Article

The Smallest Grammar Problem as Constituents Choice and Minimal Grammar Parsing

Rafael Carrascosa [1,*]**, François Coste** [2]**, Matthias Gallé** [2,*] **and Gabriel Infante-Lopez** [1]

[1] Grupo de Procesamiento de Lenguaje Natural Universidad Nacional de Córdoba/Consejo Nacional de Investigaciones Científicas y Técnicas, Argentina

[2] Symbiose Project, IRISA/INRIA Rennes-Bretagne Atlantique, France

[*] Authors to whom correspondence should be addressed; E-Mails: rafacarrascosa@gmail.com (R.C.); mgalle@gmail.com (M.G.).

Abstract: The smallest grammar problem—namely, finding a smallest context-free grammar that generates exactly one sequence—is of practical and theoretical importance in fields such as Kolmogorov complexity, data compression and pattern discovery. We propose a new perspective on this problem by splitting it into two tasks: (1) choosing which words will be the constituents of the grammar and (2) searching for the smallest grammar given this set of constituents. We show how to solve the second task in polynomial time parsing longer constituent with smaller ones. We propose new algorithms based on classical practical algorithms that use this optimization to find small grammars. Our algorithms consistently find smaller grammars on a classical benchmark reducing the size in 10% in some cases. Moreover, our formulation allows us to define interesting bounds on the number of small grammars and to empirically compare different grammars of small size.

Keywords: smallest grammar problem; hierarchical structure inference; optimal parsing; data discovery

1. Introduction

The smallest grammar problem—namely, finding a smallest context-free grammar that generates exactly one sequence—is of practical and theoretical importance in fields such as Kolmogorov complexity, data compression and pattern discovery.

The size of a smallest grammar can be considered a computable variant of Kolmogorov complexity, in which the Turing machine description of the sequence is restricted to context-free grammars. The problem is then decidable, but still hard: the problem of finding a smallest grammar with an approximation ratio smaller than $\frac{8569}{8568}$ is NP-HARD [1]. Nevertheless, a $\mathcal{O}(\log^3 n)$ approximation ratio—with n the length of the sequence—can be achieved by a simple algorithmic scheme based on an approximation to the shortest superstring problem [1] and a smaller $\mathcal{O}(\log n/g)$ (where g is the size of a smallest grammar) approximation ratio is possible through more complex mappings from the LZ77-factorization of the sequence to a context-free grammar with a balanced parsing tree [1,2].

If the grammar is small, storing the grammar instead of the sequence can be interesting from a data compression perspective. Kieffer and Yang developed the formal framework of compression by *Grammar Based Codes* from the viewpoint of information theory, defining irreducible grammars and demonstrating their universality [3]. Before this formalization, several algorithms allowing to compress a sequence by context-free grammars had already been proposed. The LZ78-factorization introduced by Ziv and Lempel in [4] can be interpreted as a context-free grammar. Let us remark that this is not true for LZ77, published one year before [5]. Moreover, it is a commonly used result that the size of a LZ77-factorization is a lower bound on the size of a smallest grammar [1,2]. The first approach that generated explicitly a context-free grammar with compression ability is *Sequitur* [6]. Like LZ77 and LZ78, Sequitur is an on-line algorithm that processes the sequence from left to right. It incrementally maintains a grammar generating the part of the sequence read, introducing and deleting production to ensure that no digram (pair of adjacent symbols) occurs more than once and that each rule is used at least twice. Other algorithms consider the entire sequence before choosing which repeated substring will be rewritten by the introduction of a new rule. Most of these offline algorithms proceed in a greedy manner, selecting in each iteration one repeated word w according to a score function and replacing all the (non-overlapping) occurrences of the repeat w in the whole grammar by a new terminal N and adding the new production $N \rightarrow w$ to the grammar. Different heuristics have been used to choose the repeat: the most frequent one [7], the longest [8] and the one that reduces the most the size of the resulting grammar (COMPRESSIVE [9]). GREEDY [10] belongs to this last family but the score used for choosing the words is oriented toward directly optimizing the number of bits needed to encode the grammar rather than minimizing its size. The running time of Sequitur is linear and linear-time implementations of the first two algorithms exists: REPAIR [11] and LONGEST FIRST [12], while the existence of a linear-time algorithm for COMPRESSIVE and GREEDY remains an open question.

In pattern discovery, a smallest grammar is a good candidate for being the one that generates the data according to Occam's razor principle. In that case, the grammar may not only be used for compressing the sequence but also to unveil its structure. Inference of the hierarchical structure of sequences was the initial motivation of Sequitur and has been the subject of several papers applying this scheme to DNA sequences [6,13,14], musical scores [15] or natural languages [7,16]. It can also be a first step to learn more general grammars along the lines of [17]. In all the latter cases, a slight difference in the size of the grammar, which would not matter for data compression, can dramatically change the results with respect to the structure. Thus, more sophisticated algorithms than those for data compression are needed.

In this article, which is an extended version of [18], we focus on how to choose occurrences that are going to be rewritten. This issue was generally handled straightforwardly in former algorithms by

selecting *all* the non-overlapping occurrences in a left to right order. Moreover, once an occurrence had been chosen for being rewritten, the result was definitive and was not altered by the words chosen in the following iterations. In order to remedy these flaws, we show how to globally optimize the choice of the occurrences to be replaced by non-terminals. We are then able to improve classical greedy algorithms by introducing this optimization step at each iteration of the algorithm. This optimization allows us to define the smallest grammar problem in terms of two complementary optimization problems: the choice of non-terminals and the choice of their occurrences. We redefine the search space, prove that all solutions are contained in it and propose a new algorithm performing a wider search by adding the possibility to discard non-terminals previously included in the grammar. Thanks to our new formulation, we are able to analyze the number of different grammars with the same minimal size and present empirical results that measure the conservation of structure among them.

The outline of this paper is the following: in Section 2 we formally introduce the definitions and in Section 3 the classical offline algorithms. Section 4 contains our main contributions. In Section 4.1 we show how to optimize the choice of occurrences to be replaced by non-terminals for a set of words and then extend offline algorithms by optimizing the choice of the occurrences at each step in Section 4.2. We present our search space and show that this optimization can also be used directly to guide the search in a new algorithm in Section 4.3. We present experiments on a classical benchmark in Section 5 showing that the occurrence optimization consistently allows to find smaller grammars. In Section 6 we consider the number of smallest grammars that may exist and discuss the consequences of our results on structure discovery.

2. Previous work and definitions

2.1. Definitions and Notation

We start by giving a few definitions and setting up the nomenclature that we use along the paper. A *string* s is a sequence of characters $s_1 \ldots s_n$, its length, $|s| = n$. ϵ denotes the empty string, and $s[i : j] = s_i \ldots s_j$, $s[i : j] = \epsilon$ if $j < i$. We say that a word w occurs at position i, if $w = s[i : i + |w| - 1]$. w is a repeat of s if it occurs more than once in s and $|w| > 1$. We denote by *repeats*(s) the set of substrings of s that occur more than once and of length greater than one. For example *repeats*$(abaaab) = \{ab, aa\}$ and aa occurs at position 2 and 3.

A *context-free grammar* is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{P}, S \rangle$, where Σ is the finite set of *terminals* and $\mathcal{N}$ the finite set of *non-terminals*, $\mathcal{N}$ and Σ disjoint. $S \in \mathcal{N}$ is called the *start symbol* and $\mathcal{P}$ is the set of *productions*. Each production (also called *rule*) is of the form $A \to \alpha$ where its *left-hand side* A is a non-terminal and its *right-hand side* α belongs to $(\Sigma \cup \mathcal{N})^*$. We say $\alpha \stackrel{1}{\Rightarrow} \beta$, if α is of the form $\delta C \delta'$, $\beta = \delta \gamma \delta'$ and $C \to \gamma$ is a production. A sequence $\alpha \stackrel{1}{\Rightarrow} \alpha_1 \stackrel{1}{\Rightarrow} \ldots \stackrel{1}{\Rightarrow} \beta$ is called a *derivation* and in this case we say that α *produces* β and that β *derives* from α (denoted by $\alpha \Rightarrow \beta$).

Given a string α over $(\mathcal{N} \cup \Sigma)^*$, its *constituents* (*cons*(α)) are the possible strings of terminals that can be derived from α. Formally, $cons(\alpha) = \{w \in \Sigma^* : \alpha \Rightarrow w\}$. The constituents of a grammar are all the constituents of its non-terminals. The *language* is the set of constituents of the start symbol S, $cons(S)$.

