portion of the input that is the expansion of a secondary nonterminal, and append that nonterminal to the start rule. If no prefix matches the expansion of a secondary nonterminal, then append the first terminal in the unprocessed portion of the input to the start rule. In either case, if the newly created pair of symbols at the end of the start rule already appears elsewhere in the grammar without overlap, then replace both occurrences by a new nonterminal whose definition is that pair. Finally, if some nonterminal is used only once after this substitution, replace it by its definition, and delete the corresponding rule.

As an example, consider the input string $\sigma = x \mid xx \mid xxxx \mid xxxxxxxx$. After six steps, the grammar is:

$$S \rightarrow x \mid xx \mid x$$

No secondary rules have been created so far, because every nonoverlapping pair of symbols occurs only once in this prefix. However, when the next x is appended to the start rule, there are two copies of the substring xx. Therefore the rule $R_1 \rightarrow xx$ is added to the grammar, and both occurrences of xx are replaced by R_1.

$$\begin{aligned} S &\rightarrow x \mid R_1 \mid R_1 \\ R_1 &\rightarrow xx \end{aligned}$$

Because the expansion of R_1 is now a prefix of the unprocessed part of σ, the next step consumes xx and appends R_1 to S. During the next few steps, the start rule expands to the following:

$$\begin{aligned} S &\rightarrow x \mid R_1 \mid R_1R_1 \mid R_1R_1 \\ R_1 &\rightarrow xx \end{aligned}$$

At this point, the pair R_1R_1 appears twice, and so a new rule is $R_2 \to R_1R_1$ is added and applied.

$$\begin{aligned} S &\to x \mid R_1 \mid R_2 \mid R_2 \\ R_1 &\to xx \\ R_2 &\to R_1R_1 \end{aligned}$$

The remainder of the input string matches the expansion of R_2, so this nonterminal is added to the start rule to complete the grammar:

$$\begin{aligned} S &\to x \mid R_1 \mid R_2 \mid R_2R_2 \\ R_1 &\to xx \\ R_2 &\to R_1R_1 \end{aligned}$$

Lower Bound

The next two theorems bound the approximation ratio of SEQUENTIAL. Both the upper and lower bounds are considerably more complex than was the case for LZ78 and BISECTION.

Theorem 11 *The approximation ratio of* SEQUENTIAL *is* $\Omega(n^{1/3})$.

Proof. We analyze the behavior of SEQUENTIAL on strings σ_k, defined as follows, for $k > 0$.

$$\begin{aligned} \sigma_k &= \alpha \mid \beta^{k/2} \\ \alpha &= 0^{k+1} \mid 0^{k+1} \mid \delta_0 \mid \delta_0 \mid \delta_1 \mid \delta_1 \mid \ldots \mid \delta_k \mid \delta_k \\ \beta &= \delta_k\delta_k \ \delta_k\delta_{k-1} \ \delta_k\delta_{k-2} \ \delta_k\delta_{k-3} \ \ldots \ \delta_k\delta_{k/2} \ 0^k \\ \delta_i &= 0^i10^{k-i} \end{aligned}$$

The prefix α forces SEQUENTIAL to create a nonterminal with expansion 0^{k+1}, a nonterminal with expansion δ_i for each i from 0 to k, and some nonterminals with shorter expansions that are not relevant here.

The remainder of the input, the string $\beta^{k/2}$, is consumed in segments of length $k+1$. This is because, at each step, the leading $k+1$ symbols of the unprocessed portion of the input string form either 0^{k+1} or else δ_i for some i. Consequently, the corresponding nonterminal is appended to the start rule at each step. Note, however, that the length of β is *not* a multiple of $k+1$. Each δ_i component of β does expand to a string of length $k+1$, but β ends with 0^k. As a result, each copy of β is represented by a different sequence of nonterminals. This is the inefficiency that we exploit.

The first copy of β is parsed almost as it is written above. The only difference is that the final 0^k at the end of this first copy is combined with the leading zero in the second copy of β and read as a single nonterminal. Thus, nonterminals with the following expansions are appended to the start rule as the first copy of β is processed:

$$\delta_k\delta_k \ \delta_k\delta_{k-1} \ \delta_k\delta_{k-2} \ \delta_k\delta_{k-3} \ \ldots \ \delta_k\delta_{k/2} \ 0^{k+1}$$

SEQUENTIAL parses the second copy of β differently, since the leading zero is already processed. Furthermore, the final 0^{k-1} in the second copy of β is combined with the two leading zeroes in the third copy and read as a single nonterminal:

$$\delta_{k-1}\delta_{k-1} \ \delta_{k-1}\delta_{k-2} \ \delta_{k-1}\delta_{k-3} \ \ldots \ \delta_{k-1}\delta_{k/2-1} \ 0^{k+1}$$

With two leading zeros already processed, the third copy of β is parsed yet another way. In general, an induction argument shows that the j-th copy (indexed from $j=0$) is read as:

$$\delta_{k-j}\delta_{k-j}\ \delta_{k-j}\delta_{k-j-1}\ \delta_{k-j}\delta_{k-j-2}\ \ldots\ \delta_{k-j}\delta_{k/2-j}\ 0^{k+1}$$

No consecutive pair of nonterminals ever comes up twice in this entire process, and so no new rules are created. Since the input string contains $k/2$ copies of β and each is represented by about k nonterminals, the grammar generated by SEQUENTIAL has size $\Omega(k^2)$.

On the other hand, there exists a grammar for σ_k of size $O(k)$. First, create a nonterminal Z_i with expansion 0^i for each i up to $k+1$. Each such nonterminal can be defined in terms of its predecessors using only two symbols:

$$Z_i \rightarrow 0Z_{i-1}$$

Next, define a nonterminal D_i with expansion δ_i for each i using three symbols:

$$D_i \rightarrow Z_i 1 Z_{k-i}$$

Now define a nonterminal A with expansion α and a nonterminal B with expansion β using the Z_i and D_i:

$$\begin{aligned} A &\rightarrow Z^{k+1} \mid Z^{k+1} \mid D_0 \mid D_0 \mid D_1 \mid D_1 \mid \ldots \mid D_k \mid D_k \\ B &\rightarrow D_k D_k\ D_k D_{k-1}\ D_k D_{k-2}\ D_k D_{k-3}\ \ldots\ D_k D_{k/2}\ Z^k \end{aligned}$$

Finally, using Lemma 3, $O(\log k)$ additional symbols suffice to define a start symbol with expansion $\alpha \mid \beta^{k/2}$. In total this grammar has size $O(k)$. Therefore the approximation ratio of SEQUENTIAL is $\Omega(k) = \Omega(n^{1/3})$. $\square$

Irreducible Grammars

Our upper bound on the approximation ratio of SEQUENTIAL relies on a property of the output. In particular, Kieffer and Yang [16] show that SEQUENTIAL produces an *irreducible* grammar; that is, one which has the following three properties:

1. All nonoverlapping pairs of adjacent symbols on the right side of the grammar are distinct.
2. Every secondary nonterminal appears at least twice on the right side of the grammar.
3. No two symbols in the grammar have the same expansion.

For example, the following grammar is not irreducible:

$$\begin{aligned} S &\rightarrow ABCAB \\ A &\rightarrow xy \\ B &\rightarrow Az \\ C &\rightarrow xyz \end{aligned}$$

Note that (1) the adjacent pair of symbols AB appears twice without overlap, (2) the nonterminal C appears only once on the right, (3) the symbols B and C have the same expansion. In contrast, the following grammar generates the same string, but is irreducible:

$$\begin{aligned} S &\rightarrow CCC \\ C &\rightarrow xyz \end{aligned}$$

Note that the pair of adjacent symbols CC does appear twice, but these two instances overlap.

In upper-bounding the approximation ratio of SEQUENTIAL, we rely on properties of irreducible grammars established in the following two lemmas.

Lemma 12 *The sum of the lengths of the expansions of all distinct nonterminals in an irreducible grammar is at most* $2n$*.*

(This result also appears as equation 9.33 in Appendix B of [17].)

Proof. Let S be the start symbol of an irreducible grammar for a string of length n, and let $R_1, \ldots, R_k$ be the secondary nonterminals. Observe that the sum of the expansion lengths of all symbols on the left side of the grammar must be equal to the sum of the expansion lengths of all symbols on the right side of the grammar. Furthermore, every secondary nonterminal appears at least twice on the right side by property (2) of irreducible grammars. Therefore, we have:

$$[S] + [R_1] + \ldots + [R_k] \quad \geq \quad 2([R_1] + \ldots + [R_k])$$

Adding $[S] - [R_1] + \ldots + [R_k]$ to both sides of this inequality gives:

$$2[S] \quad \geq \quad [S] + [R_1] + \ldots + [R_k]$$

The length of the expansion of the start symbol, $[S]$, is equal to n by definition, and so we have

$$2n \quad \geq \quad [S] + [R_1] + \ldots + [R_k]$$

as claimed. $\square$

Lemma 13 *Every irreducible grammar of size* m *contains at least* $m/3$ *distinct, nonoverlapping pairs of adjacent symbols.*

Proof. For each rule, group the first and second symbols on the right to form one pair, the third and fourth for a second pair, and so forth. If a rule has an odd number of symbols, ignore the last one. Each pair is unique by property (1) of irreducible grammars. The right side of every rule must have length at least two as a consequence of property (3) of irreducible grammars. Therefore, at most $m/3$ symbols are ignored, leaving at least $2m/3$ symbols, which are grouped into at least $m/3$ pairs. $\square$

Upper Bound

Now we upper bound the approximation ratio of SEQUENTIAL by using the fact that SEQUENTIAL always produces an irreducible grammar and showing that no irreducible grammar is too far from optimal.

Theorem 14 *Every irreducible grammar for a string is $O((n/\log n)^{3/4})$ times larger than the size of the smallest grammar for that string.*

Corollary 15 *The approximation ratio of* SEQUENTIAL *is $O((n/\log n)^{3/4})$.*

Proof. (of Theorem 14) Let σ be a string of length n. Let m be the size of an irreducible grammar generating σ, and let m^* be the size of the smallest grammar.

