A new dichotomic algorithm for the uniform random generation of words in regular languages

Johan Oudinet[a,b,c], Alain Denise[a,b,d,e], Marie-Claude Gaudel[a,b]

[a] *Univ Paris-Sud, LRI, UMR8623, Orsay, F-91405*
[b] *CNRS, Orsay, F-91405*
[c] *Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany*
[d] *INRIA Saclay - Île-de-France, AMIB project, F-91893 Orsay cedex*
[e] *Univ Paris-Sud, Institut de Génétique et Microbiologie, UMR8621, Orsay, F-91405*

Abstract

We present a new algorithm for generating uniformly at random words of any regular language $\mathcal{L}$. When using floating point arithmetics, its bit-complexity is $\mathcal{O}(q \log^2 n)$ in space and $\mathcal{O}(qn \log^2 n)$ in time, where n stands for the length of the word, and q stands for the number of states of a finite deterministic automaton of $\mathcal{L}$. We implemented the algorithm and compared its behavior to the state-of-the-art algorithms, on a set of large automata from the VLTS benchmark suite. Both theoretical and experimental results show that our algorithm offers an excellent compromise in terms of space and time requirements, compared to the known best alternatives. In particular, it is the only method that can generate long paths in large automata.

Keywords: random generation, regular languages, automata

1. Introduction

The problem of randomly and uniformly generating words from a regular language was first addressed by Hickey and Cohen (1983), as a particular case of context-free languages. Using the so-called recursive method (Wilf, 1977; Flajolet et al., 1994), they gave an algorithm in $\mathcal{O}(qn)$ space and time for the preprocessing stage and $\mathcal{O}(n)$ for the generation, where n denotes the length of the word to be generated, and q denotes the number of states of a deterministic finite automaton of $\mathcal{L}$. Later, Goldwurm (1995) showed that the memory space can be reduced to $\mathcal{O}(q)$, by using a parsimonious approach to keep in memory only a few coefficients. These results are in terms of *arithmetic complexity*, where any number is supposed to take $\mathcal{O}(1)$ space and each basic arithmetic operation takes $\mathcal{O}(1)$ time. As for the bit complexity, the above formulas must

Email addresses: oudinet@kit.edu (Johan Oudinet), denise@lri.fr (Alain Denise), mcg@lri.fr (Marie-Claude Gaudel)

be multiplied by $\mathcal{O}(n)$ due to the exponential growing of the involved coefficients according to n.

If floating point arithmetic is used (Denise and Zimmermann, 1999) for the Hickey and Cohen (1983) algorithm, almost uniform generation can be performed in $\mathcal{O}(qn \log n)$ bit complexity for the preprocessing stage (in time and memory), and $\mathcal{O}(n \log n)$ for the generation. Meanwhile, the parsimonious version of Goldwurm cannot be subject to floating point arithmetic, because of the numerical instability of the involved operations (Oudinet, 2010).

Another technique, the so-called Boltzmann generation method (Duchon et al., 2004), is well fitted for approximate size generation: it makes possible to generate words of size between $(1-\varepsilon)n$ and $(1+\varepsilon)n$, for a fixed value ε, in average linear time according to n (Flajolet et al., 2007). As for exact size generation, the average complexity of generation is in $\mathcal{O}(n^2)$, although it can be lowered to $\mathcal{O}(n)$ if the automaton of $\mathcal{L}$ is strongly connected. Bolztmann generation needs a preprocessing stage whose complexity is in $\mathcal{O}(q^k \log^{k'} n)$, for some constants $k \geq 1$ and $k' \geq 1$ whose precise values are not given in (Duchon et al., 2004) and subsequent papers, to our knowledge.

Recently, Bernardi and Giménez (2010) developed a new divide and conquer approach for generating words of regular languages, based on the recursive method. If using floating point arithmetic, their algorithm runs in $\mathcal{O}(qn \log(qn))$ in the worst case, with a preprocessing in $\mathcal{O}(q^3 \log n \log^2(qn))$ time and $\mathcal{O}(q^2 \log n \log(qn))$ space. Moreover the average complexity of the generation stage can be lowered to $\mathcal{O}(qn)$ if using a bit-by-bit random number generator.

Here we present a new algorithm named *dichopile*, also based on a divide-and-conquer approach, although drastically different from the above one. When using floating point arithmetics, its bit-complexity is $\mathcal{O}(q \log^2 n)$ in space and $\mathcal{O}(qn \log^2 n)$ in time.

The paper is organized as follows. In Section 2 we present the *dichopile* algorithm and we compute its complexity in space and time. Section 3 is devoted to experiments: we compare the running time and space requirements of C++ programs that implement *dichopile* and the other state-of-the-art algorithms, on a set of large automata from the VLTS benchmark suite (Garavel and Descoubes, 2003). Finally we discuss in Section 4 the advantages and drawbacks of each of the algorithms, from both theoretical and experimental points of view. As shown in Table 4, it turns out that, compared to the known best alternatives, our algorithm offers an excellent compromise in terms of space and time requirements.

2. The Dichopile algorithm

At first, let us briefly recall the general principle of the classical recursive method for regular languages. Let us consider a deterministic finite automaton of $\mathcal{L}$ with q states $\{1, 2, \ldots, q\}$. Obviously, there is a one-to-one correspondence between the words of $\mathcal{L}$ and the paths in $\mathcal{A}$ starting at the initial state and

ending at any final state. For each state s, we write $l_s(n)$ for the number of paths of length n starting from s and ending at a terminal state. Such values can be computed with the following recurrences on n (where $\mathcal{F}$ denotes the set of final states in $\mathcal{A}$):

$$\begin{cases} l_s(0) = 1 & \text{if } s \in \mathcal{F} \\ l_s(0) = 0 & \text{if } s \notin \mathcal{F} \\ l_s(i) = \sum_{s \to s'} l_{s'}(i-1) & \forall i > 0 \end{cases} \tag{1}$$

Let $L_n = \langle l_1(n), l_2(n), \ldots, l_q(n) \rangle$. The recursive method proceeds in two steps:

- Compute and store L_k for all $1 \leq k \leq n$. This calculation is done starting from L_0 and using Formula 1.
- Generate a path of length n by choosing each state according to a suitable probability to ensure uniformity among every path of length n. Thus, the probability of choosing the successor s_i when the current state is s and the path has already $n - m$ states is:

$$\mathbb{P}(s_i) = \frac{l_{s_i}(m-1)}{l_s(m)}. \tag{2}$$

Note that in order to choose the successor of the initial state, we only need L_n and L_{n-1}. Then, L_{n-1} and L_{n-2} allow to choose the next state and so on. Thus, if we have a method that compute efficiently $L_n, L_{n-1}, \ldots, L_0$ in descending order, we can store the two last vectors only and reduce space complexity compared to recursive method, which stores all L_k's in memory. This *inverse* approach constitutes the principle of Goldwurm (1995)'s method. In (Oudinet, 2010), the inverse recurrence is stated for regular languages, and it is shown that the algorithm is numerically instable, thus forbidding the use of floating-point arithmetics.

