Naive Infinite Enumeration of Context-free Languages in Incremental Polynomial Time

Christophe Costa Florêncio
(KU Leuven, Belgium
chris.costaflorencio@cs.kuleuven.be)

Jonny Daenen
(Hasselt University and transnational University of Limburg, Belgium
jonny.daenen@uhasselt.be)

Jan Ramon
(KU Leuven, Belgium
jan.ramon@cs.kuleuven.be)

Jan Van den Bussche
(Hasselt University and transnational University of Limburg, Belgium
jan.vandenbussche@uhasselt.be)

Dries Van Dyck
(Belgian Nuclear Research Centre (SCK-CEN),
Boeretang 200, BE-2400 Mol, Belgium
vandyck.dries@gmail.com)

Abstract: We consider the naive bottom-up concatenation scheme for a context-free language and show that this scheme has the incremental polynomial time property. This means that all members of the language can be enumerated without duplicates so that the time between two consecutive outputs is bounded by a polynomial in the number of strings already generated.

Key Words: context-free grammar, systematic generation, polynomial time
Category: F.4.2, F.2

1 Introduction

Let G be a context-free grammar that is arbitrary but fixed, i.e., G is not considered as part of the input. We can define two basic enumeration problems concerning the language $L(G)$ generated by G:

Given-length enumeration with polynomial delay: Given a natural number n, output all strings of length n belonging to $L(G)$, without duplicates, with *polynomial delay.* By "polynomial delay" we mean that the first output, and every next output, is produced within $p(n)$ time, for some fixed polynomial p. Technically, the output is ended by an "end of output" (EOO) message, and the time spent between the last output string and EOO should also be bounded by $p(n)$; moreover, if there are no strings of length n in $L(G)$, then the algorithm should output an EOO right away, again in time bounded by $p(n)$.

Infinite enumeration in incremental polynomial time: Output *all* strings in $L(G)$, without duplicates, in incremental polynomial time (IPT), meaning that the time spent between the n^{th} and the $n+1^{\text{th}}$ output is bounded by $p(n)$ for some fixed polynomial p. Here, n is not directly related to string length, but is simply a count of the number of strings that have been output so far. Since all but the most trivial context-free languages are infinite, we refer to this problem as *infinite enumeration.* In principle, an algorithm for infinite enumeration runs forever (although one may of course abort it at any time), but the incremental polynomial-time bound assures us that the time for every next output grows only polynomially.

The notions of polynomial delay and incremental polynomial time were originally introduced (in a setting unrelated to context-free languages) by [Johnson et al. 1988].

Basic as the above two problems are, the literature on them is relatively scarce. Given-length enumeration was first discussed by [Mäkinen 1997], but not solved completely; then [Dömösi 2000] presented a polynomial-delay solution to the same problem by a modification of the well-known CYK parsing algorithm. His solution has the additional benefit of enumerating the strings in lexicographic order. Later, [Dong 2009] reported linear-time improvements to Dömösi's algorithm. A related problem which has received quite some attention in the literature is the efficient generation of a true random sample of a context-free language [see Gore et al. 1997, Flajolet et al. 1994, Arnold and Sleep 1980].

So, efficient algorithms for given-length enumeration are already available. In the present paper, we consider infinite enumeration. We will show, perhaps unsurprisingly, that any algorithm for given-length enumeration with polynomial delay can be adapted to do infinite enumeration in incremental polynomial time.

The main topic of this paper, however, is the naive, bottom-up generation scheme that enumerates strings not by length, but by depth of their parse tree. While this scheme is not as efficient as the above algorithms, it is still important because it is so basic and natural. Indeed it is a natural question to ask: does the naive bottom-up generation scheme already have the IPT property? In this paper we answer this question affirmatively. We believe this result is interesting mainly from a theoretical perspective as it adds to our most basic understanding of enumerating context-free languages. The proof of our main result is elementary and is based on detailed pumping-lemma-like arguments.

Infinite enumeration may have practical applications in software testing [see Somerville 1998], where a language of test-inputs is described by a context-free grammar [see Arnold and Sleep 1980, Duncan and Hutchinson 1981, Maurer 1990]. In this situation, exhaustive testing of the software on all inputs of the language (e.g., up to a certain length, or until the time budget for testing is exhausted) can be driven by infinite enumeration of the context-free language.

Conversely, infinite enumeration may also have applications in verification of context-free languages. While this task is decidable for some properties [see Baeten et al. 1993], it is undecidable for many other properties, e.g., containment of one context-free language in another is undecidable [see Hopcroft and Ullman 1979]. In such cases, infinite enumeration may be useful to detect counterexamples to conjectured properties, or, when no counterexample is found after a sufficiently long time, it may provide confidence in the conjecture, after which the verifier may start an attempt to find a proof by other methods.

Also, there has been interest in tools for testing and debugging the grammars themselves [see Lämmel 2001, Purdom 1972, Xu et al. 2011], where again infinite enumeration may be helpful.

The paper is outlined as follows: in [Section 2] we first give the necessary definitions and in [Section 3] we give a formal specification of the naive algorithm. In [Section 4], four important results are obtained, which are used in [Section 5] to show the IPT property of the naive algorithm. In [Section 6] a general method is given for transforming a given-length enumeration algorithm with polynomial delay to an algorithm for infinite enumeration in incremental polynomial time. We conclude in [Section 7].

2 Preliminaries

A *context-free grammar* G is a tuple $(\mathcal{N}, \Sigma, \mathcal{P}, S)$, where

- $\mathcal{N}$ is a finite set of *non-terminals*;
- Σ is a finite set of *terminals*, disjoint from $\mathcal{N}$;
- $\mathcal{P}$ is a set of *productions* of the form $X \to \alpha$ with $X \in \mathcal{N}$ and $\alpha \in (\Sigma \cup \mathcal{N})^*$;
- $S \in \mathcal{N}$ is the *start symbol.*

The number of non-terminals $|\mathcal{N}|$ is denoted by η, the number of productions $|\mathcal{P}|$ is denoded by ρ.

We say a non-terminal A *derives* a string s, written as $A \Rightarrow^* s$, if one of the following holds:

- $s \in \Sigma$ and $A \to s \in \mathcal{P}$ (one-step derivation); or
- $\exists B, C \in \mathcal{N} : \exists u, v \in \Sigma^* : A \to BC \in \mathcal{P} \wedge B \Rightarrow^* u \wedge C \Rightarrow^* v \wedge s = uv$.

The *language of a non-terminal* A is defined by $L(G_A) = \{s \mid A \Rightarrow^* s\}$. The language of the start symbol S is also called the *language of* G and is defined by $L(G) = L(G_S)$.

For the rest of the paper, we assume that the grammar is in *Chomsky Normal Form* (CNF) [see Hopcroft and Ullman 1979] without ε-productions, i.e., all productions are of the following form:

- $A \to BC$, a *non-terminal production* or
- $A \to a$, with $a \in \Sigma$, a *terminal production*

where A, B and C are non-terminals and a is a terminal. As we mentioned, the empty string ε cannot be used. Importantly, this implies that we will only deal with nonempty strings. In the rest of this paper, we limit our attention to languages that do not contain ε.

We will also use the *dependency graph* of a context-free grammar. This is a directed graph having $\mathcal{N}$ as set of nodes. There is an edge from A to B if there exists a production of the form $A \to BC$ or $A \to CB$, for some non-terminal C. Note that it is possible for the dependency graph to contain self-loops. When we speak of *reachability* in a directed graph, we always mean reachability by a *directed* path. The length of a path π is equal to the number of edges it contains and is denoted by $l(\pi)$.

We classify the nodes in the dependency graph as follows: a node is *recursive* when it belongs to a directed cycle. It is *leeching* when it can reach a recursive node, but is not recursive itself. Finally, it is *restricted* when it is neither recursive nor leeching. We assume the start symbol is either recursive or leeching, in order to obtain an infinite language (see [Example 1] for the intuition). When the start symbol is restricted, the context-free language described by the grammar is finite and the enumeration problem becomes trivial. We denote the set of recursive non-terminals by $\mathcal{N}_{rec}$, the set of leeching non-terminals by $\mathcal{N}_{leech}$ and the set of restricted non-terminals by $\mathcal{N}_{res}$.

