Theoretical Informatics and Applications
Theoret. Informatics Appl. **35** (2001) 499–512

RANDOM GENERATION FOR FINITELY AMBIGUOUS CONTEXT-FREE LANGUAGES*, **

Alberto Bertoni[1], Massimiliano Goldwurm[1] and Massimo Santini[2]

Abstract. We prove that a word of length n from a finitely ambiguous context-free language can be generated at random under uniform distribution in $O(n^2 \log n)$ time by a probabilistic random access machine assuming a logarithmic cost criterion. We also show that the same problem can be solved in polynomial time for every language accepted by a polynomial time 1-NAuxPDA with polynomially bounded ambiguity.

Mathematics Subject Classification. 68Q45, 68Q25.

Introduction

Given a formal language L, consider the problem of generating, for an instance $n \in \mathbb{N}$, a word of length n in L uniformly at random. Observe that here the language is not part of the input and it is defined by some specific formalism (for instance a generating grammar). In the literature this problem has been studied in particular for unambiguous context-free languages [8, 10, 11, 15, 20]. For such languages, the algorithm with the best time complexity in the worst case generates, uniformly at random, a word of size n in the language in $O(n \log n)$ time [10]. This bound is in terms of arithmetic complexity: each step of the algorithm can require an arithmetic operation over $O(n)$-bits integers or it can generate in constant

* *The results presented in this work appeared in a preliminary form in A. Bertoni, M. Goldwurm, M. Santini, "Random Generation and Approximate Counting of Ambiguously Described Combinatorial Structures", Proc. STACS2000, LNCS No. 1770, 567-581.*

** *This research has been supported by project M.I.U.R. COFIN "Linguaggi formali e automi: teoria e applicazioni".*

[1] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, via Comelico 39/41, 20135 Milano, Italy; e-mail: bertoni@dsi.unimi.it & goldwurm@dsi.unimi.it

[2] Dipartimento di Scienze Sociali, Cognitive e Quantitative, Università degli Studi di Modena e Reggio Emilia, Via Giglioli Valle, 42100 Reggio Emilia, Italy; e-mail: msantini@unimo.it

time an integer of $O(n)$ bits uniformly at random. This result is obtained in a more general context concerning the random generation of labeled combinatorial structures defined by unambiguous formal specifications that use operations of union, Cartesian product, construction of sets, sequences and cycles [10].

On the contrary, uniform random generation for structures represented by ambiguous formalisms has not received much attention in the literature. An analysis of the complexity of uniform random generation for combinatorial structures defined by polynomial time relations is given in [17]: the authors give some evidence that, under suitable hypotheses, (almost uniform) random generation is easier than counting, but more difficult than recognizing. Following a similar approach, a subexponential time algorithm is presented in [12] for the (almost uniform) random generation of words in a (possibly ambiguous) context-free language.

In this work we study the problem for languages defined by a formal specification of bounded ambiguity. As a model of computation (discussed in Sect. 1) we assume a probabilistic random access machine, under logarithmic cost criterion, which can only use unbiased coin flips for generating random numbers. Thus, the corresponding complexity bounds take into account both the size of the operands and the number of random bits used in the computation. Our main result states that for *finitely ambiguous context-free languages*, a word of length n can be generated uniformly at random in $O(n^2 \log n)$ time and $O(n^2)$ space, using $O(n^2 \log n)$ random bits. We observe that the same bounds for time and random bits are obtained for unambiguous context-free languages by applying the procedures described in [10] assuming our model of computation.

The proof of this result is based on a multiplicity version of Earley's algorithm for context-free recognition [9], presented in details in Section 2.1; in the case of finitely ambiguous context-free languages, this procedure computes the number of derivation trees of an input word of size n in $O(n^2 \log n)$ time and $O(n^2)$ space.

We extend this result to languages accepted by *one-way nondeterministic auxiliary push-down automata* working in polynomial time and using a logarithmic amount of work-space [4, 7]. In this case, we obtain a polynomial time random uniform generator whenever the automaton has a polynomial number of accepting computations for each input word.

1. Preliminary notions

In this work, as model of computation, we assume a Probabilistic Random Access Machine (PrRAM for short) under which the complexity of a procedure takes into account the number of random bits used by the computation; a similar model is implicitly assumed in [18] to study the complexity of random number generation from arbitrary distributions.

Formally, our model is a Random Access Machine [1] equipped in addition with a 1-way read-only *random tape* and an instruction RND. The random tape contains a sequence $r = r_0, r_1 \ldots$ of symbols in $\{0, 1\}$ and, in the initial configuration, its head scans the first symbol. During the computation, the instruction "RND i"

transfers the first i unread bits from the random tape into the accumulator and moves the tape head i positions ahead. For a fixed sequence r on the random tape and a given input x, the output $M(x,r)$ of the computation of a PrRAM M, is defined essentially as in the standard RAM model; furthermore, by $M(x)$ we mean the random variable denoting the output $M(x,r)$ assuming r a sequence of independent random variables such that $\Pr\{r_i = 1\} = \Pr\{r_i = 0\} = 1/2$ for every $i \geq 0$. Hence the instruction "RND i" generates an integer in $\{0, \ldots, 2^i - 1\}$ uniformly at random. To evaluate the space and time complexity of a PrRAM computation we adopt the logarithmic cost criterion defined in [1] for the standard RAM model, assuming, in addition, a time cost i for every instruction "RND i". Due to the restriction to unbiased coins, an algorithm in this model may fail to give the correct answer and in this case it outputs a conventional symbol $\perp$.

We think this machine takes the advantages of the two main models considered in connection with the random generation of combinatorial structures, *i.e.* the "arithmetic" machine assumed in [10] and the probabilistic Turing machine used in [17]. From one side, it is suited for the specification of algorithms at high level allowing an easy analysis of time and space complexity. From the other, it also allows to carry out a somehow realistic analysis of the procedures that does not neglect the size of operands, trying to reasonably satisfy the principle that every elementary step of the machine be implementable in constant time by some fixed hardware.

