FILIERE MP : ENS (PARIS) – ENS LYON – ENS CACHAN

PAGE DE GARDE DU RAPPORT DE TIPE 2012

NOM : DELPEUCH Prénoms : Antonin, Louis

Lycée : Louis le Grand

Classe : MP*3

Ville : Paris

Concours auxquels vous êtes admissible dans la banque inter-ENS :

(Mettre une croix très visible dans la ou les case(s) vous concernant)

ENS Cachan	MP - option MP		MP - option MPI	
	Informatique	X		
ENS Lyon	MP - option MP		MP - option MPI	
	Informatique - option M	X	Informatique - option P	
ENS (Paris)	MP - option MP		MP - option MPI	
	Informatique	X		

Matière dominante du TIPE : Informatique
(mathématiques, informatique ou physique)

Titre du TIPE : Complexité de Kolmogorov et compression de Huffman

Nombre de pages (à porter dans les cases ci-dessous) :

Texte		Illustrations		Bibliographie	
T	5	I	10	B	1

Résumé imprimé (6 lignes) :

On définit la complexité de Kolmogorov, qui mesure l'information contenue dans un mot, et on en étudie quelques propriétés. Cette complexité n'étant pas calculable, on cherche à la majorer. On présente pour cela l'algorithme de compression de Huffman. Pour justifier les objectifs de cet algorithme, on introduit l'inégalité de Kraft, dont les applications à la théorie algorithmique de l'information sont par ailleurs fructueuses. Enfin, on illustre ces concepts par l'étude de la compression de certains préfixes du mot de Thue-Morse.

A PARIS, le 05/06/2012
Signature du (de la) candidat(e)

Signature du professeur responsable de la classe préparatoire dans la discipline

D. CARON

Cachet de l'établissement

Lycée Louis le Grand
075 0655 E
123, rue St Jacques
75231 PARIS CEDEX 05
Tél. 01 44 32 82 00

LYCEE LOUIS LE GRAND - Le Proviseur

Complexité de Kolmogorov et compression de Huffman

Antonin Delpeuch

14 juin 2012

Introduction

Un paradoxe, dit de Berry, est de définir un entier par « le plus petit entier naturel non descriptible en français par une expression de dix-huit mots ou moins. »

Peut-on définir la longueur de la plus courte description d'un objet ? Quelles en sont les propriétés ? Ces idées doivent bien sûr être formalisées. Pour cela, utilise la notion de fonction *calculable* (ou *semi-calculable* pour une fonction partielle), de Σ^* dans lui-même, où $\Sigma = \{0, 1\}$. On note $Dom(f)$ le domaine de f. Si $x \in Dom(f)$, on note $f(x)\downarrow$, et $f(x)\uparrow$ sinon.

Dans tout ce qui suit, les mots (et donc les programmes) sont écrits sur l'alphabet binaire, sauf mention du contraire.

1 Complexité de Kolmogorov

1.1 Définition

Il existe une fonction calculable $\mathcal{U}$ telle que pour toute fonction semi-calculable f il existe $\rho_f \in \Sigma^*$ tel que pour tout $x \in Dom(f)$, $\mathcal{U}(\rho_f \cdot x) = f(x)$. $\mathcal{U}$ agit en fait comme un interpréteur universel, et ρ_f représente un code source de f pour cet interpréteur.

Définition 1. *La complexité de Kolmogorov d'un mot ω de Σ^*, notée $C(\omega)$, est la taille du plus petit programme qui génère ω.*

$$\begin{array}{rccl} C: & \Sigma^* & \longrightarrow & \mathbb{N} \\ & \omega & \longmapsto & min\{|s|, s \in \Sigma^*, \mathcal{U}(s) = \omega\} \end{array}$$

On appelle *programme minimal* d'un mot ω un programme de taille $C(\omega)$ qui génère ω. On note $\widehat{\omega}$ l'un d'entre eux.

C est bien définie car pour tout $x \in \Sigma^*$, il existe un programme qui génère x : l'identité de Σ^* est calculable, donc $\rho_{id} \cdot x$ convient.

1.2 Rareté des mots à faible complexité

Propriété 1. *Il y a au plus $2^{n-p+1} - 1$ mots w de taille n tels que $C(w) \leq n - p$.*

Démonstration. Soit $n \in \mathbb{N}$. Considérons l'ensemble F des mots de Σ^n dont un programme minimal est de taille inférieure à $n - p$.

F s'injecte dans l'ensemble des programmes de taille inférieure ou égale à $n - p$, qui est de cardinal $2^{n-p+1} - 1$. En effet, $\mathcal{U}$ associe à tout programme un unique mot. Donc $\text{Card}(F) \leq 2^{n-p+1} - 1$. □

Les mots qui peuvent être compressés de manière significative sont donc rares. Cela renforce l'idée qu'un mot *pris au hasard* est un mot qui a une complexité de Kolmogorov forte. On peut d'ailleurs utiliser le même argument (avec $p = 1$) pour montrer la proposition suivante.

Propriété 2. *Pour tout n, il existe $\omega \in \Sigma^*$ tel que $C(\omega) \geq n$.*

1.3 Des inégalités vérifiées par C

Propriété 3. $C(\omega) \leq |\omega| + O(1)$

On a obtenu l'inégalité précédente en justifiant l'existence d'un programme générateur dans la définition de la complexité.

