ELSEVIER

Theoretical Computer Science 218 (1999) 233–248

Theoretical
Computer Science

Uniform random generation of decomposable structures using floating-point arithmetic[1]

Alain Denise[a,*], Paul Zimmermann[b]

[a] *Université de Paris-Sud XI, Centre dé Orsay, Lab. de Recherche en Informatique, CNRS U.R.A. 410, Bât. 490, F-91405, Orsay Cedex, France*

[b] *INRIA-Lorraine, Technopole de Nancy-Brabois, Campus Scientifique, 615, r. Jardin Botanique, B.P. 101, F-54600 Villers-les-Nancy Cedex, France*

Abstract

The *recursive method* formalized by Nijenhuis and Wilf (1998) and systematized by Flajolet, Van Cutsem and Zimmermann (1994), is extended here to floating-point arithmetic. The resulting ADZ method enables one to generate decomposable data structures – both labelled or unlabelled – uniformly at random, in expected $O(n^{1+\varepsilon})$ time and space, after a preprocessing phase of $O(n^{2+\varepsilon})$ time, which reduces to $O(n^{1+\varepsilon})$ for context-free grammars. © 1999 Elsevier Science B.V. All rights reserved.

Résumé

La *méthode récursive* mise au point par Nijenhuis et Wilf (1998) et systématisée par Flajolet, Van Cutsem et Zimmermann (1994), est ici étendue à l'utilisation de nombres flottants. La méthode qui en découle, appelée ADZ, permet de générer aléatoirement et uniformément des structures décomposables – étiquetées ou non – en temps et espace moyens $O(n^{1+\varepsilon})$, après un précalcul de complexité en temps $O(n^{2+\varepsilon})$, se réduisant à $O(n^{1+\varepsilon})$ pour des grammaires algébriques. © 1999 Elsevier Science B.V. All rights reserved.

1. Introduction

In 1978, Nijenhuis and Wilf presented efficient algorithms to generate various data-structures like sets, multisets, and trees [16]. This method was systematized by Flajolet et al. in [9] to *decomposable* data-structures, and is now known as the *recursive method.* It is implemented in the MAPLE computer algebra system [4], in the COMBSTRUCT package, previously known as Gaïa [18].

[*] Corresponding author. E-mail: alain.denise@lri.fr.

[1] This work was partially supported by the GDR/PRC AMI, the Eurca project of INRIA Lorraine and the Centre Charles Hermite.

PII: S0304-3975(98)00323-5

The recursive method as presented in [9] has two major drawbacks: firstly the preprocessing phase requires $O(n^2)$ arithmetic operations, and secondly the coefficient growth makes the bit complexity for one generation much higher than the $O(n \log n)$ arithmetic complexity. A workaround to the first problem is well-known for context-free grammars: the coefficients satisfy P-recurrences which enable one to compute them in $O(n)$ arithmetic complexity. But the second problem remains: with naive multiprecision multiplication, each generation costs[2] $O(n^{3+\varepsilon})$ with the boustrophedonic method as already mentioned in [9]. Hence this method is limited to structures of size about one thousand, and does not allow to generate data structures of size one million.

Trying to use floating-point numbers instead of arbitrary precision integers is a natural idea: at each point of the algorithm where a choice has to be made, only $O(n)$ different branches are possible. Therefore, it is enough to know $O(\log n)$ bits of the corresponding probabilities to be able to decide in most cases. This idea was already expressed in [15] by Mairson, and also in [9]: "*The computation times could be further decreased (at the expense of a minuscule loss of uniformity) by using floating point arithmetics...*" This method would give only a quasi-uniform generator, but it is possible to get a really uniform generator using *certified* floating-point arithmetics, for example interval arithmetics following the IEEE 754 standard [12]. With that idea, Alain Denise got in [6] an efficient uniform generator using floating-point approximations, for some classes of rational languages. In the following paper, we show this holds for all classes of decomposable structures.

Our contribution is to present a new algorithm for the uniform random generation of decomposable structures using floating-point numbers, to analyse precisely the precision of floating-point computations and the average bit complexity of our algorithm. This algorithm is close to optimal for that class, as it exhibits a quasi-linear complexity both in expected time and space. Previously known algorithms were either limited to small classes of structures: balanced parenthesis strings in [3], regular languages in [13], some kinds of trees in [2]; or they did not have a quasi-linear time or space complexity: the algorithms proposed by Hickey and Cohen [11] (resp. Mairson [15]) for context-free languages with r nonterminals either have $O(n^{r+1})$ (resp. $O(n^2)$) space complexity, or $O(n^2 \log^2 n)$ (resp. $O(n^2)$) time complexity. Goldwurm's algorithm [10] works in linear space, but does not improve the time complexity of the recursive method. The proposed algorithm requires both an arbitrary precision floating-point arithmetic with directed roudings, and an arbitrary precision integer arithmetic.

The paper is organized as follows. Section 2 recalls briefly the standard algorithm and its complexity. Section 3 recalls some basic statements about floating-point arithmetic and rounding modes, and analyses the error propagation during the preprocessing phase. Then Section 4 states and analyses two random generation algorithms using floating-point arithmetics, a quasi-uniform one and a really uniform one. These results are confirmed by the experimental data from Section 5. Finally, Section 6 concludes and states some open questions.

[2] We write $O(n^{3+\varepsilon})$ for $O(n^{3+o(1)})$, which is also sometimes written $O^{\sim}(n^3)$ ("soft-O" notation).

