
Sujet : Test de performances par génération aléatoire

Il est rappelé que le jury attend un exposé de 35 minutes, pédagogique et structuré, fondé sur ce sujet. Ce texte se conclut par une liste de pistes de réflexion pour guider la conception de l'exposé, mais toute initiative personnelle pertinente est appréciée. L'exposé doit contenir une ou plusieurs illustrations informatiques ainsi qu'une discussion autour d'une dimension éthique, sociétale, environnementale, économique ou juridique. Une piste particulière est proposée à cet effet.

1 Introduction

En algorithmique, le cadre classique est d'estimer la complexité en temps ou en espace des algorithmes dans le pire cas, afin d'en garantir les performances. Il y a cependant des situations où il est pertinent de s'intéresser à d'autres types de complexités, notamment la complexité en moyenne. C'est typiquement le cas pour éviter de disqualifier un algorithme particulièrement efficace sur de nombreuses instances d'entrée uniquement parce qu'il est moins performant dans de très rares cas. L'exemple emblématique est l'algorithme du tri rapide (QUICKSORT) qui est très utilisé en pratique, même s'il est en $\mathcal{O}(n^2)$ dans le pire cas.

Pour faire une analyse en moyenne d'un algorithme, il faut choisir un modèle aléatoire pour les données. Une fois le modèle choisi, on peut soit estimer mathématiquement la complexité en moyenne, soit créer un générateur aléatoire d'entrées de taille n pour faire des simulations de performance. C'est la deuxième approche que nous allons adopter dans ce sujet. Ces générateurs doivent être efficaces, afin de pouvoir réaliser de nombreux tests facilement.

Dans ce document nous nous intéressons à la génération aléatoire de trois types d'entrées : les permutations, les mots et les langages réguliers. Dans chacun des cas, en partant d'un algorithme de génération aléatoire uniforme, nous proposons une discussion sur la pertinence de cette approche pour l'analyse de certains algorithmes qui manipulent ce type de données.

2 Permutation aléatoires uniformes

Dans un premier temps, nous nous intéressons à l'étude expérimentale d'algorithmes de tri. L'analyse théorique de QUICKSORT permet d'obtenir le résultat suivant :

Théorème 1 *Pour tout $n \geq 1$, la complexité en moyenne de QUICKSORT est en $\mathcal{O}(n \log n)$ pour une permutation uniforme de taille n .*

Pour simuler ce scénario aléatoire, il faut un générateur aléatoire de permutations, qui est *uniforme* : toutes les permutations possibles des nombres de $\{0, 1, \dots, n-1\}$ ont la même probabilité

$\frac{1}{n!}$ d'être produites. Pour cela, en supposant qu'on peut tirer aléatoirement et uniformément un entier dans un intervalle donné, on peut utiliser l'algorithme de Fisher-Yates (parfois appelé le mélange de Knuth) dont le pseudo-code est donné ci-dessous (Algorithme 1).

Algorithme 1 : Algorithme de Fisher-Yates

Entrée : un entier $n \geq 1$
Sortie : une permutation aléatoire uniforme de $\{0, 1, \dots, n-1\}$

```

1  $\sigma = [0, 1, \dots, n-1]$ 
2 pour  $i \leftarrow n-1$  jusqu'à 1 faire
3    $j \leftarrow$  nombre entier aléatoire uniforme de  $\{0, \dots, i\}$ 
4   échanger  $\sigma[i]$  et  $\sigma[j]$ 
5 renvoyer  $\sigma$ 
```

FIGURE 1 – L'algorithme de Fisher-Yates pour générer une permutation aléatoire uniforme.

L'algorithme de Fisher-Yates est efficace pour produire des permutations uniformes et permet de réaliser des études expérimentales comme celles présentées figure 2. On peut cependant remarquer que ce choix de distribution de probabilités, bien que naturel, est arbitraire. Par exemple, les entrées générées n'ont jamais de doublons : on peut se demander si QUICKSORT reste efficace en moyenne sur des tableaux aléatoires avec peu de valeurs différentes. C'est également testé dans la figure 2, où l'on a considéré des tableaux aléatoires uniformes de taille n contenant au plus $\lceil \sqrt{n} \rceil$ valeurs différentes : on observe une dégradation des performances de l'algorithme.

