Université de Montréal

Contributions à la compression de données

par
Steven Pigeon

Département d'informatique et de recherche opérationnelle
Faculté des Arts & Sciences

Thèse présentée à la Faculté des arts et sciences
en vue de l'obtention du grade
Philosophiæ Doctor (Ph. D.)
en Informatique

Décembre 2001

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée :
Contributions à la compression de données

présentée par
Steven Pigeon

a été évaluée par un jury composé des personnes suivantes :

Balazs Kegl
président-rapporteur

Yoshua Bengio
directeur de recherche

Victor Ostrmoukhov
membre du jury

Tien Dai Bui
examinateur externe

Jacques Bélair
représentant du doyen de la FÉS

Résumé

L'objectif de cette thèse est de présenter nos contributions à la compression de données. Le texte entier n'est pas consacré à nos seules contributions. Une large part est consacrée au matériel introductif et à la recension de littérature sur les sujets qui sont pertinents à nos contributions. Le premier chapitre de contribution, le chapitre *Contributions au codage des entiers* se concentre sur le problème de la génération de codes efficaces pour les entiers. Le chapitre *Codage Huffman Adaptatif* présente deux nouveaux algorithmes pour la génération dynamique de codes structurés en arbre, c'est-à-dire des codes de type Huffman. Le chapitre *LZW avec perte* explore le problème de la compression d'images comportant un petit nombre de couleurs distinctes et propose une extension avec perte d'un algorithme originalement sans perte, LZW. Enfin, le dernier chapitre de contribution, le chapitre *Les pseudo-ondelettes binaires* présente une solution originale au problème de l'analyse multirésolution des images monochromes, c'est-à-dire des images n'ayant que deux couleurs, conventionnellement noir et blanc. Ce type d'image correspond par exemple aux images textuelles telles que produites par un processus de transmission de type facsimilé.

Mots-clefs :
Compression de données, codes universels, codes de Golomb, algorithmes, ondelettes, codes adaptatifs, compression LZ78, LZW, pseudo-ondelettes

Abstract

This thesis' purpose is to present our contributions to the field of data compression. The text is not solely composed of our contributions, as large part consists in a presentation of introductory material and a review of the literature pertaining to the subjects we treat in this thesis. The problem of devising efficient codes for the integers occupies the first chapter. The second contributory chapter presents two new algorithms for Huffman-like adaptive coding. The third contributory chapter presents a lossy modification to a lossless compression algorithm, LZW, applied to the promblem of palettized image compression. The last contributory chapter deals with binary pseudowavelets. Binary pseudowavelets constitute a novel technique of multiresolution bilevel image analysis. This image decomposition scheme can be applied to progressive image transmission as well as image compression. Bilevel images are found in document archival system and facsimile transmission.

Keywords :
data compression, universal coding, Golomb coding, algorithms,
wavelets, adaptive coding, LZ78 compression, LZW,
binary pseudowavelets

Table des matières

Liste des tableaux

Table des figures

Liste des notations

e	Constante, $e \approx 2.7182818284590452354\ldots$
π	Constante, $\pi \approx 3.1415926535897932385\ldots$
ϕ	Constante, $\phi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887498948482\ldots$
$\lvert x \rvert$	Valeur absolue, longueur d'un objet
$\lvert x \rvert_0$, $\lvert x \rvert_1$,$\lvert x \rvert_T$	Nombre de 0, nombre de 1, nombre de transitions, dans l'objet x
$\lvert A \rvert$	Cardinalité de l'ensemble A.
x_i^j	Abbréviation pour la séquence $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$
$\perp$	Séquence vide, mot vide.
$\langle\, m \,\rangle$	Une chaîne de m bits
$\langle\, a, b \,\rangle$	Tuple ou fonction de pairage de a et b
$(\, a{:}b \,)$	Concaténation des objets a et b.
$\mathbb{B}$	Les bits, $\{0,1\}$
$\mathbb{N}$	Les nombres naturels, $\{1,2,3,\ldots\}$
$\mathbb{Z}^*$	Les entiers non-négatifs, $\{0,1,2,3,\ldots\}$
$\mathbb{Z}$	Les entiers, $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$.
$\mathbb{Z}_N$	Les entiers modulo N, soit $\{0,1,\ldots,N-1\}$.
$\mathbb{Q}$	Les rationnels $\{\, \frac{a}{b} \mid a \in \mathbb{Z}, b \in \mathbb{N} \,\}$
$\mathbb{R}$	Les réels
$A \times B$	Produit cartésien. Si A et B sont des ensembles, $A \times B = \{(a,b) \mid a \in A, b \in B\}$
A^n	Puissance cartésienne. $A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ fois}}$
$A \cup B$, $A \cap B$	Union de A et B, intersection de A et B
S^*	Si S est un ensemble, $S^* = S^0 \cup S \cup S^2 \cup S^3 \cup \ldots$
S^+	Si S est un ensemble, $S^+ = S \cup S^2 \cup S^3 \cup \ldots$
$\vec{x}$	Un vecteur.
$\mathbf{A}$	Une matrice.
$\mathbf{A}^\mathsf{T}$	La transposée de la matrice $\mathbf{A}$.
$\ln x$	Logarithme naturel (base e) de x
$\log x$, $\log_{10} x$	Logarithme à base 10 de x
$\lg x$	Logarithme à base 2 de x

$\lfloor x \rfloor$	Le plus grand entier plus petit ou égal à x (« plancher » de x)
$\lceil x \rceil$	Le plus petit entier plus grand ou égal à x (« plafond » de x)
$[x]$	L'entier le plus près de x (« arrondi » de x)
$\exists$	Il existe...
$\exists!$	Il n'existe qu'un seul...
$\mathfrak{F}_f(g), \tilde{\mathfrak{F}}_f(g)$	Transformée de Fourier, transformée discrète de Fourier de la fonction $g(x)$.
$\mathfrak{H}_f(g), \tilde{\mathfrak{H}}_f(g)$	Transformée de Hartley, transformée discrète de Hartley de la fonction $g(x)$.
$a \ll b$	a est beaucoup plus petit que b
$a \gg b$	a est beaucoup plus grand que b
$(a = b)$	Fonction indicatrice. Vaut 1 si $a = b$, 0 sinon
$(a \neq b)$	Fonction indicatrice. Vaut 1 si $a \neq b$, 0 sinon
$(a < b)$	Fonction indicatrice. Vaut 1 si $a < b$, 0 sinon
$(a \leqslant b)$	Fonction indicatrice. Vaut 1 si $a \leqslant b$, 0 sinon
$(a > b)$	Fonction indicatrice. Vaut 1 si $a > b$, 0 sinon
$(a \geqslant b)$	Fonction indicatrice. Vaut 1 si $a \geqslant b$, 0 sinon
$\binom{n}{m}$	Nombre de façons de choisir m parmi n, $\binom{n}{m} = \frac{n!}{m!(n-m)!}$.
$\binom{n}{n_1 n_2 \cdots n_k}$	Nombre de façons de chosir n_1, n_2, ..., n_k symboles indistingables parmi n. $\binom{n}{n_1 n_2 \cdots n_k} = \frac{n!}{n_1! n_2! \cdots n_k!}$.
$\mathrm{E}[X]$	Espérance de X
$\mathcal{U}(a, b)$	Variable aléatoire uniforme, tirée sur $[a, b]$
$\mathcal{N}(\mu, \sigma)$	Variable aléatoire normale, de moyenne μ et d'écart type σ

À mes parents,
Frances et Jean-Noël,
pour leur soutien indéfectible

Remerciements

Je remercie mon directeur de thèse, monsieur Yoshua Bengio pour ses conseils et son support tout au long de mon doctorat. J'ai grandement apprécié la liberté de recherche qu'il m'a accordé ; probablement a-t-il senti mon côté individualiste. Il me faut remercier l'équipe de la bibliothèque de mathématique et d'informatique (Manon Bergeron, Diane Duchesne, Maryse Bérubé, Francine Caplette, Ferroudja Nazef, Louise Lamothe-Daoust, Francine Larouche, Maryna Beaulieu et Yves Groulx) qui m'ont aidé à récupérer un bon nombre d'articles difficiles à trouver. Je remercie aussi le CRSNG pour son soutien financier. Bien sûr, je ne peux oublier ni remercier assez mes parents, Frances et Jean-Noël, pour leur soutien et leur encouragements pendant toutes ces longues années.

Avant-Propos

La compression de données, c'est l'ensemble des techniques et des méthodes que l'on utilisera pour réduire le volume des données sans perdre d'information importante. La réduction s'opérera soit par un algorithme sans perte où toutes les données originales seront retrouvées, soit par un algorithme avec perte où les données récupérées après la compression représentent une reconstruction raisonnable des données originales. Réduire le volume des données permet d'emmagasiner plus d'information sur un même média, ou prendre moins de temps pour les transferts. Dans certain cas, le volume des données est tel qu'il serait quasiment impossible de les gérer sans utiliser la compression.

Le choix de la compression de données en tant que sujet de recherche peut se justifier amplement par l'intérêt du sujet lui-même. C'est en effet un champ de recherche arrivé à maturité et le choix de sujet à l'intérieur de cette discipline n'est pas restreint à quelques objets. Je suis cependant arrivé à porter un grand intérêt aux méthodes de compression par un autre chemin. À une époque reculée, les ordinateurs disposaient de très peu de mémoire et les médias de stockage étaient très limités. On devait alors utiliser des méthodes de compression *ad hoc*, la plupart du temps assez simple et assez peu performantes, pour réussir à maximiser la capacité du système ; obtenir pour ainsi dire une espèce de plus-value. La recherche de la représentation efficace des données est toujours demeurée un soucis.

Par la suite, au terme de mes travaux de maîtrise, je suis revenu à la compression par un autre chemin. Mon sujet de recherche à la maîtrise était l'utilisation des ordinateurs à logique programmable pour les réseaux de neurones et le traitement d'image. Cela amène automatiquement le problème de minimisation de fonctions logiques : les puces qui supportaient la logique programmable avaient alors une capacité assez réduite et il fallait souvent minimiser le nombre de portes logiques nécessaires pour calculer la fonction. Minimiser une fonction logique peut être considéré comme une forme de compression. L'intérêt que je n'avais jamais perdu ne pouvait que s'en trouver ravivé.

La compression de données s'imposait comme choix de sujet de recherche pour une thèse de doctorat. Or, comme mes intérêts, même concernant la compression de données, sont demeurés variés, on trouvera dans cette thèse des sujets fort différents même s'ils demeurent au sein de la même discipline générale. On trouvera des contributions sur le codage universel des entiers comme sur la compression d'image. J'ai voulu écrire cette thèse de façon à ce qu'elle soit compréhensible, pour un lecteur informaticien, sans avoir recours à d'autres ouvrages. C'est pourquoi j'ai pris soin d'inclure des chapitres qui présentent la matière introdutive à mes propres travaux.

L'objectif des chapitres introductifs n'est évidemment pas d'offrir un survol exhaustif de ce

qui se fait en compression de données. D'une part parce que ce serait une tâche impossible, d'autre part parce que le matériel présenté doit supporter le discours principal de la thèse, à savoir les contributions qui y sont présentées. Le matériel introductif offre une vue d'ensemble du champ de recherche et tente de bien représenter les grands courants de pensées. Nous ne couvrons certains sujets plus en détail que lorsqu'ils sont nécessaires pour motiver ou pour appuyer les concepts utilisés dans les chapitres contributifs. À la fin de chaque chapitre introductif, vous trouverez une section « remarques bibliographiques » où sont présentées, pour le lecteur qui voudra pousser plus loin, des suggestions de lecture pour compléter le sujet couvert par le chapitre en question. Un soin particulier a été apporté à la bibliographie, où vous trouverez plus de deux cent vingt références.

Les chapitres contributifs feront référence aux chapitres introductifs dans la mesure où c'est nécessaire pour la compréhension et la situation de mes contributions au sujet. Ces contributions sont d'ailleurs fort variées, comme il en a déjà été fait mention. Certaines portent sur le codage des entiers tirés selon une loi géométrique ou encore sur le codage universel. D'autres contributions apportent des solutions différentes à des problèmes déjà explorés sous d'autres angles ; par exemple les pseudo-ondelettes binaires offrent une nouvelle technique pour décomposer une image monochrome. Les images monochromes sont presque toujours codées dans le domaine spatial alors que les pseudo-ondelettes offrent des outils pour décomposer en fréquences ces images et pour pouvoir attaquer le codage sous un angle qui ne s'applique généralement qu'aux images en tons continus ou en couleur. Je présente aussi des algorithmes pour compresser les images à base de palettes. Alors qu'elles sont généralement codées sans perte, il y a moyen de modifier un algorithme standard pour inclure une certaine perte et améliorer le taux de compression sans nécessairement sacrifier la qualité d'image. Le chapitre 2, *Plan et objectifs de la thèse* décrit en détail les contributions ainsi que la structure générale de la thèse.

❁ ❁ ❁

Enfin arrive l'inévitable sujet des droits des œuvres reproduites. Dans le texte, on retrouve çà et là des œuvres empruntées à divers ouvrages ou organismes :

- La photographie de Samuel Finley Breese Morse, fig. 1.1, page 2, est utilisée avec la permission gracieuse de Locust Groove Museum, 2683 South Road, Poughkeepsie, New York, par l'intermédiaire de Victoria Rutherford, directrice des communications.

- La gravure de Clewberg-Edelcrantz, ainsi que la photographie d'Anders Lindberg, figs. 1.2 et 1.3, pages 5 et 6, sont la propriété du Musée des Télécommunications, sis au 7, Museivägen, Norra DjurgÄrden, boîte postale 27842, 115 93 Stockholm, qui en autorise gracieusement l'utilisation non commerciale, par l'intermédiaire de Eva Derlow, attachée aux communications.

- La photographie de Claude Elwood Shannon, fig. 3.2, page 36, est © Lucent Technologies, qui en autorise l'utilisation pour toute fin éducative et non commerciale.

- La photographie de David Huffman, fig. 6.1, page 119, est © UCSC Public Information Office, Carriage House, 1156 High Street, Santa Cruz, Californie, qui en a permi l'utilisation pour des fins non commerciales, par l'intermédiaire de Tim Stephens, du service des

photographies.

- Les images du corpus Kodak sont reproduites selon le droit limité d'utilisation non commerciale.

- Les images tirées du corpus du CCITT (maintenant le ITU) sont de domaine publique et aucune restriction particulière n'est imposée.

Chapitre 1

Introduction à la compression

1.1 Introduction

La compression de données, de façon simplifiée, c'est l'ensemble des méthodes que l'on utilise pour prendre un message long pour en faire un message court, sans perdre d'information importante. Nous trouvons de tels usages dans la vie de tous les jours et pas seulement dans les application technologiques. Nous utilisons constamment des abréviations pour prendre des notes manuscrites plus rapidement, nous remplaçons des noms d'items par leur numéros d'inventaire, etc. Puisque cette thèse ne traite pas de la compression des gaz, des fluides ou des matériaux, j'utiliserai, sans ambiguïté, simplement le mot *compression* pour parler de la *compression de données*.

Un des exemples les plus anciens de compression, c'est le code Morse, inventé par Samuel Finley Breese Morse (1791 – 1872). Le code fut conçu dès 1837, mais ne connut le succès qu'après les années 1845. Le code Morse est composé de trois symboles : le point (·), le trait (–), et une pause servant à délimiter les codes. Dans le code Morse, chaque lettre et chiffre est représenté par une série de points et de traits, terminés par une pause. On transmettait les messages traduits en code Morse par le télégraphe, un appareil électromécanique capable de représenter deux états, ouvert ou fermé. Quand l'appareil recevait un signal électrique il émettait un son, et quand il ne recevait aucun signal il demeurait muet. Par convention, un trait durait trois fois plus longtemps qu'un point ou une pause, ce qui permettait de les différencier facilement.

Le télégraphe est un moyen de transmission relativement lent car il est opéré par un humain. Un code possible aurait pu assigner, à chaque lettre, une série de points et traits de la même longueur. Supposons que chaque symbole reçoive un code de longueur six. Si les codes ont une longueur de six signaux, et chaque signal prend deux valeurs (point et trait), nous avons $2^6 = 64$ codes possibles. Cela donne suffisamment de codes pour l'alphabet, les chiffres et quelques symboles de ponctuation. Ce code aurait été plus régulier, mais il aurait été aussi très inefficace, car toutes les lettres ne sont pas utilisée avec la même fréquence.

Il semble alors tout à fait raisonnable d'utiliser des codes de longueur variable. Grâce à la pause, il devient plutôt facile de créer un code assez efficace et non-ambigu. Le code est efficace si les lettres les plus fréquentes reçoivent les codes les plus courts et il sera non-ambigu s'il n'est pas possible de décoder un message de plusieurs façons différentes. Le code de Morse n'est pas

FIG. 1.1 – Samuel Morse (1791–1872), circa 1850. Photo © Locust Groove Museum.

A	· –	K	– · –	U	· · –	4	· · · · –
B	– · · ·	L	· – · ·	V	· · · –	5	· · · · ·
C	– · – ·	M	– –	W	· – –	6	– · · · ·
D	– · ·	N	– ·	X	– · · –	7	– – · · ·
E	·	O	– – –	Y	– · – –	8	– – – · ·
F	· · – ·	P	· – – ·	Z	– – · ·	9	– – – – ·
G	– – ·	Q	– – · –	0	– – – – –	.	· – · – · –
H	· · · ·	R	· – ·	1	· – – – –	?	· · – – · ·
I	· ·	S	· · ·	2	· · – – –		
J	· – – –	T	–	3	· · · – –		

TAB. 1.1 – Le code Morse.

nécessairement très efficace, mais comme le montre le tableau 1.1, la longueur des codes varie grandement. Le code donne une longueur moyenne de messages (en nombre de signaux) nettement inférieure à six.

Après la première transmission télégraphique, le 1er mai 1844 entre Washington et Baltimore (un message d'ailleurs cryptique, extrait de la Bible, "*What God hath Wrought*"), le code Morse fut rapidement utilisé pour toutes sortes d'applications. Dès 1846, plusieurs compagnies avaient commencé à câbler les États Unis, de Washington jusqu'à Boston. En 1858, ce fut au tour de compagnies européennes d'installer des réseaux de fils télégraphiques. Bien que Morse n'opéra pas directement de compagnie télégraphique, il recevait des droits, ce qui lui permit d'acquérir une fortune considérable.

Durant la guerre du Viêt-nam, les prisonniers américains capturés par l'ennemi se retrouvaient dans des camps aux conditions déplorables. Dans ces camps, ils étaient isolés dans de petites cellules, et étaient sévèrement punis lorsqu'ils étaient surpris à chuchotter ou à se transmettre des messages sur des bouts de papier. Pour communiquer entre eux sans écrire et sans élever la voix, ils utilisèrent les *tap codes*. Le code fonctionne semblablement au code Morse, sauf que l'usage du télégraphe est remplacé par le tapotement des doigts contre le mur de la cellule.

E	13434	U	5601	Q	1260	J	475
S	7224	O	5137	F	1232	Y	276
N	6831	L	4241	B	1105	Z	128
A	6548	D	3484	V	1024	K	118
I	6521	P	2163	G	905	W	69
R	6403	M	2942	X	688		
T	6399	C	2884	H	635		

TAB. 1.2 – Les fréquences des lettres de l'alphabet, tirées d'un texte écrit en français (151 316 caractères).

	1	2	3	4	5
1	A	B	C,K	D	E
2	F	G	H	I	J
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

a)

	1	2	3	4	5
1	E	N	R	L	Q
2	S	I	O	C	G
3	A	U	M	V,W	J
4	T	P	B	H	Z
5	D	F	X	Y	K

b)

TAB. 1.3 – Deux versions des *tap codes*. a) Codes originaux, b) codes efficaces.

Le bruit sourd qui en résulte est difficile à entendre à quelque distance, diminuant ainsi le risque d'interception par les gardiens qui avaient la matraque facile.

Plutôt que d'utiliser une séquence de type Morse, le code original montré au tableau 1.3 a), arrange les lettres de l'alphabet dans une matrice 5×5, en prenant soin de mettre le k avec le c. Pour émettre une lettre, on trouve sa position dans le tableau. On tappe d'abord un nombre de fois correspondant au numéro de la rangée, on marque une pause, puis on tappe un nombre de fois correspondant au numéro de la colonne. L'arrangement du tableau fait en sorte qu'il soit facile de se souvenir du code. Ce code est simple, mais pas particulièrement efficace. La lettre e reçoit le code 1 pause 5, ce qui est plutôt long pour la lettre la plus fréquente. Pour palier à ce léger défaut du code, une série d'abréviations fut utilisée ; par exemple, DLTBBB (*dont let the bed bugs bite*), GNST (*good night, sleep tight*), etc.

Un code plus efficace aurait tenu compte des fréquences des lettres dans la langue considérée (dans ce cas, l'anglais). Le tableau 1.2 montre les fréquences des lettres obtenues à partir d'un texte assez long écrit en français. Le tableau 1.3 b) montre un arrangement qui tient compte de la fréquence, assigant à e le code 1 pause 1, qui est le plus court de tous. Les lettres sont organisées de façon à ce que moins la lettre est fréquente, plus le nombre nécessaire de tapotements est grand.

De retour au XIX^{e} siècle, on trouve un autre code que l'on pourrait associer à la compression : c'est l'alphabet Braille. On doit cette invention à Louis Braille (1809 – 1852). Suite à un accident où il se creva un œil avec une serpette à découper le cuir, Braille devint aveugle vers cinq ans. Après des études difficiles, il parvint à enseigner, dès 1828, à l'Institution des Aveugles,

A, 0	H, 7	0	V
B, 1	I, 8	P	W
C, 2	J, 9	Q	X
D, 3	K	R	Y
E, 4	L	S	Z
F, 5	M	T	
G, 6	N	U	

TAB. 1.4 – Le code Braille pour les lettres de l'alphabet et les chiffres.

fondé par Valentin Haüy en 1754, où il inventa progressivement un système pour aider ses élèves. Le système Braille est d'abord un système tactile. Chaque symbole est composé de six points (deux de large, trois de haut) qui peuvent être soit pressés en ronde-bosse dans le papier, soit laissés plats. Pour lire un texte Braille, le lecteur effleure la surface embossée du papier avec ses doigts. Les six points par symbole permettent de disposer de 64 symboles de base. Le tableau 1.4 montre quelques codes Braille.

Comme l'alphabet de base ne comporte que vingt six symboles, en ajoutant les chiffres et les ponctuations, il nous reste encore une bonne trentaine de symboles Braille non assignés. Or, plutôt que de les laisser tomber, le code Braille avancé (Grade 2) assigne des mots fréquents et des groupes de lettres aux codes restant. Les mots (ou groupes de lettres) sont alors compressés dans un seul symbole, réduisant ainsi significativement la longueur du texte. C'est d'autant plus important que la taille physique des symboles Braille ne peut être réduite indéfiniment ; il faut que leurs dimensions soient compatibles avec la sensibilité tactile moyenne. N'oublions pas qu'il faut lire du bout des doigts !

Comme le langage des signes pour les sourds et muets, l'alphabet Braille souffre des nombreux dialectes qui dépendent du lieu d'utilisation. Dans les pays francophones, les groupes de lettres assignés aux codes nonalphabétiques sont différents des groupes assignés dans les pays anglophones ou germaniques. Les dialectes rendent évidemment difficile la communication entre correspondants de cultures différentes. Ce problème fait présentement l'objet de débats visant l'uniformisation, au niveau mondial, du code. Le tableau 1.4 montre les codes pour les lettres de l'alphabet et les chiffres.

Plus ancien encore que ces deux derniers codes, les codes de télégraphes à volets de l'amirauté britannique étaient utilisés pour communiquer rapidement de Londres jusqu'aux différentes stations côtières. Les télégraphes à volets n'était pas électromécaniques comme le télégraphe de Morse, mais étaient simplement mécaniques. Au dessus d'un fortin, s'élevait un pylône supportant six panneaux qui pouvaient être pivotés sur un axe central à chaque panneau, permettant ainsi, pour un observateur lointain, de voir les panneaux comme étant fermés ou ouverts. Les fortins se trouvaient à environ cinq milles les uns des autres, permettant ainsi à un observateur muni d'une longue-vue de voir les codes émis par un fortin voisin. La structure des codes res-

FIG. 1.2 – Abraham Niclas Clewberg-Edelcrantz (1754 – 1821)

semble à la structure que l'on retrouve dans le code Braille, et on trouve aussi 64 codes possibles. Or, ces codes avaient une application militaire et changeaient constamment pour confondre les ennemis de Sa Majesté. Il n'est donc pas possible de dresser une table de la signification des différents symboles de façon réaliste. On peut cependant supposer, en tout réalisme, que, tout comme le code Braille, les codes n'étaient pas seulement assignés à des lettres simples, mais aussi à quelques mots d'usage courant. Bien que la rotation des codes ait pu prévenir les indiscrétions, les systèmes cryptographiques de la fin du XVIII^{e} siècle étant somme toute assez sommaires, on ne doit pas s'attendre à un système véritablement sophistiqué.

Les télégraphes optiques ont été introduit en Suède par Abraham Niclas Clewberg-Edelcrantz (1754 – 1821). La figure 1.2 nous montre Clewberg-Edelcrantz et la figure 1.3 nous montre une de ses cabines surmontée d'un mât à dix volets. Connaissant le succès dès leur introdution en 1794, il s'imposa face à son principal concurrent, le Français Claude Chappe. Le système de Chappe était plus simple mais plus lent que le système de Clewberg-Edelcrantz, ce qui explique que le système de Chappe n'ait jamais pris racine. Une technologie tout à fait comparable est apparue en Angleterre à la même époque, mais la technologie anglaise a plutôt donné naissance aux stations à six volets, du type Chappe, qui étaient moins efficaces.

1.2 Pourquoi la compression ?

Toutes les méthodes anciennes de compression décrites à la section précédente sont bien entendu délicieusement obsolètes. Le code Morse n'est plus utilisé que dans de rares situations, le traffic de télégrammes a été éradiqué par la venue du courrier électronique et de l'Internet. Les fortins qui se relayaient les ordres de l'amirauté ont été remplacés par les liens satellites à haut débit, de surcroît fort bien encryptés. La nature même des messages a radicalement changée. Des missives télégraphiques nous sommes passés aux communications enrichies : documents texte, images, télémétrie, son, musique et vidéo. L'avènement de l'Internet, et en général l'arrivée des médias de communication à haut débit comme des médias de stockage haute densité, a transformé profondément la façon dont nous communiquons. Nous vivons alors dans une société numérique. Nous communiquons entre nous par le biais de téléphones mobiles numériques, nous nous envoyons des messages grâce aux systèmes de courrier électronique. Nous transformons la

FIG. 1.3 – Une des installations de Clewberg-Edelcrantz. Photo d'Anders Lindeberg.

musique que nous écoutons en format numérique, et pouvons faire tenir plusieurs heures dans un baladeur de la taille d'un briquet. Nous transformons les moments cocasses d'émissions de télévision en vidéos numériques et nous nous les échangeons entre amis par le courrier électronique. Nous pouvons visiter des musées à l'autre bout du monde par le biais de leur site web et par la technologie émergeante de la réalité virtuelle. Nous pouvons maintenant faire tenir une respectable collection de tableaux de maîtres sur quelques disquettes. Nos caméras sont maintenant numériques ; plus facile que jamais à utiliser, on peut envoyer par courrier les photos prises le jour même à nos amis, dès notre retour à la maison.

Sans nécessairement discourir longuement sur la façon dont ces technologies transforment le tissu social des classes moyennes et supérieures, notre façon de travailler et d'aborder nos relations interpersonnelles, insistons seulement sur le fait que ces transformations sont le résultats de percées technologiques basées sur l'échange efficace d'information. Cet échange efficace ne pourrait être possible si nous utilisions ces données sous leur forme brute. En effet, toutes ces données, qui sont à la base de la révolution numérique, sont, à l'état brut, beaucoup trop volumineuses pour être manipulées telles quelles, et ce, malgré les prodigieuses améliorations des technologies de stockage et de télécommunication. Pour ne pas noyer nos réseaux, nous devons compresser les images, le son et la vidéo de façon très efficace.

Bien entendu, les codes ont changé. Plutôt que les points, les traits et les pauses du code Morse, nous trouvons le *bit*. C'est avec les bits que l'on fabrique les nouveaux codes. Un bit, c'est littéralement l'unité de base de l'information, comme nous le verrons dans un chapitre ultérieur. Comme le gramme ou le millilitre, le bit sert à mesurer quantitativement l'information contenue dans un message. Un bit n'accepte que deux valeurs, 0 ou 1. Lorsque les bits se regroupent en paquets de huit, nous obtenons un octet. Un kilo-octet, c'est 1024, ou 2^{10}, octets. Un méga-octet, c'est $1024^2 = 1048576$, ou 2^{20}, octets. En continuant la progression, 2^{30} représente un giga-octet, 2^{40} un téra-octet.

La plupart des images se mesurent, dans leur état non compressé, en centaines de kilo-octets, même souvent en dizaines de méga-octets. Une pièce de musique numérisée, au niveau de qualité du disque compact, « pèse » plusieurs dizaines de méga-octets. La vidéo, voire de qualité

médiocre, se mesure en centaines de méga-octets par minute. Même une simple conversation téléphonique, en supposant que nous ayons la même bande passante que les systèmes téléphoniques actuels, demanderait encore environ 8 kilo-octet par seconde, pour être de qualité jugée satisfaisante.

Bien que les infrastructures de l'Internet peuvent supporter des débits très élevés, il reste encore le problème de transporter l'information jusqu'au consommateur qui, en bout de ligne, a peut-être un modem comparativement lent. Sans la compression de données, une vidéo non compressée prendrait des jours à télécharger. Il est alors nécessaire de compresser les données, non seulement pour que ces diverses technologies soient pratiques, mais pour qu'il soit tout simplement possible de les utiliser ! Dans ce contexte, la compression de données joue un double rôle. D'une part, elle minimise le temps d'attente de l'usager qui veut obtenir des données, d'autre part, elle minimise la charge sur les infrastructures. Réduire la charge sur les infrastructures est d'autant plus important que le nombre d'utilisateurs croît très rapidement.

❁ ❁ ❁

Dans cette section, nous ferons un tour d'horizon des applications de la compression. Nous verrons comment la compression aide les télécommunications comme les microconsoles de jeu. Nous verrons aussi pourquoi la compression, comme moyen d'économiser de l'espace de stockage, a aussi un sens économique pour l'industrie.

1.2.1 Applications aux télécommunications

Tout médium de communication est défini principalement — avant même ses diverses caractéristiques physiques — par sa bande passante. Cette bande passante définit le nombre de bits par seconde que l'on peut transmettre par ce médium. Ce nombre de bits par seconde est limité par les caractéristiques physiques du médium de transmission. Le nombre de bits par seconde est aussi souvent appelé bande passante, bien qu'il s'agisse là en fait d'un abus de langage. Les médias diffèrent grandement dans leurs bandes passantes ; certains vont de quelques kilobits jusqu'à des centaines de gigabits. Les médias câblés comme les modems et les réseaux à fils de cuivre offrent de quelques dizaines de kilobits par seconde à quelques centaines de mégabits, alors que les médias comme les fibres optiques et les liens satellite offrent des bandes passantes dans l'ordre de la centaine de gigabit.

Les lignes de téléphone classiques, par exemple, contraignent les modems à tout au plus 64 kilobits par seconde. Il s'agit d'une contrainte matérielle imposée par les paramètres de conception du réseau téléphonique. À la base du réseau téléphonique, on retrouve des lignes à fils de cuivre. Ces lignes sont spécialement conçues pour ne laisser passer que les fréquences entre 300 Hz et 3500 Hz. Cette limitation en fréquence est d'origine historique. Avant l'avènement de la téléphonie numérique par commutation de paquets, les différentes conversations étaient physiquement superposées par bandes de fréquences dans le spectre pour être acheminées par le même fil de cuivre. Ainsi, en se donnant une bande par canal d'environ 3 KHz, on s'assurait d'une bonne reproduction de la voix humaine, et si on empile les canaux par bandes d'une largeur de 4 KHz en multiplexage de fréquence, on s'assure que les fréquences provenant d'un canal ne se mélangent pas à celles d'un canal voisin, nous empêchant d'entendre les autres conversations qui ont lieu au même moment. Ce qui semblait être une très bonne idée au départ s'est finalement

retourné contre nous. Ces limitations de bande passante empêchent les modems téléphoniques d'atteindre de très grandes vitesses de transmission. Quoiqu'il en soit, les modems récents offrent une bande passante garantie de 56 kilobits par seconde. Remarquons que les compagnies téléphoniques offrent maintenant des modems à « haute vitesse » (un mégabit par seconde) mais seulement pour les régions desservies par les tronçons du réseau qui ont été récemment remplacés.

Comme 56 kilobits est comparativement lent, surtout si on se réfère à la technologie de réseaux locaux qui offrent des connexions à 10 mégabits et plus, il est normal de chercher à augmenter virtuellement la bande passante du modem. Comme nous pouvons argumenter que 56 kilobits par seconde (kbs) est près de la capacité physique du canal de transmission, dû à ses limitations intrinsèques, nous devons chercher à compresser l'information. Les modems incorporent donc une gamme variée de protocoles de communication, lesquels sont assortis d'algorithmes de compression, permettant ainsi d'atteindre des taux de transfert plus intéressants sans augmenter physiquement la bande passante. Par exemple, si on transfère un fichier facile à compresser, un modem de 56 kbs peut donner un taux de transfert virtuel de 10 kilo-octet par secondes, alors que sa bande passante physique est de l'ordre de 6 kilo-octet par seconde. Il s'agit donc d'une amélioration considérable.

Bien que la compression soit nécessaire pour un modem relativement lent, elle demeure intéressante pour les médias à grande vitesse, comme les réseaux locaux ou même l'Internet — qui n'est d'ailleurs pas toujours très rapide. On diminuera le temps de transfert de façon proportionnelle au taux de compression, ce qui se traduira en des temps d'attente réduits. On remarquera que sur l'Internet, presque tout voyage de façon compressée, qu'il s'agisse de son ou d'image.

1.2.2 Applications au stockage de données

Similairement à un médium de communication, un médium de stockage d'information est essentiellement défini par sa capacité, soit le nombre de bits qu'il peut emmagasiner. Comme le dit un proverbe d'informaticien, peu importe la taille du disque rigide, il est toujours plein. Cet aphorisme souligne l'importance du stockage de masse comme la propension des utilisateurs à l'entroposage de données et à l'utilisation d'applications de plus en plus lourdes.

Parfois, les médias de stockage utilisent des techniques de correction d'erreur pour palier à leurs imperfections de fabrication. Les techniques de détection et de correction d'erreur ajoutent des bits aux données initiales pour les rendre résistantes aux erreurs. Une erreur, dans ce contexte, c'est par exemple un bit qui a changé de valeur. Dépendemment de la tolérance désirée, on ajoutera plus ou moins de bits redondants pour compenser pour des erreurs éventuelles. Les CD-ROMs, par exemple, utilisent environ 15% de leur capacité de stockage pour les codes de correction d'erreur. Un système encore plus résistant aux erreurs pourrait demander encore une plus grande partie de la « charge utile », et il serait alors intéressant d'utiliser une méthode de compression pour contrebalancer les effets de la correction d'erreur. Malheureusement, les CD-ROMs n'utilisent que la correction d'erreur et absolument aucune méthode de compression.

La compression permet d'augmenter virtuellement la capacité du médium de stockage, qu'il s'agisse d'un CD-ROM ou d'un disque rigide. Dans certains systèmes d'exploitation, dont Linux et Windows, la compression fait partie intégrante du système de gestion des fichiers et permet de manipuler les fichiers compressés sur le disque comme s'il s'agissait de fichiers normaux. Le

module de compression est caché assez profondément dans le système de gestion de fichier et compresse et décompresse à la volée les informations qui vont vers ou qui proviennent d'un fichier sur disque. Cette stratégie permet d'avoir un système de fichiers compressé, et ce d'une façon totalement transparente pour les applications. Ces algorithmes de compression sont habituellement plus optimisés pour la vitesse que pour la performance en compression. Les fichiers sont au plus réduit de moitié, ce qui est quand même satisfaisant.

Dans la même veine, on retrouvera aussi des programmes qui permettent de créer ce que l'on appele des archives. Les archives sont essentiellement des copies de sécurité structurées. Ces archives permettent de récupérer toute la structure des répertoires et des fichiers pris en copie de sécurité. Ces archiveurs permettent en général non seulement de faire une copie structurée mais aussi de conserver les données sous forme compressée. Cela fait de plus petites copies de sécurité, ce qui permet de les stocker sur moins de disques, rubans ou tout autre média que l'on utilisera pour entreposer les copies de sécurité. Si on fait les copies de sécurité pour un réseau local, chaque ordinateur sur ce réseau disposera d'un programme de copie qui, en communiquant avec le serveur principal, servira à mettre à jour seulement les fichiers modifiés. Si les fichiers modifiés voyagent dans le réseau sous forme compressée, on sauve sur le temps de transfert et aussi sur la puissance de calcul nécessaire du serveur principal. En effet, si le temps de transfert est réduit, l'ensemble des copies de sécurité peut se faire plus rapidement. Si ce sont les ordinateurs du réseau et non le serveur qui compressent les fichiers, on distribue le travail de compression pour produire l'archive. On gagne sur les deux tableaux.

1.2.3 Applications de poche

Les endroits où la compression peut trouver application semblent innombrables. On peut penser aux dictionnaires électroniques de poche. Ce sont habituellement des ordinateurs très lents et avec une mémoire très limitée. Ils sont très lents parce qu'ils consomment habituellement très peu car un même ensemble de piles doit durer des années. Cela rend rédhibitoire l'utilisation de processeurs haute performance ; on se contente alors de processeurs ayant des cadences d'horloge vraiment en deçà du mégahertz. On ne peut donc se payer le luxe d'avoir un algorithme de décompression très sophistiqué. Vraisemblablement, on aura recours à une méthode *ad hoc* spécialement conçue pour compresser les structures de données nécessaires à l'application. Ce qui est certain, c'est que pour faire entrer un dictionnaire décent dans 128 kilo-octets, il faut appliquer une forme de compression.

Les dictionnaires électroniques de poches ne sont pas les seules applications qui nécessitent des méthodes de compression. On peut penser aux receveurs GPS (*global positioning system*) qui sont maintenant abordables. Les modèles de base connaissent la position de quelques milliers de villes, le tracé des routes principales, des grands cours d'eau et des information similaires. En plus, certains sont capables de mémoriser les itinéraires suivis par le porteur, et se souvenir d'un certain nombre de bornes positionnées par l'utilisateur. S'il est vrai que l'information qu'on y trouve est souvent assez sommaire, un GPS de poche ne contient vraisemblablement pas une mémoire de plus d'un ou deux mégaoctets, d'où la nécessité d'une forme de compression pour les données.

1.2.4 Application au monde des jouets

Même les jouets profitent de la compression. Qui, pendant sa tendre enfance, n'a pas eu un jouet parlant ? Ce genre de jouet activé par un bouton ou une ficelle qui débite quelques phrases-clefs totalement inintelligibles ? Si les concepteurs de ces jouets avaient alors disposé de méthodes de compression performantes pour le son, qui pourrait dire quel niveau d'éloquence ces jouets auraient atteint ? On voit d'ailleurs depuis quelque temps l'émergence sur le marché de jouets disposant d'un vocabulaire varié. Cela s'est fait grâce à l'électronique qui est moins dispendieuse que jamais, certes, mais aussi, et peut-être surtout, grâce à la compression qui permet d'enregistrer d'assez longues phrases et de les restituer avec une qualité sonore satisfaisante.

Dans la même veine, les microconsoles de jeu profitent aussi de la compression de données. Une machine comme le Gameboy de Nintendo est matériellement capable, selon le contrôleur de mémoire sur la cartouche de jeu, d'adresser jusqu'à huit méga-octets de mémoire morte (en fait, soixante quatre mégabits) et 128 kilo-octet de mémoire vive (un mégabit). Le coût des cartouches de jeu doit cependant être maintenu à un minimum pour que celles-ci demeurent attrayantes pour la clientèle cible. Bien qu'il soit possible de fabriquer une cartouche de jeu avec huit méga-octets de mémoire morte, il est beaucoup moins dispendieux de fabriquer des cartouches avec seulement quatre méga-octets, surtout si le jeu peut encore y tenir. Habituellement, on aura recours à des méthodes *ad hoc*. Celles-ci pullulent dans le monde des consoles et microconsoles de jeu. Chaque compagnie dispose d'une game complète de méthodes heuristiques et spécialisées. Ces méthodes forment un savant équilibre entre la qualité de la compression et la vitesse de décompression ; il sera évidemment nécessaire, dû à la faible vitesse d'horloge du processeur, d'avoir des algorithmes très rapides à la décompression.

Bien qu'habituellement, on suppose que les méthodes de compression soient *symétriques*, c'est-à-dire que le temps et les ressources nécessaires pour la compression soient comparables au temps et aux ressources nécessaires pour la décompression, les règles du jeu diffèrent dans le contexte de la production de cartouches de jeu ou des applications de poche. En effet, en télécommunication on veut des méthodes aussi rapides à la compression qu'à la décompression, tout comme c'est le cas, par exemple, pour les programmes archiveurs. Dans le cas où une version compressée des données n'est calculée qu'une seule fois (au moment de produire la maquette maîtresse pour la cartouche) et où les données compressées sont décompressées à maintes reprises (à chaque séance de jeu), il devient raisonnable de penser à utiliser des méthodes *asymétriques*, c'est-à-dire que le compresseur (qui n'est probablement même pas implémenté sur la microconsole, mais sur un ordinateur général rapide) peut utiliser des algorithmes ou des heuristiques coûteux en temps et en mémoire pour générer la version compressée des données, mais la méthode de décompression opérera très rapidement la restitution des données sur la microconsole.

La motivation pour utiliser des méthodes de compression sur des plate-formes de jeu est double. D'une part, nous avons les avantages économiques découlant directement d'une moins grande utilisation de mémoire. D'autre part, nous avons les avantages que l'on pourrait qualifier de légaux. Une méthode de compression *ad hoc* est exempte de royalties. Elle peut aussi, dans une certaine mesure, protéger contre les attaques de pirates ou de compétiteurs peu scrupuleux. On remarquera d'ailleurs que l'information précise sur ces méthodes se fait plutôt rare.

1.2.5 De la validité de la compression

La compression, dans le contexte d'objets peu dispendieux devant contenir une quantité assez importante d'information peut donc être validée par des considérations purement économiques. Bien que la mémoire soit relativement peu dispendieuse pour un ordinateur personnel de quelques milliers de dollars, elle demeure quand même comparativement dispendieuse lorsqu'on considère la fabrication d'un objet à quelques dollars. Moins l'objet est dispendieux à produire et plus il contient de mémoire (ne serait-ce que virtuellement, grâce à la compression), plus il est attrayant pour le consommateur et plus il est rentable pour le producteur.

La compression aide aussi à augmenter virtuellement les bandes passantes des médias de communication. L'utilisateur moyen y voit simplement une réduction des temps de transfert, mais les fournisseurs de services y voient une façon de retarder le remplacement des anciennes lignes et l'installation de nouvelles, qui se traduit directement par un avantage financier. Nous voyons donc que les raisons d'utiliser la compression ne manquent pas.

1.3 Qu'est-ce que la compression ?

La compression de données, de façon simplifiée, disions nous, c'est l'ensemble des méthodes permettant à un grand volume de données d'occuper un volume moindre, sans perte d'information significative. Remarquez que ces méthodes ne conservent pas nécessairement *toute* l'information, mais l'information *significative*. Cette subtilité a son importance. En effet, elle peut mener à la compression de données sans perte, où chaque bit qui est soumis à la méthode de compression est restitué correctement au moment de la décompression, comme elle peut mener à la compression avec perte, où les données sont transformées de façon à ne conserver que l'information pertinente, et où les données sont restituées de façon satisfaisante au moment de la décompression. Comparons donc les deux facettes de la compression : la compression sans perte et la compression avec perte.

1.3.1 Compression sans perte

La compression sans perte (on lira *lossless* en anglais) est principalement applicable aux données qui demandent une restitution exacte. Dans le contexte de la compression sans perte, la méthode prend en entrée une série de bits X qu'elle transforme en une nouvelle série de bits Y plus courte que X. La série de bits Y est transmise ou stockée pour usage ultérieur. Lorsque l'on veut récupérer les données, on prend Y et on applique la méthode de compression inverse — la méthode de décompression — pour récupérer X intact. La figure 1.4 illustre schématiquement le processus.

On voit en effet bien mal comment un programme pourrait subir les outrages de voir ne serait-ce qu'une infime partie de ses bits modifiés par la méthode de compression. La plupart des fichiers contenus sur un ordinateur ne tolèrent pas de voir leur bits changés. Ces documents deviennent inutilisables dès qu'une poignée de bits est corrompue. Sans nous lancer dans une digression sans fin, remarquons simplement que la grande majorité des logiciels ne sont pas conçus pour être tolérant aux erreurs dans leurs fichiers, comme on fait la supposition implicite que les données sont stockées de façon absolument fiable par le système. Si certains logiciels sont capable de détecter qu'un fichier est corrompu, la plupart des programmes généreront tout simplement une erreur mortelle, ce qui mettra fin à votre session de travail, possiblement avec

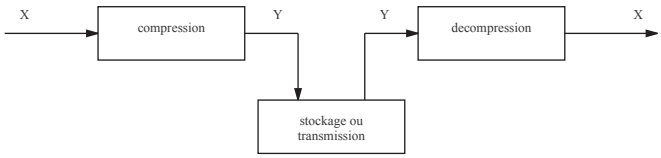


FIG. 1.4 – Une vision simplifiée de la compression sans perte.

des effets spectaculaires. Pour ce type de données, il est donc impératif d'utiliser une méthode de compression sans perte.

D'autres données nécessitent aussi une reconstruction exacte. Prenons par exemple la télémétrie provenant des sondes spatiales. Ces projets coûtent des sommes exorbitantes, certes, mais cela n'est pas ce qui donne de la valeur aux bits qui nous proviennent d'une sonde orbitant Jupiter. Ce qui leur donne de la valeur, outre leur évidente valeur scientifique, c'est que ces bits contiennent non seulement de l'information sur la télémétrie, mais aussi sur les instruments qui les ont captés. Détruire une partie de l'information nous priverait irrémédiablement de la possibilité de compenser pour un défaut du senseur. De plus, en éliminant de l'information, nous nous privons possiblement de données qui auraient permi à des tests plus sophistiqués d'extraire de nouvelles informations sur ces mondes lointains.

Enfin, en imagerie médicale, on insiste souvent pour avoir des données reconstruites exactement. Or, les images provenant d'un senseur à résonnance nucléaire sont extrêmement volumineuses. Ces senseurs produisent une image à la fois, chaque image correspondant à une « tranche » du patient. Habituellement on s'intéresse à une région en trois dimensions du patient, celle qui contient l'organe sous examen, ce qui générera une série d'images. Le nombre d'images est d'autant plus important que la région d'intérêt est grande. Pour éviter qu'un diagnostique erronné ne soit rendu dû à un effet secondaire de la méthode de compression, les données provenant d'une tomographie seront stockées sans perte, soit de façon crue, c'est-à-dire sans compression du tout, soit compressées avec un algorithme qui restitue correctement chaque bit. On observe aussi le même phénomène avec les radiographies classiques. Celles-ci sont numérisées à haute résolution et compressées sans perte pour réduire le risque qu'un mauvais diagnostique ne soit rendu par faute de détails détruits ou introduits par la méthode de compression.

1.3.2 Compression avec perte

La compression avec perte (on dira *lossy*, en anglais) joue un rôle tout à fait différent. La compression avec perte permet de choisir, judicieusement on l'espère, quelle information conserver et reconstruire et quelle information simplement détruire. Alors qu'avec la compression sans perte, les données restituées étaient identiques aux données avant la compression, avec la compression avec perte, on acceptera que les données restituées soient une approximation satisfaisante des données originales. Ce que l'on jugera alors comme une approximation satisfaisante dépend du champ d'application considéré. La figure 1.5 illustre le processus général. Ici, on compresse une série de bits X pour obtenir la série Y, qui est transmise ou stockée. Lorsqu'on décompresse Y on retourve Z, qui est potentiellement différent de X. Si Z diffère de X, il faudra alors que $Z \approx X$, selon une mesure appropriée.

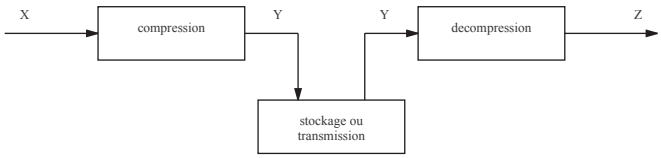


FIG. 1.5 – Une vision simplifiée de la compression avec perte. Ici, on suppose que $Z \approx X$, selon une mesure appropriée au type de données.

Pigeon␣␣␣␣␣␣␣␣␣Steven␣␣␣␣␣␣␣␣␣PIGS27542812◊

↓

Pigeon♦Steven♦PIGS27542812◊

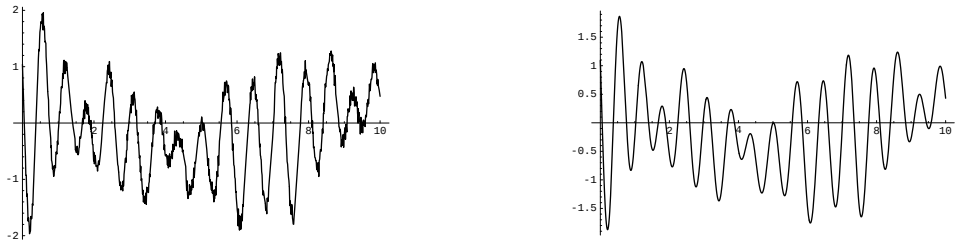
FIG. 1.6 – Une illustration de la méthode de compaction. Les espaces (symbolisés par ␣) sont remplacés par un symbol spécial ♦. La première version compte 43 symboles alors que la seconde version ne compte plus que 27 symboles. Les deux comptes incluent la fin de record, ◊.

Un exemple très simple d'algorithme de compression avec perte, c'est une méthode où les espaces répétés sont remplacées par un symbole spécial. On détruit de l'information, parce qu'après que les espaces aient été remplacés par le symbole spécial, on perd l'information nécessaire pour restituer les données originales[1]. On sait *où* se trouvaient les espaces, mais plus *combien* il y en avait. Cela ne poserait aucun problème dans une situation où le nombre d'espace importe peu, comme par exemple quand on sauvegarde une liste de records en format texte à partir d'un logiciel de base de données. Nous obtenons une certaine compression car une certaine quantité d'information — à laquelle correspond une certaine quantité de bits — a été détruite. Ce type de compression avec perte est parfois appelé *compaction* dans la littérature, pour le différencier par sa grande simplicité avec les méthodes de compression plus sophistiquées.

Un algorithme avec perte pourra, par exemple, éliminer le bruit d'un signal et n'encoder que le signal. Techniquement, nous avons eu une perte puisque le bruit, qui contenait aussi de l'information, a été détruit. Mais on peut argumenter que puisqu'il ne s'agissait que du bruit parasitant le signal, sa présence était plutôt malvenue, et que la restitution ne contienne que le signal ne constitue qu'un avantage supplémentaire. On gagne sur les deux tableaux : on a une meilleure compression et un signal nettoyé. La fig. 1.7 illustre le procédé.

Cette situation survient plus souvent que l'on pourrait le croire à prime abord. Il existe en effet une grande variété de signaux qui sont parasistés par une forme ou une autre de bruit que l'on voudrait bien voir disparaître. Tout ce qui a fait l'objet d'une conversion d'une forme analogique à une forme numérique s'expose à une introduction de bruit liée au processus d'acquisition. Prenons par exemple le cas où nous voudrions enregister un album de musique en studio. Les équipements sont, on le suppose, d'une qualité supérieure mais ils introduiront une faible quantité de bruit dans l'enregistrement. D'où vient ce bruit ? Les microphones sont des transducteurs électromécaniques. Ils traduisent les vibrations dans l'air (le son) en signaux élec-

1 Quoique dans ce cas particulier, on puisse connaître la longueur totale du record, nous permettant ainsi de déduire le nombre d'espaces perdues.



a) b)

FIG. 1.7 – Deux versions d'un signal hypothétique. a) Le signal original, bruité. b) Le signal restitué par la méthode de compression. Le bruit de faible amplitude est perdu, détruit par la méthode de compression, mais le signal demeure intelligible.

triques. Les transducteurs n'étant pas parfaits, ils introduisent une transformation sur le signal et une certaine quantité de bruit. Les signaux sont alors acheminés vers la console d'enregistrement où ils subiront la transformation numérique. Le processus de transformation numérique introduira lui aussi une certaine quantité de bruit, si infime soit-elle.

Pour enregister cette version numérique du son, il n'est pas nécessaire d'utiliser une méthode de compression qui restitue bit pour bit l'information originale, mais seulement une méthode qui rend la distinction entre la restitution et l'original à peu près impossible à faire. Il ne faut pas oublier qu'ultimement, c'est un humain qui jugera de la qualité de la restitution. Par exemple, si en studio on fait l'acquisition à 24 bits, et que le médium final n'en aura que 16 (comme c'est le cas pour un disque compact), il se peut fort bien qu'une restitution qui donne en moyenne 18 bits de précision soit plus que suffisante pour procéder au mixage ou à la fabrication des maquettes maîtresses.

Le monde de l'image n'échappe pas non plus à la compression avec perte. Les formats d'image courants utilisent le fait que l'oeil est particulièrement bon à éliminer le petit bruit de très haute fréquence pour faire disparaître cette information sans réellement affecter la qualité de l'image. Dans le contexte de la compression d'image, la définition du bruit n'est plus pas aussi intuitive que dans le contexte du son. Ici, le « bruit », c'est plutôt les fluctuations aléatoires, mais de faible amplitude, des pixels dans une région de l'image. Par hypothèse, ces fluctions sont du bruit seulement si elles ne sont pas spatialement corrélées ; ce qui est corrélé spatialement fait partie de l'image.

Nos yeux sont des machines très complexes qui prétraitent l'information reçue de la rétine avant de l'envoyer au cerveau. Parmi ces prétraitements, on retrouve une certaine élimination du bruit (car, bien que nous n'en soyons généralement pas conscients, les cellules de la rétine font des lectures bruitées), une mise en évidence des contrastes et une détection des contours. On retrouve aussi une sensibilité différente selon l'éclairage et les contrastes. Notre rétine perçoit aussi différemment les diverses couleurs, ce qui fait que nous percevons mieux certaines couleurs que d'autres. Il est donc possible d'imaginer un système où on ne code pas l'information de très haute fréquence et de faible amplitude, ce qui correspond au bruit (qui serait d'ailleurs filtré par nos yeux) et où on donne des pondérations différentes aux couleurs.

Le défi en compression avec perte est alors de savoir reconstruire les données de façon à ce que l'on ne puisse détecter la perte ; du moins la rendre tolérable. Dans le cas de la musique, il existe des algorithmes qui savent quelle information détruire sans nuire à la qualité perçue. Pour les images, il existe des standards complexes qui utilisent aussi cette stratégie pour produire des images restituées difficiles à différencier des originaux. Ces méthodes sont basées sur des modèles psychoacoustiques ou psychovisuels sophistiqués. Nous y reviendront lorsque nous considérerons plus avant la compression avec perte aux chapitres 4 et 9.

1.3.3 Les méthodes de compression

Les méthodes *ad hoc* sont des méthodes qui sont conçues pour des sources de données bien particulières et en général produisent d'assez mauvais résultats sur des données provenant d'autres sources. Ces heuristiques, bien que d'application très spécifiques, ne sont toutefois pas à dédaigner. Elles utilisent souvent toute l'information intuitive spécifique aux données d'intérêt et produisent ainsi, contre toute attente, des performances acceptables.

Les algorithmes généraux, par opposition aux méthodes *ad hoc*, ne se permettent que des suppositions très générales sur la classe de données qu'ils sont censés compresser. Les algorithmes généraux peuvent être avec ou sans perte, ils ne sont caractérisés que par le laxisme des prémisses sur les données. Considérons successivement les descriptions simplifiées des algorithmes sans perte puis avec perte.

Les trois principales catégories d'algorithmes sans perte sont les algorithmes à base de dictionnaire, les algorithmes à base de transformation et les algorithmes à base de modélisation statistique.

Les algorithmes à base de dictionnaire découpent les données en « mots » qui sont mis dans une structure de données quelconque, d'où il est possible de les récupérer par un index. Cette structure sera le « dictionnaire ». En simplifiant, la technique générale consiste alors à découper, séquentiellement, les données en « mots ». Si un mot n'est pas déjà dans le dictionnaire, on l'y ajoute, sinon on remplace le mot par son index dans le dictionnaire. On espère alors que tous les mots finiront par être déjà dans le dictionnaire, ce qui nous donnera le maximum de compression.

Les algorithmes à base de transformation appliqueront plutôt une transformation sur les données qui mettra en évidence les répétitions, de façon à pouvoir les mieux exploiter pour la compression. Ces algorithmes sont souvent suivi d'une autre étape de compression comme une méthode à base de dictionnaire ou de modélisation statistique.

Les algorithmes de modélisation statistique calculent des probabilités pour aider à la compression. Typiquement, on tentera d'estimer le plus précisément possible la fonction de distribution de la prochaine donnée, de façon à pouvoir générer un code efficace pour cette prochaine donnée. Ces algorithmes sont habituellement plus complexes et plus gourmands en ressources que les algorithmes à base de transformation ou de à base de dictionnaire, mais peuvent, en théorie, mener à une compression optimale.

Enfin, les principales méthodes de compression avec perte reposent sur des techniques de décomposition de signal et sur des méthodes de discrétisation (*quantization*, en anglais) de scalaires et de vecteurs. Ces méthodes commencent par décomposer le signal d'intérêt en sous-bandes.

Typiquement, cette décomposition est faite grâce à une technique d'analyse de fréquence de type transformée de Fourier (et ses descendants) ou plus récemment grâce aux techniques à base d'ondelettes. Nous reviendront sur le détail de ces opérations dans un chapitre ultérieur. Une fois la décomposition du signal obtenue, on procède à une analyse des composantes. Les composantes que l'on associe au bruit sont détruites et les composantes que l'on conserve seront discrétisées avant d'être codées par un algorithme de modélisation statistique. La discrétisation, c'est simplement une réduction de précision, mais une réduction intelligente, faite pour minimiser, en moyenne, l'erreur introduite dans la restitution par cette réduction. On peut ainsi coder des scalaires ou des vecteurs avec une précision moindre, c'est-à-dire en mettant des bits à la poubelle, sans toutefois affecter significativement la qualité de la restitution.

1.3.4 Mesures de performance

Jusqu'à maintenant, nous avons évité de parler de la performance des méthodes de compression. Dans la littérature technique on parlera interchangeablement de taux et de ratio de compression. Il existe plusieurs façon de voir la chose, mais j'opte pour la convention suivante. Posons n, le nombre de bits de la séquence de donnée originale, et m, le nombre de bits résultant de la méthode de compression.

Le *ratio* de compression est défini comme le ratio du nombre de bits de la séquence originale au nombre de bits produits par la méthode de compression. Le taux de compression est donc simplement n/m et s'exprime, par exemple, par $5:1$ si m est cinq fois plus petit que n.

Réciproquement, le *taux* de compression est défini comme le ratio de la différence entre le nombre de bits original et après compression sur le nombre de bit original, soit $(n-m)/n$. En prenant $n = 5$ et $m = 1$, on trouve que le taux de compression est 0.8, que l'on note parfois comme 80%. Notez que le ratio de compression n'est pas la réciproque du taux de compression.

On considérera qu'une méthode de compression est supérieure à une autre si son taux moyen de compression est plus élevé. Nous considérons que le ratio de compression indique mieux la performance que le taux de compression bien que les deux méthodes soient également mathématiquement valides. La raison en est que le ratio de compression est une mesure qui est dans un certain sens plus linéaire que le taux de compression, et, par le fait même, donne une meilleure idée de la compression obtenue. La notation, par exemple $5:1$, est aussi plus parlante.

Les ratios ou taux de compression ne sont pas les seuls indicateurs que nous pouvons utiliser pour évaluer la performance des méthodes de compression. Nous pouvons mesurer, dans le cas de méthodes avec perte, l'efficacité d'une méthode par rapport à la qualité de la reconstruction. Cette mesure de qualité peut être objective ou psychovisuelle ou psychoauditive. Dans le cas des mesures objectives nous retrouvons habituellement des mesures de distortion telles que le MSE ou encore le *signal to noise ratio*, ou SNR, en décibels. Nous reviendrons dans le détail sur ces mesures à la section 9.4. Dans le cas des mesures perceptuelles, nous aurons recours à des heuristiques basées sur des phénomènes physiologiques ou psychologiques documentés. Que nous utilisions une mesure objective ou subjective pour la qualité de la reconstruction, celle-ci est pondérée par le nombre de bits par éléments (échantillons, symboles, pixels, etc.), ce qui permet de comparer, à qualité de reconstruction égale, la performance en compression.

Nous pouvons évaluer la performance d'une méthode de compression — qu'elle soit avec ou

sans perte — grâce à des bornes inférieures théoriques, disons en terme du nombre de bits qui seraient strictement nécessaires pour représenter une séquence. Une séquence, dans ce contexte, c'est une série de données de longueur finie. Nous pouvons estimer le nombre de bits sans toutefois connaître la séquence compressée. Nous aurons recours à des mesures de complexité des séquences, telles que la mesure de complexité algorithmique de Kolmogorov et la mesure de complexité stochastique de Shannon. Nous introduirons dans le détail ces deux théories de l'information (qui sont essentiellement disjointes) au chapitre suivant.

Nous pouvons en effet considérer que la théorie de la complexité algorithmique des séquences est opposée, de par sa philosophie, à la théorie de la complexité stochastique. Dans le cadre de la complexité algorithmique, la complexité d'une séquence se mesure par le plus petit programme (sur une machine de Turing) capable d'émettre cette séquence. Il s'agit donc de trouver ou de générer un programme qui, une fois lancé, produit la séquence qui nous intéresse et puis s'arrête. Une séquence facile, très structurée, répétitive, recevera un programme générateur court, alors qu'une séquence compliquée, chaotique, recevera un programme plus long, voire même plus long que la séquence elle-même. Les séquences chaotiques sont plus nombreuses que les séquences pour lesquelles il existe un programme court. On remarquera cependant un aspect paradoxal de cette mesure de complexité : une séquence en apparence très compliquée, comme les décimales de π, peut être générée par un programme relativement court.

Mais cette théorie suppose que nous disposions de toute la séquence pour déterminer quel programme employer pour la produire. La théorie de la complexité algorithmique est donc une théorie d'information complète, où toute la séquence est connue. Cela nous donne le loisir de calculer la complexité globale de la séquence. Si la séquence est très courte, il y a peu de chances pour que sa structure intrinsèque soit exploitable : le programme qui la produit sera aussi long que la séquence. Si la séquence est au contraire très longue, s'il existe une structure exploitable, cela permettra de générer un programme plus court que la séquence. Cependant, la détermination du plus court programme produisant une séquence donnée est un problème *indécidable*.

La théorie de la complexité stochastique, élaborée par Shannon, propose une autre vision. Alors que la théorie de la complexité algorithme cherche à expliquer la complexité d'une séquence par la longueur du plus petit programme qui la produit, la théorie de la complexité stochastique mesurera la complexité d'une séquence par la facilité à prédire le prochain symbole de la séquence à partir d'un modèle probabiliste. Le modèle probabiliste peut être simple. Par exemple la prédiction du prochain symbole dans la séquence peut être formulée grâce à une distribution inconditionnelle des symboles dans la séquence. Le modèle pourra aussi être plus complexe, et l'estimation des probabilités pour le prochain symbole dans la séquence pourra dépendre de tous les symboles qui le précèdent depuis le début de la séquence. Si le modèle est adéquat et la séquence est facile à compresser, alors la distribution de probabilité sur le prochain symbole sera pointue. Si au contraire la séquence est difficile bien que le modèle soit adéquat, nous obtiendrons des distributions plus évasées. Une façon de caractériser quantitativement ces distributions de probabilités, donc ultimement la complexité de la séquence, c'est la mesure d'*entropie*.

L'entropie est une métrique de « surprise » dans la mesure où, si l'entropie est élevée, les prédictions sont difficiles à faire et si elle est faible, la séquence est facilement prévisible. Si une séquence contient beaucoup de « surprises », elle contient beaucoup d'information. Si elle ne contient pas beaucoup de « surprises », elle ne contient pas beaucoup d'information. L'analogie entre l'entropie de l'information et l'entropie thermodynamique est judicieuse : l'entropie thermo-

dynamique élevée correspond au chaos et l'entropie faible à l'ordre. Intuitivement, nous voyons pourquoi une séquence facile à prévoir contient peu d'information et pourquoi, par conséquent, elle devrait pouvoir être compressée facilement, alors qu'une séquence compliquée contiendra plus d'information et sera plus difficile à compresser.

Les deux théories se complètent toutefois assez bien. Dans les deux théories, les séquences « faciles » auront des représentations compressées courtes, et les séquences « compliquées » auront des représentations compressées longues. Les deux théories permettent de définir des complexités conditionnelles entre les séquences. Si deux séquences sont dépendantes d'une quelconque façon, connaître l'une des séquences nous aide à compresser l'autre. Les deux théories nous permettent d'élaborer une hiérarchie de complexité des séquences. Quoi qu'il en soit, nous verrons au chapitre 3 pourquoi c'est en général la théorie de la complexité stochastique qui est utilisée. Comme c'est la théorie de Shannon qui occupe la place prépondérante, nous nous référerons, au long de la thèse, aux mesures de complexité stochastique.