In this work we are mainly interested in the random generation of words from a given language. Given a finite alphabet Σ (such that $\perp \notin \Sigma$) and a language $L \subseteq \Sigma^*$, for every $n \in \mathbb{N}$, let L_n be the set $\{x \in L : |x| = n\}$ and let $C_L(n)$ be the cardinality of L_n. An algorithm A is a *uniform random generator* (u.r.g.) for a language L if, for every $n > 0$, the following conditions hold:

1. A on input n returns a value $\mathsf{A}(n) \in L_n \cup \{\perp\}$,
2. $\Pr\{\mathsf{A}(n) = x \mid \mathsf{A}(n) \neq \perp\} = C_L(n)^{-1}$ for every $x \in L_n$ and
3. $\Pr\{\mathsf{A}(n) = \perp\} < 1/4$.

Observe that the constant $1/4$ in the previous definition can be replaced by any positive number strictly less than 1, leaving the definition substantially unchanged. More precisely, assume there is an algorithm A, satisfying points 1) and 2) above, such that $\Pr\{\mathsf{A}(n) = \perp\} < \delta$, for some $0 < \delta < 1$; then, for every $0 < \delta' < 1$, there exists an algorithm A' satisfying the same conditions 1) and 2) such that $\Pr\{\mathsf{A}'(n) = \perp\} < \delta'$. Moreover, if A works in $T_{\mathsf{A}}(n)$ time and uses $R_{\mathsf{A}}(n)$ random bits, then A' works in time $O(T_{\mathsf{A}}(n))$ and uses $O(R_{\mathsf{A}}(n))$ random bits.

As an example, we describe a u.r.g. for unambiguous context-free languages obtained by adapting a well-known procedure for the random generation of combinatorial structures proposed in [10]. Our purpose here is to evaluate its time complexity with respect to our model of computation.

1.1. Unambiguous context-free languages

Let $G = \langle V, \Sigma, S, P\rangle$ be a context-free (*c.f.* for short) grammar, where V is the set of nonterminal symbols (we also call variables), Σ the alphabet of terminals,

$S \in V$ the initial variable and P the family of productions. We assume G in Chomsky normal form [16] without useless variables, *i.e.* every nonterminal appears in a derivation of some terminal word from the initial symbol S. It is well-known that every *c.f.* language not containing the empty word ϵ can be generated by such a grammar.

For every $A \in V$, let L_A be the set $\{x \in \Sigma^* \mid A \Rightarrow_G^* x\}$ and, for every $x \in \Sigma^*$, let $d_A(x)$ be the number of leftmost derivations $A \Rightarrow_G^* x$ (or, equivalently, the number of derivation trees of the word x rooted at A). Moreover, for every $\ell \in \mathbb{N}$, we denote by $T_A(\ell)$ the sum $T_A(\ell) = \sum_{|x|=\ell} d_A(x)$ and by $L_A(\ell)$ the subset $\{x \in L_A \mid |x| = \ell\}$.

It is well-known that every sequence $\{T_A(\ell)\}_{\ell \geq 1}$ has an algebraic generating function [5] and hence, by a result due to Comtet [6], there exists a set of polynomials $p_0(x), p_1(x), \ldots, p_m(x)$ with integer coefficients such that, for all ℓ large enough,

$$p_0(\ell)T_A(\ell) + p_1(\ell)T_A(\ell-1) + \cdots + p_m(\ell)T_A(\ell-m) = 0.$$

Since $T_A(n) = O(r^n)$ for some $r > 0$, the previous equation implies that the first n terms $T_A(1), \ldots, T_A(n)$ can be computed in $O(n^2)$ time on a RAM under logarithmic cost criterion. Moreover, for each $T_A(\ell)$ with $1 \leq \ell \leq n$, the integers $b_A(\ell) = \lceil \log T_A(\ell) \rceil$ can be computed in $O(\ell \log \ell)$ time (see Lem. A.2), for an overall time cost of $O(n^2 \log n)$.

Now, assume G is unambiguous. Then, for every $A \in V$ and every $x \in \Sigma^*$, $d_A(x) \leq 1$ while, for each $\ell \in \mathbb{N}$, $T_A(\ell)$ coincides with the cardinality of $L_A(\ell)$. Thus a u.r.g. for L_S can be designed which, on input $n \in \mathbb{N}$, first computes the coefficients $T_A(\ell)$, for all $1 \leq \ell \leq n$ and every $A \in V$; then, it calls Procedure Generate on input (S, n). Such a procedure, for an input $(A, \ell) \in V \times \mathbb{N}$, returns an element $w \in L_A(\ell) \cup \{\perp\}$ such that $\Pr\{w = \perp\} < 1/4$ and $\Pr\{w = x \mid w \neq \perp\} = T_A(\ell)^{-1}$ for every $x \in L_A(\ell)$. The computation is described by the following scheme, where κ is a global parameter to be fixed for increasing the probability of success of the algorithm, P_A denotes the subset of productions in P of the form $A \to BC$ with $B, C \in V$, and P_A^1 is the set of productions in P of the form $A \to a$ with $a \in \Sigma$.