Table 1
Bit complexity of the standard algorithm with large integer arithmetic

	One operation	Preprocessing		One generation
		General	Context-free	
	$O(M(n\log n))$	$O(n^2M(n\log n))$	$O(n^{2+\varepsilon})$	$O(n\log nM(n\log n))$
Naive	$O(n^{2+\varepsilon})$	$O(n^{4+\varepsilon})$	$O(n^{2+\varepsilon})$	$O(n^{3+\varepsilon})$
Schönhage	$O(n^{1+\varepsilon})$	$O(n^{3+\varepsilon})$	$O(n^{2+\varepsilon})$	$O(n^{2+\varepsilon})$

2. Standard algorithm

The standard algorithm – also mentioned hereafter as recursive algorithm – described in [9] takes as input a *combinatorial specification.* A combinatorial specification of a given class T_0 of combinatorial structures is an $(m+1)$-uple $(T_0, T_1, \ldots, T_m)$ of classes which are interrelated by means of productions made from basic objects (**1** and **Z** of size 0 and 1 respectively) and from *constructions* ($+$ for disjoint union, $\cdot$ for products, `sequence` for sequences, `set` for multisets and `cycle` for directed cycles).

The algorithm works as follows: first translate the specification into a *standard* one, where all products are binary, and the `sequence`, `set`, `cycle` constructions have been replaced with the marking and unmarking constructions Θ and Θ^{-1} (see [9]). Then the standard specification translates directly into procedures for counting the number of objects of a given size generated from a given non-terminal, or for generating one such object uniformly at random. The computation of all tables up to size n requires $O(n^2)$ operations on coefficients, then one random generation needs $O(n \log n)$ operations in the worst case using the boustrophedonic method.

2.1. Bit complexity

The integer coefficients used in the algorithm usually have an exponential growth with respect to the size n, so that an arbitrary precision arithmetic has to be used. More precisely, it is shown in [9] that the coefficients have size $O(n \log n)$.[3] Hence, with usual quadratic algorithms for integer arithmetic, each operation costs $O(n^2 \log^2 n)$, whence the preprocessing has bit-complexity $O(n^4 \log^2 n)$ and one generation has complexity $O(n^3 \log^3 n)$, as summarized in Table 1, where $O(M(n))$ stands for the cost of multiplying two n bit numbers. In the context-free case, where the `set` and `cycle` constructions are not used, the counting sequences satisfy linear recurrences with polynomials coefficients (P-recurrences or holonomic sequences). Therefore the coefficients can be computed in $O(n)$ operations between numbers of $O(n)$ and $O(\log n)$ bits, i.e. with a bit complexity of $O(n^2 \log n)$.

Another paper extends this to unlabelled objects [8]. From now on, we suppose we are given an unlabelled standard specification, with union, product, marking and unmarking constructions. The labelled case is very similar, with additional binomial coefficients.

[3] In the unlabelled case, they even have size $O(n)$: since the generating functions have a nonzero radius of convergence ρ as noticed in [9], the coefficients satisfy $\log y_n = n \log(1/\rho)(1 + o(1))$.

3. Floating point arithmetic

3.1. Basic definitions

First, we recall some properties of floating-point arithmetic, as stated for example in [5, 14]. In computers, a floating-point number is generally represented by three values: a *sign* $s_x \in \{-1, 1\}$, a *mantissa* m_x, and an *exponent* e_x, so that

$$x = s_x m_x 2^{e_x}.$$

Due to the limited size of the mantissa, arithmetic operations on floating-point numbers do not give exact results in general. Let us denote, as in [14], the basic operations $+$, $-$, $\times$, $/$ by $\oplus$, $\ominus$, $\otimes$, $\oslash$ respectively, when applied to floating-point numbers. The IEEE 754 standard [12] fixes their precise behaviour as follows. For any arbitrary floating-point numbers a and b,

$$\begin{aligned} a \oplus b &= \diamond(a + b), \\ a \ominus b &= \diamond(a - b), \\ a \otimes b &= \diamond(a \times b), \\ a \oslash b &= \diamond(a/b), \end{aligned}$$

where the function $\diamond$ is the active "rounding mode", which can be chosen by the user among the following ones: rounding towards the nearest number ($\circ$), towards 0 ($\mathscr{Z}$), towards $-\infty$ (∇), or towards $+\infty$ (Δ). This means that any basic operation on floating-point numbers is performed as if it was done with an infinite precision, and then the result rounded in order to agree with the floating-point representation.

In this paper, we will only deal with two rounding modes: towards $-\infty$ and towards $+\infty$. (In fact, since we will be handling only positive numbers, the towards $-\infty$ mode will be equivalent to the towards 0 mode.) These modes satisfy, for any real number x,[4]

$$x(1 - \varepsilon) \leqslant \nabla(x) \leqslant x$$

and

$$x \leqslant \Delta(x) \leqslant x(1 + \varepsilon).$$

The value ε is called the *computer precision* and is equal to 2^{1-b}, where b is the length of the mantissa in the floating-point representation.[5]

[4] Supposing the computer representation of x is not *denormalized* – see [12] – which holds for all numbers considered here, since they are integers.

[5] In base two, the first bit of the mantissa being always one for a non-zero normalized number, it is usually not represented. For instance, the C `double` numbers have $b = 53$, but only 52 bits are effectively stored, and $\varepsilon = 2^{-52}$.

For the sake of convenience, we will write $a \underline{\oplus} b$ (resp. $a \underline{\ominus} b$, $a \underline{\otimes} b$, $a \underline{\oslash} b$) for $\nabla(a+b)$ (resp. $\nabla(a-b)$, $\nabla(a \times b)$, $\nabla(a/b)$); and $a \overline{\oplus} b$ (resp. $a \overline{\ominus} b$, $a \overline{\otimes} b$, $a \overline{\oslash} b$) for $\Delta(a+b)$ (resp. $\Delta(a-b)$, $\Delta(a \times b)$, $\Delta(a/b)$).

