Parsing et codage universel

Aurélien Garivier

16 septembre 2003

Première partie
Introduction

La généralisation des échanges de données binaires au cours des trois dernières décennies a accentué le besoin de techniques efficaces visant à réduire le volume de celles-ci, c'est à dire à les *comprimer*. En particulier, tout un ensemble de données (textes, ADN, etc...) doivent être manipulées sans que leur contenu ne soit altéré, et nous nous intéresserons ici à des algorithmes de codage *sans perte*, c'est à dire sans destruction d'information.

Un tel mécanisme se formalise mathématiquement par une fonction injective de l'ensemble des mots d'un alphabet $\mathcal{A}$ vers l'ensemble des mots d'un autre alphabet $\mathcal{B}$:

$$code : \mathcal{A}^+ \to \mathcal{B}^+$$

A cette fonction est associée une réciproque (éventuellement partielle), c'est à dire une fonction $decode : \mathcal{B}^+ \to \mathcal{A}^+$ vérifiant $\forall x \in \mathcal{A}^+, decode(code(x)) = x$. L'objectif est de constuire $code$ de sorte que $|code(x)|$ soit le plus souvent le plus petit possible. Pour la clarté de l'exposé, nous nous limiterons ici au cas $\mathcal{A} = \mathcal{B} = \{0,1\}$, qui se généralise sans difficulté.

Il est toutefois immédiat de voir que l'on ne peut comprimer efficacement toutes les données : pour tout entier k, il y a moins de 2^k chaînes pour lesquelles $|code(x)| < k$. De fait, l'objectif intuitif est que notre procédure doit être très efficace sur une séquence comme $(001)^{3000}$, mais l'on ne peut en attendre grand'chose pour un fichier contenant les résultats de loto des trois dernières années, bien que ces deux exemples occupent à peu près la même taille avant compression. Apparaît ainsi la nécessité d'introduire une notion de *complexité* $\kappa : \mathcal{A}^+ \to \mathbb{R}^+$, l'objectif étant alors de contruire $code$ afin d'avoir : $\forall x \in \mathcal{A}^+, |code(x)| \approx \kappa(x)$. On appellera *regret* de $code$ par rapport à κ pour x la quantité : $|code(x)| - \kappa(x)$.

Dans le monde informatique, une définition naturelle pour κ émerge : la complexité κ_K dite de Kolmogorov, c'est-à-dire la taille binaire du plus court programme qui génère x. Une idée s'impose alors pour $code$: associer à x, précisément le plus court programme qui le génère. Mais cette approche, trop ambitieuse, n'est pas utilisable, ne serait-ce que parce que la détermination de $\kappa_K(x)$ est un problème *indécidable*.

Dans un article qui marque la naissance de la théorie de l'information, Shannon introduit une autre notion de complexité en faisant l'hypothèse que x est généré par une *source* probabiliste : l'*entropie* de cette source. Celle-ci se prête à une manipulation tant mathématique qu'algorithmique si élégante et efficace qu'elle s'est imposée par la suite. Nous verrons toutefois qu'elle n'est pas sans poser problème pour le cadre que nous nous sommes fixé ici, et qu'elle ne constitue en aucun cas une approche parfaite et définitive.

En effet, dans le cas où aucune notion de source n'est a priori disponible, nous nous proposerons d' associer à x celle qui lui est le mieux "adaptée" dans une famille donnée. Faisant cela, nous utiliserons donc les outils statistiques de théorie de l'information comme un pur outil, sans signification intrinsèque, qui nous permettra de donner des éléments d'étude sur une famille de schémas de compression basés sur le découpage, ou *parsing*, de x.

Nous reviendrons ensuite sur quelques membres de cette famillle : d'abord les fameux algorithmes de Ziv et Lempel ZL77 et ZL78, basés sur le principe du "déjà-vu", et que leurs auteurs ont introduit en même temps qu'une autre notion de complexité restreignant la puissance de calcul du modèle de Kolmogorov en se limitant à des machines à états finis.

Puis les schémas de compression dits *par transformée grammaticale*, formalisés par Kieffer et Yang en 1995, qui ont ré-interprété ZL78 comme un cas particulier de méthode codant x à travers une *grammaire* dont le langage est réduit à ce mot _ méthode qu'ils ont théorisée et dont ils ont donné d'autres exemples expérimentalement fructueux. On sait que les grammaires sont un modèle de calcul "intermédiaire" entre les automates finis et les machines de Turing. Elles permettent en particulier de prendre en compte les phénomènes de type "palindrôme", qui semblent avoir quelque importance dans l'étude du génôme ; en est-il de même pour ces méthodes ?

Nous verrons en tous cas que l'étude classique qui en est faite n'en capte pas toute la puissance, ce qui nous amèra à conclure à la nécessité d'une notion plus adaptée de complexité, qui fait pour l'instant défaut.

Deuxième partie
Complexité entropique et principe d'équipartition asymptotique

1 Le théorème de Shannon-MacMillan-Breiman

Plaçons-nous dans cette partie dans un cadre "classique" de théorie de l'information où est donnée une source de loi $\mathbb{P}$, stationnaire ergodique, d'entropie H_P, et qui génère une suite X d'éléments de $\mathcal{A}$ qui sera pris, pour simplifier la discussion, comme étant l'ensemble $\{0, 1\}$. Comme annoncé, Shannon propose alors de considérer comme complexité par bit pour X la valeur H_P, qui ne dépend que de $\mathbb{P}$ et pas de l'aléa, et qui représente la quantité *moyenne* d'information présente dans une réalisation de la source. Pour une présentation élémentaire de ces notions et une justification de la définition de l'entropie, voir [13] et [8]. Suivant en cela l'habitude, log désignera la fonction logarithme à base 2 (et ln le logarithme néperien).

Notre fonction de codage *code* sera dite *universelle* (au sens de Shannon) si pour toute source stationnaire ergodique d'entropie H, on a :

$$\frac{|code(x)|}{|x|} - H \to 0$$

presque sûrement (donc en probabilités) quand $|x| \to \infty$. Remarquons que la difficulté du codage *universel* tient au fait que *code* n'a a priori aucune connaissance sur la source, et il peut donc paraître miraculeux qu'il puisse faire asymptotiquement aussi bien qu'un procédé ad hoc. Nous verrons que cette généralité a un coût (mesuré par un regret généralement plus grand), mais que par ailleurs elle n'est pas si miraculeuse que cela.

Il a fallu longtemps pour réaliser exactement en quoi les hypothèses sur P étaient pertinentes. Une propriété de P apparaît particulièrement importante pour mener cette étude : l'existence d'un *princicipe d'equipartition asymptotique* (abrégé par le sigle AEP, en anglais), c'est à dire l'existence d'une valeur $\tilde{H}$ telle que :

$$-\frac{1}{n} \log P\left\{X_1^n\right\} \to \tilde{H}$$

avec une convergence presque sûre. Mais le théorème de Shannon-MacMillan-Breiman assure que si P est stationnaire et ergodique, alors P satisfait un AEP avec $\tilde{H} = H$, et par ailleurs cette notion n'est guère plus générale (cf [3] p.20).