The procedure chooses an element uniformly at random either in P_A^1 (if $\ell = 1$), or in the set $P_A \times \{1, \ldots, \ell - 1\}$ (if $\ell > 1$), the choice depending on a total order relation $\leq$ among the elements of these sets. To define $\leq$, we assume a lexicographic order $\preccurlyeq$ in both alphabets Σ and V and set $A \to a \leq A \to b$ in P_A^1 if $a \preccurlyeq b$, while $(A \to BC, h) \leq (A \to DE, k)$ in $P_A \times \{1, \ldots, \ell - 1\}$ if either $h < k$, or $h = k$ and $BC \preccurlyeq DE$. If $\ell > 1$, once a random element $(A \to BC, k) \in P_A \times \{1, \ldots, \ell - 1\}$ is computed, the same procedure is recursively called on input (B, k) and $(C, \ell - k)$, and then it returns the concatenation of the two words.

procedure Generate(A, ℓ)
$i \leftarrow 0$, $r \leftarrow \perp$, $w \leftarrow \perp$
while $i < \kappa$ and $r = \perp$ **do**
    $i \leftarrow i + 1$
    generate $u \in \{1, \ldots, 2^{\lceil \log T_A(\ell) \rceil}\}$ *uniformly at random*
    if $u \leq T_A(\ell)$ **then** $r \leftarrow u$
if $r \neq \perp$ **then**
    if $\ell = 1$ **then**
        let $A \to a$ be the r-th element of P_A^1
        $w \leftarrow a$
    else
        compute the smallest element $(A \to BC, k)$ in $P_A \times \{1, \ldots, \ell - 1\}$
        such that $\displaystyle\sum_{(A \to DE, h) \leq (A \to BC, k)} T_D(h) T_E(\ell - h) \geq r$ $\qquad (*)$
        $w_B \leftarrow$Generate(B, k)
        $w_C \leftarrow$Generate$(C, \ell - k)$
        if $w_B \neq \perp$ and $w_C \neq \perp$ **then** $w \leftarrow w_B w_C$
return w.

Let $A(n)$ be the output of Generate(A, n). We first give an upper bound to $\Pr\{A(n) = \perp\}$. Let $e(\ell)$ be the maximum of all values $\Pr\{A(\ell) = \perp\}$ for $A \in V$. One can easily show that $e(1) \leq (1/2)^\kappa$ and $e(\ell) \leq (1/2)^\kappa + \max_{1 \leq k \leq \ell - 1}\{e(k) + e(\ell - k)\}$ for every $1 \leq \ell \leq n$. A simple induction proves $e(\ell) \leq (2\ell - 1)/2^\kappa$ and hence fixing $\kappa = 3 + \lceil \log n \rceil$ we have

$$\Pr\{A(n) = \perp\} \leq e(n) < 1/4 \qquad \text{for every } A \in V.$$

Moreover, since the grammar is unambiguous, reasoning by induction on ℓ, it is easy to verify that, if $A(\ell) \neq \perp$ then $A(\ell)$ is uniformly distributed in $L_A(\ell)$. More precisely, for every $x \in L_A(\ell)$, we have

$$\Pr\{A(\ell) = x \mid A(\ell) \neq \perp\} = 1/C_{L_A}(\ell).$$

On the contrary, if the grammar is ambiguous (*i.e.* $d_A(x) > 1$ for some $A \in V$ and $x \in \Sigma^*$) then for every $x \in L_A(\ell)$, we have

$$\Pr\{A(\ell) = x \mid A(\ell) \neq \perp\} = \frac{d_A(x)}{T_A(\ell)}.$$

Concerning the time complexity, we assume to search the element $(A \to BC, k)$ of step $(*)$ by a boustrophedonic routine [10]. Also, let $N(\ell)$ be the maximum number of PrRAM instructions executed by Generate(A, ℓ) for $A \in V$. Then, for a suitable constant $c > 0$, one can write the following recursion, for every $1 \leq \ell \leq n$,

$$N(\ell) = \begin{cases} O(\kappa) & \text{if } \ell = 1 \\ O(\kappa) + \max_{1 \leq j < \ell}\{c \min\{j, \ell - j\} + N(j) + N(\ell - j)\} & \text{if } \ell > 1. \end{cases}$$

This proves that $N(n) = O(\kappa n) + cf(n)$, $f(n)$ being the solution of the minimax equation $f(n) = \max\{f(k) + f(n-k) + \min\{k, n-k\}\}$ usually arising in the evaluation of the cost of boustrophedonic search [13]. Since it is known that $f(n) = O(n \log n)$, assuming $\kappa = O(\log n)$, we get $N(n) = O(n \log n)$, which, under our model of computation, gives a total time cost $O(n^2 \log n)$ since all integers involved in the routine have $O(n)$ bits. Finally, the number $B(n)$ of random bits used by the procedure on input $n > 1$ satisfies an equation of the form $B(n) = O(n \log n) + \max_{1 \le j < n}\{B(j) + B(\ell - j)\}$ for a total amount of $O(n^2 \log n)$ random bits.

We observe that, following an idea described in [11], the computation of the coefficients $\{(T_A(\ell), b_A(\ell)) : A \in V, 1 \le \ell \le n\}$ and the actual process of generating a random word in L_S^n can be mixed together into a unique procedure which only requires space $O(n)$ under logarithmic cost criterion (leaving unchanged the order of growth of the time complexity).

This discussion is summarized by the following:

Proposition 1.1. *Every unambiguous context-free language admits a uniform random generator working in $O(n^2 \log n)$ time and using $O(n^2 \log n)$ random bits on a PrRAM under logarithmic cost criterion.*

2. Finitely ambiguous context-free languages

The previous algorithm can be used to design a simple u.r.g. for inherently ambiguous context-free languages which works in polynomial time whenever the ambiguity of the grammar is bounded by a polynomial. In this section we show that, in the case of finite ambiguity, such a u.r.g. requires $O(n^2 \log n)$ time.

Consider again a *c.f.* grammar $G = \langle V, \Sigma, S, P\rangle$ in Chomsky normal form without useless variables; assume G is finitely ambiguous, *i.e.* there exists $D \in \mathbb{N}$ such that $d_S(x) \le D$ for every $x \in \Sigma^*$. Intuitively, a u.r.g. for L_S can work as follows: first, a word x is computed by calling Procedure Generate(S, n); then, the value $d_S(x)$ is computed, a biased coin is thrown with probability $1/d_S(x)$ and in case of success the algorithm outputs x. This guarantees that all words of length n in the language are generated with equal probability. Moreover, the generating loop is repeated a suitable number of times to ensure that the probability of a failure is below the given upper bound.