Identify $m/3$ distinct, nonoverlapping pairs of adjacent symbols in the irreducible grammar. These are guaranteed to exist by Lemma 13. As preliminary observation, note that only a limited number of these pairs can expand to the same length-k substring of σ. The first nonterminal in each such pair must expand to a string with length between 1 and $k-1$. Thus, if there are k or more such pairs, then there must exist two pairs UV and XY such that $[U] = [X]$ and $[V] = [Y]$. Since all pairs are distinct, either $U \neq X$ or $V \neq Y$. In either case, we have two distinct symbols with the same expansion, which violates property (3) of irreducible grammars. Therefore, at most $k-1$ pairs can expand to the same length-k substring of σ

List all $m/3$ pairs in order of increasing expansion length. Group the first $1 \cdot 2m^*$ of these pairs, the next $2 \cdot 3m^*$, the next $3 \cdot 4m^*$, and so forth. Let g be the number of complete groups of nonterminals that can be formed in this way. Then we have:

$$1 \cdot 2m^* + 2 \cdot 3m^* + \ldots + (g+1) \cdot (g+2)m^* \quad > \quad m/3$$

And so $m = O(g^3 m^*)$.

Lemma 4 implies that σ contains at most m^*k distinct substrings of length k. The preliminary observation says that there can be at most $k-1$ pairs that expand to a given length-k substring. Therefore, at most $m^*k(k-1)$ pairs have an expansion of length k. For example, at most $1 \cdot 2m^*$ pairs expand to a string of length 2, at most $2 \cdot 3m^*$ pairs expand to a string of length 3, etc. Consequently, each pair in the first group expands to a string of length at least 2, each pair in the second group expands to a string of length at least 3, and generally each pair in the j-th group expands to a string of length at least $j+1$. Thus, the total length of the expansions of all pairs is at least:

$$1 \cdot 2^2 m^* + 2 \cdot 3^2 m^* + \ldots + g(g+1)^2 m^*$$

The $m/3$ pairs constitute a subset of the symbols on the right side of the grammar. The total expansion length of all symbols on the right side of the grammar is equal to the total expansion length of all symbols on the left. Lemma 12 upper bounds the latter quantity by $2n$. Therefore, we have:

$$1 \cdot 2^2 m^* + 2 \cdot 3^2 m^* + \ldots + g(g+1)^2 m^* \quad \leq \quad 2n$$

As a result, $g = O\left((n/m^*)^{1/4}\right)$. Substituting this bound on g into the bound on m obtained previously implies:

$$\begin{aligned} m &= O((n/m^*)^{3/4} m^*) \\ &= O((n/\log n)^{3/4} m^*) \end{aligned}$$

The second equality follows from Lemma 1, which says that $m^* = \Omega(\log n)$. $\square$

Filling in The Gap

There is a sizable gap between our upper bound on the approximation ratio of SEQUENTIAL (roughly $O(n^{3/4})$ and our lower bound (roughly $\Omega(n^{1/3})$). The following theorem shows that we can not significantly improve the upper bound using only properties of irreducible grammars.

Theorem 16 *There exist irreducible grammars $\Omega(n^{2/3}/\log n)$ times larger than the smallest grammar for the same string.*

Proof. Consider the following grammar for a string of x's of length $n = \Theta(k^3)$:

$$\begin{array}{lcl} S & \rightarrow & R_1R_1 \quad R_1R_2 \quad R_1R_3 \quad \ldots \quad R_1R_k \\ & & R_2R_2 \quad R_2R_3 \quad \ldots \quad R_2R_k \\ & & \qquad\qquad\qquad\qquad \ldots \\ & & \qquad\qquad\qquad\qquad\qquad R_{k-1}R_k \\ R_1 & \rightarrow & xx \\ R_2 & \rightarrow & R_1x \\ R_3 & \rightarrow & R_2x \\ & \ldots & \\ R_k & \rightarrow & R_{k-1}x \end{array}$$

This grammar is irreducible and has size $\Omega(k^2) = \Omega(n^{2/3})$. However, Lemma 3 implies that there is a grammar of size $O(\log k) = O(\log n)$, and the claim follows. $\square$

4.5 Global Algorithms

The remaining algorithms analyzed in this chapter all belong to a single class, which we refer to as *global algorithms*. We upper bound the approximation ratio of every global algorithm by $O\left((n/\log n)^{2/3}\right)$ with a single theorem. However, our lower bounds are all different, complex, and weak; all are $o(\log n)$. Thus, it may be that every global algorithm has an excellent approximation ratio. But, at present, we are unable to fully analyze any of them. Because they are so natural and our understanding is so incomplete, global algorithms are one of the most interesting topics related to the smallest grammar problem that could use further research.

The Procedure

A global algorithm begins with the grammar $S \rightarrow \sigma$. The remaining work is divided into rounds. During each round, one selects a *maximal string* γ. (The way global algorithms differ is in how a maximal string is selected in each round.) A maximal string γ is a string with three properties.

- It has length at least two.
- It appears at least twice on the right side of the grammar without overlap.
- No longer string appears as many times on the right side without overlap.

After a maximal string γ is selected, a new rule $T \rightarrow \gamma$ is added to the grammar. This rule is applied by working left-to-right through the right side of every other rule. Each time the string γ is encountered, it is replaced by the symbol T. The algorithm terminates when no more rounds are possible; that is, when no more maximal strings exist.

An example illustrates the range of moves available to a global algorithm. Suppose that the input string is $\sigma = abcabcabcabcaba$. We initially create the grammar

$$S \rightarrow abc\ abc\ abc\ abc\ ab\ a$$

where spaces are added for clarity. The maximal strings are ab, abc, and $abcabc$. In contrast, the string a is not maximal, because it does not have length at least two. The string $abcabcabc$ is not maximal because there are not two nonoverlapping instances. The string bc is not maximal because the longer string abc appears as many times without overlap. Suppose that we select the maximal string ab, and introduce the rule $T \to ab$. The grammar becomes:

$$\begin{array}{rcl} S & \to & Tc\,Tc\,Tc\,Tc\,T\ a \\ T & \to & ab \end{array}$$

Now the maximal strings are Tc and $TcTc$. Suppose that we select $TcTc$ and introduce the rule $U \to TcTc$. Then we obtain the grammar:

$$\begin{array}{rcl} S & \to & U\ U\ T\ a \\ T & \to & ab \\ U & \to & Tc\,Tc \end{array}$$

Now the only maximal string is Tc. Adding the rule $V \to Tc$ yields:

$$\begin{array}{rcl} S & \to & U\ U\ T\ a \\ T & \to & ab \\ U & \to & V\ V \\ V & \to & Tc \end{array}$$

No maximal strings remain, so we are done.

Upper Bound

One theorem employing a now-familiar argument bounds the approximation ratio of every global algorithm by $O((n/\log n)^{2/3})$. The argument relies on a series of lemmas saying that a grammar produced by a global algorithm is always well-conditioned. In particular, such a grammar is not only irreducible, but also has the additional property guaranteed by the following lemma.

Lemma 17 *The following invariant holds for the grammar maintained during the execution of a global algorithm. Let α and β be strings of length at least two on the right side of the grammar. If $\langle\alpha\rangle = \langle\beta\rangle$, then $\alpha = \beta$.*

Proof. The invariant holds trivially for the initial grammar $S \rightarrow \sigma$. So suppose that the invariants hold for grammar G, and then grammar G' is generated from G by introducing a new rule $T \rightarrow \gamma$. Let α' and β' be strings of length at least two on the right side of G' such that $\langle\alpha'\rangle = \langle\beta'\rangle$. We must show that $\alpha' = \beta'$. There are two cases to consider.

First, suppose that neither α' nor β' appears in γ. Then α' and β' must be obtained from nonoverlapping strings α and β in G such that $\langle\alpha\rangle = \langle\alpha'\rangle$ and $\langle\beta\rangle = \langle\beta'\rangle$. Since the invariant holds for G, we have $\alpha = \beta$. But then α and β are transformed the same way when the rule $T \rightarrow \gamma$ is added; that is, corresponding instances of the string γ within α and β are replaced by the nonterminal T. Therefore, $\alpha' = \beta'$.

Otherwise, suppose that at least one of α' or β' appears in γ. Then neither α' nor β' can contain T. Therefore, both α' and β' appear in grammar G, where the invariant holds, and so $\alpha' = \beta'$ again. $\square$

The remaining lemmas are dedicated to the proposition that grammars produced by a global algorithm are irreducible.

Lemma 18 *The following invariants hold for the grammar maintained during the execution of a global algorithm.*

1. *Every secondary nonterminal appears at least twice on the right side of the grammar.*

2. Not every instance of a nonterminal is followed by the same symbol.

3. Not every instance of a nonterminal is preceded by the same symbol.

Proof. All three invariants hold vacuously for the initial grammar $S \rightarrow \sigma$. Suppose that the invariant holds for a grammar G, and then we obtain a new grammar G' by introducing the rule $T \rightarrow \gamma$, where γ is a maximal string. Observe that all three invariants hold in G' with respect to the new nonterminal T due to the maximality of γ. What remains is to check that the invariants still hold with respect to each nonterminal U that also appeared in G.

First, we show that U appears at least twice in G'. Note that U can not appear only on the right side of the new rule $T \rightarrow \gamma$, since that would imply that the symbol preceding or following U in γ always preceded or followed it in G as well, violating invariant 2 or 3. Therefore, if U appears in γ, it also appears outside of γ, which means that it appears twice. If U does not appear in γ at all, then every instance of U in G remains in G', and so it still appears at least twice.

Next, we show that U is not always followed by the same symbol in G'. We argue by contradiction: suppose that U is always followed by the same symbol. If that symbol is T, then U was always followed by the first symbol of γ in G, violating invariant 2. If that symbol is not T, then U was always followed by that symbol in G as well, violating invariant 2 again. A symmetric argument shows that U is not always preceded by the same symbol either. $\square$

Lemma 19 *The following invariants hold for every secondary rule $T \rightarrow \gamma$ in a grammar maintained during the execution of a global algorithm:*

1. The string γ appears nowhere else in the grammar.

2. The length of γ is at least two.

Proof. The invariants hold trivially for the initial grammar $S \rightarrow \sigma$. Suppose that the invariants hold for every rule in a grammar G, and then we obtain a new grammar G' by introducing the rule $U \rightarrow \delta$.

First, we check that the invariants hold for the new rule. The string δ can not appear elsewhere in the grammar; such an instance would have been replaced by the nonterminal U. Furthermore, the length of δ is at least two, since δ is a maximal string.