The following easy lemma will be useful in the rest of the paper:

Lemma 3.1. *Let a and b be two nonnegative numbers and $\tilde{a}$ and $\tilde{b}$ two nonnegative approximations of a and b such that $a(1-\delta_a) \leqslant \tilde{a} \leqslant a$ and $b(1-\delta_b) \leqslant \tilde{b} \leqslant b$, with $\delta_a, \delta_b \geqslant 0$. Then*

$$(a+b)(1-\max(\delta_a,\delta_b)-\varepsilon) \leqslant \tilde{a} \underline{\oplus} \tilde{b} \leqslant a+b,$$
$$(a \times b)(1-\delta_a-\delta_b-\varepsilon) \leqslant \tilde{a} \underline{\otimes} \tilde{b} \leqslant a \times b.$$

If n is any positive number exactly representable, i.e. $n \leqslant 2^b$, then

$$(a \times n)(1-\delta_a-\varepsilon) \leqslant \tilde{a} \underline{\otimes} n \leqslant a \times n,$$
$$(a/n)(1-\delta_a-2\varepsilon) \leqslant \tilde{a} \underline{\oslash} n \leqslant a/n.$$

3.2. Error propagation

The aim of this subsection is to estimate the error we get when computing the coefficients during the preprocessing stage. We suppose in this subsection that the exponents of the floating point arithmetic can be arbitrary large; therefore no overflow is possible. As the coefficients have size $O(n \log n)$, the corresponding exponents have size $O(\log n)$. The main result is the following:

Proposition 3.2. *Let $(T_0, T_1, \ldots, T_m)$ be the combinatorial structure classes from a standard specification, and denote by $t_{k,l}$ the number of structures of T_k of size l. Suppose that the T_k are ordered in such a way that, in the counting algorithm, the computation of a given $t_{k,l}$ depends only on the $t_{k',l}$ with $0 \leqslant k' < k$ and on the $t_{k',l'}$ with $0 \leqslant k' \leqslant m$ and $0 \leqslant l' < l$. If we use floating-point arithmetics with precision ε and the rounding towards $-\infty$ mode to compute the $t_{k,l}$ for $0 \leqslant k \leqslant m$ and $0 \leqslant l \leqslant n$ according to the counting templates in* [9], *then we get an approximation $\tilde{t}_{k,l}$ of $t_{k,l}$, such that*

$$t_{k,l}(1-\varepsilon_{k,l}) \leqslant \tilde{t}_{k,l} \leqslant t_{k,l},$$

with

$$\varepsilon_{k,l} = 2ml^2\varepsilon - 2(m-k)l\varepsilon,$$

assuming in addition that all the coefficients of size zero $t_{k,0}$ can be represented exactly, i.e. are not larger than $2^b = 2/\varepsilon$.

Proof. We prove it by induction on k and l. The formula is true for $l=0$ since we supposed that all $t_{k,0}$ can be represented exactly, i.e. $\tilde{t}_{k,0}=t_{k,0}$ and $\varepsilon_{k,0}=0$.

Now suppose that the formula is true for any pair (k',l') such that either $0\leqslant k'<k$ and $l'=l$ or $l'<l$, and let us prove that it is true too for (k,l) with $l\geqslant 1$.

- If $T_k=\mathbf{1}$ or $T_k=\mathbf{Z}$, the formula is obvious since $\varepsilon_{k,l}=0$ for all l.
- If $T_k=T_{k_1}+T_{k_2}$ then $\tilde{t}_{k,l}=\tilde{t}_{k_1,l}\underline{\oplus}\tilde{t}_{k_2,l}$ and by Lemma 3.1, $\varepsilon_{k,l}\leqslant\max(\varepsilon_{k_1,l},\varepsilon_{k_2,l})+\varepsilon$. As necessarily $k_1,k_2\leqslant k-1$, using the induction hypothesis, we get $\varepsilon_{k,l}\leqslant 2ml^2\varepsilon+2(k-1-m)l\varepsilon+\varepsilon\leqslant 2ml^2\varepsilon+2(k-m)l\varepsilon$ since $l\geqslant 1$.
- If $T_k=T_{k_1}T_{k_2}$ then $\tilde{t}_{k,l}=(\tilde{t}_{k_1,a}\underline{\otimes}\tilde{t}_{k_2,l-a})\underline{\oplus}(\tilde{t}_{k_1,a+1}\underline{\otimes}\tilde{t}_{k_2,l-a-1})\underline{\oplus}\cdots\underline{\oplus}(\tilde{t}_{k_1,b}\underline{\otimes}\tilde{t}_{k_2,l-b})$, where a is 0 or 1 and b is $l-1$ or l according to the relative position of k_1,k_2 with respect to k. By Lemma 3.1 we deduce that $\varepsilon_{k,l}\leqslant\max_{a\leqslant i\leqslant b}(\varepsilon_{k_1,i}+\varepsilon_{k_2,l-i})+(b-a+1)\varepsilon$. Using the induction hypothesis gives $\varepsilon_{k,l}\leqslant 2\max_{a\leqslant i\leqslant b}f(i)\varepsilon+(b-a+1)\varepsilon$ with $f(i)=mi^2+(k_1-m)i+m(l-i)^2+(k_2-m)(l-i)$. The function $f(i)$ being convex, the maximum is reached either in $i=a$ or in $i=b$, and we have three cases to study:
  - $k_1,k_2<k$, i.e. $a=0$ and $b=l$. Then $f(a)$ and $f(b)$ are bounded by $ml^2+(k-1-m)l$, and $\varepsilon_{k,l}\leqslant 2ml^2\varepsilon+2(k-1-m)l\varepsilon+(l+1)\varepsilon\leqslant 2ml^2\varepsilon+2(k-m)l\varepsilon$ since again $l\geqslant 1$.
  - $k_1<k$ and $k_2\geqslant k$ (the case $k_1\geqslant k$ and $k_2<k$ is similar), i.e. $a=1$ and $b=l$ ($i=0$ is not possible since $t_{k,l}$ would depend from $t_{k_2,l}$). Then $f(a)=m(l^2-2l+2)+(k_1-m)+(k_2-m)(l-1)$ and $f(b)=ml^2+(k_1-m)l$. Using $k_1\leqslant k-1$ and $k_2\leqslant m$ gives $f(a)\leqslant ml^2+(k-1-m)l+(m+k-1)(1-l)$ and $f(b)\leqslant ml^2+(k-1-m)l$, thus $\max(f(a),f(b))\leqslant ml^2+(k-1-m)l$ since $1\leqslant l$. Therefore $\varepsilon_{k,l}\leqslant 2ml^2\varepsilon+2(k-1-m)l\varepsilon+l\varepsilon\leqslant 2ml^2\varepsilon+2(k-m)l\varepsilon$.
  - $k_1,k_2\geqslant k$, i.e. $a=1$ and $b=l-1$. Since $k_1,k_2\leqslant m$, $\max(f(a),f(b))\leqslant m(l^2-2l+2)$ and $\varepsilon_{k,l}\leqslant 2m(l^2-2l+2)\varepsilon+(l-1)\varepsilon$. This case cannot happen for $l=1$ because we need $a\leqslant b$ so that the sum is not zero; in addition we cannot have $k=0$ here since the first production is either $T_0=\mathbf{1}$ or $T_0=\mathbf{Z}$. Hence $\varepsilon_{k,l}\leqslant 2ml^2\varepsilon-2ml\varepsilon+2(2-l)m\varepsilon+(l-1)\varepsilon\leqslant 2ml^2\varepsilon-2ml\varepsilon+2l\varepsilon\leqslant 2ml^2\varepsilon+2(k-m)l\varepsilon$ since $l\geqslant 2$ and $k\geqslant 1$.
- If $T_k=\Theta T_{k_1}$ or $\Theta T_k=T_{k_1}$ then, respectively, $\tilde{t}_{k,l}=\tilde{t}_{k_1,l}\underline{\otimes}l$ or $\tilde{t}_{k,l}=\tilde{t}_{k_1,l}\underline{\oslash}l$, and $\varepsilon_{k,l}\leqslant\varepsilon_{k_1,l}+\varepsilon$ or $\varepsilon_{k,l}\leqslant\varepsilon_{k_1,l}+2\varepsilon$ from Lemma 3.1 since we suppose that l is exactly represented. Using the induction hypothesis, $k_1<k$ and $l\geqslant 1$ gives again $\varepsilon_{k,l}\leqslant 2ml^2\varepsilon+2(k-m)l\varepsilon$. □