Ce théorème amène à considérer l'ensemble $\mathcal{T}_\epsilon^n$ des séquences x_1^n dites *ϵ-typiques* - ce sont les éléments vérifiant :

$$2^{-n(H+\epsilon)} \leqslant P\left\{x_1^n\right\} \leqslant 2^{-n(H-\epsilon)}$$

Pour ϵ assez petit, toutes ces séquences sont à peu près équiprobables, et par définition leur vraisemblance vérifie :

$$\forall x_1^n \in \mathcal{T}_\epsilon^n, H - \epsilon \leqslant -\frac{1}{n} \log P\left\{x_1^n\right\} \leqslant H + \epsilon$$

On en déduit immédiatement que $|\mathcal{T}_\epsilon^n| \leqslant 2^{n(H+\epsilon)}$

Par ailleurs, l'AEP nous assure que pour tout ϵ positif et n assez grand, $P\left(\mathcal{T}_\epsilon^n\right) > 1 - \epsilon$, d'où l'on tire que

$$|\mathcal{T}_\epsilon^n| \geqslant (1-\epsilon) 2^{n(H-\epsilon)}$$

Ainsi, l'ensemble des éléments ϵ-typiques pour ϵ assez petit et n assez grand recouvre quasiment tout l'espace des possibles, et est composé d'éléments quasi équiprobables en nombre environ 2^{nH}. On en déduit facilement que H est une borne inférieure du taux de compression de toute méthode de codage, ainsi qu'un codeur artificiel énumératif si ces éléments typiques sont connus (cf par exemple [13]).

2 Le codeur universel *Simplistic*

Mais on peut même en déduire un pseudo-algorithme universel _ pseudo, parce qu'il n'a aucune prétention pratique et qu'il tient plus de l'expérience d'esprit que du programme informatique. Il est décrit dans l'article [4], et fonctionne de la sorte : étant donnés une séquence X_1^n d'éléments de $\{0,1\}$ et un entier non nul k compris entre 1 et n, découper la séquence en blocs de taille k, puis donner comme code C_k la concaténation :

1. d'un codage de k, dont le coût en bits C_k^k est équivalent à $\log k$;
2. d'un dictionnaire D_k énumérant tous les blocs de taille k ainsi obtenus, qui prend donc $C_k^{dict} = k\,|D_k|$ bits;
3. et de la séquence des index, dans D_k, des blocs successifs formant X_1^n, occupant un nombre de bits $C_k^{seq} = \frac{n}{k}\log|D_k| + o(n)$ bits.

La question est de choisir correctement k; mais comme ici aucune considération de performance n'est présente, on peut choisir la valeur pour laquelle la taille du code est minimale, c'est à dire $k^* = \arg\min_{1\leqslant k\leqslant n}|C_k|$.

Neuhoff et Shields montrent que ce schéma est universel en prenant ϵ arbitrairement petit et en choisissant pour $k = k(n)$ le plus grand entier k tel que $k2^{k(H+\epsilon)} \leqslant n$. En effet, on observe que comme $(k+1)2^{(k+1)(H+\epsilon)} > n$ on a :

$$
\begin{aligned}
\frac{n}{k} &< \frac{k+1}{k}2^{(k+1)(H+\epsilon)} \\
&\leqslant 2^{k(H+2\epsilon)}\frac{k+1}{k}2^{H+(1-k)\epsilon} \\
&\leqslant 2^{k(H+2\epsilon)}\text{pour n et donc k assez grand}
\end{aligned}
$$

On en déduit que :
- comme $|D_k| \leqslant \frac{n}{k}$, $\frac{1}{n}C_k^{seq} \leqslant H + 2\epsilon + o(1)$;
- et par une utilisation judicieuse (mais intuitive) de l'AEP,

$$\frac{1}{n}C_k^{dict} = \frac{k}{n}\,|D_k| \leqslant 2^{-k(H+\epsilon)}\,|D_k| \to_{n\to\infty} 0 \text{ ps}$$

On voit ainsi qu'un schéma de compression peut facilement être universel : il lui suffit de prendre en compte suffisamment d'enchaînements naturels de la source pour parvenir à ses fins. En particulier, le fait qu'un schéma aussi aveugle capte toute la puissance des sources stationnaires ergodiques remet en cause l'idée selon laquelle cette notion de source peut être pertinente pour modéliser les phénomènes pour lesquelles elle est pourtant utilisée (en particulier le texte, ou en bioinformatique). En fait, elle n'intègre que les régularités de répétition à champ d'action limité, et ne peut donc convenir pour les phénomènes ayant une mémoire arbitrairement longue comme les grammaires (que l'on pense seulement à la complexité grammaticale de l'oeuvre de Proust), les palindromes et l'ADN.

Troisième partie
Lois empiriques et parsing

3 Lois empiriques

Nous n'avons pour l'instant pas parlé de la question de savoir, sur un exemple concret comme un texte, une image ou toute autre donnée, quelles sont les propriétés de la source qui l'a généré _ si tant est, d'ailleurs, qu'il y en ait une : dans bien des cas, il peut sembler arbitraire de supposer que x a été produit par un mécanisme stochastique : quel est celui qui, par exemple, a produit l'oeuvre de Perec ?

Comme le signalent Ziv et Lempel au début de leur article [2] : "Any attempt to argue that a given finite sequence of fully specified numbers is a random sequence will be open to obvious criticism" (cette critique amène ces deux auteurs à définir une autre notion de source).

La faiblesse la plus handicapante de cette hypothèse est toutefois qu'elle sort de l'analyse *intrinsèque* : elle suppose des connaissances autres que celle de x.

Une manière de pallier à cette difficulté est d'associer à la séquence x une loi de probabilité *construite à partir d'elle* et censée l'avoir générée (!). On considèrera ici, pour $k \in \mathbb{N}^+$, la classe Λ^k_{fs} des lois dites de *Markov cachées* à k états (actuellement très utilisée en traitement du signal, elle englobe les modèles markoviens et ne semble pas dépourvue de pertinence pour le traitement de texte). Etant donné un ensemble S à k éléments (dits *états cachés*), on se donne un pour chaque état s un noyau de transition $p(s', a|s)$ vérifiant $\sum_{s' \in S, a \in \mathcal{A}} p(s', a|s) = 1$, et on définit pour $y \in \mathcal{A}^n$:

$$p(y|s_0) = \sum_{s_1,\dots,s_n \in S} \prod_{i=1}^{n} p(s_i, y_i | s_{i-1})$$

Enfin, pour $s_{init} \in S$ fixé, $\mathbf{p} \in \Lambda^k_{fs}$ est définie par $\mathbf{p} = p(.|s_{init})$.

A x on associe la loi de cette classe réalisant le maximum de vraisemblance dans notre classe:

$$\mathbf{p}^x_k = \arg \max_{\mathbf{p} \in \Lambda^k_{fs}} \mathbf{p}(x) = \arg \min_{\mathbf{p} \in \Lambda^k_{fs}} - \log \mathbf{p}(x)$$

Nous appelerons dans la suite *k-information* (resp: *k-information par bit*)de x la quantité $\mathcal{H}^*_k(x) = - \log \mathbf{p}^x_k(x)$ (resp: $\bar{\mathcal{H}}^*_k(x) = \frac{1}{n}\mathcal{H}^*_k(x)$). L' information d'une chaîne de caractères est décroissante en k. Remarquons que $\mathbf{p}^x_1$ est simplement la loi empirique de x, et que la 1-information est parfois appelée *entropie non normalisée.*

Le calcul effectif de $\mathbf{p}^x_k$ pose de gros problèmes informatiques. Quoi qu'il en soit, l'idée des codes *en deux temps* est de compresser x en codant d'abord $\mathbf{p}^x_k$, puis en codant x avec une méthode entropique pour $\mathbf{p}^x_k$ (par exemple, un codage arithmétique), ce qui "coûte" à peu près $\mathcal{H}^*_k(x)$ bits.