Next, we check that the invariant holds for each rule $T \to \gamma'$ in G' that corresponds to a rule $T \to \gamma$ in G. If γ' does not contain U, then both invariants carry over from G. Suppose that γ' does contain U. The first invariant still carries over from G. The second invariant holds unless $\delta = \gamma$. However, since δ is a maximal string, that would imply that γ appeared at least twice in G, violating the first invariant. $\square$

Lemma 20 *A grammar produced by a global algorithm on an input string of length at least two is irreducible.*

Proof. We must show that a grammar produced by a global algorithm satisfies the three properties of an irreducible grammar. First, all nonoverlapping pairs of adjacent symbols on the right side are distinct; a global algorithm does not terminate until this condition holds. Second, every secondary nonterminal appears at least twice on the right side by Lemma 18. Third, no two symbols have the same expansion. The start symbol can not expand to a terminal, since the grammar generates a string of length at least two. No secondary nonterminal can expand to a terminal, because Lemma 19 implies that each secondary nonterminal has an expansion of length at least two. No two nonterminals can expand to the same string either; their definitions have length at least two by Lemma 19, and therefore their expansions are distinct by Lemma 17. $\square$

Theorem 21 *The approximation ratio of every global algorithm is* $O\left((n/\log n)^{2/3}\right)$.

Proof. Suppose that on input σ of length n, a global algorithm outputs a grammar G of size m, but the smallest grammar has size m^*.

Identify $m/3$ distinct, nonoverlapping pairs of adjacent symbols in G, which are guaranteed to exist by Lemma 13. List all these pairs in order of increasing expansion length. Group the first $2m^*$ pairs, the next $3m^*$, the next $4m^*$, and so forth. Suppose that g complete groups can be formed in this way. Then the largest complete group has size $(g+1)m^*$. Therefore, we have:

$$2m^* + 3m^* + \ldots + (g+2)m^* \quad > \quad m/3$$

And so $m = O(g^2 m^*)$.

Lemma 17 implies that every pair expands to a distinct substring of σ, and Lemma 4 says that σ has at most m^*k distinct substrings of length k. Therefore, at most $2m^*$ pairs expand to a string of length 2, at most $3m^*$ expand to a string of length 3, etc. Consequently, every pair in the j-th group expands to a string of length at least $j+1$. The total length of the expansions of all pairs must be at least:

$$2^2 m^* + 3^2 m^* + \ldots + (g+1)^2 m^*$$

Since grammar G is irreducible by Lemma 20, this quantity is upper bounded by $2n$ according to Lemma 12. This implies that $g = O\left((n/m^*)^{1/3}\right)$. Substituting this bound on g into the upper bound on m gives:

$$\begin{aligned} m &= O((n/m^*)^{2/3} m^*) \\ &= O((n/\log n)^{2/3} m^*) \end{aligned}$$

As usual, the second equality follows from Lemma 1, which says that $m^* = \Omega(\log n)$. $\square$

In the following sections, we describe three natural global algorithms. The preceding theorem provides an upper bound on the approximation ratio for all of them. Below, we establish a weak lower bound on the approximation ratio for each one individually. The arguments here are quite complicated.

4.5.1 Longest Match

Kieffer and Yang [17] proposed the Longest Match procedure, a global algorithm in which one always selects the *longest* maximal string. For example, given the initial grammar

$$S \rightarrow abc\ abc\ abc\ abc\ ab\ a$$

the first rule added is $T \rightarrow abc\ abc$. The resulting grammar is:

$$\begin{aligned} S &\rightarrow T\ T\ ab\ a \\ T &\rightarrow abc\ abc \end{aligned}$$

Tools for Analysis

Longest Match has two elegant features that simplify analysis of its behavior:

1. No rule is ever introduced with a nonterminal on the right side.

2. Each nonterminal created appears in the final grammar.

If the first principle were violated and a rule with a nonterminal on the right were introduced, then the definition of that nonterminal could not have been the longest maximal string when it was created, which contradicts the definition of the algorithm. The second principle follows from the first; since every new rule has only terminals on the right, nonterminals are only added to the grammar over the course the procedure and never eliminated.

The usefulness of the second principle is more readily explained. It can be used to lower bound the size of a grammar generated by Longest Match; we need only sum up the number of nonterminals created over the course of the procedure.

The first principle allows one to simplify the grammar maintained during the execution of Longest Match in a particular way without altering the subsequent behavior of the algorithm. Specifically, the same sequence of maximal strings is selected and the same number of nonterminals is introduced afterward whether or not we carry out this simplification.

Here is how the simplification is done. During the execution of Longest Match, we can replace each nonterminal on the right by a unique symbol. This does not alter subsequent behavior, since no rule containing a nonterminal will ever be introduced anyway. The example grammar from the start of this section can be transformed in this way into the following:

$$\begin{aligned} S &\rightarrow \mid\mid ab\ a \\ T &\rightarrow abc\ abc \end{aligned}$$

Furthermore, we can append the definitions of secondary rules to the start rule, separated by unique symbols, and then delete all secondary rules. In the example, we would obtain:

$$S \rightarrow \mid\mid ab\ a \mid abc\ abc$$

Finally, we can delete unique symbols at the start and end of this rule and merge consecutive unique symbols. Transforming the example in this way gives:

$$S \rightarrow ab\ a \mid abc\ abc$$

We refer to this three-step simplification procedure as *consolidating* a grammar. In analyzing the behavior of Longest Match on an input string, we are free to consolidate the grammar at any point to simplify analysis; the subsequent behavior of the procedure is unchanged.

Lower Bound

Theorem 22 *The approximation ratio of* LONGEST MATCH *is* $\Omega(\log \log n)$.

Proof. We analyze the performance of LONGEST MATCH on the string σ_k, which is largely a concatenation of substrings of

$$\gamma_k \quad = \quad x\ y^2\ x^4\ y^8\ x^{16}\ y^{32}\ x^{64}\ \ldots\ x^{2^{k-2}}\ y^{2^{k-1}}$$

separated by unique symbols. Index the k terms of γ_k from 0 to $k-1$. Now the string σ_k contains one substring of γ_k for each i in the range 0 to $k-1$. Specifically, the i-th substring starts with term i of γ_k and contains 2^j consecutive terms, where j is as large as possible. In addition, σ_k has the suffix $\mid x^{2^k} \mid y^{2^k}$. For example, σ_{10} is depicted below with indentation and line breaks to clarify the structure.

$$\sigma_{10} \quad = \quad \begin{array}{cccccccccccc}
x & y^2 & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & & & \mid & \\
& y^2 & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & x^{256} & & \mid & \\
& & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & x^{256} & y^{512} & \mid & \\
& & & y^8 & x^{16} & y^{32} & x^{64} & & & & \mid & \\
& & & & x^{16} & y^{32} & x^{64} & y^{128} & & & \mid & \\
& & & & & y^{32} & x^{64} & y^{128} & x^{256} & & \mid & \\
& & & & & & x^{64} & y^{128} & x^{256} & y^{512} & \mid & \\
& & & & & & & y^{128} & x^{256} & & \mid & \\
& & & & & & & & x^{256} & y^{512} & \mid & \\
& & & & & & & & & y^{512} & \mid & x^{1024} \mid y^{1024}
\end{array}$$

Note that, for example, σ_{10} contains a substring of γ_{10} that begins with term $i = 3$

and contains $2^j = 2^2 = 4$ consecutive terms. (It can not contain $2^3 = 8$ consecutive terms of γ_{10}, because γ_{10} does not contain that many terms.) This substring appears on the fourth line above. Also, σ_{10} has the suffix $x^{1024} \mid y^{1024}$

Now we must determine what LONGEST MATCH does when given this string as input. Define a *segment* to be a substring that is bounded on both sides by unique symbols or an end of the string. For example, in the string σ_{10}, one segment is $x^{256}y^{512}$, and there are twelve segments in all. In these terms, the longest match in σ_k is the second longest segment that contains the last term of γ_k. This segment is wholly contained in the longest segment containing the last term of γ_k. Thus, in the example, the longest match is:

$$x^{64}\ y^{128}\ x^{256}\ y^{512}$$

In the next round, the longest match is the third longest segment containing the last term of γ_k, which is wholly contained by the second longest such segment. In the round after, the longest match is the fourth longest segment containing the last term of γ_k, and so forth. After $\log k$ rounds of this type, the next two longest matches are $x^{2^{k-1}}$ and $y^{2^{k-1}}$. After introducing rules corresponding to all these matches, we obtain the grammar:

$$
\begin{array}{lllllllllllll}
S & \to & x & y^2 & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & & & | \\
 & & & y^2 & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & x^{256} & & | \\
 & & & & x^4 & y^8 & x^{16} & y^{32} & T_1 & & & & | \\
 & & & & & y^8 & x^{16} & y^{32} & x^{64} & & & & | \\
 & & & & & & x^{16} & y^{32} & x^{64} & y^{128} & & & | \\
 & & & & & & & y^{32} & x^{64} & y^{128} & x^{256} & & | \\
 & & & & & & & & T_1 & & & & | \\
 & & & & & & & & & y^{128} & x^{256} & & | \\
 & & & & & & & & & & T_2 & & | \\
 & & & & & & & & & & & T_3 & | \quad T_3T_3 \mid T_4T_4 \\
T_1 & \to & & & & & & & x^{64} & y^{128} & T_2 & & \\
T_2 & \to & & & & & & & & & x^{256} & T_3 & \\
T_3 & \to & & & & & & & & & & x^{512} & \\
T_4 & \to & & & & & & & & & & y^{512} &
\end{array}
$$

Now we consolidate this grammar to obtain:

$$
\begin{array}{llllllllllll}
S_2 & \to & x & y^2 & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & & | \\
 & & & y^2 & x^4 & y^8 & x^{16} & y^{32} & x^{64} & y^{128} & x^{256} & | \\
 & & & & x^4 & y^8 & x^{16} & y^{32} & & & & | \\
 & & & & & y^8 & x^{16} & y^{32} & x^{64} & & & | \\
 & & & & & & x^{16} & y^{32} & x^{64} & y^{128} & & | \\
 & & & & & & & y^{32} & x^{64} & y^{128} & x^{256} & | \\
 & & & & & & & & x^{64} & y^{128} & & | \\
 & & & & & & & & & y^{128} & x^{256} & | \\
 & & & & & & & & & & x^{256} & | \quad x^{512} \mid y^{512}
\end{array}
$$

The critical observation here is that the consolidated grammar is the initial grammar for input string σ_9. After another succession of rounds and a consolidation, the definition of the start rule becomes σ_8, and then σ_7, and so forth. Reducing the right side of the start rule from σ_i to σ_{i-1} entails the creation of at least $\log i$ nonterminals. Since nonterminals created by LONGEST MATCH are never eliminated, we can lower bound the total size of the grammar produced on this input by:

$$\sum_{i=1}^{k} \log i \quad = \quad \Omega(k \log k)$$

On the other hand, there exists a grammar of size $O(k)$ that generates σ_k. What follows is a sketch of the construction. First, we create nonterminals X^{2^i} and Y^{2^i} with expansions x^{2^i} and y^{2^i} respectively for all i up to k. We can define each such nonterminal using two symbols, and so only $O(k)$ symbols are required in total.

