Génération aléatoire uniforme de mots de langages rationnels*

Alain Denise†

LaBRI, Université Bordeaux I
URA CNRS 1304

Résumé

Nous donnons deux algorithmes de génération aléatoire et uniforme de mots, qui s'appliquent à des classes particulières de langages rationnels. Leur efficacité est mesurée en termes de complexité logarithmique, en fonction de la longueur n des mots engendrés. Le premier algorithme est dédié aux langages dont les séries génératrices possèdent un unique pôle, éventuellement multiple ; sa complexité en temps est de l'ordre de $n \log n$, et l'espace mémoire occupé est en $\log n$. Le second algorithme est réservé aux langages dont les séries génératrices possèdent la propriété suivante : il existe un unique pôle de plus petit module, et ce pôle est simple. Après un pré-traitement en temps polynomial en n, le tirage aléatoire de tout mot s'effectue en temps moyen et espace linéaires.

Abstract

The problem of generating uniformly at random words of a given language has been the subject of extensive study in the last few years. An important part of that work is devoted to the generation of words of context-free languages (see, e.g., [6, 8, 9, 12]). For a given integer $n > 0$, the words of length $n > 0$ of any unambiguous context-free language can be generated uniformly at random by using algorithms derived from the general method which was introduced by Wilf [14, 15] and systematized by Flajolet, Zimmermann and Van Cutsem [7]. Clearly, this can be applied to the set of rational languages, which constitute an important special case of context-free languages.

Most authors use the *uniform* measure of complexity (see [1]) in order to compute the complexity of the algorithms of generation. This measure is based on the following hypotheses: any simple arithmetic operation (addition, multiplication) has time cost 0(1), and a constant amount of memory space is taken by any number. Thus, we know that words of any rational language can be generated by using an algorithm which, with respect to the uniform measure of complexity, runs in linear time (in terms of the length of the words) and constant space [9]. This measure is realistic only if there is a reasonable bound on the numbers involved in the operations. However, the classical random generation algorithms involve operations on numbers which grow exponentially in terms of the length of the words to be generated. Moreover, the programs which make use of these algorithms are generally used to generate very large words, for example for the purpose of studying the asymptotic behavior of some parameters. Therefore, the uniform measure does not reflect the real behavior of such programs. It turns out that the *logarithmic* measure of complexity is much more realistic: one assumes that the space taken by a number k is

*Travail partiellement financé par la subvention EC CHRX-CT93-0400 et le PRC Math-Info.

†Adrelec : `denise@labri.u-bordeaux.fr`

$O(\log k)$, and that any simple arithmetic operation can be done in time $0(\log k)$. It is with respect to this measure that we will evaluate the performance of algorithms in this paper.

Our goal is to design efficient algorithms (in terms of logarithmic complexity) to generate uniformly at random words from certain classes of rational languages. We consider rational languages defined by their minimal finite deterministic automata. When computing complexity, neither the size of the automaton nor the cardinality of the alphabet are taken in account.

In Section 2 we present some background on rational languages and their generating series. We describe briefly the classical method for generating words of such languages and we study its logarithmic complexity. We show that it is at best quadratic for most languages. This is due mainly to computations on numbers which grow exponentially with the length of the words to be generated. In order to improve significantly the efficiency of the algorithms, we must avoid handling of large numbers, or at least decrease substantially the frequency of computations on such numbers. Another alternative, briefly discussed in [7] and [12], is to compute with floating point numbers instead of integers. In this case, the logarithmic complexity is time-linear. However, using floating point numbers leads inevitably to approximations which prevent the exact uniformity of the generation.

In Sections 3 and 4 we show that, in some cases, we can avoid computations on large numbers entirely or almost entirely, while keeping the exact uniformity of the generation. We determine two classes of rational languages for which this is the case.

Section 3 concerns languages whose associated generating series have a unique singularity. We present a simple version of the classical algorithm, which totally avoids handling of large numbers. The logarithmic complexity of the method is $O(n \log n)$ in time and $O(\log n)$ in memory space.

Section 4 focuses on languages whose associated generating series have the following property: there exists a unique singularity of minimum modulus, and this singularity is simple. For such languages we give a probabilistic version of the classical algorithm which generates words randomly while avoiding most computations on large numbers. This method needs a preprocessing stage, which can be done in polynomial time and linear space in terms of the length n of the words. Following preprocessing, any word of length n can be generated in average linear time and space.

1 Introduction

La génération aléatoire uniforme de mots d'un langage est un problème très étudié depuis quelques années. Ceci est notamment dû au fait que le tirage aléatoire de certains objets combinatoires peut se ramener, en utilisant des codages appropriés, à un problème sur des mots (voir par exemple [3, 10]). Beaucoup de travaux concernent la génération de mots de langages algébriques [6, 8, 9, 12] : si l'on se donne un entier $n > 0$, les mots de longueur n de tout langage algébrique non ambigu peuvent être engendrés uniformément au hasard à l'aide d'algorithmes de la même famille que la méthode générale introduite par Wilf [14, 15] et systématisée par Flajolet, Zimmermann et Van Cutsem [7]. Ces procédés s'appliquent bien sûr à l'ensemble des langages rationnels, cas particulier important des langages algébriques.

La plupart des auteurs calculent la complexité des algorithmes de génération selon la mesure de complexité *uniforme*, définie dans [1]. Celle-ci repose sur les hypothèses suivantes : le coût d'une opération arithmétique simple (addition, multiplication) est unitaire, la place en mémoire occupée par un nombre est constante. Ainsi, on montre [9] qu'il est possible d'engendrer aléatoirement des mots de langages rationnels avec un algorithme de complexité uniforme linéaire en temps (en fonction de la longueur des mots engendrés) et constante en espace. La mesure de complexité uniforme est réaliste si les nombres mis en jeu dans les opérations sont

raisonnablement bornés (généralement par la valeur maximale d'un mot binaire en machine). Or, les algorithmes classiques de génération aléatoire font appel à des opérations sur des nombres d'ordre exponentiel en fonction de la longueur des mots à engendrer. De plus, les programmes qui mettent en œuvre ces algorithmes sont en général utilisés pour engendrer des mots de très grande taille, par exemple dans le but d'étudier le comportement asymptotique de certains paramètres. En conséquence, la mesure uniforme ne reflète pas le comportement effectif des programmes mettant en œuvre de tels algorithmes. La mesure de complexité *logarithmique* [1] s'avère alors bien plus réaliste : on suppose que la place occupée par un nombre k est de l'ordre de $\log k$; une opération arithmétique simple s'effectue en temps $O(\log k)$. C'est selon cette mesure que nous évaluerons les performances des algorithmes dans le présent travail.

Nous considèrerons qu'un langage rationnel est défini par l'automate fini déterministe minimal qui le reconnaît. Dans les calculs de complexité, nous ne prendrons pas en compte la taille de l'automate, ni le cardinal de l'alphabet sur lequel est défini le langage.