❁ ❁ ❁

Comparer une méthode de compression aux bornes inférieures estimées par l'une ou l'autre des théories nous permet de jauger de l'efficacité de la méthode autrement que par une mesure relativement subjective comme le taux de compression. Un ratio de compression peut en effet être trompeur. Si un ratio de compression de 5 : 1 peut sembler bon, la complexité algorithmique ou stochastique pourrait cependant prédire un ratio de compression atteignable de 12 : 1, ce qui indiquerait au contraire que la méthode de compression est relativement mauvaise.

1.4 Notes bibliographiques

Le lecteur intéressé aux communications des époques anciennes, médiévales et jusqu'à la seconde guerre mondiale pourra lire le livre de Simon Singh, *The Code Book* [191] qui décrit les méthodes utilisées historiquement ainsi que nombre anecdotes s'y rattachant. On pourra aussi lire *In Love and War*, du vice-amiral Jim Stockdale et de son épouse Sybil Stockdale, où les tourments des camps de prisonniers sont dépeints [198]. On y trouve aussi les *tap codes*. La paternité des codes doit cependant être attribuée à un officier anonyme des forces de l'air de l'armée américaine. Le capitaine Carlyle Harris fut fait prisonnier en 1965 et séjourna au camps de Hoa Lo, où il enseigna le code, qu'il avait lui-même appris de cet officier inconnu, aux autres prisonniers. En, il est vraisemblable cet officier fut quelque peu érudit. En effet, on retrouve ce code chez Polybe (circa 210 – 126 avant J.C), un historien grec [169, 108].

Pour une biographie de Samuel Morse, on peut s'en référer, par exemple, à *An American Leonardo : A life of Samuel F. B. Morse* [131]. Bien que Morse soit reconnu comme l'inventeur du code du même nom, il semblerait qu'il s'agisse en fait d'une invention de son assistant, Alfred Vail. Paru dans le journal *The Century : Illustrated Monthly Magazine*, en avril 1888, l'article "The American Inventors of the Telegraph, with special references to the services of Alfred Vail", écrit par Franklin Pope, lui aussi inventeur de technologie télégraphique et associé de Thomas Edison, présente Alfred Vail, l'assistant de Morse, comme l'auteur du code :

> *Alfred's brain was at this time working at high pressure, and evolving new ideas every day. He saw in these new characters the elements of an alphabetical code by which language could be telegraphically transmitted in actual words and sentences, and he instantly set himself at work to construct such a code. His general plan was to employ the simplest and shortest combinations to represent the most frequently recurring letters of the English alphabet, and the remainder for the more infrequent ones. For instance, he found upon investigation that the letter e occurs much more frequently than any other letter, and accordingly he assigned to it the shortest symbol, a single dot (·). On the other hand, j, which occurs infrequently, is expressed by dash-dot-dash-dot (− · − ·). After going through a computation, in order to ascertain the relative frequency of the occurrence of different letters in the English alphabet, Alfred was seized with sudden inspiration, and visited the office of the Morristown local newspaper, where he found the whole problem worked out for him in the type cases of the compositor. [...] In this statement I have given the true origin of the misnamed "Morse" alphabet the very foundation and corner-stone of a new system, which has since become the universal telegraphic language of the world.*

Quant à la vie de Louis Braille, le lecteur pourra s'en référer à la biographie de Pierre Henri [95]. On ne pourrait parler du code Braille sans parler du capitaine de cavalerie Nicolas-Marie-Charles Barbier de la Serre (1767 – 1841). De la Serre s'intéressera toute sa vie aux écritures secrètes et rapides. En 1821, il présente, à l'Institution Royale des Jeunes Aveugles de Paris, un système d'écriture rapide et nocturne, à être taillée au canif dans une plaquette, destinée aux applications militaires. Braille, alors âgé de seulement douze ans, assiste à la présentation. Il proposera des améliorations substancielles au système de De la Serre qui aboutiront éventuellement au code de Braille tel que nous le connaissons.

Chapitre 2

Plan et objectifs de la thèse

L'objectif de cette thèse est de présenter nos contributions à la compression de données. Le texte entier n'est pas consacré à nos seules contributions. Une large part est consacrée au matériel introductif et à la recension de littérature sur les sujets qui sont pertinents à nos contributions. Ainsi, quatre chapitres couvrent le matériel introductif et quatre chapitres présentent nos contributions.

Les chapitres de matériel introductif, outre le chapitre 1, l'introduction à la compression que vous avez déjà lue, présentent dans un plus grand détail les enjeux et les aspects de la compression de données. Il ont aussi pour but la récension des travaux qui sont directement pertinents à nos contributions.

Le premier de ces chapitres, le chapitre 3, *Mesures de Complexité*, présente les deux grandes théories de la complexité des séquences et introduit quelques relations avec d'autres disciplines, comme l'apprentissage automatique et la physique statistique. Ce chapitre introduit les notions de complexité de façon à justifier les méthodologies qui seront utilisées dans les autres chapitres. Il faut en effet disposer d'une façon précise, faisant l'objet d'un consensus, de mesurer la complexité intrinsèque d'une séquence (soit individuellement soit en tant que membre d'une classe de séquences) pour jauger adéquatement les méthodes de compression.

Le second chapitre introductif, le chapitre 4, *Les paradigmes de la compression*, couvre les grandes familles d'algorithmes de compression. Nous y discourons de la compression sans et avec perte. Les algorithmes sans perte restituent exactement les données originales à partir des données compressées. Les algorithmes avec perte se permettent de détruire une certaine quantité d'information à la compression de façon à atteindre un meilleur taux de compression tout en restituant une version acceptable des données à la décompression. Ce que constitue une version acceptable dépend du type de données et de l'application considérée. On s'entend pour dire que « acceptable » correspond à une perte qui est imperceptible ou au moins qui n'est pas dérangeante. Les algorithmes de compression avec perte s'appuient généralement sur des transformées de décomposition en fréquences, comme la transformée de Fourier et ses descendants ou encore sur les transformées ondelettes. Nous parlerons des unes et des autres. Le résultat de ces transformées est alors codé en précision réduite, ce qui permet de détruire un certain nombre de bits, grâce à une méthode de discrétisation et un codeur efficace. Les méthodes de discrétisation sont aussi introduites. Les algorithmes sans perte insistent pour restituer exactement chaque bit de

la version originale des données. La compression sans perte est nécessaire dans beaucoup d'applications. Ces algorithmes découpent généralement les données de façon à mettre en évidence les répétitions et en tirer quelque compression. Ces algorithmes sont habituellement basés sur la notion de fenêtre coulissante ou de dictionnaire, bien qu'un certain nombre utilise plutôt des transformations de la séquence, comme des permutations.

Le chapitre 5, *Le codage des entiers* aborde le problème de la compression de séquences aléatoires d'entiers. Les codes utilisés pour le codage ces entiers dans les protocoles de compression sont souvent issus de méthodes *ad hoc*, c'est-à-dire conçues pour satisfaire empiriquement un certain nombre de contraintes. Par exemple, le code doit être simple, facile et rapide à encoder et décoder, ne pas être beaucoup plus long en moyenne que le code optimal, etc. Ces codes sont souvent conçus en fonction de facteurs limitatifs comme la relativement petite taille maximale des entiers à coder ou la faible puissance du processeur utilisé. Les codes dits universels sont utilisés lorsqu'on veut coder des entiers dont la taille maximale n'est pas bornée ; c'est-à-dire qu'il est possible que nous ayons à coder un entier arbitrairement grand. Ces codes utilisent aussi des hypothèses très peu restrictives sur la distribution dont sont issus les entiers à coder. Par exemple, la distribution est telle que tout entier a une probabilité non nulle d'être observée mais les plus petits entiers ont les probabilités les plus élevées. Enfin, nous avons les codes où le domaine est très restreint. Nous avons aussi les codes énumératifs, où il s'agit d'encoder un choix binomial, c'est-à-dire m parmi n, soit un des $\binom{m}{n}$ choix possibles.

Enfin, le chapitre 6, *Codage Huffman Adaptatif* présente l'algorithme de Huffman pour le codage des entiers tirés selon une distribution arbitraire. Le but de l'algorithme de Huffman c'est de produire des codes de longueur entière qui s'approchent le plus possible de l'entropie. Nous commençons cependant par présenter l'algorithme de Shannon et Fano qui approxime l'algorithme de Huffman. L'algorithme de Huffman utilise l'information *a priori* sur la distribution pour générer un code statique, c'est-à-dire un code qui demeurera le même tout le long de la compression d'une séquence. Nous présenterons des algorithmes dynamiques, c'est-à-dire des algorithmes qui adaptent les codes au fur et à mesure que les symboles de la séquence sont observés. Nous présenterons les algorithmes de Faller, Gallager et Knuth, ainsi que la variante Λ de Vitter. Ces algorithmes sont nécessaire pour la compréhension de nos contributions présentées au chapitre *Codage Huffman Adaptatif*.

Vous trouverez, à certains endroits, dans les chapitres introductifs et dans les appendices, des sections ou des paragraphes marqués d'une étoile ★. Ces sections présentent des contributions qu'il eût été difficile ou inutile de mettre à part dans les chapitres dits de contribution. Nous avons jugé qu'il était préférable de les insérer à même le texte qui les introduisait de façon à maintenir le flot des idées. Ces sections se retrouvent dans le chapitre sur le codage des entiers, soit la section 5.3.4.2 sur les codes *phase-in* et la section 5.3.6.3 sur le pairage des entiers, et à l'appendice E, section E.7.

Ces chapitres sont suivis par nos contributions. Le premier chapitre de contribution, le chapitre 7, *Contributions au codage des entiers* se concentre sur le problème de la génération de codes efficaces pour les entiers. Le chapitre 8, *Codage Huffman Adaptatif* présente deux nouveaux algorithmes pour la génération dynamique de codes structurés en arbre, soit des codes de type Huffman. Le chapitre 9, *Compression d'image à base de palettes : LZW avec perte* explore le problème de la compression d'images comportant un petit nombre de couleurs distinctes et propose une extension avec perte d'un algorithme originalement sans perte, LZW. Enfin, le der-

nier chapitre, le chapitre 10, *Les pseudo-ondelettes binaires* présente une solution originale au problème de l'analyse multirésolution des images monochromes, c'est-à-dire des images ayant exactement deux couleurs, conventionnellement noir et blanc. Ce type d'image correspond par exemple aux images textuelles telles que produites par un processus de transmission de type facsimilé.

Le chapitre 7, *Contributions au codage des entiers* présente nos contributions aux divers sujets touchés par le codage des entiers. Nous commençons par présenter une amélioration à un algorithme très simple, le codage par bigramme, qui en augmente significativement l'efficacité. Nous passons par la suite au codage de Golomb, un code qui sert à encoder des entiers tirés selon une loi géométrique. Nous présentons de nouvelles solutions pour des variations de ce code ; ces solutions ne font pas intervenir de notions asymptotiques. Nous irons par la suite du côté des codes universels. Nous présentons deux nouvelles classes de codes universels qui permettent d'encoder un très grand entier i en $O(\lg i)$ bits. Ces codes existent en deux familles : les codes alignés et les codes non contraints. Les codes alignés sont structurés en blocs de n bits et les codes non contraints peuvent prendre n'importe quelle longueur de la forme $k + n$ pour $k \in \mathbb{Z}^*$ et $n \in \mathbb{N}$. Ces codes nous mèneront aux calculs efficaces des nombres de Fibonacci généralisés, notés $F_i^{(n)}$, et de la somme des s premiers nombres généralisés, $\sum_{i=1}^{s} F_i^{(n)}$, deux problèmes pour lesquels nous présentons de nouvelles solutions. Nous présenterons de nouveaux codes, les codes $Start/Stop$, une généralisation des codes $(Start, Step, Stop)$ et des algorithmes d'optimisation pour ces deux classes de codes. Nous verrons aussi comment ces algorithmes peuvent être modifiés pour permettre une optimisation qui tienne compte des contraintes inhérentes à la machine sous-jacente. Finalement, nous montrerons comment les codes $(Start, Step, Stop)$ et $Start/Stop$ peuvent être transformés en codes universels.

Le chapitre 8, *Codage Huffman Adaptatif* présente deux nouveaux algorithmes, l'algorithme W et l'algorithme M qui découlent de modifications à l'algorithme de Jones. L'algorithme de Jones maintient un arbre de code de type Huffman grâce à un arbre *splay* où la règle de *splaying* classique a été remplacée par une règle qui, plutôt que de propulser la dernière feuille accédée à la racine, réduit la distance qui la sépare de la rachine par deux. Cette forme d'adaptativité pondérée permet à l'algorithme de Jones d'obtenir des codes relativement bons. L'algorithme W utilise aussi la notion de *splaying*, mais applique une règle différente pour décider jusqu'où monte un nœud dans l'arbre. L'algorithme M augmente l'algorithme W en insistant que les symboles de même fréquence aient un code de même longueur. Ces deux algorithmes font beaucoup mieux que l'algorithme de Jones mais restent légèrement inférieurs à l'algorithme Λ de Vitter. Nous explorerons aussi les complications encourues lorsque nous considérons des alphabets ayant un très grand nombre de symboles.

Le chapitre 9, *LZW avec perte* présente l'algorithme de Welch et son application à la compression d'image à base de palette. L'algorithme de Welch est une variante de la classe LZ78, les algorithmes qui compressent en cherchant les concordances entre la séquence à compresser et les séquences contenues dans un dictionnaire. Cet algorithme est utilisé dans un protocole de compression d'image commun, le protocole GIF de Compuserve. Ce protocole permet de compresser des images d'au plus 256 couleurs distinctes, ce qui entraîne la nécessité d'un algorithme de réduction de couleur et de *dithering* pour amoindrir les effets de cette réduction. Nous verrons comment le *dithering* affecte la compression. Par la suite, nous présentons deux nouvelles façons de modifier cet algorithme de façon à le rendre avec perte sans trop abîmer l'image reconstruite. La première variante cherche les concordance de façon vorace dans le dictionnaire sans chercher

à optimiser globalement un critère tel que le taux de compression ou la qualité moyenne de reconstruction de l'image. La seconde version cherche dans le dictionnaire la meilleure concordance selon un critère fourni par l'utilisateur. Ce critère pourrait être de maximiser la longueur des concordance, ou miniser l'erreur de reconstruction ou plus utilement une balance savamment étudiée de ces deux facteurs. Nous comparerons nos résultats à un autre algorithme du même type proprosé par d'autres auteurs. Nous verrons que nos algorithmes donnent à la fois une meilleure compression et une meilleure qualité d'image.

Enfin, le chapitre 10, *Les pseudo-ondelettes binaires* présente une solution originale au problème de l'analyse multirésolution des images binaires. Ces images ne comportent que deux couleurs, que l'on peut associer arbitrairement à 0 et 1. Cette classe d'image correspond aux images produites par les télécopieurs ou telles qu'imprimées dans les journaux. Les images imprimées dans les journaux utilisent un procédé de tons simulés, le *halftoning* qui permet de faire percevoir des tons de gris alors qu'en fait seuls des points de tailles différentes composent l'image. Nous verrons comment ces *pseudo-ondelettes*, une nouvelle classe d'ondelettes que nous présentons ici, peuvent être utilisées pour la transmission progressive d'images textuelles et en *halftones*. Nous verrons qu'il pourrait être possible d'utiliser cette décomposition pour obtenir un schème de compression avec perte des images binaires.

Les chapitres de contribution seront suivis des inévitables appendices, bibliographie et index analytique.

Chapitre 3

Mesures de Complexité

Au chapitre 1, nous avons effleuré le sujet de la mesure de la complexité intrinsèque des séquences. Nous avons introduit deux théories de l'information : la théorie de la complexité algorithmique et la théorie de l'information stochastique. La théorie de la complexité algorithmique mesure l'information contenue dans une séquence en termes de la longueur du plus petit programme qui génère cette même séquence, alors que la théorie de la complexité stochastique jauge la quantité d'information contenue dans une séquence par la difficulté de prédire un bit dans la séquence étant donnés ceux qui le précèdent. Ces deux théories, si elles ne sont pas les seules capables d'expliquer la nature de l'information, sont certainement les deux plus importantes et les mieux développées.

Les mesures théoriques du contenu d'information d'une séquence ou d'une classe de séquences nous serviront à étalonner précisément nos diverses méthodes de compression. Si nous pouvons déduire, grâce aux mathématiques, la quantité d'information présente dans les séquences qui nous intéressent, sous certaines hypothèses, nous serons en mesure de déterminer, pour telle ou telle séquence, combien de bits seraient strictement nécessaires pour représenter cette séquence sous forme compressée. Les théories de l'information algorithmique et stochastique nous fournissent justement les outils pour calculer, à partir de nos hypothèses, ce nombre de bits pour une séquence, ou une classe de séquences, donnée.

Dans ce chapitre, nous présenterons d'abord, à la section 3.1, une brève introduction aux différentes représentations des données. Cette section sera d'une importance capitale car nous y introduirons des notions qui seront utilisées dans le reste de cette thèse. Nous y montrerons, en particulier, comment toute séquence de symboles se ramène à une séquence de bits. Dans les sections suivantes, 3.2 et 3.3 respectivement, nous introduirons les théories de la complexité algorithmique et de la complexité stochastique. Nous comparerons finalement les deux théories à la section 3.4 et expliquerons pourquoi la théorie de la complexité stochastique s'est imposée face à la théorie de la complexité algorithmique.

3.1 Représentation des séquences

Une séquence, c'est une suite de symboles ou de nombres. S'il s'agit d'une suite de symboles, ces symboles seront tirés d'un alphabet. Comme un texte, les symboles se suivent dans un ordre défini et cette séquence est composée d'un nombre fini de symboles. Quand à l'alphabet, il suffit

qu'il soit un ensemble énumérable, au sens mathématique du terme. Les naturels, $\mathbb{N}$, seraient un tel « alphabet ».

Présentons donc quelques définitions. Les définitions qui suivent seront utilisées dans ce qui suivra, tout le long de la thèse.

Définition 3.1.1 Alphabet fini. Soit un alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, une énumération finie de symboles arbitraires. Cet alphabet est partiellement ordonné si on peut écrire $\sigma_0 \leqslant \sigma_1 \leqslant \ldots \leqslant \sigma_n$. L'alphabet est strictement ordonné si $\sigma_0 < \sigma_1 < \ldots < \sigma_n$; on dira alors qu'il existe un *ordre lexicographique* des symboles (littéralement l'« ordre du dictionnaire »).

Exemple. $S = \{a, b, c, d, \ldots, z, \sqcup\}$.

Définition 3.1.2 Alphabet infini. Soit un alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$, une énumération infinie de symboles arbitraires. Cet alphabet est partiellement ordonné si on peut écrire $\sigma_0 \leqslant \sigma_1 \leqslant \ldots$. L'alphabet est strictement ordonné si $\sigma_0 < \sigma_1 < \ldots$; on dira alors qu'il existe un ordre lexicographique des symboles.

Exemple. Les naturels $\mathbb{N}$, les entiers $\mathbb{Z}$.

Définition 3.1.3 Séquence. Une séquence est une suite ordonnée et finie de symboles tirés d'un alphabet de référence.

Définition 3.1.4 Longueur de séquence. La longueur d'une séquence a est notée $|a|$, et donne le nombre de symboles qui composent a.

Définition 3.1.5 Ensemble universel. L'ensemble universel d'un alphabet S est formé de tous les « mots » formés des « lettres » de S de longueur 0,1,2,3,... Posons S^n, le produit cartésien de S avec lui-même n fois. S^0 représente le « mot » vide, noté $\perp$. Soit $S^* = S^0 \cup S \cup S^2 \cup S^3 \cup \ldots$, l'ensemble universel de S. Notez qu'aucun mot dans l'ensemble universel n'est de longueur infinie. Pareillement à l'ensemble universel, on définit $S^+ = S \cup S^2 \cup S^3 \ldots$, ce qui est équivalent à S^* moins $\perp$.

Exemple. Si $S = \{0, 1\}$, $S^* = \{\perp, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots\}$.

Définition 3.1.6 Fonction de pairage. Soit $\langle\, a, b \,\rangle$, la fonction de pairage, qui assemble a et b de façon réversible, c'est-à-dire que de la séquence résultant du pairage, on peut retrouver a et b. La fonction de pairage respecte $|\langle\, a, b \,\rangle| \geqslant |a| + |b|$. □

Nous présentons des fonctions de pairage à la section 5.3.6.2, p. 112.

Définition 3.1.7 Concaténation. La concaténation de deux mots a et b est noté $(\, a : b \,)$, et si $a, b \in \Sigma^*$, alors $(a : b) \in \Sigma^*$. Contrairement au résultat de la fonction de pairage, le résultat de la concaténation n'est pas nécessairement réversible ; c'est-à-dire qu'en ne connaissant que $(\, a : b \,)$ on ne peut pas toujours retrouver a et b. La concaténation respecte toujours $|(\, a : b \,)| = |a| + |b|$. □

Les séquences sont emmagasinées en mémoire ou sur un médium de stockage. Comme la mémoire est composée d'un certain nombre de cellules indifférenciées, une séquence peut être stockée n'importe où. La mémoire est telle que les cases mémoire qui suivent la fin d'une séquence ne diffèrent en rien des cases mémoire qui servent à emmagasiner la séquence elle-même, et à moins de déjà connaître la longueur et la position de la séquence, il n'y a aucun moyen évident pour déterminer où se trouve la fin de la séquence.

De même, si on emmagasine de façon contiguë en mémoire plusieurs séquences, on se trouve à utiliser l'opération de concaténation (déf. 3.1.7) qui ne rend pas toujours possible la restitution des séquences qui ont été concaténées. Non seulement nous ne connaissons plus les séquences originales, mais nous n'avons pas non plus délimité la séquence qui résulte de la concaténation. Si on a plutôt recours à la fonction de pairage (déf. 3.1.6), il reste encore à délimiter la séquence pairée elle-même avant de pouvoir en extraire les séquences composantes.

Il nous faut chercher des moyens de délimiter explicitement les séquences. Il existe essentiellement trois façon de délimiter les séquences : les séquences syntaxiquement délimitées (ou « autodélimitantes »), les séquences ayant leur longueur explicitement encodée, et les séquences délimitées par un marqueur de fin de séquence.

Avec les séquences syntaxiquement délimitées, il est possible de déterminer quand on a rejoint la fin de la séquence grâce à sa structure syntaxique. Par exemple, pour un programme LISP, on est capable de déterminer la fin de programme à partir du programme lui-même, sans information auxiliaire. Un programme LISP est tel que, pour chaque parenthèse ouvrante il y a une parenthèse fermante correspondante ; et le programme complet est ceint par une paire de parenthèses. Il est alors possible de déterminer, en parcourant le programme dans l'ordre de lecture normal, où il se termine. Il existe d'autres structures qui permettent de déterminer quand on a atteint la fin de la séquence. On n'a qu'à penser aux poèmes haïku qui ont exactement dix-sept syllabes ; quand on a lu la dix-septième syllabe, on sait que le poème est complet. L'information nécessaire à l'encodage de la longueur ou de la fin de séquence est pour ainsi dire récupérée par les contraintes sur la structure de la séquence : les séquences étant plus structurées, elles existent en moins grande variété que les séquences qui ne respectent pas cette structure autodélimitante (et qui composeront la quasi totalité de Σ^*), elles sont plus longues que suggérerait leur contenu d'information réel.

Les séquences dont la longueur est explicitement encodée sont en fait des tuples formés d'un entier et de la séquence elle-même. Elles diffèrent entre elles principalement par la façon dont le tuple est représenté. Si la séquence est de longueur n, on doit s'attendre à trouver $O(\lg n)$ bits la précédant (voir la section 5.3.3), et ces $O(\lg n)$ bits encodent la longueur — pour une introduction à la notion d'ordre, le lecteur est encouragé à lire, par exemple, [26]. Ces séquences peuvent être de la forme $\langle\, |s|, s \,\rangle$, pour une séquence $s \in \Sigma^*$. Ces séquences n'ont pas *a priori* à respecter une structure aussi contraignante que les séquences autodélimitantes. La tactique qui consiste à encoder les longueurs de séquences explicitement est la plus souvent utilisée.

D'autres séquences sont délimitées grâce à un symbole spécial, disons $\diamondsuit$, de telle sorte que leur alphabet est *augmenté* de ce symbole spécial. On aura Σ, l'alphabet de base, qui deviendra $\Sigma' = \Sigma \cup \{\diamondsuit\}$. Ces séquences sont alors de la forme $(\, s : \diamondsuit \,)$. Pour déterminer la fin de la séquence, il suffit de lire un à un les symboles (tirés de Σ') jusqu'à ce que l'on rencontre le symbole spécial de fin de séquence, $\diamondsuit$. Le problème avec cette méthode, c'est qu'il n'est pas

toujours pratique d'augmenter l'alphabet des séquences pour y inclure le symbole spécial $\diamondsuit$. Si la taille de l'alphabet est limitée pour une raison quelconque, nous devrons utiliser des astuces supplémentaires pour include virtuellement le symbole spécial de fin de séquence dans l'alphabet.

Une fois que la séquence est délimitée d'une façon ou d'une autre, il nous reste à la représenter dans mémoire. Typiquement, nous aurons recours à une représentation utilisant l'alphabet naturel de la séquence. Cet alphabet pourra être encodé par une représentation binaire. Le théorème suivant nous sera d'une grande utilité :

Théorème 3.1.1 Représentation des séquences. Soient $\mathbb{B} = \{0, 1\}$, l'alphabet binaire et Σ un alphabet (il peut être fini ou non). Toute séquence formée à partir d'un alphabet Σ se représente par une séquence formée à partir de $\mathbb{B}^+$.

Démonstration. Premier cas, l'alphabet est fini. Soit $n = |\Sigma|$, le nombre de symboles différents dans l'alphabet Σ. Posons $m = \lceil \lg n \rceil$. Créons $B_m = \mathbb{B}^m$, les mots formés de m bits. B_m possède $|B_m| = 2^m \geqslant n$ mots. Il suffit alors d'associer à chaque $\sigma \in \Sigma$ un $b \in B_m$, de façon à ce que chaque σ reçoive un b différent ; l'arrangement naturel étant de créer l'association $\{(\sigma_0, b_0), (\sigma_1, b_1), \ldots, (\sigma_{n-1}, b_{n-1})\}$. Deuxième cas, l'alphabet est infini. Plutôt que d'utiliser $\mathbb{B}^m$, nous utiliserons des mots de $\mathbb{B}^+$ pour représenter les symboles de l'alphabet infini. Il suffira de prouver qu'il existe une correspondance entre chaque symbole de l'alphabet et une chaîne de bits unique. Pour chaque σ_i on utilisera le code unaire, $C_\alpha(i)$, comme mot dans $\mathbb{B}^+$ (voir l'appendice A pour les codes unaires). Le code $C_\alpha(i)$ est composé de i uns suivi d'un zéro terminant la séquence. Par exemple, $C_\alpha(0) = 0$, $C_\alpha(1) = 10$, $C_\alpha(2) = 110$, etc. Pour trouver i, à partir de $C_\alpha(i)$, il suffit de compter combien de uns nous lisons avant de rencontrer le zéro. L'arrangement devient, dans le cas d'un alphabet infini, l'association $\{(\sigma_0, 0), (\sigma_1, 10), (\sigma_2, 110), \ldots\}$. De plus, la séquence peut être finie ou infinie, le résultat tient. ■

Corollaire 3.1.1 Par le théorème précédant, tout symbole de tout alphabet est représentable par une chaîne de bits.

Le théorème 3.1.1 nous dit que tout alphabet peut être encodé de façon binaire, et, par conséquent, toutes les séquences. Ce théorème nous permet de considérer tous les alphabets sur un pied d'égalité. En effet, puisque tout alphabet se ramène à une représentation binaire, aucun ne jouit d'un statut particulier, et les alphabets de même cardinalité sont alors pour ainsi dire tous équivalents. Cette équivalence pourra servir à montrer comment se comparent deux séquences d'alphabets différents par le biais de la représentation binaire. Il est alors facile d'associer des symboles de différents alphabets aux mêmes chaînes de bits, rendant possible une complète uniformisation du raisonnement quant aux séquences. Dans ce qui suivra, on pourra, sans perte de généralité, supposer que toutes les séquences sont composés de bits.

3.2 Complexité de Kolmogorov

Les idées menant au développement complet de la théorie de la complexité algorithmique des séquences prennent racines dans les années 1960. Les contributions principales ont été apportées par Andrey Nikolaevich Kolmogorov (1903–1987) [115, 116, 117, 118, 119, 120], R. J. Solomonoff [193, 194, 195] et Gregory J. Chaitin [38, 39, 40, 41, 42, 43]. En toute justice, on devrait parler de

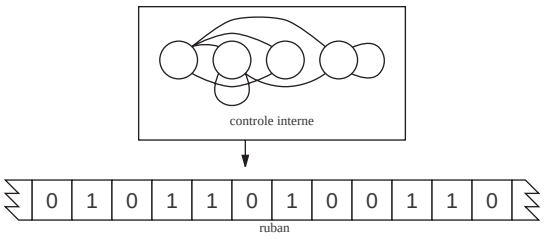


FIG. 3.1 – La machine de Turing avec son ruban.

la théorie de la complexité de Kolmogorov-Solomonoff-Chaitin, mais pour des raisons historiques fortuites seul le nom de Kolmogorov a été retenu dans l'usage. Nous retrouvons par ailleurs cette même théorie sous des noms variés, tels que théorie de la complexité des descriptions, théorie des descriptions de longueurs minimales, *Kolmogorov-Chaitin Randomness*, etc.

La théorie des probabilités classique semble ne pas pouvoir rendre compte de certains phénomènes intrinsèques aux séquences, et il a paru nécessaire de suppléer à ces déficiences par une nouvelle théorie qui ne s'intéresse pas à la probabilité d'occurence d'une séquence en particulier mais à ses structures internes. Considérons l'expérience suivante. Prenons une pièce de monnaie et jetons-la dix fois. À chaque tirage, nous constaterons que la pièce montre soit le côté face (0), soit le côté pile (1). Supposons aussi que la pièce ne soit pas truquée et que la probabilité d'avoir l'une ou l'autre des faces soit exactement $\frac{1}{2}$. La théorie classique des probabilités assignera une probabilité égale à toute série de dix jets (soit 2^{-10}) ce qui suggère que toutes les séquences de dix pile ou face contiennent exactement la même quantité d'information. Pourtant, intuitivement, la séquence 0000000000 nous semble bien moins aléatoire que, disons, 0110101001.

Il fallait donc trouver une définition satisfaisante de séquence aléatoire. Von Mises[1] proprosa de définir une séquence s comme étant aléatoire si le fait de connaître ses t premiers bits ne nous permet pas de prédire le $(t+1)$-ième bit autrement qu'en estimant la probabilité que le prochain bit soit un 1, étant donné la fraction des t premiers bits qui sont à un. Autrement dit, $P(s_{t+1} = 1) = |s_1^t|_1/t$, où $|x|_1$ compte le nombre de bits à un dans la séquence x, et où x_i^j est une abbréviation pour la séquence $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$.

Cette définition d'aléatoire n'est pas tout à fait satisfaisante car elle non plus ne tient pas compte de la structure interne de la séquence. Par exemple, pour la séquence $1010101010101010x$, la méthode de von Mises nous donne à probabilité égale 1 ou 0 pour le prochain bit x, mais il est beaucoup plus tentant de prédire que x vaudra 1 !

La théorie de la complexité algorithmique va tenter de définir « aléatoire » d'une autre façon. Plutôt que de calculer la complexité basée sur des probabilités, elle va calculer la complexité intrinsèque d'une séquence par la longueur du plus petit programme qui, une fois lancé, produit la séquence en sortie. Le cadre théorique s'appuie sur la théorie de la calculabilité, telle que décrite par Turing et Church. Allan Mathison Turing (1912–1954) proposa une théorie de la calculabilité d'une élégante simplicité et sa machine théorique, que par après on a nommé la

1 Le même von Mises qui proposa le paradoxe des anniversaires.

machine de Turing en son honneur, est une description idéalisée d'un ordinateur général [210]. La machine de Turing est cependant beaucoup plus simple qu'un ordinateur conventionnel. Elle utilise un ruban (qui peut être arbitrairement long, voire même infini) pour mémoire, qu'elle peut dérouler à gauche ou à droite, y lire ou y écrire un symbole. Le nombre d'états distincts que le contrôle de la machine peut prendre est fini. La fig. 3.1 montre une machine de Turing, telle qu'on pourrait se l'imaginer.

Turing spécule que tout problème qui est effectivement calculable (c'est-à-dire pour lequel il existe un algorithme qui calcule la solution), l'est sur une machine de Turing. Alonzo Church (1903–1995) chercha à démontrer que toute fonction qui est effectivement calculable, l'est par une classe de fonctions que l'on nomme partielles récursives [49]. Puisque toute fonction partielle récursive peut être calculée par un programme exécuté par un ordinateur général, lequel se réduit à une machine de Turing, Church émit l'hypothèse (qui reste d'ailleurs toujours à démontrer) que la machine de Turing, en étant capable de calculer les fonctions partielles récursives, était suffisamment puissante pour mener à terme toute fonction calculable.

L'idée de base de la théorie de la complexité algorithmique des séquences est assez simple : on mesure la complexité intrinsèque d'une séquence par la longueur du plus petit programme (fonction calculable sur une machine de Turing) qui produit cette séquence en sortie. Si le programme est plus court que la séquence originale, on parlera d'une séquence « facile » alors que si le plus court programme est en fait aussi long que la séquence elle-même, elle est « difficile », c'est-à-dire aléatoire au sens de Kolmorogov-Solomonoff-Chaitin.

Considérons la classe de séquences suivante :

$$\underbrace{11111\ldots1111}_{n \text{ fois}}$$

Elle est clairement très peu aléatoire ; nous n'avons qu'une répétition de uns. Le seul élément aléatoire dans cette classe de séquences, c'est la longueur, n, d'une séquence en particulier. Un programme (très simple) qui génère les séquences de cette classe à partir d'un nombre n est donné par :

```
séquence(n)
 {
  for (i=0;i<n;i++) output 1
 }
```

Ce programme prend environ une trentaine de symboles pour être décrit et prend un paramètre n en entrée. Puisque, comme nous le verrons à la section 5.3.3, il suffit d'$O(\lg n)$ bits pour représenter un entier arbitraire n, la complexité de cette classe de séquence est donc $O(\lg n + c)$, où c est la longueur du programme en bits. Dans ce cas très simple, on voit difficilement comment on pourrait faire nettement mieux pour générer cette classe de séquence. Par contre, la séquence suivante, pourtant issue d'un générateur de nombres pseudoaléatoires, donc d'un algorithme déterministe,

$$00100101110010110100010000111101110011101$$

paraît nettement moins facile à réduire.

On cherchera donc un programme optimalement court pour produire une séquence x. Supposons que programme soit donné par la fonction p qui est calculable au sens de Turing. À ce programme correspond une machine de Turing ϕ spécialement conçue et qui contient des instructions adéquates pour une implémentation efficace (en terme de longueur du programme) de p. Dans notre exemple, notre machine ϕ reconnaît des programmes qui sont écrits en pseudo-C. Caractérisons-donc formellement la complexité de Kolmogorov.

Définition 3.2.1 Complexité de Kolmogorov. Soient ϕ, une machine de Turing et p, un programme à exécuter sur la machine ϕ. La complexité de Kolmogorov d'une séquence x est

$$K_\phi(x) = \min\{\, |p| \;\big|\; \phi(p) = x \,\}$$

c'est-à-dire la longueur du plus petit programme p exécuté sur la machine ϕ qui produit la séquence x en sortie. Le programme p n'utilise aucune information extérieure pour produire x. Soit maintenant y, une séquence. Alors, la complexité conditionnelle de Kolmogorov de la séquence x selon y est donnée par

$$K_\phi(x \mid y) = \min\{\, |p| \;\big|\; \phi(p, y) = x \,\}$$

qui est la longueur du plus petit programme s'aidant de l'information y pour générer la séquence x. On peut penser à p et à y comme étant respectivement le programme et ses données, au sens où on l'entend quand on considère un programme conventionnel sur un ordinateur général. La complexité inconditionnelle et la complexité conditionnelle sont liées par l'inégalité

$$K_\phi(x \mid y) \leqslant K_\phi(x),$$

ce qui revient à dire qu'il n'est jamais nuisible d'avoir de l'information en plus. □

La complexité de Kolmogorov de la séquence x selon la machine de Turing ϕ est notée $K_\phi(x)$. La fonction $K_\phi(x)$ retourne la taille du plus court programme p pour la machine ϕ qui calcule la séquence x. La complexité d'une séquence x est alors affectée par les qualités intrinsèques de la machine ϕ. Supposons que la machine ϕ soit une machine qui reconnaisse un langage très général. La complexité de x dépend alors du plus petit programme, écrit dans un langage très général, qui n'a aucun avantage particulier quant qu'à la séquence x, qui produit la séquence x. Imaginons qu'au contraire nous ayons une machine ϕ' qui accepte un langage très spécialisé pour la classe de séquences dont x est tirée. Ce langage offre alors des instructions adaptées à la classe de séquences qui nous intéresse. Alors le programme qui produit x sur la machine ϕ' pourrait être significativement plus court que le programme pour la machine non-spécialisée ϕ.

Pour « immuniser » la mesure de complexité de tels effets, nous pouvons décomposer la complexité en deux partie. La première partie demeure la longueur du plus petit programme qui génère la séquence x sur ϕ' et la seconde partie comptera la complexité de la machine elle-même. Si la machine ϕ' est fort complexe, comme le laisserait entendre sa spécialisation, cette complexité doit entrer en compte dans la complexité algorithmique de x. Si on rapporte la complexité d'une séquence x sur une machine ϕ' contre une machine de Turing universelle u (qui a le pouvoir d'émuler tout programme), on aura

$$K_\phi(x) \leqslant K_{\phi'}(x) + K_u(\phi'),$$

c'est-à-dire que la complexité de Kolmogorov de x par rapport à une machine générale ϕ est inférieure ou égale à la complexité d'un autre programme exécuté sur une machine spécialisée ϕ' plus la complexité de son émulateur sur la machine universelle u. Donc, si on transporte le programme qui produit x de la machine optimale ϕ' vers une machine universelle u, la complexité de Kolmogorov croît d'un facteur proportionnel à la taille du programme qui émule la machine ϕ' sur la machine universelle u, et cette complexité est plus grande ou égale à la complexité obtenue en utilisant une machine générale quelconque ϕ.

Théorème 3.2.1 Complexité sous la concaténation. Puisque la concaténation ne préserve pas la séparation entre les séquences originales, nous avons que $K_\phi(\ (\ a : b\ )\ ) \leqslant K_\phi(a) + K_\phi(b)$. □

La concaténation de préserve pas la séparation entre les séquences. Nous pouvons trouver le plus court programme pour (a : b). Au pire, ce plus court programme consistera en la concaténation des programmes pour a et pour b. Ce programme sera conçu de façon à ce que lorsque le programme pour a termine, le programme pour b démarre. Cela suppose que le point de terminaison du programme pour a est à la fin du programme de façon à ce que le flot d'instruction passe naturellement de la dernière instruction du programme pour a à la première instruction du programme pour b.

Théorème 3.2.2 Complexité sous le pairage. La fonction de pairage conserve l'information sur la séparation entre les séquences originales. La complexité de Kolmogorov de deux séquences pairées est $K_\phi(\ \langle\ a, b\ \rangle\ ) \leqslant K_\phi(a) + K_\phi(b) + O(\lg \min(K_\phi(a), K_\phi(b)))$. Au mieux, on trouve un programme plus court qui génère successivement les séquences a et b tout en sachant où placer un séparateur de séquence. Au pire, le plus court programme est composé du programme pour a, du programme pour b plus un certain nombre de bits qui donne la longueur du programme pour a, de façon à ce que l'on puisse déduire la limite entre a et b. Ici aussi on suppose que le point de terminaison du programme pour a est à la fin de celui-ci, de façon à ce que le flot d'instruction passe naturellement à la première instruction du programme pour b.

Définition 3.2.2 c-incompressibilité. Une séquence est dite c-incompressible sous ϕ si $K_\phi(x) \geqslant |x| - c$. □

Une séquence x est c-incompressible si la longueur de son programme n'est pas plus court que $|x| - c$; autrement dit, on n'obtient aucune compression significative en utilisant le programme p (qui minimise $K_\phi(x)$) plutôt que la séquence x elle-même. On peut déduire, parmi une classe de séquences, combien de séquences de longueur l seront 0-incompressibles, 1-incompressibles, etc. Par exemple, il est clair que nous avons 2^l séquences (composées de bits) de longueur l. Parmi celles-ci, un certain nombre pourront être compressées significativement, mais la grande majorité d'entre elles seront essentiellement incompressibles. En fait, pour une constante $c < l$, on aura qu'*au moins* $2^l - 2^{l-c} + 1$ séquences seront c-incompressibles [127, p. 109]. On obtient le résultat de la façon suivante. Il y aura au moins une séquence de longueur l qui sera 0-incompressible. La moitié au moins sera 1-imcompressible car même si on partage les 2^l séquences en deux ensembles, il faudra un bit pour identifier l'ensemble auquel appartient une séquence. Au moins le trois quart des séquences sera 2-incompressible (soit le quart qui sera 2-compressible). Enfin, le $(1 - 2^{-c})^{\text{ième}}$ des séquences sera c-incompressible.

On peut arriver à cette conclusion différemment, entre autres par l'argument de dénombrement (en anglais, *counting argument*). Si on considère les programmes comme étant simplement

des séquences de bits, nous avons *au plus* 2^m programmes de longueur m (car toutes les séquences de m ne sont pas nécessairement des programmes valides pour ϕ). Parmi toutes les séquences de longueur l, seulement au plus 2^m d'entre elles receveront un programme de longueur m. Puis, au plus 2^{m+1} d'entre elles un programme de longueur $m+1$, etc. Nous avons donc que le $(1-2^{-m})^{\text{ième}}$ des séquences auront des programmes plus longs que m bits, c'est-à-dire c-incompressibles avec $c = l - m$.

L'argument de dénombrement sert à démontrer qu'aucune méthode de compression ou de représentation de séquences ne peut compresser *toutes* les 2^l séquences de longueur l. Si on considère que les séquences de l bits de long sont en fait des nombres écrits en base 2, on sait qu'il y a 2^l nombres distincts possibles. Si on compresse ces nombres, on leur assigne des séquences de bits plus courtes, disons de longueur m, avec $m < l$. Ces séquences de longueur m représentent en fait des nombres, mais seulement 2^m nombres distincts, et $2^m \ll 2^l$. Comme il y a moins de petits nombres que de grands nombres, tous les grands nombres ne peuvent pas recevoir un petit nombre comme représentation compressée, ce qui fait que tous les grands nombres ne sont pas compressibles. Si beaucoup de grands nombres reçoivent de petits nombres, de petit nombres devront recevoir de grands nombres pour version compressée.

Ce que l'argument de dénombrement nous dit, c'est que seule une partie infime de toutes les séquences possibles, d'une longueur donnée, sont compressibles significativement, ou encore que la majorité de ces séquences est essentiellement aléatoire, au sens de Kolmogorov-Solomonoff-Chaitin. Ce résultat est très important car il stipule, indirectement, que seules les séquences suffisamment structurées (qui sont rares par rapport aux séquences aléatoires) sont compressibles.

❁ ❁ ❁

Maintenant que nous connaissons quelques-unes des propriétés de la complexité de Kolmogorov et quelques définitions, il nous reste à aborder la question fondamentale : *comment calcule-t-on* $K_\phi(x)$? La réponse est, on s'en doutait bien, qu'on ne calcule pas $K_\phi(x)$! On peut montrer en effet que $K_\phi(x)$ n'est pas partielle récursive, donc pas calculable [127]. La preuve (dont nous n'exposerons pas les détails dans cette introduction) repose sur le fait que pour trouver p, le programme le plus court, il faut énumérer les programmes valides pour ϕ (c'est-à-dire ceux qui produisent un output) ce qui revient au problème d'arrêt, car il faut décider si chaque programme testé termine normalement avant de comparer sa sortie à x.

Comment nous sortir de ce bourbier ? Chaitin propose de restreindre la classe de fonctions à examiner [43]. Cette classe de fonctions est telle qu'il est toujours possible de tester la validité syntaxique d'un programme et d'être certain que le programme termine après un nombre fini, proportionnel à sa longueur, d'étapes. La syntaxe utilisée par Chaitin est celle d'un dialecte épuré du langage LISP. Cela permet d'énumérer tous les programmes valides de longueur n (car les règles syntaxiques sont très simples) et d'être assuré que les programmes terminent toujours. Cependant, même lorsqu'on se restreint de cette façon, la tâche de trouver un programme qui génère une séquence (ou une classe de séquences) donnée est possible mais computationnellement trop ardue pour qu'il soit réaliste d'utiliser cette technique pour estimer la complexité algorithmique. En effet, il existe un nombre exponentiel de programmes de longueur n : si s est le nombre de symboles que reconnaît le langage, alors le nombre de programmes valides est

nettement inférieur à s^n, mais encore démesuré.

Bien que la théorie de la complexité algorithmique fournisse des outils qui permettent d'articuler une pensée autour d'une mesure de la complexité intrinsèque des séquences, elle n'est pas pratique car, en général, cette mesure est non calculable. Même pour des classes restreintes de fonctions génératrices, elle n'est guère pratique ; on doit explorer un espace de fonctions immensément vaste. Malgré que la théorie de la complexité algorithmique de Kolmogorov mette à notre disposition tout l'arsenal de l'informatique théorique (théorie des automates, des langages, de la hiérarchie arithmétique, algorithmique, etc.) l'incroyable difficulté d'obtenir des estimations décentes rend l'utilisation de la complexité de Kolmogorov pour le moins rébarbative, sinon rédhibitoire !

3.3 Complexité stochastique et théorie de l'information

La théorie de la complexité algorithmique estime le contenu d'information d'une séquence en estimant la longueur du plus court programme qui produit cette séquence en sortie. Ce qui peut sembler étonnant *a priori*, c'est que c'est une mesure de complexité dont on a expurgé tout élément probabiliste. Pourtant, jusqu'au moment où fut énoncé la théorie de complexité algorithmique des séquences, la définition de séquence aléatoire était intimement liée à la notion du hasard au sens probabiliste ou statistique. On n'a qu'à penser à la définition d'une séquence aléatoire selon von Mises (voir section 3.2, page 27). Von Mises, comme tant d'autres, a tenté de maintenir la cohérence entre la notion intuitive de séquence aléatoire et ce qui était jusqu'alors connu des processus stochastiques.

Une mesure de complexité instrinsèque des séquences basée sur des mesures probabilistes est plus intuitive. Elle permet aussi de traiter naturellement des séquences dont on ne peut observer qu'une partie, comme c'est le cas lors d'une transmission. La séquence ne nous est pas donnée dans son ensemble, d'un coup ; les symboles qui la composent entrent un par un par le canal de communication, et on ne peut pas toujours attendre d'avoir toute la séquence pour commencer à la compresser — lorsque c'est même possible d'avoir toute la séquence. Une théorie de la complexité stochastique serait alors basée sur une mesure qui rend compte de la difficulté de prédire le i^{e} symbole dans une séquence lorsqu'on connaît les $i-1$ premiers.

On pourrait être tenté d'utiliser une définition un peu naïve, ressemblant à la définition d'aléatoire au sens de von Mises, pour estimer la complexité d'une classe de séquence. On se souvient que la définition de von Mises ne fait intervenir qu'un paramètre p qui estime la probabilité que le prochain bit dans la séquence soit un 1 (donc $(1-p)$ estime la probabilité qu'il s'agisse d'un 0). Ce paramètre p est estimé par la proportion de uns dans la partie de la séquence déjà observée. Cette définition est trop simple car elle ne tient aucun compte des structures internes des séquences. En fait, ce que l'on veut vraiment c'est, pour une variable aléatoire X_t de distribution inconnue qui génère le t^{e} bit de la séquence, estimer

$$P(X_t = 1 \mid x_1^{t-1})$$

Cette formulation permet de tenir compte implicitement de toutes les structures internes à séquence, puisque le prochain bit est prédit conditionnellement à la séquence x_1^{t-1} ; nous laissant toute liberté quant à la façon de réaliser notre prédiction. Cette estimation ne donne pas encore la quantité d'information contenue dans la séquence x_1^t, mais elle sera un élément essentiel

de la mesure d'information stochastique. En général, on ne peut pas calculer de façon fiable $P(X_t = 1 \mid x_1^{t-1})$. Pour estimer $P(X_t = 1 \mid x_1^{t-1})$ avec quelque confiance, il faut un nombre très grand de séquences observées. Lorsque t est grand, le nombre de séquences distinctes x_1^t que l'on pourra observer sera infinitésimal par rapport au nombre de séquences possibles — car il existe un nombre exponentiel de séquences distinctes de longueur t — et une estimation de $P(X_t = 1 \mid x_1^{t-1})$ devient risquée. Pour donner une idée de la tâche, on peut contempler le fait que la bibliothèque de l'université de Montréal compte environ trois millions d'ouvrages, d'une longueur moyenne de peut-être cinquante mille mots, soit, à six lettres par mot, trois cent mille caractères. Cela nous donne trois millions d'exemples de séquences parmi les $60^{3\times 10^5}$ séquences de même longueur possibles[2]. Aussi bien dire que la proportion observée est zéro !

Plutôt que chercher à modéliser le prochain symbole en considérant toute la séquence, on le fera en utilisant un horizon. On peut raisonnablement penser que les mots au début du texte ont une influence statistique faible sur les mots qui se trouvent à la fin du même texte. On peut même penser que l'influence des mots qui terminent un paragraphe dépendent assez peu des mots qui l'ouvrent. Cela nous permet de restreindre notre horizon à quelques phrases, voire même quelques mots, sans changer significativement l'estimation des probabilités. Puisque le nombre de mots valides est relativement faible (tout au plus quelques centaines de milliers de mots[3]), on peut même spéculer que les structures intra-mots suffisent pour prédire les lettres qui suivent immédiatement. De plus, George Kinsley Zipf (1902–1950) montre que seulement 100 mots comptent pour environ 50% des mots usuels utilisés en anglais [232, 233, 234]. Nous trouvons les mêmes résultats empiriques chez d'autres auteurs [7, 167, 208]. Cela laisse donc entendre que l'horizon de prédiction dans le cas particulier du texte peut être assez limité.

Le modèle prédictif peut donc être changé pour ne plus tenir compte que d'un horizon limité sans endommager sérieusement la qualité de la prédiction. Plutôt que d'utiliser $P(X_t = 1 \mid x_1^{t-1})$, on utilisera une version modifiée pour ne tenir compte que des dépendances sur une fenêtre de longueur ω. En limitant l'horizon à une longueur ω, nous obtenons

$$P(X_t = x_t) \approx P(X_t = x_t \mid x_{\max(1,t-\omega)}^{t-1})$$

Le paramètre ω dépend de la classe de séquences qui nous intéresse. Pour du texte, il semblerait que $\omega \approx 6$ permette des estimations raisonnables. Dans certain modèles, on utilise une forme plus sophistiquée où on remplace ω par $\omega(t)$, ce qui permet de choisir adaptativement la longueur de fenêtre qui est considérée en fonction, par exemple, du nombre d'observations de $x_{t-\omega}^t$.

On voit que les critiques formulées par les « kolmogorovistes », à savoir que les probabilités seules ne sont pas capable de tenir compte des phénomènes intrinsèques à une séquence, ne sont pas totalement fondées. S'il est vrai que certains types de structures ne sont pas modélisables à l'intérieur du paradigme des probabilités classiques seules[4], les probabilités aidées de mécanismes secondaires qui vont extraire de l'information sur les structures sont tout à fait adéquates. Il s'agit de cacher le modèle idoine à la classe de séquences considérées dans l'évaluation de $P(X_t = 1 \mid x_1^{t-1})$.

2 En supposant que nous ayons 60 symboles typographiques distincts.
3 Le Petit Robert en compte environ 60 000.
4 Par exemple, les séquences palindromiques.

Dans les premières étapes de l'évolution de la théorie de l'information, on n'établit pas encore la connexion avec la théorie de la probabilité. Dans un des premiers articles sur le sujet, on ne trouve aucune considération stochastique. Au contraire, R. V. L. Hartley propose une mesure de l'information qui ne dépend que du nombre de symboles dans l'alphabet et de la longueur du message à transmettre [90]. Il décompose le problème de transmission en deux niveaux distinct. Le premier niveau se situe au niveau du canal de communication lui-même, où on retrouve un alphabet qui ne dépend que des caractéristiques physiques du canal. Si le médium de communication permet trois niveaux de voltage, disons $\{-1, 0, +1\}$, l'alphabet primaire contiendra trois symboles. Au second niveau, on retrouve l'alphabet secondaire, dans lequel se forment les messages. À chaque symbole secondaire correspond une séquence de symboles primaires, et c'est grâce à cette traduction que l'on réussira à transmettre tout message par le canal limité, quoique possiblement au prix d'une transmission plus longue.

Hartley vise à définir la quantité d'information contenue dans un message. Il décrit ainsi la propriété principale d'une telle mesure [90, p. 540] :

> *To do this we arbitrarily put the amount of information proportional to the number of selections* [choix de symboles] *and so choose the factor of proportionality as to make equal amounts of information correspond to equal numbers of possible sequences.*

Ce qui l'amène à la définition d'une mesure information basée sur le nombre de messages distincts pour une même quantité d'information. Il raisonne ainsi. Soit, A_p, l'alphabet primaire. Posons, sans perte de généralité, $A_p = \mathbb{B} = \{0, 1\}$ et $n = |A_p| = 2$. Soit A_s, l'alphabet secondaire, qui contient un nombre de symboles qui est une puissance de n, soit $|A_s| = |A_p|^m$. Supposons que nous ayons un message composé de symboles secondaires de longueur l à transmettre. Le message contient alors $m \times l$ symboles primaires. La quantité d'information est alors donnée par

$$H = l \times m = l \log_n |A_p|^m = l \log_n |A_s|$$

Ce que Hartley démontre ainsi, c'est que l'information ne dépend pas de l'alphabet primaire car grâce au $\log_n$, son influence ne devient qu'une constante multiplicative. La quantité d'information ne dépend que de la longueur du message et du nombre de symboles dans l'alphabet secondaire. En ramenant arbitrairement $n = 2$ (en utilisant lg plutôt que log ou ln) on obtient une quantité d'information en *bits* [5] [6].

Claude Elwood Shannon (1916–2001) ira plus loin [188, 189]. Bien qu'il soit d'accord avec Hartley sur le choix de la fonction lg (car celle-ci offre de nombreux avantages mathématiques, dont celui de rendre linéaire les relations exponentielles), il juge que la mesure de Hartley est insuffisante. Son objection principale tient au fait que la mesure de Hartley ne tient aucun compte des probablités d'occurence des symboles. Shannon a l'intuition que les symboles fréquents portent moins d'information en eux que les symboles rares. À partir de cette prémisse, il énumérera une liste de contraintes qu'une mesure d'information pratique doit satisfaire :

---

5 Il faudra attendre l'article de Shannon pour voir le mot bit entrer dans la littérature.

6 Poser $n = 2$ semble naturel pour nous qui utilisons des ordinateurs binaires. Cependant, pour la transmission d'information par un moyen électromagnétique, il est presque toujours question d'au moins trois symboles, soit $\{-V, 0, +V\}$. Deux symboles sont dédiés à la transmission des bits, le troisième étant réservé à des fins de contrôle, par exemple délimiter les mots et signaler l'absence de transmission.

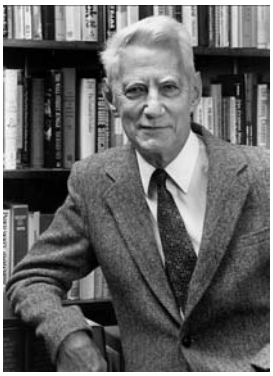


FIG. 3.2 – Claude Elwood Shannon (1916 – 2001). Photo © Lucent Technologies.

- La mesure ne doit dépendre que du nombre de symboles de l'alphabet de message, non des symboles eux-mêmes.

- La mesure doit être définie en fonction des probabilités d'occurence des symboles.

- La mesure doit être continue : un petit changement dans les probabilités mène à un petit changement dans la mesure d'information.

Le premier point stipule que la valeur sémantique des symboles ne joue aucun rôle dans la mesure d'information. On peut remplacer n'importe quel alphabet par un dictionnaire de séquences binaires (voir théorème 3.1.1, p. 27), ou encore de nombres naturels. Le second point précise que les probabilités d'occurence jouent un rôle fondamental dans la quantité d'information. On comprend intuitivement pourquoi un message fréquent et répétitif contient peu d'information. En fait, après la première annonce, les répétitions n'apportent aucune autre information hormis la répétition elle-même. Si les messages sont variés et surprenants, alors ils contiennent beaucoup plus d'information. Enfin, une mesure continue permet de faire que de petit changements dans les probabilités ne produisent que de petits changements dans la quantité d'information. À prime abord, on pourrait penser qu'il ne s'agit que d'une exigence ayant trait à des considérations analytiques, mais cela aussi correspond à une caractéristique intuitive de l'information. Si les probabilités changent légèrement, on ne s'attend pas à voir la quantité d'information changer significativement.

On s'attend donc à trouver une mesure du contenu d'information qui soit à la fois intuitive (jusqu'à un certain point) et mathématiquement pertinente. La fonction d'information H devrait aussi être compatible avec les notions probabilistes classiques. Pour un ensemble de probabilités $\{p_1, p_2, \ldots, p_n\}$ tel que $\sum_i p_i = 1$, la mesure d'information devra respecter les contraintes suivantes :

1. Elle est continue en p_i.

2. Si tous les p_i sont égaux, soit $p_i = \frac{1}{n}$, alors elle est monotone croissante en n.

3. Si on peut grouper les choix, alors elle est la somme pondérée des mesures pour les groupes de choix, plus l'information propre au choix des groupes.

Shannon s'attaque au problème de trouver une fonction H qui satisfasse ces exigences. Shannon démontre le théorème suivant :

Théorème 3.3.1 Fonction d'entropie. Toute fonction qui satisfasse les trois énoncés précédents est forcément de la forme :

$$\mathcal{H}(X) = -K \sum_x P(X = x) \log_b P(X = x) \tag{3.1}$$

où on peut poser $K = \ln b$ pour ramener le logarithme à la base naturelle, ou poser $K = 1$ pour calculer l'information en termes d'une autre base. Habituellement, on pose $b = 2$, donc on utilise lg, ce qui donne la mesure en bits. La démonstration du théorème peut être trouvée à l'appendice B. □

La contribution de Shannon n'aura pas été seulement de résoudre le problème de la mesure d'information en énonçant la formule pour $\mathcal{H}(X)$. Il proposa aussi un schéma général pour les communications (voir fig. 3.3) qui demeure essentiellemenet inchangé. Shannon décrit les six composantes d'un système de communication : l'expéditeur, l'émetteur, le canal de transmission, le receveur, le destinataire, et, optionnellement, une source de bruit.

L'expéditeur (*information source*) est, à toutes fins pratiques, une source aléatoire qui émet des symboles tirés d'un alphabet de message. Il n'y a aucune contrainte quant à la nature de la variable aléatoire contenue dans l'expéditeur. L'émetteur (*transmitter*) prend les symboles de l'expéditeur et les transforme en symboles compatibles avec le canal de communication (*communication channel*). L'émetteur peut encoder l'information de façon à ce qu'elle soit tolérante aux erreurs du canal, ou encore la compresser. Le canal de communication transporte physiquement l'information de l'émetteur vers le receveur (*receiver*). Si il est idéal, le canal transporte l'information sans erreur, sinon il peut être influencé par une source de bruit et le récepteur recevra des symboles erronés qu'il lui faudra récupérer, s'il en est capable (à supposer qu'il soit même capable de détecter qu'il reçoit des symboles corrompus). Le receveur transforme les symboles reçus du canal de communication en symboles de l'alphabet de message pour le destinataire (*information destination*). Si l'information a voyagé compressée ou encodée pour être tolérante aux erreurs, c'est au receveur qu'incombe la responsabilité de la présenter décompressée et décodée au destinataire. Le destinataire est simplement le mécanisme (ou l'individu) qui sait quoi faire avec l'information reçue.

Ce modèle demeure pertinent dans le cadre de la compression de données. L'émetteur et le receveur sont remplacés par le compresseur et le décompresseur, respectivement. Le canal demeure présent, soit sous la forme d'un canal de transmission, soit sous la forme d'un médium de stockage, qu'il soit temporaire ou permanent. Quant à la source, la sémantique demeure la même. On considérera que toute source de messages est en fait une variable aléatoire (sans faire d'hypothèses restrictives quant à la nature de la variable aléatoire) qui émet des symboles que le compresseur devra gérer de façon à produire une représentation efficace en terme du nombre de symbole du canal qu'il lui faudra utiliser pour encoder le message. On ne peut pas assumer qu'en général, le compresseur ait accès aux paramètres de la source aléatoire. Il lui faudra compter sur

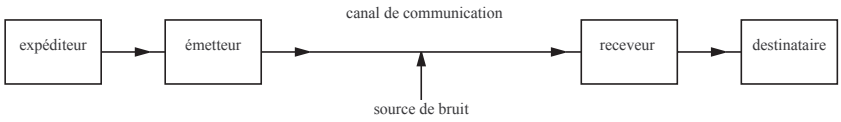


FIG. 3.3 – Le problème de la communication, tel qu'énoncé par Shannon.

des estimateurs obtenus par échantillonnage.

❁ ❁ ❁

La mesure d'entropie $\mathcal{H}(X)$ telle qu'énoncée par Shannon est très générale. En effet, elle est simple et permet des formulations analytiques pour des distributions connues, en particulier celles de la famille exponentielle, comme les variables aléatoires géométriques, de Poisson, etc. Puisqu'elle est basée sur une fonction de probabilité, elle est naturellement compatible avec notre notion intuitive de probabilité, et se soumet aux opérations que l'on peut effectuer sur des probabilités, telle que la factorisation en probabilités conditionnelles. De plus, les différentes connotations avec la notion physique de chaos (l'entropie de Bolztmann, en particulier, voir les notes bibliographiques à la fin de ce chapitre) peuvent mettre à notre disposition des techniques mathématiques jusqu'alors réservées aux applications de physique et mécanique statistiques.

3.4 Mesures de complexité dans la littérature

Bien que la théorie de la complexité algorithmique des séquences et la théorie de la complexité stochastique — que l'on nomme tout simplement *information theory*, soit « théorie de l'information » — soient toutes deux présentes dans la littérature des méthodes de compression, la complexité stochastique occupe la place de choix. Cela peut s'expliquer par plusieurs facteurs.

Une explication pragmatique se trouve dans le fait que l'on peut obtenir une bonne approximation de l'entropie en utilisant certaines hypothèses simplificatrices sur la distribution alors que la complexité de Kolmogorov repose sur un problème indécidable. La complexité de Kolmogorov demeure computationnellement impossible à utiliser même lorsqu'on considère des réductions dans la classe de fonctions admissibles ; ces réductions demandent encore que l'on examine un nombre exponentiel de programmes. Il est à noter que la détermination du meilleur modèle statistique pour une séquence repose aussi sur un problème indécidable. Toutefois, en utilisant des hypothèses simplificatrices sur la nature de la distribution, on peut obtenir des modèles statisfaisants menant à de bonnes approximations de l'entropie. De plus, pour les distributions paramétriques, il est souvent aisé d'obtenir des formes analytiques pour l'entropie.

Le fait que la mesure d'information proprosée par Shannon soit apparue beaucoup plus tôt (les premières idées apparaissent dès 1928) que la mesure proprosée par Kolmogorov suffirait pratiquement à expliquer pourquoi la première est beaucoup plus répendue dans la littérature scientifique. Un autre facteur important est que Hartley, Shannon, et les autres qui participèrent aux premiers développements de la théorie de l'information stochastique gravitèrent autour de

centres de recherches prestigieux jouissant d'un rayonnement panaméricain et les développements qu'ils proposaient s'inscrivaient plus naturellement dans la lignée des travaux déjà accomplis en théorie classique des signaux qui s'appuient fortement sur la physique statistique et la mécanique ondulatoire. Les publications sur la complexité de Kolmorogov furent comparativement tardives (dans les années 1960) et même si certaines parurent dans des journaux prestigieux, tels que les *IEEE Transactions on Information Theory*, leur impact demeura modeste car déjà la littérature de la théorie de l'information constituait un imposant corpus.

Dans la littérature concernant les télécommunications en général (codage, discrétisation de vecteurs, compression, correction d'erreur, etc.) c'est la théorie de l'information de Shannon qui domine. Comme cette thèse s'inscrit dans la ligne de pensée de la littérature des télécommunications, nous adopterons la théorie de l'information de Shannon et la fonction d'entropie pour mesurer la complexité des sources aléatoires et des séquences.