Without loss of generality, we may now make two additional assumptions about the grammar. First, we assume all nodes in the dependency graph are reachable from S. Second, we call a non-terminal *productive* when a string can be derived from it; we require all non-terminals to be productive. When a non-terminal does not satisfy any of these conditions it is called *useless*. [Hopcroft and Ullman 1979] describe an algorithm to remove the useless non-terminals from the grammar.

For each string $s \in L(G_A)$, there exists at least one parse tree that yields s and in which the root of the parse tree is labelled with A. The *depth* of a parse

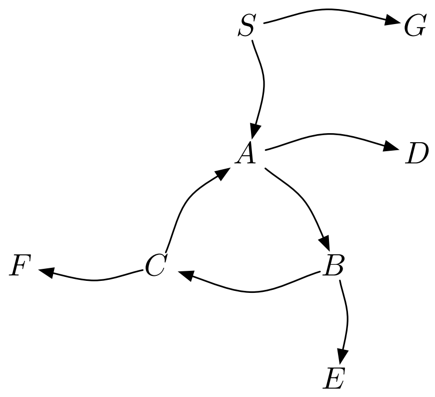

Figure 1: The dependency graph of context-free grammar G_1.

tree τ is equal to the length of a longest path (number of edges) from the root to a leaf and is denoted by $d(\tau)$. Note that we do not restrict the grammar in terms of ambiguity: ambiguous grammars are allowed, hence each string may have multiple parse trees. This leads to the notion of a *minimal parse tree* of a string: a parse tree of minimum depth. Note that a string may have more than one minimal parse tree rooted at a non-terminal A.

Example 1. Consider the following context-free grammar G_1:

$$\begin{array}{ll} S \rightarrow AG & C \rightarrow c \\ A \rightarrow BD & D \rightarrow d \\ B \rightarrow CE & E \rightarrow e \\ C \rightarrow AF & F \rightarrow f \\ & G \rightarrow g \end{array}$$

The corresponding dependency graph is shown in [Fig. 1]. We observe the following classification of the non-terminals:

- $\mathcal{N}_{rec} = \{A, B, C\}$;
- $\mathcal{N}_{leech} = \{S\}$;
- $\mathcal{N}_{res} = \{D, E, F, G\}$.

Note that all non-terminals are reachable and productive, and that $L(G_1)$ is infinite.

3 Algorithm

We now present an iterative algorithm that generates the language described by a given, fixed grammar G (not part of the input).

During the execution of the algorithm, every non-terminal is associated with a set of strings. By $\varDelta A^i$ we denote the set of all strings generated in iteration i for non-terminal A. By A^i we denote the set of all strings generated in iterations 0 to i for non-terminal A.

The iterations are computed according to the following standard inductive *concatenation scheme*:

$$\begin{aligned} A^0 &= \{a \in \varSigma \mid A \to a \in \mathcal{P}\}; \\ A^{i+1} &= A^0 \cup \{u \cdot v \mid \exists B, C \in \mathcal{N} : A \to BC \in \mathcal{P} \wedge u \in B^i \wedge v \in C^i\}; \\ \varDelta A^0 &= A^0; \\ \varDelta A^{i+1} &= A^{i+1} \setminus A^i. \end{aligned}$$

It is easy to see that $A^i \subseteq A^{i+1}$ for each $A \in \mathcal{N}$ and all $i \in \mathbb{N}$.

Remark. In the second rule of the scheme, the use of A^i instead of A^0 would yield equivalent definitions. □

The strings in S^i are called *output strings*, these are the strings in $L(G)$. Note that the terminal productions are only used in iteration 0 and the non-terminal productions are only used in the subsequent iterations.

The set $\varDelta A^i$ contains all strings that can be obtained by combining previously generated strings, according to the associated production(s) of A, except for those that have already been generated. Note that we are working with sets: duplicates are removed, but it is still possible that in the same iteration, or in two different iterations, two identical strings are generated for a non-terminal A (see [Example 2]).

We denote the length of a string s by $|s|$ and the maximal length of a string in A^i by ω^i_A. Note that it is possible for A^i to be empty when $i < \eta - 1$ (this will be shown in [Section 4.2]), in which case ω^i_A is undefined. Clearly, $\omega^{i+1}_A \geq \omega^i_A$ holds for $i \geq \eta - 1$.

We define

$$\begin{aligned} \mathcal{T}^i &= \bigcup_{A \in \mathcal{N}} A^i, \\ \varDelta \mathcal{T}^i &= \bigcup_{A \in \mathcal{N}} \varDelta A^i. \end{aligned}$$

In addition to the output strings, these sets also contain the strings that are only used as building blocks for the output strings, and are not output strings themselves. These are called the *intermediate strings*. The maximal length of a

Non-terminal N	ΔN^0	ΔN^1	ΔN^2	ΔN^3	...
S	{}	{`ab`}	{`abb`}	{`abbb`}	...
A	{`a`}	{`ab`}	{`abb`}	{`abbb`}	...
B	{`b`}	{}	{}	{}	...
C	{`a`}	{}	{}	{}	...

Table 1: Generated output and intermediate strings in the first iterations of applying the naive concatenation scheme to the context-free grammar G_2.

string in $\mathcal{T}^i$ is denoted by $\omega^i_{\mathcal{T}}$. Note that the same string might be generated for multiple non-terminals, i.e., the union $\cup_{A\in\mathcal{N}} A^i$ that defines $\mathcal{T}^i$ is generally not a disjoint union.

Example 2. Consider the following grammar G_2:

$$\begin{aligned} S &\to AB & A &\to a \\ S &\to CB & B &\to b \\ A &\to AB & C &\to a \end{aligned}$$

[Table 1] shows the results of the concatenation scheme applied on G_2 for the first few iterations. Observe that the string `ab` is generated both by both $S \to CB$ and $S \to AB$ in two different iterations. In iteration 2 the string is already present in S^1, hence it is not added to ΔS^2, even though it is generated. The output strings shown in the table (S^3) are `ab`, `abb` and `abbb`. The intermediate strings shown in the table are `a`, `b`, `ab`, `abb` and `abbb` (this set equals $\mathcal{T}^3$).

Remark. An equivalent but more efficient inductive concatenation scheme, which avoids duplicate concatenations is the well-known “semi-naive” scheme [see Ceri et al. 1990]:

$$\begin{aligned} A^0 &= \{a \in \Sigma \mid A \to a \in \mathcal{P}\}; \\ A^{i+1} &= A^i \cup \{u \cdot v \mid \exists B, C \in \mathcal{N} : (A \to BC) \in \mathcal{P} \\ &\qquad\qquad \wedge \big((u \in B^i \wedge v \in \Delta C^i) \vee (u \in \Delta B^i \wedge v \in C^{i-1})\big)\}; \\ \Delta A^{i+1} &= A^{i+1} \setminus A^i. \end{aligned}$$

Although this semi-naive scheme can give practical improvements in performance, e.g., in applications to dabatases [see Bancilhon and Ramakrishnan 1986], the theoretical worst-case complexity is of the same order as the standard scheme. In this paper we will prove that the standard scheme already runs in polynomial incremental time.

Table 2 gives an overview of the symbols used in the rest of the paper.

Symbol	Meaning
η	number of non-terminals
ρ	number of productions
$d(\tau)$	depth of tree τ
$\|s\|$	length of string s
$\|X\|$	number of elements in set X
$\mathcal{N}_{rec}$	set of recursive non-terminals
$\mathcal{N}_{leech}$	set of leeching non-terminals
$\mathcal{N}_{res}$	set of restricted non-terminals
A^i	strings generated for non-terminal A up to and including iteration i
$\varDelta A^i$	strings generated for non-terminal A in iteration i
$\mathcal{T}^i$	intermediate strings generated up to and including iteration i
$\varDelta\mathcal{T}^i$	intermediate strings generated in iteration i
ω_A^i	maximal length of a string in A^i
$\omega_{\mathcal{T}}^i$	maximal length of a string in $\mathcal{T}^i$

Table 2: Overview of symbols.