Propriété 4. $C(\omega\omega) \leq C(\omega) + O(1)$

Démonstration. La fonction $\omega \mapsto \omega\omega$ est calculable. On la compose avec une fonction générant ω, le codage en résultant ne diffère que d'une constante indépendante de ω. □

Propriété 5. $C(\widehat{\omega}) = C(\omega, C(\omega)) + O(1)$

Démonstration. Si on connait *un* $\widehat{\omega}$, on obtient $C(w) = |\widehat{\omega}|$ et $\omega = \mathcal{U}(\widehat{\omega})$. Donc $C(\omega, C(\omega)) \leq C(\widehat{\omega}) + O(1)$. Réciproquement, si on connait ω et $C(\omega)$, on obtient un $\widehat{\omega}$ en exécutant en parallèle tous les programmes de longueur $C(\omega)$ (il y en a un nombre fini) et en choisissant le premier qui s'arrête en renvoyant ω. D'où la deuxième inégalité. □

Enfin, l'écriture binaire des entiers naturels donne le résultat suivant :

Propriété 6. $C(n) \leq \log n + O(1)$

2 Inégalité de Kraft

2.1 Énoncé

Une idée naturelle pour décrire un mot de manière concise est de remplacer les facteurs les plus fréquents par des codes plus courts. Fixons donc n mots m_i, qu'on veut représenter chacun par un code binaire de longueur c_i. L'inégalité de Kraft donne une condition nécessaire et suffisante sur les c_i pour qu'il existe des $x_i \in \Sigma^*$ tels que $|x_i| = c_i$ et tels qu'il existe au plus une manière de décoder tout mot, c'est à dire que la fonction

$$\begin{array}{rccc} \phi : & \{m_1, \ldots, m_n\}^* & \longrightarrow & \Sigma^* \\ & m_{i_1} \ldots m_{i_p} & \longmapsto & x_{i_1} \ldots x_{i_p} \end{array}$$

est injective.

Théorème 1. *(Inégalité de Kraft, ou de McMillan) Il existe un code $(x_i)_{i\geq 0}$ uniquement décodable et vérifiant $|x_i| = c_i$ pour tout i si et seulement si*

$$\sum_i 2^{-c_i} \leq 1$$

Dans le cas général d'un alphabet de cardinal $D \geq 1$, le théorème reste vrai en remplaçant 2 par D.

Démonstration. On montre que l'inégalité est nécessaire. Pour l'autre sens, la preuve est celle du théorème 2.

On suppose pour commencer que la famille (c_i) est finie : $1 \leq i \leq N$. Soit $u_i = |\{1 \leq j \leq N, c_j = i\}|$ le nombre de mots de taille i. On pose $f(x) = \sum_{i\geq 0} u_i x^i$.

$$f(x)^n = \sum_{k\geq 0} \left(\sum_{c_{i_1} + \ldots + c_{i_n} = k} u_{i_1} \ldots u_{i_n} \right) x^k$$

Or le coefficient devant x^k est le nombre de découpages possibles d'un mot de k lettres en n mots. Plus précisément, posons pour $p \in \mathbb{N}$

$$\begin{array}{rccc} \phi_p : & \{m_1, \ldots, m_n\}^p & \longrightarrow & \Sigma^* \\ & m_{i_1}, \ldots, m_{i_p} & \longmapsto & x_{i_1} \ldots x_{i_p} \end{array}$$

Le coefficient devant x^k est $\left|\phi_n^{-1}(\Sigma^k)\right| \leq 2^k$ car ϕ, donc ϕ_n, est injective. De plus, la famille (c_i) étant finie, les u_i sont nuls à partir d'un certain rang P_n.

$$\left(f\left(\frac{1}{2}\right)\right)^n \leq \sum_{k=0}^{P_n} 2^k \left(\frac{1}{2}\right)^k = P_n + 1$$

Cette égalité étant vraie pour tout $n \geq 0$, et ayant $P_n = O(n)$, par croissances comparées, $f(\frac{1}{2}) \leq 1$. Donc pour tout $N \geq 0$,

$$\sum_{i=0}^{N} u_i 2^{-i} \leq 1$$

D'où la convergence de la série et la majoration cherchée. □

2.2 Complexité de Kolmogorov *préfixe*

On reprend la définition de la première partie en imposant que le domaine des fonctions calculables considérées soit préfixe (*prefix-free*). On obtient alors des propriétés analogues à celle du paragraphe 1.3, mais pas identiques.

On note encore $\mathcal{U}$ une machine universelle et la complexité de Kolmogorov préfixe est notée K. Cette nouvelle complexité vérifie une sorte de sous-additivité, puisque la concaténation de deux programmes n'est plus ambiguë.

Propriété 7. *Pour tous mots ω et τ, $K(\omega\tau) \leq K(\omega) + K(\tau) + O(1)$.*

2.3 Théorème KC

Le théorème KC (inégalité de Kraft, version Calculable) donne un moyen de calculer une suite (x_i) dont l'inégalité de Kraft affirme l'existence.

Théorème 2. *Soit (m_i) suite de mots et (c_i) suite d'entiers vérifiant $\sum_i 2^{-c_i} \leq 1$. Il existe une suite calculable (x_i) de mots de tailles (c_i) et une fonction partielle calculable telle que $f(x_i) = m_i$ pour tout i.*

La preuve ci-dessous, proposée en [4], est assez lourde : on l'illustre par la figure 3.

Démonstration. Il suffit de donner un moyen effectif de construire des x_i vérifiant $|x_i| = c_i$. Pour tout n, on note $\sigma^n = \sigma_1^n \dots \sigma_m^n$ le mot binaire tel que $0, \sigma_1^n \dots \sigma_m^n = 1 - \sum_{j \leq n} 2^{-c_j}$. On construit x_n par récurrence, en garantissant à chaque étape n la propriété suivante : pour tout m tel que $x_m^n = 1$ il existe un mot μ_m^n de longueur m, de sorte que $S_n = \{x_i : i \leq n\} \cup \{\mu_m^n : \sigma_m^n = 1\}$ est préfixe.

On pose $x_0 = 0^{d_0}$. On a $\sigma^0 = 11 \dots 1 = 1^{d_0}$. Poser $\mu_m^0 = 0^{m-1}1$ définit bien un ensemble S_0 préfixe.

Supposons S_n défini.