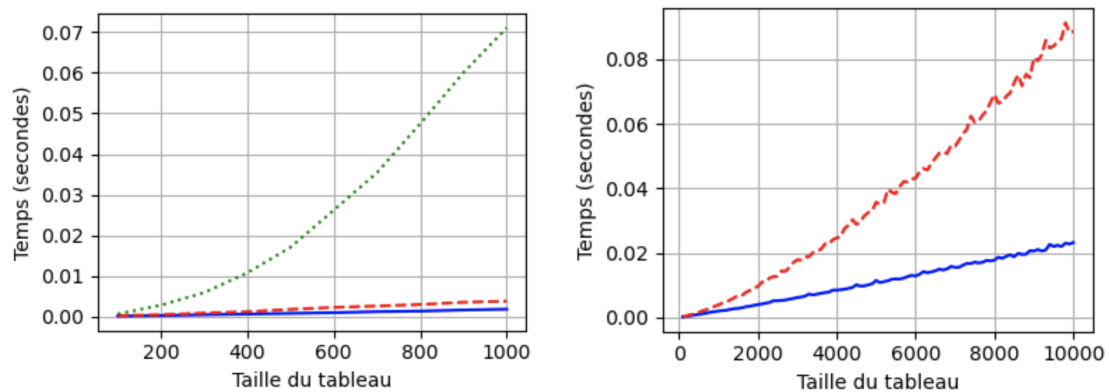


FIGURE 2 – Étude expérimentale du temps d'exécution de QUICKSORT : le pire cas en pointillé vert, le cas aléatoire uniforme en ligne pleine bleue, et le cas aléatoire avec seulement $\lceil \sqrt{n} \rceil$ valeurs différentes tracé en ligne en tirets rouges (chaque case du tableau contient un entier entre 1 et $\lceil \sqrt{n} \rceil$ pris uniformément au hasard et indépendamment). À gauche, les trois simulations sur le même graphique. Dans la version implantée de QUICKSORT, si le pivot a des doublons, ils se retrouvent dans le sous-tableau gauche de la procédure de partitionnement. À droite, le pire cas a été enlevé afin de mieux observer les différences entre les deux modèles aléatoires étudiés.

La question de quelles probabilités considérer sur les entrées est cruciale en pratique : empiriquement, les ingénieurs qui développent les langages de programmation ont observé que les algorithmes de tri étaient souvent appelés sur des entrées déjà partiellement ordonnées, ce qui a conduit à adopter des algorithmes qui savent profiter de cette particularité.

3 Textes aléatoires

On s'intéresse maintenant à générer aléatoirement des mots de longueur n sur un alphabet Σ donné. On peut facilement faire un générateur aléatoire uniforme de complexité linéaire en tirant aléatoirement chaque lettre indépendamment. Cependant, si on a un domaine d'application précis, comme des textes en français, du code source en C ou encore des séquences d'ADN, de tels mots aléatoires ne vont pas beaucoup ressembler aux entrées typiques des algorithmes considérés.

Si on dispose d'exemples, comme un long texte en français, on peut s'en servir pour essayer de faire des générateurs plus ressemblants. La première idée est de calculer la fréquence de chaque lettre et qu'elle soit générée avec une probabilité proportionnelle à sa fréquence. En faisant une simulation, on se rend vite compte que le texte généré par cette méthode n'est pas très ressemblant.

Pour aller plus loin, et capturer les dépendances entre les caractères consécutifs, on peut utiliser des *chaînes de Markov*. Dans notre contexte, si on part d'un texte de référence T , on construit un graphe orienté pondéré $\mathcal{M}_T^{(1)}$ dont les sommets sont les caractères de T . Il y a un arc entre α et β quand le mot $\alpha\beta$ est un facteur de T . On considère T comme étant circulaire : après sa dernière lettre on revient à la première ; par exemple n est suivi d'un a pour $T = \text{agregation}$. On équipe chaque arc $\alpha \xrightarrow{w} \beta$ d'un poids $w = \frac{|T|_{\alpha\beta}}{|T|_{\alpha}}$, où $|T|_{\alpha\beta}$ est le nombre d'occurrences du facteur $\alpha\beta$ circulairement dans T , et où $|T|_{\alpha}$ est le nombre d'occurrences de α .