2.1. General principle

The idea of our *dichopile* algorithm is as follows. Compute the number of paths of length n from the number of paths of length 0 while saving in a stack a logarithmic number of intermediate steps: the number of paths of length $n/2$, of length $3n/4$, of length $7n/8$, etc. When we need to compute the number of paths of length $n-i$, we compute it again from the intermediate stage that is at the top of the stack. Figure 1 illustrates the principle of this algorithm. Recall that L_j denotes the vector of q numbers of paths of length j, that is the $l_s(j)$'s for all states s.

Algorithm 1 draws uniformly at random a path of length n by successively computing the numbers of paths of length i in descending order. This algorithm takes as inputs:

- a deterministic finite automaton $\mathcal{A}$;

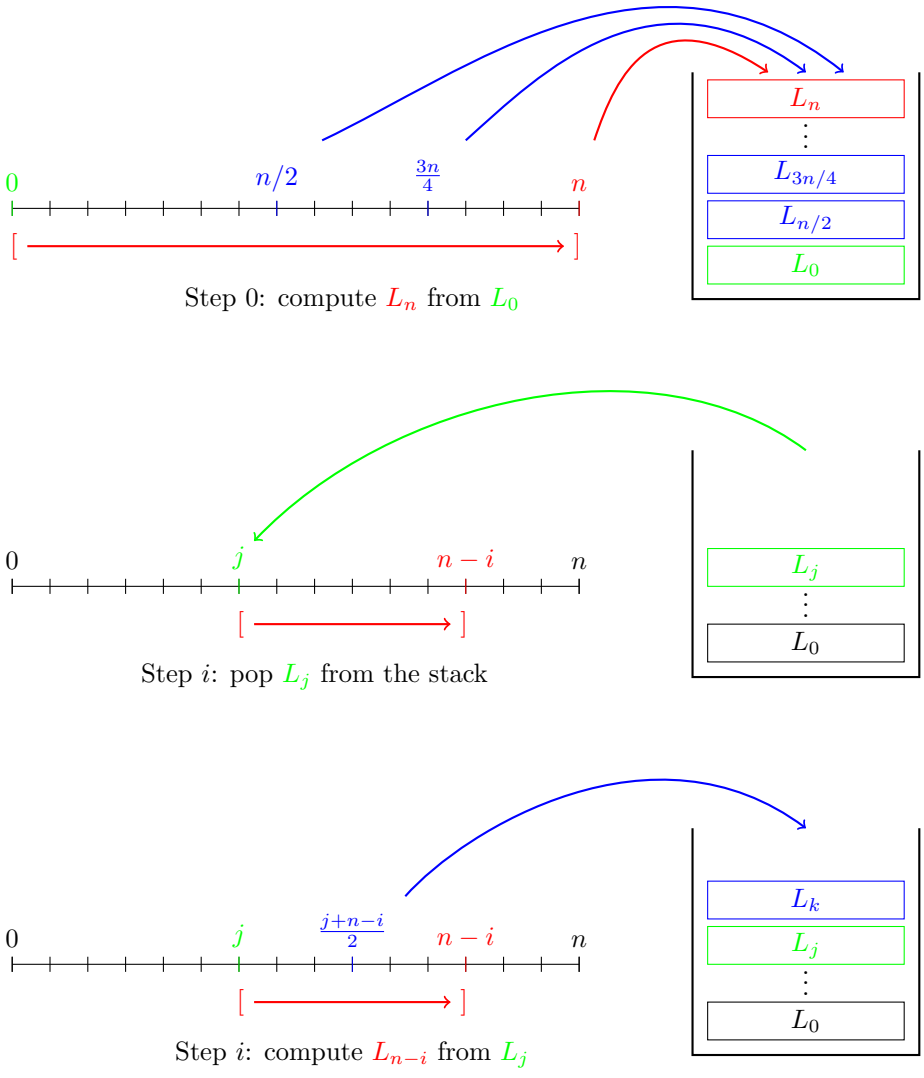

Figure 1: Principle of the *dichopile* algorithm.

Algorithm 1 Draw a path of length n with the *dichopile* algorithm.

Require: an automaton $\mathcal{A}$, a vector L_0, a length n, and a function F that computes L_j from L_{j-1}
Ensure: returns a path σ of length n.
  $s \leftarrow s_0$ {initialize s to the initial state}
  push $(0, L_0)$
  for $i \leftarrow 0$ to n **do** {Iteration for computing L_{n-i}}
    $(j, L\_cur) \leftarrow$ top of the stack
    if $j > n - i$ **then** {useless value on the stack, get the next one}
      pop from the stack
      $(j, L\_cur) \leftarrow$ top of the stack
    end if
    while $j < n - i - 1$ **do** {compute L_{n-i} from L_j}
      $k \leftarrow \frac{j+n-i}{2}$
      for $m \leftarrow j + 1$ to k **do**
        $L\_cur \leftarrow F(L\_cur)$
      end for
      push (k, L_cur) {push L_k to the stack}
      $j \leftarrow k$
    end while
    if $j = n - i - 1$ **then**
      $L\_cur \leftarrow F(L\_cur)$
    end if
    if $i > 0$ **then** {wait until L_suc is defined to have $L\_cur = L_{n-i}$ and $L\_suc = L_{n-i+1}$}
      choose the next transition t_i in $\mathcal{A}$ according to s, L_cur and L_suc
      $\sigma \leftarrow \sigma.t_i$ {concatenation of the transition}
      $s \leftarrow$ the extremity of t_i
    end if
    $L\_suc \leftarrow L\_cur$
  end for
  return σ