Because the Smallest Grammar framework seeks for a context-free grammar whose language contains one and only one string, the grammars we consider here neither branch (every non-terminal occurs at

most once as a left-hand side of a rule) nor loop (if B occurs in any derivation starting with A, then A will not occur in a derivation starting with B). In this type of grammars, any substring of the grammar has a unique constituent, in which case we will drop the set notation and define $cons(\alpha)$ as the only terminal string that can be derived from α. Note that if the grammar is in Chomsky Normal Form [19], it is equivalent to a straight-line program.

Several definitions of the grammar size exist. Following [9], we define the *size of the grammar G*, denoted by $|G|$, to be the length of its encoding by concatenation of its right-hand sides separated by end-of-rule markers: $|G| = \sum_{A \to \alpha \in \mathcal{P}} (|\alpha| + 1)$.

3. IRR

3.1. General Scheme

Most offline algorithms follow the same general scheme. First, the grammar is initialized with a unique initial rule $S \to s$ where s is the input sequence and then they proceed iteratively. At each iteration, a word ω occurring more than once in s is chosen according to a score function f, all the (non-overlapping) occurrences of ω in the grammar are replaced by a new non-terminal N_ω and a new production $N_\omega \to \omega$ is added to the grammar. We give pseudo-code for this general scheme that we name *Iterative Repeat Replacement* (IRR) in Algorithm 1. There, $\mathcal{P}$ is the set of production rules being built: this defines a unique grammar $G(\mathcal{P})$ and therefore we define $|\mathcal{P}| = |G(\mathcal{P})|$. The set of repeats of size longer than one of the right-hand sides of $\mathcal{P}$ is denoted by *repeats*$(\mathcal{P})$ and $\mathcal{P}_{\omega \mapsto N}$ is the result of the substitution of ω by the new symbol N in the right-hand sides of $\mathcal{P}$ as detailed in the next paragraph.

When occurrences overlap, one has to specify which occurrences have to be replaced. One solution is to choose all the elements in the *canonical list of non-overlapping occurrences* of ω in s, which we define to be the list of non-overlapping occurrences of ω in a greedy left to right way (all occurrences overlapping with the first selected occurrence are not considered, then the same thing with the second non-eliminated occurrence, *etc.*). This ensures that a maximal number of occurrences will be replaced. When searching for the smallest grammar, one has to consider not only the occurrences of a word in s but also their occurrence in right-hand sides of rules that are currently part of the grammar. A canonical list of non-overlapping occurrences of ω can be defined for each right-hand side appearing in the set of production rules $\mathcal{P}$. This provides a straightforward list of occurrences used in the scoring function or the replacement step by our pseudo-code defining IRR.

The IRR schema instantiates different algorithms, depending on the score function $f(\omega, \mathcal{P})$ it uses. LONGEST FIRST corresponds to $f(\omega, \mathcal{P}) = f_{ML}(\omega, \mathcal{P}) = |\omega|$. Choosing the most frequent repeat, like in REPAIR, corresponds to use $f(\omega, \mathcal{P}) = f_{MF}(\omega, \mathcal{P}) = o_\mathcal{P}(\omega)$, where $o_\mathcal{P}(\omega)$ is the length of the canonical non-overlapping list of ω in the right-hand sides of rules in $\mathcal{P}$. Note however the difference that IRR is more general than REPAIR and may select a word which is not a digram.

In order to derive a score function corresponding to COMPRESSIVE, note that replacing a word ω by a non-terminal results in a contraction of the grammar of $(|\omega| - 1) * o_\mathcal{P}(\omega)$ and its inclusion in the grammar adds $|\omega| + 1$ to the grammar size. This defines $f(\omega, \mathcal{P}) = f_{MC}(\omega, \mathcal{P}) = (|\omega| - 1) * (o_\mathcal{P}(\omega) - 1) - 2$. We call these three algorithms IRR-ML (maximal length), IRR-MF (most frequent) and IRR-MC (maximal compression), respectively.

Algorithm 1 Iterative Repeat Replacement.

IRR(s, f)

Require: s is a sequence, and f is a score function

1: $\mathcal{P} \leftarrow \{N_s \rightarrow s\}$

2: **while** $\exists \omega \; : \; \omega \leftarrow \arg\max_{\alpha \in \text{repeats}(\mathcal{P})} f(\alpha, \mathcal{P}) \wedge |\mathcal{P}_{\omega \mapsto N_\omega}| < |\mathcal{P}|$ **do**

3: $\quad \mathcal{P} \leftarrow \mathcal{P}_{\omega \mapsto N_\omega} \cup \{N_\omega \rightarrow \omega\}$

4: **end while**

5: **return** $G(\mathcal{P})$

Figure 1. Grammars returned by classical algorithms on sequence $abcdabgeabceabcd\$$.

$$
\begin{aligned}
S &\rightarrow N_1\text{dabge}N_1\text{e}N_1\text{d}\$ \\
N_1 &\rightarrow \text{a b c}
\end{aligned}
\qquad
\begin{aligned}
S &\rightarrow N_2\text{d}N_1\text{ge}N_2\text{e}N_2\text{d}\$ \\
N_1 &\rightarrow \text{a b} \\
N_2 &\rightarrow N_1 \text{ c}
\end{aligned}
\qquad
\begin{aligned}
S &\rightarrow N_1\text{abge}N_2\text{e}N_1\$ \\
N_1 &\rightarrow N_2 \text{ d} \\
N_2 &\rightarrow \text{a b c}
\end{aligned}
$$

IRR-MC   IRR-MF   IRR-ML

$$
\begin{aligned}
S &\rightarrow N_1\text{d}N_2\text{g}N_3N_3\text{d}\$ \\
N_1 &\rightarrow N_2 \text{ c} \\
N_2 &\rightarrow \text{a b} \\
N_3 &\rightarrow \text{e } N_1
\end{aligned}
$$

$$
S \rightarrow N_1 N_2 N_3 N_4 N_5 N_6 N_7 N_8 N_9 N_{10} N_{11}
$$

$N_1 \rightarrow$ a		$N_7 \rightarrow$ e	
$N_2 \rightarrow$ b		$N_8 \rightarrow N_5$ c	
$N_3 \rightarrow$ c		$N_9 \rightarrow N_7$ a	
$N_4 \rightarrow$ d		$N_{10} \rightarrow N_2$ c	
$N_5 \rightarrow N_1$ b		$N_{11} \rightarrow$ d $\$$	
$N_6 \rightarrow$ g			

Sequitur   LZ78

The complexity of IRR when it uses one of these scores is $\mathcal{O}(n^3)$ since for a sequence of size n, the computation of the scores involving only $o_\mathcal{P}(\omega)$ and $|\omega|$ of the $\mathcal{O}(n^2)$ possible repeats can be done in $\mathcal{O}(n^2)$ using a suffix tree structure [20] and the number of iterations is bounded by n since the size of the grammar strictly decreases at each step.

The grammars found by the three IRR algorithms, plus Sequitur and LZ78 are shown on a small example in Figure 1. A comparison of the size of the grammars returned by these algorithms over a standard data compression corpus are presented in Section 5. These results confirm that IRR-MC finds the smallest grammars as was suggested in [9]. Until now, no other polynomial time algorithm (including theoretical algorithms that were designed to achieve a low approximation ratio [1,2,21]) has proven (theoretically nor empirically) to perform better than IRR-MC.

3.2. Limitations of IRR

Even though IRR algorithms are the best known practical algorithms for obtaining small grammars, they present some weaknesses. In the first place, their greedy strategy does not guarantee that the compression gain introduced by a selected word ω will still be interesting in the final grammar. Each time an iteration selects a substring of ω, the length of the production rule is reduced; and each time

a superstring is selected, its number of occurrences is reduced. Moreover, the first choices mark some breaking points and future words may appear inside them or in another parts of the grammar, but never span over these breaking points.

It could be argued that there may exist a score function that for every sequence scores the repeats in such a way that the order they are presented to IRR results in a smallest grammar. The following theorem proves that this is not the case.