Si $\sigma_{c_{n+1}}^n = 1$, alors σ^{n+1} est identique à σ^n hormis $\sigma_{d_{n+1}}^{n+1}$ qui vaut 0. On peut donc poser $x_{n+1} = \mu_{c_{n+1}}^n$ et $\mu_m^{n+1} = \mu_m^n$ pour tous les $m \neq c_{n+1}$. L'hypothèse de récurrence est vérifiée au rang $n+1$.

Sinon, soit $j < d_{n+1}$ maximal tel que $\sigma_j^n = 1$. Un tel j existe car sinon $1 - \sum_{j \leq n} 2^{-c_j} < 2^{-c_{n+1}}$ et donc $\sum_{j \leq n+1} 2^{-c_j} > 1$. σ^{n+1} ne diffère de σ^n qu'aux positions $j, j+1, \dots, c_{n+1}$ où σ^{n+1} vaut respectivement $0, 1, \dots, 1$. On pose alors $x_{n+1} = \mu_j^n 0^{c_{n+1}-j}$. Pour $m < j$ ou $m > c_{n+1}$, on pose $\mu_m^{n+1} = \mu_m^n$. Pour $j < m \leq c_{n+1}$, on pose $\mu_m^{n+1} = \mu_j^n 0^{m-j-1}1$. Vérifions que ces choix engendrent un S_{n+1} valide : on n'a fait que substituer à μ_j^n un ensemble de mots commençant par μ_j^n et n'étant pas préfixes les uns des autres, donc S_{n+1} est encore préfixe. □

Une première application de ce théorème est de montrer l'existence de codages des entiers asymptotiquement économes.

Propriété 8. *Pour tout $p \in \mathbb{N}$ et tout $\epsilon > 0$, $K(n) \leq \log n + \log\log n + \dots + (1+\epsilon)\log^p n + O(1)$.*

Démonstration. Soit $L > 0$.

$$\sum_{n \geq q} 2^{-\log n - \ldots - (1+\epsilon)\log^p n - L} = 2^{-L} \sum_{n \geq q} \frac{1}{n \log n \ldots (\log^{p-1} n)^{1+\epsilon}}$$

Cette dernière série, dite de Bertrand, converge pour tout $\epsilon > 0$. Il ne reste donc qu'à choisir L assez grand pour que sa somme (complétée par des premiers termes pour les $n < q$) soit inférieure à 1. Le théorème KC achève la preuve. □

Quelles sont les fonctions calculables majorant la complexité de Kolmogorov ? La caractérisation qui suit est une deuxième application du théorème KC, qui demande un peu plus de travail.

Propriété 9. *Soit f une fonction calculable de $\mathbb{N}$ dans $\mathbb{N}$. S'équivalent :*

i. $K(n) \leq f(n) + O(1)$

ii. $\sum_n 2^{-f(n)} < \infty$

3 Compression de Huffman

3.1 Principe de la compression

Soit t un mot sur l'*alphabet* $\{0,1\}^n$. On cherche un morphisme ϕ par lequel le mot t sera encodé en un minimum de bits. On cherche donc à minimiser

$$\sum_{c \in \{0,1\}^n} |t|_c \cdot |\phi(c)|$$

où $|t|_c$ désigne le nombre d'occurrences de c dans t.

On veut d'autre part que ϕ soit injectif : les $(|\phi(c)|)_c$ vérifient alors l'inégalité de Kraft, ce qui a pour conséqence qu'on peut imposer sans augmenter le coût ci-dessus que le codage soit préfixe, d'après la construction utilisée dans la preuve du théorème 2.

3.2 Algorithme de Huffman

L'algorithme de Huffman est un algorithme glouton qui crée un arbre définissant un codage optimal pour le mot t. À l'initialisation, on crée une feuille par caractère présent dans le texte et on associe à chaque feuille le nombre d'occurences du caractère dans t. On obtient ainsi une forêt de feuilles.

Tant que la forêt n'est pas réduite à un seul arbre, on sélectionne deux arbres de fréquences minimales, et on les remplace par un seul arbre dont la racine a deux fils : les racines de ces deux arbres. On attribue à ce nouvel arbre un nombre d'occurrences égal à la somme des nombre d'occurrences de ses deux fils. Ce procédé est représenté sur la figure 1.

À la fin de l'algorithme, on obtient un arbre définissant un codage qu'on peut prouver optimal. [2]

3.3 Borne supérieure pour la complexité de Kolmogorov

On dispose alors d'un algorithme qui, pour tout mot dont on veut majorer la complexité de Kolmogorov, génère un programme affichant ce mot, d'une taille potentiellement plus courte que la solution qui consiste à écrire intégralement le mot en question dans le code. Plus précisément, on compresse le mot avec l'algorithme de Huffman, et on assemble dans un même programme l'arbre qui représente le codage et le texte compressé. Le programme restitue alors le mot original.

Ce procédé se révèle en général inutile, mais c'est sans surprise puisque la plupart des mots ont une complexité de Kolmogorov très proche de leur longueur. Et le programme généré est même plus long que la solution triviale, bien souvent : même si le codage est optimal, il faut ajouter l'arbre de Huffman et une certaine quantité de code pour opérer la décompression, ce qui est coûteux.

4 Étude d'un exemple

4.1 Définition

Définition 2. *On note $(t_n) \in \{0,1\}^{\mathbb{N}}$ la suite de Thue-Morse définie par :*

$$t_n = \begin{cases} 0 & \text{si le nombre de chiffres 1 dans l'écriture binaire de } n \text{ est pair} \\ 1 & \text{sinon} \end{cases}$$

Cette suite définit un mot binaire infini. On note T_n le préfixe de taille 2^n de ce mot : $T_n = t_0 \cdot t_1 \cdot t_2 \cdots t_{2^n-1}$.