Note that the value of $\varepsilon_{k,l}$ in Proposition 3.2 is a very general bound. The relative error will generally be lower in real cases. For any particular standard specification, it will be possible to compute a better value for $\varepsilon_{k,l}$ by using formulas of Lemma 3.1.

In the above proof, for the case $T_k=T_{k_1}\cdot T_{k_2}$, we did not explicit the order in which the associative product $(\tilde{t}_{k_1,a}\underline{\otimes}\tilde{t}_{k_2,l-a})\underline{\oplus}\cdots\underline{\oplus}(\tilde{t}_{k_1,b}\underline{\otimes}\tilde{t}_{k_2,l-b})$ was computed. Therefore the bound obtained for $\varepsilon_{k,l}$ holds for any order of computation, in particular either the sequential one or the boustrophedonic one.

4. Random generation

In this section, we describe two variations of the classical recursive method of uniform random generation. The first one – quasi-uniform generation – is not really new: it consists in applying exactly the algorithms of [9], with floating-point arithmetic (here we consider for example rounding towards $-\infty$) in place of exact arithmetic. Its precision – i.e. the maximal relative difference between the probability of a given structure to be generated and the uniform probability – strongly depends on the precision of the floating-point representation, say the number of bits in the mantissa of the floating-point numbers. This is detailed in the following theorem.

Theorem 4.1 (Precision and complexity of quasi-uniform generation). *Let C be a class of combinatorial structures whose standard specification admits $m+1$ classes. If we are given a perfect uniform generator of random real numbers between* 0 *and* 1, *the recursive algorithm using floating-point arithmetic will generate a structure of size n with a probability $\tilde{p}_{m,n}$ such that*

$$p_n(1-\alpha_{m,n}) \leqslant \tilde{p}_{m,n} \leqslant p_n(1-\alpha_{m,n})^{-1},$$

with

$$\alpha_{m,n} = \mathrm{O}(m^2 n^3 2^{-b})$$

where $p_n = 1/c_n$ is the uniform probability over the elements of C of size n, and b is the length of the mantissa of floating-point numbers.

The complexity of the preprocessing stage is $\mathrm{O}(n^2 M(b))$; *the worst-case complexity of one generation is* $\mathrm{O}(n \log n M(b))$, *where $M(b)$ is the worst-case complexity of any standard arithmetic operation on floating-point numbers having a mantissa of size b.*

Proof. In order to generate a structure from class C with size n, we make at most $(n+1)(m+1)$ choices, each of them with a probability equal to $\tilde{a}_n \underline{\oslash} \tilde{c}_n$ (sum), or to $(\tilde{a}_k \underline{\otimes} \tilde{b}_{n-k}) \underline{\oslash} \tilde{c}_n$ (product).[6] No choice has to be made for the pointing/unpointing constructions. The a's, b's and c's having been computed as in Proposition 3.2, each choice introduces a relative error of at most $\mathrm{O}(mn^2\varepsilon)$ with $\varepsilon = 2^{1-b}$. This implies the first part of the theorem. The second part follows directly from the complexity given in [9]. □

The above result gives the possibility to generate quasi-uniform random structures of reasonable size with "standard" programs in usual languages. For example, given a standard specification with nine classes, and using floating-point numbers with a mantissa of length 53 (standard "double" floating-point numbers), one can generate random objects up to a size of 500 with a relative error of order 10^{-6}, if the coefficients

[6] The exact probability depend on how we accumulate the products $\pi_k = (\tilde{a}_k \underline{\otimes} \tilde{b}_{n-k}) \underline{\oslash} \tilde{c}_n$, but this affects lower-order terms only.

are small enough (less than 1.8×10^{308}) so that no overflow occurs. This will hold when the exponential growth is at most in 4^n, which is the case for binary trees for example.

We stated this theorem for the rounding towards $-\infty$ mode, but it holds for the other rounding modes too. We do not give any precise value for the constant behind the $O(\cdot)$, since a precise analysis will be given below.