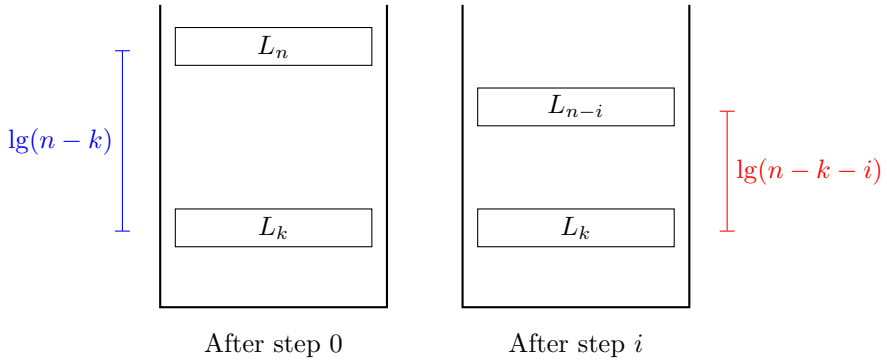

Figure 2: The stack size is maximal after the first iteration of the dichopile algorithm.

- a vector L_0 of q numbers defined as in the first two items of Formula 1;
- the path length n;
- and a function F that computes L_j from L_{j-1}, thus F computes the $l_s(j)$'s for all $s \in \mathcal{S}$ using the third item of Formula 1.

It uses a stack and two local variables *L_cur* and *L_suc*, which are vectors of q numbers. The vector *L_suc* saves the previous value of *L_cur*.
Step $i = 0$ computes L_n from L_0, pushing to the stack the vectors $L_{n/2}$, $L_{3n/4}$, $L_{7n/8}$, etc. All the divisions are integer divisions.
Step i seeks to compute L_{n-i}. For that, it starts by retrieving the top of the stack L_j (if $j > n - i$ then it takes the next item on the stack) and computes L_{n-i} from L_j, pushing to the stack a logarithmic number of intermediate vectors L_k.
At the end of each iteration of the main loop and just before updating *L_suc*, we have *L_cur* $= L_{n-i}$ and *L_suc* $= L_{n-i+1}$. Then, using Formula 2, we can draw the successor state according to the current state and values contained in these two vectors.

2.2. Complexity analysis

Theorem 1. *Using floating-point numbers with a mantissa of size $\mathcal{O}(\log n)$, bit complexities of Algorithm 1 are $\mathcal{O}(q \log^2 n)$ in space and $\mathcal{O}(dqn \log^2 n)$ in time, where d stands for the maximal out-degree of the automaton.*

Proof. Unlike the classical recursive method, there is no preprocessing phase. Values cannot be saved between two path generations because the content of the stack changes during the drawing.

The space complexity depends on the stack size. After the first iteration, there are $\lg n$ elements on the stack, and there will never be more elements on the stack in subsequent iterations: if L_k is on top of the stack at the i-th iteration, there were $\lg(n-k)$ elements above L_k after the first iteration and there will

be $\lg(n-k-i)$ elements after the i-th iteration; Hence, fewer elements. This property is illustrated in Figure 2. Each stack element contains an integer ($\leq n$) and q path numbers represented by floating-point numbers with a mantissa of size $\mathcal{O}(\log n)$, i.e. $\mathcal{O}(q \log n)$ bits per item. Hence the size occupied by the stack is $\mathcal{O}(q \log^2 n)$ bits.

The time complexity depends on the number of calls to function F and this number depends on the difference between the last stacked value (j) and the one to compute ($n-i$). To calculate this complexity, we refer to a diagram describing the successive calculations done by the algorithm using two functions, f and g, which call each other. The calculation scheme is shown in Figure 3.

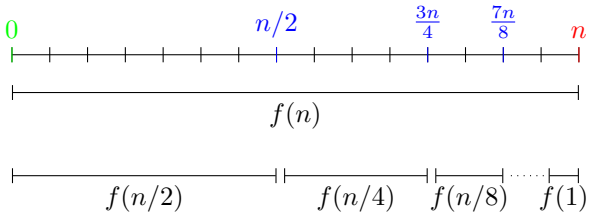

Figure 3: Recursive scheme of the number of calls to function F done by Algorithm 1. We omit floor and ceiling notations to clarify the figure.

The function f counts the number of calls to function F. For $f(n)$, it will first call F n times. Then, since the first saved value is $\lfloor n/2 \rfloor$, it will call $f(\lfloor n/2 \rfloor - 1)$. The next saved value is $\lfloor 3n/4 \rfloor$, hence the call to $f(\lfloor n/4 \rfloor - 1)$, and so on.

Thus, the number of calls to F to generate a path of length n is equal to $f(n)$, which is defined as:

$$f(1) = 1$$
$$f(n) = n + \sum_{i=1}^{\lfloor \lg n \rfloor} f(\lfloor \frac{n}{2^i} \rfloor - 1)$$

As the number of terms in the sum is not a constant, we cannot apply directly the Master theorem nor the Akra-Bazzi theorem(Akra and Bazzi, 1998). However, their theorem 1 does apply to the generalized case where the number of terms in the sum, k, is a variable of n. This theorem allows us to ignore both the floor function and the minus 1 in the index of f since the same asymptotic behavior is kept. Thus, we study the asymptotic behavior of T defined as follows:

$$T(n) = n + \sum_{i=1}^{\lfloor \lg n \rfloor} T(\frac{n}{2^i})$$

Lemma 2. *The function T has the following recurrence form:*

$$T(n) = 2T(n/2) + n/2 \quad \forall n \geq 2$$

Proof.