Par construction, la somme des poids des arcs sortants de chaque sommet est égale à 1. On peut utiliser $\mathcal{M}_T^{(1)}$ pour générer aléatoirement des textes, de longueur n donnée, qui ressemblent à T de la façon suivante :

1. On tire aléatoirement le sommet courant initial s_0 , avec une probabilité pondérée par les fréquences des caractères de T .
2. On passe du sommet s_i au sommet s_{i+1} en tirant aléatoirement l'arc sortant utilisé depuis s_i , en respectant leurs poids.

Les étiquettes de $s_0, s_1, \dots, s_{n-1}$ forment le mot généré aléatoirement par $\mathcal{M}_T^{(1)}$. Le graphe $\mathcal{M}_T^{(1)}$ est appelé la *chaîne de Markov d'ordre 1 associée à T* . Un exemple est donné figure 3.

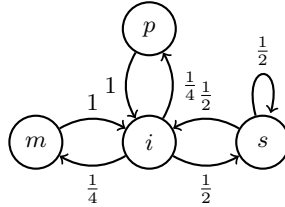


FIGURE 3 – La chaîne de Markov d'ordre 1, $\mathcal{M}^{(1)}(T)$, pour $T = \text{mississippi}$. Pour une longueur $n = 3$, cette chaîne génère le mot sim avec probabilité $\frac{4}{10} \times \frac{1}{2} \times \frac{1}{4} = \frac{1}{20}$.

Pour une meilleure modélisation, on peut facilement généraliser cette construction à des chaînes de Markov d'ordre supérieur. Pour la chaîne de Markov d'ordre 2 associée à T , notée $\mathcal{M}_T^{(2)}$, on prend comme sommets tous les facteurs de taille 2 (toujours circulairement). Il y a un arc $\alpha\beta \xrightarrow{w} \beta\gamma$ quand $\alpha\beta\gamma$ est un facteur circulaire de T . Son poids est $w = \frac{|T|_{\alpha\beta\gamma}}{|T|_{\alpha\beta}}$. La génération aléatoire se fait de façon similaire. Pour la chaîne d'ordre k , les sommets sont les facteurs circulaires de longueur k , et on a un arc $\alpha_1 \dots \alpha_k \xrightarrow{w} \alpha_2 \dots \alpha_{k+1}$ quand $\alpha_1 \dots \alpha_{k+1}$ est un facteur circulaire de T . Son poids est $w = \frac{|T|_{\alpha_1 \dots \alpha_{k+1}}}{|T|_{\alpha_1 \dots \alpha_k}}$. La figure 4 permet de se rendre compte de l'impact de l'ordre de la chaîne de Markov sur la génération de texte en français.

La table 1 permet de mesurer à quel point un texte généré par un des modèles présentés ci-dessus se comporte comme le texte d'origine pour l'algorithme de compression utilisé par **gzip**.

```

uniforme : AbSZvUOPrUghnHCglu lNAuzjLdæZSNQfNDnuøýS SqLhhenBYqmiRGyvjqY
fréquences : soea rrerxdleac dea lartptstaeteimiied ensiasrc csrevn add
ordre 1 : Catee den lemel doiceti l On Nonte tagotaus da couren ves e
ordre 2 : Attre du de de ver laituelont parcil nophattait ruessauttans
ordre 3 : vues redeconcernie au bar une Lui avait arriva de ne sans ma
ordre 4 : couve Eh bien de tournaux aux endu et pour le me dexil il s
ordre 5 : fants et de linge que le jugeraQuelques fidele Paraitraille

```

FIGURE 4 – Des mots générés dans les différents modèles (**fréquences** correspond à tirer aléatoirement et indépendamment les lettres selon leur fréquence). Le texte de référence est un roman de 524ko (*La peste*, de Camus).

