Algorithmica (2012) 62:130–145
DOI 10.1007/s00453-010-9446-5

A Linear Algorithm for the Random Sampling from Regular Languages

Olivier Bernardi · Omer Giménez

Received: 10 September 2009 / Accepted: 18 August 2010 / Published online: 3 September 2010

Abstract We present the first linear algorithm for the random sampling from regular languages. More precisely, for generating a uniformly random word of length n in a given regular language, our algorithm has worst-case space bit-complexity $O(n)$ and mean time bit-complexity $O(n)$. The previously best algorithm, due to Denise and Zimmermann (Theor. Comp. Sci. 218(2):233–248, 1999), has worst-case space bit-complexity $O(n^2)$ and mean time bit-complexity $O(n\log(n))$. The Denise et al. algorithm was obtained by performing a floating-point optimization on the general *recursive method* formalized by Nijenhuis and Wilf (and further developed by Flajolet, Zimmermann and Van Cutsem). Our algorithm combines the floating-point optimization with a new divide-and-conquer scheme.

Keywords Random sampling · Regular languages

1 Introduction

A *random generation algorithm*, or *RGA* for short, for a combinatorial class $\mathcal{C}$ is an algorithm that receives an integer n as input, and outputs an object chosen uniformly at random among the objects of size n in $\mathcal{C}$. RGAs find applications in areas such as hardware and software testing, coding theory and bioinformatics; also, they are valuable tools for conjecturing some probabilistic properties of the classes of objects they sample from. These needs have motivated the search for efficient RGAs. Although several papers are devoted to RGAs particular to a special class of objects

O. Bernardi
MIT, Cambridge, 02139 MA, USA
e-mail: bernardi@math.mit.edu

O. Giménez (✉)
Universitat Politecnica de Catalunya, Barcelona, Spain
e-mail: omer.gimenez@gmail.com

(for instance, planar graphs [10]), it is even more important to come up with general strategies for designing RGAs which apply to a whole family of classes. In particular, much attention has been set on the family of regular languages [5, 13], on the family of algebraic languages (in particular, codes for trees) [1, 2, 7, 11, 12, 15], and more generally on the family of *recursive combinatorial classes* [9, 17].

For a given recursive combinatorial class, a RGA can be obtained by following the so-called *recursive method* originating from the work of Nijenhuis and Wilf [17] and further developed by Flajolet, Zimmermann and Van Cutsem [9]. These *recursive RGAs* are the ones classically used in packages like *Combstruct* (under the computer algebra MAPLE) and *CS* (under MUPAD). A different approach, the so-called ECO method follows from the work of Barcucci et al.; this approach applies to combinatorial classes that can be described in terms of succession rules [2, 3]. More recently, Duchon et al. have shown how to design *Boltzmann samplers* for recursive combinatorial classes [8]. For a given class $\mathcal{C}$, a Boltzmann sampler receives a positive real parameter z, and outputs an object $c \in \mathcal{C}$ with a probability proportional to $z^{|c|}$, where $|c|$ is the size of c (this makes sense for z small enough).

The performance of a RGA can be measured either in terms of its *integer-complexity* (where storing an integer costs 1 unit of space and a comparison or arithmetic operation on integers costs 1 unit of time) or in terms of its *bit complexity* (where storing a bit costs 1 unit of space and a comparison or arithmetic operation on integers costs 1 unit of time). For RGAs, the two measures are really different because most algorithms have to manipulate integers growing exponentially in the length n of the word to be generated (this feature was already mentioned in [9]). For instance, the recursive method for context-free languages leads to a linear integer-complexity but to a quadratic bit-complexity in both time and space. In the following, *we shall only use the bit-complexity*, which represents a much more realistic measure of practical costs of the algorithms. Another important remark is that random generation algorithms rely on the use of a random number generator in order to perform random choices. This explains that we shall focus on *mean* time complexity (since the worst-case complexity is infinite) and consider that generating a (uniformly) random bit costs 1 unit of time.

In this article we study the problem of designing efficient RGAs for regular languages (equivalently, sets of words recognized by a deterministic finite automaton, sets of words generated by a regular expression, sets of paths within a directed graph). This family indeed deserves special attention because of its omnipresence in computer science and bioinformatics. The best known RGA for regular languages was obtained by Denise and Zimmermann in [7] by adapting the classical recursive method to certified floating-point arithmetics. It should be clear that, although using floating point arithmetic, this RGA is *exact*: the probability measure on the output is *exactly* uniform. For generating a word of length n in a given regular language, the RGA of Denise and Zimmermann has mean-time bit complexity $O(n\log(n))$ and a worst case bit-complexity $O(n^2)$ (although the space complexity is $O(n)$ in most cases). Comparing with this result, Boltzmann samplers (piled up with a rejection procedure) would give an algorithm having time-complexity of order $O(n^2)$ if no margin of error is accepted on the length n of the word to be generated (see [8, Theorem 5]). The ECO method has a time complexity of $O(n^3)$ integer operations in the general case.

In this contribution we present the first space- and time-optimal algorithm for the random sampling from regular languages. More precisely, we show an algorithm with mean-time bit-complexity $O(n)$ and worst-case space bit-complexity $O(n)$ to generate uniformly at random a word of n characters. The new idea underlying this improvement is a divide-and-conquer scheme for counting and generating words in a regular language. If implemented *naively* with exact integer representation, the *divide-and-conquer RGA* has linear bit-complexity in space but not in time. However, we show that linear time complexity can be achieved by adapting our RGA to certified floating-point arithmetics. In practice, our algorithm outperforms previously known algorithms for large values of n and allows to generate words of length one billion on a standard PC. On a more theoretical point of view, our result proves the linearity of the complexity of random sampling for a large and important class of languages. The most obvious question left open is whether a linear time RGA exists for any context-free language.

The paper is organized as follows. In Sect. 2, we recall the definition of the recursive RGA for regular languages and then describe the divide-and-conquer RGA. In Sect. 3, we discuss the adaptation of both RGAs to floating-point arithmetics and compare the complexities of the algorithms. In Sect. 4, we discuss implementation issues and compare experimental performances of both RGAs. We conclude in Sect. 5 with some remarks and possible extensions.