Formally, the algorithm is described by the following scheme, where $\kappa > 0$ is a suitable integer constant discussed later and $\operatorname{lcm} I$ denotes the least common multiple of a set $I \subseteq \mathbb{N}$.

procedure Gen_amb(G)
input n
$m \leftarrow \operatorname{lcm}\{1, \ldots, D\}$, $\ell \leftarrow \lceil \log m \rceil$
$i \leftarrow 0$, $w \leftarrow \perp$
while $i < \kappa D$ and $w = \perp$ **do**
    $i \leftarrow i + 1$

$x \leftarrow$Generate(S, n)
if $x \neq \perp$ **then**
  $d \leftarrow d_S(x)$
  generate $r \in \{1, \ldots, 2^\ell\}$ *uniformly at random*
  if $r \leq m/d$ **then** $w \leftarrow x$
return w.

We point out that a word x is actually generated with probability $h/d_S(x)$, for some $1/2 \leq h \leq 1$ (since the random integer r may be greater than m). However, the value of h is fixed and independent of x, and therefore all words of length n have the same probability to be generated by the algorithm. The value of $d_S(x)$ is computed by a procedure we show later.

More formally, to prove the algorithm is a u.r.g. for L_S, assume $T_S(n) > 0$ and let W and X be the random variables representing, respectively, the value of w and x at the end of a **while** iteration. Since X is the output of Generate(S, n), there exists $0 < \delta < 1/4$ such that, for every $u \in L_S^n$,

$$\Pr\{X = u\} = (1 - \delta)\frac{d_S(u)}{T_S(n)} \cdot$$

Hence, we obtain

$$\Pr\{W = u\} = \Pr\{X = u\}\frac{m}{d_S(u)2^{\lceil \log m \rceil}} = \frac{(1-\delta)m}{2^{\lceil \log m \rceil} T_S(n)}$$

which is independent of u. On the other hand, at the end of each **while** iteration, $\Pr\{W = \perp\} = 1 - C_{L_S}(n)\left((1-\delta)m2^{-\lceil \log m \rceil}T_S(n)^{-1}\right)$. Then, since $T_S(n) \leq C_{L_S}(n)D$, we have $\Pr\{W = \perp\} \leq 1 - \frac{3}{8}\frac{1}{D}$ and hence the probability of returning $\perp$ is $\left(1 - \frac{3}{8}\frac{1}{D}\right)^{\kappa D}$ which is smaller than $1/4$ for a suitable choice of $\kappa > 0$.

To evaluate the time complexity of this procedure we first have to describe an algorithm for the computation of $d_S(x)$. This is presented in detail in the following subsection by using Earley's algorithm [9] for context-free recognition. The main advantage of this procedure, with respect to the well-known CYK algorithm [14], is that in the case of a grammar with bounded ambiguity, the computation only requires quadratic time on a RAM under unit cost criterion [2, 9].

2.1. Earley's algorithm for counting derivations

Consider a finitely ambiguous *c.f.* grammar $G = \langle V, \Sigma, S, P\rangle$ in Chomsky normal form, without useless variables. We want to describe an algorithm that, for an input $x \in \Sigma^*$, computes $d_S(x)$, *i.e.* the number of derivation trees of x rooted at S. The procedure manipulates a *weighted* version of the so-called *dotted productions* of G, which are defined as expressions of the form $A \rightarrow \alpha \cdot \beta$, where $A \in V$, $\alpha, \beta \in (\Sigma \cup V)^*$ and $A \rightarrow \alpha\beta \in P$ (in our case, since G is in Chomsky normal form, both α and β have length 2 at most).

Given an input string $x = a_1 a_2 \ldots a_n$, the algorithm computes a table of entries $S_{i,j}$, for $0 \leq i \leq j \leq n$, each of which is a list of terms of the form $[A \rightarrow \alpha \cdot \beta, t]$,

where $A \to \alpha \cdot \beta$ is a dotted production in G and t is a positive integer. Each pair $[A \to \alpha \cdot \beta, t]$ is called *state* and t is the *weight* of the state.

We will prove that the table of lists $S_{i,j}$ computed by the algorithm has the following properties for any pair of indices $0 \leq i \leq j \leq n$:

1. $S_{i,j}$ contains at most one state $[A \to \alpha \cdot \beta, t]$ for every dotted production $A \to \alpha \cdot \beta$ in G;
2. a state $[A \to \alpha \cdot \beta, t]$ belongs to $S_{i,j}$ if and only if there exists $\delta \in V^*$ such that $S \stackrel{*}{\Rightarrow} a_1 \dots a_i A \delta$ and $\alpha \stackrel{*}{\Rightarrow} a_{i+1} \dots a_j$;
3. if $[A \to \alpha \cdot \beta, t]$ belongs to $S_{i,j}$, then $t = \#\{\alpha \stackrel{*}{\Rightarrow} a_{i+1} \dots a_j\}$, *i.e.* the number of leftmost derivations $\alpha \stackrel{*}{\Rightarrow} a_{i+1} \dots a_j$.

Note that, since there are no ϵ-productions, $[A \to \alpha \cdot \beta, t] \in S_{i,i}$ implies $\alpha = \epsilon$ for every $0 \leq i \leq n$. Furthermore, once the lists $S_{i,j}$ are completed for any $0 \leq i \leq j \leq n$, then $d_S(x)$ can be obtained as the sum $\sum_{[S \to AB\cdot, t] \in S_{0,n}} t$.