The second variation, which we call the "ADZ method" – from its inventors Alonso, Denise, Zimmermann – , is devoted to exact uniform generation using floating-point numbers. The main idea consists in computing approximate coefficients and probabilities, and to control their relative error in relation to the corresponding exact values. For example, suppose that we have to make a choice with a certain (exact) probability p (depending on the coefficients computed using the standard specification recurrences). Floating-point arithmetic does not allow us to compute p, but we can compute two floating-point numbers p^- and p^+ such that $p^- \leqslant p \leqslant p^+$. Now, in order to make a choice, we draw a random number $0 \leqslant r \leqslant 1$ and we compare it to p^+ and p^-. If $r < p^-$ or $p^+ \leqslant r$ then we can make the choice; otherwise, r is located in the "error interval".

In this case, there are two possibilities: either we compute (again) the coefficients with *exact* integer arithmetic and run the standard algorithm to continue the generation, or we continue with floating-point arithmetic using a twice longer mantissa, and so on until we can make the choice. The worst-case complexity of the latter method is not bounded, but it is better on average, by a constant factor only; we will not analyze it further.

According to these principles, we present below the main generation schemes, for the sum $C = A + B$ and the product $C = A \cdot B$, based on the corresponding ones in [9]. The other ones (initial structures, pointing and unpointing) are straightforward to write, as no choice has to be made.

```
Case: $C = A + B$.
    gC := procedure($n$: integer);
        $U$ := Uniform([0, 1]);
        $F := 1 \underline{\ominus} ((2m\varepsilon) \overline{\otimes} n \overline{\otimes} n)$;                    (*)
        $p^- := (\tilde{a}_n \underline{\oslash} \tilde{c}_n) \underline{\otimes} F$;
        $p^+ := (\tilde{a}_n \overline{\oslash} \tilde{c}_n) \overline{\oslash} F$;
        if $U < p^-$ then Return(gA($n$))
        else if $U > p^+$ then Return(gB($n$))
        else Special($U$, gA($n$), gB($n$))
    end.
```

```
Case: $C = A \cdot B$.
    gC := procedure($n$: integer);
        $U$ := Uniform([0, 1]);
        $F := 1 \underline{\ominus} ((2m\varepsilon) \overline{\otimes} n \overline{\otimes} n)$                     (*)
        $k := 0$;
        $S := (\tilde{a}_0 \underline{\otimes} \tilde{b}_n)$;
        $p^- := (S \underline{\oslash} \tilde{c}_n) \underline{\otimes} F$;
        $p^+ := (S \overline{\oslash} \tilde{c}_n) \overline{\oslash} F$;
    while $U \geqslant p^+$ do
        $k := k + 1$;
        $S := S \underline{\oplus} (\tilde{a}_k \underline{\otimes} \tilde{b}_{n-k})$;
        $p^- := (S \underline{\oslash} \tilde{c}_n) \underline{\otimes} F$;
        $p^+ := (S \overline{\oslash} \tilde{c}_n) \overline{\oslash} F$;
    if $U < p^-$
        then Return([gA($k$), gB($n - k$)])
        else Special($U$, [gA($k$), gB($n - k$)], [gA($k + 1$), gB($n - k - 1$)]);
    end.
```

The procedure call "Special(U,choice1,choice2)" does the following: compute exactly the probability p, evaluate and return "choice1" if $U < p$, evaluate and return "choice2" otherwise; then use the exact algorithm [9] for the rest of the computation.

Here are some remarks on these generation schemes. First, they involve only standard floating-point operations. In other words, they can be quite directly programmed in any language with rounding modes, provided that the given language supports arithmetic operations with arbitrary precision integers. In the calculation of F, we suppose that $2m$ and n are small enough to be represented exactly.[7] We suppose U to be a uniformly chosen number between 0 and 1; this is of course not possible strictly speaking, since it would need an infinite memory. But, if we are given a perfect generator of 0–1 bits, then U can be generated using a "lazy" process, bit by bit, and the needed comparisons done after each step. It can be proved easily that the average number of bits to be generated in order to compare U with a random number uniformly distributed in $[0, 1]$ equals 2.

In the rest of this section, we focus on the complexities of the ADZ method. The following proposition gives bounds for the "error interval" of the computed probabilities.

Proposition 4.2. *In both cases $C = A + B$ and $C = A \cdot B$, the probability of each* Return *call is less than or equal to the exact probability of the corresponding choice leading to an uniform distribution. Furthermore, if $(2mn^2 + 3)\varepsilon \leqslant 1/2$, then the probability of each* Special *call is bounded by $3(2mn^2 + 3)\varepsilon$.*

Proof. Case $C = A + B$. The probability of Return(gA(n)) being called is p^-, whereas the probability of Return(gB(n)) is $1 - p^+$, therefore we have to prove that $p^- \leqslant p \leqslant p^+$, where $p = a_n / c_n$ is the exact probability of choosing A. The probability of Special(U,gA(n),gB(n)) is clearly $p^+ - p^-$.

[7] As ε is a power of two, it can always be exactly represented, and so $2m\varepsilon$ if m does.

Let $N = 2mn^2\varepsilon$ and $\tilde{N} = (2m\varepsilon)\overline{\otimes}\, n \overline{\otimes}\, n \geqslant N$. By Proposition 3.2, we know that $a_n(1 - N) \leqslant \tilde{a}_n \leqslant a_n$ and $c_n(1-N) \leqslant \tilde{c}_n \leqslant c_n$; so $(1-N)\tilde{a}_n/\tilde{c}_n \leqslant a_n/c_n \leqslant (1-N)^{-1}\tilde{a}_n/\tilde{c}_n$. It follows that $p^- \leqslant a_n/c_n \leqslant p^+$.