2 Description of the Random Generation Algorithms

2.1 Regular Languages and Automatons

We first recall some definitions about regular languages. An *alphabet* is a finite set whose elements are called *letters*. A *word* is a finite sequence of letters; a *language* is a set of words. An (finite, deterministic and complete) *automaton* on an alphabet $\mathbf{A}$ is a quadruple $\mathbf{M} = (Q, q_0, F, \sigma)$, where Q is the set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ are the *final states*, and $\sigma : Q \times \mathbf{A} \mapsto Q$ is the *transition function*. The (labelled) *digraph* $G_\mathbf{M}$ *associated to* $\mathbf{M}$ is the digraph whose vertices are the states of $\mathbf{M}$ and where there is an arc from state p to state q with label $a \in A$ if and only if $\sigma(p, a) = q$. A word $w = a_1 a_2 \cdots a_n$ is *accepted* by the automaton $\mathbf{M}$ if w labels a directed path from the initial state to a final state. The *language accepted* by an automaton $\mathbf{M}$, denoted by $\mathcal{L}(\mathbf{M})$, is the set of words accepted by $\mathbf{M}$. A language $\mathcal{L}$ is *regular* if there is an automaton $\mathbf{M}$ such that $\mathcal{L} = \mathcal{L}(\mathbf{M})$.

In the following, we let $\mathbf{M} = (Q, q_0, F, \sigma)$ be an automaton on the alphabet $\mathbf{A}$ with accepted language $\mathcal{L} = \mathcal{L}(\mathbf{M})$. For any state p in Q, we denote by $\mathbf{M}_p$ the automaton (Q, p, F, σ). Similarly, for any pair of states q, q' in Q, we denote by $\mathbf{M}_{q,q'}$ the automaton $(Q, q, \{q'\}, \sigma)$. We also denote by $\mathcal{L}_q = \mathcal{L}(\mathbf{M}_q)$ (resp. $\mathcal{L}_{q,q'} = \mathcal{L}(\mathbf{M}_{q,q'})$) its accepted language. Finally, let $l_{q,n}$ (resp. $l_{q,q',n}$) denote the number of words of length n in this language. In particular, $\mathcal{L} = \mathcal{L}_{q_0} = \biguplus_{q' \in F} \mathcal{L}_{q_0,q'}$.

2.2 Recursive RGA

We first recall the definition of the recursive RGA for the language $\mathcal{L}(\mathbf{M})$. We consider the (random) first letter a_1 of a word $w = a_1 a_2 \cdots a_n$ chosen uniformly at random among the word of length n in $\mathcal{L}(\mathbf{M}) = \mathcal{L}_{q_0}$. For any letter $a \in A$, the probability that $a_1 = a$ is $\frac{l_{\sigma(q_0,a),n-1}}{l_{q_0,n}}$. Moreover, conditionally upon a_1, the subword $w' = a_2 \cdots a_n$ is a word of length $n-1$ chosen uniformly at random in $\mathcal{L}_{q_1}$, where $q_1 = \sigma(q_0, a_1)$. Thus, if all the numbers $l_{q,m}$ for $q \in Q$ and $m = 0, 1, \dots, n-1$ are preprocessed, one can choose the letters $a_1, a_2, \dots, a_n$ of w sequentially with suitable probabilities. This leads to the following algorithm.

Definition 1 [9, 17] Recursive RGA:

```
(0)  Input: an automaton $\mathbf{M} = (Q, q_0, F, \sigma)$ and an integer $n$.
Preprocessing:
(1)  Set $l_{q,0} = 1$ if $q$ is a final state and $l_{q,0} = 0$ otherwise.
(2)  For $i$ from 1 to $n$ do:
(3)      For all $q \in Q$, $l_{q,i} = \sum_{a \in \mathbf{A}} l_{\sigma(q,a),i-1}$.
Generation:
(4)  Set $q = q_0$.
(5)  For $i$ from 1 to $n$ do:
(6)      Choose the letter $a_i = a$ in $A$ with probability $\frac{l_{\sigma(q,a),n-i}}{l_{q,n-i+1}}$.
(7)      Set $q = \sigma(q, a_i)$.
(8)  Return the word $w = a_1 a_2 \cdots a_n$.
```

It should be clear from the above discussion that the recursive RGA outputs a word of length n uniformly at random in the language $\mathcal{L}(\mathbf{M})$.

2.3 Divide-and-Conquer RGA

Recall first that the words of the language $\mathcal{L} = \mathcal{L}(M)$ are in one-to-one correspondence with the directed paths from the initial state to a final state in the digraph $G_{\mathbf{M}}$ and that the number $l_{q,q',n}$ counts the directed paths of length n from state q to state q' in this digraph. Now, the problem of generating uniformly at random a path of length $2n$ going from a state q to a state q' can be solved recursively by choosing the *middle* state p (reached after n steps) with suitable probability and then generate uniformly at random paths of length n from q to p and from p to q'. Moreover, the *suitable probability* for choosing the middle state p is $\frac{l_{q,p,n} \cdot l_{p,q',n}}{l_{q,q',2n}}$. This simple observation underlies the divide-and-conquer RGA. For simplicity, we first describe this algorithm when the length n of the word to be generated is a power of 2.

Definition 2 Divide-and-conquer RGA:

```
(0)  Input: an automaton $\mathbf{M} = (Q, q_0, F, \sigma)$ and an integer $n = 2^k$.
Preprocessing:
(1)  Set $l_{p,q,1}$ to be the number of arcs going from $p$ to $q$ in $G_{\mathbf{M}}$.
(2)  For $i$ from 2 to $k$ do:
(3)      For all $q, q' \in Q$, set $l_{q,q',2^i} = \sum_{p \in Q} l_{q,p,2^{i-1}} \cdot l_{p,q',2^{i-1}}$.
Generation:
(4)  Choose the final state $p = q_n$ in $F$ with probability $\frac{l_{q_0,p,n}}{\sum_{f \in F} l_{q_0,f,n}}$.
```

```
(5)  For $i$ from 1 to $k$ do:
(6)      For $j$ from 1 to $2^i$ do:
(7)          Choose the state $p = q_{(2j-1)m}$ with probability $\frac{l_{q,p,m} \cdot l_{p,q',m}}{l_{q,q',2m}}$,
               where $m = 2^{k-i}$, $q = q_{(2j-2)m}$ and $q' = q_{2jm}$.
(8)  For $i$ from 1 to $n$ do:
(9)      Choose the letter $a_i$ uniformly among the letters labelling arcs from
         $q_{i-1}$ to $q_i$.
(10) Return the word $w = a_1 a_2 \cdots a_n$.
```