3.5 Notes bibliographiques

Le livre de Ming Li et Paul Vitányi, *An Introduction to Kolmogorov Complexity and its Applications* [127] constitue probablement le meilleur texte d'introduction à la théorie de la complexité algorithmique. Le livre de Chaitin, *Algorithmic Information Theory* [43] exploite le lien entre les séquences autodélimitantes (en particulier les programmes écrits en une version simplifiée du langage LISP) et les équations diophantines afin de démontrer une version forte du théorème d'incomplétude de Gödel [80], c'est-à-dire qu'en général il n'est pas possible de déterminer si un programme existe pour une tâche donnée. Les équations diophantines sont liées à ce problème en ce qu'il n'est pas toujours possible de démontrer qu'une équation diophantine n'a aucune solution en nombres entiers. Le lecteur intéressé aux méandres de la calculabilité peut se référer au livre de John E. Hopcroft et Jeffrey D. Ullman, *Introdution to automata theory, languages, and computation* [99], qui couvre en grand détail les modèles de calculabilité et les hiérarchies de fonctions. De même, *Theory of Computation*, de Derick Wood [225] est une lecture intéressante sur le même sujet.

Une bonne sélection d'articles et de rapports techniques sur les premiers développements de la théorie de l'information peut être trouvée dans *Key Papers in the Development of Information Theory* [192]. Cette anthologie est organisée en ordre chronologique, en commençant par les deux articles fondateurs de Shannon [188, 189]. On y trouvera aussi d'autres articles, comme l'article de Huffman sur sa méthode de construction de codes minimalement redondants [101], à laquelle nous consacrons d'ailleurs le chapitre 6. D'autres bons textes d'introduction à la théorie de l'information et du codage sont *Information Theory*, de Robert Ash [6] et *Coding and Information Theory* de Steven Roman [177].

Le choix de $\mathcal{H}$ pour le symbole de la fonction d'entropie n'est pas fortuit et, contrairement à ce qu'il peut paraître, n'a pas à voir avec les initiales de Hartley. En fait, il y a un lien fort entre la fonction H, qui est une mesure de diversité, et l'entropie au sens de la mécanique statistique. En particulier, on retrouve la fonction H de Boltzmann, dont les solutions prennent la forme

$$H(x) = -c \int f(x) \ln f(x) \, \partial x$$

où $f(x)$ est une fonction de probabilité définie sur x (qui peut être un scalaire, un vecteur ou un tenseur) [204].

Chapitre 4

Les paradigmes de la compression

Ce chapitre présente les grands paradigmes de la compression. Sans être exhaustifs, nous présenterons les principales méthodes de compression avec et sans perte, ainsi que des techniques de décomposition de signal et de discrétisation. Nous introduirons des techniques de décomposition de signal en fréquences, comme la transformée de Fourier, la transformée de Hartley et les transformées à base d'ondelettes. Nous présenterons aussi sommairement quelques techniques de discrétisation de scalaires et de vecteurs. Nous examinerons quelques techniques de compression basées sur les dictionnaires, comme les algorithmes des classes LZ77 et LZ78. Les algorithmes par transformations, comme l'algorithme Burrows-Wheeler et les algorithmes à base de prédicteurs seront décrits brièvement. Nous terminerons ce chapitre en présentant quelques algorithmes de modélisation statistiques et en discourant de l'importance du codage des entiers dans les algorithmes de compression.

4.1 Compression avec perte

La compression avec perte est habituellement utilisée avec les données numériques qui représentent des signaux analogiques, comme le son, les données séismiques, la vidéo, etc. Plutôt que de chercher à coder ces signaux de façon exacte, nous tenterons d'en éliminer le bruit et ne coderons que ce qui est important dans ces signaux. La détermination des composantes du signal qui correspondent au bruit dépend naturellement de la nature du signal considéré et c'est en général un problème qui n'est pas très bien résolu. On a recours à des métriques de type moindres carrés qui mènent à des solutions analytiques plus simples mais qui ne correspondent pas nécessairement à l'impact perceptuel des données détruites.

Les méthodes de compression du son et de la vidéo reposent sur des théories psychoacoustiques et psychovisuelles. La compression du son utilise le masquage temporel ou en fréquence, l'élimination de co-fréquences, la réponse de l'oreille moyenne à une fréquence étant donné le volume, et autres astuces du même genre pour déterminer quelle partie du son peut être éliminée sans en affecter vraiment la qualité. Il en va de même pour la compression des images ou de la vidéo. On cherche à éliminer les composantes de l'image qui sont associées au bruit sans affecter la qualité visuelle de l'image reconstruite. Par exemple, l'œil est beaucoup plus sensible à la différence de luminosité qu'à la différence de la teinte ou de la pureté, ou saturation, de la couleur, et on peut exploiter cette information en transformant les couleurs de l'image de l'espace de couleur RGB vers un autre espace de couleur où l'information de couleur est décomposée en

luminance (luminosité perçue), saturation (pureté de la couleur) et chrominance (la teinte). Ces composantes seront alors sous-échantillonnées et codées selon leurs importances relatives.

La compression avec perte de ces données commence par une décomposition du signal qui met en évidence ses différentes composantes. Ces composantes sont alors éliminées ou conservées, ou encore codées avec une précision moindre. La précision avec laquelle les données sont codées dépendra de l'importance que l'on voudra leur accorder, et de la qualité de reconstruction que l'on désire.

Les décompositions de signaux viennent essentiellement en deux variantes. La première est la famille des transformées dites trigonométriques, dans laquelle on trouve la transformée de Fourier, la transformée de Hartley et la transformée de cosinus. Ces transformées existent en versions continues et discrètes. Elles sont caractérisées par les fonctions de base qui sont utilisées. Les fonctions de base des transformées trigonométriques sont périodiques et s'étendent sur toute la longueur du signal. Nous verrons quelles autres propriétés de ces transformations du signal sont exploitables pour la compression de données. La seconde famille est constituée par les bases d'ondelettes qui, contrairement aux transformées trigonométriques, utilisent des bases qui ont la propriété du support compact, c'est-à-dire que les « atomes » de fréquences ne s'étendent pas sur toute la longueur du signal mais seulement sur une région finie et finement localisée. La section 4.1.1 introduit les principales méthodes de décomposition de signaux.

La précision avec laquelle les données sont encodées est contrôlée par leur importance et divers mécanismes de discrétisation. Les méthodes de discrétisation permettent de calculer pour un très grand nombre de valeurs distinctes n valeurs qui les représenteront. Ces n valeurs distinctes sont soigneusement choisies de façon à minimiser l'erreur moyenne de reconstruction tout en minimisant le nombre de bits requis pour l'encodage de ces n valeurs représentatives. La section 4.1.2 introduit les deux variantes principales, soit la discrétisation de scalaires et la discrétisation de vecteurs. On peut considérer que les scalaires peuvent être des entiers comme des nombres réels (ou du moins, des rationnels). Les vecteurs considérés par la discrétisation sont de dimension finie.

4.1.1 Les techniques de décomposition de signaux

Les techniques de décomposition de signaux peuvent être catégorisées en deux familles principales : la famille des transformées à bases trigonométriques et la famille des transformées à bases à support compact. La première famille est représentée par les transformées de Fourier, de Hartley et de cosinus. Cette dernière transformée est utilisée dans les méthodes de compression d'image JPEG, de vidéo MPEG et de son MPEG-II niveau 3, communément appelé MP3. Elles sont issues de théories classiques surtout liées aux sciences physiques. La seconde famille, représentée par les ondelettes, diffère de la première famille en ce que les bases utilisées ne s'étendent pas sur tout le signal mais sont contenues dans une région finie et délimitée ; cette propriété est connue sous le nom de propriété de support compact.

Dans cette sous-section, nous présenterons les différentes transformées utilisées pour décomposer les signaux en leur fréquences composantes.

4.1.1.1 Transformée de Fourier

La transformée de Fourier a été introduite par Jean-Baptiste Joseph Fourier (1768 – 1832). Les séries de Fourier, comme on les connaît aujourd'hui, découlent d'une des solutions proprosées aux équations différentielles gouvernant la distribution de la chaleur dans les corps [72]. L'hypothèse de Fourier était qu'une fonction suffisamment lisse, périodique sur 2π, est *toujours* représentable par une série de cosinus et de sinus de la forme

$$g(x) = \frac{1}{2}a_0 + \sum_{f=1}^{\infty} a_f \cos(fx) + \sum_{f=0}^{\infty} b_f \sin(fx)$$

où les coefficients a_f et b_f sont obtenus par l'analyse

$$a_f = \frac{1}{\pi}\int_0^{2\pi} g(x)\cos(fx)dx, \qquad b_f = \frac{1}{\pi}\int_0^{2\pi} g(x)\sin(fx)dx$$

Une version plus générale des séries de Fourier est donnée par la transformée de Fourier qui opère sur des nombres complexes. En effet, les séries de Fourier décomposent la fonction $g(x)$ en ses fréquences composantes, mais aucune information sur la phase de ces différentes fréquences n'est obtenue explictement. La transformée de Fourier donne simultanément l'information sur l'amplitude d'une fréquence et sa phase, lesquelles sont encodées par un nombre complexe. Pour une fréquence f, l'amplitude et la phase sont données par

$$\mathfrak{F}_f(g(x)) = \int g(x)e^{-2\pi i f x}dx \tag{4.1}$$

et la fonction est reconstruite par

$$g(x) = \mathfrak{F}^{-1}(\mathfrak{F}(g(x))) = \int \mathfrak{F}_f(g(x))e^{2\pi i f x}df \tag{4.2}$$

Remarquez la symétrie entre les deux transformées. La transformée de Fourier comporte un terme en $-i$ alors que la transformée inverse comporte plutôt un terme en i. Rappelons que $i = \sqrt{-1}$. Les coefficients $\mathfrak{F}_f(\cdot)$ sont des nombres complexes. La transformée est définie pour toute fonction telle que

- $\int |g(x)|dx$ existe,
- les discontinuitées sont en nombre fini,
- et $g(x)$ satisfait la condition de Lipschitz, à savoir pour $\alpha > 0$, pour $\beta \leqslant \alpha$, $B \in \mathbb{R}$, $|g(x+h) - g(x)| < B|h|^{\beta}$.

Ces trois conditions sont suffisamment générales pour englober à peu près toute fonction qui est susceptible de nous intéresser (Du Bois-Reymond a toutefois montré qu'il existe des fonctions qui satisfont les trois énoncés mais pour lesquelles la transformée de Fourier ne converge pas [122]). On peut supposer qu'un segment de signal respecte les trois conditions. Si $|g(x)|$ est fini pour tous les x, et comme la séquence est de longueur finie, l'intégrale de la première condition est définie ; les discontinuitées, s'il y en a, sont bornées par $\sup g(x) - \inf g(x)$, et il suffit de choisir les paramètres de Lipschitz de façon à respecter l'inégalité, s'il est possible de le faire.

Si $g(x)$ est une fonction analytique, on peut calculer directement une forme close pour l'équation eq. (4.1). La transformée inverse est simplement donnée par $g(x)$, que l'on connaît déjà. Dans un ordinateur, par contre, la fonction $g(x)$ (dont on ne connaît pas nécessairement l'expression analytique) est représentée par une suite finie de valeurs. Typiquement, $g(x)$ est définie pour N valeurs, que nous pouvons supposer, sans perte de généralité, régulièrement espacées sur $[0, 2\pi)$, il suffit de reparamétriser $g(x)$ avec $x_j = \frac{j}{N} 2\pi$, pour $j = 0, \ldots, N-1$.

Les éqs. (4.1) et (4.2) deviennent alors

$$\tilde{\mathfrak{F}}_f(g(x)) = \sum_{j=0}^{N-1} g(x) e^{-2\pi i j \frac{f}{N}} \tag{4.3}$$

pour $f = 0, \ldots, N-1$ et

$$g(x) = \sum_{f=0}^{N-1} \tilde{\mathfrak{F}}_f \, e^{-2\pi i x \frac{f}{N}} \tag{4.4}$$

On pourrait penser *a priori* que l'évaluation des formules données aux éqs. (4.3) et (4.4) nécessite $O(N^2)$ opérations pour calculer les N coefficients des fréquences en N points, ou pour reconstruire les N valeurs de $g(x)$. Cependant, grâce au lemme Danielson-Lanczos [220], on peut obtenir une décomposition de l'éq. (4.3) qui sépare les termes d'index pairs des termes d'index impairs. Si N est pair, alors

$$\begin{aligned}
\tilde{\mathfrak{F}}_f(g(x)) &= \sum_{n=0}^{N-1} g(n) e^{-2\pi i f \frac{n}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} g(2n) e^{-2\pi i f \frac{2n}{N}} + \sum_{n=0}^{\frac{N}{2}-1} g(2n+1) e^{-2\pi i f \frac{2n+1}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} g_n^{\text{pair}} e^{-2\pi i f \frac{n}{N/2}} + \omega_n \sum_{n=0}^{\frac{N}{2}-1} g_n^{\text{impair}} e^{-2\pi i f \frac{n}{N/2}} \\
&= \tilde{\mathfrak{F}}_n^{\text{pair}}(g) + \omega_n \tilde{\mathfrak{F}}_n^{\text{impair}}(g)
\end{aligned} \tag{4.5}$$

où $\omega_n = e^{-2\pi i/N}$. Cette factorisation permet de refactoriser récursivement $\tilde{\mathfrak{F}}_n^{\text{pair}}$ et $\tilde{\mathfrak{F}}_n^{\text{impair}}$, jusqu'à ce que $N = 2$. Cela implique par contre que $N = 2^k$ pour un $k \in \mathbb{N}$. Cette factorisation est à la base d'un algorithme de transformée rapide, proposé par Cooley et Tukey [55]. Cet algorithme permet de réduire le coût du calcul de la transformée discrète de Fourier de $O(N^2)$ à $O(N \lg N)$, ce qui représente un avantage indéniable dès que N est moindrement grand.

❁ ❁ ❁

La transformée de Fourier a ceci d'intéressant pour la reconstruction avec perte des fonctions : les amplitudes des coefficients obtenus par les éqs. (4.1) ou (4.3) sur une fonction brownienne décroissent au carré de la fréquence, soit $|\mathfrak{F}_f| \propto \frac{1}{f^{2h}}$, où h est un paramètre qui contrôle la rugosité de la fonction [59]. Une fonction $g(x)$ est brownienne si $g(x + \Delta) = g(x) + \varepsilon(h)$, où $\varepsilon(h) \sim \mathcal{U}(-\frac{h}{2}, \frac{h}{2})$. Cette propriété est intéressante car l'hypothèse que $g(x)$ est brownienne est compatible avec des suppositions raisonnables sur une fonction qui représente un certain nombre d'échantillons d'un son ou d'une image.

Même si la fonction n'est pas brownienne, l'approximation utilisant les n premiers termes de $g(x)$, $\hat{g}_n(x)$ converge aux moindres carrés vers $g(x)$ lorsque $n \to N$. Cette propriété de la transformée de Fourier indique que si on exclut les k derniers termes de la série de coefficients, l'approximation $\hat{g}_{N-k}(x)$ de la fonction $g(x)$ que l'on obtient à partir des $N-k$ coefficients restants, est telle que

$$|g(x) - \hat{g}_{N-k}(x)| \propto \frac{\pi \ln(N-k)}{\sqrt{N-k}}$$

où la constante de proportionnalité ne dépend ni de N ni de x [61, p. 143]. Cette propriété permet d'élaguer la série de coefficients tout en maintenant une certaine qualité de reconstruction de la fonction $g(x)$ par $\hat{g}_n(x)$. On peut obtenir des relations semblables pour des coefficients discrétisés. On peut caractériser l'erreur de reconstruction si on considère les différences entre les $\mathfrak{F}_f$ et les $\hat{\mathfrak{F}}_f$. Cela permet, sachant combien de bits sont nécessaires pour encoder des valeurs particulières des coefficients, de minimiser le nombre total de bits tout en maximisant la qualité de reconstruction. On peut définir une fonction-objectif qui balance adéquatement, selon un critère subjectif, la compression et la qualité de reconstruction.

4.1.1.2 Transformée discrète de Hartley

Alors que la transformée de Fourier utilise des nombres complexes, la transformée de Hartley n'utilise que des nombres réels pour mener à terme la décomposition du signal [91]. La transformée de Fourier utilise le noyau d'analyse $e^{2\pi i f x}$, qui s'exprime aussi par

$$e^{2\pi i f x} = \cos(2\pi f x) - i \sin(2\pi f x),$$

une identité qui est dûe à Euler. Hartley a proposé un noyau d'analyse qui repose sur la fonction $\mathrm{cas}(x) = \cos(x) + \sin(x) = \sqrt{2}\sin(x + \frac{\pi}{4})$ [1]. La transformée de Hartley diffère de la transformée de Fourier en ce que la décomposition de fréquence obtenue est représentée par des coefficients *réels*. La phase est indirectement encodée par le fait que la moitié des fréquences est alignée sur une phase de 0 et l'autre moitié sur une phase de $\frac{\pi}{4}$ radians (soit 45°). On obtient alors une transformée donnée par

$$\mathfrak{H}_f(g(x)) = \frac{1}{\sqrt{2\pi}} \int_0^1 g(x) \,\mathrm{cas}(2\pi f x)\, dx \tag{4.6}$$

et une transformée inverse donnée par

$$g(x) = \frac{1}{\sqrt{2\pi}} \int_0^1 \mathfrak{H}_f \,\mathrm{cas}(2\pi f x)\, df \tag{4.7}$$

Il existe une relation intéressante entre la transformée de Hartley et la transformée de Fourier :

$$\mathfrak{H}_f(g) = \mathrm{Re}\,\mathfrak{F}_f(g) + \mathrm{Im}\,\mathfrak{F}_f(g)$$

où $\mathrm{Re}\,x$ retourne la partie réelle de x, et où $\mathrm{Im}\,x$ retourne la partie imaginaire de x, sans le i. Cette relation est très utile lorsqu'on dispose déjà d'une implémentation de transformée rapide de Fourier. On peut alors réutiliser le même code et n'ajouter que le calcul des sommes.

1 Le nom cas est dérivé de *cosine and sine*.

La transformée de Hartley existe aussi à deux dimensions :

$$\mathfrak{H}_{f_1,f_2}(g(x,y)) = \frac{1}{2\pi}\int_0^1\int_0^1 g(x,y)\,\mathrm{cas}(2\pi(xf_1+yf_2))\,dx\,dy$$

et

$$g(x,y) = \frac{1}{2\pi}\int_0^1\int_0^1 \mathfrak{H}_{f_1,f_2}\,\mathrm{cas}(2\pi(xf_1+yf_2))\,df_1\,df_2$$

Ces dernières versions des équations en deux dimensions ne sont guère pratiques. Les factorisations qui mènent au tranformées rapides sont plus compliquées que si l'on utilise des versions séparables des transformées :

$$\begin{aligned}
\mathfrak{H}'_{f_1,f_2}(g(x,y)) &= \frac{1}{2\pi}\int_0^1\int_0^1 g(x,y)\,\mathrm{cas}(2\pi xf_1)\,\mathrm{cas}(2\pi yf_2)\,dx\,dy\\
&= \frac{1}{\sqrt{2\pi}}\int_0^1\left(\frac{1}{\sqrt{2\pi}}\int_0^1 g(x,y)\,\mathrm{cas}(2\pi xf_1)\,dx\right)\mathrm{cas}(2\pi yf_2)\,dy\\
&= \frac{1}{\sqrt{2\pi}}\int_0^1\left(\mathfrak{H}_{f_1}(g(x,y))\right)\mathrm{cas}(2\pi yf_2)\,dy\\
&= \mathfrak{H}_{f_2}(\mathfrak{H}_{f_1}(g(x,y)))
\end{aligned}$$

et

$$\begin{aligned}
g(x,y) &= \mathfrak{H}^{-1}_{f_2}\left(\mathfrak{H}^{-1}_{f_1}(\mathfrak{H}_{f_1,f_2})\right)\\
&= \frac{1}{\sqrt{2\pi}}\int_0^1\left(\frac{1}{\sqrt{2\pi}}\mathfrak{H}^{-1}_{f_1}(\mathfrak{H}_{f_1,f_2})\right)\mathrm{cas}(2\pi yf_2)\,dy\\
&= \frac{1}{2\pi}\int_0^1\int_0^1 \mathfrak{H}_{f_1,f_2}\,\mathrm{cas}(2\pi xf_1)\,\mathrm{cas}(2\pi yf_2)\,df_1\,df_2
\end{aligned}$$

Ces dernières équations nous disent qu'il est possible de calculer une transformée séparable de Hartley sur une fonction $g(x,y)$ en calculant d'abord sa transformée en x puis sa transformée en y. Si la fonction $g(x,y)$ est une matrice (ce qui correspondrait à une représentation discrète d'une image en mémoire), on applique la transformée de Hartley rangée par rangée puis colonne par colonne. Ce n'est, bien entendu, à strictement parler, pas une transformée de Hartley en deux dimension. C'est l'application successive de la transformée à une dimension sur chacune des dimensions. Ce type de réduction de la difficulté du problème permet de s'attaquer à des espaces avec un nombre arbitraire de dimensions sans pour autant avoir à développer des transformées pour chaque cas.

La tranformée de Hartley existe aussi en version discrète. Pour un vecteur de N points, nous avons

$$\tilde{\mathfrak{H}}_f(g(x)) = \frac{1}{\sqrt{N}}\sum_{t=0}^{N-1} g(t)\,\mathrm{cas}(2\pi f\frac{t}{N}) \tag{4.8}$$

et

$$g(x) = \tilde{\mathfrak{H}}^{-1}(\tilde{\mathfrak{H}}_{f_1}) = \frac{1}{\sqrt{N}}\sum_{f=0}^{N-1}\tilde{\mathfrak{H}}_f\,\mathrm{cas}(2\pi x\frac{f}{N}) \tag{4.9}$$

On peut aussi éliminer le facteur $1/\sqrt{N}$ dans la tranformée et remplacer le facteur $1/\sqrt{N}$ par $1/N$ dans la transformée inverse. Cela donne une plus grande résistance à la discrétisation des coefficients. Nous pouvons aussi factoriser une transformée de Hartley à une dimension d'une façon similiaire à la factorisation de la transformée de Fourier afin d'obtenir une transformée rapide de Hartley. Nous reviendrons dans un chapitre ultérieur sur les raffinements que l'on peut apporter à l'algorithme de base.

La transformée discrète de Hartley se généralise à plusieurs dimensions comme pour la transformée continue. On préférera toutefois la version séparable de la transformée pour les mêmes raisons d'implémentation.

On peut voir la transformée discrète comme un produit matrice vecteur. Soit $\mathbf{H}_N$, la matrice de transformation pour N points. Les éléments de la matrice sont définis par $\mathbf{H}_N(f,t) = \text{cas}(2\pi ft/N)$. Soit $\vec{x}$, un vecteur de longueur N. Alors $\tilde{\mathfrak{H}}(\vec{x}) = \mathbf{H}_N\vec{x}$. Si $\mathbf{P}$ est une matrice $N \times N$, ce qui pourrait être une image, $\mathbf{H}_N(\mathbf{H}_N\mathbf{P})^\mathsf{T}$ calcule la transformée séparable sur toute l'image.

Une référence sur la transformée de Hartley est sans conteste Bracewell qui a beaucoup écrit sur le sujet [22, 23, 24, 25]. D'autres aussi ont écrit sur le sujet, par exemple [123, 65, 214, 215, 158]. On retrouve aussi des références générales qui discourent de la tranformée de Hartley [170, 220].

❁ ❁ ❁

Dans [158], nous avons proposé un système de compression d'image à base de la transformée rapide de Hartley. L'avantage principal de la transformée de Hartley dans ce contexte est que la matrice de transformation 8×8 contient très peu de valeurs distintes. En fait, si on omet le facteur $\frac{1}{N}$ de normalisation, nous nous retrouvons seulement avec ± 1 et $\pm\sqrt{2}$ comme facteurs. De plus, il n'y a que quatre coefficients en $\sqrt{2}$ dans la matrice. Une transformée rapide construite pour $N = 8$ ne nécessite que deux multiplications alors que la DCT en requiert au moins 11, comme nous le verrons à la section suivante.

L'algorithme de compression proposé en [158] procède essentiellement comme l'algorithme JPEG, c'est-à-dire que l'image est transformée de RGB vers YC_rC_b (l'appendice E présente les espaces de couleur) et chaque plan Y, C_r et C_b sont traités séparément. Chaque plan est transformé grâce à la transformée rapide de Hartley. Les coefficients obtenus sont alors décimés, c'est-à-dire soit mis à zéro ou réduit en précision, avant d'être codés avec un code statique précalculé. Le système de compression obtenu est à la fois simple et performant, et se compare bien à JPEG sans en avoir toute la complexité.

4.1.1.3 Transformée discrète de cosinus (DCT)

La prochaine transformée que nous présenterons, la transformée discrète de cosinus, occupe une place importante dans la compression avec perte du son et de l'image. De nombreux standards sont basés sur la DCT (*discrete cosine transform*). Par exemple, JPEG utilise la DCT séparable à deux dimensions pour transformer l'image en bandes de fréquences qui sont par la suite discrétisées et codées efficacement grâce à un code de type Huffman [157]. Le protocole de

compression de la vidéo MPEG utilise un algorithme basé sur la DCT pour coder les « macro-blocs » (morceaux d'images) [141].

La transformée DCT est donnée par les équations suivantes :

$$\mathrm{DCT}_u(g) = \alpha(u) \sum_{x=0}^{N-1} g(x) \cos\left((2x+1)\pi\frac{u}{2N}\right) \tag{4.10}$$

alors que l'inverse est donnée par

$$g(x) = \sum_{u=0}^{N-1} \alpha(u)\mathrm{DCT}_u \cos\left((2x+1)\pi\frac{u}{2N}\right) \tag{4.11}$$

où

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}}, & \text{si } u = 0 \\ \sqrt{\frac{2}{N}}, & \text{sinon} \end{cases}$$

Comme pour la transformée de Hartley, la transformée de cosinus en plusieurs dimensions qui est utilisée est généralement la transformée séparable, donnée par :

$$\mathrm{DCT}_{u,v}(g) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x,y) \cos\left((2x+1)\pi\frac{u}{2N}\right) \cos\left((2x+1)\pi\frac{v}{2N}\right)$$

et

$$g(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v)\mathrm{DCT}_{u,v} \cos\left((2x+1)\pi\frac{u}{2N}\right) \cos\left((2x+1)\pi\frac{v}{2N}\right)$$

On peut utiliser les manipulations appliquées à la transformée de Hartley pour montrer que cette transformation est équivalente à calculer la DCT sur les rangées d'une matrice puis sur ses colonnes. Une factorisation directe des éqs. (4.10) et (4.11) à la façon des transformées rapides de Fourier ou de Hartley mène à des versions efficaces de la transformée discrète de cosinus, mais l'algorithme le plus efficace demande exactement 11 multiplications pour une transformée en 8 points [44], ce qui est mieux que l'algorithme rapide en $O(n \lg n)$ de base qui demanderait 24 multiplications.

4.1.1.4 Transformées ondelettes

Les transformées dites trigonométriques (car leurs noyaux sont basés sur des fonctions trigonométriques telles que $e^{2\pi f t}$, $\sin(x)$ ou $\mathrm{cas}(x)$) imposent un certain nombre de contraintes sur les fonctions qu'elles peuvent analyser, et ces contraintes ne sont pas nécessairement compatibles avec les signaux qui nous intéressent. En particulier, les transformées trigonométriques ont beaucoup de peine à représenter ce que l'on nomme les *transients*. Un transient, c'est un artefact du signal de courte durée et de propriétés différant du reste du signal. L'idée que l'on se fait d'une pointe de surtension bruitée correspond assez bien à ce qu'est un transient. La fig. 4.1 illustre un signal accablé d'un transient.

Les transformées trigonométriques gèrent les fonctions présentant des transients en produisant des décompositions avec beaucoup de grands coefficients associés aux hautes fréquences. Souvenons-nous qu'une fonction brownienne décomposée par la transformée de Fourier a des

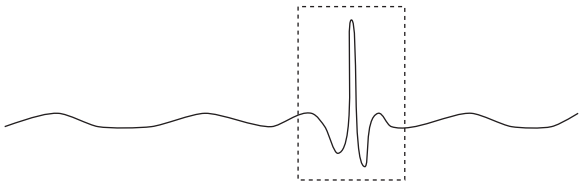


FIG. 4.1 – Un signal lisse affichant un transient (encadré).

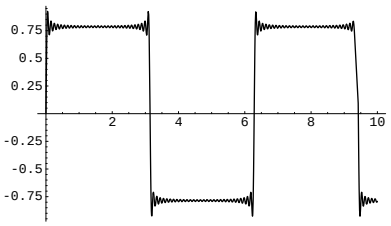


FIG. 4.2 – L'effet du phénomène de Gibbs sur une onde carrée. L'onde carrée ne peut être représentée par une série de Fourier sans exhiber (même avec un nombre infini de termes) les ondulations près des discontinuités.

coefficients dont la magnitude est proportionnelle à f^{-2}. Une fonction essentiellement lisse sauf pour un transient n'exhibe pas cette propriété et les coefficients ne décroissent plus très rapidement en fonction de la fréquence. En fait il faut, pour approximer une onde carrée par une série de Fourier, avoir un nombre infini de fréquences qui décroissent maintenant proportionnellement à f^{-1}.Une onde carrée de période π est décrite par

$$C(t) = \sum_{x=0}^{\infty} \frac{1}{2x+1} \sin\left((2x+1)t\right)$$

En plus, dans ce cas, la reconstruction montre le phénomène de Gibbs, où les discontinuités ne sont plus correctement reconstruites et montrent des oscillations. Le phénomène de Gibbs peut être corrigé par la transformée Fourier-Lanczos [220], mais le nombre de coefficients nécessaires pour obtenir une bonne approximation demeure élevé. La fig. 4.2 montre le phénomène de Gibbs sur une onde carrée.

On s'est rendu très tôt compte des limitations des transformées trigonométriques. Dès 1873, Paul Du Bois-Reymond exposa une fonction continue, périodique sur 2π, pour laquelle la reconstruction à base des séries comme de la transformée de Fourier diffère ou diverge en un certain nombre de points. Haar sera un des premiers à proposer une solution. Il s'était demandé quel autre système de fonctions orthogonales est suffisamment puissant pour représenter *toute* fonction définie sur $[0, 1]$.

Haar proposa une solution basée sur des fonctions à support compact [89]. Alors que les transformées à base trigonométriques utilisent des fonctions de base qui sont périodiques et s'étendent sur toute la longueur de la fonction, les fonctions de Haar sont compactement sup-

FIG. 4.3 – L'ondelette mère de Haar.

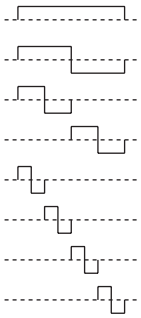


FIG. 4.4 – La base de Haar pour $N = 8$.

portées, c'est-à-dire qu'elles sont définies sur un interval et à zéro à l'extérieur de celui-ci. La fig. 4.3 montre l'aspect général des fonctions de base de Haar. Cette fonction est translatée, dilatée ou contractée en largeur et en hauteur pour obtenir toutes les autres fonctions de base. Ces fonctions sont ensuite additionnées les unes autres pour produire une approximation locale d'une fonction. Bien qu'à ce moment-là on ne parlait pas encore d'ondelettes (il faudra attendre les années 1980), Haar venait effectivement d'inventer les premières ondelettes.

Présentons donc la transformée discrète de Haar. Les vecteurs qui forment la base de Haar sont tous des copies de la même ondelette mère, mais de largeur et de hauteur différentes. Chaque vecteur est orthogonal aux autres (ce qui est nécessaire pour une base orthonormale) mais d'une façon bien particulière : les vecteurs correspondant aux ondelettes de même échelle (soit, par abus de langage, de même fréquence) sont disjoints entre eux. La fig. 4.4 montre la base de Haar pour $N = 8$.

Pour une « fréquence » f, et un vecteur de taille N, nous utiliserons la décomposition unique suivante

$$f = 2^p + q - 1 \tag{4.12}$$

où $p,\ q \in \mathbb{N}$, pour obtenir les paramètres du vecteur de base qui y correspond. Ces paramètres, p et q sont fournis à la fonction génératrice de la matrice. Cette fonction est un peu rébarbative mais très régulière, et est donnée par

$$h_{N,f}(x) = h_{N,p,q}(x) = \begin{cases} +2^{\frac{p}{2}}, & \text{si} \quad \frac{q-1}{2^p} \leq \frac{x}{N} < \frac{q-\frac{1}{2}}{2^p} \\ -2^{\frac{p}{2}}, & \text{si} \quad \frac{q-\frac{1}{2}}{2^p} \leq \frac{x}{N} < \frac{q}{2^p} \\ 0, & \text{sinon} \end{cases} \tag{4.13}$$

pour $x, f = 0, \ldots, N-1$. La matrice de Haar d'ordre N est donnée par $\mathbf{H}^a$ et $\mathbf{H}^a_{f,i} = h_{N,f-1}(i-1)$. Puisque la matrice est orthonormale, la transformée inverse est simplement la transposée de

la matrice initiale, soit $\mathbf{H}^a(\mathbf{H}^a)^\mathsf{T} = \mathbf{I}$. Considérons, en particulier, la matrice

$$\mathbf{H}_8^a = \begin{bmatrix} \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 \\ \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & \frac{1}{\sqrt{2}}^3 & -\frac{1}{\sqrt{2}}^3 & -\frac{1}{\sqrt{2}}^3 & -\frac{1}{\sqrt{2}}^3 & -\frac{1}{\sqrt{2}}^3 \\ \frac{1}{\sqrt{2}}^2 & \frac{1}{\sqrt{2}}^2 & -\frac{1}{\sqrt{2}}^2 & -\frac{1}{\sqrt{2}}^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}}^2 & \frac{1}{\sqrt{2}}^2 & -\frac{1}{\sqrt{2}}^2 & -\frac{1}{\sqrt{2}}^2 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

Ici aussi, il est possible d'obtenir une transformée rapide en factorisant récursivement la matrice de transformation. L'observation importante est que toutes les valeurs dans la matrice sont en fait soit 0, soit une puissance de $\frac{1}{\sqrt{2}}$. On peut en effet exprimer $\mathbf{H}_8^a$ par le produit suivant :

$$\begin{aligned} \mathbf{H}_8^a = & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \\ & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \\ & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \end{aligned} \tag{4.14}$$

dont on peut induire la relation générale

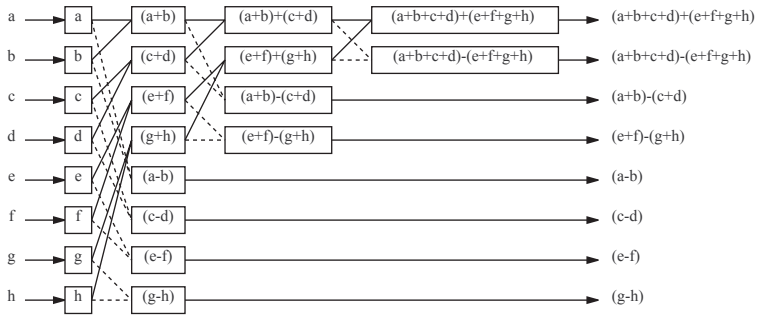


FIG. 4.5 – Le flot de données pour la transformée rapide de Haar, avec N=8.

$$\mathbf{H}_n^a = \begin{bmatrix} \mathbf{S}_2 & 0 \\ 0 & \mathbf{I}_{n-2} \end{bmatrix} \left(\begin{bmatrix} \mathbf{S}_4 & 0 \\ 0 & \mathbf{I}_{n-4} \end{bmatrix} \left(\begin{bmatrix} \mathbf{S}_8 & 0 \\ 0 & \mathbf{I}_{n-8} \end{bmatrix} (\cdots \mathbf{S}_n) \cdots \right) \right) \tag{4.15}$$

où I_{n-k} est la matrice identité $(n-k) \times (n-k)$, les zéros représentent des matrices nulles $(n-k) \times n$ et $n \times (n-k)$. $\mathbf{S}_k$ est une matrice $k \times k$ definie par

$$\mathbf{S}_k(i,j) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{si} \quad (i < \frac{k}{2}) \wedge ((j = 2i-1) \vee (j = 2i)) \\ \frac{1}{\sqrt{2}}, & \text{si} \quad (i \geq \frac{k}{2}) \wedge (j = 2(i - \frac{k}{2}) - 1) \\ -\frac{1}{\sqrt{2}}, & \text{si} \quad (i \geq \frac{k}{2}) \wedge (j = 2(i - \frac{k}{2})) \\ 0 & \text{sinon} \end{cases} \tag{4.16}$$

où i et j vont de 1 à k. La décomposition (4.15) pour tout $N \sim 2^s$, $s \in \mathbb{N}$. Finalement, le produit $\mathbf{H}_N^a \vec{x}$ est donné par

$$\mathbf{H}_n^a \vec{x} = \begin{bmatrix} \mathbf{S}_2 & 0 \\ 0 & \mathbf{I}_{n-2} \end{bmatrix} \left(\begin{bmatrix} \mathbf{S}_4 & 0 \\ 0 & \mathbf{I}_{n-4} \end{bmatrix} \left(\begin{bmatrix} \mathbf{S}_8 & 0 \\ 0 & \mathbf{I}_{n-8} \end{bmatrix} (\cdots \mathbf{S}_n \vec{x}) \cdots \right) \right) \tag{4.17}$$

La factorisation (4.15) permet de structurer un algorithme afin de calculer la transformée rapide de Haar. On remarquera qu'aux équations éqs. (4.14) et (4.15) la partie basse de la matrice est en fait une matrice identité. Cela veut dire que le produit entre une matrice de cette forme (avec une matrice carrée au coin supérieur gauche et une identité au coin inférieur droit) et un vecteur laisse les derniers éléments du vecteur inchangés. En programmation, ça ce traduit par une copie des valeurs, ou mieux, à des valeurs inchangées dans un algorithme *in situ*. On peut dériver un algorithme dont le flot de données est montrée à la fig. 4.5. Cet algorithme, plutôt qu'être en $O(N^2)$, est en temps *linéaire*, soit $O(n)$!

Les ondelettes de Haar sont discrètes mais il existe d'autres classes d'ondelettes. Il y a des ondelettes discrètes mais plus lisses que celles de Haar et aussi des ondelettes continues. Les ondelettes de Haar ont la fâcheuse caractéristique d'introduire des effets de blocs dans les images reconstruites (voir fig. 4.6) lorsque les coefficients sont lourdement discrétisés. Lorsque ces artefacts sont inacceptables (et c'est très souvent le cas) on aura recours à des ondelettes d'ordre

supérieur. Les ondelettes de Haar sont d'ordre 2, ou binaire (en anglais on trouve souvent *dyadic*) car le schéma d'analyse regroupe les valeurs du signal deux par deux (cf. fig. 4.5) pour produire la transformation. Il existe des ondelettes d'ordre 4, 6, (ondelettes Dubuc-Deslauriers 4, Daubechies-6, etc.) qui ont des formes plus « lisses » que les ondelettes de Haar.

Encore plus « lisses », on trouve les ondelettes continues. Soit $\psi(t)$, l'ondelette mère d'une ondelette continue. Les ondelettes filles sont données par

$$\psi_{u,s}(t) = \frac{1}{\sqrt{s}}\,\psi\left(\frac{t-u}{s}\right)$$

où u désigne la translation et s le changement d'échelle de l'ondelette. Les coefficients ondelettes de la fonction $g(x)$ sont obtenus par l'analyse

$$\omega(a,b) = \int g(x)\bar{\psi}_{a,b}(x)\,dx$$

où $\bar{\psi}$ est le conjugé de ψ. La synthèse, ou reconstruction, est calculée en x par

$$g(x) = \int \omega(a,b)\psi_{a,b}(x)\,\partial a\,\partial b$$

La propriété la plus importante des ondelettes, c'est la propriété de *multirésolution*. De façon informelle, une base vectorielle (de dimension finie ou infinie) possède la propriété de multirésolution (ou, simplement, est multirésolution) si ne conserver que la moitié des coefficients qui correspondent aux fréquences basses permet de reconstruire une version à résolution réduite de moitié. Par exemple, si on a une fonction en, disons, n points, et que l'on ne conserve que la moitiée des coefficients, on obtient la meilleure approximation en $n/2$ points de la fonction, ce qui revient à couper la résolution en deux. Conserver $n/4$ coefficients donnera une reconstruction à la résolution $n/4$, etc.

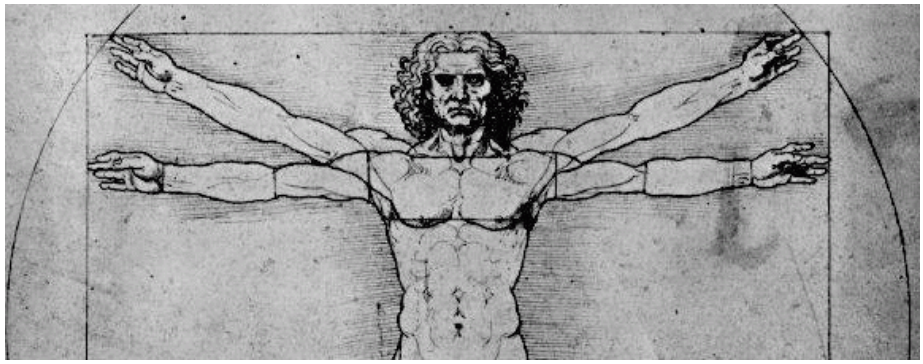
Définissons formellement la propriété de multirésolution. Une base (ondelette ou non) engendre un espace vectoriel V_N à N dimension. Tout sous-ensemble de la base engendre des sous-espaces vectoriels V_i de l'espace V. Soit une base $U = \{u_0, u_1, \ldots, u_N\}$, où les vecteurs sont ordonnés en ordre de compacité croissante. Nous ne considérerons que les sous-ensembles $\{\{u_0\}, \{u_0, u_1\}, \{u_0, u_1, u_2, u_3\}, \ldots, \{u_0, \ldots, u_N\}\}$. Chaque sous-ensemble engendre respectivement les sous-espaces V_0,V_1,...,V_N, des espaces vectoriels.

Reprenons la définition de Mallat [132, p. 221]. Une base est multirésolution si, et seulement si :

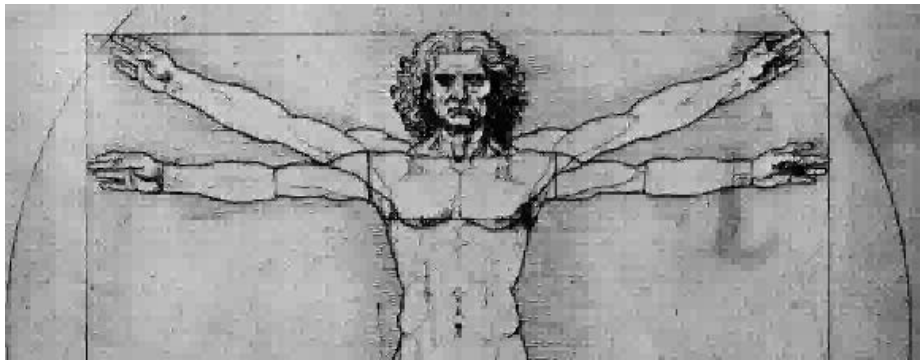
- Elle est invariante sous la translation. Appliquer une translation à une région de la fonction $g(t)$ par une quantité qui dépend de la résolution courante ne change pas la résolution de la reconstruction. Formellement : $\forall\, j, k \in \mathbb{Z}$, $(g(t) \in V_j) \;\leftrightarrow\; (g(t - 2^j k) \in V_j)$.

- Les espaces vectoriels engendrés sont strictement inclus, c'est-à-dire que $\forall\, j$, $V_j \subset V_{j+1}$.

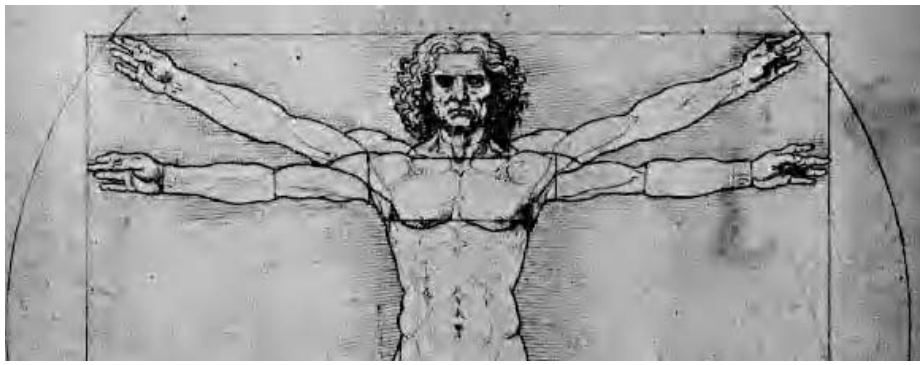
- Pour doubler la résolution on double le nombre de coefficients. $\forall\, j$, on a $(g(t) \in V_j) \;\leftrightarrow\; (g(t/2) \in V_{j+1})$.



a)



b)



c)

FIG. 4.6 – Comparaison des méthodes d'ondelettes. L'image originale est en a), l'image reconstruite avec des ondelettes de Haar (dont les coefficients ont été lourdement discrétisés) est en b), en c) on a l'image reconstruite à partir des ondelettes Dubuc-Deslauriers (4,4) avec un nombre de bits comparables au nombre de bits conservés pour reconstruire b).

– Seul le zéro appartient à tous les espaces vectoriels, $\bigcap_j V_j = \{0\}$.

– L'union de tous les sous-espaces est telle que l'énergie des fonctions est finie, soit encore $\bigcup_j V_j = \mathbb{L}^2(\mathbb{R})$.

❁ ❁ ❁

La caractéristique de multirésolution correspond naturellement à ce que l'on pourrait attendre d'une base qui sied à la compression avec perte d'un signal. La propriété de multirésolution nous dit que détruire la moitié des coefficients qui correspond aux fréquences hautes n'affecte la reconstruction que dans la mesure où la résolution obtenue est réduite par deux. Cependant, la propriété la plus importante des bases ondelettes, c'est le caractère local des ondelettes individuelles. Alors que pour représenter une discontinuité dans un signal grâce à une décomposition trigonométrique il faut utiliser un grand nombre de termes de hautes fréquences (c.f., le phénomène de Gibbs), avec une transformée ondelette, nous aurons des termes de grand module seulement autour de la discontinuité, et tous les autres termes correspondant aux hautes fréquences dans les régions lisses seront essentiellement nuls. Si on se réfère à la fig. 4.4 qui montre les ondelettes de Haar pour $N = 8$, on voit pourquoi une discontinuité très localisée n'affecte qu'un nombre logarithmique de coefficients — qui correspondent à toutes les ondelettes qui se trouvent directement sous la discontinuité. Cette propriété permet par exemple de reproduire très fidèlement les contours accentués des images sans ajouter plus de termes que nécessaire. Les méthodes de compression d'image qui utilisent des ondelettes exploitent explicitement la localité des ondelettes.

4.1.2 Les techniques de discrétisation

Les techniques de décomposition de signal vues à la section précédente transforment les données de façon à mettre en valeur certaines de leur propriétés. Pour les transformées de la famille trigonométrique, les signaux sont représentés par les différentes amplitudes et phases de leurs fréquences composantes. Pour les transformées de la famille ondelette, les signaux sont représentés par des coefficients d'ondelettes selon leur position, échelle et amplitude. Heureusement, pour les bases ondelettes, la position et l'échelle sont implicites, nous ne conservons que l'information d'amplitude. Des méthodes crues de compression utilisant l'une ou l'autre famille pourraient consister en la destruction d'un certain nombre k de coefficients, par exemple en les mettant tout simplement à zéro et en ne conservant que les $N - k$ premiers coefficients. Nous avons vu que les transformées de Fourier et de Hartley convergent quand même aux moindres carrés lorsqu'on élague la série des coefficients. Les transformées ondelettes, quant à elles, ne font que produire une version à plus faible résolution du signal, ce qui peut sembler tout aussi acceptable.

Nous avons vu, à la section 4.1.1.1, p. 42, de quelle façon était affectée la reconstruction de la fonction lorsque nous élaguions les k derniers termes d'une série de Fourier. On peut, de même, calculer les impacts sur la reconstruction lorsque les termes sont modifiés par une fonction de discrétisation $Q(\cdot)$. Dans le cas de la transformée de Fourier, la différence entre la fonction $g(x)$

et la reconstruction modifiée $\hat{g}(x)$ est donnée par

$$\begin{aligned} \|g(x) - \hat{g}(x)\| &= \left\| \int \mathfrak{F}_f \, e^{2\pi i f x} \, df - \int Q\left(\mathfrak{F}_f\right) e^{2\pi i f x} \, df \right\| \\ &= \left\| \int (\mathfrak{F}_f - Q\left(\mathfrak{F}_f\right)) e^{2\pi i f x} \, df \right\| \end{aligned}$$

pour une métrique $\|x\|$. La fonction $Q(\cdot)$ réduit la précision de son argument. Ici, réduire la précision signifie qu'un nombre x sera approximé par un autre nombre $\hat{x}$ qui demande moins de bits pour être encodé. Autrement dit, les nombres $\hat{x}$ prendront moins de valeurs distinctes que les x. L'approximation est « satisfaisante » si l'erreur $\|g(x) - \hat{g}(x)\|$ est acceptable pour l'application. La fonction $Q(\cdot)$ jouera donc un rôle essentiel pour l'optimisation de la compression. D'une part, $Q(x)$ contrôle la qualité de la reconstruction, d'autre part $|Q(x)|$ contrôle le nombre de bits nécessaires à la représentation des $\hat{x}$ selon le code que nous avons décidé d'utiliser. Le problème devient donc de balancer entre la qualité de reconstruction que nous voulons maximiser (minimiser $\|g(x) - \hat{g}(x)\|$) et la quantité de bits nécessaires que nous voulons minimiser.

Le problème de la discrétisation, c'est de trouver des valeurs $\hat{x}$ « satifaisantes ». Nous appelerons ces valeurs *prototypes*. Nous supposons que $x \sim X$, c'est-à-dire que les valeurs (scalaires ou vectorielles) sont issues d'une variable aléatoire X dont nous ne connaissons pas nécessairement les paramètres. La première donnée du problème, c'est n, le nombre de valeurs distinctes que nous voulons obtenir pour les prototypes $\hat{x}_i$, $i = 0, \ldots, n-1$. Nous voulons aussi réduire le nombre de prototypes au minimum — car un grand nombre de prototypes veut dire beaucoup de bits par index pour chaque prototype émis. Les $\hat{x}_i$ seront choisis de façon à minimiser l'erreur de reconstruction.

Lors de la discrétisation, la valeur de $Q(x)$ est déterminée par

$$Q(x) = \arg\min_{\{\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1}\}} \|x - \hat{x}_i\|$$

L'erreur moyenne encourue, pour un ensemble de valeurs prototypes $\{\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1}\}$, et $x \sim X$, est donnée par un critère, par exemple

$$E = \inf \ \mathrm{E}[\, d(x, Q(x)) \,] \tag{4.18}$$

où $d(x, Q(x))$ est une fonction de distortion, typiquement $(x - Q(x))^2$, ou encore une métrique appropriée.

On aura donc avantage à considérer la loi X pour déterminer les prototypes afin de minimiser l'erreur de reconstruction, du moins sous la métrique choisie et sous l'hypothèse que les valeurs sont i.i.d. Notons que cette formulation du problème s'applique autant aux scalaires qu'aux vecteurs. Dans cette section, nous allons très brièvement présenter quelques techniques de discrétisation de scalaires et de vecteurs. Nous allons présenter des méthodes que l'on pourrait qualifier de crues : ces méthodes ignorent la fonction de distribution des x à quantifier. Nous allons aussi présenter des algorithmes qui tiennent compte explicitement ou implicitement des distributions pour produire les meilleurs prototypes étant donné leur nombre n.

4.1.2.1 Discrétisation de scalaires

La tâche de discrétiser des scalaires consiste essentiellement à prendre un ensemble de valeurs réelles et de les représenter par n valeurs réelles, d'une façon qui minimise l'éq. (4.18).

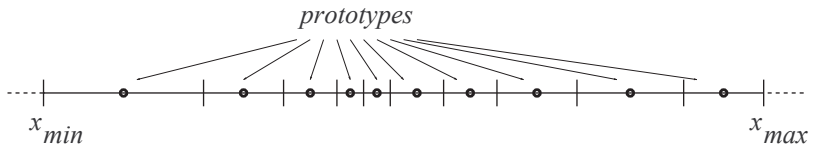


FIG. 4.7 – Un intervalle de la ligne des réels divisé en régions de discrétisation. Les prototypes sont choisis comme étant les centres des régions.

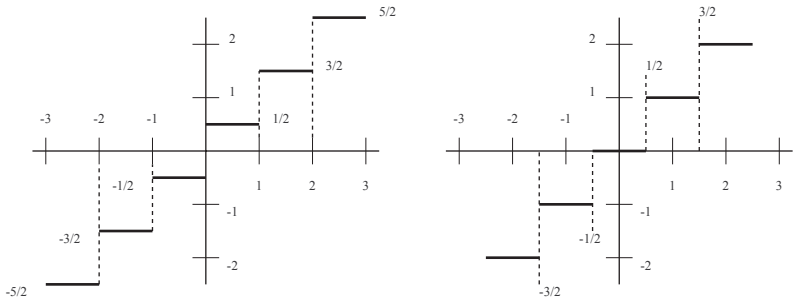


FIG. 4.8 – Les discrétiseurs *midrise* et *midthread*.

Supposons, pour la simplicité de l'exposé, que toutes les valeurs originales soient contiguës sur la droite réelle, sur l'interval $[x_{min}, x_{max}]$. Les méthodes les plus simples vont simplement découper l'intervalle $[x_{min}, x_{max}]$ en sous-intervalles, comme montré à la fig. 4.7. Dépendamment comment les intervalles sont définis par rapport aux nombres : on parle de discrétisation avec frontières *midrise* lorsque les intervalles de discrétisation sont alignés avec les nombres entiers, et avec frontières *midthread* lorsque les intervalles de discrétisation sont à cheval sur les entiers (voir fig. 4.8). La façon de déterminer les intervalles peut être très crue. Par exemple, on pourrait imaginer une méthode de discrétisation qui prend l'intervalle des valeurs permises et le divise en n intervalles de même taille. Cette méthode de discrétisation peut être décrite par les équations suivantes :

$$s = \frac{1}{n}(x_{max} - x_{min})$$

$$\tilde{x} = Q_s(x) = \lfloor \frac{1}{s}(x - x_{min}) \rfloor$$
$$\hat{x} = Q_s^{-1}(\tilde{x}) = \lfloor x_{min} + s(\tilde{x} + \tfrac{1}{2}) \rfloor$$

Cette méthode n'est pas très judicieuse mais elle est très simple. La transformation que l'on fait subir à x pour obtenir $\tilde{x}$ (qui est dans ce cas simplement le numéro de l'intervalle qui contient x) est linéaire. D'autres méthodes de discrétisation utilisent des transformations non linéaires, en particulier la transformation logarithmique. La plus simple des méthodes de discrétisation

logaritmique est donnée, pour $x \in \mathbb{N}$, par :

$$\tilde{x} = Q(x) = \lfloor \lg x \rfloor$$
$$\hat{x} = 2^{\tilde{x}} + 2^{\tilde{x}-1}$$

Cette discrétisation est très crue. Cependant, les systèmes téléphoniques américains et japonais utilisent une variante de ce type. Cette méthode, dite du *μ-law* est contrôlée par un paramètre μ. La discrétisation et la fonction d'approximation sont plus compliquées mais donnent beaucoup plus de souplesse quant aux discrétisations obtenues.

$$\tilde{x} = Q_\mu(x) = x_{max} \frac{\ln\left(1 + \mu \frac{|x|}{|x_{min}|}\right)}{\ln(1+\mu)} \text{signe}(x)$$
$$\hat{x} = Q_\mu^{-1}(\tilde{x}) = \frac{x_{max}}{\mu}\left((1+\mu)^{\frac{|\tilde{x}|}{x_{max}}} - 1\right)\text{signe}(\tilde{x}) \tag{4.19}$$

Le paramètre $\mu = 255$ pour les systèmes téléphoniques américains et japonais. En Europe, une variante, *α-law*, est utilisée. Ces méthodes exploitent le fait que notre oreille a une réponse logarithmique à la puissance du son — ce pourquoi les décibels sont le logarithme de la puissance.

Une méthode de discrétisation qui tient compte des probabilités d'occurence des x consisterait à simplement diviser l'intervalle $[x_{min}, x_{max}]$ de façon à ce que la probabilité que x soit dans l'une ou l'autre des régions soit égale. Plutôt que de diviser l'intervalle en régions de longueurs égales, on va avoir des régions de probabilités égales. Il existe plusieurs algorithmes, plus ou moins efficaces, pour déterminer exactement ces régions. Pour les scalaires, il existe une solution analytique qui génère directement les partitions selon une fonction de distribution. Il suffit en effet de déterminer les régions délimitées par les quantiles de la fonction de distribution, ce qui se fait en temps linéaire, en terme du nombre d'échantillons qui représentent la distribution.

Un autre algorithme est l'algorithme Lloyd-Max [134, 129]. C'est un algorithme itératif qui génère d'abord des intervalles réguliers sur $[x_{min}, x_{max}]$, puis change les frontières des intervalles de façon à minimiser une mesure de type moindres carrés, avec un algorithme qui rappele la descente de gradient. L'algorithme répète jusqu'à convergence. L'algorithme, tout comme l'algorithme K-means, converge très rapidement, de façon à ce que seulement quelques itérations produisent déjà une très bonne approximation de la solution optimale.

Les algorithmes de discrétisation peuvent être dynamiques, c'est-à-dire qu'ils peuvent ajuster les régions en fonction des observations. Le plus simple de ces algorithmes est la méthode de Jayant [105, 106]. La méthode, que nous ne décrirons pas dans le détail, consiste à diviser l'invervalle $[x_{min}, x_{max}]$ (que l'on suppose, pour la simplicité, centré sur zéro) en un certain nombre d'intervalles de longueur Δ. Au début, tous les intervalles sont de même longueur. Au fur et à mesure que les x sont discrétisés, on verifie si x tombe dans les intervalles intérieurs (c'est-à-dire plus près du centre que des extrémités de l'intervalle $[x_{min}, x_{max}]$) ou dans les intervalles extérieurs (qui sont plus près des extrémités de $[x_{min}, x_{max}]$ que de son centre). Si x tombe dans un des intervalles intérieurs, on réduit Δ par une petite valeur ε. Si x tombe dans un des intervalles extérieurs, on augmente Δ par ε. Le choix de ε et de la limite entre les intervalles intérieurs et extérieurs déterminent si l'algorithme s'adapte rapidement ou lentement.

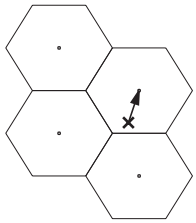


FIG. 4.9 – Les prototypes sont choisis en fonction d'un bassin d'attraction. Dans le cas d'une métrique L_2, les frontières sont rectilinéaires et les bassins d'attraction sont des partitions convexes de l'espace.

Il existe de nombreux algorithmes efficaces pour calculer la discrétisation de scalaires, mais puisque tous minimisent l'éq. (4.18), ils ne diffèrent essentiellement que par la métrique qui est utilisée et les hypothèses sur la source aléatoire. La méthode d'optimisation repose presque toujours sur un critère de type moindres carrés.

4.1.2.2 Discrétisation de vecteurs

Le problème de discrétisation de vecteurs est similaire au problème de la discrétisation de scalaires. Les discrétisations les plus simples découpent l'espace $\mathbb{R}^d$ en cuboïdes[2] ou en régions facilement déterminables. Les discrétisation à base de trellis (*lattices*, en anglais) sont un bon exemple de détermination de régions régulières. Les treillis utilisent une matrice (une base sur $\mathbb{R}^d$) pour générer les prototypes sur un grillage régulier. Les régions de discrétisation sont données par la zone d'influence des prototypes, déterminée par la mesure de distortion choisie. Sur $\mathbb{R}^2$, on trouve, entre autres, les treillis D_1 (grille carrée), D_2 (grille en quinconce, fig. 4.11) et A_2 (grille hexagonale, fig. 4.10).

$$D_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad D_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix}$$

Un vecteur est alors représenté par la combinaison $\tilde{x} \in \mathbb{Z}^2$ telle que pour le treillis B, $\tilde{x} = \arg\min \|\vec{x} - B\tilde{x}\|$ (considérez la fig. 4.9). Bien que $\tilde{x} \in \mathbb{Z}^2$, on ne peut s'attendre à ce que les vecteurs soient équiprobables. On encodera les $\tilde{x}$ avec un code efficace qui tient compte des probabilités d'occurences. Il pourra alors être nécessaire d'avoir une fonction de pairage $q : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ de façon à ce que, pour $\tilde{x}, \tilde{y} \in \mathbb{Z}^2$, $q(\tilde{x}) < q(\tilde{y}) \leftrightarrow P(\tilde{x}) \geqslant P(\tilde{y})$. Un exemple de fonction de pairage est montré à la fig. 4.11. Nous présentons quelques fonctions de pairage à la section 5.3.6.

L'algorithme K-means est particulièrement efficace pour calculer des régions dans l'espace en tenant compte des probabilités. On commence par choisir un paramètre K, qui se trouve à être le nombre désiré de prototypes. À $t = 0$, on initialise les K centroïdes $C_i(t)$ à des valeurs aléatoires mais admissibles (c'est-à-dire qui font partie de l'espace vectoriel à discrétiser). Ensuite, on assigne à chaque valeur admissible un centroïde : le centroïde le plus près, en utilisant la distance au carré. Ensuite, avec toutes les valeurs qui ont reçu le même centroïde, on recalcule le centroïde comme étant la moyenne vectorielle de ces valeurs. On recalcule ainsi tous les centroïdes, on réassigne à chaque vecteur son nouveau centroïde, et on recommence jusqu'à ce qu'il n'y a plus de changement dans les centroïdes. Sommairement, l'algorithme est donné par :

1. $t = 0$.

2 I.e., en régions rectangulaires dont les côtés sont parallèles aux axes.

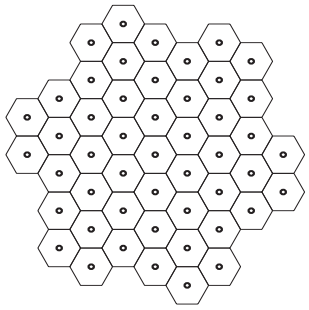


FIG. 4.10 – Le treillis A_2.

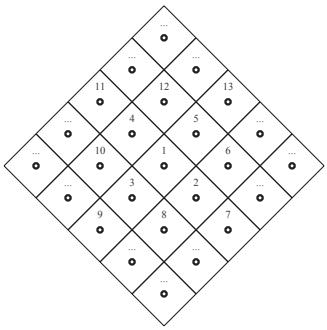


FIG. 4.11 – Le treillis D_2 avec une numérotation arbitraire.

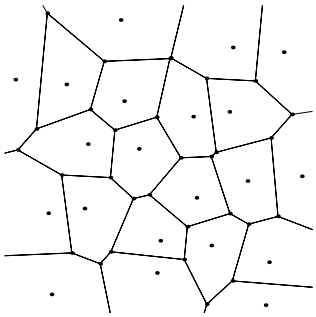


FIG. 4.12 – Les régions générées par l'algorithme K-means, avec une métrique L_2. Les points au centre des cellules représentent les prototypes.

2. Initialiser les K prototypes (les $C_i(t)$) aléatoirement.
3. Incrémenter t.
4. Calculer $D_t(\vec{x}) = \arg\min_i \|\vec{x} - C_i(t-1)\|$.
5. Calculer les nouveaux prototypes, $C_i(t+1) = \sum_{\vec{x}|D_{t-1}(\vec{x})=i} \vec{x} P(\vec{x})$.
6. Tant qu'un $C_i(t)$ a été suffisamment modifié, aller à 3).
7. Les $C_i(t)$ sont les prototypes désirés.

L'algorithme K-means est très général quant aux régions de décision qui sont produites, car elles ne dépendent que de la fonction de distance qui est utilisée. Si la fonction de distance utilisée est L_2 ($\|\vec{x}\|_2 = (\sum_i x_i^2)^{\frac{1}{2}}$), on a des régions qui sont convexes, avec les métriques L_1 ou L_∞ (respectivement, $\|\vec{x}\|_1 = \sum_i |x_i|$, $\|\vec{x}\|_\infty = \max_i |x_i|$), on a des régions possiblement concaves mais telles que les plans de décision sont parallèles, perpendiculaires ou diagonaux par rapport aux axes de l'espace. Avec d'autres fonctions de distance on peut créer des régions avec des frontières aux courbes exotiques (voir, par exemple, [15]). Pour une métrique L_2, le résultat d'une partition K-means est montré à la fig. 4.12.

L'algorithme Linde-Buzo-Gray est une généralisation à un nombre quelconque de dimensions de l'algorithme Lloyd-Max [128]. L'algorithme calcule itérativement des régions de discrétisation convexes avec des frontières linéaires. Cet algorithme, très lié à l'algorithme K-means, calcule itérativement une partition de l'espace, et le centroïde des cellules fournit le prototype associé à une région. L'algorithme, comme l'algorithme de Lloyd-Max, converge très rapidement vers un minimum local.

Il existe d'autres algorithmes de discrétisation de vecteurs. Certains construisent des hiérarchies de prototypes. On parle alors de discrétisation structurée en arbre (*tree structured quantization*). Les prototypes arrangés en arbre permettent de choisir la profondeur de la subdivision de l'espace que l'on veut et ainsi contrôler dynamiquement le nombre de prototypes (et le nombre de bits) que l'on veut utiliser. Les algorithmes *BFOS* (Breiman, Friedman, Olshen, Stone) et *generalized BFOS* qui génèrent des discrétisations en arbre sont présentés dans [27, 47, 81].