$$
\begin{aligned}
T(2) &= 2 + T(1) = 3 = 2T(1) + 1 \\
T(2n) &= 2n + \sum_{i=1}^{\lfloor \lg 2n \rfloor} T(\frac{2n}{2^i}) \\
&= 2n + \sum_{i=1}^{\lfloor 1+\lg n \rfloor} T(\frac{n}{2^{i-1}}) \\
&= 2n + \sum_{i=0}^{\lfloor \lg n \rfloor} T(\frac{n}{2^i}) \\
&= n + \left(n + \sum_{i=1}^{\lfloor \lg n \rfloor} T(\frac{n}{2^i}) \right) + T(n) \\
&= 2T(n) + n
\end{aligned}
$$

□

Thanks to Lemma 2, we can apply the master theorem (Cormen et al., 2001, sect. 4.3) on T. As $n/2 \in \Theta(n)$, we are in the second case of the master theorem, which states:

$$T(n) \in \Theta(n \log n)$$

Hence a time complexity of $f(n)$ in $\Theta(n \log n)$ calls to function F. The cost of a call to the function F is in $\mathcal{O}(dq \log n)$ because it corresponds to compute $l_s(i)$ from L_{i-1} for all $s \in \mathcal{S}$, using a floating-point arithmetic with numbers of $\mathcal{O}(\log n)$ bits. Thus, a bit complexity of $\mathcal{O}(dqn \log^2 n)$ in time. □

3. Experimental results

In this section, we present the experiments we did to measure and compare the relative efficiency of Dichopile with regards to the other algorithms for the uniform random generation of words in regular languages. We first describe the implementation and the methodology used to conduct those experiments in Section 3.1. Then, we present an explanation together with empirical evidence of the numerical instability of Goldwurm's method in Section 3.2. As written previously the proof is given in details in (Oudinet, 2010), but we also briefly present it here for the sake of completeness. Finally, we draw cross views of the time and the memory used for each algorithm (except Boltzmann) to generate paths in various automata, from 289 states to more than 10^7 states.

3.1. Implementation and methodology

All the algorithms for the uniform random generation of words in regular languages, but Boltzmann, are implemented and freely available in our free

Name	# states	# transitions	# labels	Branching factor avg [min – max]
vasy_0_1	289	1224	2	4.24 [4 – 8]
vasy_1_4	1183	4464	6	3.77 [2 – 5]
vasy_5_9	5486	9676	31	1.76 [0 – 6]
vasy_10_56	10849	56156	12	5.18 [4 – 6]
vasy_12323_27667	12323703	27667803	119	2.25 [0 – 13]

Table 1: Detailed description of the VLTS benchmark suite.

C++ library: Rukia[1]. This library is based on several other libraries: the BGL (Boost Graph Library) to manipulate graphs (Siek et al., 2000), the GMP (Gnu Multiple Precision) library (Granlund, 1991) and the Boost Random library (Maurer and Watanabe, 2000) in order to generate random numbers. Note that Goldwurm's method should not use a floating-point arithmetic because of its numerical instability (see Section 3.2 for an in-depth explanation). Unfortunately we could not implement the Boltzmann method: it needs to precompute a numerical parameter whose value depends on the generating series of the language. And computing this value for huge automata is still an unsolved problem, up to our knowledge.

In order to evaluate those algorithms, we have selected a pool of automata from the VLTS (Very Large Transition Systems (Garavel and Descoubes, 2003)) benchmark suite. These automata correspond to real industrial systems. Each name of an automaton writes *vasy_X_Y*, where X is the number of states divided by 1000, and Y is the number of transitions divided by 1000. The general properties of these automata can be found in Table 1. The number of paths, in those automata, grows exponentially according to the path length. Thus, there are for example more than 10^{479} paths of length 1000 in vasy_1_4 automaton.

We did all our experiments on a virtual machine on a dedicated server whose hardware is composed of an Intel Xeon 3GHz processor with 32GiB memory. An experiment was to measure the amount of memory used and the elapsed time by an algorithm to generate a path of a certain length in an automaton. Each experiment was run 5 times and only the average value is presented (The standard deviation is always lower than 5% of the average value, which means it is not necessary to repeat the experiment more than 5 times). In case where the path generator was too fast to be measured by the timer (less than 10 milliseconds), 100 paths were generated and the average time was divided by 100.

3.2. Numerical instability of Goldwurm's method

Denise and Zimmermann (1999) proved that the operations used by the recursive method are numerically stable. We experimentally checked that it is also true for our implementation of the recursive method: Figure 4 shows

[1] http://rukia.lri.fr

that the maximal relative error is less than 10^{-17}; hence, our implementation is numerically stable. The relative error is computed as follows:

$$Err = \max_{\substack{s \in S \\ 0<i\leq n}} \frac{|l_s(i) - \widetilde{l_s}(i)|}{l_s(i)} \tag{3}$$

where $\widetilde{l_s}(i)$ (resp. $l_s(i)$) is the number of paths of length i starting from the state s and computed with a floating-point (resp. exact) arithmetic.

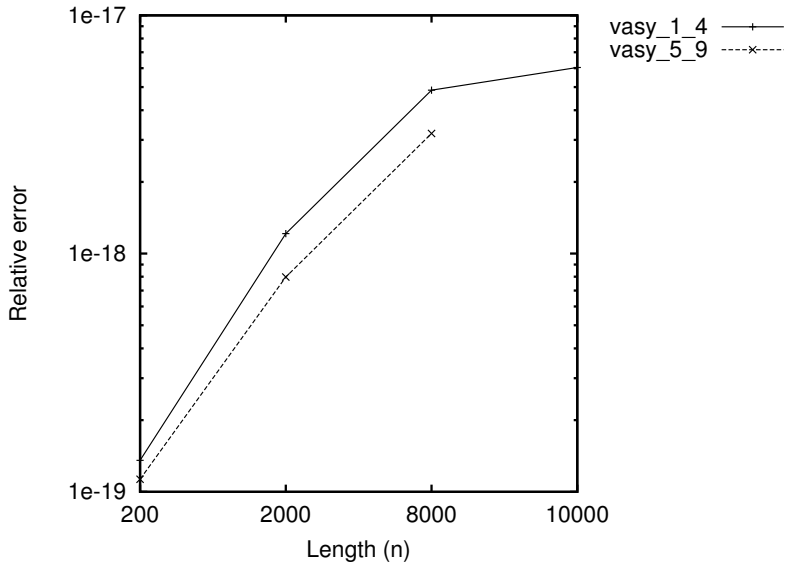

Figure 4: Measurement on two automata of the maximum relative error for the recursive method, defined by Equation (3).