Then we define a nonterminal corresponding to each segment of σ_k. We define these nonterminals in batches, where a batch consists of all nonterminals corresponding to segments of σ_k that contain the same number of terms. Rather than describe the general procedure, we illustrate it with an example. Suppose that we want to define nonterminals corresponding to the following batch of segments in σ_{10}.

$$\begin{array}{ccccccc}
x^4 & y^8 & x^{16} & y^{32} & & & \\
 & y^8 & x^{16} & y^{32} & x^{64} & & \\
 & & x^{16} & y^{32} & x^{64} & y^{128} & \\
 & & & y^{32} & x^{64} & y^{128} & x^{256}
\end{array}$$

This is done by defining the following auxiliary nonterminals, which expand to prefixes and suffixes of the string $x^4 y^8 x^{16} y^{32} x^{64} y^{128} x^{256}$:

$$\begin{array}{rclcrcl} P_1 & \to & Y^{32} & \quad & S_1 & \to & X^{64} \\ P_2 & \to & X^{16}P_1 & & S_2 & \to & S_1Y^{128} \\ P_3 & \to & Y^{8}P_2 & & S_3 & \to & S_2Y^{256} \\ P_4 & \to & X^{4}P_3 & & & & \end{array}$$

Now we can define nonterminals corresponding to the desired substrings of γ_k in terms of these "prefix" and "suffix" nonterminals as follows:

$$\begin{array}{rcl} G_1 & \to & P_4 \\ G_2 & \to & P_3S_1 \\ G_3 & \to & P_2S_2 \\ G_4 & \to & P_1S_3 \end{array}$$

In this way, each nonterminal corresponding to a substring of γ_k in σ_k is defined using a constant number of symbols. Therefore, defining all k such nonterminals requires $O(k)$ symbols. We complete the grammar for σ_k by defining a start rule containing another $O(k)$ symbols. Thus, the total size of the grammar is $O(k)$.

Therefore, the approximation ratio for LONGEST MATCH is $\Omega(\log k)$. Since the length of σ_k is $n = \Theta(k2^k)$, this ratio is $\Omega(\log \log n)$ as claimed. $\square$

4.5.2 GREEDY

Apostolico and Lonardi [1, 2, 3] proposed a variety of greedy algorithms for grammar-based data compression. The central idea, which we analyze here, is to select the maximal string that reduces the size of the grammar as much as possible. For example, given the starting grammar

$$S \rightarrow abc\ abc\ abc\ abc\ ab\ a$$

the first rule added is $T \rightarrow abc$, since this decreases the size of the grammar by 5 symbols, which is the best possible:

$$\begin{aligned} S &\rightarrow T\ T\ T\ T\ ab\ a \\ T &\rightarrow abc \end{aligned}$$

Theorem 23 *The approximation ratio of* GREEDY *is at least* $\frac{5 \log 3}{3 \log 5} = 1.137\ldots$.

Proof. We consider the behavior of GREEDY on an input string of the form $\sigma_k = x^n$, where $n = 5^{2^k}$.

Greedy begins with the grammar $S \rightarrow \sigma_k$. The first rule added must be of the form $T \rightarrow x^t$. The size of the grammar after this rule is added is then:

$$t + \lfloor n/t \rfloor + (n \bmod t)$$

The first term reflects the cost of defining T, the second accounts for the instances of T itself, and the third represents extraneous x's. This sum is minimized when $t = n^{1/2}$. The resulting grammar is:

$$\begin{aligned} S &\rightarrow T^{5^{2^{k-1}}} \\ T &\rightarrow x^{5^{2^{k-1}}} \end{aligned}$$

Since the definitions of S and T contain no common symbols, we can analyze the behavior of GREEDY on each independently. However, each of these two subproblems is of the same form as the original, but of size $k-1$ instead of k. Thus, after two more rules are added, the grammar becomes:

$$\begin{array}{rclcrcl} S & \rightarrow & U^{5^{2^{k-2}}} & \quad & U & \rightarrow & T^{5^{2^{k-2}}} \\ T & \rightarrow & V^{5^{2^{k-2}}} & \quad & V & \rightarrow & x^{5^{2^{k-2}}} \end{array}$$

Continuing in this way, we reach a grammar with 2^k nonterminals, each defined by five copies of another symbol. Each such rule is transformed as shown below in a final step that does not alter the size of the grammar.

$$X \rightarrow YYYYY \qquad \Rightarrow \qquad \begin{array}{rcl} X & \rightarrow & X'X'Y \\ X' & \rightarrow & YY \end{array}$$

Therefore, GREEDY generates a grammar for σ_k of size 52^k.

On the other hand, we show that for all n, x^n has a grammar of size $3\log_3(n) + o(\log n)$. Substituting $n = 5^{2^k}$ then proves the theorem. Regard n as a numeral in a base $b = 3^j$, where j is a parameter defined later:

$$n \;=\; d_0 b^t + d_1 b^{t-1} + d_2 b^{t-2} + \ldots + d_{t-1} b^1 + d_t$$

The grammar is constructed as follows. First, create a nonterminal T_i with expansion x^i for each i between 0 and $b-1$. This can be done with $2 \cdot b = 2 \cdot 3^j$ symbols, using rules of the form $T_{i+1} \rightarrow T_i x$. Next, create a nonterminal U_0 with expansion x^{d_0}, using the rule $U_0 \rightarrow T_{d_0}$. Create a nonterminal U_1 with expansion $x^{d_0 b + d_1}$ by tripling U_0, tripling the result, and so on j times and then appending T_{d_1}:

$$
\begin{aligned}
Z_1 &\rightarrow U_0U_0U_0 \\
Z_2 &\rightarrow Z_1Z_1Z_1 \\
Z_3 &\rightarrow Z_3Z_3Z_3 \\
&\ldots \\
Z_j &\rightarrow Z_{j-1}Z_{j-1}Z_{j-1} \\
U_1 &\rightarrow Z_jT_{d_1}
\end{aligned}
$$

This requires $3j+2$ symbols. Similarly, create U_2 with expansion $x^{d_0b^2+d_1b+d_2}$, and so on. The start symbol of the grammar is U_t. The total number of symbols used is at most:

$$
\begin{aligned}
2 \cdot 3^j + (3j+2) \cdot t &= 2 \cdot 3^j + (3j+2) \cdot \log_b n \\
&= 2 \cdot 3^j + 3\log_3 n + \frac{2}{j} \log_3 n
\end{aligned}
$$

The second equality uses the fact that $b = 3^j$. Setting $j = \frac{1}{2}\log_3 \log_3 n$ makes the last expression $3\log_3(n) + o(\log n)$ as claimed. $\square$

4.5.3 Re-Pair

Larsson and Moffat [24] proposed the Re-Pair algorithm. (The byte-pair encoding (BPE) technique of Gage [13] is based on similar ideas.) Essentially, this is a global algorithm in which one always selects the maximal string that appears most often. For example, given the starting grammar

$$
S \rightarrow abc\ abc\ abc\ abc\ ab\ a
$$

the first rule added is $T \to ab$, since the maximal string ab appears most often.

There is a small difference between the algorithm originally proposed by Larsson and Moffat and what we refer to here as Re-Pair: the original algorithm always makes a rule for the pair of symbols that appears most often without overlap, regardless of whether that pair forms a maximal string. For example, on input $xyzxyz$, the original algorithm generates the following grammar:

$$\begin{aligned} S &\to UU \\ U &\to xV \\ V &\to yz \end{aligned}$$

This is unattractive, since one could replace the single occurrence of the nonterminal V by its definition and obtain a smaller grammar. Indeed, Re-Pair, as described here, would give the smaller grammar:

$$\begin{aligned} S &\to UU \\ U &\to xyz \end{aligned}$$

The original approach was motivated by implementation efficiency issues.

Theorem 24 *The approximation ratio of* Re-Pair *is* $\Omega(\sqrt{\log n})$.

Proof. Consider the performance of Re-Pair on input strings of the form:

$$\sigma_k = \prod_{w=\sqrt{k}}^{2\sqrt{k}} \prod_{i=0}^{w-1} x^{b_{w,i}} \mid$$

where $b_{w,i}$ is an integer that, when regarded as a k-bit binary number, has a 1 at each position j such that $j \equiv i \pmod{w}$. (Position 0 corresponds to the least significant bit.) On such an input, RE-PAIR creates rules for strings of x's with lengths that are powers of two:

$$
\begin{aligned}
X_1 &\rightarrow xx \\
X_2 &\rightarrow X_1 X_1 \\
X_3 &\rightarrow X_2 X_2 \\
X_4 &\rightarrow X_3 X_3 \\
&\cdots
\end{aligned}
$$

At this point, each run of x's with length $b_{w,i}$ in σ_k is represented using one nonterminal for each 1 in the binary representation of $b_{w,i}$. For example, the beginning of σ_{16} and the beginning of the resulting start rule are listed below:

$$
\begin{array}{rclcrcl}
\sigma_{16} & = & x^{0001000100010001} \mid & & S & \rightarrow & X_{12}X_8X_4x \mid \\
& & x^{0010001000100010} \mid & & & & X_{13}X_9X_5X_1 \mid \\
& & x^{0100010001000100} \mid & & & & X_{14}X_{10}X_6X_2 \mid \\
& & x^{1000100010001000} \mid & & & & X_{15}X_{11}X_7X_3 \mid \\
& & x^{1000010000100001} \mid & \Rightarrow & & & X_{15}X_{10}X_5x \mid \\
& & x^{0000100001000010} \mid & & & & X_{11}X_6X_1 \mid \\
& & x^{0001000010000100} \mid & & & & X_{12}X_7X_2 \mid \\
& & x^{0010000100001000} \mid & & & & X_{13}X_8X_3 \mid \\
& & x^{0100001000010000} \mid \ldots & & & & X_{14}X_9X_4 \mid \ldots
\end{array}
$$

Note that no other rules are introduced, because each pair of adjacent symbols now appears only once. RE-PAIR encodes each string of x's using $\Omega(\sqrt{k})$ symbols. Since there are $\Omega(k)$ such strings, the size of the grammar produced is $\Omega(k^{3/2})$.