Cette approche permet également de fixer un nouvel étalon par rapport auquel on peut calculer le regret d'une méthode quelconque de codage, appelé *regret par rapport à la classe* Λ^k_{fs} :

$$reg(x, code, \Lambda^k_{fs}) = |code(x)| - \mathcal{H}^*_k(x)$$

On appellera *redondance minimax de code par rapport à la classe* Λ^k_{fs} (au sens des séquences individuelles) la quantité : $Red_n(code, \Lambda^k_{fs}) = \max_{x\in\mathcal{A}^n} reg(x, code, \Lambda^k_{fs})$.

On montre facilement (cf [13] p.322) en approchant un processus statitionnaire ergodique par des chaînes de Markov que si $\limsup_{n\to\infty} Red(code, \Lambda^k_{fs}) = 0$, alors *code* est universel.

4 Codage par parsing

On a vu qu'une idée classique pour coder x était de regrouper plusieurs lettres consécutives en blocs, formant ainsi un *parsing* π, dont on notera le nombre d'éléments t et la séquence des tailles de blocs $|\pi|$. La proposition suivante précise le lien entre la k-information de x et la 1-information de π.

Proposition 1

$$\mathcal{H}^*_1(\pi) \leqslant \mathcal{H}^*_k(x) + t\log k + \mathcal{H}^*_1(|\pi|)$$

DÉMONSTRATION: Soit $\mathbf{p} = (p, s_{init}) = p^x_k$, et τ la loi empirique de $|\pi|$ sur $\mathbb{N}^+$:

$$\tau(l) = \frac{|\{i \in [|1,t|] : |\pi_i| = l\}|}{t}$$

Pour $y \in \mathcal{A}^m$, posons $\tilde{q}_m = \max_{s_0\in S} p(y|s_0)$. La relation suivante est vérifiée :

$$\tilde{q}_{m+m'}(yy') \leqslant \tilde{q}_m(y)\tilde{q}_{m'}(y') \tag{1}$$

En posant $Z_m = \sum_{y\in\mathcal{A}^m} \tilde{q}_m(y)$, on peut définir la loi q_m sur $\mathcal{A}^m$ par $q_m(y) = \frac{\tilde{q}_m(y)}{Z_m}$. On a :

$$\sum_{y\in\mathcal{A}^m} p(y|s_{init}) \leqslant \sum_{y\in\mathcal{A}^m} \max_{s_0\in S} p(y|s_0) \leqslant \sum_{y\in\mathcal{A}^m}\sum_{s_0\in S} p(y|s_0)$$

Mais $\sum_{y\in\mathcal{A}^m}\sum_{s_0\in S} p(y|s_0) = \sum_{s_0\in S}\sum_{y\in\mathcal{A}^m} p(y|s_0)$ et $\forall s \in S, \sum_{y\in\mathcal{A}^m} p(y|s) = 1$, d'où :

$$1 \leqslant Z_m \leqslant k$$

Définissons maintenant la loi q sur $\mathcal{A}^+$ comme "composée" des q_m et de τ :

$$\forall \nu \in \mathcal{A}^+, q(\nu) = q_{|\nu|}(\nu)\tau\left(|\nu|\right)$$

Comme π est une séquence d'éléments de $\mathcal{A}^+$, on lui peut appliquer le résultat de ([13] p.26) assurant que sa mesure empirique réalise le maximum de vraisemblance :

$$\begin{aligned}
\mathcal{H}^*_1(\pi) &\leqslant \sum_{i=1}^t -\log q\left(\pi_i\right) && (2)\\
&= \sum_{i=1}^t -\log \tilde{q}_{|\pi_i|}\left(\pi_i\right) - \log\frac{1}{Z_{|\pi_i|}} - \log\tau\left(|\pi_i|\right) && (3)\\
&\leqslant -\log\tilde{q}_n(x) + \sum_{i=1}^t \log Z_{|\pi_i|} + \mathcal{H}^*_1\left(|\pi|\right) && (4)\\
&\leqslant \mathcal{H}^*_k(x) + t\log k + \mathcal{H}^*_1\left(|\pi|\right) && (5)
\end{aligned}$$

où (4) s'obtient par la relation 1 ci-dessus et (5) par le fait que la maximalité de $\mathbf{p}(x)$ sur Λ^k_{fs} entraîne en particulier que l'état initial de $\mathbf{p}$ réalise le maximum dans $\max_{s_0} p(y|s_0)$, donc $\tilde{q}_n(x) = \mathbf{p}(x)$ et $-\log \tilde{q}_n(x) = \mathcal{H}^*_k(x)$.

Nous verrons que le dernier terme, $\mathcal{H}^*_1(|\pi|) = t\mathcal{H}(\tau)$, est le plus difficile à manipuler. Heureusement, il peut être contrôlé de manière satisfaisante : il est évidemment positif (ou nul) et comme τ est une distribution à valeur dans $\mathbb{N}^+$ d'espérance $\frac{1}{t}\sum_{i=1}^t |\pi_i| = \frac{n}{t}$, son entropie est inférieure à celle d'une loi géométrique de même moyenne (cf [13]p.322), c'est à dire :

$$H(\tau) \leqslant \left(\frac{n}{t}+1\right)\log\left(\frac{n}{t}+1\right) - \frac{n}{t}\log\frac{n}{t} \leqslant \log\frac{n}{t} + 2\log e \tag{6}$$

Il est naturel de se demander si l'on peut obtenir une inégalité en sens inverse, pour minorer $\mathcal{H}^*_1(\pi)$ en fonction de $\mathcal{H}^*_k(x)$. Cela ne semble pas être trivialement le cas, puisque par exemple pour $x = (01)^m$, avec m entier non nul quelconque, et$\pi_i = x_{2i}^{2i+1}$, on a $\mathcal{H}^*_1(x) = 2m$ mais $\mathcal{H}^*_1(\pi) = 0$.

D'un point de vue "pratique", cette propriété suggère un nouveau codeur universel "simplissime" très semblable au précédent. Posons en effet $u = \lfloor \frac{1}{2}\log n \rfloor$ et prenons pour π le découpage de x en $t = n/u$ blocs de taille u (nous ignorons les éventuels problèmes de bord). On comprime alors x en précisant d'abord la valeur de u (sur $O(\log u)$ bits), puis par un codage arithmétique adaptatif de Krichewski-Trofimov sur l'alphabet de taille $\{0,1\}^u$ des t blocs : la proposition 1 et le théorème ([3] 3.5.3) permettent d'affirmer que le coût en bits d'un tel codage est :

$$\begin{aligned}
C(\pi) &\leqslant H^*_1(\pi) + \frac{2^u-1}{2}\log t + o\left(\frac{1}{t}\right) + O(\log u) \\
&\leqslant H^*_k(x) + t\log k + \frac{\sqrt{n}-1}{2}\log\frac{2n}{\log n - 2} + O(\log\log n) \\
&\leqslant H^*_k(x) + O\left(\frac{n}{\log n}\right)
\end{aligned}$$

La discussion ci-dessus nous assure que schéma est universel, avec une redondance par bit par rapport à Λ^k_{fs} en $O(1/\log n)$. Il ne serait pas bien difficile de tirer une version "en ligne" de cet algorithme, en parsant la source en 4^i blocs de taille i successivement pour $i = 1, 2, \ldots$ codés par un codage arithmétique indépendant pour chaque i. Mais les idées de ce type sont bien plus intelligemment utilisées par les algorithmes de Ziv-Lempel, présentés ci-dessous.