Dans la seconde section de cet article, après quelques rappels sur les langages rationnels et leurs séries génératrices, nous décrivons brièvement la méthode classique de génération de mots de langages rationnels. Puis nous étudions sa complexité logarithmique : nous montrons qu'elle est au mieux d'ordre quadratique pour la grande majorité des langages, principalement à cause de la présence d'opérations mettant en jeu des nombres entiers d'ordre exponentiel en fonction de la taille des mots engendrés. Pour améliorer sensiblement l'efficacité des algorithmes, il est nécessaire d'éviter la manipulation de grands nombres, ou du moins de diminuer très fortement la fréquence des opérations sur de tels nombres. Une autre alternative, discutée brièvement dans [7] et [12], consiste à effectuer les calculs, non sur des nombres entiers, mais sur des nombres en virgule flottante. Dans ce cas, la complexité est effectivement linéaire en temps. Cependant, les calculs en virgule flottante donnent lieu inévitablement à des approximations qui interdisent l'exacte uniformité de la génération.

Notre but est de montrer que, dans certains cas, il est possible de s'affranchir totalement ou quasi-totalement de la manipulation de grands nombres, tout en conservant un algorithme de génération exactement uniforme. Nous déterminons deux classes de langages rationnels pour lesquels cela est réalisable.

La section 3 concerne les langages dont les séries génératrices associées possèdent un seul pôle. Nous présentons une variante simple de l'algorithme classique, qui permet de s'affranchir totalement de la manipulation de très grands nombres. Nous obtenons ainsi une complexité de l'ordre de $n \log n$ en temps et $\log n$ en espace mémoire.

Dans la section 4, nous nous intéressons aux langages dont les séries génératrices associées présentent la propriété suivante : il existe un unique pôle de module minimum, et ce pôle est simple. Nous décrivons une variante probabiliste de l'algorithme classique, qui permet d'engendrer aléatoirement les mots de ces langages en évitant la plupart des calculs sur des grands nombres. Cette méthode nécessite une phase de pré-traitement qui s'effectue en temps polynomial, et en espace linéaire en n. Cette phase n'est exécutée qu'une fois, quel que soit le nombre de mots à engendrer. Après cette première étape, chaque mot de longueur n peut être engendré en temps et espace mémoire linéaires en moyenne.

2 Etude des algorithmes classiques

2.1 Langages rationnels, automates et séries génératrices

Soit L un langage rationnel sur un alphabet $A = \{a_1, a_2, \ldots, a_k\}$, et $\mathcal{A} = \langle A, Q, q_0, F, \delta \rangle$ son automate fini déterministe minimal (voir par exemple [2] pour les définitions de base concernant les automates). A chaque état $q \in Q$ correspond un langage L_q, dont un automate fini déterministe $\mathcal{A}_q$ est obtenu en changeant l'état initial de $\mathcal{A}$: q devient le nouvel état initial (en particulier, $L = L_{q_0}$). Si q_p est l'état "puits" de l'automate, nous dirons que $\{L_q : q \in Q \setminus \{q_p\}\}$ est *l'ensemble des langages associés à* L. De même, les séries génératrices des langages L_q ($q \in Q \setminus \{q_p\}$) forment l'ensemble des *séries génératrices associées à* L.

La série génératrice $\mathcal{L}(t)$ d'un langage rationnel L s'obtient aisément par résolution d'un système d'équations linéaires. Elle peut s'écrire

$$\mathcal{L}(t) = \frac{P(t)}{Q(t)}$$

où P et Q sont deux polynômes premiers entre eux, et $Q(0) = 1$. Les coefficients de P et Q sont entiers. Le polynôme $Q(t)$ se factorise dans $\mathbb{C}$ comme suit :

$$Q(t) = (1 - \gamma_1 t)^{m_1} (1 - \gamma_2 t)^{m_2} \ldots (1 - \gamma_s t)^{m_s}.$$

Les nombres $\frac{1}{\gamma_1}, \frac{1}{\gamma_2}, \ldots, \frac{1}{\gamma_s} \in \mathbb{C}$ sont les *pôles* de $\mathcal{L}(t)$; l'exposant m_i est la *multiplicité* du pôle $\frac{1}{\gamma_i}$. Soit l_n le nombre de mots de L de longueur n. On sait (voir par exemple [13]) que

$$l_n = \sum_{i=1}^{s} P_i(n) \gamma_i^n \tag{1}$$

où $P_i(n)$ ($1 \leq i \leq s$) est un polynôme de degré inférieur à la multiplicité du pôle $\frac{1}{\gamma_i}$.

Enfin, le module minimum des pôles de $\mathcal{L}(t)$ est lui-même un pôle de $\mathcal{L}(t)$. Ceci est une conséquence d'une propriété bien connue des séries rationnelles à coefficients positifs [4, p. 94].

2.2 Présentation des algorithmes classiques

Soit L un langage tel que défini dans le paragraphe 2, et q l'un des états de son automate minimal. Soient $q_1, q_2, \ldots q_k$ les successeurs de q (les q_j n'étant pas nécessairement distincts deux à deux), tels que pour tout $1 \leq j \leq k$, $\delta(q, a_j) = q_j$. La première lettre d'un mot aléatoire de L_q de longueur i est déterminée de la façon suivante : la probabilité que cette lettre soit a_j ($1 \leq j \leq k$) est égale au quotient du nombre de mots de L_{q_j} de longueur $i - 1$ par le nombre de mots de L_q de longueur i. Nous l'écrirons

$$p(i, q, a_j) = \frac{l_{q_j}[i-1]}{l_q[i]}.$$

Nous convenons d'appeler cette valeur *probabilité de sortie* de l'état q par la lettre a_j (bien qu'il n'y ait pas véritablement de "sortie" si $q_j = q$). Les lettres suivantes sont tirées de la même façon, en décrémentant i et en remplaçant q par q_j, si la lettre a_j a été engendrée. On tire un mot de L de longueur n en posant $q = q_0$ et $i = n$ lors de la première étape. Cet algorithme est présenté en Figure 1.

```
Entrée : Un automate 𝒜 de L, un entier naturel n.
Sortie : Un mot w ∈ L de longueur n.
début
      w ← ϵ
      q ← q_0
      tant que |w| < n faire
      début
            a ← une lettre tirée avec la probabilité p(n − |w|, q, a)
            w ← wa
            q ← δ(q, a)
      fin
fin
```