In the divide-and-conquer RGA, Lines (1)–(3) correspond to the preprocessing of the numbers $l_{p,q,2^i}$ for all p, q in Q and all integer $i \in \{1, \dots, k\}$. Lines (4)–(7) correspond to the choice of the sequence of states $q_0, \dots, q_n$ in a random directed path of length n from the initial state q_0 to a final state q_n. Finally, Lines (8)–(9) correspond to the choice of the label of the arc (letter a_i) between the consecutive states q_{i-1} and q_i, for $i \in \{1, \dots, n\}$. The above discussion should make the following theorem obvious (when n is a power of 2).

Theorem 3 *The divide-and-conquer RGA generates a word of length n uniformly at random in the language $\mathcal{L}(\mathbf{M})$.*

Let us now explain how to deal with the case where n is not a power of 2. An *addition scheme* for n is a sequence $n_1, n_2, \dots, n_s$ such that $n_1 = 1$, $n_s = n$ and for all $k \in \{2, \dots, s\}$ there are indices $i, j < k$ such that $n_k = n_i + n_j$. Given an addition scheme $n_1, n_2, \dots, n_s$ for n, one can preprocess all the numbers l_{q,q',n_i} for $q, q' \in Q$ and $i\{1, \dots, s\}$ recursively by using the relation $l_{q,q',n_k} = \sum_{p \in Q} l_{q,p,n_i} \cdot l_{p,q',n_j}$, where $n_k = n_i + n_j$. Then, at the generation step, a directed path of length $n_k = n_i + n_j$ from a state q to a state q' can be chosen uniformly at random by choosing the state p reached after n_i steps with probability $\frac{l_{q,p,n_i} \cdot l_{p,q',n_j}}{l_{q,q',n_k}}$ and then generating uniformly at random a path of length n_i from q to p and a path of length n_j from p to q'. We omit the specification of the algorithm for generic n and simply observe that if $b_r \dots b_1 b_0$ is the binary representation of n (that is, $n = \sum_{i=0}^{r} b_i 2^i$), then the increasing sequence whose terms are $\{2^i \mid 1 \le i \le s\} \cup \{\sum_{i=0}^{r} b_i 2^i \mid 1 \le i \le s\}$ is an addition scheme for n.

3 Complexity Analysis

In this section we analyze and compare the complexity of the recursive and divide-and-conquer RGAs. We will only be interested in the *bit-complexity* of algorithms, where storing a bit costs 1 unit of space and 1 unit of time, while comparing, adding, subtracting or multiplying two bits costs 1 unit of time. Given an automaton $\mathbf{M}$ with state space $Q = \{p_1, p_2, \dots, p_\mathbf{q}\}$ on the alphabet $\mathbf{A}$, we will express the complexity of the RGAs as a function of the number $\mathbf{q}$ of states and the length n of the word to be generated. The number of letters $\mathbf{a} = |A|$ also enters in the discussion but will be considered a (small) fixed constant.

We also recall our assumptions about random choices occurring in the RGA (for instance in Line (6) of the recursive RGA and Line (7) of the divide-and-conquer RGA). We assume in this paper that one can draw a number uniformly at random from the interval [0, 1]. From now on, the letter X will always denote such a random number. The complexity of producing and reading the first k bits of X is assumed to be $O(k)$. Given these assumptions, comparing X with a fixed real number y has constant mean time complexity but unbounded worst-case time complexity. In Line (6) of the recursive RGA, the choice of the letter a_i is made by comparing a random number X with the threshold values $f_{q,i,r} = \frac{1}{l_{q,n-i+1}} \sum_{j=1}^{r} l_{\sigma(q,\alpha_j),n-i}$ for $r = \{0, \dots, \mathbf{a}\}$ (the letter a_i is the k-th letter of the alphabet $\mathbf{A}$ if and only if $f_{q,i,r-1} \leq x < f_{q,i,r}$). Similarly the choice in Line (7) of the divide-and-conquer RGA is made by comparing X with the threshold values $f_{q,q',m,r} = \frac{1}{l_{q,q',2m}} \sum_{j=1}^{r} l_{q,p_j,m} l_{p_j,q',m}$ for $r = \{0, \dots, \mathbf{q}\}$. Here two strategies are possible: either all the values $f_{q,q',m,r}$ are stored at the preprocessing step or they are computed each time that they are necessary. We call *high-preprocessing* the version of the divide-and-conquer which computes and stores the values $f_{q,q',m,r}$ in the preprocessing step and *low-preprocessing* the other version.[1]

3.1 Exact Integer Representation

We first analyze the complexity of the recursive and divide-and-conquer RGAs using *exact* integer representation.

Proposition 4 *The bit-complexity of the RGAs using exact integer representation is given in Table 1.*

Proof We first analyze the complexity of the recursive RGA and then the two versions of the divide-and-conquer RGA.

Recursive RGA. The preprocessed numbers $l_{q,m}$, for $q \in Q$ and $m \in \{1, \dots, n\}$, are of order $\mathbf{a}^m$, hence they have binary representations of length $O(m)$. Thus, the preprocessing step has space-complexity $O(\mathbf{q} \sum_{m=1}^{n} m) = O(\mathbf{q}n^2)$. Moreover, the cost of an addition of two numbers of length $O(m)$ is $O(m)$, thus the preprocessing step has time-complexity $O(\mathbf{q}n^2)$. At the generation step, the threshold values $f_{q,m,r}$ are compared with random numbers chosen uniformly in [0, 1]. In average, a constant

Table 1 Bit-complexity of RGAs using exact integer representation

RGA:	Recursive	Divide-and-conquer	Divide-and-conquer
Version:		Low-preprocessing	High-preprocessing
Preprocessing space	$\mathbf{q}n^2$	$\mathbf{q}^2 n$	$\mathbf{q}^3 n$
Preprocessing time	$\mathbf{q}n^2$	$\mathbf{q}^3 n^2$	$\mathbf{q}^3 n^2$
Generation time	n	$\mathbf{q}n$	$\log(\mathbf{q})n$

[1] We do not consider here the distinction between high- or low-preprocessing for the recursive RGA, since their complexity are equal up to a factor depending only on the number $\mathbf{a}$ of letters.

number of bits of $f_{q,m,r}$ is sufficient for such a comparison. Hence, by computing the value of the thresholds $f_{q,m,r}$ only with the precision needed, one obtains a mean time-complexity in $O(n)$ for the generation step.