Quatrième partie
Exemples de parseurs

5 Algorithmes de Ziv-Lempel

A la fin des années 1970, Abraham Lempel et Jacob Ziv proposèrent une nouvelle famille de compresseurs qui, dans la terminologie introduite précédemment, se caractérisent par les deux propriétés suivantes :

- ce sont des *parseurs* : ils découpent l'entrée x_1^n en une séquence $(\pi_i)_{i=1..t}$ de mots sur $\mathcal{A}$;
- ils utilisent le principe du *deja-vu*, qui impose que les π_i soient tous différents.

Cette deuxième propriété a une conséquence agréable : elle entraîne en effet immédiatement que :

$$\mathcal{H}_1^*(\pi) = t \log t$$

Ainsi, la proposition 1 s'interprète comme une équation en t dont le seul terme non explicite, $\mathcal{H}_1^*(|\pi|)$, peut être contrôlé comme on l'a vu précédemment en (6). On obtient :

$$t \log t \leqslant \mathcal{H}_k^*(x) + t \log k + t \log \frac{n}{t} e^2$$

d'où:

$$t \leqslant e\sqrt{nk}\ \phi\left(\frac{\bar{\mathcal{H}}_k^*(x)}{2e}\sqrt{\frac{n}{k}}\right)$$

où $\phi :]0, +\infty[\to]1, +\infty[$ est la réciproque de la fonction $z \to z \log z$. Encadrons ϕ par des fonctions classiques :

- on a partout l'inégalité suivante qui est optimale pour u petit :

  $$\forall u > 0, 1 \leqslant \phi(u) \leqslant 1 + u$$

  Elle entraîne l'inégalité suivante, intéressante dans les cas de très faible information :

  $$t \leqslant \frac{n\bar{\mathcal{H}}_k^*}{2} + e\sqrt{kn} \tag{7}$$

  Par exemple, dans le cas $x = 0^{t(t+1)/2}$ et $k = 1$, on obtient $t \leqslant e\sqrt{\frac{t(t+1)}{2}}$ ce qui n'est pas si mauvais;
- pour u suffisamment grand, il est plus intéressant de considérer :

  $$\forall u \geqslant 2, \frac{u}{\log u} \leqslant \phi(u) \leqslant \frac{u}{\log u}\left(1 + 2\frac{\log \log u}{\log u}\right)$$

  On peut ainsi, dès que $\bar{\mathcal{H}}_k^*\sqrt{n} \geqslant 4e\sqrt{k}$, faire la majoration suivante :

  $$t \leqslant \frac{n\bar{\mathcal{H}}_k^*}{\log n\frac{\bar{\mathcal{H}}_k^{*2}}{4e^2k}}\left(1 + 2\frac{\log\log\frac{\bar{\mathcal{H}}_k^*}{2e}\sqrt{\frac{n}{k}}}{\log\frac{\bar{\mathcal{H}}_k^*}{2e}\sqrt{\frac{n}{k}}}\right) \tag{8}$$

Notons d'abord que dans tous les cas, $\mathcal{H}_k^*(x) \leqslant n$, ce qui fournit une preuve non combinatoire du fait que $t = O(\frac{n}{\log n})$ (voir la preuve classique dans [13] p.320). Plus généralement, il peut paraître remarquable que de tels arguments statistiques permettent ainsi de montrer un résultat n'ayant guère de rapport avec les probabilités.

Notons ensuite que dans le cas probabiliste où x est réellement issu d'une source à au plus k états cachés, le théorème de Shannon-Mac-Millan-Breiman assure que la k-information par bit converge presque sûrement vers entropie de la source : $\bar{\mathcal{H}}_k^* \to \mathcal{H}_k$ ps. Avec la réciproque du théorème de Shannon, qui exprime l'impossibilité de coder une source avec un taux de compression inférieur à $\mathcal{H}_k$, cela entraîne que t est équivalent à $\frac{n\mathcal{H}_k}{\log n}$. Toutefois la borne donnée par (8) ne suggère qu'une convergence *très lente*, ainsi par exemple pour $k = 1$, $\bar{\mathcal{H}}_1^* = 1$ et $n = 2^{20}$, ne donne que : $t\frac{\log n}{n} \leqslant 2.35...$

L'intérêt principal de cette formule est que, pour peu que l'on dispose d'un moyen simple d'exprimer le coût de codage d'un algorithme de Ziv-Lempel en fonction de t, elle nous permet d'en estimer la performance, au moins à l'aide d'une borne supérieure (pour avoir l'autre, il faudrait au moins avoir une réciproque satisfaisante à la propriété 1, et une estimation correcte de la distribution des $|\pi_i|$: cf [5])

Présentons ainsi deux membres gloutons de cette famille qui ont connu un succès fulgurant et qui sont utilisés encore aujourd'hui, après avoir connu de multiples petits perfectionnements.

5.1 LZ 77

Il est au coeur du programme Unix `gzip`. Son principe est simple : à l'étape i on lit la source à partir du premier caractère nouveau jusqu'à obtenir un mot $\pi_i = x_a^b$ qui ne se retrouve nulle part dans le passé. Le mot de code associé à π_i est la concaténation :

- d'un codage de R_i *temps de retour* de x_a^{b-1}, c'est à dire la longueur du premier décalage dans le passé où l'on retrouve ce motif (éventuellement 0 si x_a n'a jamais été rencontré); Cela coûte $\log R_i + O(\log\log R_i)$ bits ;
- d'un codage de $L_i = b - a + 1$, longueur du mot π_i, qui coûte $\log L_i + O(\log\log L_i)$ bits ;
- et de x_b, qui coûte un bit.

Pour le dernier mot, il peut arriver que x_a^b soit entièrement présent dans le passé ; pourtant, on le code comme précédemment (bien sûr, ça ne change pas grand chose à la taille totale du code...). Par ailleurs, on voit facilement comment, progressivement, on reconstruit x à partir de son code. Voir le programme Matlab en annexe.