4 Properties

Our main result is that the naive algorithm satisfies the IPT property described in the introduction. In order to prove this, four important results are obtained in this section:

- A formalization of the start-up phase in [Section 4.2].
- A relation between iteration number and the number of intermediate strings in [Section 4.3].
- A relation between the maximum string length and the number of strings in [Section 4.4].
- A relation between number of intermediate strings and the number of output strings in [Section 4.5].

4.1 String properties

Lemma 1. *For any non-terminal A, the set $\varDelta A^i$ consists precisely of the strings that can be derived from A and have a minimal parse tree depth of $i+1$.*

Proof. We prove the lemma by induction on i.

Base For $i = 0$, $\varDelta A^0$ contains all strings that can be derived from A in 1 step. It is obvious that all these strings have a parse tree of depth 1 and no smaller parse tree exists. Clearly, no other strings can be derived from A with a parse tree of depth 1.

Induction For $i > 0$, suppose the lemma holds for all values smaller than i. Consider a string $s = u \cdot v \in \Delta A^i$ with $u \in B^{i-1}$, $v \in C^{i-1}$ and $A \to BC \in \mathcal{P}$ for some $B, C \in \mathcal{N}$. By induction u and v have minimal parse trees τ_u and τ_v of depth at most i. Note that these parse trees cannot both have a depth smaller than i, because then we could create a parse tree for s of depth $< i + 1$; this would imply (by induction) that $s \in A^{i-1}$, which contradicts $s \in \Delta A^i$. We thus obtain that s has a minimal parse tree of depth $i + 1$.

It remains to show that all strings with a minimal parse tree of depth $i + 1$, that can be derived from A, belong to ΔA^i. Thereto, consider such a string $s \in L(G_A)$ that has a minimal parse tree τ of depth $i + 1$.

We first show that $s \in A^i$. Since $i > 0$, τ has the form of an A-root with two children τ_B and τ_C and $A \to BC \in \mathcal{P}$ for some $B, C \in \mathcal{N}$. Let u and v be the strings yielded by τ_B and τ_C respectively, so $s = u \cdot v$. Since τ has depth $i + 1$, the trees τ_B and τ_C both have a depth $\leq i$. By induction, $u \in \Delta B^j$ and $v \in \Delta C^k$, for some $j, k < i$. In particular, $u \in B^{i-1}$ and $v \in C^{i-1}$. It is now obvious from the definition of A^i that $s = u \cdot v \in A^i$.

Finally, we show that $s \in \Delta A^i = A^i \backslash A^{i-1}$ by proving that $s \notin A^{i-1}$. Suppose that $s \in A^{i-1}$. By induction, s has a minimal parse tree of depth $\leq i$, which contradicts our assumption. □

Knowledge about the iteration in which a string is generated gives us information about the length of the string. Specifically, we have:

Lemma 2. $\forall i \in \mathbb{N} : \forall s \in \Delta A^i : i + 1 \leq |s| \leq 2^i$.

Proof. [Lemma 1] shows that the minimal parse tree of a string $s \in \Delta A^i$ has depth $i + 1$. The yield of a parse tree of depth $i + 1$ has length at least $i + 1$ and at most 2^i. This because of the branching restrictions imposed by the CNF.

4.2 Start-up Phase

In this section we look at the first η iterations of the algorithm: the *start-up phase*. After this initial start-up, all non-terminals will have generated at least one (intermediate) string.

Consider the following definitions:

$$\begin{aligned}
\mathcal{N}^0 &= \{A \in \mathcal{N} \mid \exists a : (A \to a) \in \mathcal{P}\};\\
\mathcal{N}^{i+1} &= \mathcal{N}^0 \cup \{A \in \mathcal{N} \mid \exists B, C \in \mathcal{N}^i : (A \to BC) \in \mathcal{P}\};\\
\Delta\mathcal{N}^0 &= \mathcal{N}^0;\\
\Delta\mathcal{N}^{i+1} &= \mathcal{N}^{i+1} \setminus \mathcal{N}^i.
\end{aligned}$$

Note that $\mathcal{N}^i \subseteq \mathcal{N}^{i+1}$ for all i. Intuitively, a non-terminal A is in $\Delta\mathcal{N}^i$ iff it generates its first string in iteration i.

Definition 3 (Non-recursive parse tree). A *non-recursive parse tree* is a parse tree in which no path contains two nodes labeled with the same non-terminal.

Lemma 4. $\forall A \in \mathcal{N} : A \in \mathcal{N}^i \Leftrightarrow A^i \neq \emptyset$.

Proof. We prove the lemma by induction on i.

Base Let $i = 0$. If $A \in \mathcal{N}^0$, there is a production $A \to a \in \mathcal{P}$, and by definition $a \in A^0$. When $A^0 \neq \emptyset$, we know (from the concatenation scheme) there must be a $A \to a \in \mathcal{P}$. Now, by definition, $A \in \mathcal{N}^0$.

Induction When $i > 0$, assume the lemma holds for all smaller values of i. Suppose $A \in \mathcal{N}^i$. Consider two cases:

a) If $A \in \mathcal{N}^{i-1}$, by induction we know that $A^{i-1} \neq \emptyset$. It follows that $A^i \neq \emptyset$, because $A^{i-1} \subseteq A^i$.

b) Otherwise, there exists a production $A \to BC \in \mathcal{P}$, for some $B, C \in \mathcal{N}^{i-1}$. Then by induction $B^{i-1} \neq \emptyset$ and $C^{i-1} \neq \emptyset$. Consider two strings $u \in B^{i-1}$ and $v \in C^{i-1}$. By definition $u \cdot v \in A^i$, which shows $A^i \neq \emptyset$.

Now suppose $A^i \neq \emptyset$. Consider again two cases:

a) If $A^{i-1} \neq \emptyset$, by induction we know that $A \in \mathcal{N}^{i-1} \subseteq \mathcal{N}^i$.

b) Otherwise, there exists a production $A \to BC \in \mathcal{P}$, for some $B, C \in \mathcal{N}$. There exist strings $u \in B^{i-1}$ and $v \in C^{i-1}$ and therefore $B^{i-1} \neq \emptyset$ and $C^{i-1} \neq \emptyset$. By induction $B, C \in \mathcal{N}^{i-1}$ and therefore $A \in \mathcal{N}^i$, by definition of $\mathcal{N}^i$.

Lemma 5. $\mathcal{N}^{\eta-1} = \mathcal{N}$.

Proof. Consider a non-terminal A and a non-recursive parse tree τ rooted at A. We know τ exists because for each non-terminal N there exists a non-recursive parse tree rooted at N (recall no non-terminal is useless). The depth of τ is at most η, otherwise the parse tree would be recursive. Therefore, $\tau \in A^{i-1}$ and by [Lemma 4], it follows that $A \in \mathcal{N}^{\eta-1}$. Hence, all non-terminals are contained in $\mathcal{N}^{\eta-1}$. The reverse containment is immediate. □

The following corollary shows that every non-terminal contains at least one string in iteration $\eta - 1$.

Corollary 6 (Start-up phase ending). $\forall A \in \mathcal{N} : A^{\eta-1} \neq \emptyset$.

Proof. From [Lemma 5] we know $A \in \mathcal{N}^{\eta-1}$, for each $A \in \mathcal{N}$. After applying [Lemma 4] we obtain $A^{\eta-1} \neq \emptyset$.

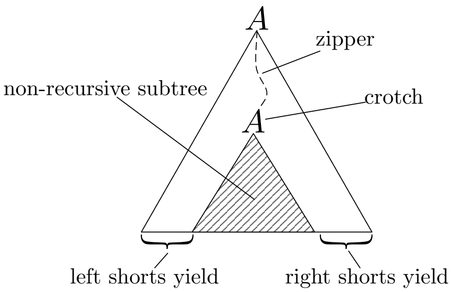

Figure 2: A filled shorts tree for a recursive non-terminal A.

