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

Johan Oudinet[1,2], Alain Denise[1,2,3], and Marie-Claude Gaudel[1,2]

[1] Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405;
[2] CNRS, Orsay, F-91405;
[3] INRIA Saclay - Île-de-France, F-91893 Orsay cedex.

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 $\mathcal{L}$. Compared to the known best alternatives, our algorithm offers an excellent compromise in terms of space and time complexities.

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 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. 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. Compared to the known best alternatives, our algorithm offers an excellent compromise in terms of space and time complexities.

2 The Dichopile algorithm

At first, let us briefly recall the general principle of the classical recursive method. 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 $\mathcal{A}$):

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

If we note the vector $L_n = \langle l_1(n), l_2(n), \ldots, l_q(n) \rangle$, the principle of the recursive method is 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 Equation 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 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 with the 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 drawing 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 $\mathcal{A}$;
- a vector L_0 of q numbers defined as in the two first items of Equation 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 Equation 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 $L_{n/2}$, $L_{3/4}$, $L_{7/8}$, etc.
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 Equation 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 contents of the stack changes during the drawing.

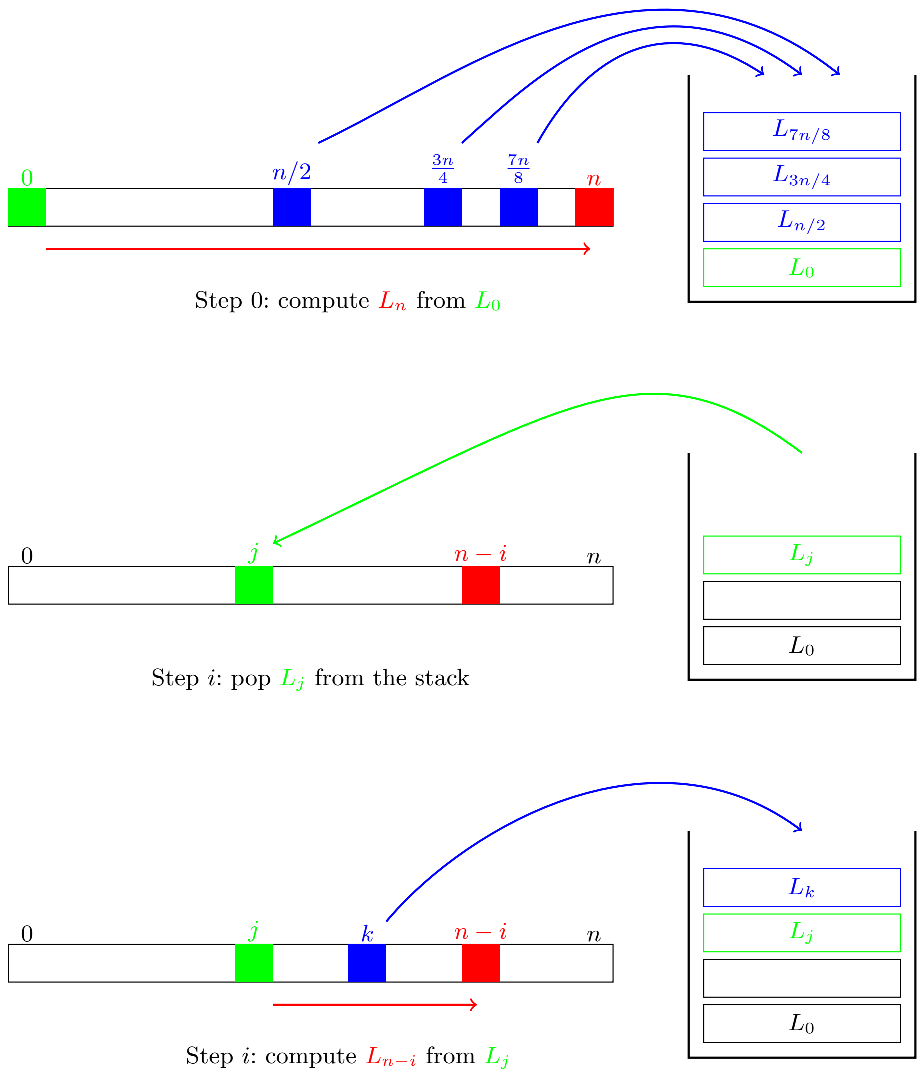

Fig. 1. Principle of the *dichopile* algorithm

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 $\sigma$ 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, gets 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 $\sigma$
```

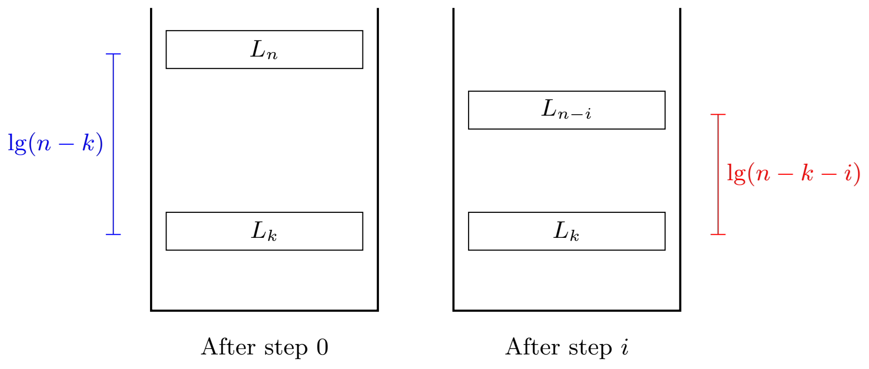

Fig. 2. The stack size is maximal after the first iteration of the dichopile algorithm

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 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 Figure 3.

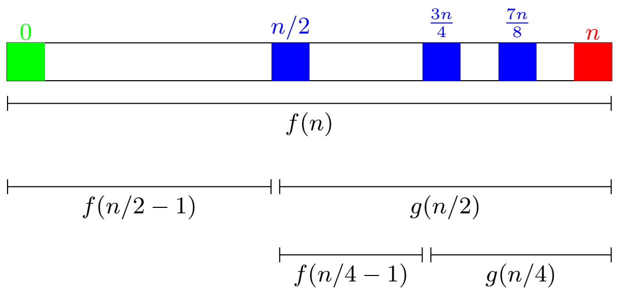

Fig. 3. Recursive scheme of the number of calls to function F done by Algorithm 1; We omit floor and ceiling notations to clarify the figure