On notera cependant que les méthodes de discrétisation vectorielles souffrent de ce que l'on nomme — un peu théâtralement peut-être — la *malédiction de la dimensionalité*, une expression

que l'on doit à Bellman [13]. Lorsque la dimension d de l'espace croît, on se retrouve face à deux problèmes. Le premier est que le volume d'espace croît exponentiellement en d, le second étant que le nombre d'observations est nessairement exponentiellement ténu par rapport au volume d'espace considéré. La complexité du calcul peut aussi, selon l'algorithme utilisé, croître de façon polynomiale ou même exponentielle en d, ce qui fait que, sauf pour des d petits, on rencontre rapidement des difficultés quasi insurmontables.

4.2 Compression sans perte

La compression avec perte utilise des techniques de décomposition de signaux pour extraire les caractéristiques importantes du signal et pouvoir détruire les détails qui importent peu ou pas dans la reconstruction. La décomposition de signal est suivie d'une étape de discrétisation, laquelle est à son tour suivie par une étape de codage efficace des index et des numéros de prototypes. La compression sans perte utilisera aussi un certain nombre de techniques de décomposition, mais plutôt qu'être essentiellement axées sur la décomposition de fréquences, les techniques de décomposition utilisées dans la compression sans perte sont axées sur des décompositions syntaxiques.

Les techniques de décomposition syntaxique vont créer un découpage qui expose les redondances de la séquence. Les techniques à base de dictionnaire dissèquent la séquences en sous-séquences pour réaliser la compression. Les techniques consistent à découper la séquence en « mots » ou « phrases » qui se répètent souvent, assigner des index à ces découpages, puis trouver un code efficace pour les index. La première famille de méthodes, dite LZ77, utilise des index qui réfèrent à une fenêtre coulissante sur la séquence pour compresser. La méthode de base a été décrite par Ziv et Lempel [235]. La deuxième famille utilise une structure de données auxiliaire, un « dictionnaire », pour emmagasiner les sous-séquences et les index refèrent maintenant aux entrées dans le dictionnaire. Cette classe d'algorithme a aussi été décrite par Ziv et Lempel [236].

Outre les techniques de décomposition syntaxique, nous trouvons aussi les techniques à base de transformations, comme la transformée Burrows-Wheeler qui opère un type particulier de permutation sur les données pour en faciliter la compression [32]. Nous trouvons aussi des méthodes basées sur les prédictions. Les prédictions sont formulées à partir d'une modélisation statistique de la séquence à compresser. La prédiction peut être une estimation de la prochaine valeur à coder dans la séquence comme elle peut être une fonction de distribution de probabilité conditionnelle pour le prochain symbole dans la séquence. Si la prédiction formulée est une fonction de distribution, celle-ci servira à générer un code efficace pour le prochain symbole dans la séquence.

Le premier type de prédicteur, celui où la valeur la plus probable pour le prochain symbole de la séquence est généré, sert à générer les différences entre la séquence et les prédictions. Ce seront les différences qui seront codées. Le but est de ramener la moyenne des valeurs à zéro et d'obtenir une distribution compacte des valeurs possibles de façon à pouvoir utiliser un code efficace pour une distribution de la famille exponentielle, la distribution géométrique, par exemple.

Le second type de prédicteur génère directement la fonction de distribution pour le prochain symbole dans la séquence. L'information sur la distribution peut être utilisée par un codeur efficace, comme le codeur de Huffman (auquel nous consacrerons un chapitre complet, le chapitre 6) ou le codeur arithmétique, pour produire un code pour le prochain symbole. Avec un codeur

efficace, les symboles les plus probables recevront des codes plus courts, et l'algorithme de compression donnera de bonnes performances si la prédiction est bonne. On considérera que la prédiction est bonne si la fonction de distribution prédite concorde avec la distribution observée *a posteriori*. Cette mesure est habituellement la mesure de Kullback-Leibler, qui mesure la divergence (en bits) entre une distribution estimée $g(x)$ et la distribution observée *a posteriori* $f(x)$. Cette mesure est donnée par

$$KL(g,f) = \mathrm{E}\left[\lg \frac{g(x)}{f(x)}\right] = \int \lg \frac{g(x)}{f(x)} g(x)dx = \int g(x)\left(\lg g(x) - \lg f(x)\right)dx$$

puisque $KL(g,f) \geqslant 0$, on peut rendre cette mesure symétrique en posant

$$KL_s(g,f) = KL(g,f) + KL(f,g)$$

Notons que la mesure de Kullback-Leibler revient à peu près à calculer $\mathcal{H}(g(x)) - \mathcal{H}(f(x))$, c'est-à-dire les différences entre les codes (que l'on suppose optimalement) efficaces générés grâce à la distribution estimée $g(x)$ et la distribution observée *a posteriori* $f(x)$.

Il existe plusieurs mécanismes pour formuler des prédictions sur des fonctions de distribution. Nous considérerons des méthodes qui sont des variantes sur le thème des chaînes de Markov de longueur variable. Les modèles de prédiction par concordances partielles (*Prediction by Partial Match*, ou simplement PPM) correspondent à une mixtures de chaînes de Markov d'ordre m et moins, pour un m qui sied à la classe de séquence qui nous intéresse. Les modèles à base d'automates enrichis correspondent plutôt à une chaîne de Markov de longueur variable et gèrent implicitement l'horizon des influences dans la séquence.

Bien que nous nous attarderons plus sur les prédicteurs qui considèrent un assez grand horizon sur la séquence, nous pouvons aussi concevoir des prédicteurs d'ordre zéro, c'est-à-dire que la distribution qui est générée est inconditionnelle, soit simplement $P(X_t \mid \theta)$, où θ est le vecteur de paramètres de la distribution, plutôt que $P(X_t = x_t \mid x_1^{t-1}, \theta)$ [3]. Il existe en effet des séquences qui sont des collections d'instances d'une variable aléatoire i.i.d. ; il est alors inutile de chercher à modéliser les dépendences entre les instances.

4.2.1 Les méthodes à base de dictionnaire

Les méthodes de compression par dictionnaire les plus simples utilisent un dictionnaire qui a été précalculé pour une classe de séquences. Parmi celles-ci, nous trouvons le codage par bigramme. Supposons que nous ayons un nombre de symboles n tel que $2^{k-1} < n < 2^k$. Nous avons donc $2^k - n$ symboles disponibles pour utiliser exactement k bits. Souvent k correspond à une frontière naturelle pour la machine, comme $k = 8$. Nous allons donc assigner ces $2^k - n$ symboles à des bigrammes. Les bigrammes sont des paires de symboles que l'on retrouve dans la séquence.

3 Du point de vue bayesien, même si on a que $P(X_t = x_t \mid x_1^{t-1}, \theta) = P(X_t \mid \theta)$, c'est-à-dire que les x_t sont i.i.d, ça ne veut pas nécessairement dire que $P(X_t = x_t \mid x_1^{t-1}) = P(x_t)$. En fait, il y a une dépendence cachée entre θ et x_i^{t-1}. Typiquement on a

$$P(X_t = x_t \mid x_1^{t-1}) = \int P(x_t \mid \theta) P(\theta \mid x_1^{t-1})\, d\theta$$

qui devient simplement, dans le cadre de l'estimation maximum de vraisemblance $P(x_t \mid x_1^{t-1}) = P(x_t \mid \theta^*) P(\theta^* \mid x_1^{t-1})$.

Si on considère les bigrammes en français, on retrouvera, par exemple, fr, ff, mb, etc. Les $2^k - n$ plus fréquents bigrammes sont choisis. Pour compresser la séquence, il suffit de la parcourir et de remplacer les bigrammes que l'on trouve par l'un des $2^k - n$ codes réservés. La compression consiste alors à émettre un code sur k bits, les codes de 0 à $n-1$ sont réservés pour les symboles simples et les codes de n à $2^k - 1$ représentent des bigrammes. On obtient une décompression extrêmement rapide, mais avec un ratio qui est contraint à demeurer sous la barre du 2 : 1 [133, 19].

D'autres méthodes un peu plus sophistiquées vont précalculer un dictionnaire, ordonner les mots selon leur rang d'occurence et assigner un code efficace aux index qui correspondent aux mots trouvés. Pike, par exemple, propose un schème où les symboles les plus fréquents tiennent sur 4 bits, et les mots les plus fréquents, les symboles un peu moins fréquents sur 8 et les symboles et mots plus rares sur 12 bits [167]. Cette approche est tout à fait justifiée si on considère la loi de Zipf, et les preuves empiriques qui démontent qu'en anglais du moins, les cents mots les plus utilisés comptent pour environ 50% des mots dans un texte [232, 233, 234, 7]. Tropper, quant à lui, propose une méthode similaire où les mots sont découpés en préfixes, radicaux et suffixes communs, en plus d'utiliser environ 900 mots choisis *a priori* et de façon statique [208]. La granularité des codes de Tropper est plus grande, ils sont sur 8 ou 16 bits.

Ces méthodes ont plusieurs défauts. Le premier d'entre eux, est qu'elles dépendent d'une connaissance *a priori* sur le langage à compresser. Si les dictionnaires sont choisis pour une langue, ces méthodes se planteront misérablement si on leur soumet un texte dans une autre langue, ou des données qui ne sont pas du texte. Toutefois, comme ces méthodes permettent de représenter tous les symboles, toutes les séquences sont admissibles, quoique possiblement au prix d'avoir une version compressée beaucoup plus longue que la version originale.

Nous consacrerons le reste de cette section aux algorithmes qui calculent incrémentalement leurs dictionnaires à partir de la séquence à compresser, en utilisant le moins possible de connaissances *a priori*. Ces algorithmes, à cause du peu d'hypothèses formulées quand à la nature des séquences à compresser, sont dits *universels*. Les algorithmes que nous présenterons jouissent aussi de propriétés théoriques intéressantes, en particulier celle de pouvoir compresser une séquence suffisamment longue avec un nombre de bits moyen par symbole arbitrairement près de l'entropie de la source qui a généré la séquence.

4.2.1.1 Lempel-Ziv 77

La technique LZ77 (d'après l'article de Ziv et Lempel de 1977 [235]) compresse la séquence en exploitant les répétitions de sous-séquences. L'algorithme parcourt la séquence de gauche à droite. La partie de gauche, déjà parcourue, est censée être déjà compressée. À droite, on a la partie qu'il reste à compresser. Le pointeur courant pointe au premier symbole de la partie non compressée. On va chercher dans la partie de gauche la plus longue sous-chaîne qui correspond au début de la partie non compressée. Lorsqu'une concordance est trouvée, on encode sa position (relative au pointeur courant), la longueur de la concordance et le premier symbole qui n'a pas concordé dans la partie de droite. Cela nous donne des tuples de la forme $\langle\, p, l, \sigma \,\rangle$, où p donne la position, l la longueur de la concordance et $\sigma \in \Sigma$, Σ étant l'alphabet utilisée par la séquence. Le processus est illustré à la fig. 4.13.

La décompression est considérablement plus simple. On lit les tuples $\langle\, p, l, \sigma \,\rangle$ un par un, et on ajoute à la séquence décompressée le segment désigné. Si s_1^t est la séquence décompressée, à

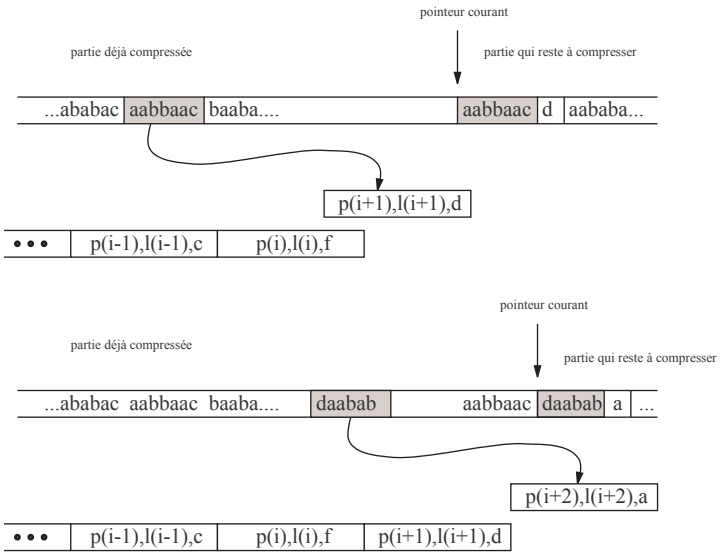


FIG. 4.13 – Démonstration de l'algoritme LZ77. Les valeurs $p(i)$ et $l(i)$ décrivent la position et la longueur de la i^{e} concordance trouvée.

la lecture du i^{e} tuple, $\langle\, p_i, l_i, \sigma_i \,\rangle$, la séquence deviendra $(s_1^t : s_{t-p_i}^{t-p_i+l_i-1} : \sigma_i)$.

La recherche de la plus longue concordance est une tâche ardue. Le problème de trouver la plus longue concordance entre le début d'une séquence de longueur m et une séquence de longueur n est $O(nm)$. On aura avantage à utiliser un algorithme de recherche de sous-chaînes rapide, par exemple des versions modifiées des algorithmes Boyer-Moore ou Knuth-Morris-Pratt [21, 113]. La solution la plus simple, c'est d'utiliser un algorithme brutal mais de limiter l'horizon de recherche. Plutôt que de commencer la recherche depuis le début de la séquence, nous nous contenterons de chercher sur, disons, les derniers 4096 symboles.

L'algorithme LZ77 a donné naissance à plusieurs variations :

– **LZR**. Cette variation cherche la séquence au complet, sans se soucier de l'horizon. Elle encode les tuples $\langle\, p, l, \sigma \,\rangle$, mais utilise des codes de longueur variable pour la longueur et la position. Cette variante est dûe à Rodeh, Pratt et Even [176].

– **LZSS**. Cette version utilise un bit pour indiquer si le code qui suit est un tuple de position et de longueur ou un symbole. Les tuples sont alors $\langle\, 0, p, l \,\rangle$ ou $\langle\, 1, \sigma \,\rangle$. La recherche est limitée par un horizon. La position et la longueur sont codés avec un code naturel de longueur fixe. Cette variation est dûe à T. C. Bell [10].

– **LZB**. Cette modification utilise le même système de bit indicateur que LZSS, à la différence que la position est codée grâce au code récursif d'Elias, $C_\gamma(\cdot)$ (voir la section 5.3.3.3 au sujet de ces codes). C'est aussi T. C. Bell qui présenta cet algorithme [11].

– **LZH**. Comme LZSS, sauf que les pointeurs sont codés grâce à une variante des codes de Huffman. Cette variante a été utilisée dans des programmes d'archivage [28].

– **LZRW-1**. La recherche exhaustive de la plus longue concordance peut être très dispendieuse, computationnellement parlant. La solution est de calculer des signatures pour les sous-séquences déjà observées. L'horizon de recherche est limité à 4096 symboles, et la plus longue concordance à 16 symboles. Les trois premiers symboles de la concordance à chercher sont convertis en une signature d'adressage dispersé (i.e., une clef de hashage). Dans la table à adressage dispersée, on maintient les pointeurs vers les concordances potentielles. On conserve aussi une liste inversée qui permet d'effacer les références qui pointeraient en dehors de l'horizon courant de recherche. Les données sont encodées comme pour LZSS, soit avec un bit qui détermine si on a affaire à une paire de position et de longueur ou à un symbole. Cette variation est dûe à Ross Williams [223, 222]. La fig. 4.14 illustre le procédé.

– **LZRW-4**. L'horizon de recherche et les concordances sont maintenant limitées à 1 mégaoctet. De plus, l'encodage des nouveaux symboles est aidé d'un code efficace qui découle d'une prédiction d'ordre 2. On modélise la fonction de probabilité du prochain symbole étant donné le dernier symbole de la concordance. Aussi dûe à Ross Williams [222]. Les algorithmes LZRWx ont été modifiés par la suite pour inclure des mécanismes de tables à adressage dispersé plus sophistiqués [179].

L'algorithme LZ77 possède la caractéristique d'encoder, pour une séquence suffisamment longue, les symboles en un nombre de bits moyen qui s'approche arbitrairement près de l'entropie de la source qui a produit la séquence [130, 182]. On suppose seulement que les positions sont encodés sur $\lceil \lg n \rceil$ bits pour une séquence de longueur n. La longueur moyenne des concordances en bits sur le nombre nécessaire de bits pour encoder la position et la longueur des concordances converge éventuellement vers l'entropie de la source.

4.2.1.2 Lempel-Ziv 78

La classe d'algorithmes LZ77 utilise un horizon (limité ou non) pour trouver les concordances qui aideront à la compression. Les algorithmes de la classe représentée par l'algorithme LZ78 vont utiliser des structures de données auxiliaires dans lesquelles seront explicitement maintenues toutes les sous-séquences possibles. Ces structures sont habituellement nommées « dictionnaires ». Le grand avantage de l'algorithme LZ78 est que les longueurs des sous-séquences est implicite car contenue dans le dictionnaire, éliminant ainsi le besoin de l'encoder explicitement dans les données compressées [236]. L'algorithme utilise non pas la position de la concordance, mais son index dans le dictionnaire. Si le dictionnaire comprend au maximum d entrées, l'index est simplement un nombre entre 0 et $d-1$.

Les structures de données qui sont utilisées pour les dictionnaires sont variées, mais l'une des plus commode, c'est la treille [74]. Une treille est montrée à la fig. 4.15. L'index d'une sous-séquence est donnée par le numéro de nœud correspondant dans la treille : la sous-séquence est formée par les symboles qui se trouvent sur le chemin entre la racine et le nœud indiqué par l'index. Considérons l'exemple de la fig. 4.15. L'index n^o 5 indique la séquence ab, l'index n^o 2, la séquence b, etc.

La séquence à compresser est parcourue de gauche à droite, et les concordances sont re-

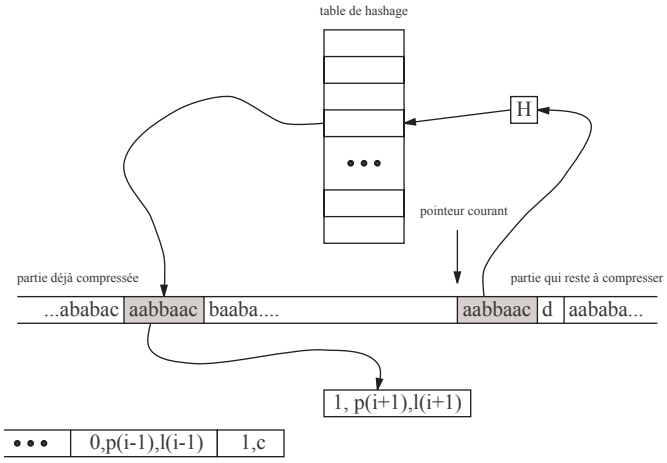


FIG. 4.14 – Démonstration de l'algoritme LZRW-1. Les valeurs $p(i)$ et $l(i)$ qui décrivent la position et la longueur de la i e concordance trouvée sont trouvées rapidement grâce à un mécanisme de table à adressage dispersée. La fonction de hashage est symbolisée par la boîte H .

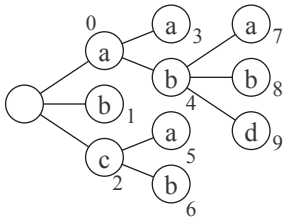


FIG. 4.15 – Les treilles sont de bonnes structures de données pour maintenir une liste de séquences et tous leurs préfixes. La treille qui est montrée ici maintient les séquences a, aa, ab, aba, abb, abd, b, et c. Les numéros des nœuds sont indiqués. Remarquez que le premier nœud n'est pas numéroté car il ne contient rien ; il s'agit de la racine qui correspond à $\perp$, la séquence vide.

cherchées dans le dictionnaire, D. Avec une structure de données comme la treille, la recherche devient très efficace, soit en temps linéaire en la longueur de la concordance. Pour trouver la concordance dans la treille, il suffit de commencer à la racine et aller au nœud indiqué par le prochain symbole à lire dans la séquence. On répète le processus et on arrive éventuellement à une feuille ou à un nœud dont aucun enfant ne correspond au prochain symbole de la séquence ; on a trouvé la concordance de longueur maximale. Dès qu'on a trouvé l'index i de la plus longue concordance entre la séquence à compresser et le dictionnaire, on émet un tuple $\langle\, i, \sigma \,\rangle$ où σ est le premier symbole qui ne concorde pas entre la séquence et le dictionnaire. Remarquons que la plus longue concordance peut être la séquence vide, $\perp$. Nous réservons un index pour la séquence vide, disons 0. La chaîne $(D_i : \sigma)$ est ajoutée au dictionnaire. Le dictionnaire peut croître indéfiniment, ou être limité à un nombre maximal d'entrées, d. Lorsque toutes les entrées dans le dictionnaire sont occupées, on procède à une remise à zéro et toutes les entrées sont détruites.

Les index sont codés avec un code binaire $C_{\beta(\lceil \lg |D| \rceil)}(i)$ (voir l'appendice A sur ces codes), qui utilise $\lceil \lg |D| \rceil$ bits, où $|D|$ représente le nombre d'entrées présentement utilisées dans le dictionnaire (ce qui n'est pas la même chose que d, le nombre d'entrées maximal). Le symbole est codé sur $\lceil \lg |\Sigma| \rceil$ bits. Comme pour LZ77, les variations de LZ78 sont nombreuses :

- **LZW**. Cette variante de Welch nous dispense de l'utilisation du symbole dans le tuple $\langle\, i, \sigma \,\rangle$ [221]. Supposons que nous parcourrions la séquence à la recherche de la plus longue concordance dans le dictionnaire. Après un certain nombre de symboles, nous trouvons une concordance dont l'index est i. Plutôt que d'émettre i et le symbole fautif σ, nous allons simplement émettre i et reprendre la recherche de concordance. Après un certain nombre de symboles, nous trouvons la plus longue concordance à l'index j. Nous savons que l'entrée D_j commence par σ, car c'est le symbole qui a mis fin à la concordance i. Puisque nous connaissons D_j, nous connaissons σ. La nouvelle séquence $(D_i : \sigma)$ est ajoutée au dictionnaire après l'émission de j. Le dictionnaire est remis à zéro lorsqu'il est plein. Le dictionnaire est initialisé avec chaque symbole de l'alphabet, éliminant ainsi le besoin d'avoir des codes réservés pour introduire de nouveaux symboles. Nous reviendrons dans le chapitre 9 sur cet algorithme dans le détail et nous y proposerons des modifications.

- **LZC**. Essentiellement pareil à LZW, sauf que le taux de compression obtenu est surveillé afin de déterminer quand on vide le dictionnaire, en plus de le remettre à zéro quand il est plein [206]

- **LZT**. Les index sont codés grâce à des codes *phase-in*, (voir la section 5.3.4.2 à ce sujet) et les séquences dans le dictionnaire sont remplacées par une méthode de type LRU (*least recently used*). La méthode de remplacement LRU assigne à chaque séquence un tampon qui contient le dernier moment d'utilisation, et ce tampon est comparé au temps présent. La séquence qui a le plus vieux tampon est remplacée par une nouvelle séquence, qui reçoit alors un tampon au temps présent. Cette variation a été proprosée par Tischer [207].

- **LZMW**. Essentiellement comme LZW, sauf que la séquence ajoutée au dictionnaire est $(D_i : D_j)$ plutôt que $(D_i : \sigma)$. Cette variante est dûe à Miller et Wegman [140].

- **LZJ**. Contraint la longueur maximale des séquences dans le dictionnaire (≈ 6 donnerait de bons résultats), et toutes les sous-séquences sont accessibles. LZJ a été introduit par

Jakobsson [104].

- **LZFG**. Il s'agit en fait d'une famille complète d'algorithmes, dûe à Fiala et Greene [70]. L'algorithme **A1** utilise un arbre PATRICIA (*Practical Algorithm To Retrieve Information Coded In Alphanumerics* [149]) plutôt qu'une table de hashage ou qu'une treille. La version **A2** utilise des codes $(start, step, stop)$ pour encoder les longueurs et les positions. La version **B1** modifie les données dans le PATRICIA de façon simplifiée, mais ne permet plus de tenir trace de toutes les concordances. La version **B2** utilise aussi les codes $(start, step, stop)$ plutôt qu'un code naturel pour les positions et les longueurs. La version **C2** utilise un encodage plus rafiné pour les pointeurs. Nous reviendrons sur les codes $(start, step, stop)$ à la section 5.3.5.2.

- **LZY**. Hidetoshi Yokoo propose plusieurs variations de l'algorithme LZ78 [231]. L'une d'entre elle consiste à encoder les tuples $\langle i, \sigma \rangle$. Une permet, outre d'avoir accès à tous les préfixes d'une séquence dans le dictionnaire (comme avec une treille) et d'avoir accès à certains suffixes. Cet algorithme est asymptotiquement optimal [112].

Nous reviendrons dans un chapitre ultérieur sur l'algorithme LZW tel qu'utilisé dans le protocole de compression sans perte d'image GIF, et comment il peut être modifié pour augmenter la compression des images. Nous verrons aussi, à la section 5.3.4.3, et au chapitre 9, quelles modifications à la façon de coder les index peuvent mener à de meilleurs taux de compression.

Outre la façon dont le dictionnaire est maintenu, les algorithmes de la classe LZ78 diffèrent des algorithmes de la classe LZ77 en ce qu'ils ne supposent plus la proximité spatiale des redondances dans la séquence. Les algorithmes LZ77 codent les redondances en émettant un pointeur relatif à la position courante et en l'encodant de façon efficace, et obtiennent ainsi de bons taux de compression. Les algorithmes de la classe LZ78, par l'utilisation du dictionnaire qui maintient les sous-séquences, n'exigent plus la localité des références. En effet, l'index d'une sous-séquence dépend uniquement de sa position dans la structure de données qui maintient le dictionnaire, et non plus de sa position originale dans la séquence.

Comme l'algorithme LZ77, l'algorithme LZ78 est asymptotiquement optimal. Une démonstration de l'optimalité de LZ78 est donnée par Savari [183]. La démonstration pour LZW est très semblable aux démonstrations pour LZ78, comparez par exemple celle de Louchard et Szpankowski avec celle de Savari [130, 182].

4.2.2 Les méthodes à base de transformations

Ces méthodes appliquent des transformations aux séquences de façon à mettre en évidence des redondances cachées. Une de ces méthodes, la méthode de Burrows-Wheeler calcule une permutation spéciale de la séquence qui augmente la chance de trouver des répétitions du même symbole dans la nouvelle séquence, ce qui en facilite potentiellement la compression [32]. D'autres méthodes, comme les méthodes aidées de prédicteurs, vont chercher à transformer la distribution des symboles de la séquence afin de les coder plus facilement. Par exemple, une ligne d'image est une suite de pixels. Si on fait l'hypothèse que les pixels qui se suivent se ressemblent, on peut s'attendre à une petite variation d'un pixel à l'autre. Plutôt que de coder les pixels eux-mêmes, on cherchera à coder la variation. C'est un exemple très simple de prédicteur où la « prédiction » dit que le prochain pixel sera le même que le précédent. Il existe heureusement des prédicteurs

plus sophistiqués pour les images, en particulier des méthodes qui tiennent aussi compte de la structure en deux dimensions de l'image, non seulement à l'échelle du pixel mais aussi à une échelle plus globale de l'image, où de plus grandes régions sont considérées.

4.2.2.1 Méthode Burrows-Wheeler

Supposons que nous ayons une séquence S de longueur n tirée de Σ^n, où $|\Sigma| = m$. Sans faire de suppositions particulières sur la séquence, nous pouvons compter le nombre d'occurences de chaque symboles de Σ. Soient ces comptes $n_1, n_2, \ldots, n_m$, tels que $n_i \in \mathbb{Z}^*$ et $\sum_{i=1}^{m} n_i = n$. Nous avons alors

$$\binom{n}{n_1\ n_2\ \cdots\ n_m} = \frac{n!}{n_1!\, n_2!\, \cdots\, n_m!}$$

permutations distinctes de cette séquence. Parmi celles-ci, se trouve une qui est entièrement constituées de répetitions de $\sigma_1, \sigma_2, \ldots, \sigma_m$ de longueur $n_1, n_2, \ldots, n_m$. Cette permutation est très simple à coder, car il suffit de coder $n_1, n_2, \ldots, n_m$, ce qui coûte $O(m \lg n)$ bits. En effet, pour $n \gg m$, $O(m \lg n) \ll O(n \lg m)$. Il faut aussi coder le numéro de la permutation, i, tel que $\pi(i)$ est la permutation des symboles qui donne la permutation originale, c'est-à-dire la séquence de départ. Le numéro de la permutation statisfait l'inégalité suivante :

$$0 \leqslant i < \binom{n}{n_1\ n_2\ \cdots\ n_m}$$

ce qui fait que même avec un code efficace (car toutes les permutations sont *a priori* équiprobables, on aura que la longueur de ce code est donnée par

$$| C(i \mid n, n_1, \ldots, n_m) | \approx \left(n + \frac{1}{2}\right) \lg n - \sum_{j=1}^{m} \left(n_j + \frac{1}{2}\right) \lg n_j$$

Cette approximation est démontrée à l'appendice C. Avec cette technique, le coût d'encoder i, n, m et $n_1, n_2, \ldots, n_m$ est

$$C(n, m, n_1, \ldots, n_m, i) \approx \lg n + \lg m + m \lg n + \left(n + \frac{1}{2}\right) \lg n - \sum_{j=1}^{m} \left(n_j + \frac{1}{2}\right) \lg n_j$$

ce qui est probablement effectivement plus court que l'encodage naïf de la séquence originale, pour peu que m soit relativement petit et les n_i relativement grands. Le problème avec cette technique, c'est la complexité du calcul de $i = \pi^{-1}(S)$, qui requiert la manipulation d'entiers de taille proportionnelle à la longueur de la séquence. Cependant, si nous nous restreignons à des permutations moins générales, nous pourrions nous éviter le coût du calcul de i et du code pour i étant donné n, m et $n_1, n_2, \ldots, n_m$, quoiqu'au coût de codage plus élevé de la permutation résultante.

Une telle méthode nous est donnée par la transformée Burrows-Wheeler, que nous devons à M. Burrows et D. J. Wheeler [32]. La transformée commence par générer $n - 1$ rotations cycliques de la séquence S. Les rotations sont alors mis dans une matrice $n \times n$, dont les rangées, considérées en tant que séquences, sont triées en ordre lexicographique croissant. La dernière colonne de la matrice exhibe plus de répétitions que la séquence originale, ce qui la rend plus facile à coder. La propriété la plus surprenante de cette transformée est qu'en sachant j, la rangée à la quelle se trouve la séquence originale après le tri, on peut reconstruire la séquence

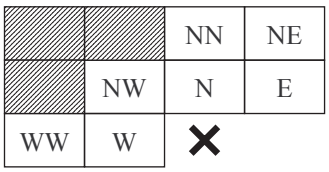


FIG. 4.16 – La région utilisée par le prédicteur CALIC. Les pixels sont nommés « cartographiquement » par rapport au pixel à prédire, indiqué par ✕.

originale en ne connaissant que la dernière colonne de la matrice !

Burrows et Wheeler proposent par la suite un simple encodage MTF (*move to front*) pour encoder la séquence. L'encodage MTF maintient une liste des symboles de l'alphabet rencontrés en ordre décroissant de rang . À chaque fois qu'un symbole est rencontré, on le place au début de la liste, et tous les autres reculent d'un rang. Pour encoder un symbole, on émet son rang courant puis on le déplace au début de la liste. On utilise un code simple, par exemple RLE ou de style Huffman (qui pourrait supposer une distribution de type Zipf) pour encoder les rangs.

Ce qui est surprenant, c'est que cet algorithme performe aussi bien que les autres algorithmes plus généraux des familles LZ7x, voire même aussi bien que les algorithmes de modélisation de contexte comme PPM (que nous présentons à la section 4.2.3.1).

4.2.2.2 Méthodes à base de prédicteurs

Les prédicteurs vont transformer une séquence d'éléments numériques de façon réversible. Le prédicteur le plus simple, c'est le prédicteur différentiel. Chaque élément de la séquence est codé non pas tel quel, mais par rapport à la différence à l'élément précédent dans la séquence. Si les éléments sont fortement corrélés, alors les différences seront en moyenne de faible magnitude, possiblement centrées sur zéro. L'hypothèse de correlation spatiale tient naturellement pour des séquences qui représentent des signaux analogiques comme le son et l'image, quoique pour ce dernier type nous ayons des séquences en deux dimensions.

Soient les x_i, $i = 0, \ldots, n-1$, les éléments de la séquence. La séquence différenciée est donnée par $\{x_0, x_1 - x_0, x_2 - x_1, \ldots, x_{n-1} - x_{n-2}\} = \{x_0, \delta_1, \delta_2, \ldots, \delta_{n-1}\}$. On parle des δ_i comme étant les *résidus*. On supposera par la suite que les $\delta_i \sim X$, pour une source aléatoire de type géométrique bilatérale (*two-sided geometric distribution*) ou normale $\mathcal{N}(0, \sigma)$. Avec une distribution centrée sur zéro, avec un faible écart-type (par hypothèse) il est possible de construire des codes efficaces pour encoder les résidus.

Wallace, dans sa proposition du standard JPEG, présente une série de prédicteurs en deux dimensions [218]. Ces prédicteurs sont énumérés au tableau 4.1. On teste successivement les prédicteurs sur toute l'image, en calculant combien de bits seraient nécessaires pour coder tous les résidus si on utilisait l'un ou l'autre des prédicteurs, et celui qui donne le meilleur taux de compression est choisi comme prédicteur pour l'image. Les résidus sont codés avec un code de Huffman précalculé.

D'autres prédicteurs sont plus complexes. On trouve parmi ceux-ci CALIC (*Context-based Adaptive Lossless Image Coding* et LOCO-I (*Low-Complexity Coder*) [227, 219]. Ces prédicteurs considèrent une région plus grande de l'image afin de mieux capturer le comportement de l'image

Code	Prédicteur
0)	$\hat{x}_{i,j} =$ littéral
1)	$\hat{x}_{i,j} = x_{i-1,j}$
2)	$\hat{x}_{i,j} = x_{i,j-1}$
3)	$\hat{x}_{i,j} = x_{i-1,j-1}$
4)	$\hat{x}_{i,j} = x_{i-1,j} + x_{i,j-1} - x_{i-1,j-1}$
5)	$\hat{x}_{i,j} = x_{i,j-1} + \frac{1}{2}(x_{i-1,j} - x_{i-1,j-1})$
6)	$\hat{x}_{i,j} = x_{i-1,j} + \frac{1}{2}(x_{i,j-1} - x_{i-1,j-1})$
7)	$\hat{x}_{i,j} = \frac{1}{2}(x_{i-1,j} + x_{i,j-1})$

TAB. 4.1 – Les prédicteurs du standard JPEG en mode sans perte. Les $x_{i,j}$ sont les pixels de l'image, $i = 0, \ldots, n-1$, $j = 0, \ldots, m-1$ pour une image $n \times m$. Les $\hat{x}_{i,j}$ sont les pixels prédits.

autour du pixel à prédire (voir la fig. 4.16 pour la région utilisée par CALIC). CALIC calcule plusieurs prédictions, à des échelles différentes de l'image. On commence par prédire une grille réduite à 16 : 1, (soit un pixel sur quatre dans les deux directions), puis la grille réduite à 4 : 1 (soit un pixel sur deux dans les deux directions), et finalement les pixels restant. LOCO-I présente une version simplifiée de CALIC mais donne aussi de très bons résultats. CALIC et LOCO-I demeurent les étalons contre lesquels tout nouveau prédicteur se compare.

4.2.3 Les méthodes à base de statistiques

Ces méthodes, plutôt que de transformer la séquence, vont modéliser la fonction de distribution de probabilité du prochain symbole. Pour chaque symbole σ_i de l'alphabet, ces méthodes vont produire une probabilité $P(X_t = \sigma_i \mid x_1^{t-1})$ telle que

$$\sum_{\sigma_i \in \Sigma} P(X_t = \sigma_i \mid x_1^{t-1}) = 1$$

possiblement en augmentant artificiellement l'alphabet pour y inclure des codes spéciaux destinés à transporter de l'information de contrôle. L'information de contrôle peut servir à informer le décodeur que l'on change de méthode de prédiction, que la séquence est terminée, ou quelqu'autre information du même genre. Nous verrons à la section 4.2.3.1 comment les prédicteurs PPM utilisent ces symboles spéciaux pour basculer d'un prédicteur à l'autre.

Lorsque la taille de l'alphabet est limitée et qu'il n'existe aucun rapport évident entre l'ordre des symboles et leurs fréquences relatives (comme c'est le cas pour l'alphabet latin pour lequel il n'existe aucune relation entre l'ordre lexicographique et la fréquence d'utilisation des lettres), la fonction de distribution de probabilité est une fonction de masse arbitraire. Si au contraire l'alphabet est très grand et qu'il existe une relation stricte entre l'ordre des symboles et la fonction de probabilité, une approximation continue par une loi simple (Poisson, gaussienne, exponentielle bilatérale, etc.) pourrait être utilisée. Les prédictions formulées en tant que fonction de distribution de probabilité (de masse ou de densité) sont alors fournies à un codeur efficace qui générera un code pour chaque symbole possible. Comme le codeur effiace assigne des codes courts aux symboles prédits comme étant les plus probables et des codes plus longs aux symboles prédits comme étant moins probables, on a intérêt à avoir une modélisa-

tion de la fonction de probabilité qui colle avec ce qui est effectivement observé dans la séquence.

Dans cette section, nous présenterons deux types de prédicteurs statistiques, PPM (*Prediction by Partial Match*) et les prédicteurs de type DMC (*Dynamic Markov Compression*). Il existe plusieurs autres types de prédicteurs, notamment les mixtures de prédicteurs. Une mixture de prédicteurs vient en deux variétés principales. La première variété comporte un superprédicteur qui assigne aux prédicteurs disponibles une probabilité indiquant leur susceptibilité à être le meilleur prédicteur. La prédiction se fait donc en deux étapes, soit la prédiction sur le choix du prédicteur, suivi de la prédiction faite par le prédicteur choisi. Cela permet d'avoir un code qui désigne d'abord le prédicteur suivi d'un code qui encode la prédiction du prédicteur ainsi désigné. Si la distribution sur les prédicteurs est très pointue, nous aurons des codes très courts pour le choix du prédicteur (soit un bit ou moins, selon la méthode de codage), et le meilleur prédicteur donnera aussi le code le plus court. Le choix peut se faire en minimisant simultanément les deux parties du code. Nous verrons que PPM constitue un exemple de cette première variété de mixtures.

La seconde variété de mixtures consiste à former une mixture avec les prédictions de chaque prédicteur en leur accordant un poids varié. On combine ainsi les prédictions selon la confiance que l'on accorde à chaque prédicteur. La prédiction prend alors une forme

$$P(X_t = \sigma_i) = \sum\nolimits_{p \,\in\, \text{prédicteurs}} \omega(p) P_p(X_t = \sigma_i \mid x_1^{t-1})$$

$P_p(\cdot)$ est la prédiction selon le prédicteur p et où $\omega(p)$, le poids du prédicteur p, peut être statique ou adaptatif. Si les poids sont adaptatifs, ce sera en fonction de la qualité des prédictions *a posteriori* des différents prédicteurs. Plus souvent un prédicteur donne le code le plus court, plus son poids sera élevé dans la mixture. Un prédicteur très mauvais se trouvera à toute fin pratique exclu de la mixture en recevant un poids près de zéro.

4.2.3.1 PPM : *Prediction by Partial Match*

On pourrait chercher à modéliser la séquence en utilisant systématiquement un contexte de longueur fixe w, et calculer $P(X_t = \sigma_i \mid x_{t-w}^{t-1})$ pour tous les symboles de l'alphabet, à chaque symbole de la séquence. Le problème avec cette approche, que nous avons d'ailleurs déjà mentionné, c'est que pour un w moindrement grand, le nombre de fois qu'un contexte en particulier est observé est toujours faible. On devrait alors limiter w, mais alors on se priverait des longs contextes qui rendent les prédictions presque déterministes.

Une méthode introduite par J. G. Cleary et I. H. Witten [52] consiste à utiliser une banque

de prédicteurs de la forme

$$\begin{aligned}
P_k(X_t = \sigma_i) &= P(X_t = \sigma_i \mid x_{t-k}^{t-1}) \\
P_{k-1}(X_t = \sigma_i) &= P(X_t = \sigma_i \mid x_{t-k+1}^{t-1}) \\
P_{k-2}(X_t = \sigma_i) &= P(X_t = \sigma_i \mid x_{t-k+2}^{t-1}) \\
&\vdots \\
P_2(X_t = \sigma_i) &= P(X_t = \sigma_i \mid x_{t-2}^{t-1}) \\
P_1(X_t = \sigma_i) &= P(X_t = \sigma_i \mid x_{t-1}) \\
P_0(X_t = \sigma_i) &= P(X_t = \sigma_i) \\
P_{-1}(X_t = \sigma_i) &= \frac{1}{\mid \Sigma \mid - \textit{observés}}
\end{aligned}$$

où *observés* représente le nombre de symboles distinct de Σ qui ont été observés jusqu'à l'étape $t-1$. Chaque prédicteur w se souvient de tous les contextes $(x_i^{i+w-1} : \sigma_j)$, pour $0 \leqslant i < t-w$ et $\sigma_j \in \Sigma$, qui ont été observés jusqu'à l'étape t. La prédiction pour $X_t = \sigma_j$ étant donné que $x_{t-w}^{t-1} = x_i^{i+w-1}$ est le nombre de fois que $(x_i^{i+w-1} : \sigma_j)$ a été observé divisé par le nombre total de fois que le contexte x_i^{i+w-1} a été observé. Cela nécessite donc, pour une séquence de longueur n, de maintenir au plus $\min(n-w+1, |\Sigma|^w)$ contextes, et chacun de ces contextes maintient $1+|\Sigma|$ compteurs.

La prédiction qui sera utilisée sera générée par le prédicteur d'ordre l ayant le plus long contexte x_i^{i-l+1} tel qu'on a effectivement observé $(x_i^{i-l+1} : x_t)$ au moins une fois. On suppose toujours que c'est le contexte le plus long, d'ordre k, mais s'il advient que c'est le contexte d'ordre $k-a$ qui est choisi, on devra émettre a fois le symbole spécial d'échappement (*escape code*) avec un code efficace qui correspond à la probabilité qu'on rencontre un symbole d'échappement aux prédicteurs $k, k-1, \ldots, k-a+1$. On augmente en fait l'alphabet de base pour inclure ce symbole spécial. Selon le contexte, on peut se trouver à descendre jusqu'au prédicteur d'ordre zéro où les symboles sont prédits inconditionnellement. Si le prochain symbole x_t de la séquence n'a pas encore été observé dans aucun contexte, nous descendront jusqu'au prédicteur d'ordre -1, où les symboles reçoivent une probabilité de $1/(|\Sigma| - \textit{observés})$, c'est à dire 1 sur le nombre de symboles que l'on a pas encore vus.

La façon d'estimer la probabilité du symbole d'échappement pour un contexte de longueur w diffère selon la variante de PPM qui est utilisée :

– **PPMA**. Dans la version originale, nous avons

$$P(\sigma_i \mid x_{t-w}^{t+1}) = \frac{C(x_{t-w}^{t+1} : \sigma_i)}{1 + C(x_{t-w}^{t+1})}$$

et

$$P(\textit{échap} \mid x_{t-w}^{t+1}) = \frac{1}{1 + C(x_{t-w}^{t+1})}$$

où $C(s)$ donne le nombre de fois que le contexte s a été observé jusqu'à l'étape $t-1$. La probabilité d'avoir le symbole d'échappement est toujours 1 sur le nombre de fois que ce

contexte a été observé. On gonfle artificiellement le compte total d'observations pour un contexte donné afin de garder une probabilité faible pour le code d'échappement.

– **PPMB**. Dans cette seconde version,

$$P(\sigma_i \mid x_{t-w}^{t+1}) = \frac{C(x_{t-w}^{t+1} : \sigma_i) - 1}{C(x_{t-w}^{t+1})}$$

et

$$P(\textit{échap} \mid x_{t-w}^{t+1}) = \frac{|\{\sigma_i \mid C(x_{t-w}^{t+1} : \sigma_i) \neq 0\}|}{C(x_{t-w}^{t+1})}$$

c'est-à-dire qu'on soustrait 1 à tous les comptes des contextes $(x_{t-w}^{t+1} : \sigma_i)$ qui ont été observés et que l'on donne au symbole d'échappement une fréquence égale au nombre de contextes distincts observés. On peut voir cela comme si on « volait » 1 à tous les contextes observés pour le donner au symbole d'échappement.

– **PPMC**. Cette version, proposée par Moffat [145], calcule plutôt :

$$P(\sigma_i \mid x_{t-w}^{t+1}) = \frac{C(\sigma_i : x_{t-w}^{t+1}) - 1}{C(x_{t-w}^{t+1}) + |\{\sigma_i \mid C(x_{t-w}^{t+1} : \sigma_i) \neq 0\}|}$$

et

$$P(\textit{échap} \mid x_{t-w}^{t+1}) = \frac{|\{\sigma_i \mid C(x_{t-w}^{t+1} : \sigma_i) \neq 0\}|}{C(x_{t-w}^{t+1}) + |\{\sigma_i \mid C(x_{t-w}^{t+1} : \sigma_i) \neq 0\}|}$$

Cette méthode donne une probabilité au code d'échappement qui est proportionnelle au nombre de symboles qui ont été observés. On aurait pu s'attendre à voir la méthode donner au code d'échappement une probabilité proportionnelle au nombre de symboles qui n'ont pas encore été observés dans ce contexte.

Il existe plusieurs autres variations (PPMD, PPM*, PPMZ, etc.) qui diffèrent de PPMA essentiellement par la façon dont la probabilité du symbole d'échappement est calculée et par la façon dont sont calculés les codes efficaces pour les prédictions.

Les méthodes PPM donnent de très bons résultats. PPMZ, que l'on doit à Charles Bloom, est d'ailleurs un des algorithmes de compression les plus performants. Le désavantage avec les algorithmes de cette famille, c'est qu'ils consomment d'autant plus de mémoire que la séquence est longue (car, rappelons nous, on a au plus $\min(n - w + 1, |\Sigma|^w) \times |\Sigma|$ compteurs à maintenir par prédicteur ayant un contexte de longueur w). Le code de chaque symbole étant composé de deux parties, à savoir la série de codes d'échappements suivie du code pour un symbole, doit aussi être très efficace. L'utilisation d'un code arithmétique est nécessaire au bon fonctionnement de l'algorithme PPM. Le codeur arithmétique, même très performant, consomme aussi des ressources et du temps. Une autre considération importante, c'est la longueur k du plus long contexte. Ici aussi, des expériences tendent à montrer que, pour l'anglais, $k = 5$ ou $k = 6$ donne les meilleurs résultats. Un contexte plus long nécessite plus de codes d'échappements et la compression est moins bonne ; un contexte plus court empêche d'exploiter les bonnes structures dans la langue et, par conséquent, la compression est moins bonne.

4.2.3.2 Automates enrichis et chaînes de Markov

Une autre façon de modéliser le contexte dans une séquence, c'est d'utiliser des automates enrichis. Les automates classiques savent « reconnaître » des séquences, mais n'ont pas les structures internes nécessaires pour prédire le prochain symbole étant donné les premiers symboles de la séquence. Nous présentons, dans cette sous-section, comment enrichir les automates pour qu'il soit possible d'obtenir des statistiques, et comment construire des automates adéquats pour une séquence donnée. Ces automates enrichis serviront à maintenir une chaîne de Markov de longueur variable qui modélise les transitions d'états dans la séquence, et où chaque transition « émet » un symbole.

Un automate classique est défini par un alphabet Σ, l'ensemble de ses états S, une fonction de transition $T : S \times \Sigma \to S$, un état initial $i \in S$ et un ensemble d'états acceptants $F \subseteq S$. L'état initial i est l'état dans lequel démarre l'automate. Les états acceptants, ou finaux, F, sont les états dans lequel l'automate peut trouver une fin de séquence. La fonction de transition prend l'état courant et le prochain symbole à être lu dans la séquence pour calculer un nouvel état. Cette transformation d'un état à l'autre est ce qui est appelé la transition.

Un état, dans un automate classique, ne contient aucune information supplémentaire. Pour un automate enrichi du type nécessaire pour la modélisation de Markov, un état doit contenir de l'information interne. Cette information est composée de $|\Sigma|$ compteurs, tous initialisés à zéro au moment où l'automate démarre. Chaque compteur est associé à un symbole de l'alphabet. Pour l'état $s \in S$, et $\sigma_i \in \Sigma$, ce compteur est noté $C_s(\sigma_i)$. À chaque fois que l'automate est dans l'état s et lit le symbole σ_i, le compteur $C_s(\sigma_i)$ est incrémenté.

Étant donné que l'automate est dans l'état s à l'étape t (après avoir lu $t-1$ symboles de la séquence x), la probabilité que le prochain symbole de la séquence soit σ_i est donné par

$$P_s(X_t = \sigma_i) = \frac{C_s(\sigma_i)}{\sum_{\sigma \in \Sigma} C_s(\sigma)} \tag{4.20}$$

La prédiction sert à calculer un code efficace, comme pour PPM. Le code pour x_t est émis, et on calcule le nouvel état par la fonction de transition, soit $s_{t+1} = T(s_t, x_t)$.

Cet algorithme est très simple et fonctionne bien pourvu que nous ayons choisi la taille de S et la fonction de transition (soit l'interconnexion entre les états) de façon appropriée. C'est, sauf pour des cas très simple, une tâche qui n'est pas triviale. Considérons la fig. 4.17, qui montre l'automate de Capon servant à modéliser les suites de zéros et de uns dans une image en noir et blanc[4]. Cet automate a deux états, l'état 0 qui correspond à l'état « être dans une suite de pixels noirs » et l'état 1 qui correspond à l'état « être dans une suite de pixels blancs ». Nous avons la fonction de transition qui dit quel est le nouvel état étant donné l'état courant et le pixel lu. Les compteurs internes (qui ne sont pas montrés sur la figure) vont aider à estimer les probabilités du prochain pixel étant donné l'état courant.

Dans ce cas très particulier, l'automate est simple car l'hypothèse sous-jacente est également simple : on suppose que l'on va rester dans l'état courant pour plusieurs pixels ; autrement dit, les pixels auront tendence à se répéter. Dans des cas plus généraux, on ne connaît pas d'avance la structure de la séquence et on ne sait pas comment produire un automate qui va aider à la

4 Conventionnellement, en infographie 0 est noir et 1 est blanc, alors qu'en imprimerie, c'est le contraire.

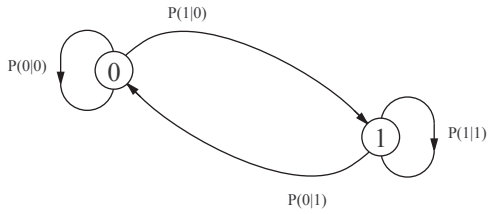


FIG. 4.17 – Un modèle prédiction par chaîne de Markov. Cet automate enrichi modélise les séquences de bits. On y calcule la probabilité qu'un 1 soit suivi d'un 1 ou d'un 0, et qu'un 0 soit suivit d'un 1 ou d'un 0. Ce modèle très simple correspond à deux variable aléatoire géométriques, l'une correspondant à l'état 0 où le paramètre $p = P(0 \mid 0)$, l'autre correspondant à l'état 1 où le paramètre $p = P(1 \mid 1)$.

compression. Une solution est de construire incrémentalement l'automate au fur et à mesure que l'on parcourera la séquence.

C'est la solution présentée par Teuhola et Raita [205]. L'algorithme sait comment fusionner des états qui sont semblables et dédoubler un état qui est rejoint par (au moins) deux contextes différents. Sans rentrer dans le détail, deux états seront fusionnés si leurs fonctions de transition sont identiques et que les prédictions formulées selon l'éq. 4.20 sont essentiellement identiques, soit $P_{s_1}(\sigma) \approx P_{s_2}(\sigma)$ pour tout $\sigma \in \Sigma$. Un état s sera dédoublé (ou cloné) si il existe deux états s_1 et s_2 qui mènent tous deux à s mais qui ont des estimations de probabilités très différentes. Cela suggère donc que l'état s n'est pas compatible simultanément avec les contextes s_1 et s_2. Si plus d'un chemin mène à un état s, ses prédictions sont alors formulées à partir d'un mélange de contextes selon s_1 et selon s_2, ce qui est potentiellement mauvais, puisque la fusion de contextes aura l'effet de brouiller les prédictions. L'état s est alors cloné pour donner un nouvel état s' et les transitions sont modifiés de façon à ce que s_1 mène à s et s_2 à s'.

Cet algorithme donne aussi de bons résultats, bien qu'il soit gourmant en mémoire. En effet, chaque état de l'automate utilise $O((c+s)|\Sigma|)$ bits de mémoire, où c représente le nombre de bits nécessaires pour un compteur et s le nombre de bits nécessaires pour indiquer le prochain état étant donné l'état courant et un symbole de Σ. Le nombre d'états croît aussi très rapidement en fonction de la facilité avec laquelle on clone les états et la complexité intrinsèque de la séquence. Le grand nombre d'états peut rendre la recherche d'états équivalents difficile, ce qui peut empêcher l'algorithme de réduire le nombre d'états par des fusions.

4.3 De l'importance du codage des entiers

Dans ce chapitre, nous avons passé en revue les grandes familles de méthodes de compression : les méthodes à base de décomposition de signaux suivies d'étapes de discrétisation, les méthodes à base de dictionnaire où la séquence est découpée syntaxiquement de façon à mettre en valeur les redondances, les méthodes à base de transformations où la séquence est transmuée en une nouvelle séquence qui exhibe plus de répétitions, les méthodes qui utilisent des prédicteurs pour transformer de façon réversible une séquence en une séquence dont les propriétés statistiques sont changées, et finalement les méthodes de prédictions statistiques qui modélisent la fonction

de distribution du prochain symbole dans la séquence.

Toutes les méthodes de décomposition que nous avons présenté (par la tranformée de Fourier, de Hartley, par les ondelettes) génèrent des coefficients qui sont discrétisés avant d'être codés efficacement. Les techniques de discrétisation très rapides sont habituellement brutales et ne tiennent pas compte des fréquence d'occurence des coefficients — sauf pour des hypothèses *a priori* simples — pour établir les régions de discrétisation. Par la suite, on doit calculer, souvent statiquement, des codes efficaces qui tiennent compte de la distribution des prototypes. Par exemple, dans le protocole JPEG, les coefficients entiers obtenus sont discrétisés de façon relativement crue puis sont codés grâce à un code de Huffman. À moins d'utiliser une discrétisation plus complexe (donc plus lente) il faut inévitablement avoir recours au codage des entiers pour le stockage des coefficients.

Les algorithmes de compression à base de dictionnaire nous demandent de stocker des tuples de formes variées, mais dont tous les éléments sont des entiers. L'algorithme LZ77 de base émet des tuples de la forme $\langle position, longueur, symbole \rangle$, tout comme LZR. Les algorithmes modifiés, comme LZSS, émettent deux sortes de tuples, un qui indique que le prochain code exprime un tuple de position et de longueur, $\langle 0, position, longueur \rangle$ ou un tuple $\langle 1, symbole \rangle$. Les algorithmes de la classe LZ78 utilisent aussi des codes pour les tuples et les entiers. L'algorithme de Ziv et Lempel utilise des tuples $\langle index, symbole \rangle$. LZW n'utilise que les index, et LZT les encode avec des codes *phase-in* (voir la section 5.3.4.2 pour ces codes). Les algorithmes de la famille LZFG proposés par Fiala et Greene, utilisent des tuples $\langle index, longueur \rangle$ où les indexes sont codés grâce à un code $(start, step, stop)$ sur lesquels nous reviendrons à la section 5.3.5.2.

Les algorithmes qui utilisent des prédicteurs pour transformer la séquence originale en une séquence plus facile à coder produisent des résidus qui seront distribués selon une loi géométrique bilatérale ou encore une loi gaussienne. Le fait que ces résidus soient (par hypothèse) distribués selon une loi unimodale à faible écart-type aidera grandement à l'obtention d'un code efficace pour ceux-ci.

Les prédicteurs statistiques, comme PPM et la modélisation dynamique de Markov produisent des fonctions de distribution de probabilité pour les symboles de l'alphabet de la séquence à compresser. Cette fonction de distribution sera utilisée pour l'obtention d'un code efficace pour les symboles. Les symboles étant essentiellement des entiers, nous nous retrouvons encore avec le problème de coder les entiers selon une fonction de distribution de probabilités sur les symboles.

❁ ❁ ❁

Toutes les méthodes présentées utilisent d'une façon ou d'une autre les entiers. Il apparaît alors primordial d'obtenir de très bons codes pour les entiers afin que la compression atteigne des ratios intéressants. La complexité de la génération du code doit rester faible pour que l'algorithme de compression ne soit pas indûment ralenti par le codage/décodage des entiers qu'il utilise. Un compresseur se doit d'être simultanément rapide et efficace en terme des ratios de compression atteints.

Vu l'importance du problème de codage des entiers, nous consacrerons deux chapitres à ce sujet. Le premier de ces chapitres, le chapitre 5, introduit les principaux concepts du codage des entiers, alors que le chapitre 6 présente l'algorithme de Huffman pour obtenir des codes optimaux étant donnée une fonction de distribution arbitraire, finalement, les chapitres 7 et 8 exposent nos contributions au sujet.

4.4 Connexions : apprentissage et compression

On préfère habituellement l'approche stochastique à l'approche algorithmique de la complexité. La complexité stochastique est plus abordable en ce sens qu'il est souvent possible de formuler des hypothèses probabilistes ou statistiques qui sont vraisemblables et raisonnables pour les classes de séquences qui nous intéressent. Bien qu'ultimement la détermination du meilleur modèle soit aussi un problème indécidable, il est souvent possible d'obtenir à partir de ces hypothèses de bonnes estimations de la complexité. La formulation probabiliste ou statistique d'un problème (à savoir, la prédiction du prochain symbole dans la séquence) implique que nous ayons un modèle qu'il nous est possible d'adapter en fonction des données observées.

Que signifie adapter le modèle ? Si nous utilisons une méthode paramétrique, soit une loi ayant un petit nombre de paramètres qui la déterminent complètement, on peut soit estimer les paramètres pour toute la séquence, soit remplacer progressivement les estimés de ses paramètres au fur et à mesure que la séquence est observée. Habituellement, pour les lois simples, nous disposons d'estimateurs pour les paramètres, les estimateurs découlant d'une méthode aux moindres carrés, de maximum de vraisemblance, etc. Si, au contraire, nous utilisons des modèles non paramétriques, c'est-à-dire des modèles qui sont très généraux quant à la forme de la distribution qu'ils sont capables de représenter, souvent au coût d'un très grand nombre de paramètres internes, nous aurons recours à des algorithmes plus sophistiqués, tels que la résolution de systèmes linéaires, la descente de gradient stochastique, le recuit simulé, etc.

Or, une discipline où foisonnent les méthodes non paramétriques, c'est l'apprentissage automatique, ou *machine learning*. Le but principal de l'apprentissage automatique, c'est de permettre à la machine d'apprendre à accomplir une tâche par elle-même, par exemple reconnaître des caractères manuscrits, prédire la prochaine valeur dans une séquence, etc. L'habileté qu'ont les ordinateurs à gérer de très grandes quantités de données et à effectuer rapidement une masse impressionnante de calculs permet l'utilisation de méthodes nouvelles où la modélisation des données repose sur un très grand nombre de lois, régles et paramètres, qu'il serait impossible à gérer sans ordinateur. Une méthode non paramétrique comme les réseaux de neurones peut exiger l'utilisation de plusieurs milliers de paramètres pour mener à bien une tâche qui nous paraît simple mais que nous ne sommes pas en mesure de bien expliquer algorithmiquement, comme par exemple, la reconnaissance de caractères manuscrits.

Ces algorithmes non paramétriques permettent d'apprendre, c'est-à-dire d'ajuster leur modèle interne de façon à fournir la réponse adéquate lorsqu'on leur soumet un exemplaire du problème qu'ils sont censés résoudre. Le problème peut être de classification ou de reconnaissance, où l'enjeu est d'assigner à un exemplaire du problème une classe prédeterminée. L'exemple de reconnaissance de lettres et chiffres manuscrits représentent un problème de classification : il s'agit de classer les images dans les classes qui correspondent aux chiffres et aux lettres. Le problème peut être de prédire, étant donné l'exemplaire fourni, quel sera le prochain symbole dans

la séquence, ou encode estimer un paramètre qui décrit le comportement de la séquence dans le futur. Un exemple de tâche de prédiction (très difficile!) serait de prédire le cours de la bourse pour certain titres étant donnés les indicateurs financiers. On voit qu'il n'existe pas forcément de relation directe entre le type de données qui sont utilisées pour formuler la prédiction et la prédiction elle-même.

L'apprentissage, s'il est supervisé, nécessite que l'on fournisse au modèle un exemplaire du problème pour lequel on connaît une réponse désirée. La réponse du modèle est comparée à la réponse désirée pour obtenir une fonction d'erreur, laquelle servira, à travers l'algorithme d'apprentissage, à modifier les (potentiellement très nombreux) paramètres internes du modèle. Beaucoup d'algorithmes sont basés sur la méthode de descente de gradient, où l'erreur à la sortie est propagée dans chacune des équations qui participent au résultat selon les contributions respectives de celles-ci, de façon à pouvoir ajuster les paramètres qu'on y trouve. Dans le cas de réseaux de neurones, l'algorithme de descente de gradient est connu sous le nom de rétropropagation de l'erreur (*backpropagation*), car les erreurs se propagent dans le réseau en sens contraire à la connectivité normale [92]. Nous n'entrerons pas dans les spécificités des réseaux de neurones ; cela dépasse le cadre du présent exposé. Les fonctions d'erreur entre la réponse obtenue et la réponse désirée dépendent de l'application exacte, bien qu'en général nous retrouvions des mesures d'erreur de type moindre carrés, choisies de façon à bien se prêter à un traitement analytique.

L'apprentissage supervisé nécessite donc que l'on dispose des exemplaires du problème et des réponses désirées pour chacun. Habituellement on divise ces données en deux ensembles, le premier, généralement beaucoup plus grand que le deuxième, est l'ensemble des données d'apprentissage que l'on nommera souvent ensemble d'entraînement, sur lequelles nous allons adapter le modèle. Le second ensemble de données, c'est l'ensemble des données de test sur lequelles nous allons vérifier que le modèle performe bien. Il est nécessaire de scindre les données en deux ensembles car, si la puissance du modèle est trop grande, il pourra apprendre « par cœur » tous les exemples et cependant être très mauvais sur de nouveaux exemples. Si le modèle n'est pas assez puissant, il sera mauvais sur tous les exemples qui lui seront fournis. Il faut chercher à atteindre une puissance adéquate, un équilibre entre la capacité d'apprendre par cœur les exemples, ce que l'on nommera spécialisation, et la capacité d'extrapoler les réponses pour les exemplaires qu'il n'aura pas vu, ce que l'on nommera généralisation. L'ensemble de test nous permettra justement de jauger la capacité de généralisation de la méthode, soit en pourcentage de classifications correctes, soit selon une mesure d'erreur approriée au problème, que ce soit des décibels, des moindres carrés, etc.

On aura compris les similitudes entre l'apprentissage supervisé, en particulier la tâche de prédiction, et la compression de données. Dans les méthodes à bases de prédicteurs statistiques que nous avons présentés à la section 4.2.3, nous avons vu des algorithmes qui utilisent des méthodes non paramétriques comme les chaînes de Markov dynamique et la modélisation PPM pour prédire le prochain symbole dans la séquence, et ces méthodes sont assorties d'algorithmes qui permettent de mettre à jour leur état interne pour tenir compte du symbole observé par rapport au symbole prédit. Cela correspond au cas de l'apprentissage supervisé car nous avons encore deux ensembles de données, celle d'entraînement qui correspondent à la partie observée de la séquence, et l'ensemble de test, qui correspond au prochain symbole dans la séquence. Si l'ensemble d'entraînement est représentatif des données et que la capacité du modèle est adéquate, nous auront une bonne prédiction moyenne, donc une bonne généralisation.

Souvent, on voit les algorithmes de compression démarrer à froid, utilisant très peu de suppositions et de connaissances *a priori* sur la séquence, bâtissant leurs modèles au fur et à mesure que la séquence est parcourue. Cela résulte en une certaine quantité de bits gaspillés durant la période d'adaptation de l'algorithme. Or, plutôt que de payer le prix de ces bits pour construire le modèle au fur et à mesure, on peut considérer démarrer le modèle avec une approximation satisfaisante, mais prenant peu de bits à coder, des paramètres du modèle, puis de lancer l'algorithme adaptatif sur la séquence. Si on sauve plus de bits dans le codage de la séquence qu'il en est nécessaire à l'encodage des paramètres du modèle, on sera gagnant. Il s'agit de trouver un équilibre entre le nombre de bits nécessaires à la description (même approximative) du modèle et le nombre de bits sauvés par cette description des paramètres du modèle.

On retrouve aussi l'apprentissage dit non supervisé en compression. Dans le cadre de l'apprentissage non supervisé, l'élève tente de construire un modèle à partir de règles simples, sans l'intervention d'un oracle ou d'un professeur qui lui puisse lui fournir les réponses désirées. On cherchera, par exemple, à estimer les modes d'une distribution, ou à établir un arbre taxomique grâce à une mesure de similitude qui est établie *a priori*.