FIG. 1 – *Génération d'un mot d'un langage rationnel.*

Le tirage d'une lettre avec la probabilité $p(n-|w|,q,a)$ s'effectue avec une méthode très classique, parfois appelée "méthode d'inversion" (voir à ce sujet l'ouvrage très complet de Devroye [5]). La Figure 2 présente un algorithme d'inversion pour le cas qui nous intéresse. Les coefficients $l_q[i]$ et $l_{q_1}[i], \dots l_{q_k}[i]$ intervenant dans cet algorithme peuvent être calculés par une récurrence très simple :

$$l_q[0] = \begin{cases} 1 & \text{si } q \text{ est un état terminal,} \\ 0 & \text{sinon.} \end{cases} \tag{2}$$

$$l_q[i] = \sum_{j=1}^{k} l_{\delta(q,a_j)}[i-1] \quad \text{si } i \neq 0. \tag{3}$$

```
Entrée : Un état q ∈ Q, un entier i.
Sortie : Une lettre a ∈ A choisie avec la probabilité p(i, q, a).
début
      h ← un nombre aléatoire entre 0 et 1
      j ← 1
      π ← p(i, q, a_j)
      tant que π < h faire
      début
            j ← j + 1
            π ← π + p(i, q, a_j)
      fin
      retourner(a_j)
fin
```

FIG. 2 – *Algorithme d'inversion pour choisir une lettre $a \in A$ avec la probabilité $p(i,q,a)$.*

Comme toute méthode de génération de ce type, celle-ci peut se décliner de deux façons au moins. Dans la première version, que nous appellerons *algorithme 1*, tous les coefficients $l_q[i]$ susceptibles d'intervenir lors de la construction d'un mot sont calculés et stockés dans un tableau lors d'une phase préliminaire. Cette phase de précalcul n'existe pas dans la seconde version (*algorithme 2*) ; à chaque étape de la génération, les coefficients $l_q[i]$ utilisés dans l'algorithme d'inversion doivent être calculés. Il existe une méthode intermédiaire, que nous

ne détaillerons pas ici. Elle est mise en oeuvre dans Gaïa [16], le "package" Maple consacré à la génération aléatoire d'objets combinatoires.

Hickey et Cohen [9] proposent une alternative aux formules 2 et 3 pour calculer les coefficients $l_q[i]$. Ils montrent qu'en utilisant la formule 1, il suffit de stocker en permanence un nombre constant de valeurs (étroitement liées aux pôles des séries génératrices de L) pour calculer, au cours de la phase de génération, les probabilités de sortie nécessaires.

2.3 Complexité

Nous nous proposons de comparer les complexités uniforme et logarithmique de chacun des trois algorithmes en fonction de n. Nous supposerons que la complexité de l'algorithme d'inversion (Figure 2) est indépendante de n ; cela est réalisable par exemple avec la méthode donnée en [5, p. 769].

Mesurons la complexité uniforme des deux algorithmes. L'algorithme 1 nécessite une phase de pré-traitement qui n'est exécutée qu'une fois, quel que soit le nombre de mots à engendrer. Cette phase s'execute en temps $O(n)$; de même, le tableau des $l_q[i]$ occupe une place en $O(n)$. La complexité uniforme en temps de l'algorithme 2 est en $O(n^2)$; en revanche, celui-ci ne stocke en permanence qu'un nombre constant de valeurs.

On montre [9] que la complexité uniforme en temps de la variante de Hickey et Cohen est en O(R(n) + mn), où R est un polynôme qui mesure le temps nécessaire pour calculer avec suffisamment de précision les pôles des fonctions génératrices de L. La complexité en espace est constante.

On déduit de la formule (1) du paragraphe précédent que, si L_q est infini, alors il existe un polynôme P_q et un réel $\gamma_q \geq 1$ tels que

$$l_q[i] \sim P_q(i)\gamma_q^i.$$

Supposons qu'il existe $q \in Q$ tel que $\gamma_q > 1$. D'après la récurrence (3), il existe au moins un successeur q' de q tel que $\gamma_{q'} > 1$. Les nombres mis en jeu dans le calcul de $l_q[i]$ sont donc d'ordre au moins exponentiel en i.

Ceci posé, la complexité logarithmique en temps de la phase de pré-traitement de l'algorithme 1 est

$$P_1(m,n) = \Theta(n^2),$$

puis chaque génération d'un mot s'effectue en temps

$$Q_1(m,n) = \Theta(n).$$

Au total, pour engendrer m mots de longueur n, la complexité en temps est égale à

$$T_1(m,n) = \Theta(n^2) + \Theta(mn). \tag{4}$$

L'espace occupé est

$$M_1(m,n) = \Theta(n^2). \tag{5}$$

L'algorithme 2 réclame le temps de calcul suivant :

$$T_2(m,n) = \Theta(mn^3). \tag{6}$$

A chaque étape, au pire, les coefficients $l_q[i]$ et $l_{q_1}[i-1]$, ..., $l_{q_k}[i-1]$ sont stockés. La place mémoire utilisée reste donc linéaire en n :

$$M_2(m,n) = \Theta(n). \tag{7}$$

Considérons maintenant la variante proposée par Hickey et Cohen. Nous savons que valeurs stockées au cours de l'exécution sont en nombre constant. Cependant, elles sont d'ordre exponentiel en fonction de la longueur des mots à engendrer. La complexité logarithmique de l'algorithme est donc linéaire en espace mémoire. Comme dans les autres méthodes, les calculs portent sur des grands nombres. En conséquence, la complexité logarithmique en temps est d'ordre quadratique.

Si pour tout $q \in Q$, $\gamma_q = 1$, alors L fait partie des rares langages rationnels pour lesquels le nombre de mots de longueur n est polynomial en n. Dans ce cas, la complexité logarithmique de chacun des algorithmes est de l'ordre de $n \log n$.

3 Un algorithme pour les langages à un seul pôle

Nous considérons ici les langages dont la série génératrice comporte un seul pôle, éventuellement multiple. Nous avons vu précédemment que les nombres d'ordre exponentiel en n qui interviennent dans le calcul des probabilités de sortie nuisent à l'efficacité de la méthode classique. Le résultat de la Proposition 1 du paragraphe 3.1 nous permettra de calculer ces probabilités en ne manipulant que des nombres d'ordre polynomial en n.

3.1 Calcul des probabilités de sortie

Proposition 1 *Soit L un langage rationnel, et $\mathcal{A} = \langle A, Q, q_0, F, \delta\rangle$ son automate fini déterministe minimal. Si pour tout $q \in Q$, la série génératrice du langage L_q a un et un seul pôle, alors il existe deux polynômes P_1 et P_2 à coefficients entiers tels que*

$$p(n,q,a) = \frac{P_2(n-1)}{P_1(n)}. \tag{8}$$

Preuve. Soit un état $q \in Q$ et γ le pôle de $\mathcal{L}_q(t)$. Alors il existe un polynôme N_q et un entier $r > 0$ tels que

$$\mathcal{L}_q(t) = \frac{N_q(t)}{(1-\gamma t)^r}$$

Les coefficients du développement en série de $\mathcal{L}(t)$ sont entiers, donc γ est entier. D'autre part, il existe un polynôme R_1 tel que le nombre de mots de L_q de longueur $n \in I\!N$ est

$$l_q[n] = R_1(n)\gamma^n.$$

Puisque γ est entier, les coefficients de R_1 sont rationnels. D'autre part, on montre sans peine que pour tout état $q' \in Q$, le pôle de la série génératrice de $L_{q'}$ est γ.