Theorem 1. *There are infinitely many sequences s_k, over a constant alphabet, such that for any score function f, $|IRR(s_k, f)|$ is greater than the size of a smallest grammar for s_k.*

Proof. Consider the infinite set of sequences

$$x\alpha x\beta x\gamma x|_1 x\beta x\gamma x\alpha x|_2 x\gamma x\alpha x\beta x|_3 x\alpha x\gamma x\beta x|_4 x\beta x\alpha x\gamma x|_5 x\gamma x\beta x\alpha x|_6 x\alpha x|_7 x\beta x|_8 x\gamma x$$

where α, β and γ are any sequences of size greater than two, with pairwise disjoint alphabets not containing the symbol x. These sequences exploit the fact that IRR algorithms replace all possible occurrences of the selected word. Let us define $G^*(s_k)$ as the following grammar:

$$S \to AS_\beta C|_1 BS_\gamma A|_2 CS_\alpha B|_3 AS_\gamma B|_4 BS_\alpha C|_5 CS_\beta A|_6 A|_7 B|_8 C$$
$$A \to xS_\alpha x \quad B \to xS_\beta x \quad C \to xS_\gamma x$$

to which we add the rules of smallest grammars for α, β, γ with as start symbols $S_\alpha, S_\beta, S_\gamma$ respectively. $|G^*| = 42 + |G_{\min}(\alpha)| + |G_{\min}(\beta)| + |G_{\min}(\gamma)|$, where $G_{\min}(s)$ denotes a smallest grammar for sequence s. Note that no IRR algorithm could generate G^* and, moreover, by enumeration we find that the smallest possible grammar that can be obtained with an IRR algorithm has size $46 + |G_{\min}(\alpha)| + |G_{\min}(\beta)| + |G_{\min}(\gamma)|$. This grammar has the same production rules as G^*. In the case where α, β and γ are of size one (we then replace S_α by α and so on), this results in an approximation ratio of $\frac{46}{42} \approx 1.095$. This is a general lower bound for *any* IRR algorithm. □

Note that Theorem 1 does not make any assumptions over the possible score function f (like, for example, polynomial running time). In order to find G^*, the choice of occurrences that will be rewritten should be flexible when considering repeats introduced in future iterations.

4. Choice of Occurrences

4.1. Global Optimization of Occurrences Replacement

Once an IRR algorithm has chosen a repeated word ω, it replaces all non-overlapping occurrences of that word in the current grammar by a new non-terminal N and then adds $N \to \omega$ to the set of production rules. In this section, we propose to perform a global optimization of the replacement of occurrences, considering not only the last non-terminal but also all the previously introduced non-terminals. The idea is to allow occurrences of words to be kept (instead of being replaced by non-terminals) if replacing other occurrences of words overlapping them results in a better compression.

We propose to separate the choice of which terminal strings will be final constituents of the final grammar from the choice of which of the occurrences of these constituents will be replaced by

non-terminals. First, let us assume that a set of constituents $\{s\} \cup Q$ is given and we want to find a smallest grammar whose constituent set is $\{s\} \cup Q$. If we denote this set by $\{s = \omega_0, \omega_1, \ldots, \omega_m\}$, we need to be able to generate these constituents and for each constituent ω_i the grammar must thus have a non-terminal N_i such that $\omega_i = cons(N_i)$. In the smallest grammar problem, no unnecessary rule should be introduced since the grammar has to generate only one sequence. More precisely such a grammar must have exactly $m + 1$ non-terminals and associate production rules.

For this, we define a new problem, called *Minimal Grammar Parsing* (MGP) Problem. An instance of this problem is a set of strings $\{s\} \cup Q$, such that all strings of Q are substrings of s. A Minimal Grammar Parsing of $\{s\} \cup Q$ is a context-free grammar $G = \langle \Sigma, \mathcal{N}, \mathcal{P}, S \rangle$ such that:

1. all symbols of s are in Σ.

2. S derives only s.

3. for each string $\omega_\ell \{s\} \cup Q$ there is a non-terminal N_ℓ that derives only ω_ℓ and $\mathcal{N} = \{N_0 \ldots N_m\}$.

4. for each non-terminal N there is a string s' of $\{s\} \cup Q$ such that N derives s'.

5. $|G|$ is of minimal size for all possible grammars that satisfy conditions 1–4.

Note that this is similar to the Smallest Grammar Problem, except that all constituents for the non-terminals of the grammar are also given. The MGP problem is related to the problem of static dictionary parsing [22] with the difference that the dictionary also has to be parsed. This recursive approach is partly what makes grammars interesting to both compression and structure discovery.

As an example consider the sequence $s = ababbababbabaabbabaa$ and suppose the constituents are $\{s, abbaba, bab\}$. This defines the set of non-terminals $\{N_0, N_1, N_2\}$, such that $\mathrm{cons}(N_0) = s$, $\mathrm{cons}(N_1) = abbaba$ and $\mathrm{cons}(N_2) = bab$. A minimal parsing is $N_0 \rightarrow aN_2N_2N_1N_1a$, and $N_1 \rightarrow abN_2a$, $N_2 \rightarrow bab$.

This problem can be solved in a classical way in polynomial time by searching for a shortest path in $|Q| + 1$ graphs as follows. Given the set of constituents, $s = \{\omega_0, \omega_1, \ldots, \omega_m\}$.

1. Let $\{N_0, N_1, \ldots, N_m\}$ be the set of non-terminals. Each N_ℓ will be the non-terminal whose constituent is ω_ℓ.

2. Define $m + 1$ directed acyclic graphs $\Gamma_0 \ldots \Gamma_m$, where $\Gamma_\ell = \langle M_\ell, E_\ell \rangle$. If $|\omega_\ell| = k$ then the graph Γ_ℓ will have $k + 1$ nodes: $M_\ell = \{1 \ldots |\omega_\ell| + 1\}$. The edges E_l are of two types:

    (a) for every node i there is an edge to node $i + 1$ labeled with $\omega_\ell[i]$.

    (b) there will be an edge from node i to $j + 1$ labeled by N_m if there exists a non-terminal N_m different from N_ℓ such that $\omega_\ell[i : j] = \omega_m$.

3. For each Γ_ℓ, find a shortest path from 1 to $|\omega_\ell| + 1$.

4. Return the labels of these paths.

The right-hand side for non-terminal N_ℓ is the concatenation of the labels of a shortest path of Γ_ℓ. Intuitively, an edge from node i to node $j+1$ with label N_m represents a possible replacement of the occurrence $\omega_\ell[i:j]$ by N_m. There may be more than one grammar parsing with minimal size. If Q is a subset of the repeats of the sequence s, we denote by $mgp(\{s\} \cup Q)$ the set of production rules $\mathcal{P}$ corresponding to one of the minimal grammar parsing of $\{s\} \cup Q$.

The list of occurrences of each constituent over the original sequence can be stored at the moment it is chosen. Supposing then that the graphs are created, and as the length of each constituent is bounded by $n = |s|$, the complexity of finding a shortest path for one graph with a classical dynamic programming algorithm lies in $\mathcal{O}(n \times m)$. Because there are $m+1$ graphs, computing $mgp(\{s\} \cup Q)$ is in $\mathcal{O}(n \times m^2)$.

Note that in practice the graph Γ_0 contains all the information for all other graphs: any Γ_ℓ is a subgraph of Γ_0. Therefore, we call Γ_0 the Grammar Parsing graph (*GP-graph*).

4.2. IRR with Occurrence Optimization

We can now define a variant of IRR, called *Iterative Repeat Choice with Occurrence Optimization* (IRCOO) whose pseudo-code is given in Algorithm 2. Different from IRR, what is maintained is a set of terminal strings, and the current grammar in each moment is a Minimal Grammar Parsing over this set of strings. Recall that $cons(\omega)$ gives the only terminal string that can be derived from ω (the "constituent"). Again, we define IRCCO-MC, IRCOO-MF, IRCOO-ML the instances of Algorithm 2 where the score function is defined as f_{MC}, f_{MF}, f_{ML}.

The computation of the *argmax* function depends only on the number of repeats, assuming that f is constant, so that its complexity lies in $\mathcal{O}(n^2)$. Like for IRR, the total number of times the while loop is executed is bounded by n. The complexity of this generic scheme is thus $\mathcal{O}(n \times (n^2 + n \times m^2))$.

Algorithm 2 Iterative Repeat Choice with Occurrences Optimization.