Exemple : pour $x = 00010101001011110$

Etape 1:	on lit $\pi_1 = 0$	$R_1 = 0$	$L_1 = 1$	$n_1 = 0$
Etape 2:	on lit $\pi_2 = 001$	$R_2 = 1$	$L_2 = 3$	$n_2 = 1$
Etape 3:	on lit $\pi_3 = 010100$	$R_3 = 2$	$L_3 = 6$	$n_3 = 0$
Etape 4:	on lit $\pi_4 = 1011$	$R_4 = 5$	$L_4 = 4$	$n_3 = 1$
Etape 5:	on lit $\pi_5 = 110$	$R_5 = 1$	$L_5 = 3$	$n_3 = 0$

En remarquant que $\forall i \in \{1, \ldots, t\}, R_i < n$, on voit que le coût total du codage de x vérifie :

$$\begin{aligned} ZL_{77}(x) &\leqslant \sum_{i=1}^{t} \lceil\log n\rceil + \lceil\log L_i\rceil + O(\log\log n) \\ &\leqslant t\log n + \sum_{i=1}^{t}\log L_i + O(t\log\log n) \end{aligned}$$

Or
$$\frac{1}{t}\sum_{i=1}^{t}\log L_i \leqslant \log\left(\frac{1}{t}\sum_{i=1}^{t}L_i\right) = \log\frac{n}{t}$$
Comme d'après (8), $t/n = \frac{\bar{\mathcal{H}}_k^*}{\log n} + O\left(\frac{\log\log n}{\log^2 n}\right)$ et comme la fonction $f : u \to u\log 1/u$ est décroissante pour $u \leqslant 1/e$, on a:
$$\frac{1}{n}ZL_{77}(x) \leqslant \bar{\mathcal{H}}_k^* + O\left(\frac{\log\log n}{\log n}\right) \tag{9}$$

Il est par ailleurs possible de faire une étude complètement différente, réellement probabiliste, de cet algorithme, en cherchant le comportement de R_n et L_n : cf l'article pionnier [1] et un développement récent [11]. Notons aussi qu'une idée ingénieuse pour stocker R_i permet d'en faire tomber la redondance à $O\left(\frac{1}{\log n}\right)$: cf [16].

5.2 LZ 78

Il est par exemple au coeur des commandes compress pour Unix, arc, WinZip ou PKZip sur PC et StuffIt sur Mac avec sa variante LZW (pour Lempel-Ziv-Welch).

Cette fois, à l'étape i, on cherche à partir du premier caractère non lu le premier motif qui n'est pas l'un des mots parsés précédemment. Ainsi, sauf si x_a n'a jamais été parsé précédemment (auquel cas on prend $M_i = 0$), le nouveau mot $\pi_i = x_a^b$ est la concaténation d'un certain $\pi_{M_i} = x_a^{b-1}$, avec $M_i < i$, et d'un bit final : pour le coder, on indique simplement M_i (sur $\lceil\log i\rceil$ bits) et x_b.

Comme précédemment, le dernier mot parsé π_t peut être présent avant, mais pour ne pas alourdir la discussion par des considérations insignifiantes nous ignorerons dans l'étude ce problème de bord. Là aussi, la procédure de décodage est évidente : voir le code Matlab en annexe.

Exemple : pour $x = 00010101001011110$

Etape 1:	on lit $\pi_1 = 0$	$M_1 = 0$	$n_1 = 0$
Etape 2:	on lit $\pi_2 = 00$	$M_2 = 1$	$n_2 = 0$
Etape 3:	on lit $\pi_3 = 1$	$M_3 = 0$	$n_3 = 1$
Etape 4:	on lit $\pi_4 = 01$	$M_4 = 1$	$n_4 = 1$
Etape 5:	on lit $\pi_5 = 010$	$M_5 = 4$	$n_5 = 0$
Etape 6:	on lit $\pi_6 = 0101$	$M_6 = 5$	$n_6 = 1$
Etape 7:	on lit $\pi_7 = 11$	$M_7 = 3$	$n_7 = 1$
Etape 8:	on lit $\pi_8 = 10$	$M_8 = 3$	$n_8 = 0$

Le nombre de bits nécessaires au codage de x est :
$$\begin{aligned} ZL_{78}(x) &= \sum_{i=0}^{t}\lceil\log i\rceil + 1 \\ &\leqslant t\log t + 2t \\ &\leqslant n\bar{\mathcal{H}}_k^* + (4+\log k)t + n\frac{t}{n}\log\frac{n}{t} \end{aligned}$$

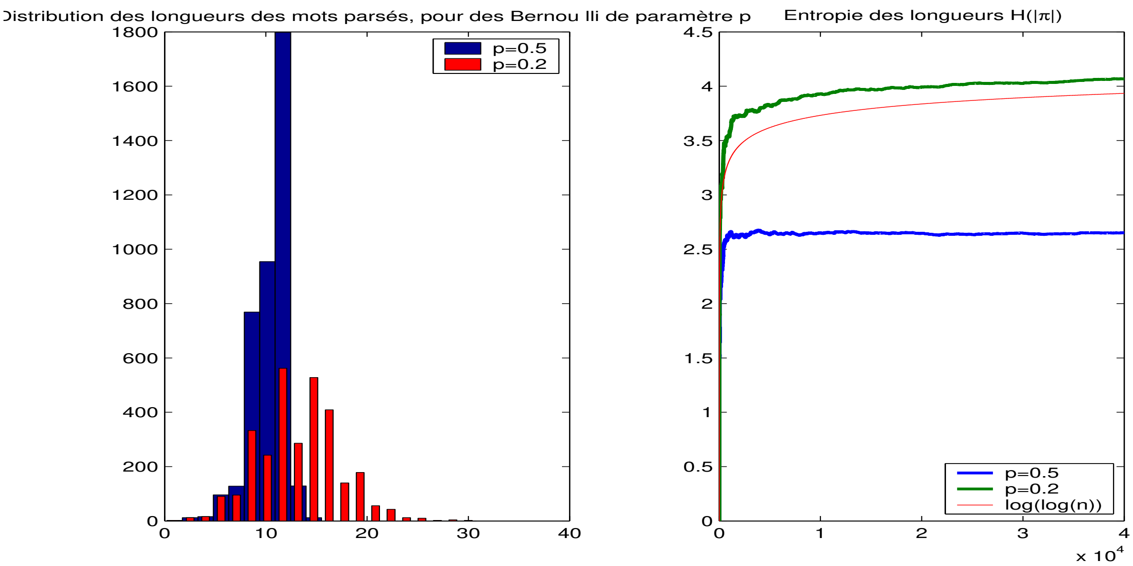
or $t/n = O(1/\log n) \to 0$ et la fonction $f : u \to u \log 1/u$ est décroissante pour $u \leqslant 1/e$, d'où :

$$\frac{1}{n} ZL_{78}(x) \leqslant \bar{\mathcal{H}}_k^* + O\left(\frac{\log\log n}{\log n}\right) \tag{10}$$

On peut légèrement améliorer l'algorithme dans deux directions : Welch propose en 1984 de ne pas perdre un bit entier pour coder les n_i (cf [15]); Tsutomu Kawabata et Hirosuke Yamamoto expliquent comment en gagner un autre par mot parsé en utilisant le fait que tous les π_k ne peuvent pas être des M_i : cf [14].

Il est facile de réaliser que LZ78 produit un parsing contenant toujours plus de mots que LZ77 : l'ensemble des mots de x_1^{a-1} est forcément plus riche qu'un sous-ensemble de celui. On pourrait donc croire qu'il est moins performant ; cela n'est pas le cas, pour au moins deux raisons : d'une part il est plus aisé à programmer en temps linéaire (la version matlab de LZ77 donnée ici est quadratique, il en existe une linéaire nécessitant des arbres de suffixes), et d'autre part chaque mot y est codé de façon plus compacte (il est inutile de préciser sa longueur).