On the other hand, there exists a grammar of size $O(k)$ that generates σ_k. First, we create a nonterminal X_j with expansion x^{2^j} for all j up to $k-1$. Then for each w

we create a nonterminal $B_{w,0}$ for $x^{b_{w,0}}$ using $O(\sqrt{k})$ of the X_j nonterminals, just as RE-PAIR does. However, we can then define a nonterminal for each remaining string of x's using only two symbols:

$$
\begin{array}{rcl}
B_{w,1} & \rightarrow & B_{w,0}B_{w,0} \\
B_{w,2} & \rightarrow & B_{w,1}B_{w,1} \\
B_{w,3} & \rightarrow & B_{w,2}B_{w,2} \\
 & \cdots &
\end{array}
$$

In total, we use $O(k)$ symbols to define the nonterminals X_j. Then we use $O(\sqrt{k})$ symbols defining $B_{w,0}$ for each of the $\sqrt{k}$ different values of w. The remaining $O(k)$ nonterminals $B_{w,i}$ cost two symbols each to define. Finally, we expend $O(k)$ symbols on a start rule, which consists of all the $B_{w,i}$ separated by unique symbols. In total, the grammar size is $O(k)$ as claimed.

To complete the argument, note that $n = |\sigma_k| = \Theta(\sqrt{k}2^k)$, and so the approximation ratio is no better than $\Omega(\sqrt{k}) = \Omega(\sqrt{\log n})$. $\square$

Chapter 5

New Algorithms

In this chapter, we present a simple $O(\log^3 n)$ approximation algorithm for the smallest grammar problem. We then give a more complex algorithm with approximation ratio $O(\log n/m^*)$ based on an entirely different approach.

5.1 An $O(\log^3 n)$ Approximation Algorithm

We begin with a simple algorithm with approximation ratio $O(\log^3 n)$. As preliminaries, we describe a useful grammatical construction, prove one lemma, and cite an old result that we shall need.

5.1.1 Preliminaries

The *substring construction* generates a set of grammar rules enabling each substring of a string $\eta = x_1 \ldots x_p$ to be expressed with at most two symbols.

The construction works as follows. First, create a nonterminal for each suffix of the string $x_1 \ldots x_k$ and each prefix of $x_{k+1} \ldots x_p$, where $k = \lceil \frac{p}{2} \rceil$. Note that each such nonterminal can be defined using only two symbols: the nonterminal for the next shorter suffix or prefix together with one symbol x_i. Repeat this construction recursively on the two halves of the original string, $x_1 \ldots x_k$ and $x_{k+1} \ldots x_p$. The recursion terminates when a string of length one is obtained. This recursion has $\log p$

levels, and p nonterminals are defined at each level. Since each definition contains at most two symbols, the total cost of the construction is at most $2p \log p$.

Now we show that every substring $\alpha = x_i \dots x_j$ of η is equal to $\langle AB \rangle$, where A and B are nonterminals defined in the construction. There are two cases to consider. If α appears entirely within the left-half of η or entirely within the right-half, then we can obtain A and B from the recursive construction on $x_1 \dots x_k$ or $x_{k+1} \dots x_p$. Otherwise, let $k = \lceil \frac{p}{2} \rceil$ as before, and let A be the nonterminal for $x_i \dots x_k$, and let B be the nonterminal for $x_{k+1} \dots x_j$.

For example, the substring construction for the string $\eta = abcdefgh$ is given below:

$$
\begin{array}{rcl@{\qquad}rcl}
C_1 & \rightarrow & d & D_1 & \rightarrow & e \\
C_2 & \rightarrow & cC_1 & D_2 & \rightarrow & D_1 f \\
C_3 & \rightarrow & bC_2 & D_3 & \rightarrow & D_2 g \\
C_4 & \rightarrow & aC_3 & D_4 & \rightarrow & D_3 h
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl@{\qquad}rcl}
E_1 & \rightarrow & b & F_1 & \rightarrow & c & G_1 & \rightarrow & f & H_1 & \rightarrow & g \\
E_2 & \rightarrow & aE_1 & F_2 & \rightarrow & F_1 d & G_2 & \rightarrow & eG_1 & H_2 & \rightarrow & H_1 h
\end{array}
$$

With these rules defined, each substring of $abcdefgh$ is expressible with at most two symbols. As examples, consider:

$$
\begin{array}{rcl}
abc & = & \langle E_2 F_1 \rangle \\
defg & = & \langle C_1 D_3 \rangle \\
bcdefgh & = & \langle C_3 D_4 \rangle
\end{array}
$$

We now turn to the lemma, which upper bounds the complexity of a string generated by a small grammar in a new way.

Lemma 25 *Let σ be a string generated by a grammar of size m. Then there exists a string β_k of length at most $2mk$ that contains every length-k substring of σ.*

Proof. We can construct β_k by concatenating all strings obtained as follows from rules $T \to \alpha$ in the grammar of size m:

1. For each terminal in α, take the length-k substring of $\langle T \rangle$ beginning at that terminal.

2. For each nonterminal in α, take the length-$(2k-1)$ substring of $\langle T \rangle$ consisting of the last character in the expansion of that nonterminal, the preceding $k-1$ characters, and the following $k-1$ characters.

In both cases, we permit the substrings to be shorter if they are truncated by the start or end of $\langle T \rangle$.

Now we establish the correctness of this construction. First, note that the string β_k is a concatenation of at most m strings of length at most $2k$, giving a total length of at most $2mk$ as claimed. Next, let γ be a length-k substring of σ. Consider the rule $T \to \alpha$ such that $\langle T \rangle$ contains γ and is as short as possible. Either γ begins at a terminal of α, in which case it is a string of type 1, or else it begins inside the expansion of a nonterminal in α and ends beyond, in which case it is contained in a string of type 2. (Note that γ can not be wholly contained in the expansion of a nonterminal in α; otherwise, we would have selected that nonterminal for consideration instead of T.) In either case, γ is a substring of β_k as desired. $\square$

Our approximation algorithm for the smallest grammar problem makes use of Blum's 4-approximation for the shortest superstring problem [6]. In this procedure, we are given a collection of strings and want to find the shortest superstring; that is, the shortest string that contains each string in the collection as a substring. The procedure works greedily. At each step, find the two strings in the collection with largest overlap. Merge these two into a single string. (For example, *abaa* and *aaac* have overlap *aa* and thus can be merged to form *abaaac*.) Repeat this process until only one string remains. This is the desired superstring, and Blum proved that it is at most four times longer than the shortest superstring.

5.1.2 The Algorithm

Now we assemble our $O(\log^3 n)$-approximation algorithm for the smallest grammar problem.

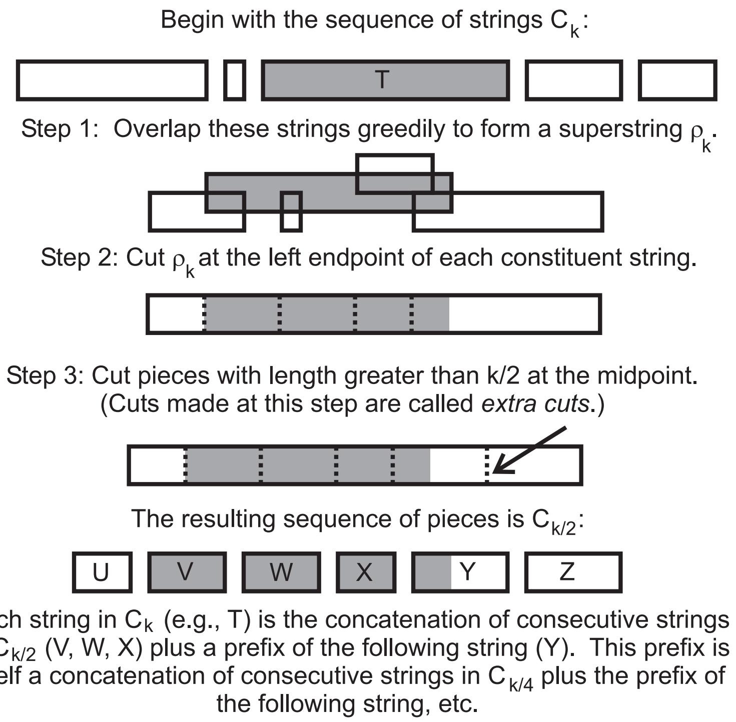
In this algorithm, the focus is on certain sequences of substrings of σ. In particular, we construct $\log n$ sequences C_n, $C_{n/2}$, $C_{n/4}$, $\ldots$, C_2, where the sequence C_k consists of some substrings of σ that have length at most k. These sequences are defined as follows. The sequence C_n is initialized to consist of only the string σ itself. In general, the sequence C_k generates the sequence $C_{k/2}$ via the following operations, which are illustrated in the figure that follows.

1. Use Blum's greedy 4-approximation algorithm to form a superstring ρ_k containing all the strings in C_k.

2. Now we are going to cut the superstring ρ_k into small pieces. First, determine where each string in C_k ended up inside ρ_k, and then cut ρ_k at the left endpoints of those strings.

3. Cut each piece of ρ_k that has length greater than $k/2$ at the midpoint. During the analysis, we shall refer to the cuts made during this step as *extra cuts*.

The sequence $C_{k/2}$ is defined to be the sequence of pieces of ρ_k generated by this three-step process. By the nature of Blum's algorithm, no piece of ρ_k can have length greater than k after step 2, and so no piece can have length greater than $k/2$ after step 3. Thus, $C_{k/2}$ is a sequence of substrings of σ that have length at most $k/2$ as desired.