The algorithm first computes the list $S_{0,0}$ of all the states $[A \to \cdot\, \alpha, 1]$ such that $S \stackrel{*}{\Rightarrow} A\delta$ for some $\delta \in V^*$. Then, it executes the cycle of *Scanner*, *Predictor* and *Completer* loops given below for $1 \leq j \leq n$, computing at the j-th loop the lists $S_{i,j}$ for $0 \leq i \leq j$. To this end the procedure maintains a family of sets $L_{B,i}$ for $B \in V$ and $1 \leq i \leq n$; each $L_{B,i}$ contains all indices $k \leq i$ such that a state of the form $[A \to \alpha \cdot B\beta, t]$ belongs to $S_{k,i}$ for some $A \in V$, $\alpha, \beta \in V \cup \{\epsilon\}$, $t \in \mathbb{N}$. Moreover, during the computation every state in $S_{i,j}$ is unmarked as long as it may be used to add new states in the table; when this cannot occur any more the procedure marks the state.

The command "ADD D TO $S_{i,j}$" simply appends the state D as unmarked to $S_{i,j}$ and adds i to the list $L_{B,j}$ whenever D is of the form $[A \to \alpha \cdot B\beta, t]$; the command "UPDATE $[A \to \alpha \cdot \beta, t]$ IN $S_{i,j}$" replaces the state $[A \to \alpha \cdot \beta, u]$ in $S_{i,j}$ for some $u \in \mathbb{N}$ by $[A \to \alpha \cdot \beta, t]$, at last, "MARK D IN $S_{i,j}$" transforms the state D in $S_{i,j}$ into a marked state.

```
1    for j = 1 ... n do

Scanner:
2    for [A → · a, t] ∈ S_{j-1,j-1} do
3        MARK [A → · a, t] IN S_{j-1,j-1}
4        if a = a_j then ADD [A → a·, t] TO S_{j-1,j}

Completer:
5    for i = j − 1 ... 0 do
6        for [B → γ·, t] ∈ S_{i,j} do
7            MARK [B → γ·, t] IN S_{i,j}
8            for k ∈ L_{B,i} do
9                for [A → α · Bβ, u] ∈ S_{k,i} do
10                   if [A → αB · β, v] ∈ S_{k,j}
11                   then UPDATE [A → αB · β, v + tu] IN S_{k,j}
12                   else ADD [A → αB · β, tu] TO S_{k,j}
```

Predictor:

13 **for** $i = 0 \dots j-1$ **do**
14 **for** $[A \to \alpha \cdot B\beta, t] \in S_{i,j}$ **do**
15 MARK $[A \to \alpha \cdot B\beta, t]$ IN $S_{i,j}$
16 **for** $B \to \gamma \in P$ **do**
17 **if** $[B \to \cdot\, \gamma, 1] \notin S_{j,j}$ **then** ADD $[B \to \cdot\, \gamma, 1]$ TO $S_{j,j}$
18 **while** $\exists$ UNMARKED $[A \to \cdot\, B\beta, t] \in S_{j,j}$ **do**
19 MARK $[A \to \cdot\, B\beta, t]$ IN $S_{j,j}$
20 **for** $B \to \gamma \in P$ **do**
21 **if** $[B \to \cdot\, \gamma, 1] \notin S_{j,j}$ **then** ADD $[B \to \cdot\, \gamma, 1]$ TO $S_{j,j}$

We are then able to prove the following:

Lemma 2.1. *The table of lists $S_{i,j}$, $0 \le i \le j \le n$, computed by the procedure described above satisfies the properties 1), 2) and 3).*

Proof. First, observe that statement 1) is easily verified: at line 4 distinct states are added to an initially empty list $S_{i,j}$; at lines 12, 17, 21 a state is added to a list provided no state with the same dotted production is already contained.

Statement 2) only refers to dotted productions appearing in the lists and does not concern the weight of the states. Moreover, disregarding the computations on the weight of the states, the procedure works on the dotted production exactly as Earley's algorithm; hence statement 2) is a consequence of its correctness (for a detailed proof see Th. 4.9 in [2]).

Now, let us prove statement 3). First note that all states in $S_{i,j}$, for $1 \le i \le j \le n$, are marked during the computation. Hence, we can reason by induction on the order of marking states. The initial condition is satisfied because all states in each $S_{j,j}$, $0 \le j \le n$, are of the form $[A \to \cdot\, \alpha, t]$ and have weight $t = 1$. Also the states of the form $[A \to a\cdot, t]$ have weight $t = 1$ and again statement 3) is satisfied.

Then, consider a state $D = [A \to \alpha B \cdot \beta, w] \in S_{k,j}$ $(k < j)$. We claim that w is the number of leftmost derivations $\alpha B \stackrel{*}{\Rightarrow} a_{k+1} \dots a_j$. A state of this form is first added by the *Completer* at line 12. Now, consider the set I_k of indices $k \le i < j$ such that $[A \to \alpha \cdot B\beta, u_i] \in S_{k,i}$ for some $u_i \in \mathbb{N}$ and, for each $i \in I_k$, let U_i be the family of states in $S_{i,j}$ of the form $[B \to \gamma\cdot, t]$. It is clear that

$$w = \sum_{i \in I_k} \sum_{[B \to \gamma\cdot, t] \in U_i} t u_i. \tag{1}$$

Moreover, each state $[A \to \alpha \cdot B\beta, u_i] \in S_{k,i}$ is marked at line 15 or 19 before D is added to $S_{k,j}$. Also all states in U_i, for all $i \in I_k$, are marked during the computation of the weight w and, by the form of the grammar, updating w cannot modify the weight of any state in U_i. As a consequence, all the states in U_i are marked before D. Hence, by inductive hypothesis, for every $i \in I_k$, we have

$$u_i = \#\left\{\alpha \stackrel{*}{\Rightarrow} a_{k+1} \dots a_i\right\} \tag{2}$$

and, for each $[B \to \gamma\cdot, t] \in U_i$,

$$t = \# \left\{ \gamma \stackrel{*}{\Rightarrow} a_{i+1} \dots a_j \right\} . \tag{3}$$