Remark. If $A \in \Delta\mathcal{N}^i$, then the "first" string of A is generated in iteration i:

$$\forall A \in \mathcal{N} : A \in \Delta\mathcal{N}^i \Leftrightarrow \Delta A^i = A^i \neq \emptyset.$$

Remark. After iteration $\eta - 1$, restricted non-terminals do not generate any new strings:

$$A \in \mathcal{N}_{res}, j \geq \eta : \Delta A^j = \emptyset.$$

4.3 Generation pace

In this section we discuss the "speed" at which strings are generated: the *generation pace.*

4.3.1 Recursive Non-terminals

The following definition is illustrated in [Fig. 2].

Definition 7 (Filled shorts tree). Let $A \in \mathcal{N}_{rec}$. A *filled shorts tree* τ for A is a parse tree rooted at A with the following properties:

1. A occurs at least twice (note that the root node is already labelled with A);
2. some non-root A-node is called the *crotch.* The path from the root to the crotch is called the *zipper.* No non-terminal occurs more than once on the zipper, except for A, which appears exactly twice on the zipper;
3. for any non-terminal node x not lying on the zipper, the subtree rooted at x is non-recursive; and

4. the subtree rooted at the crotch is non-recursive.

The *shorts yield* of τ is the yield of τ without the yield of the crotch. The *left shorts yield* (resp. *right shorts yield*) is the yield of τ before (resp. after) the yield of the crotch. The *shorts length* is the length of the shorts yield.

Remark. Every filled shorts tree τ for a recursive non-terminal has a depth of at most 2η. This can be easily seen: Consider a path π in τ from root to leaf. There are now two options:

1. An initial segment of π *equals* the zipper, followed by a path below the crotch. By definition, the zipper contains at most η edges. The path continues in a non-recursive subtree, and therefore has an additional length of at most η. Hence, in this case, the length of π is at most 2η.

2. The path π diverges from the zipper. This means π has at most $\eta - 1$ edges in common with the zipper, after which it follows 1 edge to a non-recursive subtree of depth at most η. Hence, in this case, the length of π is at most $\eta - 1 + 1 + \eta = 2\eta$.

To see that this bound can actually be reached, consider the following grammar:

$$S \to BB$$
$$B \to AA$$
$$A \to SS$$
$$A \to a$$

In this case, $\eta = 3$. Hence, a filled shorts tree has depth at most 6. In [Fig. 3], a filled shorts tree of minimal depth for S that reaches this bound is depicted.

Lemma 8. *For each $A \in \mathcal{N}_{rec}$ there exists a filled shorts tree.*

Proof. Consider a simple path π from A to itself in the dependency graph. The length of π is at most η (only A may appear twice). We now show that we can expand π to a filled shorts tree for A.

Denote the second occurrence of A on π by x_A. Consider a node x_B on π labelled with a node B that is followed directly by a node labelled with C (also on π). There must exist a production $B \to CD$ or $B \to DC$ for some non-terminal D. We know (see [Lemma 5]) that D has a non-recursive parse tree τ_D of depth at most η. Add τ_D as a child of x_B (left or right as indicated by the production). Apply this construction to all nodes on π except for x_A. Replace x_A with a non-recursive parse tree for A. Denote the resulting tree with τ.

It is now clear that τ can serve as a filled shorts tree for A, with π playing the role of the zipper.

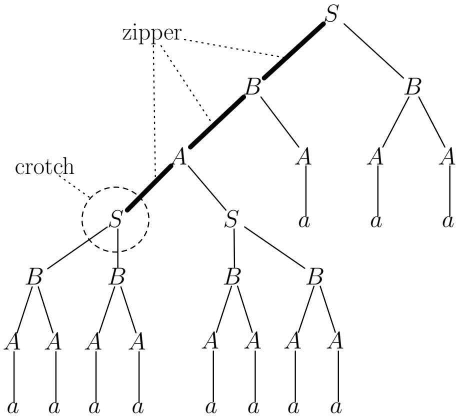

Figure 3: A filled shorts tree for a recursve non-terminal S, having minimal depth.

Lemma 9. *Let $A \in \mathcal{N}_{rec}$, let τ be a filled shorts tree for A of depth δ and zipper length ζ. For all $i \geq 0$, there exists a parse tree τ_i for A, such that*

$$d(\tau_i) = \delta + i \cdot \zeta.$$

Furthermore,

$$|s_i| = \gamma + i \cdot \rho,$$

where s_i is the yield of τ_i, γ is the length of the yield of τ and ρ is the shorts length of τ.

Proof. We first prove the existence of the parse trees τ_i by induction on i.

Base For τ_0, we can take τ itself. By definition this parse tree has depth δ.

Induction Now let $i > 0$ and suppose τ_{i-1} exists: τ_{i-1} is a parse tree rooted at A of depth $\delta + (i-1) \cdot \zeta$. Now replace the crotch of τ, which has label A, with τ_{i-1} to obtain a parse tree τ_i. We write this as $\tau_i = \tau[\tau_{i-1}]$. This is clearly a parse tree for A. Consider a path π from root to leaf. There are two possibilities:

1. The path lies fully in τ. This means the length is at most δ.
2. The path goes through the crotch and the τ_{i-1}-subtree. This means the first part of the path equals the zipper, and the second part has length at most the depth of τ_{i-1}, which equals $\delta + (i-1) \cdot \zeta$. Hence, the length of π is at most $\zeta + \delta + (i-1) \cdot \zeta = \delta + i \cdot \zeta$.

Note that there always is a path of length $\delta + i \cdot \zeta$, because τ_{i-1} has depth $\delta + (i-1) \cdot \zeta$. Hence $d(\tau_i) = \delta + i \cdot \zeta$.

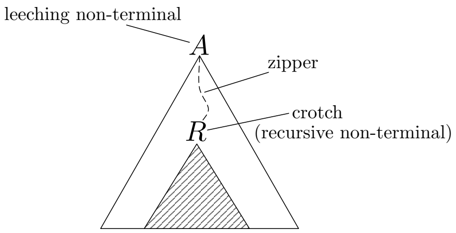

Figure 4: An R-filled shorts tree for a leeching non-terminal A.

It only remains to show that $|s_i| = \gamma + i \cdot \rho$. We show this again by induction.

Base When $i = 0$, obviously $|s_0| = \gamma$, because $\tau_0 = \tau$.

Induction When $i > 0$, suppose the lemma holds for $i - 1$. Denote the left and right shorts yields of τ by u_l and u_r, and their respective lengths by ρ_l and ρ_r. Because $\tau_i = \tau[\tau_{i-1}]$, it holds that

$$s_i = u_l s_{i-1} u_r,$$

where s_{i-1} is the yield of τ_{i-1}. Hence:

$$\begin{aligned} |s_i| &= |u_l| + |s_{i-1}| + |u_r| \\ &= \rho_l + (\gamma + (i-1) \cdot \rho) + \rho_r && \text{(by induction hypothesis)} \\ &= \gamma + (i-1) \cdot \rho + \rho && (\rho_l + \rho_r = \rho) \\ &= \gamma + i \cdot \rho. \end{aligned}$$

Lemma 10. *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{rec}, \forall i \geq 0 : |A^{c+\eta \cdot i}| > i.$$

Proof. Let τ be a filled shorts tree for A of depth δ with a zipper length of ζ. By [Lemma 9], for each $i \geq 0$, we have a different string s_i with a parse tree τ_i of depth $\delta + i \cdot \zeta$, where $\delta \leq 2\eta$ and $\zeta \leq \eta$. Hence, each τ_i yields a new string lastly in iteration $2\eta + i \cdot \eta$. Therefore, $|A^{2\eta+i\cdot\eta}| - 1 \geq i$, or $|A^{2\eta+i\cdot\eta}| > i$, for $i \geq 0$.

4.3.2 Leeching Non-terminals

The following definition is illustrated in [Fig. 4]. In [Definition 7], we have defined the notion of filled shorts tree for a recursive non-terminal. We now define the analogous notion for a leeching non-terminal; this is illustrated in [Fig. 4].

Definition 11 (Filled shorts tree). Let $A \in \mathcal{N}_{leech}$. A *filled shorts tree* for A is a parse tree for A with the following properties:

1. it contains at least one recursive non-terminal;
2. it is non-recursive; and
3. some recursive node is called the *crotch*. The path from the root to the crotch is called the *zipper*.