$\text{IRCOO}(s, f)$

Require: s is a sequence, and f is a score function on words

1: $\mathcal{C} \leftarrow \{s\}$
2: $\mathcal{P} \leftarrow \{S \rightarrow s\}$
3: **while** $(\exists \omega \; : \; \omega \leftarrow \text{argmax}_{\alpha \in \text{repeats}(\mathcal{P})} f(\alpha, \mathcal{P})) \wedge |mgp(\mathcal{C} \cup \{cons(\omega)\})| < |\mathcal{P}|$ **do**
4: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{cons(\omega)\}$
5: $\quad \mathcal{P} \leftarrow mgp(\mathcal{C})$
6: **end while**
7: **return** $G(\mathcal{P})$

As an example, consider again the sequence from Section 3.2, where α, β and γ have length one. After three iterations of IRRCOO-MC the words xax, xbx and xcx are chosen, and a MGP of these words plus the original sequence results in G^*.

IRRCOO extends IRR by performing a global optimization at each step of the replaced occurrences but still relies on the classical score functions of IRR to choose the words to introduce. But the result of the optimization can be used directly to guide the search in a hill-climbing approach that we introduce in the next subsection.

4.3. Widening the Explored Space

In this section we divert from IRR algorithms by taking the idea presented in IRRCOO a step forward. Here, we consider the optimization procedure (*mgp*) as a scoring function for sets of substrings.

We first introduce the search space, defined over all possible sets of substrings, for the Smallest Grammar Problem and, second, an algorithm performing a wider exploration of this search space than the classical greedy methods.

4.3.1. The Search Space

The *mgp* procedure permits us to resolve the problem of finding a smallest grammar given a fixed set of constituents. The Smallest Grammar Problem reduces then to find this good set of constituents. This is the idea behind the search space we will consider here.

Consider the lattice $\langle \mathcal{R}(s), \subseteq \rangle$, where $\mathcal{R}(s)$ is the collection of all possible sets of repeated substrings in s: $\mathcal{R}(s) = 2^{\text{repeats}(s)}$. Every node of this lattice corresponds to a set of repeats of s. We then define a score function over the nodes of the lattice as $score(\eta) = |\text{mgp}(\{s\} \cup \eta)|$.

An algorithm over this search space will look for a local or global minimum. To define this, we first need some notation:

Definition 1. *Given a lattice $\langle L, \preceq \rangle$:*

1. $ancestors(\eta) = \{\theta \neq \eta \mid \eta \preceq \theta \wedge (\nexists \kappa, \kappa \neq \eta, \theta \ s.t. \ \eta \preceq \kappa \preceq \theta)\}$

2. $descendants(\eta) = \{\theta \neq \eta \mid \theta \preceq \eta \wedge (\nexists \kappa, \kappa \neq \eta, \theta \ s.t. \ \theta \preceq \kappa \preceq \eta)\}$

The *ancestors* of node η are the nodes exactly "over" η, while the *descendants* of node η are the nodes exactly "under" η.

Now, we are able to define a global and local minimum.

Definition 2. *Given a lattice $\langle L, \preceq \rangle$ and an associate score function over nodes, $g(\eta)$:*

1. A node η is a global minimum *if $g(\eta) \leq g(\theta)$ for all $\theta \in L$.*

2. A node η is a local minimum *if $g(\eta) \leq g(\theta)$ for all $\theta \in ancestors(\eta) \cup descendants(\eta)$.*

Unless otherwise noted the default score function for nodes will be $score(\eta)$ (defined above). Finally, let $\mathcal{SG}(s)$ be the set of all grammars of minimal size for the sequence s. Similarly, we define $\mathcal{MGP}(\{s\} \cup \eta)$ the set of minimal grammars with constituents $\{s\} \cup \eta$. With this definition, we can state formally that this lattice is a "good" search space:

Theorem 2. $\mathcal{SG}(s) = \bigcup_{\eta : \eta \ is \ global \ minimum \ of \ \langle \mathcal{R}(s), \subseteq \rangle} \mathcal{MGP}(\{s\} \cup \eta)$

Proof. To see the first inclusion ($\subseteq$), take a smallest grammar G^*. All strings in $cons(G^*)$ have to be repeats of s, so $cons(G^*) \setminus \{s\}$ corresponds to a node η in the lattice and G^* has to be in $\mathcal{MGP}(\{s\} \cup \eta)$. Conversely ($\supseteq$), all grammars of the right expression have the same size, which is minimal, so they are all smallest grammars. $\square$

Because of the NP-hardness of the problem, it is fruitless (supposing $P \neq NP$) to search for an efficient algorithm to find a global minimum. We will present therefore an algorithm that looks for a local minimum on this search space.

4.3.2. The Algorithm

In contrast with classical methods, we can now define algorithms discarding also constituents. To perform a wider exploration of the search space, we propose a new algorithm performing a succession of greedy search in ascending and descending directions in the lattice until a local minimum is found.

This algorithm explores the lattice in a zigzag path. Therefore we denote it *ZZ*. It explores the lattice by an alternation of two different phases: *bottom-up* and *top-down*. The bottom-up can be started at any node, it moves upwards in the lattice and at each step it looks among its ascendants for the one with the lowest score. In order to determine which is the one with the lowest score, it inspects them all. It stops when no ascendant has a better score than the current one. As in bottom-up, top-down starts at any given node but it moves downwards looking for the node with the smallest score among its descendants. Going up or going down from the current node is equivalent to adding or removing a substring to or from the set of substrings in the current node respectively.

ZZ starts at the bottom node, that is, the node that corresponds to the grammar $S \rightarrow s$ and it finishes when no improvements are made in the score between two bottom-up–top-down iterations. A pseudo-code is given in Algorithm 3.


Algorithm 3 Zigzag algorithm.

$ZZ(s)$

Require: s is a sequence
1: $\mathcal{L} \leftarrow \langle \mathcal{R}(s), \subseteq \rangle$
2: $\eta \leftarrow \emptyset$
3: **repeat**
4:     **while** $\exists \eta' \in \mathcal{L} : \left(\eta' \leftarrow argmin_{\eta' \in \text{ancestors}(\eta)} \text{score}(\eta') \right) \wedge \text{score}(\eta') \leq \text{score}(\eta)$ **do**
5:         $\eta \leftarrow \eta'$
6:     **end while**
7:     **while** $\exists \eta' : \left(\eta' \leftarrow argmin_{\eta' \in \text{descendants}(\eta)} \text{score}(\eta') \right) \wedge \text{score}(\eta') \leq \text{score}(\eta)$ **do**
8:         $\eta \leftarrow \eta'$
9:     **end while**
10: **until** $\text{score}(\eta)$ doesn't decrease
11: **return** $G\left(\text{mgp}(\eta)\right)$

For example, suppose that there are 5 substrings that occur more than once in s and that they all have length greater than two. Let these strings be numbered from 0 to 4. We start the ZZ algorithm at the bottom node. It inspects nodes $\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, and $\{4\}$. Suppose that $\{1\}$ produces the best grammar, then ZZ moves to that node and starts over exploring the nodes above it. Figure 2 shows a part of the lattice being explored. Dotted arrows point to nodes that are explored while full arrows point to nodes having the lowest score. Suppose that the algorithm then continues up until it reaches node

Figure 2. The fraction of the lattice that is explored by the ZZ algorithm.
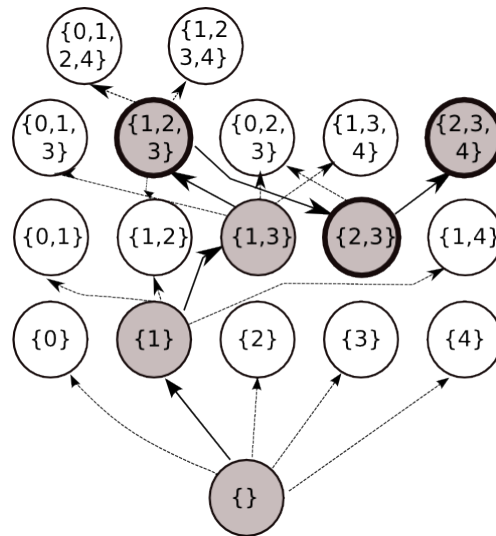
$\{1, 2, 3\}$ where no ancestor has lower score. Then ZZ starts the top-down phase, going down to node $\{2, 3\}$ where no descendant has lower score. At this point a bottom-up–top-down iteration is done and the algorithm starts over again. It goes up, suppose that it reaches node $\{2, 3, 4\}$ where it stops. Bold circled nodes correspond to nodes were the algorithm switches phases, grey nodes correspond to nodes with the best score among its siblings.