Divide-and-conquer RGA. We analyze the complexity in the special case where n is a power of 2. It can be checked that the complexity obtained is also valid for general n.

We first analyze the low-preprocessing version. The preprocessed numbers $l_{q,q',2^i}$, for $q, q' \in Q$ and $i \in \{1, \dots, \log_2(n)\}$, are of order $\mathbf{a}^{2^i}$, hence they have binary representations of length $O(2^i)$. Thus, the preprocessing step has space-complexity $O(\mathbf{q}^2 \sum_{i=1}^{\log(n)} 2^i) = O(\mathbf{q}^2 n)$. Since the multiplication of two numbers of length $O(b)$ takes time $O(b^2)$ when using a naive algorithm, the preprocessing step has time-complexity $O(\mathbf{q}^3 \sum_{i=1}^{\log_2(n)} 2^{2i}) = O(\mathbf{q}^3 n^2)$. The time-complexity of the generation step is $O(\mathbf{q}n)$ since, as in the case of the recursive RGA, the threshold values $f_{q,q',2^i,r}$ need only to be computed with the precision needed for comparison with a random number X.

We analyze the high-preprocessing version. In this version, the threshold values $f_{q,q',m,r} = \frac{1}{l_{q,q',2m}} \sum_{j=1}^{r} l_{\sigma(q,p_j),m} l_{\sigma(p_j,q'),m}$ are stored with exact representation. More precisely the numerator $s_{q,q',m,r} = \sum_{j=1}^{r} l_{q,p_j,m} l_{p_j,q',m}$ and denominator $l_{q,q',2m} = \sum_{j=1}^{\mathbf{q}} l_{q,p_j,m} l_{p_j,q',m}$ are both stored with exact representation. Since the size of the binary representation of $s_{q,q',m,r}$ is at most the size of the binary representation $l_{q,q',2m}$, the space-complexity is multiplied by at most q between the low-preprocessing and high-preprocessing versions. The time-complexity of the preprocessing step is unchanged between the two versions, since exactly the same computations are performed. The mean time-complexity of the generation step using dichotomic search to compare a random number X with the threshold values $f_{q,q',m,r}$ is then $O(\log(\mathbf{q})n)$. □

Remark Note that the computations performed in Lines (2)–(3) of the divide-and-conquer RGA correspond to a matrix multiplication. Hence, the time-complexity of the preprocessing step could be reduced to $O(\mathbf{q}^{\log_2(7)} n^2)$ by using the Strassen algorithm for matrix multiplication [4]. Time-complexity could also be improved by using more sophisticated algorithms for integer multiplications. For instance, the time-complexity would be $O(\mathbf{q}^{\log_2(7)} n^{\log_2(3)})$ if Karatsuba algorithm is used.

3.2 Floating-point Arithmetic

As already mentioned, the numbers $l_{q,m}$ or $l_{q,q',m}$ preprocessed by RGAs have a binary representation of length $O(n)$. Observe, however, that to choose a letter $a \in A$ at the generation step of the recursive RGA it is often enough to know $O(\log(\mathbf{a}))$ bits of the numbers $l_{q,m}$. This remark, already expressed in [9], can be used to obtain an efficient *approximate RGA*. For any positive real number γ, a *γ-approximate RGA* for a regular language $\mathcal{L}$ takes as input an integer n and outputs a random word W of length n in $\mathcal{L}$ such that the difference of probability $|P(W = w_1) - P(W = w_2)|$ is less than 2γ for any two words w_1, w_2 of length n in $\mathcal{L}$. As we will see below, floating-point arithmetic naturally leads to some γ-approximate RGAs. Furthermore, Denise and Zimmerman have shown in [7] how to design an efficient *exact* RGA using *certified floating-point arithmetic*. We now recall some of the ideas and results of [7] and adapt them to the divide-and-conquer RGA.

Recall from [14] that a floating-point number x is generally represented by three values: a *sign* $s_x \in \{-1, 1\}$, a *mantissa* m_x and an *exponent* e_x such that $x = s_x \cdot m_x \cdot 2^{e_x}$. When computing numbers using a fixed mantissa length b, a rounding mode Δ has to be chosen for arithmetic operations (e.g. toward 0, toward $-\infty$ or toward nearest neighbor). This choice fixes the behavior of the floating-point arithmetic operation as follows: for any two numbers x, y (represented with mantissa length b), the floating-point operation $\circledast$ corresponding to the arithmetic operation $*$ in $\{+, -, \times, /\}$ is such that $x \circledast y = \Delta(x * y)$. We now analyze the error propagation occurring during the preprocessing step of the RGAs using floating-point arithmetic.

Lemma 5 *If the preprocessing step of the recursive* (*resp. divide-and-conquer*) *RGA is performed using floating-point arithmetic with mantissa length* b, *then one obtains approximations* $\widetilde{l}_{q,m}$ *of* $l_{q,m}$ (*resp.* $\widetilde{l}_{q,q',m}$ *of* $l_{q,q',m}$) *satisfying*

$$|\widetilde{l}_{q,m} - l_{q,m}| \leq \frac{2m\mathbf{a}}{2^b} l_{q,m} \quad \textit{and} \quad |\widetilde{l}_{q,q',m} - l_{q,q',m}| \leq \frac{4m\mathbf{q}}{2^b} l_{q,q',m} \tag{1}$$

for all q, q' *in* Q *and all* $m \in \mathbb{N}$.