Remarquons que généralement, pour ZL77 comme pour ZL78, la distribution des longueurs $|\pi_i|$ ne ressemble guère à une loi géométrique (en particulier, la fréquence des mots n'est pas décroissante avec leur longueur). Le graphique suivant montre des comportements différents selon l'entropie de la source : à très haute entropie (comme pour les mots de Bruijn) $\bar{\mathcal{H}}_1(\pi)$ reste borné, alors qu'à basse entropie la croissance semble bien être en $\log\log n$. En tous cas, la majoration (6) n'est certainement pas optimale.



En fait, des travaux récents (cf. [5]) étudiant précisément cette distribution ont mis en évidence une redondance moyenne pour ZL78 en $O\left(\frac{1}{\log n}\right)$, comme la version améliorée de ZL77 : il est bien difficile de dire si l'une des deux versions est intrisèquement plus performante que l'autre.

6 Codage par transformée grammaticale

6.1 Grammaire formelles

Avant de revenir dans le vif du sujet, commençons par donner une présentation succinte de l'outil théorique sur lequel se basent les méthodes récentes de compression proposées par Kieffer et Yang dans une série d'articles datant de la fin des années 1990, et dont on peut trouver un aperçu dans [9]. Pour une introduction plus poussée, voir par exemple [7].

La notion informatique de *grammaire* a été introduite par Noam Chomsky qui, au milieu des années 1950, cherchait un modèle pour représenter la syntaxe des langages naturels. Elle est formellement définie comme la donnée d'un quadruplet (T, V, v_0, P) :

- T est un alphabet (fini) d'éléments appelés *terminaux*, ici $T = \{0, 1\}$;
- V est un ensemble fini (disjoint de T) de *variables* ; on notera $m = |V|$ et $S = T \cup V$
- v_0 est un élément de V dit *variable start*;
- et P est un ensemble de *règles de productions* qui, dans notre cas de grammaire *algébrique déterministe*, se réduit à la donnée d'une fonction $f : V \to S^+$.

Il est pratique de faire de f un endomorphisme de S^* en commençant par définir $\forall t \in T, f(t) = t$, puis en considérant l'extension naturelle au monoïde multiplicatif S^* de la fonction de S dans S^+ ainsi obtenue.

On appellera *longueur de G* la quantité $|G| = \sum_{v \in V} |f(v)|$ On imposera la condition supplémentaire que nos grammaires soient *admissibles*, c'est à dire :

- qu'elle ne soit pas cyclique, c'est à dire qu'il n'existe pas deux variables v_1 et v_2 et un entier positif i tel que v_2 apparaîsse dans $f^i(v_1)$;
- qu'elle ne comporte pas de symbole inutile : dans la suite de chaînes $(f^i(v_0))_{i=0..\infty}$, chaque variable doit apparaître au moins une fois (si ça n'est pas le cas on peut enlever celles qui ne servent à rien, ainsi que les règles de production associées, et ce défaut disparaît).

Autrement dit, il doit exister un ordre partiel sur V défini par $v_1 > v_2$ si v_2 apparaît dans $f(v_1)$, et tel que $\forall v \in V, v_0 \geqslant v$. Le graphe dirigé acyclique (DAG en anglais) associé à cet ordre est appelé *graphe de dérivations* de G.

En remarquant que la distance maximale entre deux éléments comparables dans celui-ci est m, on voit que pour toute variable v de G, $f^m(v)$ est uniquement composé de symboles terminaux : on l'appellera *mot représenté par v dans G*. Il est donc invariant par f, et pour le rappeler on notera $f^m = f^\infty$.

Notons également que $f(v)$ est une séquence de symboles à laquelle, par f^∞, correspond un parsing de $f^\infty(v)$.

On dit enfin que la grammaire G *représente* le mot $f^\infty(v_0)$. Les grammaires algébriques déterministes admissibles seront dans la suite qualifiées d'*unilexicales* pour insister sur leur vocation et les distinguer des grammaires plus générales.

6.2 Méthode et Perspectives

6.2.1 ACTG

On appelle *algorithme de compression par transformée grammaticale* (ACTG) toute méthode consistant à coder un mot x en cherchant une grammaire le représentant, puis en codant celle-ci. La spécification d'une ACTG nécessite donc la donnée de deux éléments (qui ne sont pas indépendants) : d'une part l'algorithme de construction de la grammaire, et d'autre part la méthode choisie pour la compresser.

Pour la première partie, il est naturel de rechercher la plus petite grammaire représentant x : mais Lehman montre dans [12] que ce problème est NP-complet (et même qu'il est difficile d'en approcher la solution en temps polynômial).

Toutefois, l'impossibilité pratique de trouver une grammaire la plus petite possible n'est pas forcément aussi grave qu'il n'y paraît : il peut être préférable d'avoir une grammaire plus grosse, mais dans une certaine classe, parce que l'on saura trouver un codage plus efficace pour les grammaires de cette classe (voir plus loin le retour sur l'exemple de ZL78).

6.2.2 Possibilités

De manière générale, que peut-on attendre des ACTG ? Dans [12], Eric Lehman rappelle en quelques lemmes des propriétés générales permettant de s'en faire une idée. Soit $x, y \in \mathcal{A}^n$, alors :

Lemme 1 *si G_x et G_y représentent respectivement x et y, alors il existe une grammaire de taille $|G_x| + |G_y| + 2$ représentant xy, et une grammaire de taille $|G_x| + O(\log k)$ représentant x^k.*

Lemme 2 *il existe une grammaire de taille $O(\frac{n}{\log n})$ représentant x.*

Lemme 3 *la plus petite grammaire unilexicale représentant x est de taille $\Omega(\log n)$.*

Lemme 4 *si x est généré par une grammaire de taille m, alors x contient au plus mk sous-chaînes distinctes de taille k.*

Les grammaires unilexicales sont donc très efficaces pour coder les *répétitions*, même complexes ; par exemple, la propriété 1 montre qu'il existe une grammaire de taille $O(\log k)$ codant le mot $a^{k(k+1)/2}\left(ba^k\right)^{(k+1)^2}$. On peut donc attendre de nos méthodes qu'elles construisent des grammaires suffisamment compactes pour coder efficacement ce genre de mots.

Par contre, la question de savoir si les ACTG permettent de coder efficacement les langages générés par des grammaires algébriques, comme l'ensemble des mots bien parenthésés (ou mots de Dyck) ou comme les palindrômes, n'est pas si simple. En effet, être généré par une grammaire non-déterministe $\mathcal{G}$, éventuellement compacte, et être représentable par une grammaire unilexicale G de petite taille sont deux choses tout à fait différentes, les dérivations de $\mathcal{G}$ amenant à un mot pouvant être à peu près aussi nombreuses que les caractères de celui-ci (c'est le cas des deux langages précédents).

Pour obtenir un codage efficace de ces mots, il faut être capable de repérer (et donc d'utiliser pour le codage de G) les répétitions *dans la forme des règles* de la grammaire. Ce n'est pas le cas des codages proposés par Kieffer et Yang dans leurs articles, ni de celui proposé ci-dessous. En fait, on peut penser à utiliser un codage entropique universel, mais comment étudier sa performance ?