On en déduit que s'il existe $a \in A$ tel que $\delta(q,a) = q'$, alors

$$p(n,q,a) = \frac{R_2(n-1)}{\gamma R_1(n)}$$

fig$_1$*pole.ps*

FIG. 3 – *L'automate fini déterministe complet minimal de L.*

où R_1 et R_2 sont des polynômes à coefficients rationnels. La formule (8) en découle immédiatement. □

Ce résultat détermine un algorithme de génération de mots de langages rationnels à un seul pôle. Pour chaque état $q \in Q$ et chaque lettre $a \in A$, les polynômes P_1 et P_2 de la formule (8) peuvent être calculés (formellement) et stockés dans un tableau lors d'une phase préliminaire. La place occupée par ce tableau est constante. Puis la génération s'effectue selon l'algorithme de la Figure 1. Seul le calcul des probabilités de sortie change : a chaque étape de la génération, les probabilités nécessaires sont évaluées numériquement à partir du tableau de polynômes. Leur calcul, ne faisant intervenir que des nombres d'ordre polynomial en n, s'effectue en temps $\Theta(\log n)$. On en déduit bien évidemment que la complexité logarithmique cet algorithme est de l'ordre de $n \log n$.

3.2 Exemple

Soit L l'ensemble des mots de $\{a, b, c, d\}^*$ ne comportant pas de sous-mot *abcd*. L'automate fini déterministe complet minimal $\mathcal{A}$ de L est présenté en Figure 3. On calcule aisément les séries génératrices des langages L_0, L_1, L_2, L_3 et L_4 correspondant aux états de $\mathcal{A}$.

$$\mathcal{L}_0(t) = \frac{1 - 8t + 22t^2 - 20t^3}{(1-3t)^4},$$

$$\mathcal{L}_1(t) = \frac{1 - 5t + 7t^2}{(1-3t)^3},$$

$$\mathcal{L}_2(t) = \frac{1 - 2t}{(1-3t)^2},$$

$$\mathcal{L}_3(t) = \frac{1}{(1-3t)},$$

$$\mathcal{L}_4(t) = 0.$$

Un rapide calcul permet d'obtenir le nombre de mots de longueur n pour chacun des langages L_i :

$$l_0[n] = \left(\frac{1}{162}n^3 + \frac{1}{27}n^2 + \frac{47}{162}n + 1\right) 3^n,$$

$$l_1[n] = \left(\frac{1}{18}n^2 + \frac{5}{18}n + 1\right) 3^n,$$

$$l_2[n] = \left(\frac{1}{3}n + 1\right) 3^n,$$

$$l_3[n] = 3^n.$$

On en déduit le tableau de la Figure 4, donnant les valeurs des probabilités de sortie pour chaque état de $\mathcal{A}$ et chaque lettre de $\{a, b, c, d\}$.

	a	b	c	d
0	$\frac{3n^2+9n+42}{n^3+6n^2+47n+162}$	$\frac{n^3+3n^2+38n+120}{3(n^3+6n^2+47n+162)}$	$\frac{n^3+3n^2+38n+120}{3(n^3+6n^2+47n+162)}$	$\frac{n^3+3n^2+38n+120}{3(n^3+6n^2+47n+162)}$
1	$\frac{n^2+3n+14}{3(n^2+5n+18)}$	$\frac{2n+4}{n^2+5n+18}$	$\frac{n^2+3n+14}{3(n^2+5n+18)}$	$\frac{n^2+3n+14}{3(n^2+5n+18)}$
2	$\frac{n+2}{3(n+3)}$	$\frac{n+2}{3(n+3)}$	$\frac{1}{n+3}$	$\frac{n+2}{3(n+3)}$
3	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0

FIG. 4 – *Probabilités de sortie* $p(n,q,x)$ *pour* $q \in \{0,1,2,3\}$ *et* $x \in \{a,b,c,d\}$.

4 Langages à plusieurs pôles

Nous étudions dans cette section une classe de langages plus vaste que la précédente, pour laquelle il est possible de concevoir une nouvelle variante de la méthode de génération. Elle permet d'éviter dans une grande mesure le calcul exact des probabilités de sortie, qui participe de façon importante à la complexité des algorithmes classiques.

Supposons pour simplifier notre propos qu'à une étape donnée de l'exécution de l'algorithme on doit choisir une lettre parmi deux possibles, a_1 et a_2. Soit p la probabilité de sortie de l'état courant par la lettre a_1. La méthode que nous présentons dans cette section est fondée sur une modification importante de l'algorithme d'inversion, utilisé pour tirer une lettre à chaque étape. Supposons que nous ne connaissons pas exactement p, mais les deux bornes d'un intervalle $[p_{min}, p_{max}]$ qui contient p. Pour choisir une lettre, tirons un nombre h au hasard entre 0 et 1. Si h est inférieur à p_{min} ou supérieur à p_{max}, nous savons qu'il faut choisir respectivement a_1 ou a_2. En revanche, si h est dans l'intervalle $[p_{min}, p_{max}]$, alors il faut calculer exactement p avant de pouvoir décider.

Pour que cette méthode soit efficace, il faut être capable de calculer très rapidement les valeurs p_{min} et p_{max}. Il faut de plus que la largeur de l'intervalle soit assez petite pour que le calcul exact de p s'effectue très rarement. Nous montrons que cela est réalisable pour les langages rationnels dont les séries génératrices associées possèdent un unique pôle de module minimum, à condition que ce pôle soit simple. Le paragraphe 4.1 contient le résultat théorique qui rend l'algorithme possible (Proposition 2). Dans le paragraphe 4.2, nous présentons l'algorithme dans sa globalité. Le paragraphe 4.3 est consacré aux détails techniques de réalisation de la méthode *d'inversion par approximation*, qui est la pièce essentielle de la méthode de génération. Enfin, nous montrons dans le paragraphe 4.4 que, après un prétraitement de complexité polynomiale en n, la génération de tout mot de longueur n peut s'effectuer en temps moyen et espace mémoire linéaires (Théorème 6). Un exemple d'application à un langage particulier est donné en 4.5.