Cela semble indiquer que des petits ordres de chaînes de Markov ne sont pas suffisant pour tester les performances d’algorithmes de compression pour du texte en français.

Fichier	Original	Compressé	Ratio (%)
la_pestes.txt	523 735 o.	184 388 o.	35.20
uniforme.txt	101 998 o.	74 111 o.	72.66
ordre0.txt	100 021 o.	58 463 o.	58.42
ordre1.txt	100 021 o.	50 244 o.	50.23
ordre2.txt	100 020 o.	45 049 o.	45.04
ordre3.txt	100 014 o.	41 173 o.	41.15
ordre4.txt	100 016 o.	38 763 o.	38.75
ordre5.txt	100 011 o.	37 631 o.	37.62

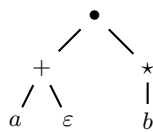
TABLE 1 – Taille compressée par gzip, en pourcentage de la taille originale. Le fichier original est “[la_pestes.txt](#)”, les autres textes ont été générés selon les différentes méthodes vues dans cette partie. Les tailles sont donnée en octets.

4 Langages réguliers aléatoires

On s’intéresse dans cette partie à la génération aléatoire uniforme de langages réguliers avec deux notions de taille différentes, selon qu’on les représente par des expressions rationnelles ou par des automates déterministes et complets. On travaille sur un alphabet Σ fixé qui contient au moins deux lettres. Dans toute la suite on ne considère que des langages non vides, et le symbole $\emptyset$ ne sera pas utilisé dans les expressions rationnelles.

4.1 Expressions rationnelles

Dans cette partie, on fixe $\Sigma = \{a, b\}$. On voit une expression rationnelle comme un arbre unaire-binaire, dont les nœuds unaires sont étiquetés par le symbole $\star$ (pour l’étoile de Kleene), les nœuds binaires sont étiquetés soit par le symbole $+$ (pour l’union), soit par le symbole $\bullet$ (pour la concaténation), et dont les feuilles sont étiquetées soit par une lettre de Σ , soit par le mot vide ε . Par exemple, l’expression rationnelle $(a + \varepsilon) \cdot b^*$ sera représentée par l’arbre :



On parlera d'*arbre d'expression* dans la suite pour désigner un tel objet. La *taille* d'un arbre d'expression est son nombre de nœuds, soit 6 sur l'exemple ci-dessus.

Pour $n \geq 1$, on note a_n le nombre d'arbres d'expression de taille n sur $\Sigma = \{a, b\}$. Les remarques suivantes permettent d'obtenir une formule de récurrence pour les a_n :

- (i) $a_1 = 3$ et $a_2 = 3$,
- (ii) pour tout $n \geq 2$, il y a a_{n-1} arbres d'expression de taille n qui ont une étoile à la racine,
- (iii) pour tout $n \geq 3$, il y a $\sum_{i=1}^{n-2} a_i \times a_{n-1-i}$ arbres d'expression de taille n qui ont une union à la racine, et autant qui ont une concaténation à la racine.

On a ainsi :

$$\begin{cases} a_1 = a_2 = 3 \\ a_n = a_{n-1} + 2 \sum_{i=1}^{n-2} a_i \times a_{n-1-i}, & \text{pour tout } n \geq 3. \end{cases}$$

Si on a pré-calculé tous les a_m pour $m \in \{1, \dots, n\}$ on peut facilement tirer aléatoirement et uniformément un arbre d'expression, en procédant récursivement :

- Si $n = 1$, on tire aléatoirement et uniformément a , b ou ε .
- Si $n = 2$, on tire aléatoirement et uniformément $\begin{smallmatrix} \star \\ a \end{smallmatrix}$, $\begin{smallmatrix} \star \\ b \end{smallmatrix}$ ou $\begin{smallmatrix} \star \\ \varepsilon \end{smallmatrix}$.
- Si $n \geq 3$ alors
 - Avec probabilité $\frac{a_{n-1}}{a_n}$ on renvoie $\begin{smallmatrix} \star \\ A \end{smallmatrix}$, où A est un arbre aléatoire de taille $n - 1$
 - Sinon on tire aléatoirement et uniformément l'étiquette de la racine $+$ ou $\bullet$, puis on tire aléatoirement la taille du sous-arbre gauche, qui vaut i avec probabilité $\frac{a_i \times a_{n-1-i}}{\frac{1}{2}(a_n - a_{n-1})}$. Il suffit ensuite de générer le sous-arbre gauche de taille i et le sous-arbre droit de taille $n - 1 - i$.