Now we translate these sequences of strings into a grammar. To begin, associate a nonterminal with each string in each sequence C_k. In particular, the nonterminal associated with the single string in C_n (which is σ itself) is the start symbol of the grammar.

All that remains is to define these nonterminals. In doing so, the following observation is key: each string in C_k is the concatenation of several consecutive strings in $C_{k/2}$ together with a prefix of the next string in $C_{k/2}$. This is illustrated in the figure above, where the fate of one string in C_k (shaded and marked T) is traced through the construction of $C_{k/2}$. In this case, T is the concatenation of V, W, X, and a prefix of Y. Similarly, the prefix of Y is itself the concatenation of consecutive strings in $C_{k/4}$ together with a prefix of the next string in $C_{k/4}$. This prefix is in turn the concatenation of consecutive strings in $C_{k/8}$ together with a prefix of the next string in $C_{k/8}$, etc. As a result, we can define the nonterminal corresponding to a string in C_k as a sequence of consecutive nonterminals from $C_{k/2}$, followed by consecutive nonterminals from $C_{k/4}$, followed by consecutive nonterminals from $C_{k/8}$, etc. For example, the definition of T would begin $T \to VWX \ldots$ and then contain sequences of consecutive nonterminals from $C_{k/4}$, $C_{k/8}$, etc. As a special case, the nonterminals corresponding to strings in C_2 can be defined in terms of terminals.

We can use the substring construction to make these definitions shorter and hence the overall size of the grammar smaller. In particular, for each sequence of strings C_k, we apply the substring construction on the corresponding sequence of nonterminals. This enables us to express any sequence of consecutive nonterminals using just two symbols. As a result, we can define each nonterminal corresponding to a string in C_k using only two symbols that represent a sequence of consecutive nonterminals from $C_{k/2}$, two more that represent a sequence of consecutive nonterminals from $C_{k/4}$, etc. Thus, every nonterminal can now be defined with $O(\log n)$ symbols on the right.

Theorem 26 *The procedure described above is an $O(\log^3 n)$-approximation algorithm for the smallest grammar problem.*

Proof. We must determine the size of the grammar generated by the above procedure. In order to do this, we must first upper bound the number of strings in each sequence C_k. To this end, note that the number of strings in $C_{k/2}$ is equal to the number of strings in C_k plus the number of extra cuts made in step 3. Thus, given that C_n contains a single string, we can upper bound the number of strings in C_k by upper bounding the number of extra cuts made at each stage.

Suppose that the smallest grammar generating σ has size m^*. Then Lemma 25 implies that there exists a superstring containing all the strings in C_k with length $2m^*k$. Since we are using a 4-approximation, the length of ρ_k is at most $8m^*k$. Therefore, there can be at most $16m^*$ pieces of ρ_k with length greater than $k/2$ after step 2. This upper bounds the number of extra cuts made in the formation of $C_{k/2}$, since extra cuts are only made into pieces with length greater than $k/2$. It follows that every sequence of strings C_k has length $O(m^* \log n)$, since step 2 is repeated only $\log n$ times over the course of the algorithm.

On one hand, there are $\log n$ sequences C_k, each containing $O(m^* \log n)$ strings. Each such string corresponds to a nonterminal with a definition of length $O(\log n)$. This gives $O(m^* \log^3 n)$ symbols in total. On the other hand, for each sequence of strings C_k, we apply the substring construction on the corresponding sequence of nonterminals. Recall that this construction generates $2p \log p$ symbols when applied to a sequence of length p. This creates an additional

$$O((\log n) \cdot (m^* \log n) \log(m^* \log n)) \quad = \quad O(m^* \log^3 n)$$

symbols. Therefore, the total size of the grammar generated by this algorithm is $O(m^* \log^3 n)$, which proves the claim. $\square$

5.1.3 Aside: The Substring Problem

One aspect of the preceding algorithm for the smallest grammar problem raises another grammar-related question that is interesting in its own right: given a string $\sigma = x_1 \dots x_p$, construct a small grammar such that every substring of σ is expressible using few symbols. For example, the substring construction of Section 5.1.1 produces a grammar of size $O(p \log p)$ such that every substring of σ is expressible using two symbols.

Alternative solutions to this problem are possible. For example, we could proceed as follows. Partition $x_1 \dots x_p$ into $p/\log p$ segments consisting of $\log p$ symbols each. Associate a nonterminal Q_i with each such segment. Apply the earlier substring construction on the sequence $Q_1 \dots Q_{p/\log p}$. Then apply this new construction recursively on each segment of $\log p$ symbols. The depth of the recursion is $\log^* p$. As a result, the size of the grammar produced is at most $2p \log^* p$, and each substring of σ is expressible using $2 \log^* p$ symbols.

These two solutions to the substring problem are compared below:

	grammar size	symbols per substring
construction in Section 5.1.1	$2p \log p$	2
construction given here	$2p \log^* p$	$2 \log^* p$

Whether the infinitesimal $\log^* p$ terms are really necessary is unclear; in particular, does there exists a grammar of size $O(p)$ that enables each substring to be expressed using $O(1)$ symbols?

5.2 An $O(\log n/m^*)$-Approximation Algorithm

We now present a more complex solution to the smallest grammar problem with approximation ratio $O(\log n/m^*)$. 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 new 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, the variety of well-behaved grammars that our procedure employs. In the same section, 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 $O(\log n/m^*)$ times larger than the smallest. In a concluding section, we offer some perspectives on the algorithm.

5.2.1 An LZ77 Variant

We begin by describing a flavor of LZ77 compression [39]. We use this both to obtain a lower bound on the size of the smallest grammar for a string and as the basis for generating a small grammar. In this scheme, a string is represented by a sequence of characters and pairs of integers. For example, one possible sequence is:

$$a\ b\ (1,2)\ (2,3)\ c\ (1,5)$$

An LZ77 representation can be decoded into a string by working left-to-right through the sequence according to the following rules:

- If a character c is encountered in the sequence, then the next character in the string is c.

- Otherwise, if a pair (x, y) is encountered in the sequence, then the next y characters of the string are the same as the y characters beginning at position x of the string. (We require that the y characters beginning at position x be represented by earlier items in the sequence.)

The example sequence can be decoded as follows:

index	1	2	3	4	5	6	7	8	9	10	11	12	13
sequence	a	b	$(1,2)$		$(2,3)$			c	$(1,5)$				
string	a	b	a	b	b	a	b	c	a	b	a	b	b

The shortest LZ77 sequence for a given string can be found in polynomial time. Make a left-to-right pass through the string. If the next character in the unprocessed portion of the string has not appeared before, output it. Otherwise, find the longest prefix of the unprocessed portion that appears in the processed portion and output the pair (x, y) describing that previous appearance. It is easy to show (and well known) that this procedure finds the shortest LZ77 sequence.

The following lemma states that this procedure implies a lower bound on the size of the smallest grammar.

Lemma 27 *The length of the shortest* LZ77 *sequence for a string is a lower bound on the size of the smallest grammar for that string.*

Proof. Suppose that a string is generated by a grammar of size m^*. We can transform this grammar into an LZ77 sequence of length at most m^* as follows. Begin with the sequence of symbols on the right side of the start rule. Select the nonterminal with longest expansion. Replace the leftmost instance by its definition and replace each subsequent instance by a pair referring to the first instance. Repeat this process until no nonterminals remain. Note that each symbol on the right side of the original grammar corresponds to at most one item in the resulting sequence. This establishes the desired inequality. $\Box$

An example illustrates the method employed in the proof. Suppose we want to map the following grammar to an LZ77 sequence.

$$\begin{aligned} S &\rightarrow ABtAu \\ A &\rightarrow vBBw \\ B &\rightarrow xyz \end{aligned}$$

We effect the transformation as follows:

$$\begin{aligned} S &\Rightarrow A\ B\ t\ A\ u \\ &\Rightarrow v\ B\ B\ w\ B\ t\ (1,8)\ u \\ &\Rightarrow v\ x\ y\ z\ (2,3)\ w\ (2,3)\ t\ (1,8)\ u \end{aligned}$$

A somewhat similar process was described in [28].

Our $O(\log n/m^*)$-approximation algorithm essentially inverts this process, mapping an LZ77 sequence to a grammar. This other direction is much more involved.

5.2.2 Balanced Binary Grammars

In this section, we introduce the notion of a balanced binary grammar. The approximation algorithm we are developing works exclusively with this restricted class of well-behaved 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

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

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".

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

`AddPair`: Produces a balanced grammar containing a nonterminal with expansion $\langle XY \rangle$ from a balanced grammar containing nonterminals X and Y. The number 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 $\langle X_1 \dots X_t \rangle$ from a balanced grammar containing nonterminals $X_1 \dots X_t$. The number of rules added is:

$$O\left(t\left(1 + \log \frac{[X_1 \dots X_t]}{t}\right)\right)$$

`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.

The `AddPair` Operation

We are given a balanced grammar with nonterminals 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. The `AddPair` operation is divided into two phases.

In the first phase, we decompose Y into a string of symbols. 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.

The second phase is extremely intricate. The analysis runs for many pages, even though we omit some routine algebra. Initially, we create a new rule $Z_1 \rightarrow XY_1$ and declare this to be the *active rule*. The remainder of the second phase is divided into 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, completing the procedure.