4.1 Approximation de la probabilité de sortie

Proposition 2 *Soient L un langage rationnel et $\mathcal{A} = \langle A, Q, q_0, F, \delta\rangle$ un automate fini déterministe de L. Si pour tout $q \in Q$, il existe un unique pôle de module minimum dans $\mathcal{L}_q(t)$, et si ce pôle est simple, alors la probabilité pour que $a \in A$ soit la première lettre d'un mot de L_q de longueur $n \in \mathbb{N}$ engendré par l'algorithme de génération aléatoire s'écrit*

$$p(n,q,a) = \hat{p}(q,a) + \epsilon(n) \tag{9}$$

où $\hat{p}(q,a)$ est un nombre algébrique indépendant de n, et il existe $n_0 \in \mathbb{N}$, $\mu \in [0,1[$ et un polynôme P tels que pour tous $n \in \mathbb{N}$, $q \in Q$, $a \in A$,

$$n \geq n_0 \Rightarrow |\epsilon(n)| \leq P(n)\mu^n.$$

Preuve. Soit $q' \in Q$ tel que $\delta(q,a) = q'$. Soient $\gamma_1, \gamma_2, \ldots, \gamma_s$ les inverses des pôles de la série génératrice de L_q. Nous savons (voir paragraphe 2.1) que le pôle de module minimal est réel. Supposons que γ_1 est l'inverse de ce pôle. Alors il existe des nombres $c_1 > 0$, $c_1' \geq 0$ et des polynômes $Q_2, \ldots, Q_s, Q_2', \ldots, Q_s'$ tels que pour tout $n > 0$,

$$l_q[n] = c_1\gamma_1^n + \sum_{i=2}^{s} Q_i(n){\gamma_i}^n \quad \text{et} \quad l_{q'}[n-1] = c_1'\gamma_1^{n-1} + \sum_{i=2}^{s} Q_i'(n-1){\gamma_i}^{n-1}.$$

Alors

$$p(n,q,a) = \hat{p}(q,a) + \epsilon(n)$$

où

$$\hat{p}(q,a) = \frac{c_1'}{c_1\gamma_1} \tag{10}$$

et

$$\epsilon(n) = \frac{c_1\gamma_1 \sum_{i=2}^{s} Q_i'(n-1){\gamma_i}^{n-1} - c_1' \sum_{i=2}^{s} Q_i(n){\gamma_i}^n}{c_1\gamma_1 \left(c_1\gamma_1^n + \sum_{i=2}^{s} Q_i(n){\gamma_i}^n \right)}. \tag{11}$$

Posons

$$E(n) = \sum_{i=2}^{s} Q_i(n)\gamma_i^n.$$

Cherchons un majorant de $|E(n)|$.

$$|E(n)| \leq \sum_{i=2}^{s} |Q_i(n)||\gamma_i^n|.$$

Soit $\bar{Q}$ le polynôme ainsi défini :

$$\forall j \geq 0, \quad [t^j]\bar{Q}(t) = \max_{2 \leq i \leq s} |[t^j]Q_i(t)|$$

et $\bar{\gamma}$ le réel suivant :

$$\bar{\gamma} = \max_{2 \leq i \leq s} |\gamma_i|.$$

Alors pour tout $n \in \mathbb{N}$, pour tout $2 \leq i \leq s$,

$$\bar{Q}(n) \geq Q_i(n)$$

et

$$\bar{\gamma}^n \geq |\gamma_i^n|.$$

Donc

$$|E(n)| \leq (s-1)\bar{Q}(n)\bar{\gamma}^n. \tag{12}$$

On définit de même

$$E'(n) = \sum_{i=2}^{s} Q'_i(n){\gamma_i}^n.$$

et on montre que

$$|E'(n)| \leq (s-1)\bar{Q}'(n)\bar{\gamma}^n \tag{13}$$

où $\bar{Q}'$ est défini similairement à $\bar{Q}$. Notons $D(n)$ le dénominateur de $\epsilon(n)$ dans l'expression (11). $D(n)$ est positif pour tout $n \in I\!N$.

$$\begin{aligned} D(n) &= c_1\gamma_1(c_1\gamma_1^n + E(n)) \\ &\geq c_1\gamma_1(c_1\gamma_1^n - |E(n)|) \\ &\geq c_1\gamma_1(c_1\gamma_1^n - (s-1)\bar{Q}(n)\bar{\gamma}^n) \\ &\geq c_1\gamma_1\left(c_1\gamma_1 - (s-1)\bar{Q}(n)\bar{\gamma}\left(\tfrac{\bar{\gamma}}{\gamma_1}\right)^{n-1}\right)\gamma_1^{n-1}. \end{aligned}$$

Soit $n_0 \in I\!N$ tel que pour tout $n \geq n_0$, $c_1\gamma_1 > (s-1)\bar{Q}(n)(\bar{\gamma}/\gamma_1)^{n-1}$. Un tel nombre existe car $\gamma_1 > \bar{\gamma}$. Alors, si $n \geq n_0$,

$$D(n) \geq c_1\gamma_1\left(c_1\gamma_1 - (s-1)\bar{Q}(n_0)\left(\frac{\bar{\gamma}}{\gamma_1}\right)^{n_0-1}\right)\gamma_1^{n-1} > 0. \tag{14}$$

Considérons maintenant $N(n)$, le numérateur de $\epsilon(n)$ dans l'expression (11).

$$\begin{aligned} |N(n)| &= |c_1\gamma_1 E'(n-1) - c'_1 E(n)| \\ &\leq c_1\gamma_1|E'(n-1)| + c'_1|E(n)| \end{aligned}$$

et donc

$$|N(n)| \leq (s-1)(c_1\gamma_1\bar{Q}'(n-1) + c'_1\bar{\gamma}\bar{Q}(n))\bar{\gamma}^{n-1} \tag{15}$$

d'après les inégalités (12) et (13).

On déduit finalement des expressions (14) et (15) que

$$|\epsilon(n)| \leq P(n)\mu^n$$

si l'on choisit P et μ de façon que

$$P(n) \geq \frac{(s-1)(c_1\gamma_1\bar{Q}'(n-1) + c'_1\bar{\gamma}\bar{Q}(n))}{c_1\bar{\gamma}\left(c_1\gamma_1 - (s-1)\bar{Q}(n_0)\bar{\gamma}\left(\frac{\bar{\gamma}}{\gamma_1}\right)^{n_0-1}\right)} \qquad \forall n \geq n_0 \tag{16}$$

et

$$\mu \geq \frac{\bar{\gamma}}{\gamma_1}. \tag{17}$$

□

4.2 L'algorithme de génération

Soit L un langage sur l'alphabet $A = \{a_1, a_2, \ldots, a_k\}$ et n la longueur du mot à construire. Rappelons que, si l'on définit pour tout état q, tout $0 \leq i \leq n$ et tout $1 \leq j \leq k$ la valeur

$$\pi(i, q, a_j) = \sum_{s=1}^{j} p(i, q, a_s),$$

l'algorithme d'inversion classique (Figure 2) consiste à tirer un nombre aléatoire h entre 0 et 1, puis retourner la lettre a_j si $\pi(i, q, a_{j-1}) < h \leq \pi(i, q, a_j)$.

Supposons que L satisfait les hypothèses de la Proposition 2 et définissons, pour chaque état q de l'automate et pour tout $1 \leq j \leq k$, la valeur

$$\hat{\pi}(q, a_j) = \sum_{s=1}^{j} \hat{p}(q, a_s).$$

Supposons de plus que μ est connu, et que tous les $\hat{\pi}(q, a_j)$ sont calculés.