On étudie la complexité de Kolmogorov de T_n et de sa croissance asymptotique quand on fait varier n. La définition précédente impose qu'elle soit faible : en effet, on peut définir T_n en donnant n et la définition de T_k, à la manière de la définition précédente. Par conséquent :

$$K(T_n) \leq K(n) + O(1)$$

Cette majoration est d'ailleurs valable pour toute fonction calculable.

4.2 Compression du mot de Thue-Morse

On compare le majorant de $K_M(T_n)$ obtenu dans la section précédente avec la longueur du programme obtenu par compression de T_n. La suite de Thue-Morse possède des propriétés qui rendent sa compression plus efficace qu'en moyenne :

Propriété 10. *Pour tout $a \in \{0,1\}$ et $\omega \in \{0,1\}^*$, $a\omega a\omega a$ n'est pas facteur de t.*

Cette propriété a des conséquences pour la compression des préfixes : un certain nombre de caractères n'apparaissent pas, ce qui réduit la taille de l'arbre de Huffman : la compression reste relativement efficace même quand on choisit une taille de caractères qui n'est pas une puissance de deux. La figure 2 montre différents arbres de Huffman obtenus sur le préfixe de taille 256.

4.3 Programmes générateurs

La compression de Huffman nous donne des programmes générateurs pour les préfixes du mot de Thue-Morse, et nous avons vu que si la taille du préfixe et celle des caractères est choisie judicieusement, l'arbre de Huffman peut être particulièrement petit, et donc la compression très efficace. Pourtant, même pour un cas aussi favorable, le programme qui en résulte est loin d'être minimal, du moins asymptotiquement.

En effet, si on fixe la taille de caractère, la taille du programme généré par compression reste linéaire en fonction du mot à compresser. Un programme traduisant la définition du mot croit beaucoup plus lentement, en $O(\log n)$ environ.

J'ai essayé d'écrire les programmes générateurs mentionnés (ceux issus de l'algorithme de Huffman et ceux issus de la définition) dans deux langages différents : le langage `bf`, proche du formalisme d'une machine de Turing, et le *Lambda Calcul Binaire*, codage préfixe de λ-termes. La place manque pour expliquer le fonctionnement de ces langages, même succintement, mais en guise d'illustration voici quelques programmes :

```
.>>+.>+>++<[-<[->+<]>[<-[++>>--]+.>>.>>++<-[+<<-]]<[->+<]>-[<--[++>>--].>>+.>>++<+[-<<+]]>>+]
```

Ce programme en `bf` écrit le mot de Thue-Morse (il ne termine donc pas).

```
# λ (λ 1 1) (λ (λ 1 1) (λ λ λ λ 1 3 (λ 1 (λ λ 6 1 2) (3 (5 5)))) (λ λ 2) (1 1 (λ λ 1)))
```

Ce λ-terme est l'équivalent du programme précédent en λ-calcul.

Conclusion

L'intérêt des méthodes de compression statistique dans le cadre de l'étude de la complexité de Kolmogorov semble donc limité. Il faudrait disposer de compresseurs exploitant des redondances beaucoup plus complexes que de simples observations statistiques. Pourtant, des recherches récentes ont montré que l'utilisation de compresseurs classiques pour approximer la complexité de Kolmogorov permet, par exemple, de mesurer la *similarité* entre deux mots. Ces techniques trouvent des applications dans des algorithmes de classification non supervisée.

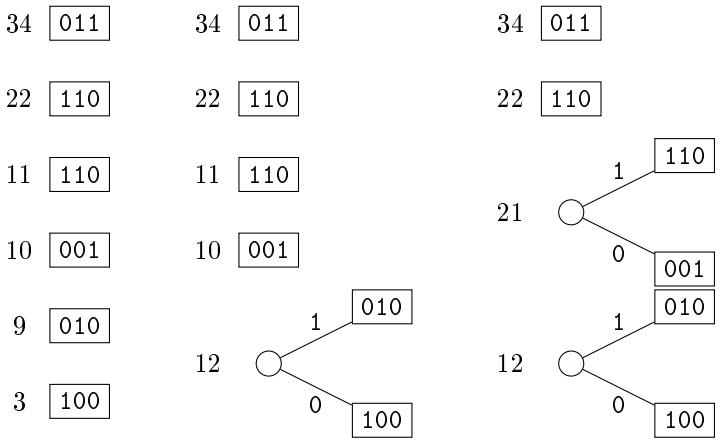
Références

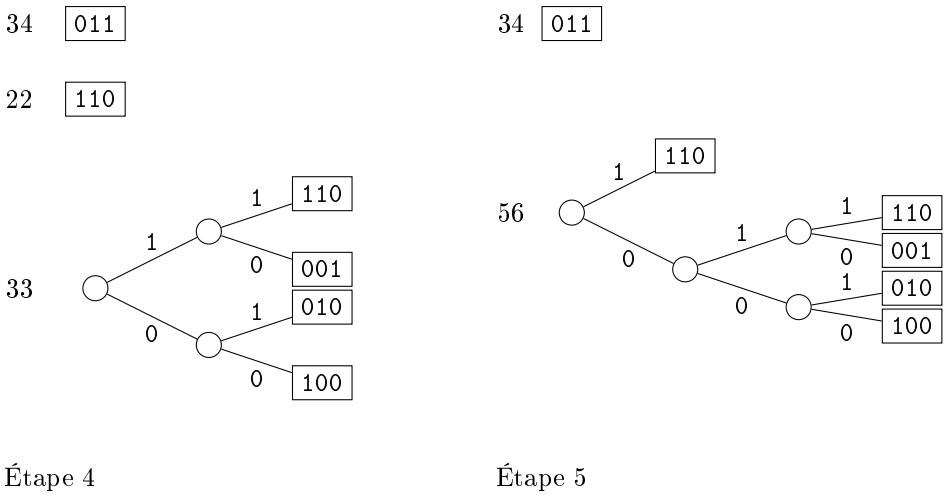
[1] Olivier Carton. *Langages formels, calculabilité et complexité.* Vuibert, 2008.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction à l'algorithmique.* Dunod, 2002.