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:

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

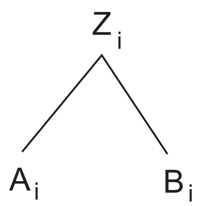
Since $\langle B_i \rangle$ is a substring of $\langle Y_1 \ldots Y_i \rangle$ by invariant 2, we can conclude:

$$\frac{\alpha}{1-\alpha} \leq \frac{[Y_{i+1}]}{[B_i]} \tag{5.1}$$

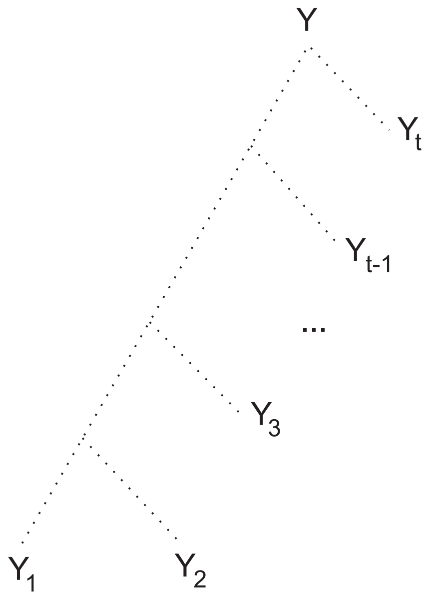
On the other hand, since $\langle Z_i \rangle$ is a superstring of $\langle Y_1 \ldots Y_i \rangle$ by invariant 1, we can conclude:

$$\frac{[Y_{i+1}]}{[Z_i]} \leq \frac{1-\alpha}{\alpha} \tag{5.2}$$

Each step in the second phase involves intricate grammar transformations. For clarity, we supplement the text with diagrams. In these diagrams, a rule $Z_i \to A_i B_i$ is indicated with a wedge:



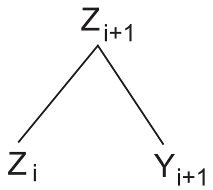
Preexisting rules are indicated with shaded lines, and new rules with dark lines. As a further example, the decomposition of XY into $XY_1 \ldots X_t$ during the first phase results from the grammar containing the following structure:



All that remains is to describe how each step of the second phase is carried out. At the start of the i-th step, the active rule is $Z_i \to A_iB_i$. 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:

$$Z_{i+1} \quad \to \quad Z_iY_{i+1}$$



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:

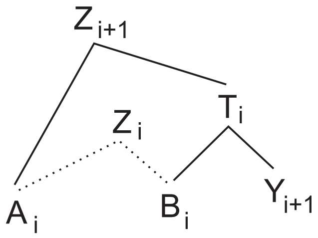
$$\frac{\alpha}{1-\alpha} \leq \frac{[Y_{i+1}]}{[Z_i]} \leq \frac{1-\alpha}{\alpha}$$

Since the right inequality is (5.2), the left inequality must be violated. Thus, hereafter we can assume:

$$\frac{\alpha}{1-\alpha} > \frac{[Y_{i+1}]}{[Z_i]} \tag{5.3}$$

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

$$\begin{aligned} Z_{i+1} &\rightarrow A_i T_i \\ T_i &\rightarrow B_i Y_{i+1} \end{aligned}$$



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 $Z_{i+1} \rightarrow A_i T_i$ is balanced by the case assumption. What remains is to show that the rule $T_i \rightarrow B_i Y_{i+1}$ 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 (5.1). For the right inequality, begin with (5.3):

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

The equality follows from the definition of Z_i by the rule $Z_i \rightarrow A_i B_i$. 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 $B_i Y_{i+1}$ are not in balance; that is, the following assertion is false:

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

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:

$$\frac{\alpha}{1-\alpha} > \frac{[A_i]}{[B_i Y_{i+1}]} \tag{5.4}$$

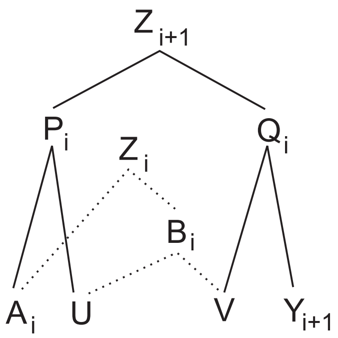
Combining inequalities (5.3) and (5.4), one can use algebraic manipulation to establish the following bounds, which hold hereafter:

$$\frac{[A_i]}{[B_i]} \leq \frac{\alpha}{1-2\alpha} \tag{5.5}$$

$$\frac{[Y_{i+1}]}{[B_i]} \leq \frac{\alpha}{1-2\alpha} \tag{5.6}$$

Case 3: Otherwise, suppose that B_i is defined by the rule $B_i \to UV$. We create three new rules:

$$\begin{aligned} Z_{i+1} &\to P_i Q_i \\ P_i &\to A_i U \\ Q_i &\to V Y_{i+1} \end{aligned}$$



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

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

The first inequality uses the fact that $B_i \to UV$ is balanced. The second inequality follows because $1 - \alpha \leq 1$. The final 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 $B_i \to UV$ is balanced, and the second follows from (5.5). The last inequality holds for all $\alpha < 0.293$.

Next, we show that $Q_i \to VY_{i+1}$ 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 $B_i \to UV$ is balanced, the second uses (5.6), and the third holds for all $\alpha < 0.293$. In the other direction, we have:

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

Again, the first inequality uses the fact that $B_i \to UV$ is balanced, the second uses $1 - \alpha < 1$, and the third is (5.1).

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 (5.5) on the top, and (5.1) 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 (5.6) 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 of the second phase. Therefore, it suffices to upper bound t. This quantity is determined during the first phase, where Y is decomposed into a string of symbols. In each step of the first phase, the length of the expansion of the first symbol in this string 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:

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

Because we take the absolute value, this bound holds regardless of whether $[X]$ or $[Y]$ is larger.

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 $\langle X_1 \ldots X_t \rangle$.

The idea is to place 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:

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

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

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

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 $c(t/2) \log \frac{s}{t/2}$, 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

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

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

$$\sum_{i=0}^{\log t} c(t/2^i)\log\frac{s}{t} + cti/2^i \quad = \quad O\left(t\left(1+\log\frac{[X_1 \ldots X_t]}{t}\right)\right)$$

as claimed.

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 *desired 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 desired suffix wholly contains $\langle B\rangle$, then we prepend B to the nonterminal sequence. The desired 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 desired 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:

$$\frac{[X_1 \ldots X_i]}{[X_{i+1}]} \leq \frac{1-\alpha}{\alpha} \tag{5.7}$$

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

In the second phase, we merge the nonterminals in the sequence $X_1 \ldots X_t$ 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 $\langle X_1 \ldots X_i \rangle$ and seek to merge in nonterminal X_{i+1}. There are two cases, distinguished by whether or not the following inequality holds:

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

- If so, then R_i and X_{i+1} are in balance. (Inequality (5.7) 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}. (It can not be too large, because of inequality (5.7).) We use `AddPair` to merge the two, which generates $O(1 + \log [X_{i+1}]/[R_i])$ new rules. Since $[R_i]$ is at most a constant times the size of its largest component, $[X_i]$, the number of new rules is $O(1 + \log [X_{i+1}]/[X_i])$.

Summing the number of rules created during this process gives:

$$\sum_{i=1}^{t} O\left(1 + \log\frac{[X_{i+1}]}{[X_i]}\right) = O(t + \log[X_t])$$
$$= O(\log|\beta|)$$

The second equality follows from the fact, observed previously, that $t = O(\log|\beta|)$ and from the fact that $\langle X_t \rangle$ is a substring of β. 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.

5.2.3 The Algorithm

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

We are given an input string σ. First, we apply the LZ77 variant described in Section 5.2.1. This gives a sequence $L_1 \ldots L_p$ of terminals and pairs. By Lemma 27, the length of this sequence is a lower bound on the size of the smallest grammar for σ; that is, $p \leq m^*$. Now we employ the tools of Section 5.2.2 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 terminal. In general, at the beginning of i-th step, the expansion of the active list is the string represented by $L_1 \ldots L_i$. 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 $L_1 \ldots L_{i+1}$.

If L_{i+1} is a terminal, we can accomplish this goal by simply appending L_{i+1} to the active list. If L_{i+1} is a pair, 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`. This produces $O(t_i(1 + \log |\beta_i|/t_i))$ rules.

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:

$$O\left(\sum_{i=1}^{p} t_i + t_i \log |\beta_i|/t_i + \log |\beta_i|\right) = O\left(\sum_{i=1}^{p} t_i + \sum_{i=1}^{p} t_i \log |\beta_i|/t_i + \sum_{i=1}^{p} \log |\beta_i|\right)$$

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 gives a bound of $2p$:

$$\sum_{i=1}^{p} t_i \leq 2p$$

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

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

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

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

The latter inequality uses the fact that $\sum_{i=1}^{p} |\beta_i| \leq n$ and that $\sum_{i=1}^{p} t_i \leq 2p$. Note that the function $x \log n/x$ is increasing for x up to n/e, and so this inequality holds only if $2p \leq n/e$. 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 p). 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:

$$p \log \frac{\sum |\beta_i|}{p} \;\leq\; p \log \frac{n}{p}$$

The total grammar size is therefore:

$$O\left(p \log \frac{n}{p}\right) \;=\; O\left(m^* \log \frac{n}{m^*}\right)$$

where we use the inequality $p \leq m^*$ and, again, the observation that $x \log n/x$ is increasing for $x < n/e$. This proves the claim.

5.3 Grammar-Based Compression versus LZ77

We have now shown that a grammar of size m can be translated into an LZ77 sequence of length at most m. In the reverse direction, we have shown that an LZ77 sequence of length p can be translated to a grammar of size $O(p \log n/p)$. Furthermore, the latter result is nearly the best possible. Consider strings of the form

$$\sigma \;=\; x^{k_1} \mid x^{k_2} \mid \ldots \mid x^{k_q}$$

where k_1 is the largest of the k_i. This string can be represented by an LZ77 sequence of length $O(q + \log k_1)$:

$$x\ (1,1)\ (1,2)\ (1,4)\ (1,8) \ldots (1, k_i - 2^j) \mid (1, k_2) \mid \ldots \mid (1, k_q)$$

Here, j is the largest power of 2 less than k_i. If we set $q = \Theta(\log k_1)$, then the sequence has length $O(\log k_1)$.

On the other hand, Theorem 6 states that the smallest grammar for σ is within a constant factor of the shortest addition chain containing $k_1, \ldots, k_q$. Pippinger [30] has shown, via a counting argument, that there exist integers $k_1, \ldots, k_q$ such that the shortest addition chain containing them all has length:

$$\Omega\left(\log k_1 + q \cdot \frac{\log k_1}{\log\log k_1 + \log q}\right)$$

If we choose $q = \Theta(\log k_1)$ as before, then the above expression boils down to:

$$\Omega\left(\frac{\log^2 k_1}{\log\log k_1}\right)$$

Putting this all together, we have a string σ of length $n = O(k_1 \log k_1)$ for which there exists an LZ77 sequence of length $O(\log k_1)$, but for which the smallest grammar has size $\Omega\left(\frac{\log^2 k_1}{\log\log k_1}\right)$. The ratio between the grammar size and the length of the LZ77 sequence is therefore:

$$\Omega\left(\frac{\log k_1}{\log\log k_1}\right) \quad = \quad \Omega\left(\frac{\log n}{\log\log n}\right)$$

Thus the $O(\log n)$ blowup given by our algorithm for mapping an LZ77 sequence to a grammar is essentially optimal. No significantly better approximation ratio can be achieved using the length of the LZ77 representation as a lower bound.