Posons $K = kP(n)$, où k est le cardinal de l'alphabet A, et P est un polynôme satisfaisant l'inégalité (16). Si l'on définit

$$\pi_{min}(i, q, a_j) = \hat{\pi}(q, a_j) - K\mu^i$$

et

$$\pi_{max}(i, q, a_j) = \hat{\pi}(q, a_j) + K\mu^i,$$

alors on a, pour $n_0 \leq i \leq n$:

$$\pi_{min}(i, q, a_j) \leq \pi(i, q, a_j) \leq \pi_{max}(i, q, a_j).$$

Notons $q_1, q_2, \ldots q_k$ les successeurs de q (non nécessairement distincts deux à deux), tels que $\delta(q, a_j) = q_j$. Nous concevons un algorithme d'inversion suivant les principes qui suivent. Soit h un nombre aléatoire entre 0 et 1. S'il existe j tel que $\pi_{max}(i, q, a_{j-1}) < h \leq \pi_{min}(i, q, a_j)$, alors l'algorithme retournera la lettre a_j. Si, au contraire, $\pi_{min}(i, q, a_j) < h \leq \pi_{max}(i, q, a_j)$, cela signifie que h se trouve dans un intervalle d'incertitude : il sera alors nécessaire de calculer $\pi(i, q, a_j)$ exactement pour pouvoir décider. Comme l'intervalle d'incertitude est exponentiellement petit en fonction de i, la probabilité de devoir effectuer ce calcul sera extrèmement faible. L'algorithme d'inversion conçu selon ces principes est illustré en Figure 5. Nous l'appellerons *algorithme d'inversion par approximation*. L'algorithme complet de génération d'un mot de L de longueur n est présenté en Figure 6.

4.3 Réalisation de l'algorithme d'inversion par approximation

Pour réaliser effectivement l'algorithme d'inversion par approximation (Figure 5), il est nécessaire de se soucier de la représentation des données numériques en machine. Nous convenons donc de représenter les réels de l'intervalle $[0, 1[$ en notation binaire. Ainsi, le mot 101001 est l'écriture binaire du nombre $2^{-1} + 2^{-3} + 2^{-6} = 0,640625$. Si w est un nombre en notation binaire, notons w_i son $i^{ème}$ bit.

Notre problème consiste à implémenter l'algorithme de la Figure 5 de façon efficace. En particulier, on s'abstiendra de calculer véritablement π_{max} et π_{min}, sous peine de perdre

```
Entrée : Un état q ∈ Q, un entier i.
Sortie : Une lettre a ∈ A choisie avec la probabilité p(i, q, a).
début
      h ← un nombre aléatoire entre 0 et 1
      j ← 1
      π̂ = π̂(q, a_1)
      π_min ← π̂ − Kμ^i
      π_max ← π̂ + Kμ^i
      tant que π_max < h faire
      début
            j ← j + 1
            π̂ = π̂(q, a_j)
            π_min ← π̂ − Kμ^i
            π_max ← π̂ + Kμ^i
      fin
      si h ≤ π_min alors retourner(a_j)
      sinon
      début
            π ← π(i, q, a_j)
            si h ≤ π alors retourner(a_j)
            sinon retourner(a_{j+1})
      fin
fin
```

FIG. 5 – *Un algorithme d'inversion par approximation.*

```
Entrée : Un automate A de L, un entier naturel n.
Sortie : Un mot w ∈ L de longueur n.
début
      calculer n_0, K et μ
      pour tout état q faire
            pour toute lettre a faire
                  calculer π̂(q, a)
      w ← ϵ
      q ← q_0
      tant que |w| < n faire
      début
            si |w| < n − n_0 alors
                  tirer une lettre a avec l'algorithme d'inversion par approximation
            sinon
                  tirer une lettre a avec l'algorithme d'inversion classique
            w ← wa
            q ← δ(q, a)
      fin
fin
```

FIG. 6 – *Un algorithme de génération fondé sur l'inversion par approximation.*

le bénéfice que l'on fait en évitant d'évaluer les $\pi(i,q,a_j)$. Pour simplifier l'énoncé, nous supposerons que l'alphabet A se compose de deux lettres seulement, a_1 et a_2. Remarquons que dans ce cas, une seule valeur détermine le choix d'une lettre à une étape donnée de la génération : $\pi(i,q,a_1) = p(i,q,a_1)$. Toutefois, la généralisation à un alphabet de plus de deux lettres pourra s'effectuer sans problème notable.

Nous savons que

$$\hat{\pi}(q,a_1) - K\mu^i \leq \pi(i,q,a_1) \leq \hat{\pi}(q,a_1) + K\mu^i.$$

On en déduit qu'il suffit de connaître un nombre fini (proportionnel à n) de bits de $\hat{\pi}(q,a_1)$ pour construire un mot aléatoire de L de longueur n. Soit en effet

$$e(i) = \lfloor -\log_2 K\mu^i \rfloor. \tag{18}$$

Le nombre $\pi(i,q,a_1)$ est donc encadré comme suit :

$$\hat{\pi}(q,a_1) - 2^{-e(i)} \leq \pi(i,q,a_1) \leq \hat{\pi}(q,a_1) + 2^{-e(i)}. \tag{19}$$

Lorsqu'aucune confusion n'est possible, convenons de noter π pour $\pi(i,q,a_1)$ et $\hat{\pi}$ pour $\hat{\pi}(q,a_1)$. Soit $\underline{\pi}$ le nombre dont l'écriture binaire est formée des $e(i)$ premiers bits de $\hat{\pi}$:

$$\underline{\pi} \leq \hat{\pi} \leq \underline{\pi} + 2^{-e(i)}$$

donc, d'après l'encadrement (19),

$$\underline{\pi} - 2^{-e(i)+1} \leq \pi \leq \underline{\pi} + 2^{-e(i)+1}.$$

Cette dernière formule détermine un intervalle d'incertitude de longueur $4.2^{-e(i)}$ pour π.

L'algorithme d'inversion que nous nous proposons de concevoir comporte deux phases :

- la première phase retourne a_1 avec la probabilité $\underline{\pi} - 2^{-e(i)+1}$, a_2 avec la probabilité $1 - \underline{\pi} - 2^{-e(i)+1}$, et `Indeterminé` avec la probabilité $4.2^{-e(i)}$;
- c'est dans ce dernier cas que la seconde phase est exécutée ; elle met en œuvre le calcul de $l_q[i]$ et $l_{q_1}[i-1]$, puis retourne a_1 ou a_2.