Proof The analysis is very similar to the one performed in [7] and we will only do it for the divide-and-conquer algorithm in the case where n is a power of 2. The basic property of the floating-point operations $\oplus$ and $\otimes$ is that if x, y are real numbers and $\widetilde{x}$, $\widetilde{y}$ are floating-point numbers (represented with mantissa of length b) satisfying $|\widetilde{x} - x| < x\delta_x$ and $|\widetilde{y} - y| < y\delta_y$, then

$$|(\widetilde{x} \oplus \widetilde{y}) - (x + y)| \leq |x + y|(\max(\delta_x + \delta_y) + 2^{1-b})$$

and

$$|(\widetilde{x} \otimes \widetilde{y}) - (x \times y)| \leq |x \times y|(\delta_x + \delta_y + 2^{1-b}).$$

Thus, by denoting δ_m the minimum real number such that $|\widetilde{l}_{q,q',m} - l_{q,q',m}| < l_{q,q',m}\delta_m$ for all pair of states q, q' in Q, one easily obtains $\delta_{2^i} \leq 2\delta_{2^{i-1}} + \frac{2\mathbf{q}}{2^b}$. From this bound, a simple induction shows that $\delta_{2^i} \leq \frac{2\mathbf{q}}{2^b}(2^{i+1} - 1)$ and the result follows. □

3.3 Approximate RGAs

The bound on error propagation given by Lemma 5 allows one to design γ-approximate RGAs. Recall that at the generation step of the recursive RGA, the random choices are made by drawing a number X uniformly at random from the interval [0, 1] and comparing it with some threshold values $f_{q,m,r}$ or $f_{q,q',m,r}$. Let us denote respectively by $\widetilde{f}_{q,m,r}$ and $\widetilde{f}_{q,q',m,r}$ the threshold values obtained by using the approximations $\widetilde{l}_{q,m}$ and $\widetilde{l}_{q,q',m}$ instead of $l_{q,m}$ and $l_{q,q',m}$. From (1) one easily gets

$$|\widetilde{f}_{q,m,r} - f_{q,m,r}| \leq \frac{4\mathbf{a}n}{2^b} \quad \text{and} \quad |\widetilde{f}_{q,q',m,s} - f_{q,q',m,s}| \leq \frac{8\mathbf{q}n}{2^b} \tag{2}$$

for all $q, q' \in Q$, $m \leq n$, $r \in \{1 \dots \mathbf{a}\}$ and $s \in \{1 \dots \mathbf{q}\}$.

It is clear from (2) that for all $q, q' \in Q$ and all $m \leq n$ there is a probability at most $\frac{8\mathbf{a}n}{2^b}$ that a uniformly random number X compares differently with the approximate threshold $\widetilde{f}_{q,m,r}$ and with the exact threshold $f_{q,m,r}$. Moreover, the number of comparisons made at the generation step of the recursive RGA is at most $\mathbf{a}n$. Hence, there is a probability at most $\mathbf{a}n \cdot \frac{8\mathbf{a}n}{2^b}$ that the word $\widetilde{w}$ generated by the recursive RGA using the approximate thresholds $\widetilde{f}_{q,m,r}$ is different from the world w that would have been generated by using the exact thresholds $f_{q,m,r}$. Thus, running the recursive RGA with mantissa length $b = \lceil \log(\frac{8\mathbf{a}^2 n^2}{\gamma}) \rceil$ gives a γ-approximate RGA.

Similarly, for the divide-and-conquer RGA, the probability that a the word $\widetilde{w}$ generated using the approximate thresholds $\widetilde{f}_{q,q',m,s}$ is different from the world w that would have been generated using the exact thresholds $\widetilde{f}_{q,q',m,s}$ is at most $\mathbf{q}n \cdot \frac{16\mathbf{q}n}{2^b}$. Thus, using a mantissa length $b = \lceil \log(\frac{16\mathbf{q}^2 n^2}{\gamma}) \rceil$ gives a γ-approximate RGA. This leads to the following result.

Proposition 6 *The complexity of the γ-approximate RGAs is given in Table 2.*

Proof Due to the previous discussion, it only remains to analyze the complexity of the preprocessing step of the recursive RGA (resp. divide-and-conquer RGA) with floating-point arithmetic using a mantissa length $b = \lceil \log(\frac{8\mathbf{a}^2 n^2}{\gamma}) \rceil = O(\log(n/\gamma))$ (resp. $b = \lceil \log(\frac{16\mathbf{q}^2 n^2}{\gamma}) \rceil = O(\log(\mathbf{q}n/\gamma))$). Moreover, the analysis made in the proof of Proposition 4 shows that the complexity of the preprocessing step performed with a fixed mantissa length b is the one indicated in Table 3. From this, Proposition 6 follows. □

3.4 Exact RGAs

We now recall the method called *ADZ* in [7] and used there in order to design an efficient exact RGA. In the *ADZ version* of the recursive RGA, the preprocessing step is performed using floating-point arithmetic with a (well chosen) mantissa length b ensuring that the absolute difference between any threshold $f_{q,m,r}$ and its approximation $\widetilde{f}_{q,m,r}$ is at most δ. Then, at the generation step, if a random number X falls in one of the error intervals $[\widetilde{f}_{q,m,r} - \delta, \widetilde{f}_{q,m,r} + \delta]$, the preprocessing step is run again but with exact integer representation so as to compare X with the real threshold $f_{q,m,r}$. The same method can be applied to the divide-and-conquer RGA. Observe

Table 2 Bit-complexity of γ-approximate RGAs

RGA:	Recursive	Divide-and-conquer	Divide-and-conquer
Version:		Low-preprocessing	High-preprocessing
Preprocessing space	$\mathbf{q}n \log(n/\gamma)$	$\mathbf{q}^2 \log(n) \log(\mathbf{q}n/\gamma)$	$\mathbf{q}^3 \log(n) \log(\mathbf{q}n/\gamma)$
Preprocessing time	$\mathbf{q}n \log(n/\gamma)$	$\mathbf{q}^3 \log(n) \log(\mathbf{q}n/\gamma)^2$	$\mathbf{q}^3 \log(n) \log(\mathbf{q}n/\gamma)^2$
Generation time	n	$\mathbf{q}n$	$\log(\mathbf{q})n$

Table 3 Complexity of the preprocessing step using floating-point arithmetic with a mantissa of length b