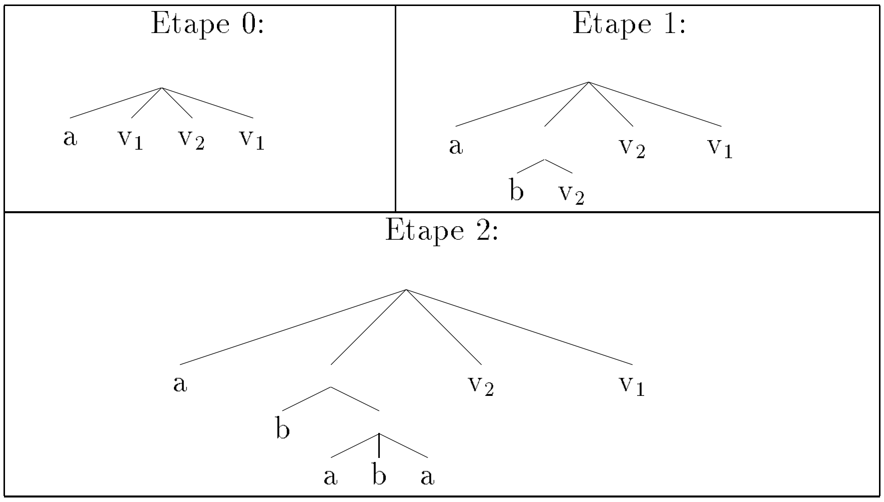
6.3 Codage d'une grammaire unilexicale

Nous proposons ici une nouvelle façon de coder une grammaire unilexicale G, largement inspirée de celles proposées par Kieffer et Yang à travers leurs différents articles, mais mettant l'accent sur une structure de données et un découpage pertinent pour l'étude des performances de la CTG.

Commençons par donner une définition constructive de l'*arbre de parsing* A associé à G. Pour cela, appelons *arbre primitif* d'une variable v_i l'arbre composé d'une racine ayant pour fils tous les éléments de $f(v_i)$, dans l'ordre où ils apparaissent.

- on part pour A d'un noeud ayant pour label v_0 ;
- puis, pour i variant de 0 à m, on remplace la première occurence de v_i rencontrée en effectuant un parcours en profondeur à gauche de A par l'arbre primitif de v_i;

Voici par exemple comment l'on construit l'arbre de parsing de la grammaire : $v_0 \to a\, v_1\, v_2\, v_1$, $v_1 \to b\, v_2$, $v_2 \to a\, b\, a$.



On comprend bien pourquoi cet objet est appelé arbre de parsing de la grammaire G pour x : en parcourant cet arbre en profondeur par la gauche, on obtient une suite de variables à laquelle, par le morphisme f_∞, est associé un parsing π_G de x.

Pour coder cet arbre, nous proposons de procéder en deux temps :

- on en code d'abord la structure, sous forme d'une suite de nombres $(r_i)_{i=1..|G|}$ indiquant l'arité de chaque noeud visité successivement par un parcours en profondeur par la gauche de l'arbre; ici par exemple, on obtient la séquence :

4	0	2	0	3	0	0	0	0	0	0

- puis on code la suite ω_G des labels de l'arbre obtenu dans ce parcours, c'est à dire, au morphisme f_∞ près, le parsing π_G, par un codage 1-entropique quelconque (par exemple, un codage arithmétique adaptatif, puisque l'on sait alors le nombre de variables auquel on a affaire)

Comme $\sum_{i=1}^{|G|} r_i = |G|$, la première étape coûte $O(|G|)$ bits. Quant à la deuxième, on sait ([3] p.33) qu'elle nécessite $H_1^*(\pi)$ bits, quantité contrôlée par la proposition 1, plus un regret en $\frac{m+|\mathcal{A}|-1}{2} \log t$.

L'idée de Kieffer et Yang est que si notre grammaire est *compacte*, c'est à dire si $\frac{|G|}{n} \to_{n\to\infty} 0$, l'essentiel du budget est dans la deuxième étape , qui comme $|G| = t-m+1$ est donc de la forme $\mathcal{H}_k^*(x_1^n)+o(n)$. Ainsi, le lemme 3 nous montre l'existence d'ACTG universels, et nous allons dans la suite en étudier quelques exemples.

Revenons tout d'abord sur le cas de ZL78. On vérifie immédiatement qu'au parsing qu'il réalise est associé la grammaire définie par :

1. $v_0 \to v_1 \dots v_t$;
2. pour $1 \leqslant i \leqslant t$, $v_i \to n_i$ si $M_i = 0$ et $v_i \to v_{M_i} n_i$ sinon.

ZL78 est donc bien un exemple d'ACTG, pour lequel $|G| = 3t$ (à un ou deux près), d'où l'on tire que $|G| = O\left(\frac{n}{\log n}\right)$: LZ78 entre dans notre étude générale. Toutefois, la forme particulière des grammaires obtenues (elles sont *régulières*) en permet un codage (indiqué plus haut) beaucoup plus économique que notre "usine à gaz".

Remarquons que, si notre but est de coder des palindrômes, une méthode ad hoc peut être utilisée : rajouter, dans le codage de l'arbre, un bit pour chaque feuille indiquant si le symbole doit être écrit dans le bon sens ou pas. Mais on ne peut en aucun cas repérer ainsi tous les palindrômes d'un mot (ceux-ci pouvant se chevaucher), et la greffe de ce codage sur l'algorithme glouton présenté ci-dessous ne permet même pas nécessairement de repérer le plus grand présent dans la source : d'arbitrairement gros peuvent être ignorés.

7 Grammaires irréductibles ; un exemple glouton

7.1 Grammaire irréductible

Dans leur article [6], Kieffer et Yang proposent la définition suivante d'une grammaire *irréductible* :

(a.1) c'est une grammaire unilexicale ;

(a.2) deux variables différentes représentent des mots différents :

$$\forall v_1, v_2 \in V(G), f_\infty(v_1) = f_\infty(v_2) \Rightarrow v_1 = v_2$$

(a.3) toutes les variables de V(G) sauf start apparaissent au moins deux fois dans ρ_G ;

(a.4) il n'y a pas deux symboles Y_1 et Y_2 tels que la chaîne $Y_1 Y_2$ apparaisse à deux endroits disjoints dans les membres de droite des règles de G.

La grammaire de ZL78 n'est par exemple pas irréductible, mais la seule règle mise en défaut est (a.3).

Les deux auteurs montrent que ces quatre propriétés suffisent à assurer que G est compacte, et même que :

$$\frac{|G|}{n} = O\left(\frac{1}{\log n}\right)$$

Ils indiquent également comment rendre une grammaire qui ne l'est pas forcément irréductible (ce qui fournit au passage une preuve du lemme 2 : il suffit de rendre irréductible la grammaire $v_0 \to x$).