La première phase peut être réalisée selon les principes suivants. Soit h un nombre que l'on construit bit par bit, en comparant à chaque étape le dernier bit tiré au bit correspondant dans $\underline{\pi}$. Supposons que $h_1 h_2 \ldots h_{j-1} = \underline{\pi}_1 \underline{\pi}_2 \ldots \underline{\pi}_{j-1}$ et $h_j \neq \underline{\pi}_j$, pour un certain $j < e(i) - 1$; considérons par exemple le cas où $h_j = 0$ et $\underline{\pi}_j = 1$ (le traitement du cas inverse est symétrique). Alors $h < \underline{\pi} - 2^{-j} < \underline{\pi} - 2^{-e(i)+1}$, *sauf* si tous les bits de $\underline{\pi}$ après le $j^{ème}$ sont nuls. Dans ce dernier cas en effet, on a

$$\begin{aligned} h &= h_1 h_2 \ldots h_{j-1} \texttt{0} \ldots ???, \\ \underline{\pi} &= h_1 h_2 \ldots h_{j-1} \texttt{10000} \ldots \texttt{0}, \\ \underline{\pi} - 2^{-e(i)+1} &= h_1 h_2 \ldots h_{j-1} \texttt{01111} \ldots \texttt{1}. \end{aligned}$$

Il est donc possible que h soit en définitive *égal* à $\underline{\pi} - 2^{-e(i)+1}$: il suffit pour cela que $h_k = 1$ $\forall k \in [j+1, e(i)-1]$. Pour être certain que h n'est pas dans l'intervalle d'incertitude, il faut s'assurer qu'il existe $j' > j$ tel que, ou bien $h_{j'} = 0$, ou bien $\underline{\pi}_{j'} = 0$. On doit donc continuer

```
Entrée : Un nombre 0 < π < 1, de longueur e(i) en notation binaire.
Sortie : a_1 avec la probabilité π − 2^{−e(i)+1}, a_2 avec la probabilité 1 − π − 2^{−e(i)+1},
         Indeterminé avec la probabilité 4.2^{−e(i)}.
début
      j ← 1
      h ← ϵ
      test ← 1
      répéter
            b ← un bit au hasard
            h ← h.b
            si test = 1 alors
                  si b_j ≠ π_j alors
                  début
                        test ← 2
                        t ← h_j
                  fin
            sinon
                  si h_j = t ou π_j ≠ t alors
                  début
                        si h_j = 0 alors retourner(a_1)
                        sinon retourner(a_2)
                  fin
            j ← j + 1
      jusqu'à j > e(i) − 1
      retourner(Indeterminé)
fin
```

FIG. 7 – *Algorithme d'inversion par approximation, première phase.*

de tirer des bits de h, jusqu'à ce que l'une de ces conditions soit remplie, ou que la longueur de h atteigne $e(i) - 1$. Cet algorithme est présenté en Figure 7. Notons qu'aucun calcul des bornes de l'intervalle d'incertitude n'est nécessaire pour sa mise en oeuvre.

Les considérations qui précèdent permettent d'énoncer le résultat suivant.

Proposition 3 *La probabilité pour que l'algorithme de la Figure 7, appliqué à un nombre binaire de e bits, fournisse la réponse* `Indeterminé` *est égale à* 4.2^{-e}.

Lorsque la réponse de la première phase de l'algorithme d'inversion par approximation est `Indeterminé`, la seconde phase (Figure 8) est exécutée. Les premiers bits de $\pi(i, q, a_1)$ sont calculés et comparés à ceux de h ; si nécessaire, l'algorithme tire des bits supplémentaires de h.

```
Entrée : Le nombre h de longueur e(i) − 1 tiré dans la première phase, l_q[i] et l_{q_1}[i − 1].
Sortie : a_1 ou a_2.
    π ← les e(i) − 1 premiers bits de l_{q_1}[i−1] / l_q[i]
    si h < π alors retourner(a_1)
    sinon si h > π alors retourner(a_2)
    sinon
    début
        j ← e(i)
        répéter
            a ← le j^{ème} bit de l_{q_1}[i−1] / l_q[i]
            b ← un bit au hasard
            j ← j + 1
        jusqu'à a ≠ b
        si a > b alors retourner(a_1)
        sinon retourner(a_2)
    fin
fin
```

FIG. 8 – *Algorithme d'inversion par approximation, seconde phase.*

4.4 Complexité

Proposition 4 *Soit* $\underline{\pi}$ *un nombre binaire de e bits, fourni en entrée de la première phase de l'algorithme d'inversion par approximation (Figure 7). Le nombre moyen* $\lambda(e)$ *de comparaisons bit à bit pour fournir une réponse* a_1 *ou* a_2 *satisfait la relation suivante :*

$$\lambda(e) \leq 4 - \frac{e^2 + 3e + 4}{2^e}.$$

Preuve. Considérons une version simplifiée de la première phase de l'algorithme d'inversion par approximation (Figure 7), dans laquelle le test $\underline{\pi}_j \neq t$ n'est pas effectué. Si $\lambda'(e)$ est le nombre moyen de bits tirés par cette version, alors $\lambda(e) < \lambda'(e)$.

On peut supposer sans perte de généralité que $\underline{\pi} = 00\ldots0$. Considérons une exécution de l'algorithme, occasionnant une réponse a_1 ou a_2 après le tirage de j bits exactement. Cela

signifie que, parmi ces j bits, deux exactement, dont le dernier, sont égaux à 1. La probabilité de cet événement est égale à $(j-1)2^{-j}$. Le nombre moyen de bits tirés est donc

$$\begin{aligned}\lambda'(e) &= \sum_{j=1}^{e} \frac{j(j-1)}{2^j} \\ &= 4 - \frac{e^2+3e+4}{2^e}.\end{aligned}$$

□

Proposition 5 *Le temps moyen $\tau(i)$ nécessaire à l'algorithme d'inversion par approximation pour fournir une réponse a_1 avec la probabilité $\pi(i,q,a_1)$ ou a_2 avec la probabilité $1-\pi(i,q,a_1)$ satisfait l'équation :*

$$\tau(i) = (1 - 4.2^{-e(i)})\lambda(e(i)) + 4.2^{-e(i)}R(i),$$

où $R(i)$ est une expression d'ordre polynomial en i, et $\lambda(e(i)) < 4$.

Preuve. Si la seconde phase de l'algorithme doit être exécutée, il faut calculer en moyenne $e(i)+1$ bits de $l_{q_1}[i-1]/l_q[i]$ pour obtenir une réponse. $R(i)$ correspond au temps de ce calcul. Ceci posé, le résultat découle directement des Propositions 3 et 4. □

Voici enfin le résultat principal de cette section, donnant la complexité moyenne de l'algorithme de la Figure 6, construisant des mots aléatoires de langages rationnels satisfaisant les hypothèses de la Proposition 2.