RGA:	Recursive	Divide-and-conquer	Divide-and-conquer
Version:		Low-preprocessing	High-preprocessing
Preprocessing space	$\mathbf{q}bn$	$\mathbf{q}^2 b\log(n)$	$\mathbf{q}^3 b\log(n)$
Preprocessing time	$\mathbf{q}bn$	$\mathbf{q}^3 b^2\log(n)$	$\mathbf{q}^3 b^2\log(n)$

Table 4 Mean bit-complexity of RGAs using the ADZ method

RGA:	Recursive	Divide-and-conquer	Divide-and-conquer
Version:		Low-preprocessing	High-preprocessing
Preprocessing space, worst-case	$\mathbf{q}n^2$	$\mathbf{q}^2 n$	$\mathbf{q}^3 n$
Preprocessing space, mean	$\mathbf{q}n\log(n)$	$\mathbf{q}^2\log(n)\log(\mathbf{q}n)$	$\mathbf{q}^3\log(n)\log(\mathbf{q}n)$
Preprocessing time, mean	$\mathbf{q}n\log(n)$	$\mathbf{q}^3\log(n)\log(\mathbf{q}n)^2$	$\mathbf{q}^3\log(n)\log(\mathbf{q}n)^2$
Generation time, mean	n	$\mathbf{q}n$	$\log(\mathbf{q})n$

that the worst-case complexity of the ADZ version of the RGAs corresponds to the complexity of the RGAs using exact integer representation. However, the mean complexity can be improved by a suitable choice of the length b of the mantissa.

Theorem 7 *The bit-complexity of the exact RGAs using the ADZ floating-point optimization method is given in Table 4.*

Proof By (2), taking a mantissa length $b = \lceil \log_2(8\mathbf{a}^2 n^3)\rceil = O(\log(n))$ ensures that $|\widetilde{f}_{q,m,r} - f_{q,m,r}| \le \delta = \frac{1}{2\mathbf{a}n^2}$ for all $q, q' \in Q$ and all $m \le n$. Since at most $\mathbf{a}n$ comparisons (between some random numbers X and some thresholds $f_{q,m,r}$) are made at the generation step of the recursive RGA, this choice of b gives a probability at most $2\mathbf{a}n\delta = O(1/n)$ of having to make the preprocessing step with exact integer representation. Moreover, the complexity of preprocessing with exact integer representation is $O(\mathbf{a}n^2)$ by Proposition 4. Thus, the mean time-complexity of exact preprocessing is $O(\mathbf{a}n)$. Furthermore, the time-complexity of floating-point preprocessing with mantissa length b is $O(\mathbf{q}n\log(n))$ by Table 3. Thus, the mean time-complexity of the ADZ version of the recursive RGA is $O(\mathbf{a}n + \mathbf{q}n\log(n)) = O(\mathbf{q}n\log(n))$. The mean space-complexity is obtained by a similar argument.

Similarly, by (2), taking a mantissa length $b' = \lceil \log_2(16\mathbf{q}^2 n^4)\rceil = O(\log(qn))$ in the ADZ version of the divide-and-conquer RGA ensures that the probability of having to perform the preprocessing step with exact integer representation is at most $\frac{16\mathbf{q}^2 n^2}{2^b} = O(1/n^2)$. Thus, the mean time-complexity of exact preprocessing is $O(\mathbf{q}^3)$ by Proposition 4, for both the low-preprocessing and high-preprocessing versions. Furthermore, the time-complexity of floating-point preprocessing with a mantissa length b' is $O(\mathbf{q}^3\log(n)\log(\mathbf{q}n)^2)$ by Table 3. Thus, the mean time-complexity of the ADZ version of the divide-and-conquer RGAs is $O(\mathbf{q}^3\log(n)\log(\mathbf{q}n)^2)$. The mean space-complexity is obtained by a similar argument. □

4 Implementation

We have implemented the γ-approximate versions of the recursive RGA and of the low and high-preprocessing divide-and-conquer RGAs. All implementations have been done in C++. Our aim is to compare the efficiency of the different algorithms, in conditions as similar as possible. In what follows we discuss the technical details of our implementations. Then, we provide experimental results comparing their efficiency.

4.1 Floating-point Numbers

We have opted to implement the γ-approximate version of the algorithms instead of the much slower exact arithmetic version. We have chosen arbitrarily $\gamma = 10^{-100}$. This choice of γ requires floating-points numbers with a mantissa size b between 300 and 400 bits, depending on the particular algorithm, the number of states of the automaton and the length of the generated words, as described in (2). Typically, hardware-based floating-point numbers have mantissas of at most 100 bits, so we have used the well-known GMP library (http://gmplib.org) for high-precision floating-point numbers.

4.2 Matrix Product

Our implementation uses standard matrix multiplication to compute the preprocessed matrices $(l_{q,q',2^i})_{q,q'}$ in the low-preprocessing divide-and-conquer RGA, so it is not particularly efficient for large values of $\mathbf{q}$.

4.3 Pseudo-random Numbers

We use the pseudo-random generator provided by the GMP library, which defaults to the Mersenne-Twister algorithm [16]. This algorithm, which was designed with Monte Carlo and other statistical simulations in mind, is known to provide a good balance between fast generation and high-quality pseudo-random numbers.

4.4 Disk Usage

Both algorithms use preprocessed data at the generation step. Reading these data from disk has a significant impact in the running time of the RGA. Our implementations use text-based files and standard C++ streams to store and retrieve preprocessed data. This choice is clearly inefficient, but makes development and debugging easier.

An implementation aiming for efficiency should read and write the numbers in binary form, rather than text form, preferably in the same way the arithmetic library stores the numbers in memory. In this way, we eliminate the parsing overhead, and the file size is reduced by a constant factor $\log_2(256)/\log_2(10) \simeq 2.41$.

4.5 Caching Computations in the Low-preprocessing Divide-and-Conquer RGA

In the recursive RGAs all random choices for the generation of a word are done on *different probabilities* (that is, by comparison of a random number X with *different threshold values*). However, the random choices of the divide-and-conquer RGAs may be done several times on the same probabilities. This occurs when the generated word uses often the same transitions of the automaton. In our implementation we take advantage of this fact by caching the computations required for the random choices of the low-preprocessing divide-and-conquer RGA. Note that the performance gain of this approach depends both on the automaton and the actual word being generated.