7.2 Algorithme glouton pour une grammaire irréductible

Dans [6] est introduit un algorithme glouton capable de construire au fur et à mesure de la lecture de x une grammaire irréductible le représentant. Son principe est simple : on part d'une grammaire réduite à la règle $S_0 \to$ puis, à l'étape k, on lit sur ce qu'il reste de x le plus grand mot $x_{n_k}^{n_{k+1}-1}$ représenté par une variable S_{j_k} de G, ou un caractère c_k s'il n'y en a pas. La grammaire est mise à jour en ajoutant $\beta = S_{j_k}$ ou c_k à la fin de $f(v_0)$, puis en effectuant les transformations nécessaires pour rendre G irréductible qui sont de l'un des trois types suivants :

Type 1: l'ajout de β à la fin de $f(v_0)$ ne crée pas de répétition disjointe de motif de taille deux dans G : alors la grammaire est irréductible ;

Type 2: un motif $\alpha\beta$ est présent à la fois à la fin de $f(v_0)$ et à un autre endroit de G, et la dernière modification était de type 1 : alors on ajoute une nouvelle variable v associée à la règle $v \to \alpha\beta$, on enlève les deux occurrences de $\alpha\beta$ et on les remplace par v ;

Type 3: un motif $\alpha\beta$ est présent deux fois et la dernière modification n'était pas de type 1 (elle a vu la création de la variable v) : alors on enlève β des deux occurrences de $\alpha\beta$, et on l'ajoute à la fin de $f(v)$.

La grammaire est enfin codée, par exemple comme indiqué ci-dessus, et d'après notre discussion précédente ce schéma est donc universel, de redondance minimax par rapport à la classe Λ_{fs}^k $O\left(\frac{\log\log n}{\log n}\right)$.

Cette méthode, comme ZL78, est ainsi basée sur un algorithme de reconnaissance de motifs, mais cette fois le dictionnaire des motifs évolue de façon non monotone, certains mots pouvant disparaître alors que d'autres peuvent ou non apparaître.

Exemple : $x = 00010101001011110$

m_i	G_i	type	modification	motifs
0	$v_0 \to 0$	1		
0	$v_0 \to 00$	1		
0	$v_0 \to 000$	1		
1	$v_0 \to 0001$	1		
0	$v_0 \to 00010$	1		
1	$v_0 \to 000101$	2	$\begin{cases} v_0 & \to & 00v_1v_1 \\ v_1 & \to & 01 \end{cases}$	$v_1 \ : \ 01$
01	$\begin{cases} v_0 & \to & 00v_1v_1v_1 \\ v_1 & \to & 01 \end{cases}$	1		$v_1 \ : \ 01$
0	$\begin{cases} v_0 & \to & 00v_1v_1v_10 \\ v_1 & \to & 01 \end{cases}$	1		$v_1 \ : \ 01$
01	$\begin{cases} v_0 & \to & 00v_1v_1v_10v_1 \\ v_1 & \to & 01 \end{cases}$	2	$\begin{cases} v_0 & \to & 0v_2v_1v_1v_2 \\ v_1 & \to & 01 \\ v_2 & \to & 0v_1 \end{cases}$	$v_1 \ : \ 01$ $v_2 \ : \ 001$
01	$\begin{cases} v_0 & \to & 0v_2v_1v_1v_2v_1 \\ v_1 & \to & 01 \end{cases}$	3	$\begin{cases} v_0 & \to & 0v_2v_1v_2 \\ v_1 & \to & 01 \\ v_2 & \to & 0v_1v_1 \end{cases}$	$v_1 \ : \ 01$ $v_2 \ : \ 00101$
1	$\begin{cases} v_0 & \to & 0v_2v_1v_21 \\ v_1 & \to & 01 \\ v_2 & \to & 0v_1v_1 \end{cases}$	1		$v_1 \ : \ 01$ $v_2 \ : \ 00101$
1	$\begin{cases} v_0 & \to & 0v_2v_1v_211 \\ v_1 & \to & 01 \\ v_2 & \to & 0v_1v_1 \end{cases}$	1		$v_1 \ : \ 01$ $v_2 \ : \ 00101$
1	$\begin{cases} v_0 & \to & 0v_2v_1v_2111 \\ v_1 & \to & 01 \\ v_2 & \to & 0v_1v_1 \end{cases}$	1		$v_1 \ : \ 01$ $v_2 \ : \ 00101$
0	$\begin{cases} v_0 & \to & 0v_2v_1v_21110 \\ v_1 & \to & 01 \\ v_2 & \to & 0v_1v_1 \end{cases}$	1		$v_1 \ : \ 01$ $v_2 \ : \ 00101$

L'ordre associé à cette grammaire est : $v_0 > v_1 > v_2$ (**???**)

Cinquième partie
Conclusion

Parser intelligemment la source, en s'appuyant sur l'algorithmique des mots, est donc un bon moyen d'obtenir facilement des méthodes de compressions universelles, qui semblent toutefois condamnées à garder une forte redondance. En particulier, les méthodes grammaticales présentées ici semblent pouvoir tirer parti de régularités de la source qui ne sont pas prises en compte par la théorie de Shannon.

Mais il n'existe à ce jour aucun algorithme ni aucune analyse capable de cela : un moyen d'y parvenir serait de créer une transformée grammaticale à la fois compacte et suffisamment fidèle, le cas échéant, aux dérivations qui mènent à la source si celle-ci provient d'une grammaire hors-contexte.

De manière alternative, il serait possible de rechercher les propriétés d'une autre notion de complexité : celle qui à un mot associe la plus petite taille, au sens d'espace de stockage, d'une grammaire unilexicale le générant.

Références

[1] Jacob Ziv Aaron D. Wyner. Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression. *IEEE-IT 35, p.1250*, 1989.

[2] Jacob Ziv Abraham Lempel. On the complexity of finite sequences. *IEEE-IT 22, p.75*, 1976.

[3] Stéphane Boucheron. Source coding, class notes. Université d'Orsay, 2001-2002.

[4] Paul C. Shields David L. Neuhoff. Simplistic universal coding. *IEEETIT: IEEE Transactions on Information Theory*, 44, 1998.

[5] Wojciech Szpankowski Guy Louchard. On the average redundancy rate of the lempel-ziv code. In *Data Compression Conference*, pages 92–101, 1996.

[6] En-hui Yang John C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform–part one: Without context models. *IEEE-IT 46, p.755*, 2000.

[7] Jeffrey D. Ullman John E. Hopcroft, Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2 edition, 2001.

[8] A. I. Khinchin. *Mathematical Fundations of Information Theory*. Dover Publications, 1953.

[9] Kieffer and Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEETIT: IEEE Transactions on Information Theory*, 46, 2000.

[10] John C. Kieffer. Class notes. http://www.ece.umn.edu/users/kieffer/ece5585.html.

[11] Ioannis Kontoyiannis. Asymptotic recurrence and waiting times for stationary processes. *Journal of Theoretical Probability 11, p.795*, July 1998.

[12] Eric Lehman. *Approximation Algorithms for Grammar-Based Data Compression*. PhD thesis, Massachussetts Institute of Technology, February 2002.

[13] Joy A. Thomas Thomas M. Cover. *Elements of Information Theory*. Wiley Series in Telecommunication, 1991.

[14] Hirosuke Yamamoto Tsutomu Kawabata. A new implementation of the ziv-lempel incremental parsing algorithm. *IEEE-IT 37, p.1439*, September 1991.

[15] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer 17, p.8*, 1984.

[16] Wyner and Wyner. Improved redundancy of a version of the lempel-ziv algorithm. In *ISIT: Proceedings IEEE International Symposium on Information Theory, sponsored by The Information Theory Society of The Institute of Electrical and Electronic Engineers*, 1994.