When the crotch is labeled with a recursive non-terminal R, we call the tree a *R-filled shorts tree* for A. The notions of (left and right) shorts yield, and of shorts length are defined in the same way as in [Definition 7].

Lemma 12. *For each $A \in \mathcal{N}_{leech}$ there exists a filled shorts tree.*

Proof. Consider a simple path π from A to a recursive non-terminal R that consists of only leeching non-terminals (π must exist, because A is leeching). The path has length at most $\eta - 1$. We now show that we can expand π to an R-filled shorts tree for A.

Consider a node x_B on π labelled with a (leeching) node B which is followed by a node labelled with C. There must exist a production $B \to CD$ or $B \to DC$ for some non-terminal D. We know that D has a non-recursive parse tree τ_D of depth at most η. In particular, we know that τ_D does not contain any non-terminals that appear before B on π, else these would be recursive, contradicting our assumption. Add τ_D as a child of x_B (left or right as indicated by the production). Apply this construction to all leeching nodes on π and replace R with a non-recursive parse tree τ_R for R. By similar reasoning, τ_R does not contain any non-terminals that appear before R on π. Denote the resulting tree with τ.

It is obvious that τ is a parse tree for A, which is non-recursive and contains at least one recursive non-terminal (R). Hence, τ is an R-filled shorts tree for A.

Remark. Every filled shorts tree τ for a leeching non-terminal has a depth of at most 2η.

Lemma 13. *Let $A \in \mathcal{N}_{leech}$ and let τ be a B-filled shorts tree for A having depth δ and zipper length ζ. Let τ_B be a filled shorts tree for B having depth δ_B and zipper length ζ_B. For all $i \geq \eta$ there exists a parse tree τ_i for A, such that*

$$d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B.$$

Furthermore

$$|s_i| = \rho + \gamma_B + i \cdot \rho_B,$$

where s_i is the yield of τ_i, ρ is the shorts length of τ, γ_B is the length of the yield of τ_B and ρ_B is the shorts length of τ_B.

Proof. We know from [Lemma 8] and [Lemma 9] that B has parse trees $\tau_{B,j}$, for $j > 0$, such that $d(\tau_{B,j}) = \delta_B + j \cdot \zeta_B$ and $|s_{B,j}| = \gamma_B + j \cdot \rho_B$, where $s_{B,j}$ is the yield of $\tau_{B,j}$. Now, construct the parse tree τ_i, for $i \geq 0$, as follows: replace the crotch of τ by a parse tree $\tau_{B,i}$ for B of depth $\delta_B + i \cdot \zeta_B$. We write this as $\tau_i = \tau_s[\tau_{B,i}]$.

To see that actually $d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B$, consider a path π in τ_i from root to leaf. There are two possibilities:

1. π is fully contained in τ and hence has a maximal depth of η.
2. π contains the crotch and ends in a leaf in $\tau_{B,i}$. In this case, π consists of the zipper, followed by a path of length at most $\delta_B + i \cdot \zeta_B$. The total length of π is at most $\zeta + \delta_B + i \cdot \zeta_B$.

Because $d(\tau_{B,i}) = \delta_B + i \cdot \zeta_B$, there exists at least one path in τ_i that has length $\zeta + \delta_B + i \cdot \zeta_B$. Hence $d(\tau_i) \geq \zeta + \delta_B + i \cdot \zeta_B$. Since clearly $\zeta + \delta_B + i \cdot \zeta_B \geq \eta$ for $i \geq \eta$, we obtain $d(\tau_i) \leq \zeta + \delta_B + i \cdot \zeta_B$. We may now conclude that $d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B$.

Finally, by construction, it is clear that $|s_i| = \rho + |s_{B,i}| = \rho + \gamma_B + i \cdot \rho_B$.

Remark. In the previous lemma, we can use the same construction to get a different parse tree for each $i < \eta$, all yielding different strings. But, when $i < \eta$, the depth of the parse tree is equal to $\max(d(\tau), \zeta + \delta_B + i \cdot \zeta_B)$. □

The next lemmas show a relationship between the iteration number and the number of generated strings up to that iteration.

Lemma 14. *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{leech}, \forall i \geq 0 : |A^{c+\eta \cdot i}| > i.$$

Proof. Consider a recursive non-terminal R, reachable by A, having a shorts tree (by [Lemma 12]) of depth δ_R with a zipper length of ζ_R. By [Lemma 13] and [Lemma 12], for each $i \geq \eta$, we have a different string with a parse tree of depth $\zeta + \delta_R + i \cdot \zeta_R$, where $\zeta \leq \eta - 1$, $\delta_R \leq 2\eta$ and $\zeta_R \leq \eta$. Hence, each τ_i yields a new string lastly in iteration $\eta - 1 + 2\eta + i \cdot \eta = 3\eta + i \cdot \eta - 1$. Therefore, $|A^{3\eta+i\cdot\eta-1}| - 1 \geq i - \eta$, or $|A^{3\eta+i\cdot\eta-1}| > i - \eta$, for $i \geq \eta$. By substituting i' for $i - \eta$ we obtain that, for $i' \geq 0$, $|A^{3\eta+(i'+\eta)\cdot\eta-1}| > i'$, or $|A^{3\eta+i'\cdot\eta+\eta^2-1}| > i'$.

Corollary 15 (Non-terminal lower bound). *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}, \forall i \geq 0 : \eta \cdot |A^{i+c}| > i.$$

Proof. Consider the constants c_1 and c_2 respectively from [Lemma 10] and [Lemma 14]. We can choose c_3 as the maximum of c_1 and c_2 and get:

$$\forall A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}, \forall k \geq 0 : |A^{c_3+\eta\cdot k}| > k.$$

Now let $A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}$ and i be an arbitrary natural number. We distinguish the following two cases to show that the corollary holds:

(a) i is a multiple of η.
Hence, $i = \eta \cdot k$. We can now argue as follows:

$$\begin{aligned} |A^{c_3+\eta\cdot k}| > k &\Leftrightarrow |A^{c_3+\eta\cdot\frac{i}{\eta}}| > \frac{i}{\eta} \\ &\Leftrightarrow \eta \cdot |A^{c_3+i}| > i \end{aligned}$$

(b) i is not a multiple of η.
Let i' be the next multiple of η larger than i. We have $i < i' < i + \eta$ and by (a) we know that $i' < \eta \cdot |A^{c_3+i'}|$. Also $|A^j| \leq |A^{j'}|$ when $j \leq j'$, since $A^j \subseteq A^{j'}$. Combining these observations we get:

$$i < i' < \eta \cdot |A^{c_3+i'}| < \eta \cdot |A^{c_3+i+\eta}|.$$

When we choose $c = c_3 + \eta$, we have $\eta \cdot |A^{i+c}| > i$, for $i \geq 0$.

Corollary 16 (Intermediate lower bound). *There exists a constant c such that*

$$\forall i \geq 0 : \eta \cdot |\mathcal{T}^{i+c}| \geq i.$$

Proof. Fix some arbitrary $A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}$. Clearly $|\mathcal{T}^j| \geq |A^j|$ for all j. Then, by [Corollary 15] there exists a constant c such that:

$$\eta \cdot |\mathcal{T}^{i+c}| \geq \eta \cdot |A^{i+c}| \geq i.$$

4.4 Length bound

Lemma 17. *Let $s \in A^i$ with $|s| \geq 2^\eta$. Then A^i also contains a shorter string s' with*

$$|s| - 2^\eta < |s'| < |s|.$$

Proof. Consider a parse tree τ for s of depth δ. Since $|s| \geq 2^\eta$, it follows from [Lemma 2] that $\delta \geq \eta + 1$.

Let π be a path of maximal length from root to leaf in τ. Let π' be the final segment of π of length $\eta + 1$ (π has length $\geq \eta + 1$ since $\delta \geq \eta + 1$). On π', some non-terminal B occurs more than once; let nodes x and y, in that order on π', be labelled with B. The yields of the subtrees rooted at x and y are denoted by s_x and s_y respectively.