The function f counts the number of calculations required when the stack is empty, while the function g counts the number of calculations performed when the stack holds intermediate values. For function $f(n)$, it will first call F n times, then repeat the same work on the first half, while there are several intermediate values in the second half, hence the call to g. Function g calls f for the first half because the smallest value in the stack is half away. Thus, the number of calls to F to generate a path of length n is equal to $f(n)$, which is defined as:

$$\begin{aligned} f(n) &= n + f(\lfloor n/2 \rfloor - 1) + g(\lceil n/2 \rceil) \\ g(n) &= f(\lfloor n/2 \rfloor - 1) + g(\lceil n/2 \rceil) \\ f(1) &= 1 \\ g(1) &= 0 \end{aligned}$$

To study the complexity of this algorithm, we bound this formula by the functions $f_-(n)$ and $f_+(n)$ defined such that:

$$f_-(n) \leq f(n) \leq f_+(n)$$

$$\begin{aligned}
f_-(n) &= n + f_-(n/2-2) + g_-(n/2)\\
g_-(n) &= f_-(n/2-2) + g_-(n/2)\\
f_-(1) &= 1\\
g_-(1) &= 0
\end{aligned}$$

$$\begin{aligned}
f_+(n) &= n + f_+(n/2-1) + g_+(n/2+1)\\
g_+(n) &= f_+(n/2-1) + g_+(n/2+1)\\
f_+(1) &= 1\\
g_+(1) &= 0
\end{aligned}$$

Then, we can find a lower bound of $f_-(n)$:

$$\begin{aligned}
f_-(n) &\geq n + f_-(n/2-2) + f_-(n/4-2) + g_-(n/4)\\
&\geq n + (n/2-2) + f_-(n/4-3) + g_-(n/4-1) + f_-(n/4-2) + g_-(n/4)\\
&\geq n + (n/2-2) + 2\left[f_-(n/4-3) + g_-(n/4-1)\right]\\
&\geq n + \sum_{i=1}^{\lg(n)/2} (n/2 - i2^{i-i})\\
&\geq n + \frac{n\lg(n)}{4} - \frac{2^{\lg(n)/2}(\lg(n)-2)}{2} - 1\\
f_-(n) &= \Omega(n\log n)
\end{aligned}$$

And an upper bound of $f_+(n)$:

$$\begin{aligned}
f_+(n) &\leq n + (n/2-1) + f_+(n/4-1) + g_+(n/4+1) + f_+(n/4) + g_+(n/4+2)\\
&\leq n + (n/2-1) + 2\left[f_+(n/4) + g_+(n/4+2)\right]\\
&\leq n + (n/2-1) + 2\left[n/4 + f_+(n/8-1) + g_+(n/8+1) + f_+(n/8) + g_+(n/8+2)\right]\\
&\leq n + \sum_{i=1}^{\lg(n)-1} n/2\\
&\leq n + \frac{n\lg(n)}{2}\\
f_+(n) &= \mathcal{O}(n\log n)
\end{aligned}$$

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 Conclusion

Table 1 summarizes the bit complexities in time and in space of several 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 1. 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~~	~~$\mathcal{O}(q\log n)$~~	~~$\mathcal{O}(q^2+qn\log n)$~~	~~$\mathcal{O}(qn\log n)$~~
dichopile	float	$\mathcal{O}(q\log^2 n)$	$\mathcal{O}(1)$	$\mathcal{O}(qn\log^2 n)$