Computational Complexity In the previous section we showed that the computational complexity of computing the score function for each node is $\mathcal{O}(n \times m^2)$, where n is the length of the target string and m is the number of substrings in the node. Every time ZZ looks for a substring to add or remove it has to inspect all possible candidates with the aim of finding the one that minimizes the score. Depending on the number of substrings that are already in the node, there might be at most $\mathcal{O}(n^2)$ candidate strings. As a consequence, each step upwards or downwards is made in $\mathcal{O}(n^2 \times n \times m^2)$. Next, we need to give an upper bound for the length of the path that is potentially traversed by the algorithm. In order to define it, we first note two important properties: the score of the bottom node is equal to n and the score of any node containing more than $n/2$ substrings is at least n. The bottom-up phase visits at most $n/2$ nodes, and consequently, the top-down can only go down at most $n/2$ steps. Adding them together, it turns out that a bottom-up–top-down iteration traverses at most n nodes. Now, each of these iteration decreases the score by at least 1, otherwise the algorithm stops. Since the initial score is n plus the fact that the score is always positive, there can be at most n bottom-up–top-down iterations. This results in a complexity for the ZZ algorithm of $\mathcal{O}(n^5 \times m^2)$.

4.3.3. Non-Monotonicity of the Search Space

We finish this section with a remark on the search space. In order to ease the understanding of the proof, we will suppose that the size of the grammar is defined as $|G| = \sum_{A \to \alpha \in \mathcal{P}} (|\alpha|)$. The proof extends easily if we consider our definition of size, but is more cumbersome (basically, instead of taking blocks of size two in the proof, take them of size three).

We have presented an algorithm that finds a local minimum on the search space. Locality is defined in terms of its direct neighborhood, but we will see that the local minimality of a node does not necessarily extend further:

Proposition 1. *The lattice* $\langle \mathcal{R}(s), \subseteq \rangle$ *is not monotonic for function* $score(\eta) = |mgp(\eta \cup \{s\})|$*. That is, suppose* η *is a local minimum. There may be a node* θ *such that* $\eta \preceq \theta$ *and* $score(\eta) > score(\theta)$*.*

Proof. Consider the following sequence:

$$abcd|_1cdef|_2efab|_3cdab|_4efcd|_5abef|_6bc|_7bc|_8de|_9de|_{10}fa|_{11}fa|_{12}da|_{13}da|_{14}fc|_{15}fc|_{16}be|_{17}be|_{18}ab|_{19}cd|_{20}ef.$$

The set of possible constituents is $\{ab, bc, cd, de, ef, fa, da, fc, be\}$, none of which has size longer than 2. Note that the digrams that appear in the middle of the first blocks (of size four) appear repeated twice, while the others only once. Also, the six four-size blocks are all compositions of constituents $\{ab, cd, ef\}$ (each of which is only repeated once at the end). Consider now the following grammar:

$$
\begin{aligned}
S \quad &\rightarrow \quad aB^Cd|_1cD^Ef|_2eF^Ab|_3cD^Ab|_4eF^Cd|_5aB^Ef|_6B^C|_7B^C|_8D^E \\
&\qquad |_9D^E|_{10}F^A|_{11}F^A|_{12}D^A|_{13}D^A|_{14}F^C|_{15}F^C|_{16}B^E|_{17}B^E|_{18}ab|_{19}cd|_{20}ef \\
B^C \quad &\rightarrow \quad bc \\
D^E \quad &\rightarrow \quad de \\
F^A \quad &\rightarrow \quad fa \\
D^A \quad &\rightarrow \quad da \\
F^C \quad &\rightarrow \quad fc \\
B^E \quad &\rightarrow \quad be
\end{aligned}
$$

of size 68, which is a smallest grammar given this set of constituents. Moreover, adding any of the three remaining constituents would increase the size of the grammar by one. But, adding all three of them would permit to parse the blocks of size 4 with only two symbols each, plus parsing the three trailing blocks with only one symbol. This means gaining 9 symbols and losing only 6 (because of the introduction of the new right-hand sides). □

5. Experiments

In this section we experimentally compare our algorithms with the classical ones from the literature. For this purpose we use the Canterbury Corpus [23] which is a standard corpus for comparing lossless data compression algorithms. Table 1 lists the sequences of the corpus together with their length and number of repeats of length greater than one.

Not all existing algorithms are publicly available, they resolve in different ways the case when there are more than two words with the best score, they do not report results on a standard corpus or they use different definitions of the size of a grammar. In order to standardize the experiments and score, we implemented all the offline algorithms presented in this paper in the IRR framework. For the sake of completeness, we also add to the comparison LZ78 and Sequitur. Note that we have post-processed the output of the LZ78 factorizations to transform them into context-free grammars. The first series of experiments aims at comparing these classical algorithms and are shown in the middle part of Table 2. On

this benchmark, we can see that IRR-MC always outputs the smallest grammars, which are on average 4.22% smaller than those of the second best (IRR-MF), confirming the partial results of [9] and showing that IRR-MC is the current state-of-the-art practical algorithm for this problem.

Table 1. Corpus statistics.

Sequence	Length	# of Repeats
alice29.txt	152,089	220,204
asyoulik.txt	125,179	152,695
cp.html	24,603	106,235
fields.c	11,150	56,132
grammar.lsp	3,721	12,780
kennedy.xls	1,029,744	87,427
lcet10.txt	426,754	853,083
plrabn12.txt	481,861	491,533
ptt5	513,216	99,944,933
sum	38,240	666,934
xargs.1	4,227	7,502

Then, we evaluate how the optimization of occurrences improves IRR algorithms. As shown in the IRRCOO column of Table 2, each strategy for choosing the word is improved by introducing the occurrence optimization. The sole exceptions are for the MF strategy on `grammar.lsp` and `xargs.1`, but the difference in these cases is very small and the sequences are rather short. More importantly, we can see that IRRCOO-MC is becoming the new state-of-the-art algorithm, proposing for each test a smaller grammar than IRR-MC, and being outperformed only on `plrabn12.txt` by IRRCOO-MF.

If given more time, these results can still be improved by using ZZ. As shown in column ZZ of Table 2, it obtains in average 3.12% smaller grammars than IRR-MC, a percentage that increases for the sequences containing natural language (for instance, for `alice29.txt` the gain is 8.04%), while it is lower for other sequences (only 0.1% for `kennedy.xls` for example). For the latter case, one can remark that the compression ratio is already very high with IRR-MC and that it may be difficult or impossible to improve it, the last few points of the percentage gain being always the hardest to achieve. As expected, ZZ improves over previous approaches mainly because it explores a much wider fraction of search space. Interestingly enough, the family of IRRCOO algorithms also improves state-of-the-art algorithms but still keeps the greedy flavour, and more importantly, it does so with a complexity cost similar to pure greedy approaches. The price to be paid for computing grammars with ZZ is its computational complexity. This problem already showed up with `plrabn12.txt`, `lcet10.txt` (where each repeat individually does not compress the sequence much, so lots of iterations are necessary) and `ptt5` (which contains about 99 millions of repeats).

6. Non-Uniqueness of Smallest Grammar and Structure Discovery

Depending on the implementation, the *mgp* algorithm could return different small grammars using the same set of constituents because there is usually more than one shortest path in the *GP-graph*, and

Table 2. Grammar sizes on the Canterbury corpus. The files over which ZZ did not finished are marked with a dash.

Sequences	Algorithms from the literature					Optimizing occurrences			
				IRR			IRRCOO		
	Sequitur	LZ78	MF	ML	MC	MF	ML	MC	ZZ
alice29.txt	49,147	116,296	42,453	56,056	**41,000**	39,794	52,351	**39,251**	**37,701**
asyoulik.txt	44,123	102,296	38,507	51,470	**37,474**	36,822	48,133	**36,384**	**35,000**
cp.html	9,835	22,658	8,479	9,612	**8,048**	8,369	9,313	**7,941**	**7,767**
fields.c	4,108	11,056	3,765	3,980	**3,416**	3,713	3,892	**3,373**	**3,311**
grammar.lsp	1,770	4,225	1,615	1,730	**1,473**	1,621	1,704	**1,471**	**1,465**
kennedy.xls	174,585	365,466	167,076	179,753	**166,924**	166,817	179,281	**166,760**	166,704
lcet10.txt	112,205	288,250	92,913	130,409	**90,099**	90,493	120,140	**88,561**	–
plrabn12.txt	142,656	338,762	125,366	180,203	**124,198**	**114,959**	164,728	117,326	–
ptt5	55,692	106,456	45,639	56,452	**45,135**	44,192	53,738	**43,958**	–
sum	15,329	35,056	12,965	13,866	**12,207**	12,878	13,695	**12,114**	**12,027**
xargs.1	2,329	5,309	2,137	2,254	**2,006**	2,142	2,237	**1,989**	**1,972**

therefore there are multiple paths from which the algorithm can choose. From the point of view of data discovery, all these grammars are equally interesting if we only consider their size, but different grammars might have different structures despite having the same size.