On the other hand, the extra operations performed by Goldwurm's method are numerically unstable as illustrated by the maximal relative error computed on our implementation of this method (see Figure 5). Such extra operations for rational languages correspond to the computation of the following formulae (see (Oudinet, 2010) for details):

$$l_s(i) = \sum_{s'} r_{s,s'} \times l'_s(i+1) \quad \text{with } r_{s,s'} \in \mathbb{Q}$$

If ε_i^s is the error of $l_s(i)$, then the error propagation is as follows:

$$\begin{aligned}
\varepsilon_{n-1}^s &= \sum_t \varepsilon_n^t \times |r_{st}| \\
\varepsilon_n &= \max_s \varepsilon_n^s \\
\varepsilon_{n-1} &\leq \max_s \sum_t \varepsilon_n |r_{st}| \\
\varepsilon_{n-1} &\leq \left[\max_s \sum_t |r_{st}| \right] \varepsilon_n
\end{aligned}$$

Thus, the error propagation follows a geometric distribution with parameter $\max_{s,s'} |r_{s,s'}|$. In other word, Goldwurm's method is numerically unstable as soon as there is a $|r_{s,s'}|$ greater than 1, which is the case in the automata studied.

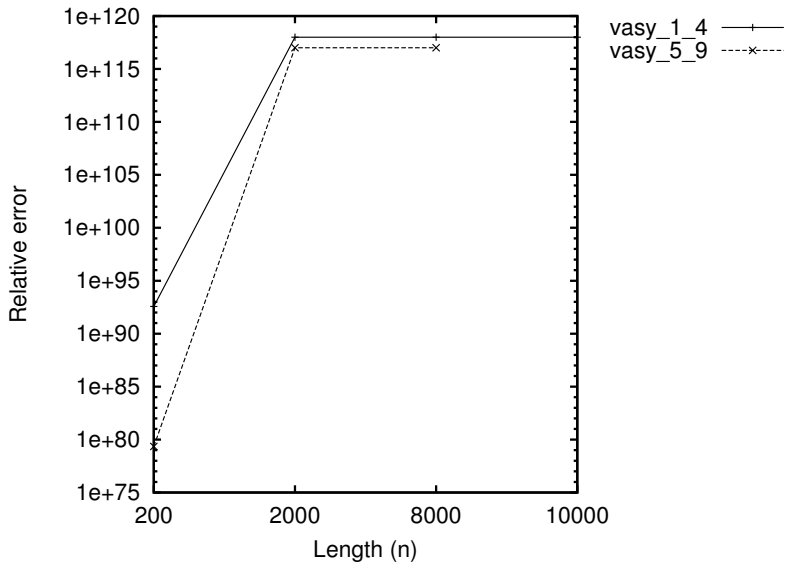

Figure 5: Measurement on two automata for the Goldwurm's method of the maximum relative error, defined by Equation (3).

3.3. Memory requirement

Here we present the memory consumption for the three methods that use floating-point arithmetic: the recursive method (Denise and Zimmermann, 1999), the Divide and Conquer method (Bernardi and Giménez, 2010), and our Dichopile algorith. (We also conducted the same experiments using exact arithmetic, those results are presented in Appendix A.)

		Memory (MiB)		
Automaton	Length	Rec	DaC	Dichopile
vasy_0_1	1000	33	67	**16**
	2000	49	71	**16**
	4000	79	76	**16**
	8000	141	80	**16**
	16000	265	85	**16**
	32000	512	90	**16**
	64000	1005	95	**17**
	128000	1993	101	**18**
vasy_1_4	1000	81	840	**17**
	2000	144	915	**17**
	4000	271	990	**17**
	8000	523	1064	**17**
	16000	1029	1139	**17**
	32000	2040	1215	**18**
	64000	4061	1290	**18**
	128000	8105	1366	**19**
vasy_5_9	1000	312	-	**21**
	2000	605	✘	**21**
	4000	1191	✘	**22**
	8000	2363	✘	**22**
	16000	4707	✘	**23**
	32000	9395	✘	**23**
	64000	18770	✘	**24**
	128000	✘	✘	**25**
vasy_10_56	1000	601	✘	**29**
	2000	1181	✘	**29**
	4000	2340	✘	**30**
	8000	4657	✘	**31**
	16000	9293	✘	**31**
	32000	✘	✘	**32**
	64000	✘	✘	**33**
	128000	✘	✘	**35**
vasy_12323_27667	1000	✘	✘	**13111**
	2000	✘	✘	**13769**
	4000	✘	✘	**14427**
	8000	✘	✘	-

Table 2: Comparison of the memory requirement to generate paths in various automata by: the recursive method (Rec) using floating-point arithmetic, the divide and conquer method (DaC), and the Dichopile method. The symbol ✘ means that the method needs more than 32GiB of memory, and the symbol - means the value is unavailable due to a timeout during the preprocessing

Table 2 shows the results of comparing the memory requirement for the three methods, when generating a path of a defined length in various automata. Even though the space complexity of Bernardi and Giménez's divide-and-conquer (DaC) method is close to the space complexity of the recursive method in terms of path length, the difference of space consumption for those two methods is huge due to the size of automata considered here. In consequence, the divide-and-conquer method could not be used with automata of 5-thousand states and more. Dichopile performed much better than the two other methods and was the only one capable of generating paths in an automaton of more than 10^7 states.

3.4. Execution time

In addition to the measurement of space consumption, we also measured elapsed time for the three same methods in order to generate a path in various automata. Results for the other methods are provided in Appendix A.

Table 3 shows elapsed time for the three methods. The preprocessing time (Pre), which has to be done only once whatever the number of paths generated, is separated from the running time (Draw)[2]. For example, the elapsed time necessary to generate 1000 paths with one method is equal to $\texttt{Pre} + 1000 \times \texttt{Draw}$.