Par exemple, pour le problème de *clustering*, on cherchera à agglutiner ensemble les exemples qui sont près les uns des autres selon notre métrique et à assigner aux exemples suffissamment différents des agglomérats distincts. Ce problème ressemble évidemment beaucoup à la discrétisation de vecteurs, où les vecteurs étant semblables sous la métrique se retrouvent dans le même agglomérat et partagent le même prototype. Nous avons présenté le problème à la section 4.1.2.2. On pourrait aussi considérer le cas du clustering hiérarchique, qui correspond au problème de la construction d'un arbre taxonomique. Un arbre taxonomique est un arbre où les feuilles qui sont supportées par les mêmes branches sont semblables, d'autant plus semblables que la branche commune est proche des feuilles. On voit les arbres tanoxomiques appliqués aux espèces animales dans de nombreux livres de biologie.

Le clustering (hiérarchique ou non) peut tenir compte de la distribution des exemples pour construire le modèle. Dans le cas de la discrétisation de vecteurs, le calcul des centroïdes tient compte des probabilités d'occurence implicitement en calculant la moyenne des vecteurs qui se trouvent dans la région convexe de l'espace que l'on associe au prototype. De même, dans le clustering hiérarchique, il est possible de calculer les nœuds internes de l'arbre en tenant compte des fréquences des individus qui se trouvent sur les feuilles, en utilisant le même calcul que pour les centroïdes du clustring non hiérarchique.

❁ ❁ ❁

Il existe donc des liens plus étroits qu'il n'y paraît entre la compression de données vue sous l'angle stochastique et le domaine de l'optimisation en général, et de l'apprentissage automatique en particulier. Alors qu'il est plutôt facile de voir le lien entre les méthodes de compression à base de chaînes de Markov ou encore de modélisation de contexte de style PPM et les algorithmes d'apprentissage basés sur des méthodes non paramétriques, et que la discrétisation est très fortement liée aux problèmes de clustering de l'apprentissage non supervisé, il est moins évident de voir comment certaines autres méthodes de compression, par exemple les méthodes à base de dictionnaire, correspondent aussi à des méthodes non paramètriques, où le modèle est représenté

par le dictionnaires et où la prédiction du modèle consiste à supposer que le prochain bout de séquence se trouve déjà dans le dictionnaire et l'apprentissage, soit la correction du modèle suite à une erreur, c'est simplement l'ajout d'une entrée dans le dictionnaire.

4.5 Notes bibliographiques

Les ouvrages entièrement dédiés aux séries et aux transformées de Fourier et leur applications ne manquent pas. Notons seulement le livre *Fourier Analysis* de T. W. Körner [122] et le livre *Fourier Series and Orthogonal Functions* de Harry Davis [61]. Le livre *Théorie Analytique de la Chaleur* de Fourier peut, de nos jours, être trouvé en réédition [72]. Bien que Cooley et Tukey soient reconnus comme étant les pères de la transformée rapide de Fourier [55], le fait que le lemme de Danielson-Lanczos précède chronologiquement la publication de 1965, nous laisse à penser. Il y a en effet des références à des articles (en Russe !) qui précéderaient l'article de Cooley et Tukey d'une décennie !

Je suis venu en contact avec la transformée de Hartley, par hasard, pendant mon séjour à AT&T Research, Holmdel, à l'automne 1995. C'est en cherchant dans leur très complète bibliothèque que je suis tombé sur l'article paru dans *Byte*, intitulé *Faster than Fast Fourier*, par Mark O'neill [155]. Le titre seul est suffisant pour stimuler l'imagination. La référence définitive en ce qui a trait à la transformée de Hartley est sans contredit Norman R. Bracewell qui a écrit de nombreux articles et même une monographie sur le sujet [22, 23, 24, 25]. On retrouve dans la littérature des versions de la transformée de Hartley où $\mathrm{cas}(x) = \cos(x) - \sin(x)$, ce qui n'a par ailleurs aucune incidence réelle sur la nature même de la transformée, car même la relation $\mathfrak{H}_f(g) = \mathrm{Re}\,\mathfrak{F}_f(g) + \mathrm{Im}\,\mathfrak{F}_f(g)$ devient simplement $\mathfrak{H}_f(g) = \mathrm{Re}\,\mathfrak{F}_f(g) - \mathrm{Im}\,\mathfrak{F}_f(g)$. Le lecteur est invité à lire les différents articles traitant de la transformée de Hartley et ses applications [123, 65, 214, 215, 158].

Rao et Yip présentent une monographie sur la transformée discrète de cosinus [173]. Les applications de celle-ci aux protocoles JPEG et MPEG peuvent être trouvées dans les deux ouvrages de Mitchell, Pennebaker *et al*, *JPEG Still Image Data Compression Standard* et *MPEG Video Compression Standard* [157, 141]. Ces deux ouvrages décrivent JPEG et MPEG avec suffisamment de détails pour mener à une implémentation de l'un ou l'autre protocole.

Les ouvrages sur les ondelettes ne manquent pas. Les livres de Stéphane Mallat *A Wavelet Tour of Signal Processing* et d'Ingrid Daubechies *Ten Lectures on Wavelets* sont probablement les meilleures références sur le sujet [132, 60]. *Wavelets : Theory and Application* d'Yves Meyer constitue une bonne introduction au sujet [138]. Le champ de recherche étant très actif, le lecteur intéressé surveillera les différents journaux de traitement de signal où sont publiés la majorité des développements.

Les ouvrages qui donnent de l'information quantitative sur les propriétés de la perception sont quand même assez rares. On peut trouver des courbes de réponse aux fréquences en fonction du nombre de décibels dans *Music, Physics and Engineering*, de Harry Olson [154]. Ce livre survole les différents média de reproduction (radio, télévision, bande magnétique, etc.) et est une bonne source d'information sur les phénomènes liés à la perception du son. Quant à la perception des couleurs, qui elles aussi sont sujètes à la discrétisation, le lecteur peut se référer aux travaux de Munsell sur la couleur, au livre *Psychologie de la perception* de André Delorme

[62], au chapitre 7, ou encore à la littérature colorimétrique, bien que celle-ci s'intéresse plus souvent au phénomène physique qu'à la perception de la couleur.

Pour la discrétisation, la monographie de Siegfried Graf et Harald Luschgy, *Foundations of Quantization for Probability Distributions* est une bonne bonne référence, quoique de niveau avancé [86]. Un autre survol du champ est présenté dans la publication DIMACS #14, *Coding and Quantization*, publié par l'*American Mathematical Society* [34]. Les algorithmes Lloyd-Max et Linde-Buzo-Gray sont décrits dans le livre de Khalid Sayood, *Introduction to Data Compression*, dont j'ai eu le plaisir de réviser le manuscrit dans son entier [184, p. 241+]. Les discrétisations structurées en arbre sont décrites au chapitre 3 de *Classification And Regression Trees* (souvent référé simplement comme *CART* dans la littérature) de Breiman, Freidman, Olshen et Stone, qui demeure la référence définitive sur tout ce qui est arbre de classification.

Les algorithmes des classes LZ7x ont suscité beaucoup d'enthousiasme. Plusieurs archiveurs commerciaux ou gratuits utilisent l'une ou l'autre des variations présentés aux sections 4.2.1.1 et 4.2.1.2. La meilleure façon de trouver les concordances n'est pas toujours d'utiliser la plus longue concordance trouvée. Lempel et Gavish explorent des fonctions de longueur de concordance alternatives [126]. L'idée étant de maximiser le taux de compression, on choisira la concordance qui maximise le ratio longueur de concordance *vs* le nombre de bits nécessaires pour coder cette concordance. On peut encore se contenter de trouver des concordances approximatives, pourvu que les corrections ne coûtent pas trop cher à coder. En effet, une longue concordance avec quelques erreurs pourrait donner un meilleur taux de compression, même en comptant l'information nécessaire pour corriger ces erreur [180]. Une comparaison des divers compresseurs peut être trouvée dans [83] et [12].

La transformée de Burrows-Wheeler a aussi suscité beaucoup de recherche. Il y a un grand nombre d'articles qui présentent des modifications à l'algorithme qui construit la matrice ou à l'algorithme qui la trie [178, 187]. Il y a aussi une version tolérante aux erreurs, qui fut présentée à la conférence DCC 2001 par Lee Butterman et Nasir Memon [33].

Il existe plusieurs prédicteurs pour les images et les autres signaux. Pour les images, le lecteur consultera les articles de Memon et Wu, *Recent Development in Context-Based Predictive Techniques for Lossless Image compression* et de Memon, Wu et Sayood, *A Context-Based Adaptive Lossless Coding Scheme for Continuous Tone Images* [137, 228]. On trouve aussi un certain nombre de prédicteurs dans le livre de Gibson *et al*, *Digital Compression for Multimedia* [83]. Pour les algorithmes PPM, la meilleure implémentation est probablement celle de Charles Bloom, PPMZ, elle-même une variation de PPM*, dûe à Cleary et Teahan [51].

Cormack et Horspool ont présenté la compression de Markov dynamique dans le cas spécifique d'une source aléatoire binaire [56]. Teuhola et Raita ont généralisé leurs algorithmes de façon à gérer les sources ayant un nombre quelconque de symboles [205]. Il n'y avait pas de raison *a priori* de limiter les algorithmes à des sources binaires, sauf peut-être la quantité de mémoire nécessaire pour représenter l'automate. Dans l'article de Teuhola et Raita, on trouve un tableau comparatif du nombre d'états en fonction de la longueur de contexte maximale considérée ; on voit que le nombre d'états est très élevé, même pour des longueurs de contextes modérées.

Chapitre 5

Le codage des entiers

5.1 introduction

Dans le chapitre précédent, nous avons mis en évidence la nécessité du codage des entiers dans diverses méthodes de compression. Ce chapitre présente quelques techniques d'encodage des entiers. Quelques unes des méthodes utilisées sont des méthodes *ad hoc*, construites pour produire une longueur de code moyenne réduite, mais sans vraiment tenir compte de la nature intrinsèque de la distribution qui génère les entiers. D'autres méthodes font peu de suppositions sur la distribution des entiers sauf, par exemple, dans le cas des codes dits universels, qu'elle est éventuellement non-croissante et différente de zéro pour tout $n \in \mathbb{N}$. D'autres codes construits systématiquement supposeront que nous avons plutôt affaire à un sous-ensemble des entiers, typiquement $\mathbb{Z}_N$, sans pour autant vraiment tenir compte de la distribution, sinon par un certain laxisme sur les hypothèses. Enfin, il y a les codes efficaces qui tiennent tout à fait compte de la fonction de distribution (du moins, d'une approximation car il est rare que la fonction de distribution réelle soit connue) des entiers pour générer un code efficace qui s'approche de l'entropie de la source.

Les codes en général existent en quatre familles : de longueur fixe vers longueur fixe, de longueur fixe vers longueur variable, de longueur variable vers longueur variable et de longueur variable à longueur fixe. Les codes de longueur fixe vers longueur fixe prennent un nombre fixe de symboles d'entrée et produisent un nombre fixe de symboles en sortie. Ce type de code ne fournit habituellement de la compression qu'au coût d'une perte dans le nombre de valeurs distinctes récupérées à la décompression. Les codes de longueur fixe à longueur variable prennent un nombre fixe de symboles d'entrée et les représentent grâce à un nombre variable de symboles de sorties. Les codes que nous présentons pour les entiers dans ce chapitre correspondent à cette catégorie. Les codes de longueur variable à longueur variable prennent un nombre variable de symboles d'entrée et produisent un nombre variable de symboles de sortie. Un compresseur de type LZ77 avec un encodage à longueur variable des positions et des longueurs correspondrait à cette classe. Finalement, les codeurs de longueur variable vers longueur fixe prendront des séquences de longueur variables et leur associeront des codes de longueur fixe. Un exemple de ces codes, ce sont les codes de Tunstall qui associent à des séquences de longueur variable un code sur un nombre fixe de bits (voir section 6.3.3, p. 128).

Les codes de longueur fixe vers longueur variable créent généralement des codes dont la lon-

gueur est entière, en terme du nombre de symboles de sortie par code. Cela permet de ne pas avoir à tenir compte du code suivant dans la séquence compressée pour décoder le prochain symbole. Les codes de longueur variable à longueur fixe réussissent à assigner un nombre fractionnaire de bits aux symboles. La notion de « fraction de bits » semble contre-intuitive mais s'explique aisément. Supposons que nous ayons une longueur fixe des codes en sortie de 16 bits — ce qui correspond à une limite naturelle pour un ordinateur. Par un algorithme quelconque, disons que nous pouvons trouver un code d'exactement 16 bits pour une séquence de 5 symboles. Nous avons donc $16/5 = 3\frac{1}{5}$ bits par symboles ! Il est donc facile d'obtenir un nombre fractionnaire de bits pour un code ; les codeurs arithmétiques sont toutefois plus astucieux et utilisent la distribution des données pour générer leurs codes efficaces. Le plus connu de ces codeurs, c'est le codeur arithmétique *Q Coder*, dont il existe plusieurs variantes, comme le *QM coder* et le Z-coder [20].

Nous n'aborderont pas les techniques du codage arithmétique, puisque la présente thèse ne propose pas d'améliorations aux diverses techniques, ni à la modélisation des probabilités qui convient à l'utilisation d'un codeur arithmétique. À ce sujet, le lecteur consultera la section 5.4, les notes bibliographiques.

❁ ❁ ❁

Nous commencerons par présenter le théorème Kraft-McMillan qui établit les conditions nécessaires à la lecture non-ambiguë des codes, et discourerons brièvement de considérations générales quant à la nature et à longueur des codes. Nous présenterons par la suite des codes *ad hoc*, quelques codes universels, des codes à domaines restreints et finalement des codes qui tiennent compte de la fonction de distribution des entiers.

5.2 Considérations sur les codes

Dans le contexte bien particulier de la compression de données, ce que l'on veut, c'est minimiser la longueur moyenne des codes, afin d'obtenir les meilleurs taux de compression. Nous voulons aussi que les codes soient uniquement décodables, c'est-à-dire qu'une séquence de bits formant une représentation compressée d'un message ne puisse pas être lue de plusieurs façon différentes. Une condition nécessaire et suffisante pour une lecture non-ambiguë des codes est donnée par les théorèmes de Kraft et McMillan, que nous couvrons à la section 5.2.2. Nous présentons d'abord les méthodes de calcul des longueurs moyennes des codes, qui reviendront tout au long du chapitre.

5.2.1 Longueur moyenne des codes

La longueur moyenne, indépendamment de la forme du code, dépend de la fonction de distribution des entiers à coder. Soit X, une source aléatoire qui émet des entiers tirés de D, le domaine de X. Notons que D n'a pas à satisfaire de propriétés particulières, sauf être récursivement énumérable. Typiquement, $D \subseteq \mathbb{N}$ ou $D \subseteq \mathbb{Z}$, au pire $D \subseteq \mathbb{Q}$ [1]. Soit $C_i(x)$, le code

1 Ce qui nous ramène à la notion d'ensemble récursivement énumérable. Tout ensemble de cardinalité $\aleph_0$ peut être mis en bijection avec les naturels, ce qui est une condition suffisante pour être récursivement énumérable, peut être considéré comme un domaine acceptable.

considéré pour $x \in D$, et $L_{C_i}(x \mid X) = \mid C_i(x \mid X) \mid$, la longueur du code pour x étant donné la variable aléatoire X, que nous ne connaissons pas nécessairement. La longueur moyenne du code C_i est alors donnée par :

$$\bar{L}_{C_i}(X) = \mathrm{E}[L_{C_i}(X)] = \sum\nolimits_{x \in D} P(X = x) L_{C_i}(x \mid X) \tag{5.1}$$

qui n'est pas sans rappeler la formulation de l'entropie, l'éq. (3.1), vue à la page 37, section 3.3 :

$$\mathcal{H}(X) = -K \sum\nolimits_{x} P(X = x) \log_b P(X = x)$$

Cette formulation de la longueur moyenne du code est donc naturelle dans la mesure où elle correspond à la formulation de l'entropie ; et il sera tout aussi naturel de comparer le résultat de l'éq. (5.1) au résultat de l'éq. (3.1) pour le code sous examen afin d'en estimer la performance. Nous verrons que cette formulation nous permettra de découler des méthodes d'optimisation pour certains codes.

La précédente formulation est pertinente lorsque le code C_i assigne à chaque entier un code distinct. Parfois, par contre, nous aurons un code qui assigne des classes d'entiers à des classes de codes, où tous les entiers appartenant à une même plage reçoivent des codes qui ont une même longueur. La formulation de la longueur moyenne devient alors

$$\bar{L}_{C_i}(X) = \sum\nolimits_{k} P(l(k) \leqslant X < h(k)) L_{C_i}(x \mid k, X) \tag{5.2}$$

où $l(k)$ et $h(k)$ sont les fonctions indiquant les bornes inférieures et supérieures du k^{e} intervalle, et en général nous avons $h(k) = l(k + 1)$. Cette seconde formulation nous sera aussi utile pour découler des méthodes d'optimisation.

5.2.2 Unicité de décodage et théorèmes Kraft-McMillan

Bien qu'en général, dans la littérature, on lise souvent *le* théorème Kraft-McMillan, il s'agit en fait d'un groupe de trois théorèmes. Ils sont dûs à L. G. Kraft et B. McMillan [121, 136]. Ces théorèmes fournissent des conditions nécessaires et suffisantes pour s'assurer qu'un code donné soit décodable de façon non-ambiguë.

Théorème 5.2.1 Théorème de Kraft (1). Soit C, un ensemble de n codes de longueurs $l_1, l_2, \ldots, l_n$ exprimés en base b. Si l'ensemble de codes C est instantanément décodable (un code est instantanément décodable si, aussitôt le dernier symbole du code lu, on reconnaît le code) alors les longueurs satisfont

$$\sum\nolimits_{k=1}^{n} b^{-l_k} \leqslant 1 \tag{5.3}$$

Théorème 5.2.2 Théorème de Kraft (2). Si $l_1, l_2, \ldots, l_n$ et b sont tels que l'inégalité de l'éq. (5.3) soit satisfaite, alors il existe un ensemble de codes C exprimés en base b qui est uniquement décodable. ■

La preuve du précédant théorème repose sur la construction explicite d'un code uniquemenet décodable qui satisfait obligatoirement l'inégalité de Kraft. Le théorème de McMillan est pour ainsi dire complémentaire aux deux théorèmes de Kraft. Le théorème de McMillan stipule que tout code uniquement décodable respecte nécessairement l'inégalité de Kraft.

Théorème 5.2.3 Théorème de McMillan. Si C est un ensemble de codes uniquement décodables, alors les longueurs de codes $l_1, l_2, \ldots, l_n$ satisfont obligatoirement l'inégalité de Kraft. ■

L'importance des théorèmes Kraft-McMillan n'est pas à être sous-estimée. Ces théorèmes définissent des moyens pour tester la décodabilité unique d'un ensemble de codes en n'en connaissant que les longueurs ; de même, cela permet de déterminer s'il existe en effet un ensemble de longueurs qui satisfasse l'inégalité. Puisque l'inégalité de Kraft est une condition *nécessaire et suffisante* à la décodabilité unique, il suffit de la vérifier pour déterminer la validité de la structure du code sous examen.

5.3 Le codage des entiers

Nous présentons, dans cette section, quelques codes simples pour les entiers. Certains sont *ad hoc* et sont utilisés dans diverses implémentations de protocoles de communication ou de compression, et ne tiennent que vaguement compte de la distribution des nombres. Nous présentons aussi les codes dits universels, qui permettent de coder des nombres arbitrairement grands et qui sont asymptotiquement optimaux pour peu que la distribution soit éventuellement non-croissante. Nous verrons aussi quelques codes spéciaux, en particulier des codes pour des intervalles finis et les codes énumératifs, qui permettent d'encoder un choix possible parmi $\binom{n}{m}$. Nous présentons aussi des codes statistiques simples qui ont des paramètres qui sont optimisés en tenant compte de la fonction de distribution des nombres. Enfin, nous discuterons des fonctions de pairages et des fonctions de repli qui permettent de replier $\mathbb{Z}$ sur $\mathbb{N}$.

5.3.1 Un peu de notation

Avant de s'attaquer aux diverses méthodes de codage des entiers, présentons notre notation. Considérons les schémas suivants :

x_3	x_2	x_1	x_0	1	0	y_4	y_3	y_2	y_1	y_0

Cela représente un code de longueur fixe, de 11 bits de long, dont la première partie, les y_i sont situés aux bits de poids *fort* — nous renversons ainsi la convention d'écriture normale des nombres, dont l'importance des chiffres va de droite à gauche. Les bits sont numérotés de la façon suivante :

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}

Collectivement, considérons que la séquence de bits $y_4y_3y_2y_1y_0$ forme un nombre sur 5 bits, y. Il en ira de même pour la séquence formée par $x_3x_2x_1x_0$, qui forme le nombre x. Les bits qui sont représentés explicitement par 0 ou 1 sont des bits constants, de valeur fixe pour ce code. On peut avoir besoin de bits fixés à des valeurs précises pour des fins de délimitation ou encore de vérification, soit encore pour indiquer que certains motifs de bits désignent une classe particulière du code.

Pour les bits qui forment des nombres, l'ordre conventionnel est respecté, c'est-à-dire que les nombres se lisent de droite à gauche. La raison en est que certains codes ne peuvent être décodés

que si on connaît les quelques premiers bits qui forment le préfixe, qui, traditionnellement, sont écrits de gauche à droite. Le préfixe introduit un certain nombre de bits formant un nombre, lequel, étant de longueur connue, peut être lu directement. Il n'y a alors aucun problème à écrire le nombre qui suit de droite à gauche.

Grâce à cette notation, nous pouvons nous dispenser d'une notation fastidieuse en parlant de x plutôt que du nombre formé par les bits $x_3x_2x_1x_0$. L'aspect visuel permet aussi de bien saisir les positions relatives des éléments d'un code donné. Cette convention est utilisée implicitement dans plusieurs ouvrages, dont [181, 184].

5.3.2 Codes *ad hoc*

De nombreuses méthodes de compression utilisent des codes *ad hoc* pour obtenir un peu de compression. Dans cette sous-section, nous considérerons quelques codes utilisées dans des méthodes de compressions anciennes, prévues pour des ordinateurs lents, ou simplement pour des systèmes embarqués tels que des modems et des cartes réseau. Ces méthodes utilisent généralement un code qui prend plus avantage des frontières naturelles des mots en mémoire que de la compression qui pourrait être réalisable.

5.3.2.1 Méthode de Hall

Cette méthode est utilisée par les anciens fichiers d'aide de Windows. Les textes des fichiers d'aide étaient sommairement compressés grâce à une méthode à base de dictionnaire de type LZ78. Les tuples émis par le compresseur sont un ou deux octets de long, définis par le code suivant :

Octet	Action
$x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ \vert\ 0$	Émettre x
$x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ \vert\ 0\ 1$	Émettre la phrase $64(x+1)+$ prochain octet.
$x_4\ x_3\ x_2\ x_1\ x_0\ \vert\ 0\ 1\ 1$	Copier les $x+1$ prochains octets
$x_3\ x_2\ x_1\ x_0\ \vert\ 0\ 1\ 1\ 1$	Émettre $x+1$ espaces
$x_3\ x_2\ x_1\ x_0\ \vert\ 1\ 1\ 1\ 1$	Émettre $x+1$ nuls

Ce code est utilisé pour représenter des nombres de 0 à 32 pour les copies, de 0 à 127 pour les littéraux, de 0 à 16383 pour les codes de « phrases », et enfin des nombres de 1 à 16 pour des répétions. Ce code permet deux représentations pour un espace simple : l'une avec le codes pour les littéraux, qui permet d'émettre directement x, une autre avec 1 répétition d'espaces. On cherchera cependant à éviter d'avoir plusieurs représentations pour un même objet, cela mène à des inefficacités indésirables et souvent facilement évitables ; ici il aurait suffit de faire en sorte que le code ne puisse représenter que $x+2$ espaces.

5.3.2.2 Compression RLE

RLE (*Run Length Encoding*, parfois aussi *bitpack*) encode les longueurs des répétitions de symboles dans la séquence à compresser. Si la longueur est limitée (et c'est presque toujours le cas), on aura un encodage dit RLL (*Run Length Limited*). Les répétitions sont introduites par un code qui donne simultanément l'information qu'il s'agit d'une répétition et le nombre de répétitions. Si les prochains symboles ne présentent pas de répétitions, on utilisera plutôt un code qui signale qu'une série de symboles encodés littéralement sont à recopier.

Cette technique consiste en l'utilisation de deux classes de codes qui sont distingués entre elles par un seul bit. Typiquement, ici aussi, les codes sont d'une longueur compatible avec les frontières naturelles de la mémoire, presque toujours un octet. Par exemple, les codes de la forme

R	c_6	c_5	c_4	c_3	c_2	c_1	c_0

disposent d'un bit R qui agit comme une valeur booléenne. $R = 1$ indiquerait un code répétition et $R = 0$ un code de littéraux. La valeur de c détermine le nombre de l'un ou l'autre, sauf que plutôt de représenter directement c, on prendra soin d'utiliser un *biais*. Comme les littéraux sont au moins au nombre de un (ça ne sert à rien de représenter zéro littéraux), l'intervalle représenté par c, si $R = 0$, va de 1 à 128 plutôt que de 0 à 127, donnant ainsi un biais de +1. De même pour les répétitions ; puisqu'on obtient un gain qu'à partir de répétitions de plus de deux symboles, c représentera de 3 à 130, soit un biais de +3.

Cette méthode de compression n'est pas très astucieuse car elle donne des codes de longueur égale à toutes les répétitions bien qu'en fait les répétitions ne soient pas distribuées uniformément. Comme nous le verrons à la section 5.3.5.1, nous pouvons concevoir un code très efficace en supposant que les répétions soient distribués selon une loi géométrique. De plus, la méthode dépend du nombre de répétitions effectivement présentes dans la séquence à compresser. À l'époque où cette méthode de compression a commencé à être utilisée (les années soixantes ?) les données pouvaient effectivement contenir de très longues répétitions. S'il s'agissait de données textuelles, on pouvait avoir de longues séquences d'espaces (c.f. la figure 1.6, page 13) ce qui donnait une compression acceptable.

Appliquée aux images, cette méthode était relativement efficace car si on dispose de très peu de valeurs distinctes pour les pixels, la probabilité que deux pixels successifs soient de même valeur est quand même élevée. Dans les protocoles de fac-similés (fax), où on ne dispose que de deux « couleurs », soit le noir et le blanc, on utilise une variation de RLE où les longueurs sont données par des codes de type Huffman. Naturellement, cette méthode est catastrophique lorsque le nombre de valeurs distinctes pour les pixels est très grand. Si la méthode est plutôt bonne pour les images monochromes, elle devient très mauvaise pour les images naturelles où presqu'aucun pixel n'a la même valeur que son voisin.

5.3.2.3 Compression *Bitpack*

Il arrive parfois que, bien que le mot naturel pour l'ordinateur soit l'octet, nous utilisions beaucoup moins que 256 valeurs distinctes. Si on a, par exemple, que 32 valeurs distinctes, on gaspille effectivement trois bits par octet, puisque 5 bits par symbole suffisent amplement ! La technique *bitpack* consiste à trouver le plus petit commun multiple entre le nombre de bits par

mots et le nombre de bits nécessaires par symbole. Dans notre exemple, $\text{ppcm}(5,8) = 40$. On peut donc placer 8 symboles sur 5 octets, récupérant ainsi $\frac{3}{8}^{\text{e}}$ des bits. Cette technique est plutôt une technique de compaction, malgré le fait qu'aucune information ne soit perdue.

En général, si Σ est l'alphabet à coder, nous aurons besoin de $s = \lceil \lg |\Sigma| \rceil$ bits pour coder chaque symbole. Soit b, le nombre de bits contenu dans notre mot naturel. Typiquement $b = 4$ ou $b = 8$. Alors nous encoderons $\text{ppcm}(s,b)/s$ symboles sur $\text{ppcm}(s,b)$ bits. Il est facile de voir que le plus petit commun multiple de s et b est en effet le plus petit nombre de bits qui permet d'encoder un nombre entier de symboles. Si $b = 2^c$, nous avons un cas spécial. Soit $s = 2^d n_0$, avec $d \geqslant 0$, et tel que d correspond à l'exposant de la factorisation canonique (maximale). Alors $\text{ppcm}(b,s) = \text{ppcm}(2^c, 2^d n_0) = 2^{\max(c,d)} n_0$.

5.3.2.4 Bigrammes

La méthode de compression par bigramme, comme la méthode *bitpack* suppose que nous ayons beaucoup moins de valeurs utilisées que de valeurs disponibles. Par exemple, dans le cas où nous avons du texte, nous utilisons environ 100 symboles distincts, y compris les codes de contrôle, ce qui nous laisse 156 symboles inutilisés, donc disponibles pour représenter de l'information. Nous allons assigner à chacun des symboles disponibles un bigramme, c'est-à-dire une paire de symboles.

Pour ce qui suit, posons $N = 2^n$, le nombre de symboles disponibles et $0 < k \leqslant N$, le nombre de symboles effectivement utilisés dans la séquence. Soit D, le dictionnaire de bigrammes.

La stratégie la plus simple consiste à construire le dictionnaire de bigrammes (au nombre de $N - k$), par une méthode systématique, par exemple en prenant le produit cartésien des consonnes et des voyelles. Si on obtient plus de $N - k$ paires, on ne conserve que les $N - k$ premières pour former le dictionnaire D. Cette méthode ne demande pas de stocker le dictionnaire dans la version compressée car, étant conventionnel, le décodeur le connaît aussi.

Une meilleure stratégie consiste à calculer le dictionnaire de bigrammes à partir de la séquence elle-même. Le calcul d'un dictionnaire optimal étant un problème $\mathcal{NP}$-ardu, on utilise habituellement l'heuristique qui consiste à choisir les $N - k$ bigrammes les plus fréquents, sans tenir compte de l'alignement. C'est à dire, que le mot « bigramme » donnera les bigrammes bi, ig, gr, ra, am, mm, et me. On compte les bigrammes et le nombre d'occurence et on ne conserve que les $N - k$ premiers. Cela demande de stocker $2N - k$ symboles au début du fichier, ce qui devient négligeable lorsque la séquence est longue.

La longueur moyenne du code par symbole est déterminée par la probabilité que la prochaine paire de symboles soit dans le dictionnaire et n, la longueur en bits du code de base. La longueur moyenne, en supposant que les bigrammes soient i.i.d. entre eux, est donnée par :

$$\begin{aligned} L_{bi}(X) &= P(x_t^{t+1} \in D)\frac{n}{2} + P(x_t^{t+1} \notin D)n \\ &= n\left(\frac{1}{2}(1 - P(x_t^{t+1} \notin D)) + P(x_t^{t+1} \notin D\right) \\ &= n\left(\frac{1}{2} + \frac{1}{2}P(x_t^{t+1} \notin D)\right) \end{aligned}$$

Très clairement, $\frac{n}{2} \leqslant L_{bi} \leqslant n$, ce qui fait que le ratio de compression obtenu est entre 1 : 1 et 2 : 1. Le comportement de $P(x_t^{t+1} \in D \mid x_{t-2}^{t-1} \in D)$ est étudié de façon simplifiée par Bookstein et Fouty [19]. Bien que le ratio de compression ne soit pas extraordinaire, l'algorithme de compression est très simple et le découpage vorace de la séquence est optimal. En effet, il suffit de comparer les deux prochains symboles de la séquence avec le dictionnaire. S'ils y figurent, on émet le code pour le bigramme et on avance de deux symboles dans la séquence. S'ils ne s'y trouvent pas, on émet le code pour le prochain symbole dans la séquence et on avance de un. Cet algorithme vorace est asymptotiquement optimal pour le découpage de la séquence en bigrammes [12]. La décompression est très simple, puisqu'à chaque étape, on lit un code qui nous dit si on doit émettre un seul symbole ou un bigramme.

Nous reviendrons à la section 7.2 sur des améliorations à apporter à cette méthode de façon à en augmenter la performance. En effet, on conçoit qu'il y ait une inefficacité à utiliser un code pour un symbole très rare alors que ce code pourrait être utilisé pour encoder un bigramme beaucoup plus fréquent.

5.3.2.5 Méthode de Karlgren

Supposons que nous ayons un alphabet Σ dont le nombre de symboles excède largement le nombre de valeurs représentables grâce à un octet, et que nous devions représenter une séquence de ces symboles de la façon la plus compacte possible. Une approche pourrait être d'utiliser $\lceil \lg |\Sigma| \rceil$ bits par symboles, au coût de manipulations de bits supplémentaires s'il s'avert que le nombre de bits requis est plutôt quelconque et ne correspond pas à une frontière naturelle pour la machine — auquel cas nous revenons à une méthode de type *bitpack*.

Karlgren propose la solution suivante [109]. Divisons l'alphabet Σ en plusieurs registres. L'alphabet original est découpé en plages d'une certaine longueur, de façon à ce que la probabilité que deux symboles consécutifs dans les séquences appartiennent au même registre soit élevé. On réservera pour chaque registre un certain nombre de codes pour la commutation entre les registres. Donc, étant donné l'alphabet Σ, on veut calculer des registres qui contiennent, disons 256 symboles, y compris les codes de commutation.

Si n est le nombre de bits par code, nous aurons $2^n - r + 1$ codes par registre pour les symboles de l'alphabet Σ, plus $r - 1$ codes pour la commutation de registre[2]. Le nombre de registres r, étant donné n et $|\Sigma|$ est tel que

$$r = \frac{|\Sigma|}{2^n - r + 1}$$

soit,

$$r = \frac{1}{2}\left(1 + 2^n \pm \sqrt{(2^n + 1)^2 - 4\,|\Sigma|}\right)$$

où on prend la solution minimale, entière, mais positive pour r. La longueur moyenne des codes

2 Il n'est pas utile de commuter vers le registre courant, éliminant ainsi un code de commutation. Toutefois, on pourrait être tenté de commuter vers le registre courant de temps en temps afin de s'immuniser contre les erreurs de transmission qui peuvent transformer un symbole ordinaire en symbole de commutation ou vice-versa.

pour une séquence représentée grâce à la méthode de Karlgren est :

$$\begin{aligned} Karlgren(X) &= P(\rho(x_{t+1}) = \rho(x_t))n + P(\rho(x_{t+1}) \neq \rho(x_t))2n \\ &= n(2 - P(\rho(x_{t+1}) = \rho(x_t))) \end{aligned}$$

où $\rho(x)$ donne le registre du symbole x. Nous avons un gain dès que

$$\begin{aligned} Kalgren(X) &\leqslant \lceil\, |\Sigma| \,\rceil \\ n(2 - P(\rho(x_{t+1}) = \rho(x_t))) &\leqslant \lceil\, |\Sigma| \,\rceil \\ P(\rho(x_{t+1}) = \rho(x_t)) &\geqslant 2 - \frac{\lceil\, |\Sigma| \,\rceil}{n} \end{aligned}$$

Si on ne contraint pas n à être, disons 8 ou 16, on peut minimiser $Karlgren(X)$ en fonction de r et de n simultanément. Le problème majeur demeure de déterminer avec quelque précision $P(\rho(x_{t+1}) = \rho(x_t))$, ce qui dépend en retour de la distribution des symboles dans les divers registres. L'affectation des symboles aux registres n'est pas triviale.

5.3.2.6 *Binary Coded Text*

Tropper propose une méthode de compression à base de dictionnaire dans laquelle les mots sont décomposés en préfixes, suffixes et radicaux communs, en plus d'utiliser une méthode pour construire incrémentalement un dictionnaire de nouveaux mots [208]. Nous en avons brièvement discouru à la section 4.2.1. Les codes utilisés par la méthode de Tropper sont alignés sur les octets, et le premier octet contient un bit qui indique la présence facultative d'un second octet complétant le code. Les deux octets combinés fournissent ainsi des codes plus longs. Les codes sont structurés de la façon suivante :

c	a_6	a_5	a_4	a_3	a_2	a_1	a_0

et si $c = 1$, le code est continué par

s_3	s_2	s_1	s_0	d	a_9	a_8	a_7

Le nombre a détermine l'adresse du mot dans le dictionnaire. Si $c = 0$, alors a est un nombre sur 7 bits, si plutôt $c = 1$, a est un nombre sur 10 bits. Si $d = 0$ (et d sera zéro si $c = 0$), il s'agit d'un mot du dictionnaire précalculé, et si $d = 1$, il s'agira du dictionnaire dynamique. Le suffixe du mot est déterminé par s, ce qui laisse 16 suffixes, dont un suffixe nul. Les mots sont ajoutés d'une façon très simple. On émet un numéro d'entrée qui dépasse la dernière entrée dans le dictionnaire dynamique (ce qui réfère donc à un mot inexistant) puis le mot en clair, suivi d'un espace qui le délimite. Cette méthode donne de relativement bons résultats car les mots qui sont mis dans le dictionnaire statique sont précalculés en fonction de la loi de Zipf.

5.3.2.7 *4 Bit Coded Text*

Cette méthode, dûe à Pike, utilise un code aligné sur quatre bits plutôt qu'un octet [167]. La méthode de Pike suppose que nous ayons un dictionnaire de symboles très fréquents, codés sur 4 bits, de mots très fréquents et de symboles moins fréquents, codés sur 8 bits, et enfin un

dictionnaire de symboles rares et de mots moins fréquents, dont les index sont codés sur 12 bits. Le tableau résume la structure des codes utilisés :

Code	Action
$a_3\ a_2\ a_1\ a_0$	Émettre un des 13 symboles fréquents ($a \geqslant 3$)
$0\ 0\ 0\ 0\ \vert\ x_3\ x_2\ x_1\ x_0$	Émettre le mot x
$0\ 0\ 0\ 1\ \vert\ s_3\ s_2\ s_1\ s_0$	Émettre le symbole s
$0\ 0\ 1\ 0\ \vert\ r_7\ r_6\ r_5\ r_4\ r_3\ r_2\ r_1\ r_0$	Émettre le mot r

Ici encore, on s'appuie sur l'hypothèse de Zipf pour choisir les symboles fréquents et les mots des différents dictionnaires. Le dictionnaire étant choisi *a priori*, il n'est pas nécessaire de le transmettre. Aucune disposition particulière n'est prévue pour les texte en une langue autre que l'anglais, bien qu'il eût suffit de prévoir plus d'un dictionnaire et quelques bits pour indiquer le choix de la langue.

5.3.2.8 Compression FTP

Le protocole FTP (*file transfer protocol*) est utilisé pour transférer des fichiers d'un ordinateur à un autre via une connexion de type telnet ou internet. Dans le mémo RFC #468, Broden propose un mécanisme simple de compression pour accélérer les transferts [29]. Le mécanisme s'inspire un système précédant, le système HASP-II (*Houston Automatic Spooling Program*, d'IBM, qui contrôlait les tâches à être exécutées sur l'ordinateur IBM 360 [102]. Cette méthode est essentiellement une variante de la compression RLE (voir section 5.3.2.2). Les codes utilisés par le protocole FTP sont donnés par :

Code	Action
$0\ \vert\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$	Copier les x prochains symboles.
$1\ 0\ \vert\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$	Copier x fois le prochain octet.
$1\ 1\ \vert\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$	Émettre x espaces (ou nuls).
$0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$	Le prochain octet est un littéral.

Ces codes sont en effet très semblables aux codes utilisées dans HASP-II :

Code	Action

1	1	x_5	x_4	x_3	x_2	x_1	x_0	Copier les x prochains symboles.
1	0	0	x_4	x_3	x_2	x_1	x_0	Émettre x espaces (ou nuls).
1	0	1	x_4	x_3	x_2	x_1	x_0	Copier x fois le prochain octet.
0	0	0	0	0	0	0	0	Fin de record.

malgré que les codes pour HASP-II ne prévoient pas de mécanisme d'échappement permettant de représenter un seul symbole nu ; on a affaire soit à un groupe de répétitions, soit à un groupe de littéraux. Dans le cas des codes FTP comme pour les codes HASP-II on a des biais sur les valeurs des comptes de répétions et de littéraux.

5.3.3 Codes universels

Les codes que nous avons présenté à la section précédente ont tous été conçus pour minimiser la longueur moyenne des codes, mais sans nécessairement tenir grand compte de la fonction de distribution des entiers qu'ils sont cencés coder. Ces codes faisaient implicitement la supposition que si $x < y$, alors $P(x) \leqslant P(y)$. Les codes universels utilisent aussi cette supposition mais n'imposent pas de valeur maximale sur les entiers à coder ; les codes universels sont capables de coder des entiers arbitrairement grands, et ce, d'une façon qui est asymptotiquement optimale sous des conditions particulières que nous énonçons dès maintenant.

5.3.3.1 Conditions d'universalité

La définition d'un code universel nous est donnée pour la première fois par Elias [66]. Un code est universel, si, pour une fonction de distribution $P(n)$ telle que $\forall\, n \in \mathbb{N},\ P(X = n) \neq 0$, et $\forall\, n \geqslant n_0$, pour un $n_0 \in \mathbb{N},\ P(X = n+1) \leqslant P(X = n)$ il respecte

$$\lim_{n\to\infty} \frac{\sum_{i=1}^{n} P(X = i) L_u(i)}{\sum_{i=1}^{n} P(X = i) \lg i} = k \geqslant 1 \tag{5.4}$$

pour la fonction de longueur de code L_u.

La première condition, $P(X = n) \neq 0$, nous dit que tout nombre naturel est susceptible d'être observé, bien que la condition deux, soit $P(X = n+1) \leqslant P(X = n)$, stipule que les très grands nombres sont aussi très improbables. La condition deux impose que la fonction de distribution de probabilité soit éventuellement décroissante. Cela permet d'avoir une fonction masse, avec n_0 tel qu'il dépasse le mode, dont la queue s'amenuise rapidement lorsque n croît. La troisième exigence, soit l'existance de la limite de l'éq. (5.4), contraint les codes à avoir une longueur asymptotique proportionelle à $\lg n$, pondérée par les probabilités d'occurence. Un code qui a une limite de k est dit k-universel. Les codes les plus efficaces, que l'on dira simplement universels, auront $k = 1$, ce qui est aussi une borne inférieure pour les codes autodélimitants.

L'utilité des codes universels n'est pas visible de suite ; en général, nous n'avons que de relativement petits nombres à coder et il existe des codes plus efficaces que les codes universels

pour y parvenir. Or, les codes universels permettent justement de ne pas avoir de contrainte sur la taille des entiers que nous codons, s'il advient que nous ayons à le faire. Les codes universels donnent aussi des codes courts aux petits entiers, ce qui peut les rendre utilisables pour des entiers de magnitudes usuelles, tout en ayant la souplesse nécessaire pour coder des entiers arbitrairement grands.

5.3.3.2 Codes de Chaitin

Le premier code de Chaitin, est le code intercallaire [38]. Pour encoder un entier n, grâce à un code C_β, il faut $\lceil \lg n \rceil$ bits, mais il faut encore compter les bits nécessaires à l'encodage de la longueur. Dans le code intercallaire de Chaitin, chaque bit de $C_\beta(n)$ est précédé d'un bit qui indique s'il reste des bits à lire dans le code. Le code a donc la forme

$$C_\gamma(n) = n_0 : 0 : n_1 : 0 : n_2 : 0 \ldots n_{\lceil \lg n \rceil - 2} : 0 : n_{\lceil \lg n \rceil - 1} : 1 : n_{\lceil \lg n \rceil}$$

ou, si on réarrange :

$$C'_\gamma(n) = \underbrace{000\ldots000}_{\lceil \lg n \rceil - 1 \text{ fois}} 1 : C_\beta(n) = C_\alpha(\lceil \lg n \rceil) : C_\beta(n)$$

ce qui donne une longueur de code

$$|C_\gamma(n)| = |C'_\gamma(n)| = L_\gamma(n) = 2\lceil \lg n \rceil$$

Nous pouvons aussi réduire de 1 la longueur du code en utilisant le code $C_{\hat{\beta}}(n)$, qui omet le bit le plus significatif (car puisque le bit le plus significatif est toujours à 1, on n'a pas besoin de le coder explicitement). Ce code est 2-universel, donc pas tellement efficace. Posons donc

$$C'_{\hat{\gamma}}(n) = C_\alpha(\lceil \lg n \rceil) : C_{\hat{\beta}}(n)$$

qui a la fonction de longueur

$$L_{\hat{\gamma}}(n) = 2\lceil \lg n \rceil - 1$$

Le code est plutôt long car la longueur est encodée de façon unaire. Si nous encodions le préfixe différemment, nous pourrions obtenir un code plus court. Imaginons la stratégie suivante :

$$C''_{\hat{\gamma}}(n) = C'_{\hat{\gamma}}(\lceil \lg n \rceil) : C_{\hat{\beta}}(n)$$

Ce code est déjà beaucoup plus court que le code $C_{\hat{\gamma}}(n)$, car maintenant

$$\begin{aligned} |C''_{\hat{\gamma}}(n)| = L''_{\hat{\gamma}}(n) &= (\, 2\lceil \lg \lceil \lg n \rceil \rceil - 1 \,) + (\, \lceil \lg n \rceil - 1 \,) \\ &= 2\lceil \lg \lceil \lg n \rceil \rceil + \lceil \lg n \rceil - 2 \end{aligned}$$

et cette fonction de longueur correspond effectivement à un code universel car

$$\lim_{n \to \infty} \frac{2\lceil \lg \lceil \lg n \rceil \rceil + \lceil \lg n \rceil - 2}{\lg n} = 1$$

puisque

$$\lg n + 2 \lg \lg n \leqslant L''_{\hat{\gamma}}(n) \leqslant \lg n + 2 \lg \lg n + 3$$

Ces codes ont été introduits par Chaitin pour faciliter l'étude de la complexité algorithmique des séquences (voir la section 3.2) où il lui fallait disposer d'un moyen d'encoder les longueurs des séquences de façon auto-délimitante.

5.3.3.3 Codes d'Elias et codes Even-Rodeh

Si nous poussons plus loin l'idée de Chaitin de coder récursivement la longueur du code, nous obtenons les codes d'Elias. Elias a eu l'idée d'encoder complètement récursivement la longueur [66]. Puisque la longueur est codée récursivement, il faut commencer par lire les premiers bits. Le premier bloc est de longueur 2. Un bloc de bits qui commence par un zéro détermine la fin du code, et le bloc précédent contient l'entier n à lire. Si on lit le zéro dès le premier coup, le nombre est $n = 1$. Si le code commence plutôt par un un, alors on lit le nombre de bits (moins un, puisque nous avons déjà lu le premier bit) indiqué par le bloc précédent. Ce nombre de bits indique la taille du prochain bloc, et on poursuit jusqu'à ce que l'on trouve un premier bit à zéro.

Le code pour n est donné par

$$C_\omega(n) = \begin{cases} 0 & \text{si } n = 1 \\ H_\omega(n) : 0 & \text{si } n > 1 \end{cases}$$

où

$$H_\omega(n) = \begin{cases} 0 & \text{si } n = 1 \\ 10 & \text{si } n = 2 \\ 11 & \text{si } n = 3 \\ H_\omega(\lfloor \lg n \rfloor) : C_\beta(n) & \text{si } n \geqslant 4 \end{cases}$$

Le code généré par cette formule est donné au tableau 5.1. La longueur du code pour n est elle aussi définie récursivement :

$$|C_\omega(n)| = L_\omega(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 + L'_\omega(n) & \text{sinon} \end{cases}$$

et

$$L'_\omega(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2 & \text{si } n = 2, 3 \\ L'_\omega(\lfloor \lg n \rfloor) + \lceil \lg n \rceil & \text{sinon} \end{cases}$$

d'où on trouve

$$\begin{aligned} |C_\omega(n)| &= \lceil \lg n \rceil + \lceil \lg \lfloor \lg n \rfloor \rceil + \cdots \\ &\approx \lg n + \lg \lg n + \lg \lg \lg n + \cdots \end{aligned}$$

En utilisant le même argument que nous avons utilisé pour les codes de Chaitin, nous trouvons que les codes d'Elias sont universels.

Une variante de ces codes a été proposée par Even et Rodeh [67]. Les codes de Even et Rodeh sont structurés de façon identique aux codes d'Elias, sauf pour les conditions initiales qui prévoient des blocs d'une longueur de trois bits. Even et Rodeh appliquent leur code à l'encodage des séparateurs de mots (qu'ils nomment en terme générique « virgules »). Puisque les codes de Even et Rodeh sont aussi universels, leur méthode est optimale si on ne considère que vaguement la probabilité sur la longueur des mots. Ici aussi, Zipf nous dit que la longueur des

1	0
2	10 0
3	11 0
4	10 100 0
5	10 101 0
6	10 110 0
7	10 111 0
8	11 1000 0
⋮	⋮
14	11 1110 0
15	11 1111 0
16	10 100 10000 0
17	10 100 10001 0
⋮	⋮

TAB. 5.1 – Le code d'Elias pour les entiers.

mots est contrainte à être courte et que les mots très longs sont aussi très rares.

Un code universel, quel qu'il soit, est contraint d'avoir une fonction de longueur asymptotique supérieure ou égale à $\lg n + \lg\lg n + \lg\lg\lg n + \cdots$ pour un entier n. Cette longueur est $O(\lg n)$. On peut naturellement établir des conditions de base qui font en sorte que la longueur des codes pour de petits entiers soit légèrement plus courte que la longueur asymptotique, mais au coût d'avoir des codes beaucoup plus long pour les grands entiers.

5.3.3.4 Codes de Stout

Les codes proprosés par Stout s'attaquent au problème de la longueur des codes pour les petits entiers [200, 4]. Les codes de Stout utilisent un paramètre de contrôle pour ajuster la longueur des codes. Bien qu'asymptotiquement les codes de Stout respectent la longueur minimale des codes universels, les codes pour les petits entiers peuvent être plus courts que les codes d'Elias.

Comme pour les codes d'Elias, les codes de Stout sont exprimés récursivement. La fonction de longueur du code pour un entier n, paramétrisée par un entier d, est donnée par

$$\lambda_{[d]}(n) = \lfloor \lg n \rfloor - d$$

et $\lambda^k_{[d]}(n)$ est la k-composition de $\lambda_{[d]}$ que l'on définit ainsi : $\lambda^k_{[d]}(n) = \lambda_{[d]}(\lambda_{[d]}(\cdots(n)\cdots)$. Les codes sont générés par

$$C_S(n,d) = \begin{cases} C_{\beta(d)}(n) : 0 & \text{si } n < 2^d \\ H_S(\lambda_{[d]}(n), d) : C_{\beta(d)}(n) : 0 & \text{sinon} \end{cases}$$

et

$$H_S(n,d) = \begin{cases} C_{\beta(d)}(n) & \text{si } n < 2^d \\ H_S(\lambda_{[d]}(n), d) : C_{\beta(d)}(n) & \text{sinon} \end{cases}$$

n	$d=0$	$d=1$	$d=2$
1	0	00	000
2	10	10	010
3	1 10 0	1 10 0	100
4	1 11 0	1 11 0	110
5	1 10 100 0	1 100 0	00 100 0
6	1 10 101 0	1 101 0	00 101 0
7	1 10 110 0	1 110 0	00 110 0
8	1 10 111 0	1 111 0	00 111 0
9	1 11 1000 0	0 10 1000 0	01 1000 0
10	1 11 1001 0	0 10 1001 0	01 1001 0
11	1 11 1010 0	0 10 1010 0	01 1010 0
12	1 11 1011 0	0 10 1011 0	01 1011 0
13	1 11 1100 0	0 10 1100 0	01 1100 0
14	1 11 1101 0	0 10 1101 0	01 1101 0
15	1 11 1110 0	0 10 1110 0	01 1110 0
16	1 11 1111 0	0 10 1111 0	01 1111 0
17	1 10 100 10000 0	0 11 10000 0	10 10000 0
18	1 10 100 10001 0	0 11 10001 0	10 10001 0
⋮	⋮	⋮	⋮

TAB. 5.2 – Les codes de Stout pour quelques valeurs de d.

Alors qu'avec les codes d'Elias on commence par lire 2 bits, avec les codes de Stout, on en lit d. La fonction de longueur des codes est donnée par

$$\begin{aligned}|C_s(n,d)| = L_S(n,d) &= d+1+\sum\nolimits_{i\geqslant 0,\ \lambda^i_{[d]}(n)\geqslant 0}\left(\lambda^i_{[d]}(n)+d+1\right)\\ &= (k+1)(d+1)+\sum\nolimits_{i=1}^{k}\lambda^i_{[d]}(n)\end{aligned}$$

où $k=\max\{i \mid \lambda^i_{[d]}(n)\geqslant 0\}$.

Les codes de Stout généralisent les codes d'Elias. En particulier, nous avons que $C_S(n,0) = 1 : C_\omega(n)$ (donc $L_S(n,0) = 1 + L_\omega(n)$). Les codes de Stout pour quelques valeurs de d sont donnés au tableau 5.2.

5.3.3.5 Codes de Fibonacci

On doit la suite de Fibonacci à Leonardo Pisano, dit Fibonacci (1170–1250). Dans son livre révolutionnaire, *Liber Abaci* (le « livre des abaques ») publié pour la première fois en 1202 (cette édition est perdue) et réédité en 1228, Leonardo Pisano présente les chiffres dits arabes et les méthodes de calcul les utilisant [168]. Fibonacci s'était vu fortement influencé par l'œuvre de Muḥammad Ibn Mūsā al-Khwarizmi (dont une corruption du nom a fini par nous donner *algorithme*). Jusqu'alors, les opérations mathématiques se faisaient grâce aux nombres romains et une table abaque, mais Fibonacci prônera plutôt une utilisation des chiffres arables et des

algorismes, c'est-à-dire essentiellement des méthodes de calcul à l'aide d'un crayon. L'introduction par les marchands des chiffres arabes en Europe provoqua un véritable choc culturel, où la contre-culture algorismiste s'opposa à l'ordre séculaire imposée par les nombres romains.

Malgré les contributions importantes à la culture occidentale de Leonardo Pisano, on ne le connaît habituellement que par les nombres « de Fibonacci », définis par la récurrence suivante :

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Cette récurrence illustrait la croissance d'une population de lapins en tenant compte de leur période de maturation. Or, il s'avert que les séquences de Fibonacci — séquences au pluriel car il suffit de changer les valeurs initiales pour obtenir des séquences différentes — apparaissent dans de nombreux phénomènes naturels, géométriques et combinatoires, ce qui rend leur étude particulièrement intéressante. Les nombres de Fibonacci, considérés comme un ensemble, forment un ensemble complet, au sens mathématique, c'est-à-dire que tout nombre naturel s'exprime par une combinaison linéraire de ces nombres, où les coefficients de la combinaison sont contraints à être 0 ou 1. Cette propriété n'est pas unique aux nombres de Fibonacci ; la récurrence de Lucas et certaines récurrences d'Ulam possèdent aussi cette propriété [220].

La série de coefficients de la combinaison linéaire pour un nombre se nomme la représentation de Zeckendorf de ce nombre [111, 30, 211]. Cette représentation est en fait une séquence infinie de bits, où, après un nombre fini de coefficients à un, tous les coefficients sont zéro. Soit

$$\text{Fibolog}(n) = \max\{i \mid F_i \leqslant n\}$$

l'indice du plus grand nombre de Fibonacci inférieur ou égal à n. Il existe plusieurs combinaisons linéaires pour un même nombre, mais utilisons spécialement la décomposition où les plus grands nombres de Fibonacci sont d'abord choisis. On nomme cette décomposition la décomposition duale (*Dual Zeckendorf*), par opposition à la décomposition simple de Zeckendorf. Par exemple,

$$11 = F_6 + F_4 = 8 + 3$$

converti en coefficients, donne $\{0, 0, 0, 1, 0, 1, 0, \dots\}$. La représentation tronquée de Zeckendorf de 11, étant donné les nombres de Fibonacci, est donc $\{0, 0, 0, 1, 0, 1\}$. Cet algorithme vorace donne toujours 0 pour F_0, puisque $F_1 = 1$; on peut donc élaguer sans problème le coefficient de F_0, donnant ainsi la représentation finale $\{0, 0, 1, 0, 1\}$.

La représentation tronquée et élaguée de Zeckendorf ne nous donne pas encore un code uniquement décodable. En utilisant l'observation que cette décomposition vorace ne nous donne jamais deux coefficients consécutifs à un et sachant que le dernier coefficient retenu est toujours à un, on peut délimiter le code en lui ajoutant simplement un 1 terminal. La table 5.3 donne les codes de Fibonacci pour les quelques premiers entiers.

La procédure pour générer les codes de Fibonacci repose sur une décomposition récursive :

$$Z_{fib}(n) = \begin{cases} \varnothing & \text{si } n = 0 \\ \{1\} & \text{si } n = 1 \\ \{\text{Fibolog}(n)\} \cup Z_{fib}(n - F_{\text{Fibolog}(n)}) & \text{sinon} \end{cases}$$

et le code lui-même est donné par

$$C_{fib}(n) = (w : 1) \mid (\,|w| = \text{Fibolog}(n)) \wedge ((w_{i-1} = 1) \leftrightarrow F_i \in Z_{fib}(n))$$

c'est-à-dire une séquence de Fibolog(n)+1 bits, où le bit $i-1$ est à un si F_i est dans la décomposition de Zeckendorf de n, $Z_{fib}(n)$. La longueur des codes de Fibonacci est donc gouvernée par la fonction Fibolog(n) :

$$|C_{fib}(n)| = \text{Fibolog}(n) + 1 \tag{5.5}$$

La fonction Fibolog(n) croît lentement, mais plus rapidement que $L_\omega(n)$, la fonction de longueur des codes d'Elias. Les croissances sont comparées :

n	$L_\omega(n)$	Fibolog(n)	$\lg n$
10	7	6	3.32
10^{10}	46	49	33.21
10^{100}	349	480	332.19
10^{1000}	3341	4786	3321.92
10^{10000}	33243	47851	33219.28

Sachant que

$$\lim_{n\to\infty} \frac{\text{Fibolog}(n)}{\lg n} = \frac{1}{\lg\phi} = \frac{\ln 2}{\ln\phi} \approx= 1.44042009\ldots$$

où $\phi = \frac{1+\sqrt{5}}{2}$, le nombre d'or, on déduit que les codes de Fibonacci ne sont pas strictement universels, mais $\frac{1}{\lg\phi}$-universels. Faisons en la démonstration. Nous utiliserons l'identité

$$F_n = \left[\frac{\phi^n}{\sqrt{5}}\right] = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

où $[x]$ est l'arrondi de x. Puisque la fonction Fibolog(n) est défini comme $\text{Fibolog}(n) = \max\{i | F_i \leqslant n\}$, cela revient à résoudre

$$\frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \leqslant n$$

pour la plus grande valeur de i satisfaisant l'inégalité. Dès lors, la dérivation est simple :

$$\begin{aligned}
\frac{\phi^i}{\sqrt{5}} + \frac{1}{2} &= n \\
\frac{\phi^i}{\sqrt{5}} &= n - \frac{1}{2} \\
\lg\left(\frac{\phi^i}{\sqrt{5}}\right) &= \lg\left(n - \frac{1}{2}\right) \\
i\lg\phi - \lg\sqrt{5} &= \lg\left(n - \frac{1}{2}\right) \\
i &= \frac{\lg(n - \frac{1}{2}) + \lg\sqrt{5}}{\lg\phi}
\end{aligned}$$

n	$C_{fib}(n)$	n	$C_{fib}(n)$
1	1**1**	12	10101**1**
2	01**1**	13	0000011
3	001**1**	14	1000011
4	101**1**	15	0100011
5	00011	16	0010011
6	10011	17	1010011
7	01011	18	0001011
8	000011	19	1001011
9	100011	20	0101011
10	010011	...	...
11	001011		

TAB. 5.3 – Les codes de Fibonacci pour les quelques premiers entiers. En gras, le 1 délimiteur.

Pour satisfaire l'inégalité, on aura

$$\text{Fibolog(n)} = \left\lfloor \frac{\lg(n-\frac{1}{2}) + \lg\sqrt{5}}{\lg\phi} \right\rfloor$$

Cela nous permet de montrer que les codes de Fibonacci sont bien universels :

$$\begin{aligned}
\lim_{n\to\infty} \frac{\text{Fibolog}(n)+1}{\lg n} &= \lim_{n\to\infty} \frac{\left\lfloor \frac{\lg\sqrt{5}+\lg(n-\frac{1}{2})}{\lg\phi} \right\rfloor + 1}{\lg n} \\
&= \lim_{n\to\infty} \frac{\lg\sqrt{5} + \lg(n-\frac{1}{2})}{\lg\phi\ \lg n} + \frac{1}{\lg n} \\
&= \lim_{n\to\infty} \frac{\lg\sqrt{5}}{\lg\phi\ \lg n} + \frac{\lg(n-\frac{1}{2})}{\lg\phi\ \lg n} + \frac{1}{\lg n} \\
&= \frac{1}{\lg\phi} \\
&\approx 1.44042009\ldots
\end{aligned}$$

concluant ainsi la démonstration de la $\frac{1}{\lg\phi}$-universalité des codes de Fibonacci. ■

5.3.4 Codes à domaine restreint

Les codes universels nous offrent la possibilité de coder des entiers arbitrairement grands, mais souvent nous n'avons besoin de coder que des entiers sur un intervalle donné, possiblement très grand, mais fini. Par exemple, plusieurs méthodes de compression vont générer des tuples $\langle\, position, longueur \,\rangle$ où l'un et l'autre des nombres est contraint à de relativement petites valeurs. Dans cette section, nous présentons quelques codes utilisés pour encoder les entiers tirés d'un intervalle limité.

5.3.4.1 Codes *ad hoc*

Les protocoles MNP5 et MNP7 (*Microcom Network Protocol*) sont d'autres schèmes simples de type *move to front* qui utilisent des codes statiques. Le lecteur se souviendra que nous avons introduit une méthode MTF à la section 4.2.2.1, page 69, pour la transformée de Burrows-Wheeler. La méthode MNP5 utilise deux types de compression. Outre l'encodage de type MTF des symboles individuels, elle utilise une forme de codage RLE. Plutôt que de réserver explicitement des codes pour les répétitions qui se différencient des codes pour les symboles littéraux, MNP5 déclenche le mode RLE après la lecture de trois symboles consécutifs identiques. Donc, après trois symboles répétés, le décodeur s'attend à lire un tuple de répétition, soit un compte (qui peut être zéro, car les trois répétitions sont peut-être une fausse alerte) suivi d'un symbole littéral. C'est en fait assez astucieux de procéder ainsi ; en effet, on ne limite pas les codes pour les littéraux en prévoyant un cas spécial pour les comptes de répétitions et comme le seuil de rentabilité pour une répétition est de quatre symboles, on n'a pas vraiment de perte en utilisant cette technique, le coût est amorti.

Le code utilisé dans MNP5 est un code de longueur variable, dont la longueur est déterminé par un préfixe de taille fixe, de trois bits. Ces trois premiers bits déterminent combien de bits il reste à lire pour le code. Le code est structuré de la façon suivante :

Code	Intervalle représenté
0 0 0 x_0	0 – 1
0 0 1 x_0	2 – 3
0 1 0 $x_1 x_0$	4 – 7
0 1 1 $x_2 x_1 x_0$	8 – 15
1 1 1 $x_3 x_2 x_1 x_0$	16 – 31
1 0 1 $x_4 x_3 x_2 x_1 x_0$	32 – 63
1 1 0 $x_5 x_4 x_3 x_2 x_1 x_0$	64 – 127
1 1 1 $x_6 x_5 x_4 x_3 x_2 x_1 x_0$	128 – 255*

*Le dernier code est donné par 111 | 11111110, pour éviter une perte de synchronisation lorsqu'on trouve une longue chaîne de 1.

Le protocole MNP7 utilise essentiellement la même technique sauf qu'au lieu d'utiliser une unique file MTF, elle utilise 256 files, chacune associée au symbole précédent dans la séquence, donc conditionnée par celui-ci. Cela revient à modéliser $P(x_t \mid x_{t-1})$ grâce à une file ordonnée qui dépend de x_{t-1}. Alors que les codes pour le protocole MNP5 ont une longueur minimale de quatre bits, les codes pour le protocole MNP7 ont une longueur minimale de seulement un bit,

donnant ainsi la possibilité d'atteindre, localement du moins, un ratio de compression de 8 : 1 alors que MNP5 était limité à 2 : 1.

Le protocole Microsoft pour la compression des paquets PPP est basée sur une variante de LZ77 qui utilise une fenêtre coulissante de 8192 octets [156]. Le protocole MPPC (*Microsoft Point to Point Compression*) utilise une astuce qui permet d'utiliser la même structure de code pour les littéraux et les positions, les positions introduisant obligatoirement un code de longueur. Le code pour les littéraux et les positions est donné par :

Code	Représente...
$0 \mid x_6 \cdots x_0$	littéraux, 0 – 127
$1 \quad 0 \mid x_6 \cdots x_0$	littéraux, 128 – 255
$1 \quad 1 \quad 0 \mid x_{12} \cdots x_0$	positions, 320 – 8191
$1 \quad 1 \quad 1 \quad 0 \mid x_7 \cdots x_0$	positions, 64 – 319
$1 \quad 1 \quad 1 \quad 1 \mid x_5 \cdots x_0$	positions, 0 – 63

et les codes pour les longueurs sont :

Code	Intervalle représenté
0	3
$1 \quad 0 \mid x_1 x_0$	4 – 7
$1 \quad 1 \quad 0 \mid x_2 x_1 x_0$	8 – 15
$1 \quad 1 \quad 1 \quad 0 \mid x_3 x_2 x_1 x_0$	16 – 31
⋮	⋮
$1 \quad 1 \quad \cdots \quad 1 \quad 0 \mid x_{10} \cdots x_0$	4096 – 8191

où le dernier préfixe a une longueur de douze bits. Dans ce code, il n'est pas clair pourquoi il existe deux séries de codes pour littéraux. Si on fait l'hypothèse que l'on compresse du texte, et qui plus est, en anglais, il est tout à fait raisonnable d'avoir deux séries de codes pour les octets littéraux de 0 à 127 et un autre de 128 à 255. Les codes pour les lettres accentuées et les symboles spéciaux se trouvent en effet dans les plages de 128 à 255 ; les codes exacts dépendant

du code de caractères utilisé sur le système considéré.