Dans la figure 5 nous avons étudié la taille moyenne de l'automate de Glushkov (en regardant son nombre de transitions) construit à partir d'arbres d'expression aléatoires. Bien que dans le pire des cas un arbre de taille n peut produire un automate de Glushkov avec un nombre quadratique de transitions, on observe un comportement moyen linéaire. C'est d'ailleurs prouvé mathématiquement :

Théorème 2 *Pour un alphabet avec au moins deux lettres, le nombre moyen de transitions de l'automate de Glushkov d'un arbre d'expression de taille n pris uniformément au hasard est linéaire en n .*

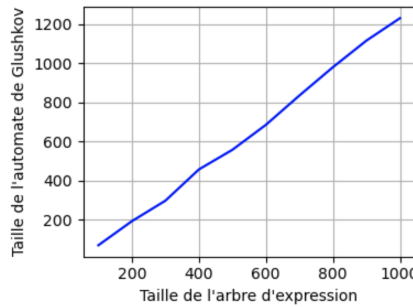


FIGURE 5 – Taille moyenne, en nombre de transitions, de l'automate de Glushkov d'un arbre d'expression aléatoire uniforme.

Le problème du Théorème 2 est que si la distribution uniforme sur les arbres a du sens au niveau syntaxique, elle induit une distribution dégénérée sur les langages réguliers.

On peut formaliser cette notion en appliquant des règles de réduction syntaxiques directement sur les arbres d'expression : on introduit un nouveau type de nœud Σ^* et on utilise les règles de

réécriture locales suivantes, pour tout arbre d'expression A , et tout arbre d'expression A_ε dont le langage contient ε :

$$\begin{array}{c} \star \\ | \\ \wedge \\ a \quad b \end{array} \mid \begin{array}{c} \star \\ | \\ \wedge \\ b \quad a \end{array} \mapsto \Sigma^*, \quad \begin{array}{c} + \\ | \\ \wedge \\ \Sigma^* \quad A \end{array} \mid \begin{array}{c} + \\ | \\ \wedge \\ A \quad \Sigma^* \end{array} \mapsto \Sigma^*, \quad \text{et} \quad \begin{array}{c} \bullet \\ | \\ \wedge \\ \Sigma^* \quad A_\varepsilon \end{array} \mid \begin{array}{c} \bullet \\ | \\ \wedge \\ A_\varepsilon \quad \Sigma^* \end{array} \mapsto \Sigma^*. \quad (1)$$

On applique ces règles autant de fois que possible, jusqu'à ce qu'aucune ne s'applique, comme illustré dans la figure 6 (le résultat ne dépend pas de l'ordre dans lequel on applique les règles de simplification). L'arbre obtenu à la fin est appelé l'*arbre réduit* de l'arbre d'expression.

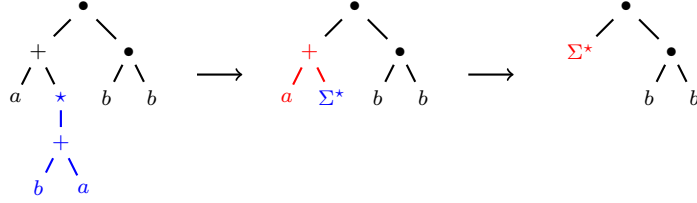


FIGURE 6 – Exemple d'application des règles de réduction.

Le théorème ci-dessous formalise le fait qu'il est difficile de générer aléatoirement des langages intéressants en passant par des arbres d'expression uniformes : on peut majorer la taille moyenne de leur arbre réduit par une constante qui ne dépend pas de la taille de l'arbre d'expression généré.