The analysis in this chapter suggests that the best grammar-based compressors and LZ77 should achieve roughly comparable compression performance. This is hardly surprising since the two representations are quite similar. Which approach achieves superior compression in practice depends on many considerations beyond the scope of our theoretical analysis. For example, one must bear in mind that a grammar symbol can be represented by fewer bits than an LZ77 pair. In particular, each LZ77 pair requires about $2\log n$ bits to encode, although this may be somewhat reduced by representing the integers in each pair with a variable-length code. On the other hand, each grammar symbol can be naively encoded using about $\log m$ bits, which could be as small as $\log\log n$. This can be further improved via Huffman or arithmetic encoding. Thus, the fact that grammars are typically somewhat larger than LZ77 sequences is roughly offset by the fact that grammars translate better to bits. Empirical comparisons are emerging, but do not yet seem definitive one way or the other [16, 10, 27, 24, 3, 1].

The procedures presented here are not ready for immediate use as practical compression algorithms. The numerous hacks and optimizations needed in practice are lacking. Furthermore, as noted in Section 4.1, our evaluation criterion, approximation ratio, reveals little about performance on high-entropy strings, such as ordinary English text. (In fact, recall that if the input string σ is compressible by less than a factor of $2e$, which is about the limit for ASCII text, our best algorithm outputs the trivial grammar $S \to \sigma$.) Furthermore, our algorithms are designed not for practical performance, but for good, *analyzable* performance. In practice, the best grammar-based compression algorithm may yet prove to be a simple scheme like RE-PAIR, which we do not yet know how to analyze.

Chapter 6

Future Directions

This chapter discusses some interesting open problems related to the smallest grammar problem.

Analysis of Global Algorithms

Our analysis of previously-proposed algorithms for the smallest grammar problem leaves a large gap of understanding surrounding the global algorithms, Greedy, Longest Match, and Re-Pair. In each case, we upper bound the approximation ratio by $O((n/\log n)^{2/3})$ and lower bound it by some expression that is $o(\log n)$. Elimination of this gap would be significant for several reasons. First, these algorithms are important; they are simple enough to be practical for applications such as compression and DNA entropy estimation. Second, there are natural analogues to these global algorithms for other hierarchically-structured problems. Thus, a deeper understanding of them in the context of the smallest grammar problem may have wider ramifications for hierarchical optimization in general. Third, all of our lower bounds on the approximation ratio for these algorithms are well below the $\Omega(\log n/\log\log n)$ hardness implied by the reduction from the addition chain problem. Either there exist worse examples for these algorithms or else a tight analysis will yield progress on the addition chain problem. Either resolution would be interesting.

Approximation Algorithms for Addition Chains

As noted in Chapter 3, the addition chain problem has been studied extensively, but not from the perspective of approximation algorithms. Yao's algorithm [38] approximates the shortest addition chain containing $k_1, \ldots, k_p$ to within $O(\log n / \log \log n)$, but makes no use of special relationships between the integers k_i. Progress on this problem seems possible and would be interesting for at least two reasons. First, the approximation ratio for the smallest grammar problem achieved in Chapter 5, $O(\log n/m^*)$, can not be significantly improved without progress on this problem. Second, the addition chain problem has considerable interest in its own right. The need to efficiently exponentiate in cryptographic applications provides some motivation, but the problem also has intrinsic interest; much work on the problem was done before the cryptographic application appeared.

Monomial Evaluation

Pippenger [30] considered the problem of evaluating a set of monomials over several variables using the minimum number of multiplications. For example, how many multiplications are needed to compute $x^8y^3z^5$, $x^{23}y^9z^{10}$, and x^8y^{12}? This can be regarded as a vectorial version of the addition chain problem. In the example, we must find the shortest addition chain beginning with the vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ and containing the vectors $(8, 3, 5)$, $(23, 9, 10)$, and $(8, 12, 0)$. Pippenger's elegant paper demonstrates results for this general problem analogous to Yao's for the single-variable problem. Thus, the door is open to an approximation algorithm.

This can also be regarded as a variant of the smallest grammar problem. Now one is given several input strings instead of just one. In the example, the input strings would be $x^8y^3z^5$, $x^{23}y^9z^{10}$, and x^8y^{12}. The goal is roughly to create a grammar containing nonterminals that expand to these strings. However, we disregard the order of symbols within a string. For example, we now regard the strings $xxyz$ and $zxyx$ as equivalent. Note that global algorithms for the smallest grammar problem have natural analogues in this domain. For example, in analogy to Re-Pair one might always create a nonterminal for the pair of symbols that appears most frequently, now disregarding the order of symbols within a string.

Even the special case of this problem where every variable has degree at most one is interesting. It is equivalent to the *smallest AND-circuit* problem. A digital circuit has several input signals and several output signals. Each output is required to be the logical AND of a specified subset of the inputs. How many two-input AND gates must the circuit contain to satisfy the specification?

Algebraic Extraction

The *algebraic extraction problem* is of great practical importance and points out the need for a better understanding of hierarchical approximation problems beyond the smallest grammar problem. It can be regarded as a generalization of the smallest AND-circuit problem. As before, a digital circuit has several inputs and several outputs. Now, however, the function of each output is a specified sum-of-products over the input signals. The basic problem is to find the smallest circuit satisfying this specification. However, to simplify matters, one regards each distinct literal in the sums-of-products as a distinct real-valued variable. Logical OR is regarded as addition, logical AND is regarded as multiplication, and only transformations that are valid over the reals are admissible. For example, $ab + a\overline{c} = a(b + \overline{c})$ is a valid transformation, but $a\overline{b} + ab = a$ is not.

This problem has been studied extensively in the context of automated circuit design. (In practice, some transformations that do not correspond to identities over the reals are sometimes allowed in order to enhance performance.) Interestingly, the best known algorithms for this problem are closely analogous to the GREEDY and RE-PAIR algorithms for the smallest grammar problem. (For details on these analogues, see [8, 7] and [31] respectively.) No approximation guarantees are known. This underlines the need for a better understanding of global algorithms.

String Complexity in Other Natural Models

One motivation for studying the smallest grammar problem was to shed light on a computable and approximable variant of Kolomogorov complexity. This raises a natural follow-on question: can the complexity of a string be approximated in 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 $T(P) \rightarrow PP$, and write the string $xxyzyz$ as $T(x)T(yz)$. Presumably as model power increases, approximability decays to uncomputability. Good approximation algorithms for strong string-representation models could be applied wherever the smallest grammar problem has arisen: data compression, deducing linguistic structures, revealing patterns in DNA, etc.

Bibliography

[1] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *IEEE Data Compression Conference, DCC*, pages 119–128, March 1998.

[2] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *IEEE Data Compression Conference, DCC*, pages 143–152, March 2000.

[3] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, November 2000.

[4] Piotr Berman and Marek Karpinski. On some tighter inapproximability results, further improvements. Technical Report TR98-065, Electronic Colloquium on Computational Complexity, 1998.

[5] Daniel Bleichenbacher. *Efficiency and Security of Cryptosystems Based on Number Theory.* PhD thesis, Universitat Bern, 1996.

[6] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. In *Symposium on Theory of Computing*, pages 328–336, 1991.

[7] R. Brayon, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangle covering problem. In *International Conference on Computer Aided Design*, pages 66–69, November 1987.

[8] R. K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*, pages 49–54, 1982.

[9] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[10] Carl G. de Marcken. *Unsupervised Language Acquisition*. PhD thesis, MIT, 1996.

[11] Peter Downey, Benton Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, August 1981.

[12] Paul Erdos. Remarks on number theory III. *ACTA Arithmetica*, VI:77–81, 1960.

[13] P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

[14] Takuya Kida, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. A unifying framework for compressed pattern matching. In *International Symposium on String Processing and Information Retrieval*, pages 89–96, 1999.

[15] J. C. Kieffer, P. Flajolet, and E.-H. Yang. Data compression via binary decision diagrams. In *IEEE International Symposium on Information Theory*, 2000.

[16] John C. Kieffer and En hui Yang. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform– part one: Without context models. *IEEE Transactions on Information Theory*, IT-46(3):755–777, May 2000.

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

[18] John C. Kieffer, En hui Yang, Gregory J. Nelson, and Pamela Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, IT-46(5):1227–1245, July 2000.

[19] Donald E. Knuth. *Seminumerical Algorithms*. Addison-Wesley, 1981.

[20] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, pages 1–7, 1965.

[21] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 2000.

[22] Chung-Hung Lai and Tien-Fu Chen. Compressing inverted files in scalable information systems by binary decision diagram encoding. In *Supercomputing*, 2001.

[23] J. Kevin Lanctot, Ming Li, and En hui Yang. Estimating DNA sequence entropy. In *Symposium on Discrete Algorithms*, pages 409–418, 2000.

[24] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. *Proceedings of the IEEE*, 88:1722–1732, November 2000.

[25] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-23(1):75–81, January 1976.

[26] G. Nelson, J. Kieffer, and P. Cosman. An interesting hierarchical lossless data compression algorithm. In *IEEE Information Theory Society Workshop*, 1995.

[27] Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, 1996.

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

[29] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence*, 7:67–82, 1997.

[30] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, May 1980.

[31] J. Rajski and J. Vasudevamurthy. The testability-preserving concurrent decomposition and factorization of boolean expressions. *IEEE Transactions on Computer-Aided Design*, June 1992.

[32] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte pair encoding: A text compression scheme that accelerates pattern matching. Technical Report DOI-TR-CS-161, Department of Informatics, Kyushu University, April 1999.

[33] James A. Storer. *Data Compression: Methods and Complexity.* PhD thesis, Princeton University, 1978.

[34] James A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[35] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[36] Edward G. Thurber. Efficient generation of minimal length addition chains. *SIAM Journal on Computing*, 28(4):1247–1263, 1999.

[37] Terry A. Welch. A technique for high-performance data compression. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 17(6):8–19, 1984.

[38] Andrew Chi-Chih Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, March 1976.

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

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