[3] Jean-Paul Delahaye. *Information, complexité et hasard.* Hermès, 1999.

[4] Rod Downey and Denis Hirschfeldt. *Algorithmic Randomness and Complexity.* Springer Verlag, 2010.





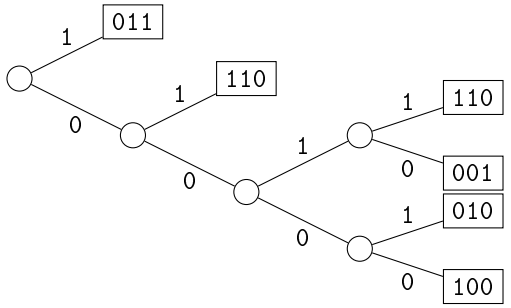
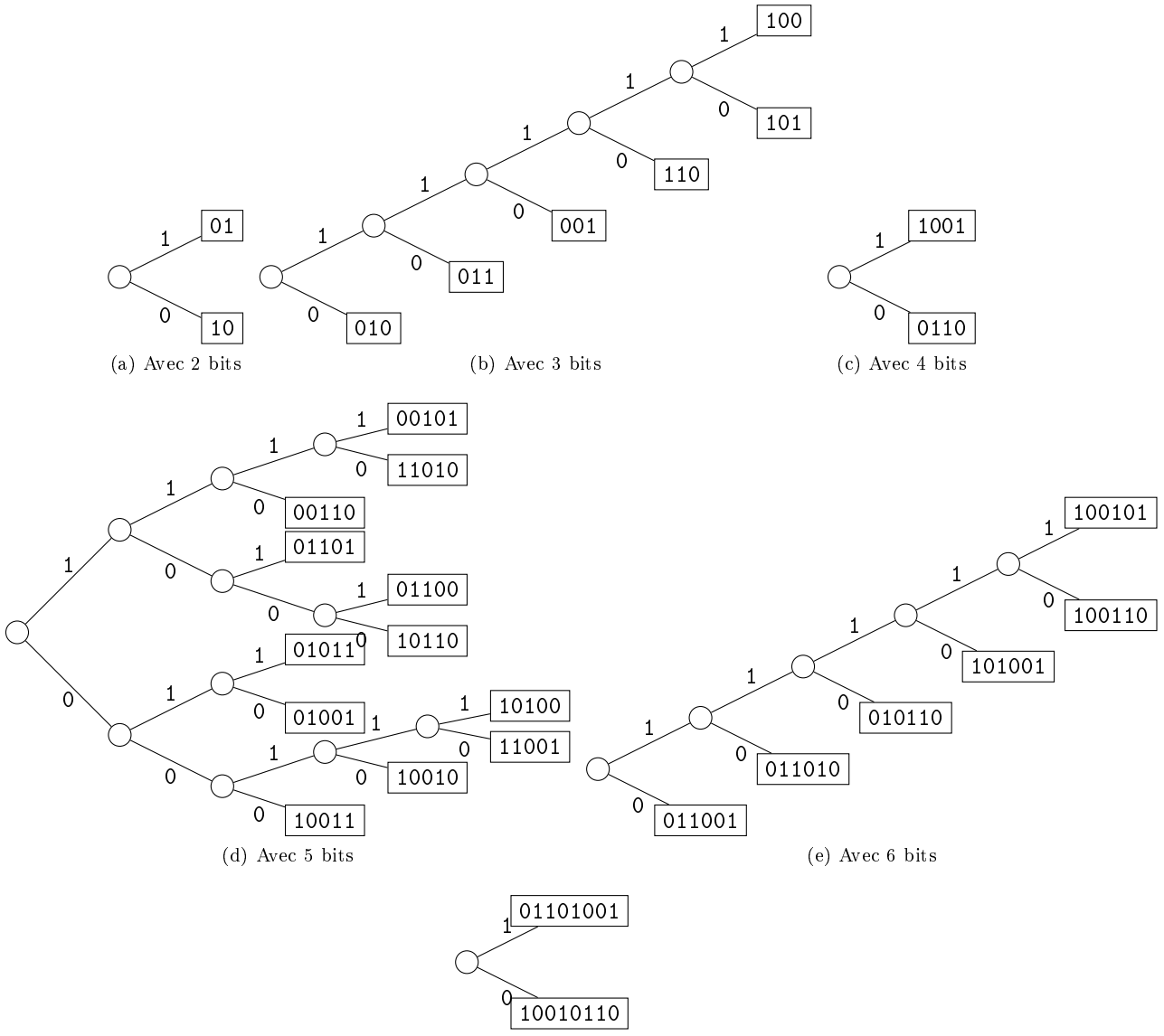


FIGURE 1 – Le déroulement de l'algorithme de Huffman



(a) Avec 2 bits (b) Avec 3 bits (c) Avec 4 bits

(d) Avec 5 bits (e) Avec 6 bits

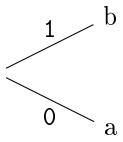
(f) Avec 8 bits

```
0110100110010110 1001011001101001 1001011001101001 0110100110010110
1001011001101001 0110100110010110 0110100110010110 1001011001101001
1001011001101001 0110100110010110 0110100110010110 1001011001101001
0110100110010110 1001011001101001 1001011001101001 0110100110010110
```

(g) Le mot donné à compresser : le 256è préfixe du mot de Thue-Morse

FIGURE 2 – Les arbres de Huffman obtenus en choisissant différentes tailles de caractère

Dans cette figure, on omet les étiquettes :



On détaille les trois premières étapes de l'algorithme pour $c_0 = 3$, $c_1 = 1$, $c_2 = 5$
qui correspondent aux trois cas décrits dans la preuve.