Théorème 3 *Pour tout alphabet avec au moins deux lettres, il existe une constante $C > 0$ telle que pour tout $n \geq 1$, la taille moyenne de l'arbre réduit d'un arbre d'expression de taille n pris uniformément au hasard est majorée par C .*

4.2 Automates

Puisqu'utiliser les expressions rationnelles aléatoires pour générer des langages réguliers est d'un intérêt limité, on tente dans cette partie de se baser sur les automates finis. Si on considère des automates non-déterministes, on retombe sur le même genre de difficultés que celle identifiée dans le Théorème 3.

On s'intéresse donc à la génération aléatoire uniforme d'automates déterministes et complets : on fixe un alphabet Σ avec au moins deux lettres, et on considère les automates $\mathcal{A} = (n, \delta, F)$ où

- n est le nombre d'états de l'automate, et son ensemble d'état est $Q_n = \{1, \dots, n\}$;
- δ est la fonction de transition qui à tout état $q \in Q_n$ et toute lettre $\alpha \in \Sigma$ associe un état $\delta(q, \alpha)$;
- l'état initial est toujours 1 ;
- $F \subset Q_n$ est l'ensemble des états finaux.

Pour tout $n \geq 1$, on peut facilement générer un tel automate en tirant aléatoirement uniformément et indépendamment les images $\delta(q, \alpha)$, pour tout $q \in Q_n$ et tout $\alpha \in \Sigma$, puis en prenant un sous-ensemble de Q_n uniformément au hasard pour F .

On s'aperçoit sur les simulations que de tels automates sont très souvent non accessibles : il existe des états qu'on ne peut pas atteindre depuis l'état initial et qui sont donc inutiles si on s'intéresse au langage reconnu. Cependant, si on réduit l'automate en ne gardant que les états accessibles, on ne perd qu'une fraction des états (voir la figure 7) avec grande probabilité. Un résultat théorique confirme ces constatations expérimentales :

Théorème 4 *La taille moyenne de la partie accessible d'un automate déterministe et complet aléatoire avec n état est asymptotiquement équivalente à $\lambda \cdot n$, où $0 < \lambda < 1$ est solution de l'équation $x = 1 - \exp(-|\Sigma| \cdot x)$.*

On trouve par exemple que pour $|\Sigma| = 2$, la constante $\lambda \approx 0.7968$, et donc que la partie accessible contient à peu près 80% des états. On note dans la suite $\text{acc}(\mathcal{A})$ l'automate obtenu en enlevant de $\mathcal{A}$ les états non accessibles, puis en renumérotant ses états pour avoir des entiers consécutifs (en gardant le même ordre que dans $\mathcal{A}$).

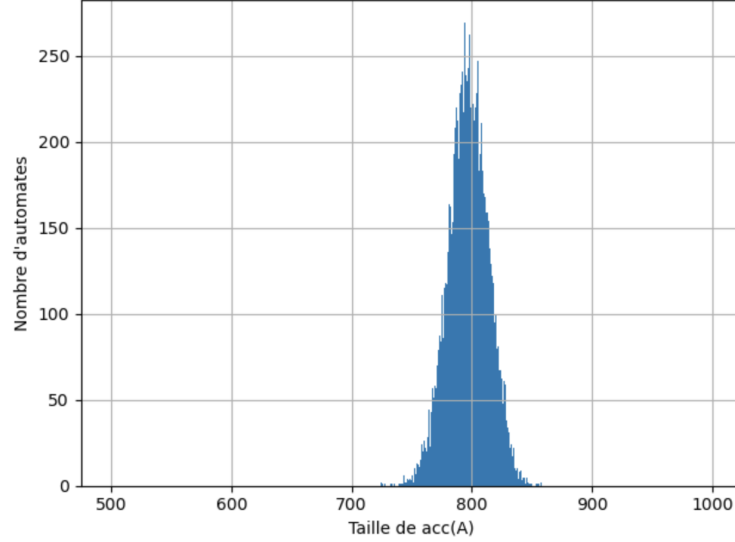


FIGURE 7 – Étude expérimentale de la taille de la partie accessible d'un automate déterministe et complet aléatoire. Pour un alphabet à deux lettres, on a calculé la taille de la partie accessible de 10 000 automates générés aléatoirement et uniformément, et représenté combien de fois chaque taille a été atteinte.