By the properties of floating-point operations, we have $N \leqslant \tilde{N} \leqslant N(1+\varepsilon)^2$. Thus, since $(1+\varepsilon)^2 \leqslant (1-\varepsilon)^{-2}$, we have $1 - N(1-\varepsilon)^{-2} \leqslant 1 - \tilde{N} \leqslant 1 - N$. Therefore,

$$1 - \varepsilon - N(1-\varepsilon)^{-2} \leqslant F \leqslant 1 - N$$

where $F = 1 \underline{\ominus} \tilde{N}$ as stated in the above algorithms.

Now let us look at p^-. We have $(\tilde{a}_n/\tilde{c}_n)F(1-\varepsilon)^2 \leqslant (\tilde{a}_n \underline{\oslash}\, \tilde{c}_n) \underline{\otimes} F \leqslant (\tilde{a}_n/\tilde{c}_n)F$; thus

$$\frac{\tilde{a}_n}{\tilde{c}_n}((1-\varepsilon)^3 - N) \leqslant p^- \leqslant \frac{\tilde{a}_n}{\tilde{c}_n}(1-N)$$

since $p^- = (\tilde{a}_n \underline{\oslash}\, \tilde{c}_n) \underline{\otimes} F$. Similarly, we find that

$$\frac{\tilde{a}_n}{\tilde{c}_n}\frac{1}{1-N} \leqslant p^+ \leqslant \frac{\tilde{a}_n}{\tilde{c}_n}\frac{1}{(1-\varepsilon)^3 - N}.$$

Therefore we get

$$p^+ - p^- \leqslant \frac{\tilde{a}_n}{\tilde{c}_n}\left(\frac{1}{(1-\varepsilon)^3 - N} - ((1-\varepsilon)^3 - N)\right)$$

and thus, since $1 - 3\varepsilon \leqslant (1-\varepsilon)^3$, we have

$$p^+ - p^- \leqslant \frac{\tilde{a}_n}{\tilde{c}_n}\left(\frac{1}{1-N'} - (1-N')\right)$$

where $N' = N + 3\varepsilon = (2mn^2 + 3)\varepsilon$. If we suppose, as stated in the Proposition, that $N' \leqslant 1/2$, then $1/(1-N') \leqslant 1 + 2N'$ and it follows that

$$p^+ - p^- \leqslant \frac{\tilde{a}_n}{\tilde{c}_n} 3N',$$

and the proof is complete since $\tilde{a}_n \leqslant \tilde{c}_n$.

Case $C = A \cdot B$. Let p_k^-, p_k^+ denote the values of p^-, p^+ at step k, with $p_{-1}^+ = 0$ by convention. The statement Return([gA(k), gB($n-k$)]) is executed when $p_{k-1}^+ \leqslant U < p_k^-$, and the Special statement when $p_k^- \leqslant U < p_k^+$.

If we substitute a_n by $s_k = a_0 b_n + a_1 b_{n-1} + \cdots + a_k b_{n-k}$ and $\tilde{a}_n$ by S in the above proof for $C = A + B$, we obtain that $p_k^- \leqslant s_k/c_n \leqslant p_k^+$ and $p_k^+ - p_k^- \leqslant 3(2mn^2 + 3)\varepsilon$. Whence the probability of Return([gA(k), gB($n-k$)]) being called is $p_k^- - p_{k-1}^+ \leqslant s_k/c_n - s_{k-1}/c_n$, the latter probability corresponding to the uniform distribution. The probability of a Special call at step k is $p_k^+ - p_k^- \leqslant 3(2mn^2 + 3)\varepsilon$. □

Now we are able to compute the average-case complexity of the ADZ method, according to n and to the computer precision ε. (Recall that $\varepsilon = 2^{1-b}$ where b is the length of the mantissa of floating point numbers.) In these results, we consider m as a constant, since this number only depends on the class of structures to be generated.

Theorem 4.3 (Average and worst-case complexities of the ADZ method). *The average bit-complexity of the ADZ method preprocessing, according to n and to the computer precision ε, is*

$$P_1(n,\varepsilon) = \mathrm{O}\left(n^2 M\left(\log\frac{1}{\varepsilon}\right)\right);$$

the average bit-complexity for the generation of one structure is

$$C_1(n,\varepsilon) = \mathrm{O}\left(n\log nM\left(\log\frac{1}{\varepsilon}\right) + n^6\varepsilon M(n\log n)\right),$$

where $M(x)$ stands for the cost of multiplying two x-bit numbers. The average space complexity is $\mathrm{O}(n\log(1/\varepsilon) + n^6\varepsilon\log n)$. The corresponding worst-case complexities, both in time and space, are the same that the ones for generation with exact arithmetic, as stated in Table 1.

Proof. The results concerning $P_1(n,\varepsilon)$ and the worst-case complexities are straightforward. So let us focus on $C_1(n,\varepsilon)$, and let us bound first the total probability to be forced to run the algorithm which uses exact coefficients. It follows from Proposition 4.2 that, at each step, the probability to run the procedure Special( ) is $\mathrm{O}(n^3\varepsilon)$, since there are at most n "error intervals" in the case $C = A\cdot B$. And we know that there are $\mathrm{O}(n)$ choices to be done during the whole generation. Thus the total probability to run Special( ) during the generation is $\mathrm{O}(n^4\varepsilon)$.

The integer coefficients occurring in the recursive method having size $\mathrm{O}(n\log n)$ [9], the worst-case complexity of Special( ) is $\mathrm{O}(n^2M(n\log n))$; this is the complexity of generating a structure with the exact algorithm (including the preprocessing stage). On the other hand, the complexity of generating a structure if there is no call to Special( ) (once the preprocessing is done and using the boustrophedonic algorithm) is $\mathrm{O}(n\log nM(\log(1/\varepsilon)))$, since the value $\log(1/\varepsilon)$ represents the number of bits of the mantissa of floating-point numbers. Hence the average-case complexity of the algorithm is $\mathrm{O}(n\log nM(\log(1/\varepsilon))) + \mathrm{O}(n^4\varepsilon)\cdot\mathrm{O}(n^2M(n\log n))$.