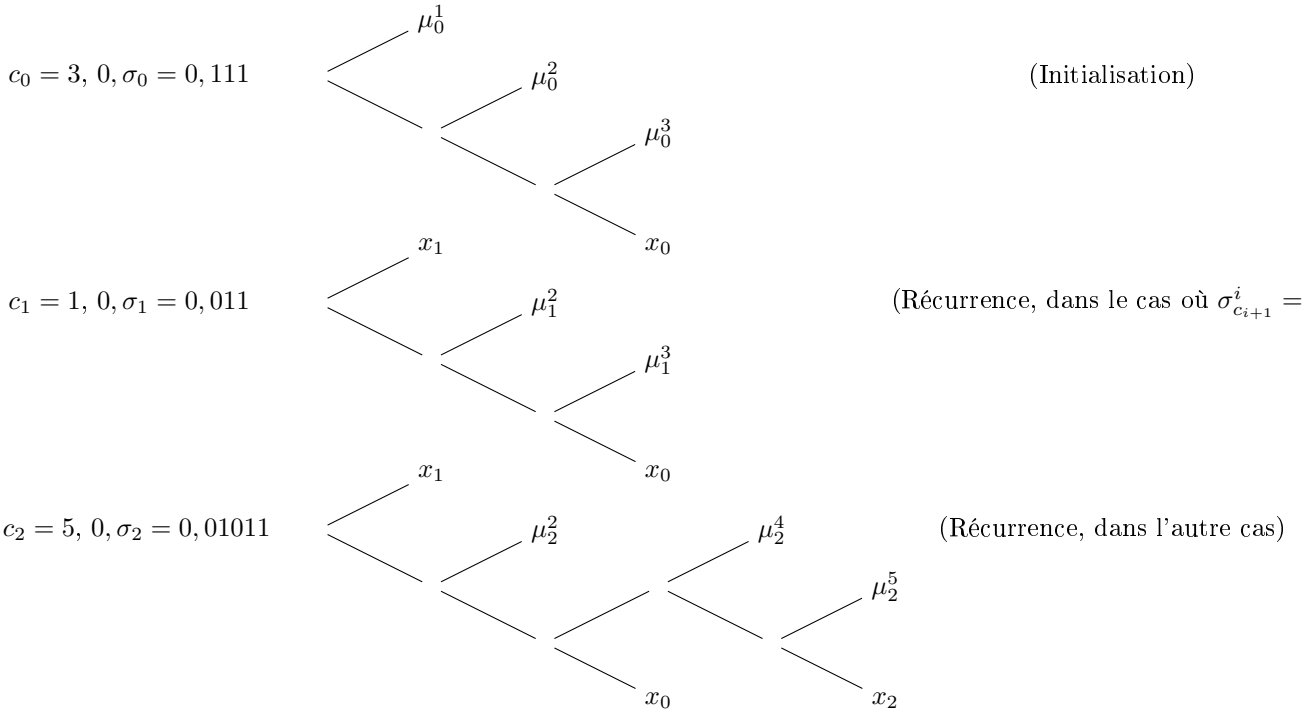


FIGURE 3 – Construction par étapes de l'arbre des x_i

A Code source

A.1 Module `Compression`

```
(*************************************************************
 * OUTILS GÉNÉRIQUES POUR LA COMPRESSION AVEC UN CODE PRÉFIXE *
 *************************************************************)

open Utilitaires

 (* Un mot est une liste de nombres, a priori. En pratique, on
  * travaille volontiers avec un alphabet binaire.  *)
type mot = int list;;

 (* Type de l'arbre de Huffman *)
type codage =
  | Code of mot
  | Noeud of codage * codage;;

 (* Ordre sur les arbres pondérés : nécessaire pour la gestion du tas. *)
let ordre_tas a b = (fst a) > (fst b);;

 (* Taille initiale de la table de hachage en fonction de la taille des caractères :
  * dans le pire des cas, 2^p caractères.
  * En pratique, moins sur des textes particuliers. *)
let taille_table_de_hachage p = p ;;

 (* Décoder un texte compressé en utilisant l'arbre de Huffman passé en paramètre
  * *)
let rec decoder arbre pos lst = match lst,pos with
  | ([],Noeud(_)) -> []
  | (_,Code(m)) -> concat_listes (decoder arbre arbre lst) m
  | (0::t,Noeud(p,q)) -> decoder arbre p t
  | (1::t,Noeud(p,q)) -> decoder arbre q t
  | (_::t,n) -> decoder arbre n t (* Pour que le filtrage soit complet *);;

 (* Coupe le texte en mots de la taille donnée *)
let rec decouper_mots taille_mot restant accu texte = match texte,restant with
  | (_,0)    -> accu::(decouper_mots taille_mot taille_mot [] texte)
  | ([],n)   -> []
  | (h::t,n) -> decouper_mots taille_mot (n-1) (h::accu) t ;;

 (* Construit un tableau indexé par les mots donnant le codage compressé de ces
  * mots. *)
let construire_traduction arbre taille_mot =
   (* La table de compression, notée trad (qui associe à chaque caractère son codage)
    * est une table de hachage *)
  let trad = Hashtbl.create (taille_table_de_hachage taille_mot) in

   (* On réalise un parcours en profondeur *)
  let rec parcourir_arbre trad accu = function
    | Code(m) -> Hashtbl.add trad m (rev accu)
    | Noeud(p,q) -> (parcourir_arbre trad (0::accu) p);
                    (parcourir_arbre trad (1::accu) q) in
    (parcourir_arbre trad [] arbre); trad ;;

```

```
 (* Compression d'un texte à partir d'un arbre *)
let compresser arbre taille_mot texte =
  let trad = (construire_traduction arbre taille_mot) in

    (* Prend une liste de caractères et renvoie la liste de leurs
     * équivalents compressés *)
  let rec compresser_mots trad = map (fun h -> Hashtbl.find trad h) in

    (* Concatène les mots pour obtenir le texte compressé *)
  let concatene_mots = it_list (fun a b -> a @ b) [] in

    (* On assemble le tout... *)
  concatene_mots (compresser_mots trad (decouper_mots taille_mot taille_mot [] texte));;
```