Le résultat suivant montre que se restreindre à la partie accessible n'introduit pas de biais dans les probabilités.

Théorème 5 Soient $1 \leq m \leq n$ des entiers. Soient $\mathcal{B}_1$ et $\mathcal{B}_2$ deux automates accessibles et complets avec m états, et soit $\mathcal{A}$ un automate déterministe et complet aléatoire uniforme avec n états. Les probabilités que $\text{acc}(\mathcal{A}) = \mathcal{B}_1$ et que $\text{acc}(\mathcal{A}) = \mathcal{B}_2$ sont égales.

On en déduit un algorithme de type Las Vegas pour générer aléatoirement des automates déterministes complets et accessibles :

Algorithme 2 : Algorithme de génération d'automates accessibles

Entrée : un entier $n \geq 1$ et un alphabet Σ

Sortie : un automate déterministe complet et accessible uniforme de taille n

1 Calculer la solution λ du Théorème 4

2 **répéter**

3 $\mathcal{A} \leftarrow$ automate déterministe aléatoire uniforme avec $\lceil \frac{n}{\lambda} \rceil$ états

4 $\mathcal{B} \leftarrow \text{acc}(\mathcal{A})$

5 **jusqu'à** $\mathcal{B}$ possède n états

6 **renvoyer** $\mathcal{B}$

FIGURE 8 – Un algorithme de type Las Vegas pour la génération aléatoire d'automates déterministes complets et accessibles avec n états.

Cet algorithme est de complexité moyenne $\mathcal{O}(n\sqrt{n})$. Si on s'autorise une marge d'erreur sur la taille, par exemple qu'elle soit entre $(1 - \epsilon)n$ et $(1 + \epsilon)n$ pour un $\epsilon > 0$ donné, l'algorithme devient linéaire en moyenne.

Un dernier résultat nous permet de nous assurer que les langages obtenus ne sont pas dégénérés comme dans le cas des expressions régulières. On dit qu'un automate déterministe et complet est *minimal* s'il n'existe pas d'automate déterministe et complet avec strictement moins d'états qui reconnaît le même langage. Le résultat suivant est valable dès que $|\Sigma| \geq 2$:

Théorème 6 *La probabilité qu'un automate déterministe complet et accessible avec n états pris uniformément au hasard soit minimal tend vers une constante $c > 0$ quand n tend vers l'infini.*

Pistes de réflexion pour l'exposé

1. Expliquer pourquoi l'algorithme de Fisher-Yates renvoie bien une permutation uniforme de $\{0, \dots, n-1\}$.
2. Étudier expérimentalement la complexité en moyenne d'un autre algorithme de tri.
3. Étudier expérimentalement quel impact a le nombre de valeurs différentes dans l'entrée sur la complexité en moyenne de QUICKSORT.
4. Écrire un programme pour calculer la chaîne de Markov d'ordre 1 associée au fichier `dna_sample` qui contient une partie d'un génome humain. Que peut-on observer ?
5. Écrire un programme pour calculer la chaîne de Markov de n'importe quel ordre associée à un texte. Discuter de comment choisir l'ordre pour une application donnée ?
6. Écrire le programme pour générer aléatoirement des arbres d'expression uniformes de taille n donnée. Étudier expérimentalement la probabilité qu'un tel langage aléatoire contienne le mot vide.
7. Comment calculer efficacement l'arbre réduit d'un arbre d'expression ?
8. Écrire le générateur aléatoire uniforme d'automates déterministes, complets et accessibles avec n états.
9. Reproduire l'expérience de la figure 7 pour des alphabets quelconques, afin d'identifier expérimentalement comment varie λ avec la taille de l'alphabet.
10. Pour ajuster au mieux les modèles aléatoires, on peut utiliser des données accessibles sur internet, et même collecter les données des utilisateurs quand ils se servent de l'algorithme. Discuter de la faisabilité et de l'impact sociétal et économique de telles approches.