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	$\mathcal{O}(q)$	$\mathcal{O}(q^k\log^{k'} n)$	$\mathcal{O}(n^2)$

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. The inverse method has similar problems, with it $\mathcal{O}(qn)$ 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).

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. Note that it is possible to perform exactly uniform generation 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 $\mathcal{O}(q^2n)$ 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. Again, *divide and conquer* and *dichopile* perform well: they are linear in q and almost linear in n, with an advantage for *divide and conquer*. Moreover, the average complexity of the generation stage can be lowered to $\mathcal{O}(qn)$ for *divide and conquer* and $\mathcal{O}(qn \log n)$ for *dichopile* if 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 a few memory but is slow according to n. Both *divide and conquer* and *dichopile* are excellent compromises when considering both space and time complexities. The latter offers a better space complexity, and comparatively the increase of time complexity according to n is quite low.

However, if a 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 $\mathcal{O}(n)$.

Bibliography

Bernardi, O. and Giménez, O. (2010). A linear algorithm for the random generation of regular languages. 11 pages. Submitted. Preprint available on internet. 2

Denise, A. and Zimmermann, P. (1999). Uniform random generation of decomposable structures using floating-point arithmetic. *Theoretical Computer Science*, 218:233–248. 1, 8

Duchon, P., Flajolet, P., Louchard, G., and 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. 1, 2

Flajolet, P., Fusy, E., and Pivoteau., C. (2007). Boltzmann sampling of unlabelled structures. In *Proceedings of the 4th Workshop on Analytic Algorithms and Combinatorics, ANALCO'07 (New Orleans)*, pages 201–211. SIAM. 1

Flajolet, P., Zimmermann, P., and Cutsem, B. V. (1994). A calculus for the random generation of labelled combinatorial structures. *TCS*, 132:1–35. 1

Goldwurm, M. (1995). Random generation of words in an algebraic language in linear binary space. *Information Processing Letters*, 54(4):229–233. 1, 3

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

Oudinet, J. (2010). Random exploration of models. Technical Report 1534, LRI, Université Paris-Sud XI. 15 pages, submitted to Discrete Event Dynamic Systems. 1, 3

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