In τ we can now replace the subtree τ_x rooted at x with the subtree τ_y rooted at y. Since the depth of τ' is at most that of τ, and $s \in A^i$, it follows from [Lemma 1] that also $s' \in A^i$. The resulting parse tree τ' has a yield s' with length

$$|s'| = |s| - |s_x| + |s_y|. \tag{1}$$

Every node on the path from x to y (x included, y excluded) has precisely two non-terminal children: one ancestor of y and one non-ancestor of y. Each subtree rooted at a non-ancestor of y has a non-empty yield. It follows that the yield of τ_y is a strict substring of the yield of τ_x and hence $|s_y| - |s_x| < 0$. It now follows from (1) that

$$|s'| < |s|. \tag{2}$$

Furthermore, since the depth of τ_x is at most $\eta + 1$, the string yielded by τ_x has length at most 2^η: $|s_x| \leq 2^\eta$. We also know that τ_y yields a string of at least length 1. It now follows from (1) that

$$|s'| \geq |s| - 2^\eta + 1 \geq |s| - 2^\eta. \tag{3}$$

Combining (2) and (3) gives us the desired lengths bounds for s'.

Lemma 18. $\forall A \in \mathcal{N}, \forall i \geq \eta - 1 : \omega^i_A < 2^\eta \cdot |A^i|$.

Proof. Since $i \geq \eta - 1$, we know by [Corollary 6] that $|A^i| \geq 1$. If $\omega^i_A < 2^\eta$ then the lemma is trivial. Else, let $j = \lfloor \omega^i_A / 2^\eta \rfloor$. [Lemma 17] can be repeatedly applied at least j times, starting from $s_0 = s$, yielding j additional distinct strings $s_1, s_2, \ldots, s_j \in A^i$. Hence, $|A^i| \geq j + 1$, and therefore $|A^i| > \omega^i_A / 2^\eta$.

Corollary 19 (Length bound). $\forall i \geq \eta - 1 : \omega^i_{\mathcal{T}} < 2^\eta \cdot |\mathcal{T}^i|$.

Proof. For a given $i \geq \eta - 1$, let $s \in \mathcal{T}^i$ be a string of maximal length: $s = \omega^i_{\mathcal{T}}$. By definition, $s \in A^i$ and $|s| = \omega^i_A$, for some $A \in \mathcal{N}$ (otherwise s could not have maximal length). Now by [Lemma 18] we know obtain the desired inequality:

$$\omega^i_{\mathcal{T}} = |s| = \omega^i_A < 2^\eta \cdot |A^i| \leq 2^\eta |\mathcal{T}^i|.$$

4.5 Intermediate string bound

In this section we bound the number of intermediate strings by the number of output strings.

Lemma 20 (Past bound). $\forall i \geq 0 : \forall 0 \leq k \leq i : |\mathcal{T}^i| \leq 2^{2^k - 1} \cdot |\mathcal{T}^{i-k}|^{2^k}$.

Proof. We prove the lemma by induction on i.

Basis For $i = 0$, the only possible value for k is 0 and the inequality $|\mathcal{T}^0| \leq 2^{2^0 - 1} \cdot |\mathcal{T}^0|^{2^0}$ becomes trivial.

Induction For $i > 0$, assume the lemma holds for $i - 1$:

$$|\mathcal{T}^{i-1}| \leq 2^{2^k - 1} \cdot |\mathcal{T}^{i-1-k}|^{2^k} \qquad (0 \leq k \leq i - 1).$$

Now note that for all $j \geq 0 : \mathcal{T}^{j+1} \subseteq (\mathcal{T}^j \cdot \mathcal{T}^j) \cup \mathcal{T}^0$. This is immediate from the iteration schema. We get:

$$\begin{aligned}
|\mathcal{T}^i| &\leq |(\mathcal{T}^{i-1} \cdot \mathcal{T}^{i-1}) \cup \mathcal{T}^0| \\
&\leq |\mathcal{T}^{i-1}| \cdot |\mathcal{T}^{i-1}| + |\mathcal{T}^0| \\
&= |\mathcal{T}^{i-1}|^2 + |\mathcal{T}^0| \\
&\leq 2 \cdot |\mathcal{T}^{i-1}|^2 && (\mathcal{T}^0 \subseteq \mathcal{T}^{i-1}) \\
&\leq 2 \cdot \left(2^{2^k-1} \cdot |\mathcal{T}^{i-1-k}|^{2^k}\right)^2 && \text{(by induction hypothesis, } 0 \leq k \leq i-1) \\
&= 2^{2^{k+1}-1} \cdot |\mathcal{T}^{i-(k+1)}|^{2^{k+1}} && (0 \leq k \leq i-1) \\
&= 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k} && (1 \leq k \leq i) \\
&= 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k} && (0 \leq k \leq i, k = 0 \text{ is immediate})
\end{aligned}$$

Now by the principle of induction the lemma holds.

Lemma 21 (String Growth). *For each edge $A \rightarrow B$ in the dependency graph the following holds:*

$$\forall i \geq \eta - 1 : \forall s \in B^i : \exists s' \in A^{i+1} : s \textit{ is a strict substring of } s'.$$

Proof. In iteration $i \geq \eta - 1$, every non-terminal contains at least one non-empty string (Corollary 6). The edge from A to B indicates the presence of either a rule $A \rightarrow BC$ or $A \rightarrow CB$ for some non-terminal C. The string $s \in B^i$ will be concatenated with a string $u \in C^i$ to form some string $s' \in A^{i+1}$, where $s' = s \cdot u$ or $s' = u \cdot s$, depending on the production. Clearly s is a strict substring of s'.

Lemma 22 (Future bound). $\forall i \geq \eta - 1 : |\mathcal{T}^i| \leq (\omega_S^{i+\eta})^2 \cdot |S^{i+\eta}|$.

Proof. We first prove that every intermediate string will appear as a substring of an output string, several iterations later. Next, we bound the number of substrings of these output strings to obtain the desired bound.

Any string $s \in \mathcal{T}^i$ appears in A^i for some A. Consider a simple path π from S to A in the dependency graph; π has length at most η. By repeatedly applying [Lemma 21] we know that $S^{i+l(\pi)}$ contains a string s' that is a superstring of s. Because $l(\pi) \leq \eta$ it holds that $s' \in S^{i+l(\pi)} \subseteq S^{i+\eta}$. Hence, each string in $\mathcal{T}^i$ has a superstring in $S^{i+\eta}$.

The number of substrings of a string s' is bounded [see 1] by $|s'|^2$. Consequently, the number of substrings we can create using strings in $S^{i+\eta}$ is bounded by $(\omega_S^{i+\eta})^2 \cdot |S^{i+\eta}|$. Together with the first observation, this gives:

$$|\mathcal{T}^i| \leq (\omega_S^{i+\eta})^2 \cdot |S^{i+\eta}|.$$

[1] A much more precise bound could be used, but this would not improve our results.

Corollary 23 (Present bound). $\exists c \in \mathbb{N} : \forall i \geq \eta - 1 : |\mathcal{T}^i| \leq c \cdot |S^i|^{2^{\eta+2}}$.