4.6 Sequential Access and the Memory Hierarchy

It is well known that algorithms that access data sequentially perform better over those that use random access. The performance hit is enormous when data is stored in disk due to the latency of physical devices. A similar phenomenon, albeit at a smaller scale, occurs when working with RAM memory and cache memory.

From this point of view the recursive RGA has an advantage over the divide-and-conquer RGA described in Definition 2, since the latter generates the word non-sequentially. In fact, the actual letters forming the word are not decided until steps (8)–(9). We can make the divide-and-conquer RGA output the letters sequentially by implementing it recursively (as it is customary with divide-and-conquer algorithms), instead of the given iterative implementation. We have implemented both the iterative and the recursive versions of the divide-and-conquer RGA.

4.7 Parallelization

Our implementations are single processor. When asked to generate $w > 1$ words of the same length, our recursive RGA implementation generates the w words in parallel while reading the preprocessed data from disk, thus effectively amortizing the disk reading time among all the generated words.

The recursive RGA does not appear to be parallelizable when generating a single word. This stays in sharp contrast with the divide-and-conquer RGA which, like all divide-and-conquer algorithms, can be easily and efficiently parallelized into any factor desired, by dividing recursively the generation of a word of length n into two independent generation of words of length $n/2$, with the appropriate initial and final states.

Finally, note that if we parallelize using several processors, all the parallel programs do need to have access to the same preprocessed data. Thus if we do not parallelize using a shared-memory architecture the disk running time would increase proportionally to the number of processors. This also degrades the benefits of caching in the low-preprocessing divide-and-conquer RGA.

4.8 Experimental Results

We show experimental data from our implementations of the γ-approximate RGAs, obtained for different values of n, $\mathbf{q}$ and w. The automata we use in our experiments are randomly generated automata on alphabets of size $\mathbf{a} = 2$, where half of

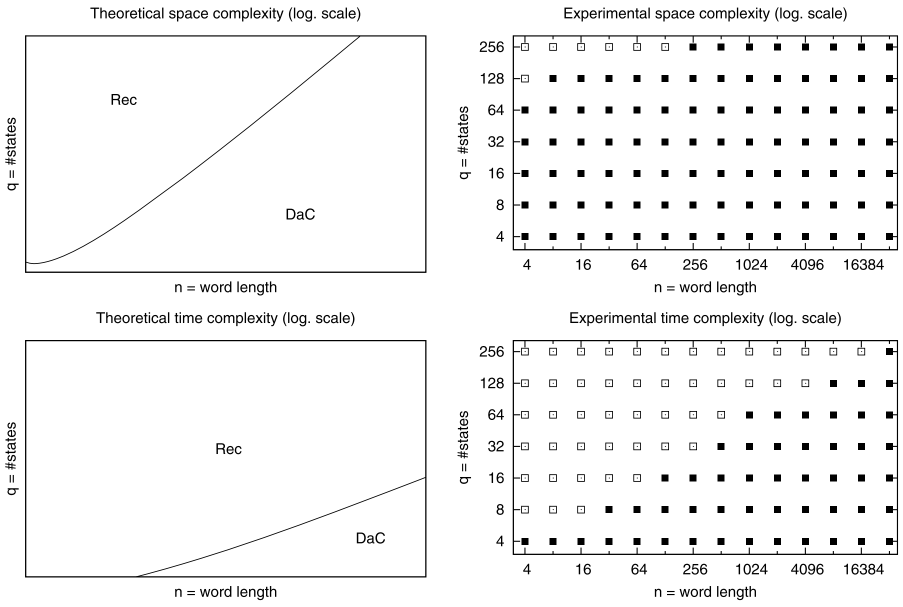

Fig. 1 Theoretical and experimental space-complexity of preprocessing (*top diagrams*) and time-complexity of preprocessing and one word generation (*bottom diagrams*). Here and below the notation *DaC* (resp. *Rec*) stands for the divide-and-conquer RGA (resp. recursive RGA)

the states are accepting states, and transitions between states are chosen uniformly at random. (The same experiments have been run with a family of non-random automata, those recognizing words on the alphabet {a, b} without **q** consecutive a's. The results are completely analogous to those obtained with random automata.) We compare the recursive RGA with the low-preprocessing divide-and-conquer RGA. The performance of the high-preprocessing divide-and-conquer RGA is similar to the low-preprocessing version for small values of **q**, but it quickly degrades for large **q**.

All experiments have been done in a desktop computer with an Intel Core 2 Duo processor and 2 gigabytes of RAM, running the 64-bits version of Ubuntu 8.10. Both implementations have been compiled with `gcc` version 4.3.2 with level 3 optimizations and without debugging support. We obtain the running time of an experiment by running it 5 times, discarding the quickest and the slowest times, and averaging the times of the remaining ones. We remark that, due to the choice of $\gamma = 10^{-100}$, it is highly improbable that the γ-approximate RGAs will fall back to the exact arithmetic implementation during its execution.

We study first which algorithm is best for several combinations of **q** and n, both in terms of space-complexity (size of the files containing the preprocessed data) and time-complexity (time of one word generation, including preprocessing). Figure 1 compares the theoretical expectation with the actual experimental data. The plots on the left show the theoretical border between those situations where the recursive RGA is preferable (large **q** and small n) and those where the divide-and-conquer RGA is preferable (small **q** and small n). They have been obtained by comparing the expressions in Table 2; note that these expressions do not take account of the

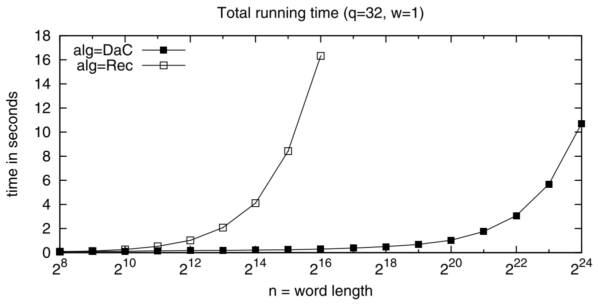

Fig. 2 Total running time (preprocessing and generation) when modifying the length n of the generated words (in logarithmic scale)