Friend et Monsour présentent une modification de l'algorithme de Storer et Syzmanski [75, 199] où les codes des littéraux et des tuples de position et de longueur sont différenciés par un bit, comme pour LZSS (section 4.2.1.1). Les codes pour les littéraux et les positions sont donnés par :

Code	Représente...
0 | $x_7 \cdots x_0$	Littéraux, 0 – 255
1 0 | $x_6 \cdots x_0$	Positions, 0 – 127
1 1 | $x_{10} \cdots x_0$	Positions, 128 – 2195

et les codes pour les longueurs ressemblent aux codes universels :

2	00
3	01
4	10
5	11 00
6	11 01
7	11 10
8	11 11 0000
⋮	⋮
22	11 11 1110
23	11 11 1111 0000
⋮	⋮

Tous ces codes sont de longueur variable, produisant un code moyen plus court qu'un code naturel (c'est-à-dire une simple réprensation binaire du nombre), sans toutefois véritablement tenir compte de la distribution réelle des entiers qu'ils sont censés coder. Par exemple, les codes de Friend et Monsour pour les positions assument une distribution vaguement géométrique alors qu'il est fort à parier qu'une distribution exponentielle avec un mode significativement éloigné de zéro modélisera mieux la distribution des positions. On retrouve essentiellement le même problème avec les codes suggérés par Hall.

5.3.4.2 ★ Codes *phase-in*

Supposons que nous ayons à coder des nombres $0 \leqslant n < N$, tirés selon une loi uniforme. Un codage naturel, connaissant N, nécessiterait $\lceil \lg N \rceil$ bits, ce qui mènerait potentiellement au gaspillage de presqu'un bit complet, surtout lorsque $2^k < N \ll 2^{k+1}$. Il nous faudrait plutôt un code qui donne très près de $\lg N$ bits en moyenne.

Une façon d'y parvenir, c'est d'utiliser les *phase-in codes*. La décomposition unique $N = 2^k + b$ nous sera particulièrement utile. Soient donc $N = 2^k + b$, $k = \lfloor \lg N \rfloor$, et $0 \leqslant b < 2^k$. Posons

$\beta = 2^k - b$. Dans le code *phase-in* pour N, nous aurons β codes courts (de k bits) et $N - \beta$ codes longs (de $k+1$ bits), selon

$$C_{\phi(N)}(n) = \begin{cases} C_{\beta(k)}(n) & \text{si } n < \beta \\ C_{\beta(k)}(\beta + \left\lfloor \frac{n-\beta}{2} \right\rfloor) : C_{\beta(1)}((n-\beta) \bmod 2) & \text{si } \beta \leqslant n < N \end{cases}$$

où $C_{\beta(k)}(i)$ est la représentation binaire de i sur k bits.

Le tableau 5.4 donne des exemples des codes générés pour quelques N. Les dérivations pour la longueur moyenne et l'écart à la longueur idéale sont probablement des contributions originales car ni l'une ni l'autre ne se retrouve dans la littérature. Nous présentons donc ici ces dérivations comme des contributions. La longueur des codes est donnée par :

$$|C_{\phi(N)}(n)| = L_{\phi(N)}(n) = \begin{cases} k & \text{si } n < \beta \\ k+1 & \text{sinon} \end{cases} \tag{5.6}$$

où $k = \lfloor \lg N \rfloor$. En supposant que $n \sim X = \mathcal{U}(0, N-1)$, nous avons

$$\begin{aligned} \bar{L}_{\phi(N)}(X) &= P(X < \beta)k + P(X \geqslant \beta)(k+1) \\ &= P(X < \beta)k + (1 - P(X < \beta))(k+1) \\ &= k + 1 - P(X < \beta) \\ &= k + 1 - \frac{2^k - b}{2^k + b} \\ &= k + \frac{2b}{2^k + b} \end{aligned} \tag{5.7}$$

Les codes *phase-in* sont donc très efficaces lorsque $b \approx 0$ ou $b \approx 2^k$, et gâchent le plus de bits lorsque $b \approx (2\ln 2 - 1)2^k$, soit $b \approx 2^k\, 0.386294$. Pour dériver ce résultat, on a que

$$\bar{L}_{\phi(N)}(X) - \lg N = k + \frac{2b}{2^k + b} - \lg(2^k + b)$$

est convexe en b et en résolvant

$$\frac{\partial}{\partial b}\left(k + \frac{2b}{2^k + b} - \lg(2^k + b)\right) = \frac{(2\ln 2 - 1)2^k - b}{(2^k + b)^2 \ln 2} = 0$$

pour b, on trouve $b = (2\ln 2 - 1)2^k \approx 2^k\, 0.386294$. En posant $b = (2\ln 2 - 1)2^k$, ce qui est le pire cas, on trouve que

$$\begin{aligned} \bar{L}_{\phi(N)}(X) - \lg N &= \frac{1}{\ln 2}\left(k\ln 2 + 2\ln 2 - \ln(2^{k+1}\ln 2) - 1\right) \\ &= \frac{1}{\ln 2}\left(k\ln 2 + 2\ln 2 - \left(\ln 2^{k+1} + \ln\ln 2\right) - 1\right) \\ &= \frac{1}{\ln 2}\left((k+2)\ln 2 - (k+1)\ln 2 - \ln\ln 2 - 1\right) \\ &= \frac{1}{\ln 2}\left(\ln 2 - \ln\ln 2 - 1\right) \\ &\approx 0.0860713\ldots \end{aligned} \tag{5.8}$$

n	$C_{\phi(11)}(n)$	n	$C_{\phi(13)}(n)$	n	$C_{\phi(18)}(n)$
0	000	0	000	0	0000
1	001	1	001	1	0001
2	010	2	010	2	0010
3	011	3	0110	3	0011
4	100	4	0111	4	0100
5	1010	5	1000	5	0101
6	1011	6	1001	...	...
7	1100	7	1010	12	1100
8	1101	8	1011	13	1101
9	1110	9	1100	14	11100
10	1111	10	1101	15	11101
		11	1110	16	11110
		12	1111	17	11111

TAB. 5.4 – Codes *phase-in* pour quelques N.

cela nous donne

$$\bar{L}_{\phi(N)}(X) - \lg N \leqslant 0.0860713\ldots$$

Les codes *phase-in* sont effectivement très efficaces ! — sous l'hypothèse que les entiers sont effectivement tirés selon une loi uniforme. Acharya et Já Já réfèrent aux codes *phase-in* sous le nom de *economy codes* [1, 2]. C'est l'une des améliorations qu'ils proposent au codage des index dans l'algorithme de compression LZW. L'autre amélioration qu'ils proposent fait l'objet de la section suivante, la section 5.3.4.3.

❁ ❁ ❁

Ces résultats ont été soumis aux *Information Processing Letters*, et sont en attente de revue.

5.3.4.3 Codes récursivement *phase-in*

Acharya et Já Já présentent une méthode pour aider la compression LZW, où le codage naturel des index est remplacé par des codes récursivement *phase-in* [1, 2]. La méthode originale proposée pour générer ces codes demande la maintenance d'un arbre et de nombreuses manipulations. Heureusement, nous présentons ici un algorithme qui génère ces codes de façon beaucoup plus simple[3].

3 Il s'agit d'une redécouverte que j'ai fait alors que je cherchais des codes systématiques significativement plus courts pour les petits nombres que pour les grands. La formulation récursive est cependant beaucoup plus simple que le mécanisme présenté par Acharya et Já Já.

Comme pour les codes simplement *phase-in*, supposons que $N = 2^k + b$, avec les mêmes contraintes, soit $k = \lfloor \lg N \rfloor$ et $0 \leqslant b < 2^k$. Les codes sont définis récursivement :

$$C_{\rho(N)}(n) = \begin{cases} 0 : C_{\rho(b)}(n) & \text{si } 0 \leqslant n < b \\ C_{\beta(k)}(n) & \text{si } b = 0 \\ 1 : C_{\beta(k)}(n-b) & \text{sinon} \end{cases}$$

La longueur du code pour n est

$$L_{\rho(N)}(n) = \begin{cases} 1 + L_{\rho(b)}(n) & \text{si } 0 < n < b \\ k & \text{si } b = 0 \\ 1 + k & \text{sinon} \end{cases}$$

La longueur moyenne pour les codes récursivement *phase-in* est plus compliquée à obtenir que pour les codes *phase-in*. Pour obtenir la longueur moyenne, nous allons commencer par compter le nombre de bits de tous les codes. Le nombre total de bits est donné par

$$S_{\rho(N)} = \begin{cases} 2^k k & \text{si } b = 0 \\ 2^k(k+1) + b + S_{\rho(b)} & \text{sinon} \end{cases}$$

Considérons les deux cas. Dans le premier cas, $N = 2^k$ et seulement k bits sont nécessaires par code, lesquels sont au nombre de 2^k. Dans le second cas, $N = 2^k + b$. D'après la structure du code, il y aura 2^k codes de k bits, et b de $k-1$ bits. Comme chaque classe de bits reçoit un bit de préfixe, soit 0 pour les codes courts et 1 pour les codes longs, les codes longs font maintenant $k+1$ bits et étant au nombre de 2^k contribuent $2^k(k+1)$ bits. Les codes courts sont au nombre de b, mais comme on les définit récursivement, on ne compte que les bits de préfixe, qui comptent pour b bits. La longueur moyenne, sous hypothèse de tirage uniforme est alors donnée par

$$\bar{L}_{\rho(n)} = \frac{1}{N} S_{\rho(N)} \tag{5.9}$$

La forme analytique de l'éq. (5.9) est beaucoup moins élégante que la formule de la longueur moyenne des codes *phase-in*, l'éq. (5.7). En considérant la fig. 5.1, on voit que la longueur moyenne est très chaotique. C'est que la longueur moyenne dépend de la décomposition binaire de N. La décomposition binaire de N a $k \leqslant \lfloor \lg n \rfloor < k+1$ bits, et si beaucoup de ces bits sont à un, le code est beaucoup plus long qu'il devrait l'être considérant simplement N.

Les codes générés sont montrés au tableau 5.5 et comparés aux codes *phase-in* à la fig. 5.1. Ces codes performent beaucoup moins bien que les simples codes *phase-in* si les nombres sont effectivement tirés selon une loi uniforme. On voit en comparant les tableaux 5.4 et 5.5 que les codes récursifs croissent plus rapidement. Cependant, si on suppose que la distribution est biaisée vers les petits nombres, les codes récursivement *phase-in* sont potentiellement nettement meilleurs.

En fait, dans l'article d'Acharya et Já Já, les codes sont inversés dans la mesure où ce sont les grands nombres qui reçoivent les codes les plus courts. Cela est parfaitement raisonnable puisque dans l'algorithme LZW qu'ils se proposent d'améliorer, les index qui sont les plus susceptibles d'être utilisés correspondent aux dernières concordances trouvées, lesquelles viennent

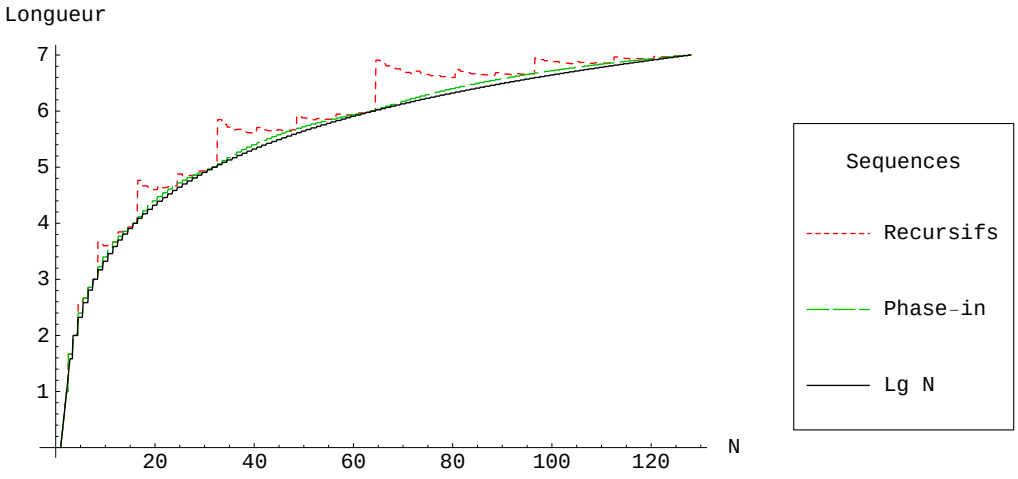


FIG. 5.1 – Comparaisons des longueurs moyennes des codes *phase-in* et récursivement *phase-in* contre la fonction $\lg N$. Les codes *phase-in* collent la fonction $\lg N$ de près alors que les codes récursivements définis sont beaucoup plus chaotiques et leur performance beaucoup moins bonne. Le graphique montre des N de 1 à 128 ; la longueur moyenne est donnée en bits.

d'être ajoutées au dictionnaire et ont reçu par conséquent un index près de la fin du dictionnaire. On peut transformer trivialement les codes récursifs présentés ici en codes qui assignent des codes courts aux grands entiers : $C'_{\rho(N)}(i) = C_{\rho(N)}(N-1-i)$.

5.3.4.4 Codes énumératifs

Les codes énumératifs ont un usage bien particulier. Alors que tous les codes que nous avons présentés jusqu'à maintenant permettent d'encoder un nombre entier, possiblement contraint à $0 \leqslant n < N$, les codes énumératifs serviront à encoder un choix combinatoire, c'est-à-dire l'un des $\binom{n}{m}$ choix possibles lorsque exactement m éléments parmi n sont choisis. Ces codes peuvent être utilisés pour encoder des tirages ou des combinaisons de jeu de loto, ou encore pour déterminer les cases occupées au jeu d'échecs ou de dames.

La méthode d'encodage des choix est présenté par Schalkwijk et on la retrouve dans l'article de Cover qui l'utilise comme substitut au codage arithmétique [185, 58]. L'algorithme serait cependant dû à Lehmer [125].

Supposons que nous connaissions n et m, et que $0 \leqslant m \leqslant n$, de façon à ce que $\binom{n}{m}$ soit défini normalement. Soit $t_{\mathbf{2}} = \{t_1, t_2, \ldots, t_k\}$, la représentation binaire de t. Le code pour t, étant donné n, est

$$C_{\binom{n}{m}}(t) = \sum_{i=1}^{k} t_i \binom{n-i}{m - \sum_{j=1}^{i-1} t_j}$$

où $\binom{n-i}{m-\sum_{j=1}^{i-1} t_j}$ est le nombre de combinaisons de longueur $i-1$ avec $\sum_{j=1}^{i-1} t_j$ uns qui précèdent

n	$C_{\rho(11)}(n)$	n	$C_{\rho(13)}(n)$	n	$C_{\rho(18)}(n)$
0	00	0	00	0	00
1	010	1	0100	1	01
2	011	2	0101	2	10000
3	1000	3	0110	3	10001
4	1001	4	0111	4	10010
5	1010	5	1000	5	10011
6	1011	6	1001	...	...
7	1100	7	1010	12	11010
8	1101	8	1011	13	11011
9	1110	9	1100	14	11100
10	1111	10	1101	15	11101
		11	1110	16	11110
		12	1111	17	11111

TAB. 5.5 – Codes récursivement *phase-in* pour quelques N.

la combinaison courante. Nous avons aussi que

$$0 \leqslant C_{\binom{n}{m}}(t) \leqslant \binom{n}{m} - 1$$

ce qui permet d'avoir un entier de $\lceil \lg \binom{n}{m} \rceil$ bits de long, lequel peut être codé nûment puisque nous connaissons n et m et par conséquent sa longueur. Pour retrouver t, on utilise la décomposition

$$C^{-1}_{\binom{n}{m}}(i) = \begin{cases} \perp & \text{si } n = 0 \\ 1 : C^{-1}_{\binom{n-1}{m-1}}\left(i - \binom{n-1}{m}\right) & \text{si } i \geqslant \binom{n-1}{m} \\ 0 : C^{-1}_{\binom{n-1}{m}}(i) & \text{sinon} \end{cases}$$

Le tableau 5.6 montre un exemple de code. Outre le codage des choix m parmi n, on peut utiliser cette méthode comme substitut de codage arithmétique binaire, où la probabilité qu'un bit soit à un est contrainte à être exactement $\frac{m}{n}$ sur une séquence de longueur n.

5.3.5 Codes statistiques simples

Les deux codes que nous présentons dans cette dernière sous-section utilisent de l'information sur la distribution des entiers à coder pour ajuster leur paramètres et ainsi produire un code plus efficace. Les codes de Golomb et les codes $(start, step, stop)$ de Fiala et Greene en sont des exemples. Nous les présentons ici brièvement car il font l'objet de contributions au chapitre 7.

5.3.5.1 Codes de Golomb

L'article de Solomon W. Golomb, *Run-Length Encodings* [85], débute de façon plutôt humoristique :

$C_{\binom{6}{2}}(t)$		t
0		000011
1		000101
2		000110
3		001001
4		001010
5		001100
6		010001
7	$\leftrightarrow$	010010
8		010100
9		011000
10		100001
11		100010
12		100100
13		101000
14		110000

TAB. 5.6 – Un exemple de codage énumératif, pour $\binom{6}{2}$.

Secret Agent 00111 is back at the Casino again, playing a game of chance, while the fate of mankind hangs in the balance. Each game consists of a sequence of favorable events (probability p), terminated by the first occurence of an unfavorable event (probability $q = 1 - p$). More specifically, the game is roulette, and the unfavorable event is the occurence of 0, which has a probability of q=1/37. No one serioulsy doubts that 00111 will come through again, but the Secret Service is quite concerned about communicating the blow-by-blow description back to Whitehall.

The bartender, who is a free-lance agent, has a binary channel available, but he charges a stiff fee for each bit sent. The problem perplexing the Service is how to encode the vicissitudes of the wheel so as to place the least strain on the Royal Exchequer. It is easily seen that, for the case $p = q = \frac{1}{2}$, the best that can be done is to use 0 and 1 to represent the two possible outcomes. However, the case at hand involves $p \gg q$, for which the "direct coding" method is shockingly inefficient.

Le problème posé par l'agent 00111 est d'encoder la longueur de la série d'évènements favorables. Le lecteur aura compris qu'il s'agit d'encoder un entier avec une fonction de distribution géométrique de paramètre p, laquelle est donnée par $P(X = i) = (1-p)^i p$, pour $i \in \mathbb{Z}^*$ — notre p étant le q de Golomb.

Nous reviendrons, à la section 7.3, sur les détails. Golomb propose d'approximer la distribution par une série d'intervalles de taille égale, distribués de façon à ce que le i e intervalle ait une probabilité $\approx \left(\frac{1}{2}\right)^{i+1}$. Cela permet d'encoder le numéro du plateau de façon unaire et de faire suivre le préfixe par l'index du nombre dans le plateau désigné. Golomb suggère d'utiliser des intervalles contenant b nombres, et b est obtenu grâce à la solution (approximative) $b = \lceil -\lg p \rceil$.

Ayant b, on calcule, pour un nombre $n \geqslant 0$ à coder, le paramètre

$$d = \left\lfloor \frac{n}{b} \right\rfloor$$

qui est utilisé pour générer le code

$$C_{G(p)}(n) = C_\alpha(d) : C_{\beta(b)}(n \bmod 2^b)$$

Cette solution fonctionne le mieux lorsque $p \sim 2^{-l}$ mais est généralement sous-optimale. Gallager et van Voorhis proposent de trouver $m \in \mathbb{Z}^*$ tel que m satisfasse $(1-p)^m + (1-p)^{m+1} \leqslant 1 < (1-p)^{m-1} + (1-p)^m$ et d'utiliser m plutôt que 2^b pour générer le code [77]. Nous montrerons, à la section 7.3, que la solution de Gallager et van Voorhis est meilleure que la solution originale de Golomb, mais n'est cependant pas encore optimale. Il existe en effet une meilleure façon d'attaquer le problème : il s'agit de minimiser directement la longueur moyenne du code pour un paramètre b, soit

$$\begin{aligned} \bar{L}_{G(p)(X)} &= \sum_{i=0}^{\infty} P(X = i) L_{G(p)}(i) \\ &= \sum_{i=0}^{\infty} P(X = i)(1 + b + \lfloor \frac{i}{2^b} \rfloor) \end{aligned}$$

sans faire intervenir de notions asymptotiques.

5.3.5.2 Codes $(Start, Step, Stop)$

Ces codes nous sont dûs à Fiala et Greene [70]. Ces codes, utilisés pour encoder les positions et les longueurs dans LZFG (voir section 4.2.1.2), sont contrôlés par trois paramètres, *Start*, *Step* et *Stop*. Le paramètre *Stop* est obligatoirement de la forme $Start + k_{max} Step$, ce qui fait que l'on peut réduire la paramétrisation du code à $(Start, Step, k_{max})$. Le code est composé d'un préfixe qui encode, en unaire tronqué, k, le suffixe étant formé de $Start + k\ step$ bits. Comme on connaît k_{max}, on peut utiliser un code unaire tronqué, où les deux derniers codes sont de la même longueur. Un codage unaire tronqué à la longueur N est donné par :

$$C_{\hat{\alpha}(N)}(i) = \begin{cases} C_\alpha(i) & \text{si } i < N \\ C_{\hat{\alpha}}(i) & \text{si } i = N \end{cases}$$

où $C_\alpha(i)$ est le codage unaire de base et $C_{\hat{\alpha}}(i)$ est le codage unaire sans le dernier bit. Par exemple, un code (3,2,9) a la forme :

Code	Nombres représentés
0 | x_2 x_1 x_0	0 – 7
1 0 | x_4 x_3 x_2 x_1 x_0	8 – 39
1 1 0 | x_6 x_5 x_4 x_3 x_2 x_1 x_0	40 – 167
1 1 1 | x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0	168 – 679

Ce dernier exemple est tiré de Fiala et Greene [70]. Le nombre de codes distincts pour un code $(Start, Step, Stop) = (Start, Step, k_{max})$ est donné par

$$\begin{aligned} S_{(Start,Step,k_{max})} &= \sum_{j=0}^{k_{max}} 2^{Start+j\,step} \\ &= \frac{2^{Start+(k_{max}+1)Step} - 2^{Start}}{2^{Step}-1} \end{aligned}$$

Ce résultat est obtenu grâce à une série géométrique : posant $a = 2^{Start}$, $r = 2^{Step}$, on a

$$\begin{aligned} \sum_{j=0}^{k_{max}} 2^{Start+j\,Step} &= \sum_{j=0}^{k_{max}} \left(2^{Start}\right)\left(2^{Step}\right)^{j} \\ &= \sum_{j=0}^{k_{max}} a\, r^{j} \\ &= \frac{a\, r^{k_{max}+1} - a}{r-1} \\ &= \frac{2^{Start+(k_{max}+1)Step} - 2^{Start}}{2^{Step}-1} \end{aligned}$$

La longueur moyenne du code dépendra de la fonction de distribution des nombres à coder. Puisque les codes sont biaisés vers les codes courts, on peut supposer que la distribution est strictement non-croissante. Pour calculer la longueur moyenne, nous aurons recours à la forme de l'éq. (5.2), à savoir

$$\begin{aligned} \bar{L}_{(Start,Step,k_{max})}(X) &= \sum_{k=0}^{k_{max}} P(l(k) \leqslant X < h(k)) L_{(Start,Step,k_{max})}(x \mid k) \\ &= \sum_{k=0}^{k_{max}} P(l(k) \leqslant X < h(k)) \left(\min(k_{max}, k+1) + Start + k\, Step\right) \end{aligned}$$

où $l(0) = 0$ et

$$l(k+1) = h(k) = \frac{2^{Start+(k+1)Step} - 2^{Start}}{2^{Step}-1}$$

Fiala et Greene ne présentent pas d'algorithme explicite pour optimiser les paramètres d'un code $(Start, Step, Stop)$ à partir de la distribution de probabilité. De par la forme du code, nous voyons que ces codes correspondent le mieux aux distributions strictement décroissantes, de type exponentiel ou géométrique. À la section 7.5.1, nous présenterons un algorithme pour optimiser ces codes ainsi que la distribution qu'ils représentent parfaitement.

Les codes $(Start, Stop, Step)$ peuvent être utilisés pour émuler d'autres types de codes, par exemple un code $(q, 0, q)$ correspond à $C_{\beta(q)}$, un code $(0, 0, \infty)$ au code unaire C_α, etc. On peut même les utiliser comme codes universels, il suffit d'avoir un code (a, b, ∞), avec $a \geqslant 0$ et $b \geqslant 1$.

5.3.6 Codes sur $\mathbb{Z}$

Les codes que nous avons présentés jusqu'à maintenant se contentent de coder des nombres naturels, ou peut-être des entiers non-négatifs. Cependant, nous sommes susceptibles d'avoir à

coder des nombres négatifs et les codes ne sont pas tous conçus pour nous le permettre ; dans cette sous-section, nous présentons quelques astuces pour transformer les entiers $\mathbb{Z}$ en entiers non-négatifs, soit $\mathbb{Z}^*$. Nous présenterons aussi quelques fonctions de pairage qui permettent de fusionner de façon réversible deux entiers ou plus.

5.3.6.1 Replis de $\mathbb{Z}$ sur $\mathbb{Z}^*$

La solution naturelle et tentante, c'est d'utiliser tout simplement un bit de signe que l'on colle devant le code pour un entier. Cette solution est simple mais provoque un effet secondaire qui peut être indésirable : deux représentations pour le zéro ! On gaspille donc un code, en plus d'introduire une redondance inutile dans les autres codes ! Il faut donc trouver un moyen de replier (*fold*) les entiers sur les entiers, de façon à n'avoir qu'une seule représentation pour le zéro et à se débarasser du signe. Une fonction de repli possible est donnée par

$$f(n) = \begin{cases} 2|n| & \text{si } n \geqslant 0 \\ 2|n| + 1 & \text{sinon} \end{cases} \tag{5.10}$$

La fonction de repli est $F : \mathbb{Z} \to \mathbb{Z}^*$ (on se souviendra que $\mathbb{Z}^* = \{0, 1, 2, \dots\}$), mais est facilement modifiable pour avoir plutôt $F : \mathbb{Z} \to \mathbb{N}$, si nécessaire — il est en effet trivial de passer de $\mathbb{Z}^*$ à $\mathbb{N}$ ou vice versa en utilisant un biais systématique de $+1$ ou -1.

L'éq. (5.10) permet de ne représenter qu'un seul zéro, au prix d'un bit supplémentaire pour chaque autre code — ce qui est parfaitement acceptable, car zéro n'étant ni positif, ni négatif, n'a pas à recevoir de signe. Un autre problème lié au repli est la transformation de la fonction de distribution. S'il s'agit d'une distribution symétrique par rapport au mode, il suffit de ramener le mode à zéro (en appliquant un biais systématique qui correspond au mode) et de réajuster la fonction de distribution pour tenir compte du repli. Ce réajustement est plus ou moins compliqué dépendemment de la fonction de distribution. Si la fonction de distribution est clairement asymétrique, utiliser une fonction de repli n'est pas la solution idéale, car elle introduirait des aberrations dans la fonction de distribution repliée, ce qui nuirait grandement à la compression.

5.3.6.2 Fonctions de pairage

Nous avons discuté, à la section 3.1 des fonctions de pairage sans entrer dans le détail (voir en particulier la définition 3.1.6, page 25). Une fonction de pairage est une fonction qui prend en entrée deux entiers pour en former un nouveau, mais de façon réversible ; c'est-à-dire qu'à partir du résultat de la fonction de pairage, on peut retrouver les deux entiers originaux.

La première fonction de pairage que nous présentons, c'est la fonction de Cantor :

$$\langle\, i, j \,\rangle_{\text{Cantor}} = \frac{1}{2}(i + j)(i + j + 1) + j \tag{5.11}$$

Cette fonction, par le théorème de Fueter-Pólya, est la seule fonction quadratique à coefficients réels qui replie $\mathbb{Z}^* \times \mathbb{Z}^*$ sur $\mathbb{Z}^*$ [197, pages 448–452]. Cette fonction surgit naturellement lorsqu'on veut montrer qu'il existe un repli de $\mathbb{Q}$ sur $\mathbb{Z}^*$. Puisque $\mathbb{Q} \subseteq \{\, \frac{a}{b} \mid a \in \mathbb{Z}, b \in \mathbb{N} \,\}$, on commence par replier les $\mathbb{Z}$ sur $\mathbb{Z}^*$, grâce à la l'éq. (5.10) par exemple, puis on replie $\mathbb{Z}^* \times \mathbb{Z}^*$ sur $\mathbb{Z}^*$ grâce à la fonction de pairage de Cantor. Le résultat de cette fonction de pairage est montré au tableau 5.7.

Une variante de cette fonction de pairage, cette fois de $\mathbb{N} \times \mathbb{N}$ vers $\mathbb{N}$, est présentée dans Hopcroft et Ullman [99, p. 169]. Cette fonction est donnée par :

$$\langle\, i, j \,\rangle_{\mathrm{HU}} = \frac{1}{2}(i + j - 1)(i + j - 2) + i \tag{5.12}$$

où nous pouvons voir que

$$\begin{aligned} \langle\, i, j \,\rangle_{\mathrm{HU}} &= \frac{1}{2}(i + j - 1)(i + j - 2) + i \\ &= \Delta(i + j - 2) + i \end{aligned}$$

où $\Delta(n) = \frac{1}{2}n(n+1)$, est la somme des n premiers entiers. Nous trouvons que l'inverse de cette fonction est donné par :

$$\begin{aligned} h &= \langle\, i, j \,\rangle \\ c &= \lfloor \sqrt{2h} - \frac{1}{2} \rfloor \\ i &= h - \Delta(c) \\ j &= c - i + 2 \end{aligned}$$

Le repli donné par cette fonction de pairage est montré au tableau 5.8. Après un bref examen des tableaux 5.7 et 5.8, on constate qu'il s'agit, en fait, de la même fonction avec une paramétrisation différente. On trouve l'identité :

$$\langle\, i, j \,\rangle_{\mathrm{Cantor}} = \langle\, i + 1, j + 1 \,\rangle_{\mathrm{HU}} - 1$$

Cette équation suggère d'utiliser l'inverse de la fonction de pairage de Hopcroft et Ullman pour calculer l'inverse de la fonction de pairage de Cantor.

$$\begin{aligned} k &= \langle\, i, j \,\rangle_{\mathrm{Cantor}} + 1 \\ \{i', j'\} &= \langle\, h \,\rangle^{-1}_{\mathrm{HU}} \\ i &= i' - 1 \\ j &= j' - 1 \end{aligned}$$

Il existe des variantes boustrophédoniques[4] de la fonction de pairage de Cantor mais aucun bénéfice réel ne vient de leur utilisation. D'une part parce que la magnitude des entiers générés ne change pour ainsi dire pas (à peu près les mêmes nombres se retrouvent sur les mêmes diagonales et l'ordre est inversée une diagonale sur deux), d'autre part parce que la complexité des fonctions augmente significativement. La figure 5.2 montre deux variations boustrophédoniques.

Les fonctions de pairages permettent de composer deux entiers de façon simple. Les cas où nous devons « pairer » plus de deux entiers ne sont pas rares. La solution est toute simple, il suffit d'utiliser une composition de pairage. Cependant, pour minimiser la taille des entiers produits, nous devons minimiser la profondeur de composition. Ainsi, si $\langle\, i, j, k \,\rangle = \langle\, i, \langle\, j, k \,\rangle\rangle$, nous avons cependant $\langle\, i, j, k, l \,\rangle = \langle\, \langle\, i, j \,\rangle, \langle\, k, l \,\rangle\,\rangle$.

4 Le terme boustrophédonique fait allusion aux sillons faits par le bœuf (*bous*) en tournant (*strophein*) à chaque limite du champ. Ce terme désignait anciennement exclusivement une forme d'écriture archaïque (du grec et de l'étrusque), où les lignes allaient de gauche à droite, puis de droite à gauche sans interruption du texte.

⋮	⋮						
5	15	22	30	39	49	60	
4	10	16	23	31	40	50	
3	6	11	17	24	32	41	
2	3	7	12	18	25	33	
1	1	4	8	13	19	26	
0	0	2	5	9	14	20	···
	0	1	2	3	4	5	···

TAB. 5.7 – La fonction de pairage de Cantor.

⋮	⋮					
5	15	20	26	33	41	
4	10	14	19	25	32	
3	6	9	13	18	24	
2	3	5	8	12	17	
1	1	2	4	7	11	···
	1	2	3	4	5	···

TAB. 5.8 – La fonction de pairage de Hopcroft et Ullman.

⋮					
11	20	24	33	41	
10	12	19	25	32	
4	9	13	18	26	
3	5	8	14	17	
1	2	6	7	15	···

a)

⋮					
17	18	19	20	21	
16	15	14	13	22	
5	6	7	12	23	
4	3	8	11	24	
1	2	9	10	25	···

b)

FIG. 5.2 – Deux variantes boustrophédoniques de la fonction de pairage de Cantor. En a), la variante classique, où le sens des diagonales alterne et en b), la variante de Stein [197].

⋮	⋮								
7	42	43	46	47	58	59	62	63	
6	40	41	44	45	56	57	60	61	
5	34	35	38	39	50	51	54	55	
4	32	33	36	37	48	49	52	53	
3	10	11	14	15	26	27	30	31	
2	8	9	12	13	24	25	28	29	
1	2	3	6	7	18	19	22	23	
0	0	1	4	5	16	17	20	21	···
	0	1	2	3	4	5	6	7	···

TAB. 5.9 – La fonction de pairage proposée.

5.3.6.3 ★ Contribution aux fonctions de pairage

Je propose ici une nouvelle fonction de pairage. Cette fonction est très simple à calculer grâce à un ordinateur et ne requiert aucune opération arithmétique sauf les décalages et le ou inclusif arithmétique, qui sont beaucoup moins dispendieux que les multiplications et les extractions de racines carrées.

Soient $i_{\mathbf{2}} = i_{k-1}i_{k-2}\cdots i_2 i_1 i_0$ les bits de i, jusqu'au bit le plus significatif, et $j_{\mathbf{2}} = j_{l-1}j_{l-2}\cdots j_2 j_1 j_0$ les bits de j ; on suppose donc que $\lceil \lg i \rceil = k$ et $\lceil \lg j \rceil = l$.

La fonction de pairage est donnée par :

$$\langle\, i, j \,\rangle_{\mathrm{SP}} = \begin{cases} \bot & \text{si } i = 0 \text{ et } j = 0 \\ i_0 : j_0 : \langle\, \lfloor i/2 \rfloor, \lfloor j/2 \rfloor \,\rangle_{\mathrm{SP}}, & \text{sinon} \end{cases} \tag{5.13}$$

Cette formulation récursive ne sert que la simplicité de l'exposé ; un programme générera le pairage itérativement. Notons que la division $\lfloor i/2 \rfloor$ ne requiert pas de division ; en effet nous avons que $\lfloor i/2 \rfloor = i_1^{k-1}$, soit simplement i dont on laisse tomber le bit de poids faible, ce qui est calculé efficacement par un registre à décalage. Pour des entiers de longueur fixe (par exemple 32 bits), on pourrait fort bien considérer un circuit sans portes logiques, uniquement composé de câblage, pour réorganiser les bits de deux registres source vers un registre destination, ce qui s'implémente fort bien en FPGA ou encore en VLSI.

Cette fonction de pairage a la fonction de longueur

$$|\, \langle\, i, j \,\rangle_{SP} \,| = 2 \lg \max(i, j)$$

puisque celle-ci est dominée par le plus grand des deux entiers.

Le tableau 5.9 montre le pairage obtenu. S'il est difficile de le voir à partir du tableau, ce pairage induit une marche sur $\mathbb{Z}^* \times \mathbb{Z}^*$ qui couvre le plan. Comme pour la courbe de Hilbert, cette « courbe » couvre éventuellemenet $[0, 1] \times [0, 1]$ lorsque le nombre de points tend vers l'infini. La fig. 5.3 montre le résultat de raffinements successifs de la « fractale » générée par la fonction de pairage.

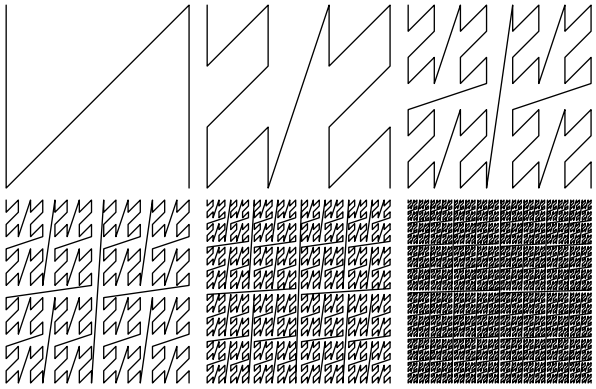


FIG. 5.3 – La couverture du plan induite par la fonction de pairage proposée. À chaque étape, le nombre de points est multiplié par quatre, soit 4, 16, 64, 256, 1024 et 4096. La limite de ce processus couvre la région du plan $[0,1] \times [0,1]$.

5.4 Notes bibliographiques

Le codeur arithmétique a été introduit il y a déjà un moment [12]. Il en existe plusieurs variantes, dont la plupart sont malheureusement protégées par des brevets, ce qui les rends difficilement utilisables. Par contre, certaines des variantes, comme le codeur ELS et le Z-Coder sont, sinon de domaine public, dotées de licences d'utilisation libre [20].

Quant aux codes de Karlgren on trouvera d'autres dérivations dans *Data Structure Techniques*, de Standish [196]. Les codes de Stout ont été modifiés par Yamamoto [229]. Plutôt que d'utiliser un bit pour indiquer si le bloc courant est le dernier ou s'il reste des blocs à lire, Yamamoto utilise des préfixes interdits pour récupérer le bit délimiteur, ce qui fait que plutôt que de perdre un bit, on perd seulement une petite fraction de bit, ce qui donne des codes d'une longueur moyenne encore plus près du minimum théorique de $\lg n + \lg\lg n + \cdots$

On sait peu de choses de Leonardo Pisano, dit Fibonacci. Les données biographiques sont rares et les données autobiographiques encore plus. Dans la préface à la seconde édition de 1228, après la dédicace à Michael Scott, astrologue à la cour de Frédérick II, empereur du Saint Empire Romain, on trouve un court paragraphe où Pisano explique ce qui l'a mené à l'étude des algorismes et des chiffres de l'Hindus. Fibonacci est tellement émerveillé par la relative simplicité des calculs fait grâce aux chiffres indiens qu'il se demande — sacrilège s'il en est un ! — si les méthodes traditionnelles depuis Pythagore ne sont pas simplement une erreur de la nature [88]. La première édition du *Liber Abaci* de 1202 est malheureusement perdue. La seconde édition de 1228, revue, corrigée et augmentée, fait donc authorité sur le sujet [168]. Cette édition est d'ailleurs reproduite intégralement par Boncompagni (pendant les années 1854 à 1857) qui est sans doute l'auteur qui a le plus écrit sur Fibonacci [16, 18, 17]. En plus d'une réédition complète de l'original en latin, on y trouve des commentaires et une traduction à l'italien. Plus près de nous, la *International Conference on Fibonacci Numbers and their Applications* se tient chaque année sur les applications des nombres de Fibonacci. La revue *Fibonacci Quarterly* est sans doute *la* revue mathématique où trouver tout ce qui touche de près ou de loin aux nombres de Fibonacci et les séquences du même genre.

Bien que l'on tienne souvent Fibonacci pour l'un des grands responsables de la popularisation des chiffres arabes en occident, il n'en est peut-être pas ainsi car dans son autobiographie, Pisano mentionne que ces chiffres sont déjà utilisés largement par les marchands en Égypte, en Syrie, en Grèce, en Sicile et... en Provence ! Peut-être peut-on y voir la progression des chiffres arabes depuis la première édition de son livre, vingt six ans plus tôt. Sur la progression des chiffres arabes dans le monde occidental, la monumentale *Histoire Universelle des Chiffres* de Georges Ifrah offre quelques chapitres sur le sujet, en particulier le chapitre 26 [103].

Le calcul rapide des nombres de Fibonacci est souvent nécessaire et Rocach propose quelques solutions [175]. Nous proposerons notre solution à ce problème à la section 7.4.2.4. Pour une introduction générale aux nombres de Fibonacci et au théorème de Zeckendorf (énoncés et preuves), voir [98]. Le livre de Graham, Knuth et Patashnik *Concrete Mathematics : A Foundation for Computer Science* a un chapitre complet sur les nombres spéciaux, comme les nombres de Stirling des deux espèces et les nombres de Fibonacci [87]. On y retrouve par exemple une dérivation de l'identité $F_n = [\phi^n/\sqrt{5}]$ qui nous a été utile pour résoudre la fonction Fibolog(n). Nous aurions aussi bien pu utiliser la formule de Binet $F_n = \frac{1}{\sqrt{5}}(\phi^n - (-\phi^{-1})^n) = \frac{1}{\sqrt{5}}(\phi^n - (-1^n)\phi^{-n})$, qui donne aussi exactement le n^{e} nombre de Fibonacci et où le terme en ϕ^{-n} devient rapidement négligeable lorsque n croît, rendant aussi possible une analyse asymptotique.

Le nombre d'or est noté par ϕ en l'honneur du sculpteur et architecte grec Phidias (circa 490 – 430 av. J. C.), à qui l'on doit le parthénon et sa statue d'Athéna ainsi que l'une des sept merveilles du monde antique, la statue colossale de Zeus à Olympie. Phidias aurait utilisé sciemment cette auguste proportion dans ses œuvres pour les rendre agréables et harmonieuses. La façade du parthénon à Athènes est effectivment un rectangle dont le ratio hauteur/longueur est approximativement ϕ.

Les dérivations de longueur des codes *phase-in*, section 5.3.4.2, autant que j'aie pu vérifier, sont des contributions originales malgré leur relative simplicité. La documentation sur les codes *phase-in* et récursivement *phase-in*, outre les articles Acharya et Já Já, est essentiellement anecdotique.

Tenkasi Ramabadran, propose de coder les combinaisons $\binom{n}{m}$ grâce au codage arithmétique [172]. Il suffit en effet d'ajuster les probabilités de 0 et de 1 en fonction de n et m ; c'est-à-dire poser $P(X = 1) = \frac{m}{n}$ et de lancer le codeur arithmétique. À chaque fois qu'un 1 est codé, les nouvelles probabilités deviennent $P(X = 1) = \frac{m-i}{n-t}$, où i est le nombre de uns déjà émis et t l'étape de codage. L'article de D. H. Lehmer remonte aux années 1960 [125], mais dû à la nature combinatoire du problème, il ne serait pas étonnant de trouver une solution dûe à (l'un des) Bernoulli ou encore à Faulhaber !

Le lien entre les fonctions de pairage et les fractales est connu depuis longtemps. La fonction de Hilbert, dite la « courbe » de Hilbert remonte à 1891 [96]. Il existe aussi un lien entre la courbe de Hilbert et la décomposition binaire des nombres formant les coordonnées qu'elle visite [220]. La courbe de Hilbert n'est pas sans rappeler la courbe de Peano dont la limite, elle aussi, remplit le plan.

Chapitre 6

Codage Huffman

6.1 Introduction

Les codes peuvent être catégorisés en fonction de la généralité de la distribution qu'ils sont censés coder. Le codage universel assume seulement que la distribution est éventuellement non-croissante, les codes de Golomb sont conçus pour les distributions géométriques de paramètre $p \leqslant \frac{1}{2}$ [85], et les codes $(start, step, stop)$ supposent une distribution uniforme par parties, chaque partie étant distribuée exponentiellement selon $p \approx \frac{1}{2}$ [70]. Les codes de Huffman, qui font l'objet de tout ce chapitre, sont plus généraux que les autres codes que nous avons présenté jusqu'à maintenant, car rien de particulier n'est supposé au sujet de la distribution, sauf peut-être que tout symbole a une probabilité non nulle d'être observé. Cette généralité permet d'utiliser l'algorithme de Huffman non seulement pour une classe précise de fonctions de distributions, mais pour toute distribution, y compris celles où il n'y a pas de dépendence évidente entre la probabilité d'occurence et le numéro d'un symbole, comme c'est le cas pour le texte. Dans le texte, l'ordre alphabétique conventionnel, de a à z n'a aucune relation évidente avec la fréquence des lettres individuelles.

Il y a de nombreuses façon de gérer ce cas. On pourrait par exemple construire une fonction de permutation qui associe chaque numéro de symbole à son rang, nous donnant ainsi une distribution non-croissante. Une fois cette nouvelle distribution obtenue, on pourrait utiliser un des codes que nous avons déjà présentés pour coder les symboles. Il semble toutefois peu probable que la distribution obtenue par ce réordonnancement concorde bien avec l'une des distributions supposées par les codes plus simples, ce qui mènerait alors à une inefficacité au codage, sans compter le coût de la description de la fonction de permutation. Le premier à donner une procédure, exacte, optimale (sous certaines conditions) pour générer des codes à partir d'une distribution arbitraire, fut David Huffman (1925 – 2000), voir fig. 6.1 [101]. Dans ce chapitre, nous allons voir comment l'algorithme de Huffman produit des codes non seulement efficaces, mais optimaux. Nous présenterons quelques autres variations, comme les codes de Shannon-Fano, qui en fait précèdent historiquement les codes de Huffman. Nous verrons aussi quelques algorithmes adaptatifs qui permettent de changer les codes au fur et à mesure que les symboles sont observés et que leurs probabilités changent dans le temps.

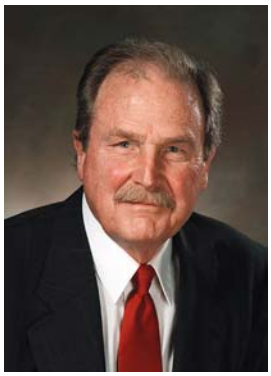


FIG. 6.1 – David Huffman (1925 – 2000). Photo de Don Harris © UCSC Public Information Office.

6.2 Les codes de Huffman

L'algorithme de Huffman produit des codes optimaux, dans la mesure où il produit des codes de longueur moyenne minimale, et bien qu'il puisse exister un certain nombre de codes équivalents, aucune procédure ne peut produire des codes plus courts en moyenne sous les mêmes conditions, à savoir que les codes individuels sont de longueur variable mais entière — et en faisant abstraction du coût de transmission de la description du code. La procédure de Huffman prévoit aussi le cas où nous avons plus de deux symboles de sortie : bien que nous ne soyons généralement concernés que par le cas binaire, où l'alphabet de sortie est $\mathbb{B} = \{0, 1\}$, nous pouvons nous trouver dans une situation où nous avons à notre disposition un nombre quelconque (mais toujours supérieur ou égal à deux) de symboles de sortie.

6.2.1 Codes Shannon-Fano

Shannon et Fano ont introduit séparément le même algorithme pour calculer un code efficace à partir d'une distribution arbitraire [69]. Cet algorithme est assez simple. Voici comment il procède. Les symboles sont d'abord triés en ordre décroissant de fréquence. Ce nouvel arrangement des codes formera la liste que nous utiliserons pour le reste de l'algorithme. Au début, tous les codes sont à $\perp$, c'est-à-dire initialisés comme étant la séquence vide. Ensuite, cette liste est divisée en deux parties (sans changer l'ordre des symboles) de façon à ce que la somme des fréquences dans la première partie soit la plus égale possible à la somme des fréquences de la seconde partie. À tous les codes des symboles de la première partie de la liste on ajoute 0 à la fin, et 1 à la fin de tous les codes des symboles de la seconde partie. On répète la même procédure, récursivement, sur les deux sous-listes, jusqu'à ce que nous atteignions des listes n'ayant qu'un élément.

Cette procédure nous fournit un code uniquement décodable qui a la caractéristique supplémentaire d'être structuré en arbre (*tree structured code*). Des codes résultants de l'algorithme sont montrés en exemple au tableau 6.1.

Il y a des problèmes avec cet algorithme. Un de ces problème est qu'il n'est pas toujours possible d'être certain où couper la liste en deux. Couper de façon à minimiser la différence

Symbole	Fréquence	Code
a	38416	00
b	32761	01
c	26896	100
d	14400	101
e	11881	1100
f	6724	1101
g	4225	1110
h	2705	1111

a)

Symbole	Fréquence	Code
a	34225	000
b	28224	001
c	27889	01
d	16900	100
e	14161	101
f	4624	110
g	2025	1110
h	324	1111

b)

TAB. 6.1 – Des exemples de codes créés par l'algorithme de Shannon-Fano. Les huit symboles sont tirés d'une distribution fictive et quelconque. En a), un code de longueur moyenne de 2.67 bits, alors que l'entropie pour cette même source est de 2.59 bits par symbole, laissant une inefficacité de 0.08 bits par symbole. En b), un exemple de code exhibant le problème relié à la coupe vorace de la liste. La longueur moyenne du code est de 2.8 bits par symbole alors que l'entropie n'est que de 2.5 bits par symboles, nous donnant une inefficacité de 0.3 bits par symboles ! C'est bien pire que l'inefficacité du code en a).

entre les sommes des fréquences des sous-listes ne donne pas toujours le code optimal. Un exemple de code exhibant ce défaut est montré au tableau 6.1, b). Bien que cet algorithme soit facile à implémenter, conceptuellement simple, soit en $O(n \lg n)$ pour n symboles, et qu'il donne en général des codes assez bons, il est toutefois significativement moins efficace en moyenne que l'algorithme de Huffman pour une même distribution.

6.2.2 Construction des codes de Huffman

Dans son article de 1952, Huffman donne la procédure pour créer les codes de longueur variable optimaux pour une distribution arbitraire et un alphabet de taille finie [101]. La procédure est bâtie à partir de quelques conditions qui, satisfaites, donnent des codes optimaux. Les premières conditions sont :

1. Le code est non-ambigu, c'est-à-dire qu'on ne peut le décoder que d'une seule façon.
2. Aucune information auxiliaire n'est nécessaire pour délimiter les codes.

La première condition exige que le code soit uniquement décodable (c.f. l'inégalité Kraft-McMillan, section 5.2.2), ne laissant ainsi aucune ambiguïté quant au message décodé. La deuxième condition exige qu'aucun délimiteur, ou symbole spécial, ne soit nécessaire — comme la pause dans le code Morse, par exemple — pour délimiter les codes entre eux.

Une façon de satisfaire les deux conditions d'un coup, c'est de bâtir un code structuré en arbre. Si les codes sont structurés en arbre, alors il existe un arbre binaire complet (*full*) avec tous les symboles sur les feuilles. Un arbre binaire est dit complet si tout nœud est soit une feuille, soit un nœud interne avec exactement deux fils. Le code pour un symbole est obtenu en descendant l'arbre depuis la racine jusqu'à la feuille qui contient le symbole, assignant un 0 lorsqu'on descend à gauche et un 1 lorsqu'on descend à droite. En atteignant une feuille, on

délimite naturellement le code, satisfaisant ainsi la deuxième condition. Serions-nous capables d'indiquer des nœuds internes, que nous aurions des codes qui seraient des préfixes d'autres codes, violant ainsi la première condition. La première condition est alors satisfaite car seules les feuilles de l'arbre sont accessibles.

Soit donc $A = \{a_1, a_2, \ldots, a_n\}$, l'alphabet pour lequel nous allons calculer le code. Supposons, sans perte de généralité, que l'alphabet A soit déjà réarrangé de façon à ce que les symboles soient énumérés en ordre décroissant de fréquence. Cela nous dispensera d'alourdir indûment la notation en utilisant partout une fonction de permutation $\pi(i)$ qui donne l'index du i^{e} symbole de A en ordre décroissant de fréquence. Cet alphabet satisfait donc

$$P(a_1) \geqslant P(a_2) \geqslant \cdots \geqslant P(a_n) \tag{6.1}$$

où $P(a)$ doit être compris comme $P(X = a)$, X étant la source aléatoire qui émet les symboles de A. Pour cette source et cet alphabet, nous voulons créer un ensemble de codes optimal avec des longueurs telles que

$$L(a_1) \leqslant L(a_2) \leqslant \cdots \leqslant L(a_n) \tag{6.2}$$

Il nous faudra modifier légèrement l'éq. (6.2). Soit L_n la longueur du plus long code. Supposons que nous ayons plus de deux codes de longueur L_n qui ne partagent pas de préfixes de longueur $L_n - 1$ entre eux. Puisque par hypothèse, nous avons donc déjà un code optimal, il n'y a pas d'autre codes qui sont des préfixes des codes de longueur L_n. Alors, puisque les codes ne partagent aucun préfixe entre eux, cela signifie que ces codes ont au moins un bit de trop. Alors L_n devrait être $L_n - 1$, une contradiction ! Cela signifie donc que nous pouvons laisser tomber le dernier bits de ces codes sans en affecter la décodabilité. Cela garantit que l'algorithme ne produira pas de codes plus longs que strictement nécessaire. De plus, on a pour conséquence qu'au moins deux codes sont de longueur L_{max}, et partagent un préfixe commun de longueur L_{max-1}. Enfin, l'éq. (6.2) devient

$$L(a_1) \leqslant L(a_2) \leqslant \cdots \leqslant L(a_{n-1}) = L(a_n) \tag{6.3}$$

où le dernier $\leqslant$ est devenu $=$. Nous avons maintenant les conditions suivantes à satisfaire :

1. Le code est uniquement décodable (non-ambigu).
2. Aucune information auxiliaire n'est nécessaire pour délimiter les codes.
3. $L(a_1) \leqslant L(a_2) \leqslant \cdots \leqslant L(a_{n-1}) = L(a_n)$,
4. Exactement deux codes de longueur L_{max} sont identiques sauf pour le dernier bit.
5. Tous les codes possibles de longueur $L_{max} - 1$ sont déjà utilisés ou ont un de leurs préfixes qui est un code.

De façon plutôt surprenante, un algorithme relativement simple permet de remplir toutes ces condtions. La condition la plus importante, c'est la condition n$^{\text{o}}$ 3. Bien que l'algorithme de Huffman puisse être utilisé avec un alphabet ayant un nombre quelconque (mais supérieur à un) de symboles de sortie, comme nous le verrons à la section 6.2.3, nous nous restreindrons ici au cas usuel, soit l'alphabet de sortie $\mathbb{B} = \{0, 1\}$.

L'algorithme procède de façon itérative. Le processus est illustré à la fig. 6.2. Au début, tous les symboles occupent un nœud qui est la racine de son propre sous-arbre. Mis à part le symbole et sa fréquence relative, le nœud contiendra des pointeurs vers ses fils de droite et de gauche.

Ils sont initialisés à nul, symbolisé ici par $\perp$. Tous ces nœuds-racines sont placés dans une liste L. La condition n° 3 exige que les deux symboles les moins fréquents reçoivent des codes de même longueur. Retirons de L les deux nœuds correspondant à ces deux symboles les moins fréquents. Soient a et b ces deux symboles. Nous créerons un nouveau nœud, c, n'ayant pas de symbole associé mais ayant une fréquence de $P(a) + P(b)$ et ayant pour fils les nœuds pour a et b. Nous ajoutons à L ce nouveau sous-arbre. Cela forcera a et b à partager un préfixe commun — le code qui se rendra jusqu'à c. Ainsi, L contient maintenant un sous-arbre de moins. Nous répéterons ce procédé jusqu'à ce que nous n'ayons plus qu'un seul sous-arbre. Ce sous-arbre respectera toutes les conditions énoncées par Huffman pour un code optimal. L'algorithme, en pseudo-code, ressemblera à :

$L = \{(a_1, P(a_1), \perp, \perp), (a_2, P(a_2), \perp, \perp), \ldots, (a_n, P(a_n), \perp, \perp)\}$
Tant que $|L| > 1$
{
  $a = \min_P L$
  $L = L - \{a\}$
  $b = \min_P L$
  $L = L - \{b\}$
  $c = (\perp, P(a) + P(b), b, a)$
  $L = L \cup \{c\}$
}

Dès lors, on dispose de l'arbre qui décrit le code, mais pas encore les codes eux-mêmes. Nous savons toutefois comment assigner les codes aux différents symboles, puisque nous avons déjà décrit la méthode quelques paragraphes plus haut. Il suffit de partir à la racine avec un code vide. À chaque fois que l'on descend l'arbre à gauche, on ajoutera 0 au code, et à chaque fois que l'on descend l'arbre à droite, on ajoutera 1 au code [1]. Atteindre une feuille délimite le code, et nous avons le code pour le symbole contenu dans cette feuille. Il ne reste plus qu'à construire un tableau, indexé par les symboles, qui contient les codes et leur longueurs.

Le processus de compression est direct. Il suffit d'émettre le code contenu dans le tableau, à l'index indiqué par le symbole à coder. Le décodage est à peine plus compliqué. Comme nous ne connaissons pas *a priori* la longueur du code, on ne peut lire directement n bits et décoder le symbole. Nous devrons descendre l'arbre, un bit à la fois, descendant à gauche si nous lisons un 0, descendant à droite si nous lisons un 1, jusqu'à ce que nous atteignions une feuille. La feuille contiendra le symbole décodé. Cette méthode de décodage peut être améliorée de nombreuses façons, comme nous le verrons à la section 6.5

6.2.3 Codes de Huffman N-aires

Bien que les codes binaires soient les plus courants, il y a des cas où nous avons à notre disposition plus de deux symboles de sortie. Par exemple, dans le contexte où nous utilisons un canal de communication électromécanique, nous avons souvent plus de deux valeurs possibles, et il serait bien de les prendre à notre avantage ! Les communications peuvent être rendues beaucoup plus efficaces lorsque nous avons $m \gg 2$ symboles de sortie à notre disposition. La première idée qui nous vient à l'esprit, c'est de généraliser la condition n° 3 pour grouper les m symboles les moins fréquents plutôt que seulement les deux moins fréquents. Bien que ça puisse ressembler à

1 Notons que cela est tout à fait arbitraire ; nous pourrions aussi bien assigner 1 à gauche et 0 à droite.

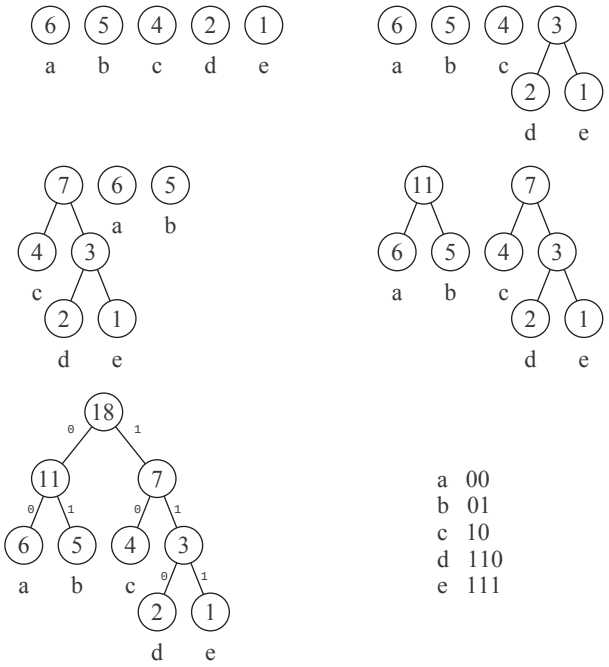


FIG. 6.2 – Comment l'algorithme de Huffman produit un code optimal. Au début, tous les nœuds sont les racines de leur propre sous-arbres. L'algorithme fusionne les sous-arbres ayant les plus petites fréquences en premier (notez que ce ne sont pas les probabilités mais les *fréquences* qui sont montrées dans les nœuds), et répète la procédure jusqu'à ce qu'il ne reste plus qu'un sous-arbre. Le code résultant, pour cet exemple, est donné par $\{a \to 00, b \to 01, c \to 10, d \to 110, e \to 111\}$, avec une longueur moyenne de 2.17 bits par symbole. L'entropie de la source est de 2.11 bits par symbole, faisant que le code a une inefficacité de 0.06 bits par symbole.

une bonne idée, nous voyons rapidement le problème. Supposons que nous ayons $n = 6$ symboles dans l'alphabet A et $m = 3$ symboles possibles en sortie. En appliquant l'algorithme de base modifié, c'est-à-dire en choisissant trois symboles plutôt que deux, après deux itérations, nous nous retrouvons avec seulement deux nœuds dans la liste! Ne pouvant donc choisir les trois nœuds les plus fréquents, l'algorithme casse et nous sommes bien embêtés.

Heureusement, il y a une modification relativement simple à l'algorithme de Huffman qui nous permettra de calculer des codes avec des $m \geqslant 2$ quelconques. Plutôt que de prendre les m nœuds avec les plus petites fréquences, nous en pendrons $2 \leqslant m' < m$. Pour être certain que toutes les autres fusions se feront avec exactement m sous-arbres, nous trouvons $a = n \bmod (m-1)$, puis $m' \equiv a \pmod{m-1}$. Puisque $2 \leqslant m' \leqslant m$, nous testons les valeurs possibles de m' jusqu'à ce que nous trouvions une valeur qui satisfasse $m' \equiv a \pmod{m-1}$.

Par exemple, pour $n = 6$, $m = 3$ on trouve $m' = 2$. Pour $n = 7$, $m = 3$, $m' = 3$. Pour $n = 8$, $m = 3$, nous trouvons $m' = 2$ encore. L'algorithme généralisé aura les nouvelles conditions :

3′. au moins deux, mais au plus m codes seront de longueur maximale,
4′. au moins deux, mais au plus m, codes de longueur L_{max} seront identiques, sauf pour le dernier symbole.

6.2.4 Codage canonique de Huffman

Le lecteur peut avoir remarqué que les codes produits par l'algorithme décrit à la section 6.2.2 sont en ordre numérique, lorsqu'on considère les séquences de bits comme des nombres. L'algorithme génère un code tel que si $P(a_i) \leqslant P(a_j)$, alors le code pour a_i précède numériquement le code pour a_j. Cela est fait parce que le sous-arbre le plus fréquent est systématiquement associé au fils de gauche, lequel reçoit le bit 0. Les symboles les plus fréquents reçoivent ainsi le plus petit code (considéré numériquement) encode disponible. Par cette convention, nous obtenons un code *canonique*, tel que $code(a_i) < code(a_j)$ ssi $P(a_i) \geqslant P(a_j)$.

La propriété de canonicité peut être utilisée pour créer des codeurs et décodeurs rapides, comme nous en verrons à la section 6.5. Si les codes sont assignés de façon à peu près aléatoire, il n'y a aucune façon de connaître la longueur du code sans l'avoir lu au complet, un bit à la fois. Si, au contraire, les codes sont bien assignés, on peut exploiter une régularité pour décoder rapidement. Par exemple, prenons le code obtenu à la fig. 6.2. Le code $\{00, 01, 10, 110, 111\}$ nous permet de lire deux bits d'un coup. Si, numériquement, ces deux bits sont inférieurs à 11, on sait qu'on a fini de lire le code et on retourne le symbole correspondant. Si, au contraire, on a lu 11, on sait qu'il nous reste un bit à lire. On a donc coupé le nombre moyen de comparaisons par $\frac{12}{7}$, ce qui n'est pas véritablement spectaculaire, mais dans des codes avec beaucoup plus de symboles, nous verrons plus de codes groupés de cette façon, et nous obtiendront potentiellement une plus grande accélération au décodage.

6.2.5 Performance des codes de Huffman

Montrons maintenant les bornes sur l'efficacité des codes de Huffman. Considérons d'abord la longueur moyenne des codes de Huffman pour une source X, dont nous supposerons connaître

la vraie distribution :

$$\bar{L}_H(X) = \sum\nolimits_{a_i \in A} P(X = a_i) L(a_i) \tag{6.4}$$

Nous savons, par les théorèmes Kraft-McMillan, que si les codes sont uniquement décodables, ils satisfont obligatoirement

$$\sum\nolimits_{a_i \in A} 2^{-L(a_i)} \leqslant 1. \tag{6.5}$$

Sachant cela, montrons que

$$\mathcal{H}(x) \leqslant \bar{L}_H(X) < \mathcal{H}(X) + 1 \tag{6.6}$$

où $\mathcal{H}(X)$ est l'entropie pour la source X. Nous montrerons que les deux bornes tiennent, grâce à une preuve en deux parties. La première partie de la preuve consistera à montrer que $\mathcal{H}(X) \leqslant \bar{L}_H(X)$, la seconde à démontrer que $\bar{L}_H(X) < \mathcal{H}(X) + 1$.