In this section we investigate these phenomena from two different perspectives. From a theoretical point of view, we provide bounds on the number of different smallest grammars, both globally and locally (fixing the set of constituents). And from an empirical point of view, we explore and compare the actual structural variances among different grammars on some real-life sequences.

6.1. Bounds on the Number of Smallest Grammars

It is clear that a smallest grammar is not necessarily unique. Not so clear is how many smallest grammars there can be. First we will prove that for a certain family of sequences, any node of the search space corresponds to a smallest grammar. As in the proof of Proposition 1 and only to ease the understanding of the proof, we will use $|G| = \sum_{A \to \alpha \in \mathcal{P}}(|\alpha|)$ as the definition of grammar size.

Proposition 2. *Let $m(k) = \max_{s \, : \, |s|=n}$(number of global minima for $\langle \mathcal{R}(s), \subseteq \rangle$).*
Then, $m(n) \in \Omega(c^n)$, with $c > 1$ constant.

Proof. It is sufficient to find one family of sequences for which the number of global minima is exponential. Consider the sequence

$$s_k = a_1 a_1 |_1 a_1 a_1 |_2 a_2 a_2 |_1 a_2 a_2 |_2 \ldots |_2 a_k a_k |_1 a_k a_k |_2 = \prod_{i=1}^{k} (a_i a_i |_1 a_i a_i |_2)$$

of length $n = 6k$ over an alphabet of size $k + 2$. The a_i's are single symbols. The set of repeated substrings longer than one is $\{a_i a_i, 1 \le i \le k\}$, of size k. For any subset, there is only one minimal grammar with this constituent set. Adding any remaining constituents to this grammar reduces the length of the axiom rule by 2, but does not reduce anything in the remaining rules, and adds 2 to the grammar size. The same happens when a constituent is removed. Therefore, any node of the lattice is a local minimum, and therefore a global. As there are $2^{\frac{n}{6}}$ nodes, this proves the proposition. $\qquad \square$

Now, we will suppose that the set of constituents is fixed and consider the number of smallest grammars that can be built with this set. Given a set of constituents Q we denote with $\mathcal{G}_Q$ the set of all the smallest grammars that can be formed with Q. Different implementations of the *mgp* algorithm may return different grammars from $\mathcal{G}_Q$.

As the following Proposition shows, there are cases where the number of different smallest grammars can grow exponentially for a given set of constituents.

Proposition 3. *Let* $m(n) = \max\limits_{s \in \Sigma^k, \, Q \subseteq repeats(s)} (|G_{Q \cup \{s\}}|).$
Then $m(n) \in \Omega(c^n)$*, with* $c > 1$ *constant.*

Proof. Let s_k be the following sequence $(aba)^k$ and let Q be $\{ab, ba\}$. Then the *mgp* algorithm can parse each aba in only one of the two following ways: aA or Ba, where A and B derives ab and ba respectively. Since each occurrence of aba can be replaced by aA or Ba independently, we can see that there are 2^k different ways of rewriting the body of rule s_k. Moreover, all of them have the same minimal length, so there are 2^k grammars of the same (minimal) size that can be formed for s_k taking Q as its constituents. $\qquad\square$

Proposition 2 is complementary to Proposition 3. In the former, we prove that the number of global minima (and therefore, the number of grammars with the same minimal size) is exponential. To prove it, we provided a sequence with the property that any constituent set would yield a grammar with minimal size. In the latter we show that even if the set is fixed, still there can be an exponential number of smallest grammars with this constituent set. Not however that for the proof of Proposition 2 we needed an unbounded alphabet.

These two propositions suggest that it might not be possible to find one *gold* smallest grammar that could be used for structure discovery. If we consider only the size of the grammar, then all grammars in $\mathcal{G}_Q$ are equally interesting, but different grammars might have different structures despite having the same size.

In the next section we analyze the differences among grammars of the same smallest size for a few real-life sequences. We explore their structural variances using similarity metrics and from different points of view.

6.2. An Empirical Analysis

We will now introduce a way of measuring the difference between any pair of grammars taken from $\mathcal{G}_Q$. Our measure is based on the fact that our grammars have a single string in their language and that string has only one derivation tree. Therefore it seems natural to use a metric that is commonly used to compare parse trees, such as Unlabeled Brackets F_1 [24](also see [25] for a different explanation) in order to compare different grammars (hereafter UF_1).

Summarizing the definition, the UF_1 measures the overlap of the *brackets* sets of two parse trees. Given a grammar with a single string s in its language (which is equivalent to a parse tree) the set of *brackets* for that grammar is defined as the pairs of indexes in string s for which there is a non-terminal symbol that expands to that position. For instance, the initial symbol S of a grammar expands to s, so $(0, |s| - 1)$ belongs to the bracket set of that grammar.

Full overlap of the brackets sets ($UF_1 = 1$) implies that the two smallest grammars are equal, and the converse also holds: if two smallest grammars are equal then $UF_1 = 1$. Also, given two smallest grammars over the same fixed constituent set, the opposite may happen. Consider again the sequence $s_k = (aba)^k$, as defined in Proposition 3, and take G_1 as the grammar that rewrites every aba as aA while grammar G_2 rewrites it as Ba. G_1 and G_2 do not share any brackets, so $UF_1(G_1, G_2) = 0$.

In our experiments we will use UF_1 as the way to compare structure between different smallest grammars.

Propositions 2 and 3 suggest that there are sequences for which the number of smallest grammars and the number of grammars for a given set of constituents are both exponential. Both results are based on sequences that were especially designed to show these behaviours. But it might be the case that this behaviour does not occur "naturally". In order to shed light on this topic we present four different experiments. The first is directly focused on seeing how Proposition 3 behaves in practice and consists in computing the number of grammars with smallest size that are possible to build using the set of constituents found by the ZZ algorithm. The other three experiments analyze how different all these grammars are. Firstly, we compare the structure of different grammars randomly sampled from the set of possible grammars. Then we compare structure, as in the previous experiment, but this time, the used metric discriminates by the length of the constituents. Finally, we consider in how many different ways a single position can be parsed. The first experiment provides evidence supporting the idea of Proposition 3 being the general rule more than an exceptional case. The second group of experiments suggests that there is a common structure among all possible grammars. In particular, it suggests that longer constituents are more stable than shorter ones and that, if for each position we only consider the longest bracket that englobes it, then almost all of these brackets are common between smallest grammars.

All these experiments require a mechanism to compute different grammars sharing the same set of constituents. We provide an algorithm that not only computes this but also computes the total number of grammars that are possible to define for a given set of constituents. Let s be a sequence, and let Q be a set of constituents. An *MGP-graph* is a subgraph of the *GP-graph* used in Section 4.1 for the resolution of the MGP problem. Both have the same set of nodes but they differ in the set of edges. An edge belongs to an *MGP-graph* if and only if it is used in at least one of the shortest path from node 0 to node $n + 1$ (the end node). As a consequence, every path in an *MGP-graph* connecting the start node with the end one is in fact a shortest path in the *GP-graph*, and every path in *MGP-graph* defines a different way to write the rule for the initial non-terminal S. Similar subgraphs can be built for each of the non-terminal rules, and collectively we will refer to them as the *MGP-graphs*. Using the *MGP-graphs* it is possible to form a smallest grammar by simply choosing a smallest path for every *MGP-graph*. It only remains to explain how to compute a *MGP-graph*. We will do so by explaining the starting symbol case, for the rest of the non-terminals it follows in a similar fashion.

The algorithm traverses the *GP-graph* deleting edges and keeping those that belong to at least one shortest path. It does it in two phases. First it traverses the graph from left to right. In this phase, the algorithm assumes that the length of the smallest path from node 0 is known for every node $j < i$. Since all incoming edges to node i in the *GP-graph* come from previous nodes, it is possible to calculate the smallest path length from node 0 to node i and the algorithm only keeps those edges that are part of such

paths deleting all others. The first phase produces a graph that includes paths that do not end in the final node. In order to filter out this lasts edges, the second phase of the algorithm runs a breadth-first search from the node $n + 1$ and goes backwards removing edges that are not involved in a smallest path.