Figure 6 shows memory consumption and elapsed time to generate 1000 paths for each method. In those experiments, the recursive method was faster than the two other methods but Dichopile was capable of generating longer paths in larger automata. Again, even though the time complexity of divide-and-conquer is the best with regards to the path length, the automata considered in our experiments are too large to see the advantage of divide-and-conquer.

Finally, the recursive method is fast but does not scale up very well (both in terms of long path or large automaton). The divide-and-conquer algorithm is well-suited for very long paths but for small automata only. Thus, the Dichopile algorithm is an efficient algorithm to generate long paths in large automata.

[2]The stack used by Dichopile has to be computed again from scratch in order to generate a new path. Thus, there is no preprocessing time for this method.

		Time (s)					
		Rec		DaC		Dichopile	
Automaton	Length	Pre	Draw	Pre	Draw	Pre	Draw
vasy_0_1	1000	**0.1**	**0.001**	48	0.04	0	0.2
	2000	**0.2**	**0.002**	53	0.08	0	0.5
	4000	**0.3**	**0.004**	60	0.16	0	1.0
	8000	**0.6**	**0.008**	65	0.32	0	2.2
	16000	**1.3**	**0.016**	69	0.64	0	4.8
	32000	**2.6**	**0.032**	72	1.3	0	10
	64000	**5.2**	**0.063**	78	2.5	0	22
	128000	**10**	**0.13**	84	5.1	0	46
vasy_1_4	1000	**0.3**	**0.001**	3442	0.4	0	0.8
	2000	**0.6**	**0.002**	3839	0.8	0	1.9
	4000	**1.3**	**0.004**	4222	1.6	0	3.9
	8000	**2.6**	**0.009**	4540	3.4	0	8.4
	16000	**5.1**	**0.019**	4939	7	0	18
	32000	**10**	**0.038**	5279	13	0	38
	64000	**21**	**0.08**	5638	28	0	80
	128000	**43**	**0.15**	5907	56	0	169
vasy_5_9	1000	**1.4**	**0.001**	∞		0	2.5
	2000	**3.1**	**0.002**	-		0	5.2
	4000	**6.2**	**0.003**	-		0	11
	8000	**13**	**0.007**	-		0	23
	16000	**27**	**0.015**	-		0	48
	32000	**52**	**0.03**	-		0	100
	64000	**107**	**0.06**	-		0	208
	128000	-	-	-		0	**432**
vasy_10_56	1000	**7**	**0.001**	-		0	11
	2000	**18**	**0.001**	-		0	24
	4000	**41**	**0.003**	-		0	52
	8000	**81**	**0.008**	-		0	111
	16000	**165**	**0.016**	-		0	236
	32000	**333**	**0.032**	-		0	500
	64000	-		-		0	**1058**
	128000	-		-		0	**2222**
vasy_12323_27667	1000	-		-		0	**10033**
	2000	-		-		0	**21047**
	4000	-		-		0	**44053**
	8000	-		-		0	∞

Table 3: Comparison of the execution time to generate paths in various automata by: the recursive method (Rec) using floating-point arithmetic, the divide and conquer method (DaC), and the Dichopile method. The symbol ∞ means that the generation was not complete after 24 hours of execution, and the symbol - means the value is unavailable due to a memory overflow during the preprocessing.

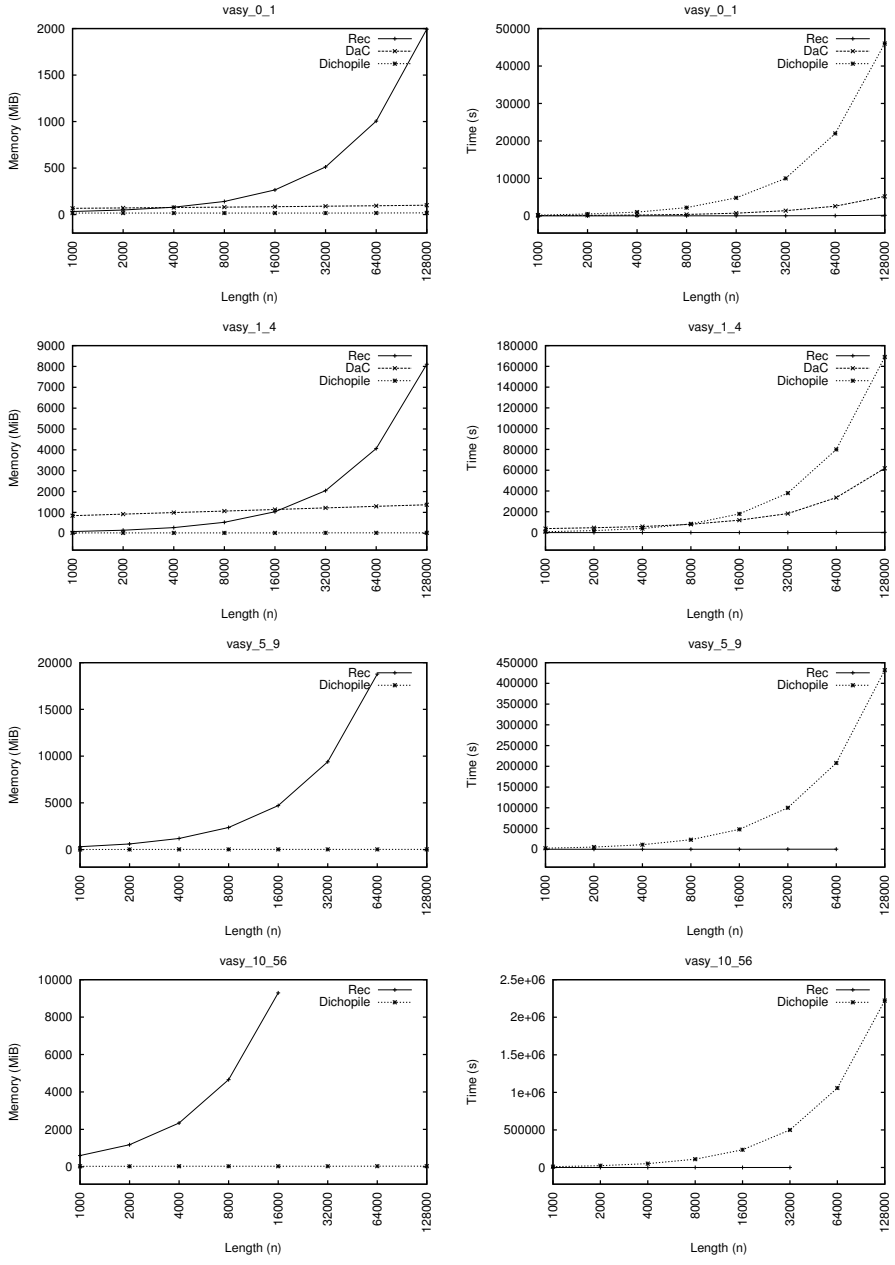

Figure 6: Cross view of the results presented in Table 2 and Table 3, assuming 1000 paths are generated. Thus, each time value corresponds to `Pre` $+ 1000 \times$ `Draw`.