La différence entre l'entropie et la longueur moyenne des codes est donnée par

$$\begin{aligned}
\mathcal{H}(X) - \bar{L}_H(X) &= -\sum\nolimits_{a_i \in A} P(a_i) \lg P(a_i) - \sum\nolimits_{a_i \in A} P(a_i) L(a_i) \\
&= \sum\nolimits_{a_i \in A} P(a_i) \left(-\lg P(a_i) - L(a_i)\right) \\
&= \sum\nolimits_{a_i \in A} P(a_i) \left(-\lg P(a_i) + \lg 2^{-L(a_i)}\right) \\
&= \sum\nolimits_{a_i \in A} P(a_i) \lg \frac{2^{-L(a_i)}}{P(a_i)} \\
&\leqslant \lg \left(\sum\nolimits_{a_i \in A} 2^{-L(a_i)}\right) \leqslant 0
\end{aligned} \tag{6.7}$$

Cette dernière inégalité est une application de l'inégalité de Jensen, qui stipule que si une fonction f est convexe, alors $\mathrm{E}[f(X)] \leqslant f(\mathrm{E}[X])$, où $\mathrm{E}[X]$ est l'espérance de X [220]. La dernière partie de l'éq. (6.7), juste avant le $\leqslant$, est exactement cela : l'espérance du nombre de bits gaspillés par le code. Puisque $\mathcal{H}(X) - L(X) \leqslant 0$, nous avons $L(X) \geqslant \mathcal{H}(X)$. La longueur moyenne n'est donc jamais inférieure à l'entropie, ce qui raisonnable pour tout code respectant l'inégalité de Kraft-McMillan.

Pour montrer que la borne supérieure est respectée, nous montrerons qu'un code optimal a une longueur moyenne qui est au plus moins que un bit plus long que ce que l'entropie dicte. Idéalement, nous voudrions avoir des codes d'exactement $-\lg P(a_i)$ bits de long, mais cela demanderait des codes avec des longueurs réelles, et nous sommes contraints à utiliser des codes de longueur entière, puisque les codes produits par l'algorithme de Huffman sont de longueur entière. En posant, par hypothèse de l'optimalité du code

$$L(a_i) = \lceil -\lg P(a_i) \rceil$$

et tel que

$$-\lg P(a_i) \leqslant L(a_i) < 1 - \lg P(a_i) \tag{6.8}$$

L'inégalité stricte tient parce que, si $\lceil x \rceil = x + \varepsilon(x)$, alors $\lceil x \rceil - x = \varepsilon(x) < 1$. Cela signifie aussi que $2^{-L(a_i)} \leqslant P(a_i)$, ce qui nous donne

$$\sum\nolimits_{a_i \in A} 2^{-L(a_i)} \leqslant \sum\nolimits_{a_i \in A} P(a_i) = 1$$

Donc un ensemble de codes avec la fonction de longueur $L(a_i)$ satisfait l'inégalité de Kraft-McMillan, ce qui en retour signifie qu'il existe obligatoirement un ensemble de codes ayant ces longueurs ! En utilisant l'inégalité à droite de l'éq. (6.8), on obtient enfin

$$\bar{L}_H(X) = \sum\nolimits_{a_i \in A} P(a_i) L(a_i) < \sum\nolimits_{a_i \in A} P(a_i) \left(-\lg P(a_i) + 1\right) = \mathcal{H}(X) + 1$$

ce qui conclut la démonstration de $\mathcal{H}(X) \leqslant \bar{L}_H(X) < \mathcal{H}(X) + 1$. ■

Les codes générés par l'algorithme de Huffman sont donc au pire moins que un bit plus longs que strictement nécessaire. Des preuves plus avancées prouvent des bornes plus serrées. Par exemple, Gallager montre que si la distribution est dominée par une probabilité $p_{max} = \max_{a_i \in A} P(X = a_i)$, la longueur attendue sera bornée supérieurement par $\mathcal{H}(X) + p_{max} + \sigma$, où $\sigma = 1 - \lg e + \lg \lg e \approx 0.0860713320\ldots$ une constante que nous avons déjà vue à l'éq. (5.8) ! [76] Ce résultat est obtenu par l'analyse des fréquences des nœuds internes, définies comme les sommes des fréquences des symboles contenus dans les feuilles qui descendent de ces nœuds. R. M. Capocelli *et al.* donnent des bornes pour des cas spéciaux de p_{max}, en particulier $\frac{2}{9} \leqslant p_{max} \leqslant \frac{4}{10}$ et sans contraintes pour les autres p_i, ainsi que pour des distributions où $\frac{1}{3} \leqslant p_{max} \leqslant \frac{4}{10}$ et où seul p_{min} est connu, sans information sur les autres p_i [36]. Buro, pour toute fonction de distribution telle que $P(a_i) \neq 0$ pour tous les symboles et telle que l'éq. (6.1) tienne, caractérise la longueur maximale des codes générés en fonction de ϕ, le nombre d'or ! [31]

6.2.5.0.1 Disgression #1. Bien que nous ayons montré que la longueur moyenne des codes reste relativement près de l'entropie pour une source aléatoire X, nous n'avons pas montré quelle pouvait être la longueur maximale d'un code. Nous voyons qu'un alphabet avec une fonction de probabilité $P(a_i) = 2^{-\min(i, n-1)}$ générera un code avec une fonction de longueur $L(a_i) = \min(i, n-1)$, ce qui est le pire cas. Campos indique une autre façon de produire des codes maximalement longs [35]. Supposons que pour n symboles, nous ayons la fonction de probabilité suivante :

$$P(X = a_i) = \frac{F_i}{F_{n+2} - 1}$$

où F_i es le i^{e} nombre de Fibonacci. La somme des n premiers nombres de Fibonacci est donnée par

$$\sum\nolimits_{i=1}^{n} F_i = F_{n+2} - 1$$

ce qui donne aussi la somme des fréquences, faisant en sorte que la fonction de probabilité somme bel et bien à 1. Cet ensemble de probabilité générera aussi des codes de longueur maximale $n-1$. Pour éviter de construire des codes ridiculement longs — n est souvent 256 — on utilisera un des algorithmes qui produisent des codes de Huffman avec des longueures maximales contraintes. Nous en discourrons à la section 6.3.4.

6.2.5.0.2 Disgression #2. Nous avons noté que l'assignation de 0 au fils de gauche et de 1 au fils de droit avait quelque chose d'arbitraire, bien que menant à des codes de la forme canonique. Si on ne tient pas à la canonicité du code, on peut bien renverser l'utilisation de 0 et de 1. Cela nous mène à l'observation que pour un seul arbre, une multitude de codes sont possibles. Supposons, par exemple, qu'à chaque embranchement de l'arbre nous décidions aléatoirement de l'assignation sur les fils. Cela demeurerait un arbre de codes valide. Rappelons nous qu'un arbre de code produit par l'algorithme de Huffman est complet (*full*), et que pour n feuilles, nous avons $n-1$ nœuds internes. Puisque nous pouvons choisir l'assignation de chacun de ces $n-1$ nœuds internes, cela revient à assigner un état binaire à chaque nœud (un état ($0 \rightarrow$ gauche, $1 \rightarrow$ droite) et un état ($0 \rightarrow$ droite, $1 \rightarrow$ gauche), donc deux valeurs distinctes), donnant 2^{n-1} assignations possibles sur tout l'arbre ; donc nous avons ainsi 2^{n-1} arbres de codes équivalents, tous optimaux.

6.3 Variations sur un thème

Comme les codes de Huffman sont efficaces, il n'est pas surprenant de les retrouver çà et là et sous diverses formes. Bien qu'ils soient effiaces, les codes dans leur forme originale ne satisfont pas nécessairement tout le monde. D'aucuns auront besoin d'encoder des symboles tirés de très grands alphabets, d'autres les voudront contraints en longueur. Il y a aussi des cas où les codes de Huffman ne sont pas aussi efficaces que l'on pourrait le souhaiter. Dans cette section, nous présentons des algorithmes modifiés qui produisent des codes légèrement différents. Nous discourerons des codes de Huffman modifiés, des codes à préfixe Huffman et des codes de Huffman étendus.

6.3.1 Codes de Huffman modifiés

Dans la compression fac-similé (aussi connue sous le nom de Fax), le document à transmettre est numérisé en lignes de 1728 pixels, à concurence de 3912 lignes par page. Chaque ligne du document est transformée en répétitions de pixels blancs et de pixels noirs et ces répétitions sont codées, selon les recommandations du Groupe 3 du CCITT, grâce à un code de Huffman modifié (*modified Huffman codes*). Cette modification à l'algorithme de Huffman alloue deux ensembles de codes, l'un pour les codes de répétitions des pixels blancs, l'autre pour les répétitions de pixels noirs. La raison en est que les répétitions de pixels blancs ont une distribution nettement différente de la distribution des répétitions des pixels noirs, comme on peut facilement s'en convaincre en regardant la page que vous lisez, où à peu près 10% de la surface est recouverte par de l'encre ; 90% de la page restant blanc. Les répétitions de pixels noirs correspondent à des fragments de lettres, lesquelles sont habituellement petites, menant donc à de petites répétitions. Utiliser un code unique pour les deux classes de répétitions serait évidemment très inefficace.

Ces codes sont « modifiés », non pas à cause de la séparation entre les codes pour les répétitions de pixels blancs et les répétitions de pixels noirs, mais parce qu'au lieu d'allouer des codes pour toutes les longueurs l possibles, $0 \leqslant l < 1728$, seul un nombre restreint de longueurs distinctes sera permi. Des codes sont alloués pour les longueurs $0 \leqslant l < 63$, puis pour tout les longueurs multiples de 64, jusqu'à 2560. Les codes pour les longueurs $0 \leqslant l < 63$ sont appelés les codes terminaux (*termination codes*) car ils servent à encoder la fin d'une répétition, alors que les autres codes sont appelés codes de remplissage (*make-up codes*) car ils composent le corps principal des répétitions [184].

6.3.2 Codes à préfixes Huffman

Le problème avec un alphabet ayant un très grand nombre de symboles, c'est qu'il est probablement impossible de faire tenir l'arbre de code correspondant en mémoire, sans compter que la description de l'arbre qui doit être transmise au décompresseur serait tellement coûteuse à coder que la compression du reste de la séquence deviendrait tout à fait futile.

Une façon de concevoir un ensemble de code pour un très grand nombre de symboles pourrait être de diviser les symboles en classe d'équivalence et de calculer un code de Huffman pour les codes de classes seulement. Ainsi, chaque symbole dans une classe obtiendrait un préfixe commun avec les autres symboles dans la classe, et ce préfixe serait optimisé grâce à la procédure de Huffman. Supposons que nous ayons un alphabet avec un très grand nombre de symboles, quelque chose dans l'ordre de 2^{32}. De toute évidence, il est impossible de conserver en mémoire l'arbre de codes. Nous découperons l'alphabet en partitions de façon à ce que tous les symboles qui se retrouvent dans une même partition aient à peu près la même probabilité. Chaque partition reçoit une probabilité égale à la somme des probabilités des symboles qui s'y trouvent et on lance l'algorithme de Huffman sur les probabilités des partitions.

Définissons les classes d'équivalence et la notion de « probabilités à peu près égales ». Deux symboles a et b, tirés de l'alphabet A, seront équivalents, notés $a \equiv_\alpha b$ si, et seulement, $\lceil -\alpha \lg P(a) \rceil = \lceil -\alpha \lg P(b) \rceil$. Soit donc C_{p_i}, la classe d'équivalence de la probabilité p_i. Le paramètre α contrôle la finesse des classes d'équivalence. La probabilité associée à une classe d'équivalence est donnée par $P(C_{p_i}) = \sum_{a \in C_{p_i}} P(a)$.

L'ensemble de codes pour les préfixes satisfera l'éq. (6.6), ce qui veut dire que les codes pour les préfixes seront au plus moins de un bit plus long que l'entropie. La partie suffixe, qui encode simplement l'index d'un symbole à l'intérieur d'une classe d'équivalence, est un entier de longueur fixe et pourrait être codé par un code binaire naturel, donné par $C_\beta(\cdot)$, soit par un code plus asticieux comme un code *phase-in*. Si le code utilisé pour le suffixe est un code binaire naturel, le code complet qui en résulte est alors au plus moins de deux bits plus long que l'entropie. Si le code utilisé pour le suffixe est un code *phase-in*, le code complet sera au plus 1.086071... bits trop longs (voir la section 5.3.4.2). Le tableau 6.2 montre un code obtenu avec une telle stratégie.

6.3.3 Codes de Huffman étendus

Si l'alphabet source est plutôt grand, p_{max} sera probablement plutôt petit. Toutefois, si l'alphabet contient très peu de symboles, les chances sont que p_{max} sera plutôt grand comparativement aux autres probabilités. À la section 6.2.5, nous avons vu que la longueur moyenne des codes générés sera plus petite ou égale à $\mathcal{H}(X) + p_{max} + 0.086...$ Cela ressemble à une garantie que les codes ne pourront jamais être très mauvais.

Ce n'est malheureusement pas le cas. Considérez le cas suivant. Supposons que notre alphabet source soit $A = \{0, 1\}$ et que l'alphabet de sortie soit aussi $B = \{0, 1\}$. Indépendamment des probabilitées des symboles de l'alphabet original, la longueur moyenne des codes résultant est toujours $\mathcal{H}(X) \leqslant L(X) < \mathcal{H}(X) + p_{max} + 0.086\ldots$, mais *absolument aucune compression n'est obtenue*, parce que le code de Huffman calculé sera $\{0 \to 0, 1 \to 1\}$.

Probabilités	Codes	Intervalle représenté
$P(0 \leqslant X < 4) = \frac{1}{2}$	0 | $x_1 x_0$	0 – 3
$P(4 \leqslant X < 11) = \frac{1}{4}$	10 | $x_2 x_1 x_0$	4 – 11
$P(12 \leqslant X < 27) = \frac{1}{16}$	1100 | $x_3 x_2 x_1 x_0$	12 – 27
$P(28 \leqslant X < 59) = \frac{1}{16}$	1101 | $x_4 x_3 x_2 x_1 x_0$	28 – 59
$P(60 \leqslant X < 128) = \frac{1}{16}$	1110 | $x_5 x_4 x_3 x_2 x_1 x_0$	60 – 123
$P(124 \leqslant X < 251) = \frac{1}{16}$	1111 | $x_6 x_5 x_4 x_3 x_2 x_1 x_0$	124 – 251

TAB. 6.2 – Un exemple d'un « très grand » alphabet réduit à un petit nombre de classes d'équivalence, et le code à préfixe de Huffman qui en résulte.

Comment nous tirer de ce bourbier ? Une solution, c'est d'utiliser les alphabets étendus. L'alphabet étendu d'ordre m généré à partir de A est donné par

$$A^m = \underbrace{A \times A \times \cdots \times A}_{m \text{ fois}} = \{a_1 a_1 \dots a_1 a_1, \underbrace{a_1 a_1 \dots a_1 a_2}_{m \text{ symboles}}, \dots, a_n a_n \dots a_n\}$$

où $A \times A$ est le produit cartésient de A avec lui-même, et la probabilité d'un symbole a_i^m de A^m est simplement (si on assume que les symboles de A sont i.i.d) :

$$P(a_i^m) = \prod_{j=1}^{m} P(a_{ij}^m)$$

On obtient alors n^m symboles et le même nombre de probabilités. Le code de Huffman calculé pour le nouvel alphabet, sous hypothèse que les symboles originaux sont i.i.d, satisfera

$$\mathcal{H}(X^m) \leqslant L(X^m) < \mathcal{H}(X^m) + 1$$

ce qui donne, pour les symboles de A, une longueur de code moyenne de

$$\frac{1}{m}\mathcal{H}(X^m) \leqslant \frac{1}{m}L(X^m) < \frac{1}{m}\left(\mathcal{H}(X^m) + 1\right)$$

ou, après simplification

$$\mathcal{H}(X) \leqslant L(X) < \mathcal{H}(X) + \frac{1}{m}$$

On peut donc correctement conclure que de compacter les symboles nous permet de nous approcher arbitrairement près de l'entropie, pour autant que nous soyons prêts à accepter un grand m. Le problème avec un grand m, c'est que le nombre de nouveaux symboles (et leur probabilitées) pour un alphabet original de taille n est de n^m, ce qui vient rapidement incontrôlable sauf pour des valeurs de n et m trivialement petites. Remarquons que $n = 2$ et $m = 8$ revient

à calculer un code de Huffman sur des octets, ce qui est très raisonnable quant à la taille de l'arbre de code.

Dans cette situation, d'autres techniques de compression peuvent être préférable à la méthode de Huffman. Par exemple, on pourrait considérer bâtir un code de Tunstall de k bits [209]. Un code de Tunstall associe une séquence de longueur variable de symboles d'entrée à un code de longueur fixe. On peut bâtir un tel code avec les $2^k - n$ premiers codes associés aux $2^k - n$ séquences les plus probables *a priori* sous l'hypothèse i.i.d, et prendre n pour représenter les symboles seuls. On pourrait aussi utiliser un algorithme complètement différent, comme une variante de LZ77 ou de LZ78, voir même l'utilisation d'un codeur arithmétique. L'utilisation d'un codeur arithmétique nécessiterait une estimation des probabilités avec un modèle statistique de faible ordre, comme une chaîne de Markov d'ordre c ou un mécanisme équivalent.

6.3.4 Codes de Huffman à longueurs restreintes

On peut vouloir limiter la longueur des codes produits par l'algorithme de Huffman. Il y a plusieurs raisons de vouloir le faire. Une raison possible de limiter la longueur des codes, c'est d'éviter d'obtenir des codes trops longs pour les symboles dont la probabilité risque d'être sous-estimée. Souvent, on obtient une estimation des probabilité en n'échantillonnant qu'une partie restreinte des données. Pour les symboles plus rares, il est possible que l'on ne les observe pas du tout, ce qui résulte en une fréquence estimée de zéro, bien qu'en fait la probabilité soit plus grande. Une autre raison pourrait être de limiter le nombre de bits à tenir en mémoire (ou dans un registre du processeur) nécessaires pour décoder un symbole. On pourrait aussi vouloir limiter le nombre d'étapes nécessaires au décodage d'un symbole. Contraindre ainsi le temps de décodage est d'une grande importance dans les applications multimédia où la synchronisation est cruciale.

Il existe plusieurs algorithmes pour calculer des codes contraints en longueur. Fraenkel et Kline présentent un algorithme exact de complexité de calcul $O(n \lg n)$ et une utilisation de mémoire $O(n)$ [73]. Subséquemment, Milidiú *et al* présentèrent une version rapide mais approximative de l'algorithme, demandant $O(n)$ étapes de calcul et nécessitant $O(n)$ espace mémoire [139]. L'algorithme de Moffat et Katajainen calcule la longueur des codes *in situ* grâce à une méthode de type programmation dynamique avant de générer les codes [146]. Enfin, Larmore et Hirschberg présentent aussi un algorithme pour calculer des codes de Huffman de longueur restreinte [124]. Ce dernier article est complet dans la mesure où il décrit en détail les fondements mathématiques comme les détails d'implémentation.

6.4 Codes de Huffman adaptatif

Les codes produits par l'algorithme de Huffman sont dits *statiques*. Les codes sont générés en deux étapes. La première étape demande d'amasser des statistiques sur la distribution des symboles selon une source aléatoire. Cela peut se faire de deux façons. La première façon est de parcourrir complètement les données afin d'obtenir les fréquences des symboles, mais on peut avoir à traiter une grande quantité de données, et nous n'avons pas nécessairement le loisir de passer deux fois sur les données. De plus « toutes les données » peut être un concept illusoire. En effet, en télécommunication, nous n'avons pas toutes les données puisque celles-ci nous arrivent en petits paquets par le canal de communication. Nous devons alors échantillonner les données,

en nous exposant au risque de faire de mauvaises estimations si l'échantillon n'est pas suffisamment grand ou pas assez représentatif. L'un ou l'autre résulterait en un code plutôt mauvais.

Heureusement, il y a un bon nombre de façons de faire face à ce problème. Les solutions portent le nom commun de codage de Huffman adaptatif ou dynamique. Ces algorithmes changent les estimations des fréquences au fur et à mesure qu'ils traitent la séquence à compresser, ce qui permet de produire des codes qui s'adaptent aux caractéristiques locales de la séquence, générant ainsi un code plus efficace. Dans cette section, nous passons en revue les méthodes qui ont été proposées pour calculer les codes de Huffman adaptatifs.

6.4.1 Algorithmes de force brute

Une façon naïve de concevoir des codes de Huffman adaptatifs, c'est la méthode de force brute. On pourrait recalculer le code au complet après chaque nouveau symbole observé grâce à une implémentation efficace de l'algorithme de Huffman — comme la méthode de Moffat et Katajainen, section 6.5, qui demande $O(n)$ étapes pour créer le code. Même à $O(n)$, il semble peu pratique de recalculer le code au complet après chaque observation.

Et si nous ne recalculions les codes que de temps en temps ? On pourrait facilement imaginer qu'on ne recalcule les codes qu'à tous les k symboles, divisant ainsi le temps nécessaire par k. Si nous mettons à jour les codes à tous les k symboles, nous réduisons le temps de calcul par un facteur k, mais au coût d'une perte en compression. Une meilleure solution serait de recalculer les codes quand nous déterminons que les statistiques ont significativement changé.

Cela est facile à faire avec une structure de données qui maintient les rangs. Une telle structure est montrée à la fig, 6.3. Un index qui utilise le symbole comme clef pointe dans la table de rangs, où se trouve l'information de fréquence pour ce symbole. Au fur et à mesure que les symboles sont observés, on incrémente les fréquences associées aux symboles observés. Le symbole est alors percolé jusqu'à ce que les rangs soient revenus en ordre décroissant. Comme pour l'infâme tri à bulles, on échange l'entrée courante de la table avec celle qui la précède tant que la fréquence courante est plus élevée. Pendant la percolation, nous ajustons aussi l'index principal pour que chaque entrée dans l'index continue de pointer au bon symbole dans la table des fréquences. Cette procédure, bien que fort coûteuse pour un tri, est quand même essentiellement $O(1)$ pour maintenir les rangs car après un bon nombre d'observations, l'information de rang est bonne et les changements peu fréquents. Toutefois, au début, cette procédure est $O(n)$ parce que les symboles observés la première fois vont percoler jusqu'au début de la table de rangs. Après, et seulement si, un échange a eu lieu, on recalcule le code de Huffman grâce à une implémentation efficace.

Des expériences que nous avons menés nous-même montrent que cet algorithme adaptatif brutal performe légèrement mieux que l'algorithme de Vitter (que nous présentons à la section 6.4.3) et que, pour des fichiers textes ou d'image (en particulier sur le corpus de Calgary, lequel est décrit à l'appendice D) les recalculs des codes se font relativement rares. En effet, l'algorithme va recalculer le code pour environ de 2 à 5% des symboles, donnant ainsi un temps d'exécution tout à fait acceptable.

Cet algorithme n'est pas aussi sophistiqué que les algorithmes que nous allons présenter dans les quelques prochaines sous-sections. En effet, on voit qu'il fait beaucoup de travail inutile. Avec

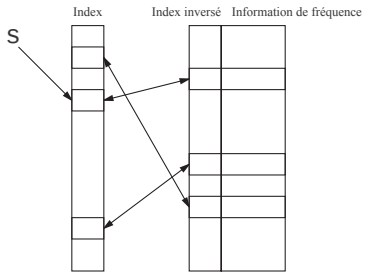


FIG. 6.3 – Une structure qui conserve l'information de rang des symboles. L'index permet d'utiliser le symbole pour trouver son emplacement dans la table secondaire. L'index inversé (qui est en fait seulement le symbole qui occupe ce rang) permet de modifier l'index principal lorsque le symbole change de rang.

cet algorithme simple, on recalcule le code à chaque fois qu'un échange de rang a lieu, mais un échange ne signifie pas que les codes changent ! En effet, deux codes qui échangent leurs positions peuvent conserver des codes de même longueur ; à la limite on échange leur codes.

6.4.2 L'algorithme FGK : Faller, Gallager, et Knuth

Faller et Gallager ont introduit essentiellement le même algorithme, à quelques années d'intervalle [68, 76]. Ce même algorithme sera plus tard amélioré par Cormack et Horspool [57]. Les mêmes améliorations reviennent un an plus tard, cette fois proprosées par Knuth [114]. La méthode proposée par Gallager demande l'introduction de matériel préparatoire, qui fera l'objet des quelques prochains paragraphes.

Un arbre a la *sibling property* si chaque nœud interne autre que la racine a un frère, et si tous les nœuds peuvent être énumérés en ordre croissant de probabilités. Dans un tel arbre, il y a $2n-2$ nœuds satisfaisant cette propriété (celui qui manque pour faire $2n-1$ nœuds, c'est la racine). Un théorème introduit par Gallager stipule qu'un arbre complet (*full*) est un arbre de Huffman si, et seulement si, il a la *sibling property*. La profondeur d'un nœud est definie comme le nombre d'arcs qui le sépare de la racine. La racine a une profondeur de 0. Les probabilités des noeuds internes sont définies comme étant la somme des probabilités des symboles contenus dans les feuilles qui descendent de ce nœud.

Nous maintiendrons un arbre tel que chaque nœud, à une profondeur donnée d, a une probabilité qui est inférieure ou égale à toutes les probabilités de tous les nœuds à la profondeur $d-1$ et moins. Cet arbre est dit ordonné si, pour tous les nœuds, le fils ayant la plus grande probabilité reçoit le 0. Un arbre sera dit lexicographiquement ordonné si, pour toute profondeur $d \geqslant 1$, tous les nœuds à la profondeur d ont des probabilités plus petites que les nœuds aux profondeurs $d-1$ et moins, et que tous les nœuds à la profondeur d peuvent être énumérés en ordre croissant de probabilités, de gauche à droite dans l'arbre. Enfin, un arbre correspond à un code de Huffman canonique si, et seulement si, il est lexicographiquement ordonné.

La procédure de modification de l'arbre proprosée par Gallager nécessite que la propriété d'ordre lexicographique soit maintenue en tout temps. Nous ne discuterons pas des structures de données nécessaires à la maintenance efficace (en temps comme en mémoire) d'un tel arbre, puisque l'article de Knuth discute ces aspects en grand détail [114].

Quand un nouveau symbole est observé, son ancien code (qui est obtenu de la même fa-

çon qu'avec l'algorithme de Huffman, c'est-à-dire en descendant dans l'arbre jusqu'à la feuille qui contient ce symbole) est transmit, de façon à ce que le décodeur demeure synchronisé avec l'encodeur. Après l'émission du code, on commence la procédure de mise à jour de l'arbre. La fréquence associée au symbole (qui est dans une feuille) est incrémenté. Soit a, cette feuille, à la profondeur d. Après l'incrément, nous vérifions si tous les nœuds à la profondeur d sont encore en ordre lexicographique, c'est-à-dire en ordre croissant de probabilité, de gauche à droite. Si ce n'est pas le cas, nous échangeons la feuille a avec la feuille la plus à droite qui a maintenant une fréquence inférieure à a, disons la feuille b. Comme la fréquence de b correspond à l'ancienne fréquence de a, aucune mise à jour n'est nécessaire pour son parent. Par contre, le nouveau parent de a sera récursivement mis à jour jusqu'à la racine, en faisant des échanges au besoin (car il faut respecter la propriété lexicographique à toutes les profondeurs). Les figs. 6.4 et 6.5 illustrent le processus. L'état initial de l'arbre est adapté de l'exemple donné par Gallager [76].

Cet algorithme ne permet que des incréments unitaires et positifs et il ne traite pas explicitement les symboles qui ont une fréquence de zéro. Le problème avec l'utilisation des incréments positifs des fréquences, c'est que si l'algorithme converge éventuellement vers de bonnes estimations des fréquences, il ne capture aucune fluctuation locale. Gallager proprose une variation où les fréquences sont multipliées par une constante quelconque $0 < \alpha \leqslant 1$ de temps en temps pour permettre aux incréments de provoquer de bons changements dans l'arbre. Un problème avec cette méthode c'est déterminer le « temps en temps », c'est-à-dire la fréquence à laquelle on applique la constante α. Un autre, c'est qu'il faut maintenir la condition d'ordre lexicographique et il faut faire attention à la façon dont les fréquences sont tronquées ou arrondies.

Cormack et Horspool proposent un algorithme qui permet d'augmenter les fréquences par n'importe quelle quantité entière et positive. Cette procédure a été par la suite généralisée par Knuth pour permettre des « incréments » négatifs. Cela permet de ne maintenir un code qu'à partir des fréquences amassées sur une fenêtre de longueur finie plutôt que lentement converger sur les fréquences pour toute la séquence. Si l'hypothèse que la source aléatoire est stationnaire paraît raisonnable pour une classe de séquence, nous n'avons pas besoin réellement d'avoir une mécanisme d'adaptativité qui capture les fluctuations des probabilités en ne maintenant des statistiques que sur une fenêtre finie. Il n'est toutefois pas toujours pertinent de supposer que la source est stationnaire. Même capturer de petites fluctuations peut mener à un code significativement plus performant.

L'algorithme de Gallager s'initialise avec un arbre complet, où chaque feuille représente chaque symbole. Cet arbre est complet (*complete*) car pour un alphabet de taille $n = 2^k$, tous les symboles sont sur des feuilles de profondeur k, donnant ainsi, rien de surprenant, des codes de longueur k à chaque symbole. Si $n = 2^k + b$, l'arbre aura ses feuilles sur exactement deux profondeurs distinctes... comme pour les codes *phase-in* (voir section 5.3.4.2 pour une dérivation de combien de feuilles se trouvent sur chaque niveau). Knuth a modifié l'algorithme de Gallager pour que l'algorithme s'initialise avec un arbre vide, sauf pour une racine-feuille qui contient un symbole spécial d'échappement. Le symbole d'échappement sert à introduire un nouveau symbole dans l'arbre. C'est en effet avantageux de ne pas créer l'arbre avec tous les symboles, puisqu'il est probable que la séquence à compresser ne contienne qu'un sous-ensemble des symboles possibles.

L'algorithme de Knuth procède ainsi : au moment de l'initialisation, il n'y a qu'une racine-feuille qui contient le symbole d'échappement. Quand on veut ajouter un nouveau symbole, on émet le code spécial d'échappement suivi du symbole à introduire, soit un nombre entre 0 et

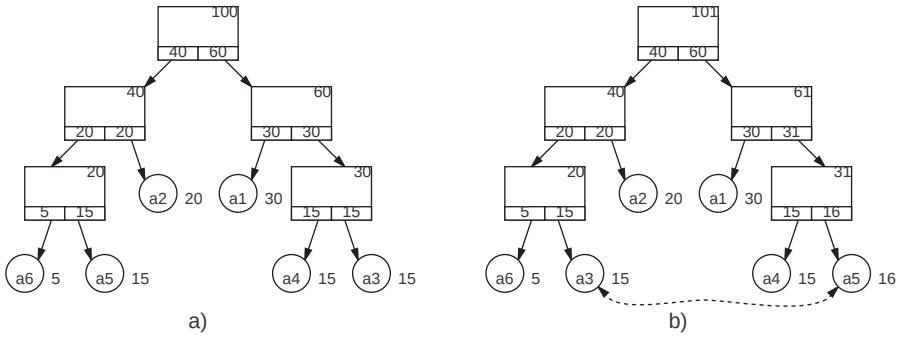


FIG. 6.4 – Une mise à jour simple. La partie a) montre l'arbre avant la mise à jour et la partie b), l'arbre après la mise à jour. Le symbole qui a été observé est a_5. Les flèches pointillées montrent où ont eu lieu les échanges.

$n - r - 1$, où r est le nombre de symboles déjà ajoutés à l'arbre. Le symbole est ajouté dans l'arbre comme le frère du symbole d'échappement ; un nouveau nœud interne est créé pour servir de parent au symbole d'échappement et au nouveau symbole. Cela maintient exactement un seul symbole d'échappement. Quand le dernier symbole possible est introduit, il remplace simplement le symbole d'échappement qui n'aura plus sa raison d'être. Après qu'un symbole soit observé ou ajouté, on incrémente sa fréquence de un et l'algorithme procède comme pour l'algorithme de Gallager.

Il est tout à fait raisonnable d'utiliser un code d'échappement. D'abord, cela permet d'avoir des codes très courts dès le départ de la compression, quitte à les voir allonger au fur et à mesure que nous avançons dans la séquence. Ensuite, si la séquence ne contient que très peu de symboles distincts, nous ne construirons pas l'arbre au complet pour n'en utiliser qu'une petite partie.

Knuth analyse la complexité computationnelle des algorithmes de mise à jour positive et négative et trouve que l'un et l'autre est $O(-\lg P(X = a_i))$ pour l'observation d'un symbole a_i. Sur une séquence de longueur m, la complexité est $O(m\mathcal{H}(X))$, puisque la valeur moyenne de $-\lg P(X = a_i)$ est $\mathcal{H}(X)$. La demande en mémoire de l'algorithme est $O(c(2n-1))$, où c est le coût (en bits ou en octets) pour un nœud dans l'arbre. Cet algorithme est donc tout à fait adéquat pour le calcul adaptatif des codes de Huffman.

6.4.3 L'algorithme de Vitter : algorithme Λ

L'algorithme Λ de Vitter est une variation de l'algorithme FGK [216, 217]. La différence principale entre cet algorithme et l'algorithme FGK est la façon dont l'algorithme Λ numérote les nœuds de l'arbre. L'algorithme Λ trie les nœuds de la même façon que l'algorithme FGK, à la différence que les feuilles à une profondeur donnée précèdent les nœuds internes de même profondeur et de même fréquence. Cette modification, il est montrée par Vitter, est suffisante pour garantir qu'une séquence de longueur s n'est jamais codée avec plus de s bits de plus que ce que l'algorithme statique de Huffman nous aurait donné, en ne comptant pas ce qu'aurait nécessité le transfert de l'information nécessaire à la construction du code statique [216]. Une implémentation complète en pseudo-pascal est donné par Vitter, utilisant la plupart des struc-

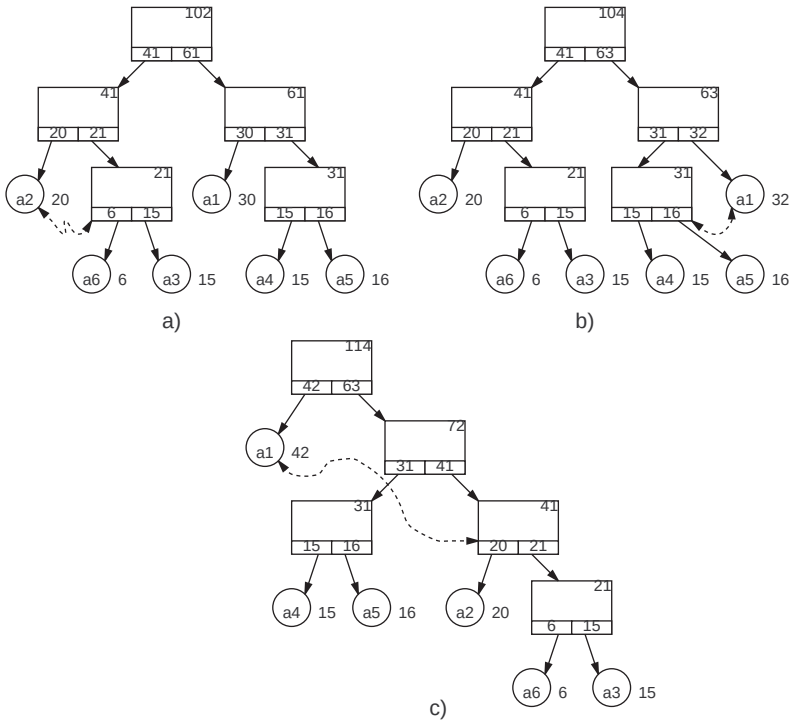


FIG. 6.5 – Après de multiples mises à jour. La fig. a) montre l'arbre après l'observation du symbole a_6. La fig, b) montre l'arbre après deux observations du symbole a_1. À ce moment, le symbole a_1 est aussi loin à droite qu'il peut l'être à ce niveau, la profondeur 2. S'il est mis à jour encore, il faudra le faire monter d'un niveau, au niveau 1. La partie c) montre l'abre mis à jour après dix observations de a_1. Notez qu'un sous-arbre complet a été échangé, comme le montre la flèche pointillée.

tures présentées par Knuth [217, 114]. Le code est suffisamment clair pour qu'il soit facile d'en faire une implémentation dans son langage préféré.

6.4.4 Autres algorithmes adaptatifs

Jones présenta un algorithme qui utilise les *splay trees* pour gérer l'adaptativité [107]. Un *splay tree* est un arbre de recherche où un élément fraîchement accédé est promu à un nœud près de la racine, sans toutefois changer l'ordre des clefs et sans altérer la possibilité de faire des recherches dans l'arbre [203]. Dans le cas d'un arbre de Huffman, les nœuds internes ne contiennent pas de clefs, seules les feuilles portent les symboles. Cependant, un *splay tree* requiert que tous les nœuds portent une clef. Jones propose une légère modification qui fait que seuls les nœuds internes sont percolés vers la racine en prenant soin de ne pas changer les feuilles. La percolation, telle que spécifiée par Tarjan, ramène une clef à la racine ; celle préconisée par Jones contraint la percolation de façon à ne réduire la profondeur que par un facteur deux. Nous reviendrons sur le détail de cet algorithme à la section 8.2, où nous verrons aussi comment le splaying de base mène à des conditions pathologiques pour l'arbre de code. Cette modification aux *splay trees* pondère l'adaptativité et empêche ainsi que le dernier symbole observé soit catapulté à la racine au détriment des autres symboles pourtant beaucoup plus fréquents. Cet algorithme performe cependant moins bien que les autres algorithmes adaptatifs, sauf en de rares occasions. Si la séquence comporte localement de longues séries de répétitions, les autres algorithmes adaptatifs vont compenser plutôt lentement pour ce changement soudain dans les statistiques mais l'algorithme de Jones donnera très rapidement à ce symbole répété le code le plus court. Il suffit de penser qu'après quelques répétition, même le symbole avec le code le plus long, en montant vers la racine deux étages à la fois, se retrouvera rapidement à la racine. Cette volatilité contrôlée donne à l'algorithme de Jones une mémoire à court terme des statistiques de la séquence.

Pigeon et Bengio ont aussi proprosé un algorithme utilisant le principe des *splay tree* pour offrir une certaine adaptativité, mais la percolation est mieux contrôlée [162, 163, 165, 164]. Cette variante, l'algorithme M, sera décrite en grand détail au chapitre 8, où nous présenterons aussi l'algorithme W, intermédiaire entre l'algorithme de Jones et l'algorithme M. Un nœud provoque une percolation seulement s'il a une fréquence plus grande que son oncle. Bien que l'algorithme Λ batte systématiquement l'algorithme M, ce n'est que d'environ 5% sur la longueur moyenne des codes, et ce, pour de petits alphabets, la situation est renversée lorsque l'on considère de très grands alphabets, disons quelques milliers de symboles. Cela laisse entendre une utilité possible pour la compression de séquences codés en Unicode (un code qui permet de représenter 65536 symboles typographiques, de l'alphabet romain jusqu'au sanskrit devanagari). La performance de l'algorithme M est inférieure à la performance d'un code statique de Huffman, et bien qu'en principe nous puissions avoir un cas où l'algorithme M produit des codes jusqu'à deux bits de plus que l'entropie, en général, la longueur moyenne est très près de ce que donnerait un codeur de Huffman statique. Une autre particularité de l'algorithme M, c'est que les symboles ne reçoivent pas une feuille individuelle. Ils sont regroupés, par classe d'équivalence de fréquences dans un même ensemble. Ce sont les ensembles qui reçoivent les feuilles, ce qui mène potentiellement à une économie de mémoire lorsque le nombre de symboles qui partagent une même feuille est grand, bien que cela pose d'autres problèmes, comme retrouver efficacement où est rendu quel symbole, et comment représenter les ensembles ténus.

6.4.5 Une observation sur les codes de Huffman

McIntyre et Pechura préconisent l'utilisation des codes de Huffman statiques pour de petits fichiers ou de courtes séquences, puisque le processus d'adaptation est relativement lent et demande quand même un grand nombre de symboles observés pour produire une approximation satisfaisante de la distribution des symboles [135]. Cette situation survient naturellement lorsque le nombre de symboles observé est petit et comparable au nombre de symboles distincts qui composent l'alphabet. Par exemple, si on a un alphabet de 256 symboles, et que l'on observe une séquence de 20 symboles, c'est bien évident que la compression obtenue grâce à un algorithme adaptatif à partir d'une mauvaise estimation initiale de la fonction de probabilité sera médiocre. Une solution possible, c'est d'utiliser un code de Huffman précalculé pour une classe de séquence, connu du compresseur comme du décompresseur, ou encore un code très simple du genre codes à préfixes Huffman ; nous en avons discuté à la section 6.3.2.

6.5 Implémentations efficaces

Les travaux sur les implémentations efficaces des **co**deurs/**dec**odeurs, ou codecs, pour les codes de Huffman prennent essentiellement deux directions quasiment opposées : les codecs efficaces en terme de la mémoire utilisée et les codecs qui sont efficaces en terme du temps d'encodage et décodage. Les algorithmes efficaces en terme de la mémoire utilisée vont chercher à utiliser aussi peu de mémoire que possible. Ce but est généralement atteint grâce à l'utilisation de structures de données alternatives qui nous dispensent de l'utilisation de l'arbre de code. Les algorithmes efficaces en terme de vitesse ne cherchent qu'à optimiser le temps d'encodage et de décodage, quitte à utiliser une quantité beaucoup plus grande de mémoire.

6.5.1 Algorithmes économes en mémoire

Si on opte pour une représentation explicite de l'arbre de code en mémoire, on a peu de choix. Soit on utilise une structure classique d'arbre avec des nœuds qui contiennent de l'information intermédiaire, plus les pointeurs nécessaires à la connectivité de l'arbre, etc., soit on utilise une structure avec une structure d'adressage en monceau. Un monceau (*heap*) peut être représenté dans un tableau linéaire, où un nœud à la case $k \geqslant 0$ voit son fils de gauche alloué à la case $2k+1$ et son fils de droite à la case $2k+2$. Ce type de stockage peut-être éparse et ne convient qu'aux arbres complets dont la profondeur maximale reste faible. Nous laissons le lecteur considérer la quantité de mémoire qui peut être gaspillée par un tel schème.

Moffat et Katajainen proposent un algorithme de type programmation dynamique pour calculer la longueur des codes en $O(n)$ étapes et en utilisant $O(1)$ mémoire de travail, pour peu que les fréquences soient déjà triées en ordre décroissant [146]. L'idée est que cet algorithme réécrit la longueur des codes par dessus l'information de fréquence. Puisque la liste de fréquence ne requiert aucune autre information supplémentaire, l'algorithme élimine le besoin d'utiliser d'autres données pour maintenir un arbre, etc. Puisque la mémoire nécessaire pour stocker un nœud est plusieurs fois celle nécessaire pour emmagasiner une fréquence (qui n'est qu'un entier), on obtient une économie de mémoire substantielle. Dans des articles satellites, Moffat et Turpin décrivent un algorithme qui permet d'utiliser seulement $O(L_{max})$ cases mémoire ! [147, 148]. Cet algorithme utilise quelques tables qui décrivent, non l'arbre au complet, mais seulement les différentes longueurs et le nombre de codes de cette longueur. Puisque les symboles seront placés en ordre décroissant de fréquence les codes sont nécessairement triés en ordre croissant

de longueur. Sachant les longueurs et les multiplicités, on peut encoder et décoder des codes de Huffman canoniques. Cet algorithme n'est cependant pas dû à Moffat et Katajainen car on en trouve description faite par Hirschberg et Lelewer sept ans auparavant, et si on se fie aux insinuations dans l'article, il semblerait que cet algorithme était déjà connu bien avant [97]. S'il suffit de $O(L_{max})$ bits pour décrire l'arbre, il reste à résoudre le problème de la description de la permutation appliquée sur les symboles, un problème qui est souvent glissé sous le tapis.

6.5.2 Algorithmes rapides

Les algorithmes que nous avons présenté jusqu'à maintenant encodent ou décodent un bit à la fois. Pour accéler les choses, Choueka *et al*, Siemiński et Tanaka présentent divers algorithmes et structures de données, toutes reposant sur des automates finis déterministes [48, 190, 202]. Bien que l'utilisation d'un automate ne soit pas explicite dans tous les articles, l'idée de base demeure la même : plutôt que de décoder bit par bit, le décodage procède par blocks de b bits, et l'état interne du décodeur est représenté par un automate augmenté.

Un automate, au sens de l'informatique théorique, est un quintuplet (Σ, S, F, i, T). Σ est l'alphabet d'entrée, S l'ensemble des états de l'automate, $i \in S$ l'état initial, $F \subseteq S$ est l'ensemble des états finaux, états dans lesquels il est légal de rencontrer une fin de séquence, et T est la fonction de transition $T : S \times \Sigma \to S$. La variété d'automates que nous allons considérer ici possèdent aussi un alphabet de sortie, Ω, de façon à ce que la fonction de transition devienne $T : S \times \Sigma^* \to S \times \Omega^*$. La fonction de transition prend pour entrée l'état courant, un certain nombre de symboles d'entrées, produit un certain nombre de symboles de sortie et change l'état courant pour un nouvel état. L'automate démarre à l'état i, et il ne peut arrêter que s'il rejoint une fin de séquence quand il est dans un état final.

Comment on peut utiliser un automate au décodage rapide n'est pas évident. Dans le cas qui nous intéresse, $\Sigma = \mathbb{B} = \{0, 1\}$ et Ω est l'alphabet original de la séquence. Puisque nous somme intéressés à lire b bits à la fois, nous aurons à tenir compte de toutes les séquences de b bits ainsi que leurs préfixes et suffixes. Supposons que nous ayons un code donné par $\{a \to 0, b \to 10, c \to 11\}$ et une taille de bloc de décodeur $b = 2$. Cela nous donne quatre blocs possibles, soit $\{00, 01, 10, 11\}$. Comme l'aligment au temps t est quelconque, le premier bloc, 00, peut être aa, la fin de b et a. Le bloc 01 peut être a suivi du début de b ou c, la fin de b et le début de b ou c. Le bloc 10 est b seul ou la fin de c et le début de a. Enfin, le bloc 11 est c seul, ou la fin de c et le début de b ou c. Dépendamment de ce qui est lu avant le bloc courant, on peut reconstruire la séquence originale en maintenant le contexte — c'est-à-dire le passé pertinent de la séquence — grâce à l'automate. Le nouvel état de l'automate est généré en fonction de l'état courant et du bloc lu. Pendant la transition, en fonction de l'état courant et du bloc lu, l'automate émet une série de symboles. C'est ainsi que le décodage est réalisé. Le décodage se poursuit jusqu'à ce que nous atteignions un état acceptant et qu'il ne reste plus de symboles à lire. La fig. 6.6 montre un automate pour le code que nous avons présenté en exemple ici, et le tableau 6.3 montre une forme tabulaire équivalente de l'automate.

Comme vous l'avez probablement compris, le nombre d'états et de transitions croît rapidement en fonction du nombre et de la longueur des codes ainsi que de la taille du bloc. La vitesse de décodage gagnée se paye par la consommation d'une grande quantité de mémoire. On peut contruire l'automate naïvement, mais il sera préférable d'utiliser un algorithme de minimisation d'automate afin de réduire le nombre d'états et de transitions, et par conséquent la quantité de

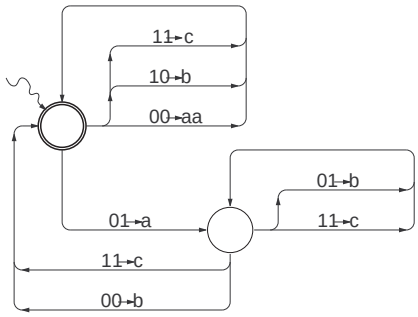


FIG. 6.6 – Un automate de décodage rapide pour le code $\{a \to 0, b \to 10, c \to 11\}$. La flèche ondulante indique l'état initial. L'état acceptant est montré par un double cercle. Les étiquettes sur les flèches de transitions, comme $b_1b_2 \to s$ se lisent « à la lecture de b_1b_2, émettre s ».

État	Lire	Émettre	Aller à
0	00	aa	0
	01	a	1
	10	b	0
	11	c	0
1	00	ba	0
	01	b	1
	10	ca	0
	11	c	1

TAB. 6.3 – Une représentation tabulaire de l'automate de la fig. 6.6.

mémoire utilisée. Nous devons aussi tenir compte du coût de construction de l'automate minimisé pour évaluer la performance du décodeur : la quantité de travail nécessaire à la construction de l'automate n'est pas négligeable.

D'autres auteurs ont proposé de réduire la profondeur de l'arbre de codes de Huffman en augmentant le facteur de branchement. En passant d'un arbre binaire à un arbre quaternaire, on réduit effectivement la profondeur de l'arbre par deux. Cette technique est dûe à Bassiouni et Mukherjee [8]. Ils suggèrent une structure de donnée pour aider au décodage rapide qui n'est autre qu'un automate, bien que le mot automate ne soit pas utilisé. Cheung *et al* proposent un environnement où le décodage rapide est réalisé grâce à un assemblage de circuit à logique programmable, de tables d'index et de génération et compilation de code dynamique [45].

Cependant, aucune de ces techniques ne tient compte des codes adaptatifs. Générer un automate pour un code qui change constamment est impensable, même pour de petits alphabets. Utiliser des circuits à logique programmable n'est guère plus utile car le temps de reconfiguration des circuits les plus rapides demeure dans l'ordre de la seconde (sans compter le temps nécessaire pour la génération de la description du circuit lui-même) rendant ainsi leur utilisation redhibitoire.

6.6 Notes Bibliographiques

Parmi les sujets dont nous n'avons pas discuté, nous avons les codes alphabétiques. Un code alphabétique est un code tel que si a précède lexicographiquement b dans l'alphabet, relation que l'on note habituellement $a \prec b$, alors les codes sont tels que $code(a)$ précède lexicographiquement $code(b)$, lorsque les codes ne sont plus considérés comme des valeurs numériques mais des séquences de symboles de sortie. Notons qu'ici, l'ordre lexicographique auquel nous référons n'a pas la même signification qu'à la section 6.4.2, où nous avons présenté l'algorithme FGK, mais le sens de la définition 3.1.1, p. 25. L'intérêt particulier des codes alphabétiques réside dans le fait que si une séquence dans l'alphabet original s précède une séquence t, alors le code pour s précédera lexicographiquement le code pour t. Les codes pour une séquence, dans ce cas, sont simplement les concaténations des symboles individuels. Cela offre la possibilité de faire des recherches directement dans un index où les mots sont compressés grâce à un code alphabétique. À ce sujet, le lecteur voudra consulter les articles de Gilbert et Moore, *Variable Length Binary Encodings*, de Yeung, *Alphabetic Codes Revisited* et de Hu et Tucker, *Optimum Computer Search Trees and variable Length Alphabetic Codes*, où plusieurs algorithmes sont décrits pour calculer des codes alphabétiques de longueur moyenne minimale [84, 230, 100].

Tous les algorithmes que nous avons présentés supposent que les symboles de sortie, typiquement $\mathbb{B} = \{0, 1\}$, ont un coût égal. Varn présente un algorithme où le coût des symboles de sortie peut différer [213]. Cette situation peut survenir lorsque le canal de communication dépense plus d'énergie pour certains symboles que pour d'autres. On peut fort bien imaginer un mécanisme de communication qui utilise divers niveaux de voltage pour encoder pour émettre ses symboles. Évidemment, on voudra minimiser le temps de communication et l'énergie consommée. L'algorithme de Varn permet d'optimiser le code en tenant compte des coûts relatifs des symboles de sortie. C'est essentiel d'optimiser la consommation lorsqu'on a affaire à des machines de faible consommation comme des ordinateurs de poche, des téléphones cellulaires, etc.

La méthode de production de codes maximalement longs avec une distribution de Fibonacci n'est pas dûe à Campos, et se retrouve chez divers auteurs. Cependant, la méthode serait dûe à Katona et Nemetz qui utilisent les probabilités de forme Fibonacci pour montrer que, pour les symboles de très faibles probabilités, la longueur des codes obtenus par rapport à $-\lg P(a_i)$ est au pire approximativement $1/\lg\phi$ [110].

Chapitre 7

Contributions au codage des entiers

7.1 Introduction

Dans ce chapitre, nous présentons quelques contributions à la compression. En premier lieu, nous présentons une petite modification à l'algorithme des bigrammes qui en améliore la performance de façon non négligeable. Nous discourrons des codes de Golomb et de la façon d'obtenir le paramètre optimal qui minimise la longueur moyenne du code. Nous comparerons notre solution aux solutions qui ont déjà été présentées par d'autres auteurs. Ensuite, nous décrirons dans le détail deux nouvelles classes de codes universels, les codes tabou alignés et les codes tabou non contraints. Nous présenterons les méthodes de génération de code ainsi que les preuves d'universalité. Enfin, nous présentons des algorithmes d'optimisation pour les codes $(Start, Step, Stop)$, dont nous avons déjà parlé à la section 5.3.5.2, ainsi que pour les codes $Start/Stop$, une généralisation des codes $(Start, Step, Stop)$. Les codes $(Start, Step, Stop)$ avaient été introduits par Fiala et Greene [70], lesquels n'avaient pas présenté d'algorithme d'optimisation. Les codes $Start/Stop$ sont beaucoup plus souples que les codes $(Start, Step, Stop)$, en terme des distributions qu'ils sont capable de représenter efficacement. Nous présentons aussi des algorithmes d'optimisation pour ces codes, et nous comparerons les résultats des codes $(Start, Step, Stop)$ et $Start/Stop$ sur des distributions usuelles, telle que les distributions géométrique, exponentielle, Zipf, etc. Enfin, nous montrerons que les codes $Start/Stop$ et $(Start, Step, Stop)$ peuvent être transformés en codes universels efficaces pour les petits entiers.

7.2 Codage bigrammes modifié

Nous avons présenté, à la section 5.3.2.4, la méthode de compression par bigrammes. Si nous avons N symboles possibles dont seulement k sont utilisés pour l'alphabet de base, les $N - k$ symboles restant seront utilisés pour encoder des bigrammes. Les bigrammes sont des paires de symboles, choisies de façon à maximiser la probabilité de les rencontrer dans la séquence à compresser.

Les caractères semblent être approximativement distribués, dans les textes en langue française et anglaise, selon la loi de Zipf. La figure 7.1 montre la distribution des symboles qui composent le texte du chapitre 4 contre une loi de Zipf calculée pour le même nombre de symboles. La conséquence de cette observation, pour le codage en bigramme, c'est que beaucoup de codes

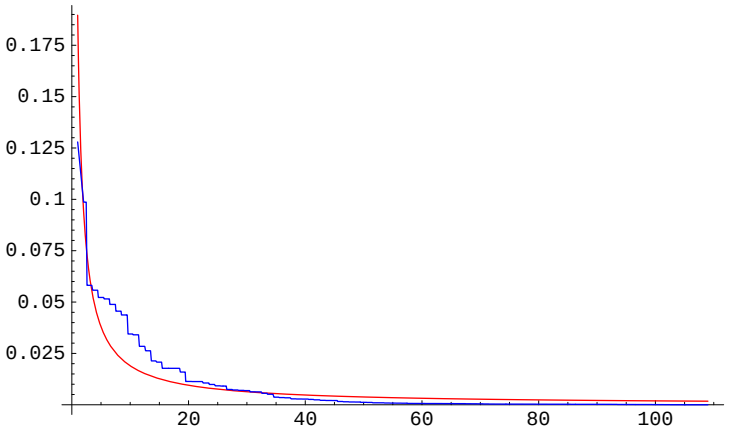


FIG. 7.1 – La distribution des symboles (réordonnés par rang) composant le texte du chapitre 4, y compris les codes LaTeX. La ligne pleine est la distribution de Zipf, et la lignée brisée est la distribution observée.

sont assignés à des symboles qui comptent pour peu dans la distribution. Soit

$$P(Z_n = i) = \frac{1}{iH_n}$$

la distribution de Zipf pour n nombres distincts, où H_n est le n^{e} nombre harmonique, défini par $H_n = \sum_{i=1}^{n} \frac{1}{i}$. Cette définition n'est pas celle que l'on retrouve habituellement pour la loi de Zipf, mais le terme en H_n est nécessaire pour que les probabilités s'additionnent exactement à un, peu importe n. Nous aurons aussi que

$$P(a \leqslant Z_n \leqslant n) = 1 - P(Z_n < a) = \frac{1}{H_n}\left(H_n - H_{a-1}\right)$$

Dans le cas du texte, nous aurons $n \approx 100$, ce qui fait que, pour trouver a tel que $P(a \leqslant Z_n \leqslant n) \approx 0.1$, on a $a \approx 60$. Donc, de 100 symboles distincts on peu dire que nous en avons ≈ 40 qui sont rares et qui ne comptent que pour 10% des symboles observés. Que devons nous en déduire ? Nous devons en déduire que nous avons assigné à des symboles rares des codes qui auraient pu être assignés à des bigrammes, pour le reste possiblement plus fréquents dans la séquence que ces symboles.

L'algorithme de base des bigrammes consiste à voir si les deux prochains symboles dans la séquence forment un bigramme que l'on retrouve dans D, le dictionnaire de bigramme. S'ils s'y trouvent, on émet le code correspondant au bigramme et on avance de deux positions dans la séquence, sinon, on émet le code correspondant au premier symbole et on n'avance que de un dans la séquence. Si nous avons k symboles dans l'alphabet, et N codes disponibles, la taille du dictionnaire de bigrammes est de $N - k$ bigrammes.

Imaginons une modification à l'algorithme des bigrammes. Formons R, l'ensemble des r symboles les plus rares de l'alphabet. Le dictionnaire D sera composé des $N - k + r$ bigrammes les plus fréquents. Le nouvel algorithme procédera ainsi. On vérifie si les deux prochains symboles

de la séquence forment un bigramme que l'on trouve dans D. Si c'est le cas, on émet un des $N-k+r$ codes assignés aux bigrammes. S'ils ne s'y trouvent pas, on vérifie si le premier symbole se trouve dans R. S'il ne s'y trouve pas, on émet un des $k-r-1$ codes réservés aux symboles fréquents. Si le symbole est dans R, on émet un code spécial qui introduira le symbole rare. Le nouvel algorithme assigne au total $N-1$ codes aux bigrammes et aux symboles fréquents et se réserve un code spécial pour introduire un symbole rare.

La fonction de longueur moyenne de code par symbole devient, avec le nouvel algorithme et pour $N = 2^n$,

$$\begin{aligned}\bar{L}'_{bi}(X) = {} & P(X_t^{t+1} \notin D)\left(nP(X_t \notin R \mid X_t^{t+1} \notin D) + 2nP(X_t \in R \mid X_t^{t+1} \notin D)\right) \\ & + \frac{1}{2}nP(X_t^{t+1} \in D)\end{aligned} \tag{7.1}$$

et nous voudrons la minimiser. La difficulté réside dans la détermination de R et dans la modélisation de $P(X_t^{t+1} \in D)$ que nous devons comprendre comme $P(X_t^{t+1} \in D \mid X_0^{t-1})$.

Pour déterminer R, nous pourrions avoir recours à une méthode heuristique, par exemple y mettre les symboles qui comptent pour 10% de la distribution, mais nous allons plutôt chercher à minimiser directement l'éq. (7.1) selon r. Comme nous établissons D de façon heuristique en prenant les bigrammes les plus fréquents sans compter les recouvrements, le choix de r n'affecte D que dans la mesure où le nombre de bigrammes dans le dictionnaire varie, et les bigrammes ajoutés (en augmentant r) sont moins fréquents que ceux qui se trouvent déjà dans le dictionnaire. On estime l'éq. (7.1) en appliquant la compression sur la séquence ou, si la séquence est très longue, sur un bon segment de celle-ci.

L'examen de la fig. 7.2 nous révèle que l'éq. (7.1) est d'apparence globalement convexe mais comporte un certain nombre d'aspérités qui pourrait rendre impossible une optimisation basée seulement sur la propriété de convexité de la fonction : il nous faut oublier les méthodes de type bissection et réduction de l'intervalle qui supposent que la fonction est suffisamment lisse. La rugosité de la fonction provient de $P(X_t^{t+1} \in D \mid X_0^{t-1})$ qui peut varier brusquement au changement du contenu de D ; changer les bigrammes peut changer le découpage de la séquence et faire varier significativement la compression. Même si nous recalculions D de façon exacte après chaque variation de r, il n'est pas clair que la fonction deviendrait lisse en r, pour la même raison. Alors la méthode pour trouver le r adéquat, c'est de chercher exhaustivement le meilleur r. On peut améliorer le temps de recherche en choisissant r de façon à ce que R contienne de 20% à 80% des symboles.

Le tableau 7.1 montre les résultats pour divers fichiers. Les fichiers testés proviennent de divers corpus (consultez l'appendice D pour des descriptions et une discussion détaillées sur le choix et la composition des corpus) et de sources diverses. Les résultats sont très variables. Pour certains fichiers, on obtient une augmentation appréciable, alors que pour d'autres c'est à peine si on observe un changement.

7.3 Codes de Golomb optimaux

Nous avons très brièvement introduit les codes de Golomb à la section 5.3.5.1. Les codes de Golomb sont conçus pour encoder $i \in \mathbb{Z}^*$ tiré selon une loi géométrique de paramètre p. Nous

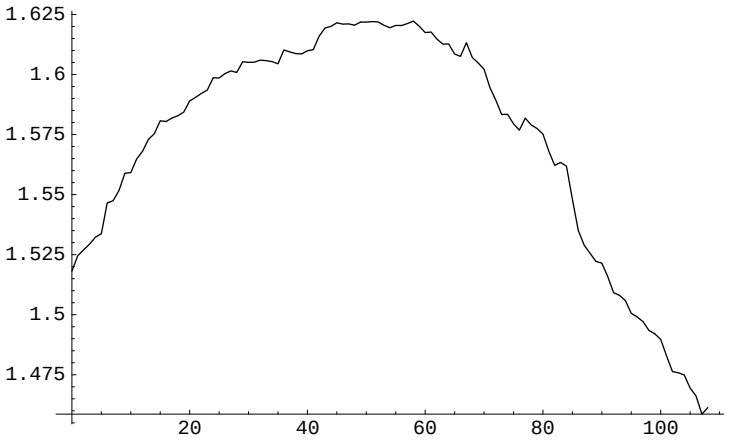


FIG. 7.2 – Les taux de compression obtenus en variant la taille de R, en utilisant le texte du chapitre 4. L'algorithme de base donne un taux de compression de 1.51 :1 et l'algorithme modifié un taux de 1.62 :1, une amélioration de 7%.

Fichier	k	Taux original	r	Taux modifié	Gain
book1 †	82	1.715293	26	1.778912	3.7%
book2 †	96	1.560894	40	1.637754	4.9%
pic †	159	1.895073	93	1.922755	1.4%
alice29.txt ‡	74	1.796553	8	1.804054	0.3%
bible.txt ‡	63	1.781656	10	1.794279	0.7%
3mousq.txt *	100	1.776060	34	1.821047	2.5%
jar_ubur.txt *	91	1.634786	32	1.748831	7.0%
paradigmes.tex	109	1.518154	58	1.622240	6.9%
pensees.txt	104	1.717568	31	1.778933	3.6%

TAB. 7.1 – Les résultats de l'algorithme de bigrammes modifié. † Calgary corpus. ‡ Canterbury Corpus. * Corpus ABU. Le fichier paradigmes.tex est le source LaTeX du chapitre 4. Le fichier pensees.txt est une liste de maximes, de proverbes et de mots d'esprits. Sauf pour le fichier pic, qui est une image, tous les fichiers sont des fichiers texte.

décrirons, dans cette section, les différentes formes des codes de Golomb et comment déterminer le paramètre optimal qui minimise la longueur moyenne des codes selon la forme choisie.

7.3.1 La distribution géométrique

La probabilité d'occurence de $i \in \mathbb{Z}^*$ est donnée par

$$P(X = i) = (1-p)^i p$$

Pour découler nos résultats, nous utiliserons

$$P(X \leqslant i) = 1 - (1-p)^{i+1}$$

d'où on découle

$$P(a \leqslant X \leqslant b) = (1-p)^{a+1} - (1-p)^{b+1}$$

une autre identité qui nous sera fort utile. L'espérance est donnée par $E[X] = \frac{1}{p} - 1$, et l'entropie par

$$\mathcal{H}(X) = \frac{1}{p}\,(\ (p-1)\lg(1-p) - p\lg p\ )$$

Notons que nous utilisons le paramètre p, et non θ ou $q = (1-p)$ comme le font les auteurs auxquels nous faisons référence. La notation basée sur le seul paramètre p (plutôt que $\theta = (1-p)$) est plus près de ce que l'on retrouve dans la littérature des probabilités et des statisques.

7.3.2 Structure du code

Golomb propose une structure de code pour les nombres tirés selon une loi géométrique. De par la nature de la variable aléatoire, on doit avoir un code qui peut représenter des nombres d'une magnitude quelconque, sans limite. Golomb propose un code contrôlé par un paramètre m composé d'un préfixe unaire suivi de $\lceil \lg m \rceil$ bits qui encodent le nombre. Pour $i \in \mathbb{Z}^*$, le code est donné par

$$G_m(i) = C_\alpha(\lfloor \tfrac{i}{m} \rfloor) : C_{\beta(\lceil \lg m \rceil)}(i \bmod m) \tag{7.2}$$

où $C_\alpha(x)$ est le code unaire de x et $C_{\beta(b)}(x)$ le code binaire de x sur exactement b bits. Ce code a une fonction de longueur

$$|G_m(i)| = L_m(i) = 1 + \lceil \lg m \rceil + \left\lfloor \frac{i}{m} \right\rfloor \tag{7.3}$$

donc une fonction de longueur moyenne

$$\bar{G}_m(X) = \sum_{i=0}^{\infty} P(X = i) L_m(i) \tag{7.4}$$

Pour décoder le nombre, il suffit de lire le préfixe, ce qui nous donne $\lfloor \frac{i}{m} \rfloor$, suivi de la lecture de $\lceil \lg m \rceil$ bits, qui nous donnent $i \bmod m$, pour enfin trouver que $i = \lfloor \frac{i}{m} \rfloor m + i \bmod m$. Ce code revient à partitionner la fonction de probabilité en intervalles de longueurs égales. Les nombres contenus dans l'intervalle j ont un code dont le préfixe est de longueur $j+1$ suivi de $\lceil \lg m \rceil$ bits

qui encodent l'index des nombres dans cet intervalle.