Proof. We combine the Future and Past bounds to bound $|\mathcal{T}^i|$:

$$\begin{aligned}
|\mathcal{T}^i| &\leq 2^{2^\eta - 1} \cdot |\mathcal{T}^{i-\eta}|^{2^\eta} && \text{(Past bound)} \\
&\leq 2^{2^\eta - 1} \cdot \left((\omega_S^i)^2 \cdot |S^i| \right)^{2^\eta} && \text{(Future bound)} \\
&= 2^{2^\eta - 1} \cdot (\omega_S^i)^{2^{\eta+1}} \cdot |S^i|^{2^\eta} \\
&\leq 2^{2^\eta - 1} \cdot (2^\eta \cdot |S^i|)^{2^{\eta+1}} \cdot |S^i|^{2^\eta} && \text{(Lemma 18)} \\
&\leq c \cdot |S^i|^{2^{\eta+2}}
\end{aligned}$$

Remark (Present length bound). Although we will not make use of the following theorem, we note out of interest:

$$\forall i \geq \eta - 1 : \omega_{\mathcal{T}}^i \leq (\omega_S^i)^{2^\eta}.$$

Proof. Consider a string s in $\mathcal{T}^i$ having a length of $\omega_{\mathcal{T}}^i$, with $s \in A^i$. In the dependency graph, there exists a simple path π from S to A having a length at most η. This means there exists a string $s' \in S^{i+\eta}$ that is a *strict* superstring of s (Lemma 21). Therefore:

$$\omega_{\mathcal{T}}^i < \omega_S^{i+\eta}.$$

On the other hand, it is obvious that in the worst case, the maximal string length doubles each iteration, therefore:

$$\omega_{\mathcal{T}}^i \leq (\omega_{\mathcal{T}}^{i-\eta})^{2^\eta}.$$

Combining the inequalities above gives us:

$$\omega_{\mathcal{T}}^i \leq (\omega_{\mathcal{T}}^{i-\eta})^{2^\eta} < (\omega_S^{i-\eta+\eta})^{2^\eta} = (\omega_S^i)^{2^\eta}.$$

5 Complexity

In order to prove that our naive generation algorithm yields an enumeration in *incremental polynomial time* in the sense of [Johnson et al. 1988], we only require the following proposition, which we prove using the results above:

Proposition 24. *There exists a fixed polynomial p such that after each iteration i, the total time spent by the algorithm so far is bounded by $p(|S^{i-1}|)$.*

Proof. We will first look at the time necessary to generate one string, then at the time necessary to generate one iteration and finally at the time needed to generate strings up to an iteration i.

Consider an intermediate string $s \in A^i$. When $i = 0$, the only things that needs to happen is to store s, given that there are no duplicate productions. When $i > 0$, the following steps need to be performed:

1. concatenate two strings to form s;
2. check if the string has already been generated for A (duplicate check);
3. save the string in order to check for duplicates later.

The concatenation of two strings, resulting in s, can be done in time $\mathcal{O}\left(|s|\right)$. A lookup and insertion, to keep track of the string, can both be done in time $\mathcal{O}\left(|A^i| \cdot |s|\right)$ [see 1].

Next, we construct a bound for the total number of intermediate strings calculated in iteration $i > 0$. In the worst case, all strings in $\mathcal{T}^{i-1}$ will be pairwise combined, for each production. Hence, the total number of candidates in iteration i is bounded by

$$\rho \cdot |\mathcal{T}^{i-1}|^2,$$

where ρ is equal to the number of productions in the grammar.

Combining the two observations above gives us an upper bound on the total work in iteration i:

$$\mathcal{O}\left(\omega_{\mathcal{T}}^{i} \cdot |\mathcal{T}^i| \cdot |\mathcal{T}^{i-1}|^2\right). \tag{4}$$

From the Past bound we know that $|\mathcal{T}^i| = \mathcal{O}\left(|\mathcal{T}^{i-1}|^2\right)$. The total work done up to and including iteration i is therefore bounded by

$$\mathcal{O}\left(\sum_{j=1}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4\right).$$

Note that the work in iteration 0 is constant, since it requires storing just one string for each terminal production. The work in the first η iterations is also bounded by a constant:

$$\sum_{j=1}^{\eta-1} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 \leq \eta \cdot \omega_{\mathcal{T}}^{\eta} \cdot |\mathcal{T}^{\eta-1}|^4 = \mathcal{O}\left(1\right).$$

Hence, the total time spent up to and including iteration $\eta - 1$ is considered constant.

In the remainder of the proof, we bound $\mathcal{O}\left(\sum_{j=\eta}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4\right)$ by a poly-

[1] Actually, a much better bound can be obtained, but for other reasons the algorithm will be polynomial anyway.

nomial in $|S^{i-1}|$. First, observe the following:

$$\begin{aligned}
\sum_{j=\eta}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 &\leq i \cdot \omega_{\mathcal{T}}^{i} \cdot |\mathcal{T}^{i-1}|^4 \\
&< c_1 \cdot i \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^4 && \text{(Corollary 19)} \\
&\leq c_2 \cdot |\mathcal{T}^{i+c_3}| \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^4 && \text{(Corollary 16)} \\
&\leq c_4 \cdot |\mathcal{T}^{i-1}|^{c_5} && \text{(Past bound)} \\
&\leq c_6 \cdot |S^{i-1}|^{c_7} && \text{(Present bound)}
\end{aligned}$$

for constants $c_1, \ldots, c_7$.

Note that the applied lemmas only hold from iteration $\eta - 1$ on. This is not a problem as they are only applied for $j \geq \eta$. From the above we can conclude:

$$\sum_{j=1}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 = \mathcal{O}\left(|S^{i-1}|^c\right),$$

for some constant c.

The time needed by the algorithm to calculate all intermediate strings up to and including interation i is bounded by $\mathcal{O}\left(|S^{i-1}|^c\right)$, which is clearly polynomial in the size of S^{i-1}, as desired.

Theorem 25. *There is a fixed polynomial p such that the entire language $L(G)$ can be enumerated without duplicates in such a way that the time needed to output the $n+1$th output string is bounded by $p(n)$.*

Proof. Consider the $n+1$th output string s. We know that $s \in \Delta S^i$ for some i and we also know, by [Proposition 24] that the time needed to calculate all strings up to and including iteration i is bounded by $\mathcal{O}\left(|S^{i-1}|^c\right)$, for some constant c. Since $|S^{i-1}| \leq n$, we obtain a polynomial in n as desired.

6 From Given-Length to Infinite Enumeration

The purpose of this section is to show that we can always use an algorithm for given-length enumeration with polynomial delay (GLEPD) to obtain an algorithm for infinite enumeration in incremental polynomial time (IEIPT).

For a fixed context-free grammar G, consider a GLEPD-algorithm that, given a natural number n, enumerates all strings $w \in L(G)$ with $|w| = n$. We treat the algorithm as a black box and denote it by $\texttt{Enumerate}_G(n)$. The polynomial delay property holds for the algorithm: there exists a fixed polynomial p_D such that, on input n, the time before the first output, the time between two outputs and the time after the last output until the algorithm terminates, is bounded by $p_D(n)$.

```
Input: None
Output: all strings in $L(G)$
for $i \leftarrow 1$ to $\infty$ do
    $\texttt{Enumerate}_G(i)$
end
```

Figure 5: The algorithm $\texttt{Enumerate}_{G,\infty}$.

From this algorithm, we can derive the algorithm $\texttt{Enumerate}_{G,\infty}$ [Fig. 5]. In the remainder of this section, we prove that $\texttt{Enumerate}_{G,\infty}$ enumerates the entire language $L(G)$ in IPT.

Lemma 26. *For each infinite context-free language $\mathcal{L}$, there exist two constants $c \in \mathbb{N} \setminus \{0\}$ and $d \in \mathbb{N}$ such that for each $l \in \mathbb{N}$ the language $\mathcal{L}$ contains at least one string of length $c \cdot l + d$.*

Proof. Consider a context-free grammar G such that $L(G) = \mathcal{L}$. Let S be the start symbol of G. We know S must be recursive or leeching for $L(G)$ to be infinite. From [Lemma 9] and [Lemma 13] we know that there exist two constants $c, d \in \mathbb{N}$ such that for each $l \in \mathbb{N}$ there is a string $s \in L(G)$ with $|s| = c \cdot l + d$.

Theorem 27 ($\texttt{Enumerate}_{G,\infty}$ runs in incremental polynomial time). *Let G be a context-free grammar. There exists a fixed polynomial p such that in the algorithm $\texttt{Enumerate}_{G,\infty}$ the time spent between the mth output and the $m+1$th output is bounded by $p(m)$, where $m > 0$.*

Proof. Consider the mth and the $m+1$th output strings that are generated consecutively by the algorithm and denote them by s_m and s_{m+1} respectively. For algorithm $\texttt{Enumerate}_G(n)$ we have the polynomial $p_D(n)$, guaranteed by the polynomial delay property. We may assume p_D is monotonically increasing over the natural numbers. [see 1]

There are two cases to consider:

- $|s_m| = |s_{m+1}|$.
  This means that the strings are generated in the same iteration i. Let c and d be the constants given by [Lemma 26]. We consider two further cases:
  (a) $i \leq d$.
  Let c_0 be the total time performed by algorithm $\texttt{Enumerate}_{G,\infty}$ in the iterations up to and including iteration d. Then clearly the time between the outputs s_m and s_{m+1} is bounded by c_0.