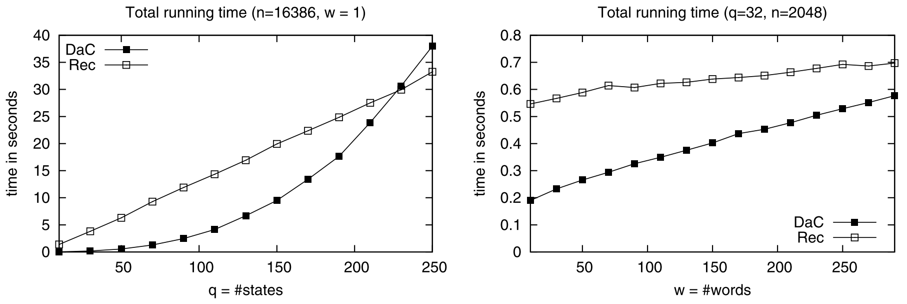

Fig. 3 Total running time (preprocessing and generation) when modifying the number of states $\mathbf{q}$ of the automaton or the number w of words generated in parallel

constant terms. The diagrams on the right show the outcome of our experiments. A white square (resp. a black square) means that the recursive RGA (resp. the divide-and-conquer RGA) performed better in our experiments.

Analyzing Fig. 1, we see that the divide-and-conquer RGA is more space efficient than expected. This is probably due to the way we store the numbers in the preprocessing step: small integer numbers, like those appearing in some of the matrices of the divide-and-conquer RGA preprocessing, are represented using very few bytes. With respect to the time-complexity, the results are roughly as expected.

We next show several plots where we compare the performances when modifying one parameter but fixing the remaining ones. In Fig. 2 we show (on a logarithmic scale) the effect of modifying n, and in Fig. 3 we show the effect of modifying the number $\mathbf{q}$ of states and the number w of generated words. Figure 2 illustrates the fact (expected from theoretical analysis) that the divide-and-conquer RGA is the only valid alternative for generating large words (of length $n = 10^6$ say).

5 Concluding Remarks

The ADZ version of the divide-and-conquer RGA is the first known algorithm to generate words of length n uniformly at random in expected time-complexity $O(n)$. This improves over the previously best know complexity $O(n \log n)$ given by the ADZ version of the recursive RGA. The space requirements of the preprocessing step are also greatly reduced, from $O(n \log n)$ to $O((\log n)^2)$; the worst-case space-complexity is also reduced from $O(n^2)$ to $O(n)$. Moreover, the divide-and-conquer RGA is almost as easy to implement as the recursive RGA and experiments show that it behaves well in practice.

The above analysis is a bit biased by the fact that the complexity of the algorithms depend on the number $\mathbf{q}$ of states of the automaton. In practice, the number $\mathbf{q}$ can be quite large, so that it is more realistic to consider the complexity as a function of both $\mathbf{q}$ and n. In terms of the worst-case space-complexity the divide-and-conquer RGA performs better than the recursive RGA as soon as $\mathbf{q}$ is smaller than $n/\log(n)$. In terms of time-complexity (for generating 1 word), the divide-and-conquer RGA performs better than the recursive RGA if $\mathbf{q}$ is smaller than $\sqrt{n}/\log(n)$.

Finally, let us mention that the divide-and-conquer RGA (as the recursive RGA) can be extended so as to generate words with non-uniform probabilities. Indeed, for any choice of weight on the transitions of the automaton (i.e. on the edge of the corresponding digraph), it is straightforward to modify the RGAs in such a way that any given word is generated with a probability proportional to the product of the weights on the path labelled by this word. This kind of improvement is particularly relevant for bioinformatics applications [6].

Acknowledgements We are very grateful to Philippe Duchon and Alain Denise for fruitful discussions.

References

1. Alonso, L., Schott, R.: Random Generation of Trees: Random Generators in Computer Science. Kluwer Academic, Norwell (1995)
2. Barcucci, E., Del Lungo, A., Pergola, E.: Random generation of trees and other combinatorial objects. Theor. Comp. Sci. **218**, 219–232 (1999)
3. Barcucci, E., Del Lungo, A., Pergola, E., Pinzani, R.: ECO: a general methodology for the enumeration of combinatorial objects. J. Differ. Equ. Appl. **5**, 435–490 (1999)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms, 1st edn. MIT Press, Cambridge (1990)
5. Denise, A.: Génération aléatoire uniforme de mots de langages rationnels. Theor. Comp. Sci. **159**(1), 43–63 (1996)
6. Denise, A., Rocques, O., Termier, M.: Random generation of words of context-free languages according to the frequencies of letters. In: Colloquium on Mathematics and Computer Science, pp. 113–125 (2000)
7. Denise, A., Zimmermann, P.: Uniform random generation of decomposable structures using floating-point arithmetic. Theor. Comp. Sci. **218**(2), 233–248 (1999)
8. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann samplers for the random generation of combinatorial structures. Comb. Probab. Comput. **3**(4–5), 577–625 (2004)
9. Flajolet, P., Zimmermann, P., Van Cutsem, B.: A calculus for the random generation of labelled combinatorial structures. Theor. Comp. Sci. **132**(1), 1–35 (1994)
10. Fusy, E.: Uniform random sampling of planar graphs in linear time. Random Struct. Algorithms **35**(4), 464–522 (2009)
11. Goldwurm, M.: Random generation of words in an algebraic language in linear binary space. Inf. Process. Lett. **54**(4), 229–233 (1995)
12. Hickey, T., Cohen, J.: Uniform random generation of strings in a context-free language. SIAM J. Comput. **12**(4), 645–655 (1983)
13. Kannan, S., Sweedyk, Z., Mahaney, S.: Counting and random generation of strings in regular languages. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 551–557 (1995)

14. Knuth, D.: The Art of Computer Programming. Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1969)
15. Mairson, H.G.: Generating words in a context free language uniformly at random. Inf. Process. Lett. **49**(2), 95–99 (1994)
16. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
17. Nijenhuis, A., Wilf, H.S.: Combinatorial Algorithms. Academic Press, San Diego (1978)