In our experiments, we use `alice29.txt`, `asyoulik.txt`, and `humdyst.chr`. The first two are sequences of natural language from the Canterbury corpus, while the last one is a DNA sequence from the historical corpus of DNA used for comparing DNA compressors [26]. In all cases, the set of constituents was produced by the ZZ algorithm.

6.2.1. Counting

Using the *MGP-graphs* it is possible to recursively calculate the number of shortest paths that exist from the start node n_0 to any other node n_i. Clearly, the base case states that the number of shortest paths to n_0 is equal to one. The recursive case states that the number of shortest paths to a node n_{i+1} is equal to the sum of the number of shortest paths of nodes n_j, $j \leq i$, that have an edge arriving to node n_{i+1}.

Table 3 shows the number of different grammars that exist for our sequences. The experiments provide evidence that the number of small grammars is huge and that it might not even be tractable by a computer. These huge number of grammars pose an important question: How similar are these grammars?

Table 3. Sequence length, grammar size, number of constituents, and number of grammars for different sequences.

Sequence	**humdyst.chr.seq**	**asyoulik.txt**	**alice29.txt**
Sequence length	38,770	125,179	152,089
Grammar size	10,035	35,000	37,701
Number of constituents	576	2391	2749
Number of grammars	2×10^{497}	7×10^{968}	8×10^{936}

6.2.2. UF_1 with Random Grammars

To answer this question, we sample elements from $\mathcal{G}_Q$ and compare them using UF_1 as was described at the beginning of Section 6.2. We compare pairs of grammars, and we estimate and report average similarity between elements in $\mathcal{G}_Q$ in Table 4.

In order to sample from $\mathcal{G}_Q$ uniformly, we need to be able to sample each rule body, which means that we need to be able to sample complete paths (from start to end) in each *MGP-graph*. Using the previous algorithm for the number of smallest paths we can extend it to an algorithm that samples with uniform probability: starting at the end node, the algorithm can work its way back to the start node by repeatedly choosing one of the incoming edges with a probability proportional to the amount of smallest paths that go through that edge. The chosen edge labels form a rule body uniformly sampled among all possible ones.

Table 4. Different figures for a sample of 1000 grammars uniformly taken from $\mathcal{G}_Q$.

Sequence	humdyst.chr.seq	asyoulik.txt	alice29.txt
UF_1 mean	66.02%	81.48%	77.81%
UF_1 standard deviation	1.19%	1.32%	1.52%
Smallest UF_1	62.26%	77.94%	73.44%
Largest UF_1	70.64%	85.64%	83.44%

6.2.3.   Accounting for Constituent Sizes: $UF_{1,k}$

Our second experiment aims to discover the class of brackets that make the main difference. Note that the standard UF_1 measure gives the same weight to a bracket of size 2 than to longer brackets. The following experiment is quite similar to the previous one, but in order to evaluate the impact of smaller brackets in the score we introduce a new custom metric: $UF_{1,k}$, which is the same that UF_1 but the brackets whose length are equal or smaller than a given size k are ignored in the calculation. So when $k = 1$, $UF_{1,k}$ is equal to UF_1, but for larger values of k more and more brackets are ignored in the calculation. Table 5 reports the results for different values of k. For each sequence, the table contains two columns: one for $UF_{1,k}$ and one for the percentage of the total brackets that were included in the calculation.

As it can be seen, the F-measure increases along k. This indicates that bigger brackets are found in most grammars of $\mathcal{G}_Q$, but it also shows that smaller brackets are much more numerous.

Table 5. $UF_{1,k}$ for different values of k each from a sample of 100 grammars uniformly taken from $\mathcal{G}_Q$.

k	asyoulik.txt.ij		alice29.txt.ij		humdyst.chr.seq.ij	
	$UF_{1,k}$	Brackets	$UF_{1,k}$	Brackets	$UF_{1,k}$	Brackets
1	81.50%	100.00%	77.97%	100.00%	67.32%	100.00%
2	88.26%	50.86%	83.70%	53.14%	71.11%	45.99%
3	92.49%	29.57%	87.94%	32.42%	75.93%	37.54%
4	95.21%	19.85%	89.60%	22.01%	82.17%	15.69%
5	96.35%	11.78%	88.88%	14.36%	88.51%	3.96%
6	97.18%	8.23%	89.45%	9.66%	95.46%	1.24%
7	97.84%	5.72%	91.50%	6.44%	98.38%	0.44%
8	97.82%	3.83%	92.78%	4.30%	99.87%	0.19%
9	98.12%	2.76%	92.37%	2.95%	100.00%	0.06%
10	98.35%	1.88%	91.87%	2.10%	100.00%	0.04%

6.2.4.   Conserved Parsing among Minimal Grammar Parsings

In the previous section we analyzed how different the grammars are considering different sizes of brackets. Here, we consider the differences between the grammars on single positions. The objective of this experiment is to measure the amount of different ways in which a single position of the original

sequence s can be parsed by a minimal grammar. For this we will consider the partial parse tree where only the first level is retained. Doing this for each minimal grammar, we compute for each position the number of different subtrees it belongs to. This is equivalent to the number of edges that cover that position on the *MGP-graph* of s. If the number for one position is one for instance, this means that in all minimal grammars the same occurrence of the same non-terminal expands on this position.

Figure 3. The X axis are the number of different symbols that expand to one position, the Y axis the number of positions that have this number of expansions. Note that Y is in logarithmic scale.
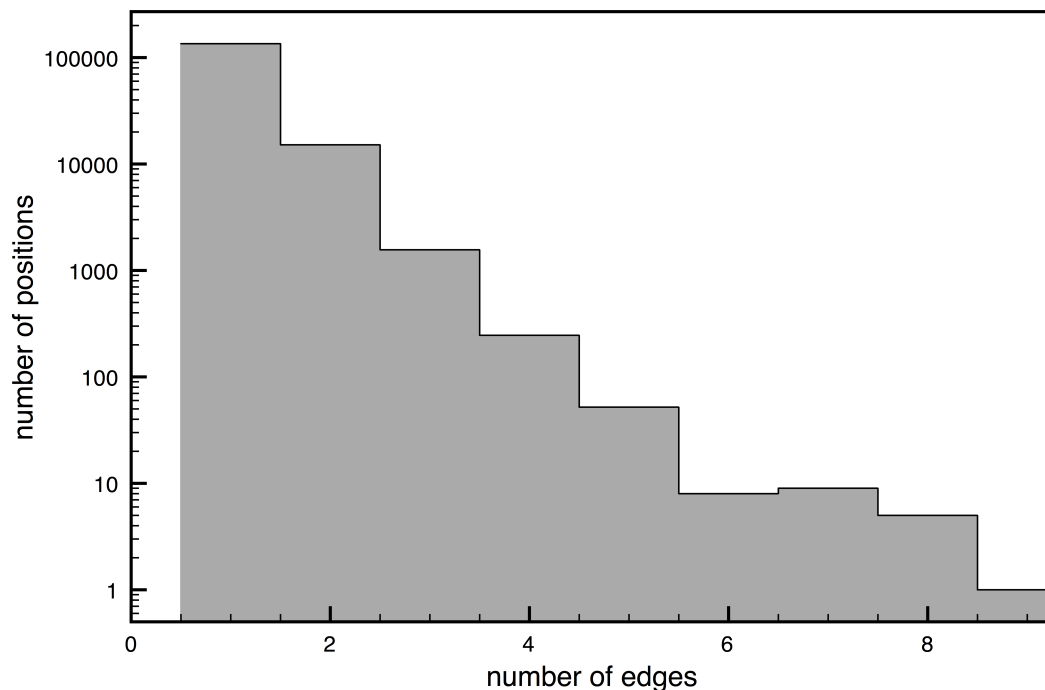
On `alice29.txt`, 89% of the positions are parsed exactly the same way. A histogram for all values of different parses can be seen in Figure 3. Note that the y-axis is in logarithmic scale and that the number of positions reduce drastically if the number of parses is increased: only 10% of positions are parsed in two different ways, 1% in three and all others in less than 0.2%.