In the preprocessing, $\mathrm{O}(n)$ approximate coefficients of size $\mathrm{O}(\log(1/\varepsilon))$ are computed, while in the case where Special( ) is called – which occurs with probability $\mathrm{O}(n^4\varepsilon)$ – $\mathrm{O}(n)$ exact coefficients of size $\mathrm{O}(n\log n)$ are computed. Therefore the average space complexity is $\mathrm{O}(n\log(1/\varepsilon) + n^6\varepsilon\log n)$. □

The above result is particularly interesting if there is a possibility to adjust the computer precision (i.e. the length of the mantissa) according to n. In this case, the following easy corollary holds.

Corollary 4.4. *If $\varepsilon = 1/n^7$, then*

$$C_1(n,\varepsilon) = \mathrm{O}(n\log nM(\log n)).$$

This follows from the fact that $M(n\log n) \leqslant M(n)M(\log n) \leqslant n^2M(\log n)$.

Remark. The preprocessing cost $P_1(n,\varepsilon)$ depends linearly on the number m of structures of the specification, like the space complexity, while the average bit-complexity $C_1(n,\varepsilon)$ depends quadratically on m. One factor m comes from the probability of a Special call (Proposition 4.2) and another factor of m from the number of choices to be done during the generation, i.e. the depth of the parse tree, which is $\mathrm{O}(mn)$.

5. Experimental results

In this section, we demonstrate the efficiency of the original method presented in this paper. We will show first the accordance of floating-point approximations obtained with Proposition 3.2, then study the failure probability of the exact uniform random generation algorithm of Section 4, i.e. the probability one has to restart the whole computation using an arbitrary precision arithmetic, and the efficiency of the quasi-random generation algorithm compared to existing packages such as Combstruct.

We take as example Motzkin trees, whose random generation was already considered in the literature [1]. Motzkin (or unary–binary) trees are defined by the specification

$$M = \mathbf{Z} + \mathbf{Z}\cdot M + \mathbf{Z}\cdot M\cdot M,$$

or in standard form with $M = T_4$: $T_0 = \mathbf{Z}$, $T_1 = T_0 \cdot T_3$, $T_2 = T_4 \cdot T_4$, $T_3 = T_2 + T_4$, $T_4 = T_0 + T_1$.

5.1. Accuracy

Let M_n denote the number of Motzkin trees of size n. Due to the exponential growth of M_n (M_{700} is too large to fit in a `double` which is limited to 10^{308} or so), we had to write a special interval arithmetic library using a `double` as mantissa (53 significant bits) and an `int` as exponent (32 bits). Instead of using the result of Proposition 3.2, which enables one to compute only a lower bound of the coefficients, we have computed both lower and upper floating-point bounds using the rounding functions provided by the IEEE 754 standard. The approximations obtained are much better, since they depend on the actual specification. We proceeded in three different ways: (i) first by the usual quadratic method, accumulating convolutions $c_n = \sum a_k b_{n-k}$ from the left to the right; (ii) secondly using the same quadratic method, but accumulating convolutions from the middle terms to the left and right; (iii) using the linear recurrence

$$M_n = \frac{2n-1}{n+1} M_{n-1} + \frac{3n-6}{n+1} M_{n-2}$$

satisfied by the numbers M_n. Such a recurrence exists for any context-free grammar (i.e. when only the union and product constructions are used), and it can be computed from the grammar using the Gfun package [17]. Thanks to the IEEE 754 standard, the computed lower and upper bounds are guaranteed to be exact, but differ according to the way of computation.

Table 2

n	M_n	$-\lg \varepsilon^1_{4,n}$	$-\lg \varepsilon^2_{4,n}$	$-\lg \varepsilon^3_{4,n}$
1000	2×10^{472}	38.5	40.7	40.1
2000	1×10^{949}	37.0	39.5	39.0
5000	6×10^{2379}	35.0	38.0	37.5
10 000	8×10^{4764}	33.5	36.8	36.5
Fit		$53.5 - 1.5 \lg n$	$52.3 - 1.2 \lg n$	$51.0 - 1.1 \lg n$

Table 3

n	Special( ) calls
10^4	$1/10\,000 \approx 0.0001$
10^5	$7/1000 \approx 0.007$
2×10^5	$17/500 \approx 0.034$
5×10^5	$64/200 \approx 0.32$
10^6	$95/100 \approx 0.95$

Table 2 indicates for different sizes the accuracy[8] $-\lg \varepsilon_{4,n}$ with $\varepsilon_{4,n}$ as in Proposition 3.2 for the nonterminal $T_4 = M$, i.e. the number of common correct bits between the lower and upper bounds, obtained with each of the three ways of computing M_n.

We can conclude from Table 2 that method (ii) is slightly better than method (i). This can be explained by the fact that the coefficients M_n grow like $a^n n^{-3/2}$, which holds for most data structures having an algebraic generating function like various kinds of trees, and therefore the middle terms in the convolutions are smaller than the outer terms by a factor of about $n^{3/2}$. Another conclusion is that in all three cases the accuracy is better than the worst-case of $c - 2\lg n$ given by Proposition 3.2. The linear recurrence even gives a quasi-linear behaviour.

5.2. Special calls

Table 3 indicates the proportion of random generations which required calls to Special( ) for several experiments with the algorithm using floating-point intervals, still for Motzkin trees. For size 10^5 for instance, only 7 random generations over 10^3 called the Special( ) function.

It appears from Table 3 that the bound of $6mn^4\varepsilon$ which follows from Proposition 4.2 is very pessimistic. The actual failure probability seems to behave quadratically with n.

5.3. Efficiency

Table 4 compares the ADZ method with the Maple implementation of the standard algorithm in the Combstruct package [18], for the generation of Motzkin trees. Comb-

[8] We denote by lg the binary logarithm.