A.2 Module Tas

```
open Compression

(* Structure retenue pour le tas :
 * un tableau dont la première case est le nombre d'éléments stockés dans le tas
 * et les éléments qui suivent après, selon la hiérarchie suivante : *)

let pere i = i/2;;
let gauche i = 2*i;;
let droite i = 2*i + 1;;

let taille_tas t = (fst t.(0));;
let incr_taille_tas t =
  t.(0) <- ((taille_tas t)+1,(snd t.(0)));;
let decr_taille_tas t =
  t.(0) <- ((taille_tas t)-1,(snd t.(0)));;

(* L'ordre avec lequel le tas est trié est donné par la fonction
 * ordre_tas qui doit être définie avant l'inclusion de ce fichier
 * Signification : ordre_tas a b renvoie true si a < b *)

(* Échange deux éléments dans un tableau *)
let echange tab i j =
  let temp = tab.(i) in
    tab.(i) <- tab.(j);
    tab.(j) <- temp;;

(* Entasse un élément dont les sous-arbres sont valides *)
let rec entasser_haut tab = function
  | 1 -> ()
  | n -> if (ordre_tas tab.(pere n) tab.(n))  then
         begin
           echange tab n (pere n);
           entasser_haut tab (pere n)
         end;;

(* Entasse un élément dont le père est valide *)
let rec entasser_bas tab elem =
  let max = ref elem in
  if ((gauche elem) <= (taille_tas tab) && (ordre_tas tab.(!max) tab.(gauche elem))) then
    max := (gauche elem);
```

```
  if ((droite elem) <= (taille_tas tab) && (ordre_tas tab.(!max) tab.(droite elem))) then
    max := (droite elem);
  if !max <> elem then
  begin
    echange tab !max elem;
    entasser_bas tab !max
  end;;


(* Insère un élément dans un tas *)
let insere_dans_tas tab elem =
  if (Array.length tab) <= (taille_tas tab)+1 then
    failwith "Tas plein";
  incr_taille_tas tab;
  tab.(taille_tas tab) <- elem;;

(* Supprime le minimum du tas *)
let pop_minimum tab =
  if (taille_tas tab) = 0 then
    failwith "Tas vide";
  let mini = tab.(1) in
  tab.(1) <- tab.(taille_tas tab);
  decr_taille_tas tab;
  entasser_bas tab 1;
  mini;;
```

A.3 Module Huffman

```
 (* Définit des fonctions classiques d'arithmétique ou de manipulation de listes *)
open Utilitaires
 (* Définit le codage et le décodage d'un texte avec un arbre de codage préfixe donné *)
open Compression

(***********************************
 * GÉNÉRATION DE L'ARBRE DE HUFFMAN *
 ***********************************)

 (* Affichage du tableau du nombre d'occurences de chaque caractère *)
let afficher_occurences compte =
  let afficher_paire w occur =
    print_list "" " : " w;
    print_int occur;
    print_newline () in

   Hashtbl.iter afficher_paire compte ;;

  (* Incrémente la case d'une table de hachage correspondant à la clé w *)
let incr_occur tbl w =
  let oldval =
    try
      Hashtbl.find tbl w
    with Not_found -> 0 in

    if oldval <> 0 then
      Hashtbl.remove tbl w;
    Hashtbl.add tbl w (oldval + 1);;
```

```
29
30  (* Compte les occurences de chaque mot de taille donnée dans le texte
31   * (toujours avec notre définition particulière d'une occurence d'un
32   * mot : il faut que le mot commence à une position multiple de
33   * taille_mot) *)
34  let compter_occurences taille_mot texte =
35    let compte = Hashtbl.create (taille_table_de_hachage taille_mot) in
36    let rec compte_entiers = function
37      | [] -> ()
38      | h::t -> incr_occur compte (h);
39                compte_entiers t in
40      compte_entiers (decouper_mots taille_mot taille_mot [] texte);
41
42     afficher_occurences compte;
43      compte;;
44
45
46  (* Définit les fonctions de manipulation d'un tas *)
47  open Tas
48
49  (* Construit le tas initial des feuilles *)
50  let construire_tas taille_mot compte =
51    let tab = Array.make ((Hashtbl.length compte)+1) (0,Code([])) in
52    tab.(0) <- (0,Code([]));
53
54     (* Copie des occurences non nulles dans le tas *)
55    let ajouter_mot_dans_tas mot occur =
56      incr_taille_tas tab;
57      tab.(taille_tas tab) <- (occur,Code(mot) ) in
58
59     Hashtbl.iter ajouter_mot_dans_tas compte ;
60
61     (* Création d'un tas valide *)
62    for i = 1 to (taille_tas tab) do
63     entasser_haut tab i;
64    done;
65
66     tab;;
67
68  (* Fusionne les noeuds tant qu'il y en a plusieurs *)
69  let rec fusionner_noeuds tab =
70    if (taille_tas tab) > 1 then
71    begin
72      let (p1,n1) = (pop_minimum tab) in
73      let (p2,n2) = (pop_minimum tab) in
74      insere_dans_tas tab (p1+p2, Noeud(n1,n2));
75      fusionner_noeuds tab
76    end
77    else snd tab.(1);;
78
79
80  (* Mise en commun : génération de l'arbre de Huffman d'un texte, pour une
81   * taille de mot particulière *)
82  let arbre_huffman taille_mot texte =
83    let etape1 = compter_occurences taille_mot texte in
84    let etape2 = construire_tas taille_mot etape1 in
```

```
  let etape3 = fusionner_noeuds etape2 in
  etape3;;

(*******************************
 * GÉNÉRATION DU DÉCODEUR EN bf *
 *******************************)

let bf_affichage_mot m =
  let rec afficher_mot etat_init restant = match etat_init,restant with
    | (0,[]) -> ""
    | (1,[]) -> "-"
    | (_,[]) -> "[-]" (* Pour que le filtrage soit complet *)
    | (0,0::h) -> strconcat "." (afficher_mot 0 h)
    | (1,1::h) -> strconcat "." (afficher_mot 1 h)
    | (0,1::h) -> strconcat "+." (afficher_mot 1 h)
    | (1,0::h) -> strconcat "-." (afficher_mot 0 h)
    | (_,c::h) -> afficher_mot c h (* Pour que le filtrage soit complet *)
  in
  afficher_mot 0 m;;

let bf_condition est_racine a b =
  let debut = (if est_racine then "" else ",") in
  String.concat "" [debut; ">+<[>>"; b; "<-<-]>[->"; a; "<]<"];;

let decompresseur_bf arbre =
  let rec traduire_arbre est_racine = function
    | Code(m) -> bf_affichage_mot (rev m)
    | Noeud(a,b) -> bf_condition est_racine (traduire_arbre false a) (traduire_arbre false b) in
  String.concat "" [",+[-"; (traduire_arbre true arbre); ",+]"];;
```