Now the number of leftmost derivations $\alpha B \stackrel{*}{\Rightarrow} a_{k+1} \dots a_j$ is clearly given by

$$\sum_{k \le i < j} \# \left\{ \alpha \stackrel{*}{\Rightarrow} a_{k+1} \dots a_i \right\} \sum_{B \to \gamma \in P} \# \left\{ \gamma \stackrel{*}{\Rightarrow} a_{i+1} \dots a_j \right\}$$

and the claim follows from equation (1) by applying statement 1) and equalities (2) and (3). □

Theorem 2.2. *Given a finitely ambiguous context-free grammar G in Chomsky normal form, the algorithm described above computes the number of derivation trees of a string of length n in $O(n^2 \log n)$ time and $O(n^2)$ space (under logarithmic cost criterion).*

Proof. We first observe that every list $S_{i,j}$ for $0 \le i \le j \le n$ contains at most a constant number of states, each of which can be stored by using $O(1)$ binary space since G is finitely ambiguous. This implies a space complexity $O(n^2)$.

As far as the time complexity is concerned, note that in each loop, for a fixed $1 \le j \le n$, the *Scanner* and *Predictor* phase execute $O(j)$ commands, while the *Completer* phase requires a number of unit steps proportional to

$$\sum_{B \in V, 0 \le i < j} \sharp L_{B,i}.$$

Since G is finitely ambiguous, each $k \in \{0, 1, \dots, j-1\}$ can appear only in a constant number of lists $L_{B,i}$, for $B \in V$ and $0 \le i < j$, and hence the sum above is bounded by $O(j)$. As a consequence, we only need $O(\log n)$ logarithmic time to locate each state in the table yielding a total time complexity $O(n^2 \log n)$. □

Applying this result to the u.r.g. defined at the beginning of this section, we obtain the following

Corollary 2.3. *If L is a finitely ambiguous context-free language, then there exists a u.r.g. for L working in $O(n^2 \log n)$ time and $O(n^2)$ space on a PrRAM under logarithmic cost criterion.*

3. One-way auxiliary pushdown automata

In this section we consider languages accepted by a *one-way nondeterministic auxiliary pushdown automaton* (1-NAuxPDA, for short). We show that these languages admit a polynomial time u.r.g. whenever they are accepted by a 1-NAuxPDA that works in polynomial time and has a polynomially bounded ambiguity.

We recall that a 1-NAuxPDA is a nondeterministic Turing machine having a one-way read-only input tape, a pushdown tape and a *log-space bounded* two-way read-write work tape [4, 7]. It is known that the class of languages accepted by a polynomial time 1-NAuxPDA corresponds exactly to the class of decision problems that are reducible to context-free recognition *via* one-way log-space reductions [19].

Given a 1-NAuxPDA M, we define by $\hat{d}_M(x)$ the number of accepting computations of M on input $x \in \Sigma^*$ and call *ambiguity* of M the function $d_M : \mathbb{N} \to \mathbb{N}$ defined by $d_M(n) = \max_{x \in \Sigma^n} \hat{d}_M(x)$, for every $n \in \mathbb{N}$. Then, M is said *polynomially ambiguous* if, for some polynomial $p(n)$, we have $d_M(n) \leq p(n)$ for every $n > 0$. Moreover, it is known that, given an integer input $n > 0$, a *c.f.* grammar G_n can be built such that $L(G_n) \cap \Sigma^n = L(M) \cap \Sigma^n$, where $L(G_n) \subseteq \Sigma^*$ is the language generated by G_n and $L(M) \subseteq \Sigma^*$ is the language accepted by M. This allows us to apply the results of the previous section to the languages accepted by a 1-NAuxPDA.

Here, we describe a modified version of the usual construction of G_n [7] that allows to bound the ambiguity of G_n with respect to the ambiguity of M. First of all, we assume w.l.o.g. that the automaton cannot simultaneously consume input and modify the content of the stack, at most one symbol can be pushed or popped in a single move, there is only one final state and, finally, the input is accepted if and only if the automaton reaches the final state with both the pushdown store and the work tape empty.

A *surface configuration* of a 1-NAuxPDA M on input of length n is a 5-tuple (q, w, i, Γ, j) where q is the state of M, w the content of its work tape, $1 \leq i \leq |w|$ the work tape head position, Γ the symbol on top of the stack and $1 \leq j \leq n+1$ the input tape head position. Observe that there are $n^{O(1)}$ surface configurations on any input of length $n \geq 0$. Two surface configurations C_1, C_2 form a *realizable pair* (C_1, C_2) *(on a word $y \in \Sigma^+$)* if M can move (consuming input y) from C_1 to C_2, ending with its stack at the same height as in C_1, without popping below this height at any step of the computation. If (C_1, D) and (D, C_2) are realizable pairs on y' and y'' respectively, then (C_1, C_2) is a realizable pair on $y = y'y''$. Let S_n be the set of surface configurations of M on inputs of length n and define the *c.f.* grammar $G_n(M) = \langle N, \Sigma, S, P \rangle$, where $N = \{S\} \cup \{(C_1, C_2, \ell) : C_1, C_2 \in \mathrm{S}_n \text{ and } \ell \in \{0, 1\}\}$ and the set P of productions is given by the following rules:

1. P contains both $S \to (C_{\mathsf{in}}, C_{\mathsf{fin}}, 0)$ and $S \to (C_{\mathsf{in}}, C_{\mathsf{fin}}, 1)$, where C_{in} and C_{fin} represent respectively the initial and final surface configuration of M;
2. $(C_1, C_2, 0) \to \sigma \in P$ iff (C_1, C_2) is a realizable pair on $\sigma \in \Sigma \cup \{\varepsilon\}$ *via* a single move computation;
3. $(C_1, C_2, 0) \to (C_1, D, 1)(D, C_2, \ell) \in P$, for $\ell \in \{0, 1\}$, iff $C_1, C_2, D \in \mathrm{S}_n$;
4. $(C_1, C_2, 1) \to (D_1, D_2, \ell) \in P$, for $\ell \in \{0, 1\}$, iff $C_1, D_1, D_2, C_2 \in \mathrm{S}_n$, D_1 can be reached from C_1 in a single move pushing a symbol a on top of the stack and C_2 can be reached from D_2 in a single move popping the same symbol from the top of the stack.