Table 4

n	Maple/Combstruct		ADZ method	
	Count	Draw	Count	Draw
100	0.9/0.04	0.08	0.03/0.00	0.002
200	3.7/0.1	0.18	0.08/0.01	0.005
500	61/0.45	0.65	0.55/0.01	0.012
1000	613/2.0	2.66	2.3/0.02	0.028
2000	4468/5.7	9.7	9.8/0.04	0.056
5000	NA/35.	82	66/0.08	0.163
10 000	NA/162	586	282/0.18	0.411
Fit	$n^{3.6}/n^{2.3}$	$n^{3.2}$	$n^{2.0}/n^{1.2}$	$n^{1.1}$

struct uses Maple's integers and interpreted language to implement the counting and random generation routines, while the ADZ method was implemented in the C language, using machine floating-point numbers (`double`). The column "count" gives the time (in seconds on an Ultra Sparc machine) required for the preprocessing, while the column "draw" the average time for one random generation (over 100 generations). The entry NA stands for a computing time greater than two hours.

In the "count" column, the times on the left were obtained with the default $O(n^2)$ method, and those on the right with the linear recurrence computed by the Gfun package [17], after typing `'combstruct/usegfun':=true` in Maple.

6. Conclusion and open questions

In this paper, we have extended to certified floating-point computations the *recursive method* for the random generation of decomposable structures. This extension enables one to generate an object of size n in quasi-linear expected time and space, after a preprocessing of time $O(n^{2+\varepsilon})$, and $O(n^{1+\varepsilon})$ in the context-free case. This method only improves the average complexity. The worst-case complexity remains the same as the standard algorithm with integer arithmetic, both in time and space, as the standard algorithm is called when the one with floating-point arithmetic fails.

In addition to the nice theoretical bounds, the new method also behaves very well in practice, as shown by the experimental figures from Section 5.

Nevertheless, some open questions and places for improvements remain. It would be interesting to analyse exactly the bit-complexity of the standard algorithm. (It will depend on the specification.) Another problem is that the standard floating-point numbers on 64 bits cannot be used for large sizes, because the coefficients become too big. A possible solution that would be interesting to study is the following. Instead of computing floating-point approximations for the coefficients $t_{k,n}$, store the values $t_{k,n}/\rho_k^n$ where ρ_k is the radius of convergence from the generating function associated to the

kth nonterminal. In such a way, only the polynomial part – which is much smaller – would be stored.

Thanks to Isabelle Dutour, there exists now a MuPAD package which implements the algorithms given in [8, 9] and some of the improvements given here, including additional features [7]. For example, the package is able to automatically generate, given a decomposable class, the source of a C program for almost uniform random generation. Moreover, when the generating series of a decomposable class is holonomic, it can compute, using Gröbner basis calculation and Gaussian elimination, a linear recurrence that the coefficients of the generating series satisfy. This improves the complexity of counting and enables the generation of very large structures.

But a lot of questions remain. In the general case, does a recurrence like that found by Euler for partition numbers exist for all decomposable structures? How to guess and prove such a recurrence?

Acknowledgements

We thank Laurent Alonso for giving us the original idea to use floating-point arithmetic, Philippe Flajolet and Jean-Guy Penaud for their active support during the redaction of this paper.

References

[1] L. Alonso, Uniform generation of a Motzkin word, Theoret. Comput. Sci. 134 (1994) 529–536.

[2] L. Alonso, J.-L. Rémy, R. Schott, A linear-time algorithm for the generation of trees, Algorithmica 17 (1997) 162–182.

[3] D.B. Arnold, M.R. Sleep, Uniform random generation of balanced parenthesis strings, ACM Trans. Program. Lang. Systems 2 (1) (1980) 122–128.

[4] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, S.M. Watt, Maple V: Language Reference Manual, Springer, Berlin, 1991.

[5] M. Daumas, J.-M. Muller, Qualité des calculs sur ordinateur, Vers des arithmétiques plus fiables? Masson, Paris, 1997.

[6] A. Denise, Génration aléatoire et uniforme de mots de langages rationnels, Theoret. Comput. Sci. 159 (1) (1996) 43–63.

[7] A. Denise, I. Dutour, P. Zimmermann, CS: a MuPAD package for counting and randomly generating combinatorial structures, Proc. FPSAC'98 (software demonstration), Toronto, June 1998. Also in MathPAD 8 (1) (1998) 23–30. Package available at the following address: http://www.dept-info.labri.u-bordeaux.fr/~dutour/CS.

[8] P. Flajolet, P. Zimmermann, B.V. Cutsem, A calculus of random generation: Unlabelled structures, in preparation.

[9] P. Flajolet, P. Zimmermann, B.V. Cutsem, A calculus for the random generation of labelled combinatorial structures, Theoret. Comput. Sci. 132 (1–2) (1994) 1–35.

[10] M. Goldwurm, Random generation of words in an algebraic language in linear binary space, Inform. Process. Lett. 54 (1995) 229–233.

[11] T. Hickey, J. Cohen, Uniform random generation of strings in a context-free language, SIAM J. Comput. 12 (4) (1983) 645–655.

[12] IEEE standard for binary floating-point arithmetic, Tech. Rep. ANSI-IEEE Standard 754-1985, New York, 1985. approved 21 March 1985: IEEE Standards Board, approved July 26, 1985: American National Standards Institute, 18 pages.

[13] S. Kannan, Z. Sweedyk, S. Mahaney, Counting and random generation of strings in regular languages, Proc. 6th Annual ACM-SIAM Symp. on Discrete Algorithms, San Francisco, California, January 1995, pp. 551–557.

[14] D. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms of Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, Reading, MA, 1969.

[15] H.G. Mairson, Generating words in a context free language uniformly at random, Inform Process. Lett. 49 (2) (1994) 95–99.

[16] A. Nijenhuis, H.S. Wilf, Combinatorial Algorithms, 2nd ed., Academic Press, New York, 1978.

[17] B. Salvy, P. Zimmermann, Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable, ACM Trans. Math. Software 20 (2) (1994) 163–177.

[18] P. Zimmermann, Gaïa: a package for the random generation of combinatorial structures, MapleTechnol. 1 (1) (1994) 38–46.