Théorème 6 *Soient L un langage rationnel et $\mathcal{A} = \langle A, Q, q_0, F, \delta\rangle$ son automate fini déterministe minimal. Supposons que pour tout $q \in Q$, il existe un unique pôle de module minimum dans $\mathcal{L}_q(t)$, et que ce pôle est simple. Alors la complexité moyenne en temps de l'algorithme de la Figure 6 pour générer m mots de L de longueur n est*

$$\mathcal{T}(m,n) \sim F(n) + 4mn,$$

où $F(n)$ est une fonction d'ordre polynomial en n. Sa complexité moyenne en espace est

$$\mathcal{M}(m,n) = O(n).$$

Preuve. Les nombres n_0, K et μ peuvent être évalués en temps constant en fonction de n. Pour tout état q et toute lettre a, le calcul des n décimales de $\underline{\pi}(q,a)$ fait appel à des opérations arithmétiques classiques sur des nombres algébriques (somme, produit, puissance ...). Il peut donc être effectué en temps polynomial en n [11]. Cette phase préliminaire de l'algorithme ne s'effectue qu'une fois, quel que soit le nombre de mots à construire. $F(n)$ représente son temps de calcul.

Le temps de génération d'un mot, une fois les calculs préliminaires effectués est :

$$T_2(1,n_0) + \sum_{i=n_0+1}^{n} \tau(i) \sim 4n.$$

où $T_2(1,n_0)$ est la complexité de l'algorithme 2 pour engendrer un mot de longueur n_0 (section 2, expression (6)). □

fig$_f$*ibo.ps*

FIG. 9 – *L'automate fini déterministe complet minimal du langage de Fibonacci.*

4.5 Exemple

Soit F le *langage de Fibonacci*, défini par l'automate de la Figure 9. La seule probabilité à calculer est probabilité de sortie de l'état 1 par la lettre a. On montre que

$$l_1[n] = c_1\gamma_1 + c_2\gamma_2$$

et

$$l_0[n] = c'_1\gamma_1 + c'_2\gamma_2$$

avec

$$\begin{aligned} c_1 &= \frac{5+3\sqrt{5}}{10}, \quad c_2 = \frac{5-3\sqrt{5}}{10}, \\ c'_1 &= \frac{5+\sqrt{5}}{10}, \quad c'_2 = \frac{5-\sqrt{5}}{10}, \\ \gamma_1 &= \frac{1+\sqrt{5}}{2}, \quad \gamma_2 = \frac{1-\sqrt{5}}{2}. \end{aligned}$$

Appliquons la Proposition 2. Nous pouvons prendre $n_0 = 1$, et calculer $\hat{\pi}(1,a) = \hat{p}(1,a)$, $K = P(n)$ et μ selon les formules (10), (16) et (17).

$$\hat{\pi}(1,a) \;=\; \frac{c'_1}{c_1\gamma_1} \;=\; \frac{3-\sqrt{5}}{2}.$$

$$P(n) \;\geq\; \frac{c_1\gamma_1|c'_2| + c'_1|\gamma_2||c_2|}{c_1|\gamma_2|(c_1\gamma_1 - |c_2||\gamma_2|)} \;\approx\; 0,46.$$

$$\mu \;\geq\; \frac{|\gamma_2|}{\gamma_1} \;\approx\; 0,38.$$

Prenons $P(n) = \mu = 1/2$. Dans ce cas,

$$\pi(n,1,a) = \frac{3-\sqrt{5}}{2} + \epsilon(n)$$

et $|\epsilon(n)| \leq 2^{-(n+1)}$ pour tout $n > 1$. D'après la formule (18), $e(n)$ est égal à $2n$. Le Théorème 6 nous permet de conclure que la complexité moyenne en temps de l'algorithme de construction de m mots de Fibonacci de longueur n est :

$$\mathcal{T}(m,n) = O(n^2) + 4m(n-1).$$

La complexité en espace est linéaire en moyenne.

5 Problème

Nous nous faisons ici l'écho d'une question posée par Jean Berstel et Jean-Guy Penaud. Les deux classes de langages rationnels que nous considérons en sections 3 et 4 sont définies de façon indirecte : pour montrer qu'un langage appartient à l'une ou l'autre de ces classes, il est nécessaire de calculer les pôles de sa série génératrice. Serait-il possible de décider plus aisément de cette appartenance ? En d'autres termes, peut-on déterminer précisément le lien entre la définition d'un rationnel (par une grammaire, un automate ou une expression régulière) et certaines propriétés des pôles de sa série génératrice ?

Références

[1] Aho (A. V.), Hopcroft (J. E.) et Ullman (J. D.). – *The design and analysis of computer algorithms*. – Addison-Wesley, 1974.

[2] Autebert (J-M.). – *Langages algébriques*. – Masson, 1987.

[3] Barcucci (E.), Pinzani (R.) et Sprugnoli (R.). – Génération aléatoire des animaux dirigés. *In : Actes de l'Atelier Franco-Québécois de Combinatoire, 1991, publi LaCIM 10*, éd. par Labelle (J.) et Penaud (J. G.). – Université du Québec à Montréal.

[4] Berstel (J.) et Reutenauer (C.). – *Les séries rationnelles et leurs langages*. – Masson, 1984.

[5] Devroye (L.). – *Non-uniform random variate generation*. – Springer Verlag, 1986.

[6] Flajolet (P.), Goldwurm (M.) et Steyaert (J. M.). – Random generation and context-free languages. – 1990. Manuscrit.

[7] Flajolet (Ph.), Zimmermann (P.) et Van Cutsem (B.). – A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, vol. 132, 1994, pp. 1–35.

[8] Goldwurm (M.). – Random generation of words in an algebraic language in linear binary space. *Information Processing Letters*, vol. 54, 1995, pp. 229–233.

[9] Hickey (T.) et Cohen (J.). – Uniform random generation of strings in a context-free language. *SIAM. J. Comput*, vol. 12, n° 4, 1983, pp. 645–655.

[10] Hochstättler (W.), Loebl (M.) et Moll (C.). – Generating convex polyominoes at random. *In : Actes du 5ème Colloque Séries Formelles et Combinatoire Algébrique, 1993*, éd. par Barlotti (A.), Delest (M.) et Pinzani (R.). Université de Florence, pp. 267–278.

[11] Knuth (D.E.). – *The art of computer programming*. – Addison Wesley, 1969, *Addison-Wesley series in computer science and information processing*, volume 2 : Seminumerical algorithms.

[12] Mairson (H. G.). – Generating words in a context free language uniformly at random. *Information Processing Letters*, vol. 49, 1994, pp. 95–99.

[13] Mignotte (M.). – Séries rationnelles en une variable. *In : Séries formelles en variables non commutatives et applications, actes de la cinquième Ecole de Printemps d'informatique théorique*, éd. par Berstel (J.). – Vieux-Boucau les Bains, 1977.

[14] Nijenhuis (A.) et Wilf (H. S.). – *Combinatorial algorithms*. – Academic Press Inc., 1979.

[15] Wilf (H. S.). – A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, vol. 24, 1977, pp. 281–291.

[16] Zimmermann (P.). – Gaïa: a package for the random generation of combinatorial structures. *MapleTech*, vol. 1, n° 1, 1994, pp. 38–46.