For the sake of brevity define, for each realizable pair (C_1, C_2), the set $\mathrm{C}(C_1, C_2)$ of all computations of M starting from C_1 and ending in C_2 with the stack at

the same height as in C_1, without popping below this height at any point of the computation; define also the subset $\mathrm{C}_1(C_1, C_2) \subseteq \mathrm{C}(C_1, C_2)$ of such computations (of at least two steps) during which the stack height never equals the stack height at the extremes C_1, C_2 and the subset $\mathrm{C}_0(C_1, C_2) = \mathrm{C}(C_1, C_2) \setminus \mathrm{C}_1(C_1, C_2)$ of one step computations and longer computations during which the stack height equals, at least once, the stack height at the extremes C_1, C_2. Following [7] and [3], it is possible to prove that:

Proposition 3.1. *Given a polynomial time* 1*-NAuxPDA* M*, there exists an algorithm computing, for an input* $n > 0$*, the grammar* $G_n(M)$ *in polynomial time on a RAM under logarithmic cost criterion. Moreover, for every* $\ell \in \{0, 1\}$*,* $C_1, C_2 \in S_n$ *and* $y \in \Sigma^*$*,* (C_1, C_2) *is a realizable pair on* y *via a computation in* $C_\ell(C_1, C_2)$ *if and only if* $(C_1, C_2, \ell) \stackrel{*}{\Rightarrow} y$*.*

In order to apply the result on *c.f.* grammars in this case, we have to bound the ambiguity of $G_n(M)$.

Proposition 3.2. *For every polynomial time* 1*-NAuxPDA* M*, the number of leftmost derivations* $(C_1, C_2, \ell) \stackrel{*}{\Rightarrow} y$ *in* $G_n(M)$ *is less than or equal to the number of computations in* $C_\ell(C_1, C_2)$ *consuming* y*.*

Proof. We want to prove that the correspondence given in Proposition 3.1 defines, for every $C_1, C_2 \in \mathrm{S}_n$ and $y \in \Sigma^*$, a bijective map from the leftmost derivations $(C_1, C_2, \ell) \stackrel{*}{\Rightarrow} y$ to the computations in $\mathrm{C}_\ell(C_1, C_2)$ consuming y.

By the previous proposition, we simply have to show that such a map is injective, *i.e.* if the computations associated with two leftmost derivations are equal, then the two derivations themselves are the same. We reason by induction on the number k of steps of the computation.

If $k = 1$ then the statement is obvious.

Assume $k > 1$ and $\ell = 0$; for $i \in \{1, 2\}$, let $(C_1, C_2, 0) \to (C_1, D_i, 1)(D_i, C_2, \ell_i) \stackrel{*}{\Rightarrow} y$ be two leftmost derivations (for some $\ell_i \in \{0, 1\}$ and $D_i \in \mathrm{S}_n$). By the construction of the grammar, if the associated computations are equal, then $D_1 = D_2$, $\ell_1 = \ell_2$ and there exist two words y', y'' such that $y = y'y''$, $(C_1, D_i, 1) \stackrel{*}{\Rightarrow} y'$ and $(D_i, C_2, \ell_i) \stackrel{*}{\Rightarrow} y''$, for $i \in \{1, 2\}$. Then, by inductive hypothesis, $(C_1, D_i, 1) \stackrel{*}{\Rightarrow} y'$ are the same leftmost derivation for $i \in \{1, 2\}$ and $(D_i, C_2, \ell_i) \stackrel{*}{\Rightarrow} y''$ are the same leftmost derivation for $i \in \{1, 2\}$. Hence $(C_1, C_2, 0) \to (C_1, D_i, 1)(D_i, C_2, \ell_i) \stackrel{*}{\Rightarrow} y$ are the same leftmost derivation for $i \in \{1, 2\}$.

Now assume $k > 1$ and $\ell = 1$ and for $i \in \{1, 2\}$, let $(C_1, C_2, 1) \to (D_{1,i}, D_{2,i}, \ell_i) \stackrel{*}{\Rightarrow} y$ be two leftmost derivations (for some $\ell_i \in \{0, 1\}$ and $D_{1,i}, D_{2,i} \in \mathrm{S}_n$). If the associated computations are equal, then we have $D_{1,1} = D_{1,2}$, $D_{2,1} = D_{2,2}$, $\ell_1 = \ell_2$, and hence $(D_{1,i}, D_{2,i}, \ell_i) \stackrel{*}{\Rightarrow} y$ for any $i \in \{1, 2\}$. Moreover, by inductive hypothesis, the two derivations for $i \in \{1, 2\}$ coincide, hence $(C_1, C_2, 1) \to (D_{1,i}, D_{2,i}, \ell_i) \stackrel{*}{\Rightarrow} y$ are the same leftmost derivation for $i \in \{1, 2\}$. □

Now, we are able to state the main property of this section.

Theorem 3.3. *Every language L accepted by a polynomial time 1-NAuxPDA with polynomially bounded ambiguity admits a u.r.g. working in polynomial time.*

Proof. Let M be a polynomially ambiguous 1-NAuxPDA accepting L and working in polynomial time. Moreover, assume $p(n)$ is a polynomial such that $d_M(n) \leq p(n)$, for every n. Then a u.r.g. for L can be designed that, on input $n \in \mathbb{N}$, first computes the grammar $G_n = G_n(M)$ and then it applies procedure Gen_amb(G_n) where, however, the constant D is replaced by the value $p(n)$. Recall that, by Proposition 3.2, the ambiguity of G_n is not greater than $p(n)$. Also observe the number of leftmost derivations of a terminal string x in G_n can be computed in time polynomial in $|x|$ and $|G_n|$. Therefore, reasoning as in Section 2 and applying Lemma A.1 given in the Appendix, it is easy to show that the overall algorithm works in polynomial time. □

Appendix A. Technical lemmas

Here, we show some complexity bounds we have used in our proofs.