A.4 Calcul du mot de Thue-Morse

```
(*******
 * Première méthode de génération du mot de Thue-Morse :
 *
 * Définir le morphisme σ : 0 ⟼ 01 ; 1 ⟼ 10 calculant
 * σ(ω) à partir de ω représenté sous la forme d'une liste.
 *
 * Il suffit alors d'itérer le morphisme jusqu'à avoir le
 * nombre désiré de termes
 ********)

let rec print_list = function
  | [] -> print_newline ()
  | t::q -> print_int t; print_list q ;;

let rec morphisme = function
  | [] -> []
  | 0::q -> 0::(1::(morphisme q))
  | _::q -> 1::(0::(morphisme q)) ;;

let rec iterer x = function
  | 0 -> x
  | n -> morphisme (iterer x (n-1)) ;;

(*******
```

```
 * Autre méthode, avec l'évaluation paresseuse
 *
 * Implémentation de l'évaluation paresseuse
 * largement inspirée de l'article de Pierre Weiss
 * dans la Lettre de Caml no. 2
 ********)

(* Le type glaçon est l'outil de base de l'évaluation paresseuse :
 * Un objet peut être soit connu, soit calculable plus tard en
 * exécutant la fonction ne prenant aucun argument. *)

type 'a glacon =
  | Connu of 'a
  | Gele of (unit -> 'a ) ;;

(* Une liste paresseuse est constituée, comme les listes classiques,
 * de [] et de hd::tl. La différence ici est que tl n'est pas forcément
 * connu, grace au type glaçon. L'avantage est de pouvoir gérer des
 * listes « infinies » : on ne les calcule que jusqu'à un certain rang. *)

type 'a liste_par =
  | Nil
  | Cons of 'a cellule
 and 'a cellule = {hd : 'a; mutable tl : 'a liste_par glacon} ;;

(* La fonction force permet de connaitre un élément de plus de la liste
 * donnée. Elle renvoie la queue de cette liste. *)

let force cel =
  match cel.tl with
    | Connu v -> v
    | Gele f -> let resultat_f = f () in
        cel.tl <- Connu (resultat_f);
        resultat_f ;;

let print_bool = function
  | false -> print_string "0"
  | true -> print_string "1" ;;

(* Un do_list pour liste paresseuse : il faut s'arrêter au bout de
 * n étapes. *)

let rec do_list_par f n = function
  | Nil -> print_newline ()
  | Cons cel ->
      if n = 0 then print_newline ()
      else
        begin
          f cel.hd;
          do_list_par f (n-1) (force cel);
        end ;;

(* Fonction map pour les listes paresseuses *)

let rec map_par f = function
  | Nil -> Nil
```

```
  | Cons cel -> Cons {hd = f cel.hd; tl = Gele (fun () -> map_par f (force cel))} ;;

(* Application : calcul du mot infini de Thue-Morse
 * On utilise la relation de récurrence suivante :
 * - Le premier terme est 0.
 * - Connaissant les 2^n premiers termes, les 2^n suivants
 *   sont les négations des 2^n premiers, dans le même ordre. *)

let rec recopier_en_niant = function
  | Nil -> Nil
  | Cons cel -> Cons {hd = not cel.hd; tl = (match cel.tl with
      | Connu v -> Connu (recopier_en_niant v)
      | Gele _ -> Gele (fun () -> recopier_en_niant tm))}
and
tm = Cons {hd = false; tl = Gele (fun () -> recopier_en_niant tm) } ;;

do_list_par print_bool 36 tm ;;


(* Troisième méthode
 *
 * Généralisable pour obtenir un point fixe de
 * n'importe quel morphisme de la forme
 * σ : a ⟼ aω
 *     b ⟼ σ(b)
 *     ⋮
 *     h ⟼ σ(h)
 * dont le point fixe commençant par a est
 * a ω σ(ω) σ²(ω) ...
 *)

(* Équivalent de tl pour les listes paresseuses *)

let tail_par = function
  | Nil -> Nil
  | Cons cel -> (force cel);;

(* Morphisme du mot de Thue-Morse *)
let rec phi = function
  | Nil -> Nil
  | Cons cel -> let reste = Gele (fun () -> phi (force cel)) in
      match cel.hd with
      | 0 -> Cons {hd = 0; tl = Connu
            (Cons {hd = 1; tl = reste})}
      | _ -> Cons {hd = 1; tl = Connu
            (Cons {hd = 0; tl = reste})} ;;

let rec point_fixe = Cons { hd = 0; tl = Connu
                        (Cons { hd = 1; tl = Gele
                            (fun () -> (phi (tail_par point_fixe))) })
                              };;

do_list_par print_int 36 point_fixe ;;
```