There were two regions that presented peaks on the number of different symbols. Both correspond to parts in the text with long runs of the same character (white spaces): at the beginning, and in the middle during a poetry.

While this experiment is only restricted to the first level of the parse tree, it seems to indicate that the huge number of different minimal parses is due to a small number of positions where different parses have the same size. Most of the positions however are always parsed the same way.

6.3. Some Remarks

Summarising this section, we have the following theoretical results:

- There can be an exponential amount of sets of constituents such that all of them produce smallest grammars (Proposition 2).

- There might be two smallest grammars not sharing a single constituent other than s (from the proof of Proposition 2).

- There can be an exponential amount of smallest grammars even if the set of constituents is fixed (Proposition 3).

- Two smallest grammars with the same set of constituents might not share a single bracket other than the one for s (from the proof of Proposition 3).

Thus, from a theoretical point of view, there may exist structures completely incompatible between them, and nevertheless equally interesting according to the minimal size criteria. Given these results, it may be naive to expect a single, correct, hierarchical structure to emerge from approximations of a smallest grammar.

Accordingly, our experiments showed that the number of different smallest grammars grow well beyond an explicitly tractable amount.

Yet, the UF_1 experiment suggested the existence of a common structure shared by the minimal grammars while the $UF_{1,k}$ experiment showed that this conservation involves more longer constituents than the smaller ones. The last experiment seems to indicate something similar, but concerning conservation of first-level brackets (which may not be the same as the longest brackets). One interpretation of these experiments is that, in practice, the observed number of different grammars is maybe due to the combination of all the possible parses by less significant non-terminals, while a common informative structure is conserved among the grammars. In that case, identifying the relevant structure of the sequence would require to be able to distinguish significant non-terminals, for instance by statistical tests. Meaningless constituents could also be discarded by modifying the choice function according to available background knowledge or by shifting from a pure Occam's razor point of view to a more Minimum Description Length oriented objective, which would prevent to introduce non informative non-terminals in the grammars.

7. Conclusions and Future Work

In this paper, we analyzed a new approach to the Smallest Grammar Problem, which consisted in optimizing separately the choice of words that are going to be constituents, and the choice of which occurrences of these constituents will be rewritten by non-terminals. Given a choice of constituents, we resolve the optimal choice of constituents with the polynomial-time algorithm *mgp*. We improve classical offline algorithms by optimizing at each step the choice of the occurrences. This permits to overcome a restriction of these classical algorithms which does not permit them to find smallest grammars.

The separation allows to define the search space as a lattice over sets of repeats where each node has an associated score corresponding to the size of the MGP of this node. We propose then a new algorithm that finds a local minimum. It explores this search space by adding, but also removing, repeats to the current set of words. Our experiments show that both approaches outperform state-of-the-art algorithms.

We then analyzed more closely how this approach can be used for structure discovery, a main application of the smallest grammar problem. While in applications of data compression and Kolmogorov complexity, we look for the size of a smallest grammar, in structure discovery the structure itself given by the grammar is the goal. We used our formalization of the smallest grammar problem to analyze the number of possible smallest grammars (globally and given a fixed set of constituents). We proved that there may be an exponential number of grammars with the same minimal size, and analyzed *how* different they are. Because finding the smallest grammar is intractable, we contented ourself here to study the differences between smallest grammars given the best set of constituents our algorithms were able to find. While in theory there may exist several incompatible smallest grammars, our experiments seem to confirm that, in practical cases, there is an overall stability of the different parses. We think that these results can give both some warnings and clues on how to use the result of an approximation to the smallest grammar problem for structure discovery.

The optimization of the choice of occurrences opens new perspectives when searching for the smallest grammars, especially for the inference of the structure of sequences. In future work, we want to study how this scheme actually helps to find better structure on real applications. Our efficiency concerns in this paper were oriented toward not being exponential. In future work we plan to investigate more efficient algorithms and study the compromise between execution time and final grammar size.

Acknowledgements

References

1. Charikar, M.; Lehman, E.; Liu, D.; Panigrahy, R.; Prabhakaran, M.; Sahai, A.; Shelat, A. The smallest grammar problem. *IEEE Trans. Inf. Theory* **2005**, *51*, 2554–2576.
2. Rytter, W. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.* **2003**, *302*, 211–222.
3. Kieffer, J.; Yang, E.H. Grammar-based codes: a new class of universal lossless source codes. *IEEE Trans. Inf. Theory* **2000**, *46*, 737–754.
4. Ziv, J.; Lempel, A. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536.
5. Ziv, J.; Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343.
6. Nevill-Manning, C.G. Inferring Sequential Structure. Ph.D. Thesis, University of Waikato, Hamilton, New Zealand, 1996.
7. Wolff, J. An algorithm for the segmentation of an artificial language analogue. *Br. J. Psychol.* **1975**, *66*, 79–90.
8. Bentley, J.; McIlroy, D. Data compression using long common strings. In *Proceeding of Data Compression Conference*; IEEE: New York, NY, USA, 1999; pp. 287–295.

9. Nevill-Manning, C.; Witten, I. On-line and off-line heuristics for inferring hierarchies of repetitions in sequences. In *Proceeding of Data Compression Conference*; IEEE: New York, NY, USA, 2000; pp. 1745–1755.

10. Apostolico, A.; Lonardi, S. Off-line compression by greedy textual substitution. *Proc. IEEE* **2000**, *88*, 1733–1744.

11. Larsson, N.; Moffat, A. Off-line dictionary-based compression. *Proc. IEEE* **2000**, *88*, 1722–1732.

12. Nakamura, R.; Inenaga, S.; Bannai, H.; Funamoto, T.; Takeda, M.; Shinohara, A. Linear-Time Text Compression by Longest-First Substitution. *Algorithms* **2009**, *2*, 1429–1448.

13. Lanctot, J.K.; Li, M.; Yang, E.H. Estimating DNA sequence entropy. In *ACM-SIAM Symposium on Discrete Algorithms*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2000; pp. 409–418.

14. Evans, S.C.; Kourtidis, A.; Markham, T.; Miller, J. MicroRNA Target Detection and Analysis for Genes Related to Breast Cancer Using MDLcompress. *EURASIP J. Bioinf. Sys. Biol.* **2007**, *2007*, 1–16.

15. Nevill-Manning, C.G.; Witten, I.H. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *J. Artif. Intell. Res.* **1997**, *7*, 67–82.

16. Marcken, C.D. Unsupervised language acquisition. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

17. Sakakibara, Y. Efficient Learning of Context-Free Grammars from Positive Structural Examples. *Inf. Comput.* **1992**, *97*, 23–60.

18. Carrascosa, R.; Coste, F.; Gallé, M.; Infante-Lopez, G. Choosing Word Occurrences for the Smallest Grammar Problem. LATA: Trier, Germany, May 24–28; 2010.

19. Karpinski, M.; Rytter, W.; Shinohara, A. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.* **1997**, *4*, 172–186.

20. Gusfield, D. *Algorithms on Strings, Trees, and Sequences—Computer Science and Computational Biology*; Cambridge University Press: Cambridge, UK, 1997.

21. Sakamoto, H.; Maruyama, S.; Kida, T.; Shimozono, S. A Space-Saving Approximation Algorithm for Grammar-Based Compression. *IEICE Trans.* **2009**, *92-D*, 158–165.

22. Schuegraf, E.J.; Heaps, H.S. A comparison of algorithms for data base compression by use of fragments as language elements. *Inf. Storage Retr.* **1974**, *10*, 309–319.

23. Arnold, R.; Bell, T. A Corpus for the Evaluation of Lossless Compression Algorithms. In *Proceeding of Data Compression Conference*; IEEE: Washington, DC, USA, 1997; p. 201.

24. Abney, S.; Flickenger, S.; Gdaniec, C.; Grishman, C.; Harrison, P.; Hindle, D.; Ingria, R.; Jelinek, F.; Klavans, J.; Liberman, M.; Marcus, M.; Roukos, S.; Santorini, B.; Strzalkowski, T. Procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of the Workshop on Speech and Natural Language*; Black, E., Ed.; Association for Computational Linguistics: Stroudsburg, PA, USA, 1991; pp. 306–311.

25. Klein, D. The Unsupervised Learning of Natural Language Structure. Ph.D. Thesis, University of Stanford, Stanford, CA, USA, 2005.

26. Manzini, G.; Rastero, M. A simple and fast DNA compressor. *Softw. Pract. Exp.* **2004**, *34*, 1397–1411.