Lemma A.1. *The least common multiple (*lcm*) of $\{1, \ldots, n\}$ is an integer of $O(n)$ bits and can be computed in time $O(n^2)$ by a RAM under logarithmic cost criterion.*

Proof. First, we recall that $\mathrm{lcm}\{1, \ldots, n\} \leq n^{\pi(n)}$, where $\pi(n)$ is the number of primes less than n ([22], Lem. 4.1.2). As a consequence, $\mathrm{lcm}\{1, \ldots, n\}$ has $O(n)$ bits since it is well-known that $\pi(n) \sim n/\log n$. Now, to compute $\mathrm{lcm}\{1, \ldots, n\}$, a naïve iterative procedure LCM(n) can be designed that first calculates recursively the value $a = \mathrm{LCM}(n-1)$ and then determines the least common multiple of a and n by using Euclid's algorithm for computing the GCD. Since computing such a GCD requires $O(n)$ time, the overall computation can be executed in $O(n^2)$ time. □

Lemma A.2. *The value $\lceil \log N \rceil$ can be computed on input $N \geq 1$ in $O(\log N \log\log N)$ time by a RAM under logarithmic cost criterion.*

Proof. To compute $\lceil \log N \rceil$ in our model of computation, a recursive procedure can be designed that, on input $N \geq 1$, returns 0 if $N = 1$; otherwise, it first computes the largest power $k = 2^h$ ($h \in \mathbb{N}$) such that $2^k \leq N$ in $O(h2^h)$ time, then it recursively calculates $a = \lceil \log \lfloor N/2^k \rfloor \rceil$ and returns $a+k$. Such a procedure, for an input N of r bits, works in time $T(r) = O(r \log r) + T(\lfloor r/2 \rfloor)$ and hence $T(r) = O(\sum_{h=1}^{\log r} h2^h) = O(\log N \log\log N)$. □

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA (1974).

[2] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling - Vol. I: Parsing*. Prentice Hall, Englewood Cliffs, NJ (1972).

[3] E. Allender, D. Bruschi and G. Pighizzini, The complexity of computing maximal word functions. *Comput. Complexity* **3** (1993) 368-391.
[4] F.-J. Brandenburg, On one-way auxiliary pushdown automata, edited by H. Waldschmidt, H. Tzschach and H.K.-G. Walter, in *Proc. of the 3rd GI Conference on Theoretical Computer Science*. Springer, Darmstadt, FRG, *Lecture Notes in Comput. Sci.* **48** (1977) 132-144.
[5] N. Chomsky and M.-P. Schützenberger, *The algebraic theory of context-free languages*, edited by P. Braffort and D. Hirschberg. North-Holland, Amsterdam, The Netherlands, *Computer Programming and Formal Systems* (1963) 118-161.
[6] L. Comtet, Calcul pratique des coefficients de Taylor d'une fonction algébrique. *Enseign. Math.* **10** (1964) 267-270.
[7] S.A. Cook, Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM* **18** (1971) 4-18.
[8] A. Denise and P. Zimmermann, Uniform random generation of decomposable structures using floating-point arithmetic. *Theoret. Comput. Sci.* **218** (1999) 233-248.
[9] J. Earley, An efficient context-free parsing algorithm. *Commun. ACM* **13** (1970) 94-102.
[10] P. Flajolet, P. Zimmerman and B. Van Cutsem, A calculus for the random generation of labelled combinatorial structures. *Theoret. Comput. Sci.* **132** (1994) 1-35.
[11] M. Goldwurm, Random generation of words in an algebraic language in linear binary space. *Inform. Process. Lett.* **54** (1995) 229-233.
[12] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk and S. Mahaney, A quasi-polynomial-time algorithm for sampling words from a context-free language. *Inform. and Comput.* **134** (1997) 59-74.
[13] D.H. Greene and D.E. Knuth, *Mathematics for the analysis of algorithms*, Vol. 1. Birkhäuser, Basel, CH, *Progress in Comput. Sci.* (1981).
[14] M.A. Harrison, *Introduction to Formal Language Theory*. Addison-Wesley, Reading, MA (1978).
[15] T. Hickey and J. Cohen, Uniform random generation of strings in a context-free language. *SIAM J. Comput.* **12** (1963) 645-655.
[16] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Language, and Computation*. Addison-Wesley, Reading, MA (1979).
[17] M.R. Jerrum, L.G. Valiant and V.V. Vazirani, Random generation of combinatorial structures from a uniform distribution. *Theoret. Comput. Sci.* **43** (1986) 169-188.
[18] D.E. Knuth and A.C. Yao, The complexity of nonuniform random number generation, edited by J.F. Traub. Academic Press, *Algorithms and Complexity: New Directions and Recent Results* (1976) 357-428.
[19] C. Lautemann, On pushdown and small tape, edited by K. Wagener, *Dirk-Siefkes, zum 50. Geburststag (proceedings of a meeting honoring Dirk Siefkes on his fiftieth birthday)*. Technische Universität Berlin and Universität Ausgburg (1988) 42-47.
[20] H.G. Mairson, Generating words in a context-free language uniformly at random. *Inform. Process. Lett.* **49** (1994) 95-99.
[21] M. Santini, *Random Uniform Generation and Approximate Counting of Combinatorial Structures*, Ph.D. Thesis. Dipartimento di Scienze dell'Informazione (1999).
[22] A. Szepietowski, *Turing Machines with Sublogarithmic Space*. Springer Verlag, *Lecture Notes in Comput. Sci.* **843** (1994).

Received April 24, 2001. Revised February 7, 2002.