[1] This can be achieved by converting all negative coefficients to positive.

(b) $i > d$.

Let $l = \lfloor \frac{i-1-d}{c} \rfloor$. By [Lemma 26], at least $l+1$ strings have already been generated before iteration i. Hence $l + 1 < m$. We can now argue as follows:

$$\begin{aligned} l+1 < m \Rightarrow l < m & \\ &\Leftrightarrow \left\lfloor \frac{i-1-d}{c} \right\rfloor < m \\ &\Rightarrow \frac{i-1-d}{c} < m \\ &\Leftrightarrow i < m \cdot c + d + 1 \\ &\Leftrightarrow i \leq m \cdot c + d. \end{aligned}$$

As the time between s_m and s_{m+1} is bounded by $p_D(i)$, it is also bounded by $p_D(m \cdot c + d)$, because p_D is monotonically increasing. This is clearly a polynomial in m.

– $|s_m| < |s_{m+1}|$.

This means that the strings are generated in different iterations. Let i be the iteration in which s_m was generated and j be the iteration in which s_{m+1} was generated. Clearly, $1 \leq i < j$. The total time spent between outputs s_m and s_{m+1} consists of three parts:

- the time spent in iteration i after the generation of s_m;
- the time spent in iteration j before the generation of s_{m+1};
- the time spent in iterations $i + 1 \ldots j - 1$.

The first two parts are bounded by $p_D(i)$ and $p_D(j)$ respectively. In every iteration k between i and j, the time needed to verify that there is no string of length k in $L(G)$ is bounded by $p_D(k)$. Hence, the total time spent between s_m and s_{m+1} is bounded by

$$p_D(i) + p_D(j) + \sum_{k=i+1}^{j-1} p_D(k) \leq p_D(j) + p_D(j) + (j-2) \cdot p_D(j) = j \cdot p_D(j).$$

We know from [Lemma 26] that the maximal number of consecutive lengths for which no string exists is bounded by a constant. Hence, for some constant c_{wait} we have $j - i \leq c_{\text{wait}}$. The total time spent between s_m and s_{m+1} is therefore bounded by

$$p'(i) := (i + c_{\text{wait}}) \cdot p_D(i + c_{\text{wait}}),$$

which is clearly a polynomial in i. As in the previous case, $i \leq c \cdot m + d$, so we obtain $p'(c \cdot m + d)$ as a polynomial in m.

The proof is concluded by taking for $p(m)$ the larger of the two polynomials from the two cases, increased by the constant c_0.

7 Conclusion

The fact that the simple algorithm, based on the naive bottom-up concatenation scheme and described in [Proposition 24], already achieves the Incremental Polynomial Time criterion, is, we hope, an interesting theoretical (if not didactical) contribution of this paper, as we have not seen this noted elsewhere. Moreover, an elementary approach as presented here has the best chances of being generalizable. Indeed, we are currently investigating how the insights developed here can be extended to apply to the more general setting of context-free sets of arbitrary combinatorial objects as introduced by [Courcelle and Engelfriet 2012] and [Flajolet and Sedgewick 2009, Flajolet et al. 1991]. A major additional problem in this context is to keep the duplicate check (step 2 in the proof of Proposition 24) polynomial. Fortunately, in the HR approach to graph rewriting, every context-free graph language has bounded treewidth. In combination with imposing connectedness and a degree bound [see Matoušek and Thomas 1992] this may produce a polynomial duplicate check.

We also note that for unambiguous grammars, the methods of [Flajolet and Sedgewick 2009] can be used to count exactly the number of strings (or derivation trees, which coincides for unambiguous grammars) of a given size.

Acknowledgement

We thank Hendrik Blockeel and Frank Neven for their contributions in the initial phase of the project.

References

[Arnold and Sleep 1980] Arnold, D. B., Sleep, M. R.: "Uniform Random Generation of Balanced Parenthesis Strings"; ACM Trans. Program. Lang. Syst.; 2, 1 (1980), 122–128.

[Baeten et al. 1993] Baeten, J. C. M., Bergstra, J. A., Klop, J. W.: "Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages"; J. ACM; 40, 3 (1993), 653–682.

[Bancilhon and Ramakrishnan 1986] Bancilhon, F., Ramakrishnan, R.: "An Amateur's Introduction to Recursive Query Processing Strategies"; SIGMOD Rec.; 15, 2 (1986), 16–52.

[Ceri et al. 1990] Ceri, S., Gottlob, G., Tanca, L.: "Logic Programming and Databases"; Springer (1990).

[Courcelle and Engelfriet 2012] Courcelle, B., Engelfriet, J.: “Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach”; volume 138 of Encyclopedia of mathematics and its applications; Cambridge University Press (2012).

[Dömösi 2000] Dömösi, P.: “Unusual Algorithms for Lexicographical Enumeration”; Acta Cybern.; 14, 3 (2000), 461–468.

[Dong 2009] Dong, Y.: “Linear algorithm for lexicographic enumeration of CFG parse trees.”; Science in China Series F: Information Sciences; 52, 7 (2009), 1177–1202.

[Duncan and Hutchinson 1981] Duncan, A. G., Hutchinson, J.: “Using Attributed Grammars to Test Designs and Implementations”; Proceedings 5th International Conference on Software Engineering; IEEE Press (1981); 170–178.

[Flajolet et al. 1991] Flajolet, P., Salvy, B., Zimmermann, P.: “Automatic Average-Case Analysis of Algorithm”; Theor. Comput. Sci.; 79, 1 (1991), 37–109.

[Flajolet and Sedgewick 2009] Flajolet, P., Sedgewick, R.: “Analytic Combinatorics”; Cambridge University Press (2009).

[Flajolet et al. 1994] Flajolet, P., Zimmermann, P., Cutsem, B. V.: “A Calculus for the Random Generation of Labelled Combinatorial Structures”; Theor. Comput. Sci.; 132, 2 (1994), 1–35.

[Gore et al. 1997] Gore, V., Jerrum, M., Kannan, S., Sweedyk, Z., Mahaney, S. R.: “A Quasi-Polynomial-Time Algorithm for Sampling Words from a Context-Free Language”; Inf. Comput.; 134, 1 (1997), 59–74.

[Hopcroft and Ullman 1979] Hopcroft, J. E., Ullman, J. D.: “Introduction to Automata Theory, Languages, and Computation”; Addison-Wesley, Reading, Massachusetts (1979).

[Johnson et al. 1988] Johnson, D. S., Yannakakis, M., Papadimitriou, C. H.: “On Generating All Maximal Independent Sets”; Information Processing Letters; 27, 3 (1988), 119–123.

[Lämmel 2001] Lämmel, R.: “Grammar Testing”; FASE; (2001); 201–216.

[Mäkinen 1997] Mäkinen, E.: “On Lexicographic Enumeration of Regular and Context-Free Languages”; Acta Cybern.; 13, 1 (1997), 55–62.

[Matoušek and Thomas 1992] Matoušek, J., Thomas, R.: “On the complexity of finding iso- and other morphisms for partial k-trees”; Discrete Mathematics; 108, 1-3 (1992), 343–364.

[Maurer 1990] Maurer, P. M.: “Generating Test Data with Enhanced Context-Free Grammars.”; IEEE Software; 7, 4 (1990), 50–55.

[Purdom 1972] Purdom, P.: “A Sentence Generator for Testing Parsers”; j-BIT; 12, 3 (1972), 366–375.

[Somerville 1998] Somerville, I.: “Software Engineering”; Addison-Wesley (1998); 5th edition.

[Xu et al. 2011] Xu, Z., Zheng, L., Chen, H.: “A Toolkit for Generating Sentences from Context-Free Grammars”; Int. J. Software and Informatics; 5, 4 (2011), 659–676.