4. Conclusion

Table 4 summarizes the bit complexities in time and in space of the known algorithms to generate random words in regular languages. Since our goal is to be able to explore at random very large models, we are interested in the complexity in terms of both the path length n and the automaton size q. For the sake of clarity, we consider as constants the following values: the maximum degree d of the automaton and the mantissa size b chosen for the floating-point numbers. The inverse method using floating-point arithmetic is crossed out since it can not be used due to its numerical instability.

Table 4: Summary of the binary complexities in time and in space according to the method used. We consider the path length n and the number of states q in the automaton.

			Time	
Method	Arith	Space	Preprocessing	Generation
recursive	exact	$\mathcal{O}(qn^2)$	$\mathcal{O}(qn^2)$	$\mathcal{O}(n^2)$
inverse	exact	$\mathcal{O}(qn)$	$\mathcal{O}(q^2 + qn^2)$	$\mathcal{O}(qn^2)$
recursive	float	$\mathcal{O}(qn \log n)$	$\mathcal{O}(qn \log n)$	$\mathcal{O}(n \log n)$
~~inverse~~	~~float~~	~~$\mathcal{O}(q \log n)$~~	~~$\mathcal{O}(q^2 + qn \log n)$~~	~~$\mathcal{O}(qn \log n)$~~
divide and conquer	float	$\mathcal{O}(q^2 \log n \log(qn))$	$\mathcal{O}(q^3 \log n \log^2(qn))$	$\mathcal{O}(qn \log(qn))$
Boltzmann	float	$\mathcal{O}(q)$	$\mathcal{O}(q^k \log^{k'} n)$	$\mathcal{O}(n^2)$
dichopile	float	$\mathcal{O}(q \log^2 n)$	$\mathcal{O}(1)$	$\mathcal{O}(qn \log^2 n)$

From the point of view of space complexity, obviously the best algorithm is the Boltzmann method with its $O(q)$ complexity. The main limitation of the recursive method is the space needed to store the counting table. Even when using floating-point arithmetic, the space complexity is still $\mathcal{O}(qn \log n)$, which becomes very problematic for large n and q. The inverse method has similar problems, with its $\mathcal{O}(qn)$ complexity. Both *divide and conquer* and *dichopile* perform well due to their polylogarithmic complexity in n, but *dichopile* uses more than q times less memory than *divide and conquer* (up to a constant factor). This is illustrated by the experimental results (Table 2) where *dichopile* can manage huge automata while *divide and conquer* cannot. On the other hand, for small automata and long paths, *divide and conquer* uses much less memory than the recursive method, as expected.

If for any reason, the tiny difference from the uniformity induced by the use of a floating-point arithmetic is not acceptable, the *inverse* method can be used as it offers the best space complexity, but it requires a long generation time (experimental results for this method can be seen in Appendix A.2). Exactly uniform generation can also be done by combining floating-point arithmetic and exact arithmetic (Denise and Zimmermann, 1999), but the space complexity becomes larger than for the quasi-uniform generation shown here. For example, the bit space complexity becomes $\mathcal{O}(qn^2)$ for the recursive method and $\mathcal{O}(q^2 n)$ for the divide-and-conquer algorithm.

Regarding the time complexity only, the best algorithm is the classical recursive scheme with its $\mathcal{O}(n \log n)$ complexity in floating point arithmetic. As for Boltzmann, the rejection procedure necessary to obtain paths of length n raises the time complexity to $\mathcal{O}(n^2)$ in the general case. Both *divide and conquer* and *dichopile* are linear in q and almost linear in n, with an advantage for *divide and conquer*. Regarding the experiments (Table 3), this hierarchy is respected: when *divide and conquer* has enough memory for running, it runs faster than *dichopile*; and both are outperformed by the recursive method. Note that, as stated by Bernardi and Giménez (2010), faster versions of these three algorithms could be implemented, by using a bit-by-bit random number generator.

Altogether, the classical recursive method is fast (after the preprocessing stage), but is unusable for large n and q due to its huge space requirement. On the other hand, the Boltzmann method needs little memory but is slow[3] according to n. Unfortunately, as written in Section 3, we could not experiment the Boltzmann method because computing the value of its parameter for large automata is still an unsolved problem, up to our knowledge. Both *divide and conquer* and *dichopile* are excellent compromises. From a theoretical point of view, the latter offers a better space complexity, and comparatively the increase of time complexity according to n is quite acceptable. Experiments confirm that *divide and conquer* is well fitted for small automata and long words: it needs much less memory than the recursive method and the time for generating a word remains very reasonable. On the other hand, *dichopile* is the only method that can generate long paths in large automata.

Acknowledgement

We warmly thank Carine Pivoteau for helpful discussions on Boltzmann generation.

References

Akra, M., Bazzi, L., May 1998. On the solution of linear recurrence equations. Computational Optimization and Applications 10 (2), 195–210. 7

Bernardi, O., Giménez, O., 2010. A linear algorithm for the random sampling from regular languages. Algorithmica, 1–16. 2, 11, 13, 17

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., 2001. Introduction to Algorithms, 2nd Edition. MIT Press and McGraw-Hill. 8

Denise, A., Zimmermann, P., 1999. Uniform random generation of decomposable structures using floating-point arithmetic. Theoretical Computer Science 218, 233–248. 2, 9, 11, 16

[3]Meanwhile, if an error margin is tolerated on the path length, that is if it suffices to generate paths whose length lay in the interval $[(1-\varepsilon)n, (1+\varepsilon)n]$ for a fixed $\varepsilon > 0$, then the time complexity of the Boltzmann method is $\mathcal{O}(n)$.

Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G., 2004. Boltzmann samplers for the random generation of combinatorial structures. Combinatorics, Probability and Computing 13 (4–5), 577–625, special issue on Analysis of Algorithms. 2

Flajolet, P., Fusy, E., Pivoteau., C., 2007. Boltzmann sampling of unlabelled structures. In: Proceedings of the 4th Workshop on Analytic Algorithms and Combinatorics, ANALCO'07 (New Orleans). SIAM, pp. 201–211. 2

Flajolet, P., Zimmermann, P., Cutsem, B. V., 1994. A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science 132, 1–35. 1

Garavel, H., Descoubes, N., July 2003. Very large transition systems. http://tinyurl.com/yuroxx. 2, 9

Goldwurm, M., 1995. Random generation of words in an algebraic language in linear binary space. Information Processing Letters 54 (4), 229–233. 1, 2, 3, 8, 9, 10, 11, 19, 21

Granlund, T., 1991. GNU MP: The GNU Multiple Precision Arithmetic Library. http://gmplib.org/manual/. 9

Hickey, T., Cohen, J., 1983. Uniform random generation of strings in a context-free language. SIAM J. Comput. 12 (4), 645–655. 1, 2, 19

Maurer, J., Watanabe, S., June 2000. Boost random number library. http://www.boost.org/libs/random/. 9

Oudinet, J., June 2010. Random exploration of models. Tech. Rep. 1534, LRI, Université Paris-Sud XI, 15 pages. Submitted. 2, 3, 8, 10

Siek, J., Lee, L.-Q., Lumsdaine, A., June 2000. Boost graph library. http://www.boost.org/libs/graph/. 9

Wilf, H. S., 1977. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. Advances in Mathematics 24, 281–291. 1

Appendix A. Experimental results using exact arithmetic

Beside results presented in Section 3, we also conducted experiments using exact arithmetic for the recursive method and the Goldwurm's method. Comparing performances of those methods with the ones using floating-point arithmetic is not useful because methods with floating-point arithmetic are faster and need less memory than their counterparts using exact arithmetic. For the sake of completeness, we present here results for the recursive method then for the Goldwurm's method using exact arithmetic.

Appendix A.1. The recursive method using exact arithmetic

In this section, we measured the performance of the original algorithm from Hickey and Cohen (1983). Its bit complexity is in $\mathcal{O}(qn^2)$ space and time for the preprocessing stage and $\mathcal{O}(n^2)$ for the generation, where n denotes the length of the path to be generated, and q denotes the number of states of a deterministic finite automaton.

Table A.5 shows the memory usage and the elapsed time to generate a path depending on the length of the path and the size of the automaton. We note that the generation is very fast but the space needed to store the counting table is very important and thus limits the length of paths that can be drawn.

Automaton	Length	Mem (MiB)	Pre (s)	Draw (s)
vasy_0_1	1000	63	0.2	0.001
	2000	178	0.6	0.004
	4000	614	2.1	0.013
	8000	2312	7.5	0.043
	16000	9015	29	0.14
	32000	✘	-	
vasy_1_4	1000	177	0.7	0.002
	2000	560	2.1	0.005
	4000	1996	6.9	0.013
	8000	7555	25	0.04
	16000	29423	93	0.13
	32000	✘	-	
vasy_5_9	1000	465	2.6	0.001
	2000	1339	7.1	0.004
	4000	4378	20	0.009
	8000	15634	55	0.024
	16000	✘	-	
vasy_10_56	1000	1961	16	0.003
	2000	6888	46	0.006
	4000	25753	147	0.019
	8000	✘	-	
vasy_12323_27667	1000	✘	-	

Table A.5: Generation of a path of length n by the recursive method using exact arithmetic. The symbol ✘ means there is not enough memory to build the counting table, and the symbol - means the value is unavailable due to a memory overflow during the preprocessing.

Appendix A.2. The inverse recursive method using exact arithmetic

In this section, the parsimonious version of Goldwurm is used to keep in memory a few coefficients only. As this version is numerically unstable, floating-point arithmetic cannot be used. The space complexity is in $\mathcal{O}(qn)$ bits. Generation of a path of length n is done in $\mathcal{O}(qn^2)$, after a pre-processing time in $\mathcal{O}(q^2 + qn^2)$.

Table A.6 shows memory usage and elapsed time to generate a path in the automata vasy_0_1, vasy_1_4 and vasy_5_9. Path length varies from 1000 to $128,000$. The computation of the values needed for reversing the recurrences was not possible for the other automata, due to insufficient memory resources.

Thanks to reducing space complexity, it is possible to generate much longer paths. Thus, a path of length $64,000$ was generated with this method in the automaton vasy_5_9. While with the recursive method using exact arithmetic, we could not get paths of length $16,000$ in the same automaton. However, the generation time is much longer: almost 10 minutes for a path of length $8,000$ while the same path can be generated in 24 milliseconds by the recursive method.

Automaton	Length	Mem (MiB)	Pre (s)	Draw (s)
vasy_0_1	1000	19	0.1	12
	2000	20	0.3	29
	4000	20	1.2	83
	8000	20	4.7	269
	16000	22	18	926
	32000	25	77	3426
	64000	31	342	13296
	128000	43	1462	52413
vasy_1_4	1000	44	0.3	301
	2000	44	1.0	730
	4000	45	3.6	1933
	8000	49	14	5881
	16000	58	63	19559
	32000	61	266	70664
	64000	110	1079	∞
	128000	180	4351	∞
vasy_5_9	1000	31	0.4	36
	2000	33	1.0	83
	4000	35	3.8	210
	8000	40	19	579
	16000	49	79	1798
	32000	69	312	6104
	64000	108	1208	22145
	128000	298	4680	∞

Table A.6: Generation of a path of length n by the inverse method using exact arithmetic. The symbol ∞ means that the generation was not complete after 24 hours of execution.