Cette structure de code, telle que proposée par Golomb, gaspille un certain nombre de bits dans les suffixes. En effet, les préfixes introduisent des nombres de 0 à $m-1$ mais on utilise $\lceil \lg m \rceil$ bits plutôt qu'exactement $\lg m$ bits. On peut soit contraindre m à être de la forme 2^k, c'est-à-dire à être une puissance de 2, ou encore on peut utiliser des codes *phase-in* pour encoder $i \bmod m$ en utilisant très près de $\lg m$ bits en moyenne.

7.3.3 Solutions précédentes

Les solutions proposées précédemment (tout comme les solutions que nous proposons) ne minimisent pas toutes les mêmes fonctions de longueur de code. La longueur d'un code, selon Golomb, est donnée par l'éq. (7.3). Gallager et van Voorhis utilisent aussi cette fonction de longueur mais utiliseront une approche très différente pour déduire le paramètre optimal.

Golomb propose la solution $m = \lceil -1/\lg p \rceil$ ce qui maximise le contenu d'information dans les bits du préfixe en s'assurant que la probabilité qu'un nombre soit dans le j^{e} intervalle est $\approx (\frac{1}{2})^{j+1}$. Cette solution ne minimise pas la longueur moyenne du code, mais seulement la longueur moyenne du préfixe. Cette solution est meilleure lorsque $p = 2^{-s}$, pour $s \in \mathbb{N}$.

Gallager et van Voorhis proposent une solution basée sur les codes de Huffman [77]. Cette solution repose sur le calcul de classes d'équivalence sur les nombres, où chaque nombre dans une même classe reçoit un code de la même longueur. Leur solution minimise l'éq. (7.14). Ils trouvent qu'un $m \in \mathbb{N}$ satisfaisant $(1-p)^m + (1-p)^{m+1} \leqslant 1 < (1-p)^{m-1} + (1-p)^m$ est la solution optimale. Serroussi *et al.* proposent une paramétrisation différente (mais équivalente) de cette solution, soit $m = \lceil \ln(2-p)/\ln((1-p)^{-1}) \rceil$ [186].

7.3.4 Solutions proposées

Dans cette section, nous présentons trois solutions au problème des codes de Golomb. D'abord une solution exacte pour les codes de la forme de l'éq. (7.2) qui ont une fonction de longueur moyenne donnée par l'éq. (7.4), fonction que nous expliciterons dans un instant. Ensuite une solution particulière pour le cas où on contraint m à être de la forme 2^k, c'est-à-dire à être une puissance de deux. Enfin, nous présentons la fonction de longueur moyenne des codes lorsque les suffixes sont encodés grâce à un code *phase-in*, pour laquelle le paramètre optimal est obtenu par la méthode de Gallager et Seroussi.

Les trois variantes de code que nous allons résoudre sont donc :

$$G_m(i) = C_\alpha(\lfloor \tfrac{i}{m} \rfloor) : C_{\beta(\lceil \lg m \rceil)}(i \bmod m) \tag{7.5}$$

$$G_{2^b}(i) = C_\alpha(\lfloor \tfrac{i}{2^b} \rfloor) : C_{\beta(b)}(i \bmod 2^b) \tag{7.6}$$

et

$$G_{\phi(m)}(i) = C_\alpha(\lfloor \tfrac{i}{m} \rfloor) : C_{\phi(m)}(i \bmod m) \tag{7.7}$$

où $C_{\phi(m)}(i)$ est le code *phase-in* pour $i \in \{0, \ldots, m-1\}$. Les codes *phase-in* sont présentés à la section 5.3.4.2.

7.3.4.1 Solutions pour $G_m(i)$

Nous allons résoudre pour m dans l'éq. (7.4). La première étape est d'expliciter cette fonction. La présence de plafonds et de planchers nécessite un peu plus de travail que nous avions affaire seulement à $\lg m$ et $\frac{i}{m}$. La fonction de longueur moyenne, pour p et m est donnée par :

$$\begin{aligned}
\bar{G}_m(X) &= \sum_{i=0}^{\infty} P(X=i) L_m(i) \\
&= \sum_{i=0}^{\infty} P(X=i) \left(1 + \lceil \lg m \rceil + \left\lfloor \frac{i}{m} \right\rfloor\right) \\
&= 1 + \lceil \lg m \rceil + \sum_{i=0}^{\infty} P(X=i) \left\lfloor \frac{i}{m} \right\rfloor \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} \sum_{i=jm}^{(j+1)m-1} P(X=i) j \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} j \sum_{i=jm}^{(j+1)m-1} P(X=i) \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} j P(jm \leqslant X \leqslant (j+1)m-1) \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} j \left(P(X \leqslant (j+1)m-1) - P(X \leqslant jm-1)\right) \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} j \left((1-(1-p)^{(j+1)m}) - (1-(1-p)^{jm})\right) \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} j \left((1-p)^{jm} - (1-p)^{(j+1)m}\right) \qquad (7.8) \\
&= 1 + \lceil \lg m \rceil + \sum_{j=0}^{\infty} j \left(a^j - a^{j+1}\right) \qquad \text{en posant } a = (1-p)^m \\
&= 1 + \lceil \lg m \rceil + \left(\sum_{j=0}^{\infty} j a^j\right) - \left(a \sum_{j=0}^{\infty} j a^j\right) \\
&= 1 + \lceil \lg m \rceil + (1-a) \sum_{j=0}^{\infty} j a^j \\
&= 1 + \lceil \lg m \rceil + (1-a) \frac{a}{(1-a)^2} \qquad \text{par l'identité de Gabriel} \\
&= 1 + \lceil \lg m \rceil + \frac{a}{1-a} \\
&= 1 + \lceil \lg m \rceil + \frac{(1-p)^m}{1-(1-p)^m} \\
&= 1 + \lceil \lg m \rceil - 1 + \frac{1}{1-(1-p)^m} \\
&= \lceil \lg m \rceil + \frac{1}{1-(1-p)^m}
\end{aligned}$$

On pourrait chercher à résoudre l'éq. (7.8) en remplaçant $\lceil \lg m \rceil$ par simplement $\lg m$ et résoudre la dérivée

$$\frac{\partial \bar{G}_m(X)}{\partial m} = \frac{1}{m \ln 2} + \frac{(1-p)^m \ln(1-p)}{(1-(1-p)^m)^2} = 0$$

pour m. Dans cette équation, il n'est pas possible d'isoler m de façon à poser $m = f(p)$. Cependant, comme cette équation est lisse et n'a qu'un seul zéro, il est facile de le trouver très

rapidement grâce à une méthode de bissection et réduction. Puisque l'éq. (7.8) n'est pas très coûteuse à calculer, on peut chercher le m optimal par un algorithme de recherche par raffinement s'apparentant à la recherche par interpolation [196, p.137]. À partir d'une première estimation $\hat{m} = \lceil \ln(2-p)/\ln((1-p)^{-1}) \rceil$, cet algorithme permet de trouver le m optimal en $O(\lg \lg m)$ étapes.

La fig. 7.3 montre les résultats obtenus par la méthode de Golomb, de Gallager et van Voorhis et notre solution par recherche par raffinement.

7.3.4.2 Solutions pour $G_{2^b}(i)$

On peut vouloir, pour des raisons d'efficacité d'implémentation, ne considérer que des $m \sim 2^b$. Le calcul de $i \bmod 2^b$ se fait très efficacement en logiciel ou en matériel, car $i \bmod 2^b$ revient à ne conserver que les b bits de poids faible de i. Similairement, le calcul de $\lfloor i/2^b \rfloor$ revient à éliminer les b bits de poids faible de i. Le code est donné par $G_{2^b}(i)$, soit l'éq. (7.6).

Nous allons chercher à minimiser directement la longueur moyenne du code selon p. On pourrait être tenté de remplacer $\lfloor i\,2^{-b} \rfloor$ dans l'éq. (7.3) par $i\,2^{-b}$ pour obtenir une fonction plus simple à optimiser. Avec cette relaxation, nous trouvons, pour un b donné,

$$\begin{aligned} \bar{G}'_{2^b}(X) &= \sum_{i=0}^{\infty} P(X=i) L'_{2^b}(i) \\ &= \sum_{i=0}^{\infty} P(X=i) \left(1 + b + \frac{i}{2^b}\right) \\ &= 1 + b + \frac{1}{2^b} \sum_{i=0}^{\infty} i P(X=i) \\ &= 1 + b + \frac{1}{2^b} E[\,X\,] \\ &= 1 + b + \frac{1}{2^b}\left(\frac{1}{p} - 1\right) \end{aligned} \tag{7.9}$$

C'est une fonction concave pour $b \in [0, \infty)$, qui a une dérivée convexe en b ayant un zéro unique. En résolvant

$$\frac{\partial \bar{G}'_b(X)}{\partial b} = 1 - \frac{1}{2^b}\left(\frac{1}{p} - 1\right) \ln 2 = 0$$

pour b, on trouve l'approximation du paramètre optimal

$$\hat{b} = \frac{\ln\left(\left(\frac{1}{p} - 1\right) \ln 2\right)}{\ln 2} \tag{7.10}$$

Comme la fonction décrite par l'éq. (7.9) n'est pas symétrique par rapport à son minimum, il faudra prendre soin de déterminer si $\lfloor\, \hat{b}\, \rfloor$ ou $\lceil\, \hat{b}\, \rceil$ donne la meilleure longueur moyenne. En pratique, cette approximation est près de la solution optimale.

Nous avons obtenu une solution approximative en relâchant les contraintes, c'est-à-dire en approximant la fonction de longueur par une fonction qui a son terme $\lfloor i2^{-b} \rfloor$ remplacé par $i2^{-b}$.

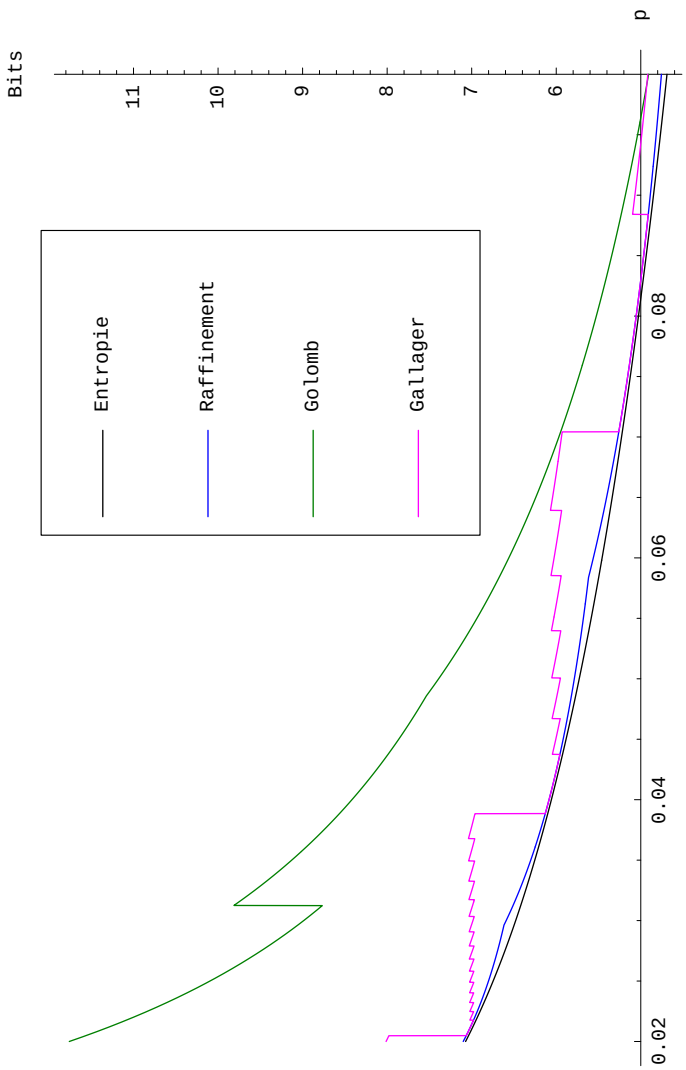


FIG. 7.3 – Comparaisons des solutions pour $G_m(i)$. Sous toutes les courbes, nous avons l'entropie, juste au dessus, la solution trouvée par perturbation. La fonction en dents de scie est la longueur moyenne optenue grâce à la solution de Gallager et van Voorhis. Enfin, dominant toutes les courbes, la solution très approximative de Golomb.

Nous pouvons cependant calculer la longueur moyenne exacte :

$$\begin{aligned}\bar{G}_{2^b}(X) &= \sum_{i=0}^{\infty} P(X=i) L_{2^b}(i) \\ &= \sum_{i=0}^{\infty} P(X=i)\left(1+b+\left\lfloor \frac{i}{2^b} \right\rfloor\right) \\ &= b + \frac{1}{1-(1-p)^{2^b}}\end{aligned} \tag{7.11}$$

La dérivation l'éq.(7.11) suit le même type de développements qu'avec la dérivation de l'éq. (7.8), en remplaçant m par 2^b. Cette fonction, comme l'éq. (7.9), est concave et a une dérivée convexe n'ayant qu'un seul zéro. Nous voulons résoudre

$$\frac{\partial \bar{G}_{2^b}(X)}{\partial b} = 1 + \frac{2^b (1-p)^{2^b} \ln 2 \ln(1-p)}{(1-(1-p)^{2^b})^2} = 0 \tag{7.12}$$

pour b. Cette formulation ne permet pas d'isoler b. Cependant, cette fonction est très lisse et une méthode de bissection et raffinement successif peut être utilisée pour résoudre l'éq. (7.12) efficacement et obtenir b, la valeur optimale. Comme b est contraint à être un entier, par la définition du code, on devra avoir recours à la fonction

$$\Theta(b,p) = \begin{cases} \lfloor b \rfloor & \text{si } \bar{G}_{2^{\lfloor b \rfloor}}(X) \leqslant \bar{G}_{2^{\lceil b \rceil}}(X) \\ \lceil b \rceil & \text{sinon} \end{cases} \tag{7.13}$$

qui nous dit le quel de $\lfloor b \rfloor$ ou $\lceil b \rceil$ nous donne la meilleure longueur moyenne pour le code.

Sur la figure 7.4, les deux solutions, pour b et pour $\hat{b}$ semblent équivalentes. Or, il existe des régions où réponses sont différentes. Par exemple, pour $p = \frac{1}{10}$, on trouve $\hat{b} \approx 2.64116$ et $b \approx 2.65771$. Bien qu'après avoir décidé de la valeur entière à prendre grâce à l'éq. (7.13), on optienne le même b, donc la même longueur moyenne de code, il n'en est pas toujours ainsi, surtout dans les régions où p est très petit.

7.3.4.3 Solution pour $G_{\phi(m)}(i)$

Enfin, nous nous attaquons aux codes dont le suffixe est encodé par un code *phase-in*. Cette solution est souvent trouvée dans la littérature, et m est obtenu grâce à la solution de Gallager et van Voorhis. La dérivation de la fonction de la longueur moyenne est passablement plus compliquée que pour les deux autres variations.

Rappelons l'éq. (5.6) qui donne la longueur d'un code *phase-in* :

$$L_{\phi(m)}(i) = \begin{cases} k & \text{si } i < 2^k - b \\ k+1 & \text{sinon} \end{cases}$$

où k et b sont tels que $2^k + b = m$. Pour découler la longueur moyenne des codes étant donnés

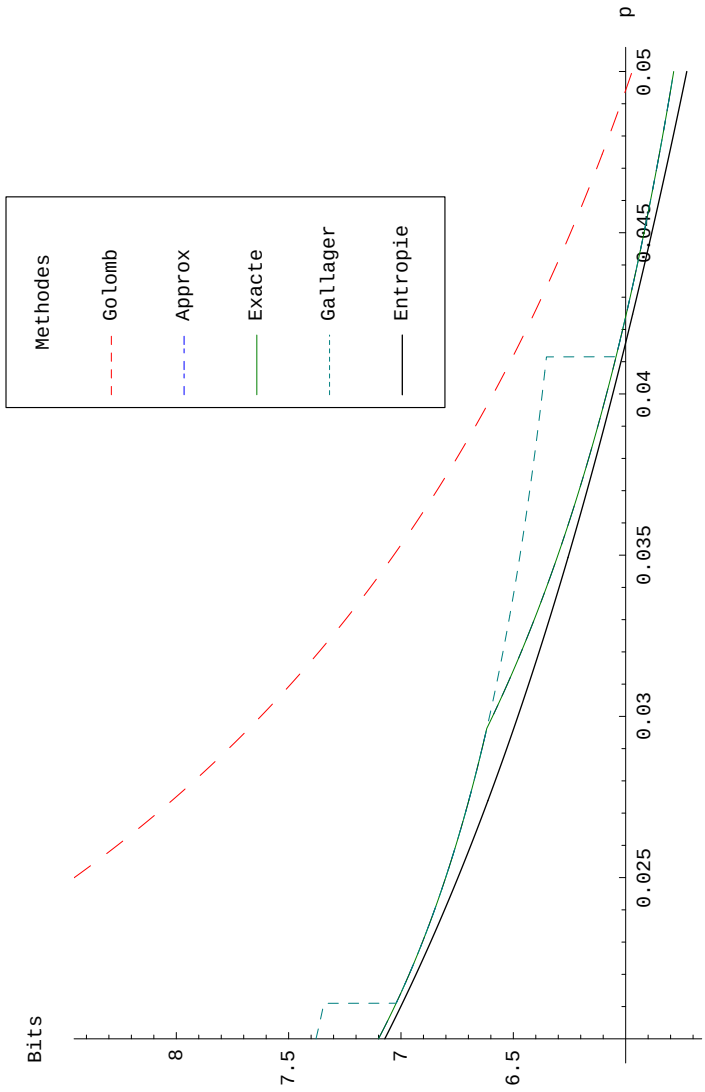


FIG. 7.4 – Les longueurs moyennes des codes données par les différentes méthodes comparées à l'entropie de la source. Pour trouver b grâce à la méthode de Gallager et van Voorhis, on a utilisé $b = \lceil \lg m \rceil$, où m est estimé par $m = \ln(2-p)/\ln((1-p)^{-1})$ (sans l'arrondi). Pour la méthode de Golomb, on a $b = \lceil \lg(-\lg p) \rceil$. Au dessus de toutes les méthodes, en grosses hachures, donc la pire, c'est la méthode de Golomb. En escaliers, en petites hachures, c'est la méthode de Gallager et van Voorhis. En vagues, près de l'entropie, se trouvent les deux solutions proposées — soient les solutions obtenues des équations (7.10) et (7.11) — l'une exacte et l'autre approximative. Après l'application de $\Theta(\cdot,\cdot)$, les deux donnent essentiellement les mêmes longueurs moyennes, bien qu'elles diffèrent en certains endroits. En dessous de toutes, en ligne pleine, l'entropie.

m et p, nous allons suivre la même tactique que pour les autres formules déjà découlées.

$$\begin{aligned}
\bar{G}_{\phi(m)}(X) &= \sum_{i=0}^{\infty} P(X=i) L'_{\phi(m)}(i \bmod m) \\
&= \sum_{i=0}^{\infty} P(X=i) \left(1 + \left\lfloor \frac{i}{m} \right\rfloor + L_{\phi(m)}(i \bmod m)\right) \\
&= 1 + \left(\sum_{i=0}^{\infty} P(X=i) \left\lfloor \frac{i}{m} \right\rfloor\right) + \left(\sum_{i=0}^{\infty} P(X=i) L_{\phi(m)}(i \bmod m)\right) \\
&= 1 + \frac{1}{1-(1-p)^m} - 1 + \sum_{i=0}^{\infty} P(X=i) L_{\phi(m)}(i \bmod m) \\
&= \frac{1}{1-(1-p)^m} + \sum_{j=0}^{\infty} \sum_{i=jm}^{(j+1)m-1} P(X=i) L_{\phi(m)}(i \bmod m) \\
&= \frac{1}{1-(1-p)^m} + \left(\sum_{j=0}^{\infty} \sum_{i=jm}^{jm+2^k-b-1} P(X=i) k\right) + \left(\sum_{j=0}^{\infty} \sum_{i=jm+2^k-b}^{(j+1)m-1} P(X=i)(k+1)\right) \\
&= \frac{1}{1-(1-p)^m} + k\frac{1-(1-p)^{2^k-b}}{1-(1-p)^m} + (k+1)\frac{(1-p)^{2^k-b} - (1-p)^m}{1-(1-p)^m} \\
&= 1 + k - \frac{(1-p)^{2^k-b}}{(1-p)^m - 1}
\end{aligned} \tag{7.14}$$

Si $m \sim 2^k$, on retombe sur l'éq. (7.11). L'éq. (7.14) ne permet pas que l'on utilise sa dérivée pour trouver le minimum. En effet, il faut composer avec k et b qui sont des entiers qui dépendent de m. La fig. 7.5 présente les résultats obtenus par la méthode de Gallager et van Voorhis et de Golomb. Or, il se trouve que la solution de Gallager et van Voorhis donne le résultat exact dans le cas particulier où les suffixes sont des codes *phase-in*. Cette figure montre aussi les résultats obtenus par l'algorithme de raffinements successifs qui minimise l'éq. (7.14).

7.3.4.4 Adaptativité

Lorsque nous codons une liste d'entiers tirés selon une loi (que l'on présume) géométrique, on commence par obtenir une estimation $\hat{p}$ du paramètre p. Cette estimation peut être obtenue grâce à l'estimateur maximum de vraisemblance

$$\hat{p} = \frac{1}{1 + \frac{1}{n}\sum_i x_i}$$

ce qui est tout à fait raisonnable puisque $\bar{X} = \frac{1}{p} - 1$ et $\bar{X} = \frac{1}{n}\sum_i x_i$. Pour estimer p, il suffit donc de ne conserver qu'un compteur n et la somme des observations, $\sum_i x_i$. L'obtention du paramètre optimal (m ou b, selon la structure du code choisie) peut être réalisée soit grâce à l'une ou l'autre des méthodes décrites dans cette section, soit par l'intermédiaire d'une table où seulement un certain nombre de valeurs de p sont représentées.

Posons $f(p)$, la fonction qui donne le paramètre optimal considérant p et la forme choisie pour le code. Puisque $f(p)$ est un nombre entier, on sait que plusieurs valeurs de p (en fait, une infinité) nous donneront la même valeur pour le paramètre optimal. Si on calcule les intervalles $[p, p+\varepsilon(p))$ qui donnent les mêmes valeurs pour le paramètre optimal, on peut construire une table dont la clef est l'intervale $[p, p+\varepsilon(p))$ et la valeur contenue à cette adresse est $f(p)$. Ainsi,

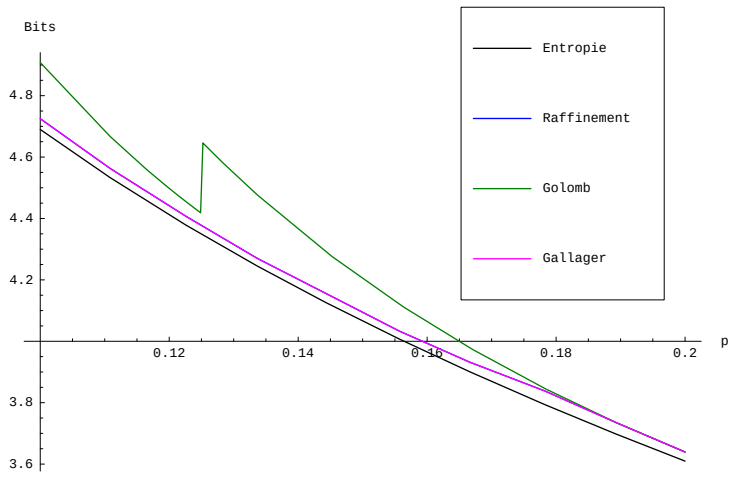


FIG. 7.5 – Les longueurs moyennes des codes données par les différentes méthodes comparées à l'entropie de la source, lorsque le suffixe du code est un code *phase-in*. La solution trouvée par raffinement est la même que celle trouvée par la méthode de Gallager et van Voorhis. Ici encore, la solution de Golomb est la pire.

il devient très facile de déterminer dans quel intervalle se trouve notre estimateur $\hat{p}$ et de lire la valeur correspondante de $f(p)$ dans la table. Encore une fois, un algorithme de type bissection et réduction peut nous aider à calculer cette table. Cette table étant statique, on peut se contenter de la calculer une bonne fois pour toute et l'utiliser sans la modifier. La façon dont $\hat{p}$ change n'affecte pas le contenu de la table, seule la forme du code choisie aura un impact.

Une considération importante est la taille de la table. D'après les formes des expressions pour les longueurs moyennes, on déduit que lorsque $p \to 0$, on a aussi que $\varepsilon(p) \to 0$, donc que le nombre d'entrées dans la table croît rapidement lorsqu'on s'approche de $p = 0$. Comme $p = 0$ n'est pas une valeur admissible (car $0 < p \leqslant \frac{1}{2}$) on peut limiter p selon la mémoire que l'on considère admissible d'allouer pour la table. On peut décider arbitrairement de contraindre $10^{-4} < p \leqslant \frac{1}{2}$. Dans le cas où les codes sont contraints à la forme de l'éq. (7.6), la table est contrainte à ne contenir au plus 16, 32, voire 64 entrées, car le paramètre b représente le nombre de bits contenus dans le suffixe, qui est limité par les capacités arithmétiques du processeur considéré. Pour les deux autres formes, m n'est qu'un entier quelconque. Cependant, les tables ne sont pas nécessairement bien grandes. Pour les codes de la forme de l'éq. (7.7), c'est-à-dire où les suffixes sont des codes *phase-in*, en posant $\varepsilon(p) = 10^{-k}$, pour $k = 1, \ldots, 6$, on trouve des tables ayant 3, 13, 49, 164, 523 et 1663 entrées. On obtient des tables avec un nombre d'entrées comparable lorsque on utilise plutôt les codes de la forme de l'éq. (7.5).

Une fois cette table précalculée, l'adaptativité se résume à comparer notre paramètre $\hat{p}$ avec les bornes du tableau et y lire le paramètre optimal associé. Cela n'implique pas que nous cherchions la table en $O(\lg n)$ à que coup. Il nous suffit de maintenir un index sur l'intervalle courant et vérifier si $\hat{p}$ change d'intervalle. Pour ce faire, il suffit de vérifier les bornes inférieure et supérieure de l'intervalle, ce qui se fait en $O(1)$.

7.3.4.5 Résultats et considérations numériques

Les figures 7.3, 7.4 et 7.5 comparent graphiquement les résultats obtenus par les différents estimateurs et solutions sur les différents types de codes. Chaque graphique utilise, comme fonction de longueur moyenne, cette qui est idoine à la forme du code considéré.

Les codes de la forme de l'éq. (7.6) demandent que l'on utilise soit le paramètre approximatif $\hat{b}$, calculé grâce à l'éq. (7.10), soit que l'on trouve le zéro de l'éq. (7.12). L'algorithme utilisé pour trouver le zéro de l'éq. (7.12) est un algorithme de bissection et réduction d'intervalle s'apparentant à la méthode Newton-Raphson [171, pp. 350–366]. Cette méthode converge très rapidement et est facile à implémenter. L'algorithme pour trouver le paramètre optimal pour les codes de la forme de l'éq. (7.5), est un algorithme par raffinement sucessif. Comme la fonction de longueur, l'éq. (7.8) est concave, on peut choisir une valeur initiale de m (par exemple, donnée par la méthode de Gallager et van Voorhis) et explorer la région, par exemple avec une modification de la recherche dicotomique, pour trouver le minimum. Comme m est un entier, le nombre de valeurs à inspecter est limité à $O(\lg m)$. La solution pour m, avec les codes à la forme de l'éq. (7.7), est donnée par $m = \lceil \ln(2-p)/\ln((1-p)^{-1}) \rceil$. Cette formulation peut poser des problèmes numériques, dépendamment de la fiabilité de $\ln x$. Cette classe de routines (ln, exp, etc.) est connue pour avoir un comportement bizarre et imprévisible lorsque les arguments sont extrêmement petits ou extrêmement grands [171].

❁ ❁ ❁

Ces résultats ont été soumis aux *Information Processing Letters*, et sont en attente de revue.

7.4 Codes tabou

Les méthodes de codage universel que nous avons présentées au chapitre 5 aux sections 5.3.3.3 et 5.3.3.4 utilisent un encodage récursif de la longueur de l'entier qui précède l'entier lui-même, alors qu'à la section 5.3.3.5 le codage de Fibonacci utilise une caractéristique de la décomposition duale de Zeckendorf pour produire des représentations uniquement décodable des entiers.

Dans cette section, nous proposons deux nouvelles classes de codes universels. La première classe, ce sont les codes tabou alignés. Ces codes sont composés d'une série de blocs de n bits terminés par un motif spécial, réservé, le *tabou*. Le tabou peut être n'importe la quelle des 2^n combinaisons de n bits, car, à cause des contraintes des blocs, la reconnaissance du tabou ne peut être ambiguë. Toutefois, sans perte de généralité, nous choisirons une série de n zéros comme tabou. Nous démontrerons que ces codes sont universels. La seconde classe de code n'est pas contrainte à être formée de blocs n bits, donc à avoir une longueur qui est un multiple de n, et le tabou délimite une séquence de bits de longueur quelconque. Pour cette raison, cette seconde classe est dite non contrainte.

❁ ❁ ❁

Ces résultats ont été soumis au *SIAM Journal on Computing*. Le SIAM (*Society of Industrial & Applied Mathematics*) publie plusieurs journaux dont le sujet est l'application des mathématiques dans le contexte industriel ou simplement pratique.

7.4.1 Codes tabou alignés

Considérons d'abord la classe des codes alignés. Ces codes sont composés d'une série de blocs de n bits terminée par un bloc spécial de n bits, le tabou. Sans perte de généralité, le tabou sera une série de n zéros consécutifs. La série de n bits formant le tabou pourrait être n'importe quelle des 2^n combinaisons possibles car, dûe à la contrainte d'alignement — on lit n bits à la fois — le décodage du motif tabou ne peut pas être ambigu. Nous verrons cependant, à la section suivante, pourquoi le taboo sera préférablement une séquence de zéros consécutifs.

Soit t, le motif tabou, d'une longueur de n bits. Contraignons les codes à être de la forme $\langle\, kn \,\rangle : t$, où $\langle\, n \,\rangle$ dénote toujours une chaîne (non spécifiée) de n bits et : la concaténation, et avec $k \geqslant 1$. Le tableau 7.2 illustre la structure générale du code.

On remarquera que la somme $\sum_{i=1}^{k}(2^n-1)^i$ qui borne les intervalles, comme montré au tableau 7.2, est familière. Il s'agit en effet d'une série géométrique finie. Soit

$$g_n(k) = \sum_{i=1}^{k}(2^n-1)^i = \frac{((2^n-1)^k-1)(2^n-1)}{2^n-2}$$

Pour trouver le nombre de blocs nécessaire à l'encodage d'un entier $m \in \mathbb{Z}^*$, nous devons déterminer quel intervalle contient cet entier m. L'index de l'intervalle est donné par $k_n(m) = \min\{\, k \mid m < g_n(k) \,\}$. À partir de n, $g_n(k)$ et m, on résoud pour k :

$$\begin{aligned} m &\leqslant g_n(k) - 1 \\ m &\leqslant \frac{((2^n-1)^k-1)(2^n-1)}{2^n-2} \\ k &\geqslant \frac{\lg\left(m+2-\frac{m+1}{2^n-1}\right)}{\lg(2^n-1)} \end{aligned}$$

et, pour satisfaire l'inégalité $m < g_n(k)$, on obtient

$$k_n(m) = \left\lceil \frac{\lg\left(m+2-\frac{m+1}{2^n-1}\right)}{\lg(2^n-1)} \right\rceil$$

Le code pour m contiendra $k_n(m)+1$ blocs de n bits, pour une longueur de $n\,(k_n(m)+1)$ bits. Le code pour l'entier m est, fondamentalement, la représentation base (2^n-1) de $m - g_n(k_n(m)-1)$. La représentation en base (2^n-1) requiert des « chiffres » de n bits de long. Étant donnés n, m et $c = m - g_n(k_n(m)-1)$, les blocs individuels du code sont donnés par :

$$b_i = (\lfloor c(2^n-1)^{-i} \rfloor \mod (2^n-1)) + 1, \quad i = 0, 1, \ldots, k_n(m)-1 \tag{7.15}$$

et le $(k_n(m)+1)^{\text{e}}$ bloc, $b_{k_n(m)}$, est le tabou qui délimite le code. Le décodage de m est obtenu grâce à

$$m = g_n(k-1) + \sum_{i=0}^{k-1}(b_i - 1)(2^n-1)^i \tag{7.16}$$

et k est récupéré en comptant le nombre de blocs lus avant d'atteindre le bloc tabou. Nous encodons seulement $c = m - g_n(k_n(m) - 1)$ car, connaissant k, il ne nous reste qu'à encoder l'index de m dans le k^{e} intervalle. Enfin, les ajustements à +1 et -1 aux éqs. (7.15) et (7.16) tiennent compte du tabou, qui est, considéré numériquement, le zéro.

Ce code gaspille $n + k(n - \lg(2^n - 1))$ bits, en comptant le tabou qui le termine. La partie $n - \lg(2^n - 1)$ est négligeable, comme le montrera la preuve d'universalité qui suit. Le ratio asymptotique entre la longueur du code tabou aligné pour m et $\lg m$, pour $n \geqslant 2$, est donné par

$$\begin{aligned}
\lim_{m\to\infty} \frac{(k(m)+1)n}{\lg m} &= \lim_{m\to\infty} \frac{n\left(\left\lceil \frac{\lg\left(m+2-\frac{m+1}{2^n-1}\right)}{\lg(2^n-1)} \right\rceil + 1\right)}{\lg m} \\
&\leqslant \lim_{m\to\infty} \frac{n\left(\frac{\lg\left(m+2-\frac{m+1}{2^n-1}\right)}{\lg(2^n-1)} + 2\right)}{\lg m} \\
&= \lim_{m\to\infty} \frac{n\left(\lg\left(m+2-\frac{m+1}{2^n-1}\right) + 2\right)}{\lg m \ \lg(2^n-1)} \\
&= \lim_{m\to\infty} \frac{n}{\lg(2^n-1)} \frac{\lg\left(m+2-\frac{m+1}{2^n-1}\right) + 2}{\lg m} \\
&= \frac{n}{\lg(2^n-1)}
\end{aligned}$$

démontrant ainsi la $\frac{n}{\lg(2^n-1)}$-universalité des codes. Rappelons que la définition d'universalité se retrouve à la section 5.3.3.1, page 93. Puisque $\frac{n}{\lg(2^n-1)} > 1$ pour tout $n \in \mathbb{N}$, ces codes ne sont pas strictement universels, c'est-à-dire 1-universel, au sens déjà défini. Cependant, en choisissant n moindrement grand, nous pourrons nous approcher de 1 aussi près que nous le désirons.

En choisissant $n = 2$, nous obtenons un code qui est $\frac{2}{\lg 3}$- ou 1.26-universel. Ce code est montré au tableau 7.3. Déjà avec $n = 4$, nous obtenons un code qui est 1.02-universel. Le cas spécial $n = 1$ nous donne le codage unaire, soit une série de uns suivi d'un zéro final. Ce code n'est pas universel, mais nous trouvons que $\frac{n}{\lg(2^n-1)} = \frac{1}{0} = \infty$, ce qui est cohérent avec l'énoncé d'universalité du code et avec la définition même d'universalité.

Le fait que les codes soient contraints à être composés de blocs de n bits permettra de procéder à un encodage et à un décodage rapides. Les équations (7.15) et (7.16) se prêtent bien à des implémentations à la fois efficaces et simples. Par exemple, le décodeur peut fort bien lire un bloc à la fois, tout en maintenant un état interne qui donne la somme $m - g_n(k(m) - 1)$ à la lecture du dernier bloc. La fonction $g_n(k)$ n'est pas très exigeante en terme de la quantité de calculs qui sont nécessaires. De plus, on peut choisir n pour être une taille qui est compatible avec les capacités du processeur et obtenir une implémentation qui tire avantage de l'organisation naturelle de la mémoire et du processeur.

❁ ❁ ❁

Il existe un lien fort entre la façon dont nous écrivons les nombres ordinairement et les codes tabou alignés. Quand nous écrivons, par exemple, le nombre 42, nous supposons qu'il suffit

Codes	Nombre de valeurs	Longueur	Intervalle
$\langle n \rangle : t$	$2^n - 1$	$2n$	$0 \quad$ à $\quad (2^n - 1) - 1$
$\langle 2n \rangle : t$	$(2^n - 1)^2$	$3n$	$(2^n - 1) \quad$ à $\quad (2^n - 1) + (2^n - 1)^2 - 1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\langle kn \rangle : t$	$(2^n - 1)^k$	$(k+1)n$	$\sum_{i=1}^{k-1} (2^n - 1)^i \quad$ à $\quad -1 + \sum_{i=1}^{k} (2^n - 1)^i$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

TAB. 7.2 – Structure des codes tabou alignés.

m	Code	m	Code	m	Code	m	Code
0	01 **00**	4	01 10 **00**	8	10 11 **00**	12	01 01 01 **00**
1	10 **00**	5	01 11 **00**	9	11 01 **00**	13	01 01 10 **00**
2	11 **00**	6	10 01 **00**	10	11 10 **00**	14	01 01 11 **00**
3	01 01 **00**	7	10 10 **00**	11	11 11 **00**	$\vdots$	

TAB. 7.3 – Un code tabou aligné avec $n = 2$. Le tabou, c'est-à-dire les zéros terminant le code, sont en gras pour les mettre en évidence.

strictement de deux chiffres. Or, nous oublions que nous déterminons le début du nombre par sa position ou par le fait que nous passons de symboles typographiques quelconques aux chiffres, et nous savons que le nombre se termine où, inversement, nous retournons aux symboles qui ne sont pas des chiffres : espace, virgule, texte, etc. Si nous écrivons une liste de nombres, nous les séparerons par des virgules. À ce moment, et dans ce contexte, cela revient à écrire les nombres en base 11, c'est-à-dire les 10 chiffres normaux plus le symbole séparateur. Si on reprend les calculs de l'universalité des codes tabou alignés mais en posant $n = \lg 11$, nous trouvons que notre façon d'écrire les nombres est $\frac{\lg 11}{\lg 10}$-universelle, avec une constante d'universalité de ≈ 1.041.

7.4.2 Codes tabou non contraints

Considérons maintenant le cas où les codes ne sont plus contraints à être formés d'une série de blocs de n bits. Ces codes non contraints pourront être de longueur n, $n+1$, $n+2$,... Ces codes ont une longueur d'au moins n, car, comme les codes tabou alignés, les codes non contraints se terminent par une séquence tabou de n bits. Ici cependant, il faut choisir avec soin quelle sera la séquence utilisée comme tabou. Puisque nous ne pouvons plus recourir aux blocs pour nous assurer d'un décodage non ambigu, nous devons trouver une séquence qui limite le moins possible le choix des codes. Avec une séquence tabou formée par une séquence de zéros, il suffit d'un dernier bit à 1 avant le tabou pour le délimiter de façon non ambiguê. Eussions nous eu recours à une série quelconque de n bits, nous pourrions trouver plusieurs façon de faire correspondre un morceau du code avec un morceau du tabou, nécessitant alors un séparateur beaucoup plus long. Pour la concision, nous écrirons code tabou-n pour un code tabou dont le tabou est de longueur n.

7.4.2.1 Structure des codes

Un code tabou-n valide de longueur $l+n$ est une séquence de bits telle que les l premiers bits n'exhibent aucune série de n ou plus zéros consécutifs et se terminent par un 1, et les n derniers

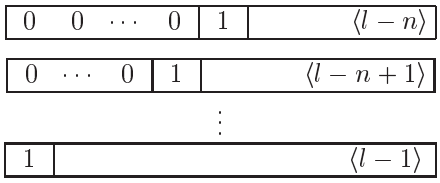


FIG. 7.6 – La structure générale du préfixe du code tabou non contraint de longueur l. Ici, les différentes régions, ou banques de codes, sont montrées.

bits sont des zéros. Les codes ont nécessairement un 1 avant le tabou, car, s'ils se terminaient avec un zéro ou plus, la fin du code se confondrait avec le tabou, rendant ainsi le décodage ambigu.

Le nombre de séquences valides de l bits, sans série de n zéros ou plus, se terminant en 1, est donné par

$$\left\langle\!\left\langle \begin{matrix} l \\ n \end{matrix} \right\rangle\!\right\rangle = \begin{cases} 1 & \text{si } \ l = 0 \\ 2^{l-1} & \text{si } \ l < n \\ \sum_{i=1}^{n} \left\langle\!\left\langle \begin{matrix} l-i \\ n \end{matrix} \right\rangle\!\right\rangle & \text{sinon} \end{cases} \tag{7.17}$$

On peut déduire la formule pour $\left\langle\!\left\langle \begin{matrix} l \\ n \end{matrix} \right\rangle\!\right\rangle$ de la façon suivante. Si $l = 0$, nous n'avons à considérer que la séquence vide qui est valide puisqu'on n'y trouve évidemment pas de séries de n zéros ou plus. Si $l > 0$ mais $l < n$, aucune séquence n'a de série de n zéros ou plus, mais comme elles se terminent toutes par 1, cela ne nous fait que 2^{l-1} séquences valides. Si $l \geqslant n$, nous avons une décomposition des 2^l séquences possibles telle que montrée à la fig. 7.6. Nous avons les séquences qui commencent avec un 1, dont $\left\langle\!\left\langle \begin{matrix} l-1 \\ n \end{matrix} \right\rangle\!\right\rangle$ valides. Puis nous avons les codes qui commencent par 01, laissant $\left\langle\!\left\langle \begin{matrix} l-2 \\ n \end{matrix} \right\rangle\!\right\rangle$ séquences valides. Puis viendront les séquences débutant par 001, puis 0001, etc, jusqu'à ce que viennent les séquences qui débutent par $n-1$ zéros suivis d'un 1, et parmi les 2^{l-n} séquences possibles, nous en avons $\left\langle\!\left\langle \begin{matrix} l-n \\ n \end{matrix} \right\rangle\!\right\rangle$ qui sont valides. Les séquences qui commencent par n zéros ou plus sont nécessairement invalides. Nous trouvons que, si $l \geqslant n$, le nombre de séquences n'exhibant aucune série de n zéros ou plus est donné par $\left\langle\!\left\langle \begin{matrix} l \\ n \end{matrix} \right\rangle\!\right\rangle = \sum_{i=1}^{n} \left\langle\!\left\langle \begin{matrix} l-i \\ n \end{matrix} \right\rangle\!\right\rangle$.

Cette récurrence pourrait vous paraître familière. Le lecteur est invité à examiner le tableau 7.4. La colonne associée à $\left\langle\!\left\langle \begin{matrix} l \\ 2 \end{matrix} \right\rangle\!\right\rangle$ forme une série spéciale, que vous avez su reconnaître, la série des nombres de Fibonacci. Pour $\left\langle\!\left\langle \begin{matrix} l \\ 2 \end{matrix} \right\rangle\!\right\rangle$ nous avons la relation $\left\langle\!\left\langle \begin{matrix} l \\ 2 \end{matrix} \right\rangle\!\right\rangle = F_{l+1}$, où F_i est le i^{e} nombre de Fibonacci, et F_i est une récurrence décrite à la section 5.3.3.5. La colonne de $\left\langle\!\left\langle \begin{matrix} l \\ 3 \end{matrix} \right\rangle\!\right\rangle$ est composée des nombres de « tribonacci », soit $\left\langle\!\left\langle \begin{matrix} l \\ 3 \end{matrix} \right\rangle\!\right\rangle = F^{(3)}_{l+1}$, où $F^{(3)}_i = F^{(3)}_{i-1} + F^{(3)}_{i-2} + F^{(3)}_{i-3}$ avec conditions initiales $F^{(3)}_1 = 1$, $F^{(3)}_2 = 1$, et $F^{(3)}_3 = 2$. La colonne de $\left\langle\!\left\langle \begin{matrix} l \\ 4 \end{matrix} \right\rangle\!\right\rangle$ correspond aux nombres « tetranacci »[1], soit $\left\langle\!\left\langle \begin{matrix} l \\ 4 \end{matrix} \right\rangle\!\right\rangle = F^{(4)}_{l+1}$, où $F^{(4)}_i = F^{(4)}_{i-1} + F^{(4)}_{i-2} + F^{(4)}_{i-3} + F^{(4)}_{i-4}$ avec conditions initiales $F^{(4)}_1 = 1$, $F^{(4)}_2 = 1$, $F^{(4)}_3 = 2$, et $F^{(4)}_3 = 4$. Enfin, la n^{e} colonne correspond aux *n-step Fibonacci numbers* [220], qui sont définis par

$$F^{(n)}_l = \sum_{i=1}^{n} F^{(n)}_{l-i}$$

1 Bien qu'en fait « quadrinacci » eut été plus indiqué ; histoire de ne pas sauter du latin au grec.

$\left\langle\!\left\langle {n \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {1 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {2 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {3 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {4 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {5 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {6 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {7 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {8 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {9 \atop l} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {10 \atop l} \right\rangle\!\right\rangle$
$\left\langle\!\left\langle {0 \atop n} \right\rangle\!\right\rangle$	1	1	1	1	1	1	1	1	1	1
$\left\langle\!\left\langle {1 \atop n} \right\rangle\!\right\rangle$	1	1	1	1	1	1	1	1	1	1
$\left\langle\!\left\langle {2 \atop n} \right\rangle\!\right\rangle$	1	2	2	2	2	2	2	2	2	2
$\left\langle\!\left\langle {3 \atop n} \right\rangle\!\right\rangle$	1	3	4	4	4	4	4	4	4	4
$\left\langle\!\left\langle {4 \atop n} \right\rangle\!\right\rangle$	1	5	7	8	8	8	8	8	8	8
$\left\langle\!\left\langle {5 \atop n} \right\rangle\!\right\rangle$	1	8	13	15	16	16	16	16	16	16
$\left\langle\!\left\langle {6 \atop n} \right\rangle\!\right\rangle$	1	13	24	29	31	32	32	32	32	32
$\left\langle\!\left\langle {7 \atop n} \right\rangle\!\right\rangle$	1	21	44	56	61	63	64	64	64	64
$\left\langle\!\left\langle {8 \atop n} \right\rangle\!\right\rangle$	1	34	81	108	120	125	127	128	128	218
$\left\langle\!\left\langle {9 \atop n} \right\rangle\!\right\rangle$	1	55	149	208	236	248	253	255	256	256
$\left\langle\!\left\langle {10 \atop n} \right\rangle\!\right\rangle$	1	89	274	401	464	492	504	509	511	512

TAB. 7.4 – Les quelques premières valeurs de $\left\langle\!\left\langle {n \atop l} \right\rangle\!\right\rangle$.

avec les conditions initiales

$$\begin{aligned} F_1^{(n)} &= 1 \\ F_i^{(n)} &= 2^{i-2}, \qquad i = 2, \ldots, n \end{aligned}$$

Nous avons donc l'identité

$$\left\langle\!\left\langle {n \atop l} \right\rangle\!\right\rangle = F_{l+1}^{(n)} \tag{7.18}$$

qui nous sera des plus utiles pour démontrer l'universalité des codes tabou non contraints.

7.4.2.2 Génération des codes

Considérons maintenant la génération d'un code tabou-n pour un nombre $i \in \mathbb{Z}^*$. Le tableau 7.5 montre la structure générale d'un code tabou-n. Comme pour les codes alignés, nous allons d'abord chercher la longueur du code qui sera nécessaire pour encoder i. Nous devons trouver l tel que

$$h_n(l-1) \leqslant i < h_n(l)$$

où

$$h_n(l) = \sum\nolimits_{i=0}^{l} \left\langle\!\left\langle {n \atop i} \right\rangle\!\right\rangle \tag{7.19}$$

ce qui revient à résoudre

$$l_n(i) = \min\{\, l \mid i < h_n(l) \,\} \tag{7.20}$$

Codes	Nombre de valeurs	Longueur	Intervalle
t	$\left\langle\!\left\langle {0 \atop n} \right\rangle\!\right\rangle$	n	$0 \quad$ à $\quad \left\langle\!\left\langle {0 \atop n} \right\rangle\!\right\rangle - 1$
$\langle 1\rangle : t$	$\left\langle\!\left\langle {1 \atop n} \right\rangle\!\right\rangle$	$1+n$	$\left\langle\!\left\langle {0 \atop n} \right\rangle\!\right\rangle$ à $\left\langle\!\left\langle {0 \atop n} \right\rangle\!\right\rangle + \left\langle\!\left\langle {1 \atop n} \right\rangle\!\right\rangle - 1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\langle l\rangle : t$	$\left\langle\!\left\langle {l \atop n} \right\rangle\!\right\rangle$	$l+n$	$\sum_{i=0}^{l-1} \left\langle\!\left\langle {i \atop n} \right\rangle\!\right\rangle$ à $-1 + \sum_{i=0}^{l} \left\langle\!\left\langle {i \atop n} \right\rangle\!\right\rangle$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

TAB. 7.5 – La structure d'un code tabou-n. Ici aussi, t est le tabou, une séquence de n zéros.

avec un cas spécial pour $n = 2$:

$$h_2(l) = \sum_{i=0}^{l} \left\langle\!\left\langle {i \atop 2} \right\rangle\!\right\rangle = \sum_{i=1}^{l+1} F_i = F_{l+3} - 1$$

Alors qu'il était plutôt simple de générer le code pour i avec les codes alignés, nous aurons besoin d'une procédure considérablement plus compliquée pour les codes non contraints. Soit encore $c = i - h_n(l_n(i) - 1)$. Certainement, $0 \leqslant c < \left\langle\!\left\langle {l_n(i) \atop n} \right\rangle\!\right\rangle$. Le code, sans le tabou, sera alors la c^{e} séquence de longueur $l_n(i)$, finissant avec un 1 et n'ayant aucune sous-séquence de n zéros ou plus, en ordre lexicographique. Posons

$$b(l, n, k) = \sum_{i=1}^{k+1} \left\langle\!\left\langle {l-n+i \atop n} \right\rangle\!\right\rangle$$

le nombre de codes valides jusqu'au k^{e} préfixe inclusivement (voir la fig. 7.6). Le préfixe commençant avec $n-1$ zéros est le préfixe zéro et le dernier le $n-1^{\text{e}}$. Sans le tabou ni le 1 final, le code pour $0 \leqslant c < \left\langle\!\left\langle {l_n(i) \atop n} \right\rangle\!\right\rangle$ est donné par

$$C(l, n, c) = \begin{cases} C_{\beta(l)}(c) & \text{si } \ l < n \\ C_{\beta(l)}(c+1) & \text{si } \ l = n \\ 0^k 1 : C(l, n, c - b(l, n, k-1)) & \text{si } \ b(l, n, k-1) \leqslant c < b(l, n, k) \end{cases}$$

où $C_{\beta(l)}(x)$ est le code binaire de x sur l bits et où 0^n est une séquence de n zéros. Le code complet pour i est alors donné par $C(l, n, c) : 1 : 0^n$.

Par exemple, le code tabou-3 est montré au tableau 7.6. On pourrait voir cette méthode de génération de code comme une généralisation de la recherche binaire récursive, où l'espace est divisé récursivement en portions fibonaccéennes plutôt que simplement égales.

7.4.2.3 Universalité des codes tabou non contraints

Montrons maintenant que les codes tabou non-contraints sont universels. Soit $L_n(i)$, la fonction de longueur du code pour i étant donné un tabou de $n \geqslant 2$. Nous montrerons que nous avons

$$1 < \lim_{i \to \infty} \frac{L_n(i)}{\lg i} \leqslant \frac{1}{\lg \phi}$$

La preuve pour $n = 2$ repose sur la décompositon duale de Zeckendorf. Pour $n \geqslant 2$, nous utiliserons une formule approchée pour $h_n(l)$, ce qui permettra de calculer une approximation

0	**000**	7	111**000**	14	1111**000**	21	10011**000**
1	1**000**	8	0011**000**	15	00101**000**	22	10101**000**
2	01**000**	9	0101**000**	16	00111**000**	23	10111**000**
3	11**000**	10	0111**000**	17	01001**000**	24	11001**000**
4	001**000**	11	1001**000**	18	01011**000**	25	11011**000**
5	011**000**	12	1011**000**	19	01101**000**	26	11101**000**
6	101**000**	13	1101**000**	20	01111**000**	27	⋮

TAB. 7.6 – Code tabou-3. Ici aussi, la séquence tabou est montrée en gras.

i	Fibonacci	i	Tabou	i	Fibonacci	i	Tabou
1	11	0	00	11	001011	10	110100
2	011	1	100	12	101011	11	111100
3	0011	2	0100	13	0000011	12	0101100
4	1011	3	1100	14	1000011	13	0110100
5	00011	4	01100	15	0100011	14	0111100
6	10011	5	10100	16	0010011	15	1010100
7	01011	6	11100	17	1010011	16	1011100
8	000011	7	010100	18	0001011	17	1101100
9	100011	8	011100	19	1001011	18	1110100
10	010011	9	101100	20	0101011	19	1111100

TAB. 7.7 – Parallèle entre les codes de Fibonacci et les codes tabou-2.

de $l_n(i)$, donc de $L_n(i)$.

Commençons par le cas $n = 2$. Les codes de Fibonacci (voir section 5.3.3.5) sont obtenus à partir de la décomposition duale de Zeckendorf, et nous avons déduit, à l'éq. (5.5), une formule exacte pour la longueur du code de Fibonacci pour un entier i :

$$|C_{fib}(i)| = L_{fib}(i) = \text{Fibolog}(i) + 1 = \left\lfloor \frac{\lg(i - \frac{1}{2}) + \lg\sqrt{5}}{\lg\phi} \right\rfloor + 1 \tag{7.21}$$

Considérez le tableau 7.7. Mis à part le fait que les codes de Fibonacci ne peuvent représenter le zéro (ce que l'on peut circonvenir trivialement en encodant $i + 1$ si $i \in \mathbb{Z}^*$ plutôt que $i \in \mathbb{N}$) alors que les codes tabou en sont capables, nous voyons que la longueur du code de Fibonacci pour i est la même que pour le code tabou-2 de $i - 1$, et ce, pour tous les codes. Il peut être démontré (à travers une démonstration du théorème dual de Zeckendorf [211, 98]) que les codes de Fibonacci sont en fait des codes tabou-2 où le tabou est 11 plutôt que 00. De plus, comme les codes tabou sont générés en ordre lexicographique, le code de Fibonacci pour i n'est pas simplement le complément binaire du code tabou-2 correspondant. Cette relation nous donne directement la fonction de longueur pour les codes tabou-2, c'est à dire que $L_2(i) = L_{fib}(i + 1)$.

Par l'éq. (7.21), on a

$$\begin{aligned}\lim_{i\to\infty}\frac{L_{fib}(i)}{\lg i} &= \lim_{i\to\infty}\frac{\left\lfloor\frac{\lg\sqrt{5}+\lg(i-\frac{1}{2})}{\lg\phi}\right\rfloor+1}{\lg i}\\ &= \lim_{i\to\infty}\frac{\lg\sqrt{5}+\lg(i-\frac{1}{2})}{\lg\phi\ \lg i}+\frac{1}{\lg i}\\ &= \lim_{i\to\infty}\frac{\lg\sqrt{5}}{\lg\phi\ \lg i}+\frac{\lg(i-\frac{1}{2})}{\lg\phi\ \lg i}+\frac{1}{\lg i}\\ &= \frac{1}{\lg\phi}\\ &\approx 1.44042009\ldots\end{aligned}$$

un résultat montré à la section 5.3.3.5. Pour $i\in\mathbb{N}$,

$$\lim_{i\to\infty}\frac{L_2(i)}{\lg i}=\lim_{i\to\infty}\frac{L_{fib}(i+1)}{\lg(i+1)}=\frac{1}{\lg\phi}$$

ce qui conclut la première partie de la démonstration, à savoir que les codes tabou-2 sont universels en étant $\frac{1}{\lg\phi}$-universels.

Pour $n>2$, nous aurons à déterminer $l_n(i)$ qui dépend sur $h_n(\cdot)$, l'éq.(7.19). Nous utiliserons l'approximation

$$F_j^{(n)}\approx c_{(n)}^j$$

où

$$c_{(n)}=\lim_{j\to\infty}\frac{F_{j+1}^{(n)}}{F_j^{(n)}}$$

et $c_{(n)}$ est aussi la plus petite solution réelle plus grande que 1 à l'équation polynomiale

$$x^n(2-x)=1 \tag{7.22}$$

décrite dans [220, p. 629]. On peut alors poser

$$h_n(l)=\sum_{i=0}^{l}\left\langle\!\left\langle {l \atop n}\right\rangle\!\right\rangle=\sum_{i=1}^{l+1}F_i^{(n)}\approx\sum_{i=1}^{l+1}c_{(n)}^i=\frac{(c_{(n)}^{l+1}-1)c_{(n)}}{c_{(n)}-1}$$

Nous sommes maintenant prêts à résoudre (approximativement) l'éq. (7.20). Nous cherchons le plus petit l tel que $i<h_n(l)$. Nous avons

$$i<h_n(l)\approx\frac{(c_{(n)}^{l+1}-1)c_{(n)}}{c_{(n)}-1}$$

que nous résolvons pour l, trouvant

$$l\gtrapprox\frac{\lg\left(i-\frac{i}{c_{(n)}}+1\right)}{\lg c_{(n)}}-1$$

ce qui donne

$$L_n(i) = l_n(i) + n \approx \frac{\lg\left(i - \frac{i}{c_{(n)}} + 1\right)}{\lg c_{(n)}} + n - 1$$

Finalement,

$$\begin{aligned}\lim_{i\to\infty} \frac{L_n(i)}{\lg i} = \lim_{i\to\infty} \frac{l_n(i)+n}{\lg i} &\approx \lim_{i\to\infty} \frac{\lg\left(i - \frac{i}{c_{(n)}} + 1\right) + n - 1}{\lg i \ \lg c_{(n)}} \\ &= \lim_{i\to\infty} \frac{\lg\left(i - \frac{i}{c_{(n)}} + 1\right)}{\lg i \ \lg c_{(n)}} + \frac{n-1}{\lg i \ \lg c_{(n)}} \\ &= \frac{1}{\lg c_{(n)}}\end{aligned}$$

ce qui termine la seconde et dernière partie de la preuve d'universalité. ■

Quant aux valeurs des constantes $c_{(n)}$ obtenues à partir de l'équation (7.22), on trouve, pour $n = 2$, $c_{(2)} = \phi$. Pour $n = 3$, nous trouvons

$$c_{(3)} = \sqrt[3]{\frac{19}{27} + \frac{1}{9}\sqrt{33}} + \frac{4}{9\sqrt[3]{\frac{19}{27} + \frac{1}{9}\sqrt{33}}} + \frac{1}{3} \approx 1.8392867552141611326\ldots$$

un résultat dû à Simon Plouffe[2]. Un code tabou-3 est donc 1.14-universel. On trouve aussi

$$\begin{aligned} c_{(4)} &\approx 1.9275619754829253043\ldots \\ c_{(5)} &\approx 1.9659482366454853372\ldots \\ c_{(6)} &\approx 1.9835828434243263304\ldots \\ c_{(7)} &\approx 1.9919641966050350210\ldots \\ c_{(8)} &\approx 1.9960311797354145898\ldots \end{aligned}$$

donnant, respectivement, des constantes d'universalité de ≈ 1.05621, ≈ 1.0254, ≈ 1.01203, ≈ 1.005284, et ≈ 1.00287. On voit que n n'a pas à être bien grand pour obtenir une bonne constante d'universalité, donc un code asymptotiquement très efficace. De grands n donneront des codes asymptotiquement plus efficaces, au prix de codes plus long pour les petits entiers ; réciproquement, de petits n donnent des codes plus courts aux petits entiers mais au coût de codes asymptotiquement moins efficaces.

La preuve d'universalité pose $n \geqslant 2$. Qu'advient-il lorsque $n = 1$? Le cas $n = 1$ correspond à toutes les séquences ne contenant aucun zéro sauf pour le tabou, c'est-à-dire une série de uns terminée par un unique zéro, ce qui nous donne encore le codage unaire, qui n'est pas universel. En effet, nous trouvons $c_{(1)} = 1$ et $\frac{1}{\lg 1} = \infty$, ce qui est cohérent avec la preuve et la formulation du problème.

2 Ce résultat peut être trouvé dans une correspondance non publiée, mais que l'on peut trouver sur le web, sur la page de l'auteur, `www.lacim.uqam.ca/plouffe`.

7.4.2.4 Calcul efficace de $F_i^{(n)}$ et $h_n(i)$

L'implémentation récursive naïve de F_i mène à $O(F_i)$ appels, ce qui est ridicule pour i moindrement grand. Une approche de programmation dynamique nous permettra de calculer F_i en un temps et espace de $O(i)$. C'est déjà mieux que la version naïve, mais encore prohibitif pour un grand i, surtout lorsqu'on considère la mémoire nécessaire. Si, plutôt que d'utiliser la méthode qui part de F_i et qui descend jusqu'à F_1, nous utilisons une méthode qui commence par calculer F_1 et qui remonte jusqu'à F_i, nous pouvons calculer F_i en temps linéaire en i et en n'utilisant qu'une quantité constante de mémoire : il suffit de trois registres ayant $O(\lg F_i)$ bits. Cependant, nous pouvons calculer $F_i^{(n)}$ en un temps $O(n^3 \lg i)$ et une mémoire de $O(n^2)$, en comptant les opérations arithmétique sur les grands entiers comme unitaires et que $F_i^{(n)}$ tienne dans un registre de la machine. Considérez la relation, pour $k = 0, 1, 2, \ldots$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix}$$

On supposera que la puissance 0 de la matrice est la matrice identité. Bien que cela ne semble pas, à prime abord, être une amélioration, cette relation permet de calculer le k^{e} nombre de Fibonacci en $O(\lg k)$ étapes. L'exponentiation se calcule en temps logarithmique en prenant avantage d'une factorisation particulière de l'exponentiation. Par exemple, x^k avec $k = 25$ se calcule aussi par $((((x^2 x)^2)^2)^2)x$, ce qui ne compte plus que 6 multiplications alors qu'une méthode itérative pour le calcul de l'exponentiation en demanderait 25. Comme le produit de matrice est associatif, on peut utiliser cette méthode pour calculer l'exponentiation de la matrice en $O(\lg k)$ produits matriciels. Comme le coût d'une multiplication de deux matrices $n \times n$ est au plus $O(n^3)$, on obtient une complexité totale de $O(n^3 \lg k)$ [3]. Pour les nombres de Fibonacci, $n = 2$, et on pourrait considérer le terme n^3 comme étant constant.

On peut généraliser cette tactique pour calculer $F_k^{(n)}$ pour un n quelconque. Considérez maintenant

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}^k \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{k+3}^{(3)} \\ F_{k+2}^{(3)} \\ F_{k+1}^{(3)} \end{pmatrix}$$

La première rangée de la matrice calcule le prochain terme dans la série, $F_{k+2}^{(3)}$ et les deux autres rangées décalent les termes précédents de la première et seconde à la seconde et troisième positions dans le vecteur résultat. Ici encore, la relation tient parce que le produit matriciel est associatif. Le vecteur (2 1 1) encode les conditions initiales pour la série $F^{(3)}$, mais remplacer ces valeurs par d'autres valeurs permettrait de calculer une généralisation des nombres tribonacci.

Le principe peut être étendu à tout $n \in \mathbb{N}$. La première rangée de la matrice $n \times n$ demeure remplie de uns, pour calculer $\sum_{i=1}^{n} F_{k-i}^{(n)}$, et les $n-1$ autres rangées demeurent les rangées de décalage ; la i^{e} rangée de la matrice est la $i-1^{\text{e}}$ rangée de la matrice identité $n \times n$. Ici encore nous obtenons un algorithme en $O(n^3 \lg k)$ nécessitant $O(n^2)$ mémoire. Bien que le terme en n^3 puisse paraître dangereux, le lecteur doit considérer le fait que la constante d'universalité des codes tabou converge rapidement vers 1 lorsque n croît, limitant ainsi la nécessité d'un grand

3 Si n est suffisament grand, on pourra tirer profit d'un algorithme de multiplication rapide comme l'algorithme de Strassen, qui multiplie deux matrices $n \times n$ en $O(n^{\lg 7})$.

n. De plus, n^3 devient asymptotiquement négligeable par rapport à $\lg k$ — et encore plus par rapport à k si on devait utiliser une méthode en temps linéaire pour calculer $F_k^{(n)}$.

Est-ce que $h_n(l)$ peut être calculée efficacement ? Pour $n = 2$, nous avons déjà l'identité $h_2(l) = F_{l+3} - 1$, une identitée connue sur la somme des $l+1$ premiers nombres de la séquence de Fibonacci. Qu'en est-il pour $n > 2$? Bien qu'il soit possible de dériver des identitées particulières à chaque n — peut-être existe-t-il une identité générale pour tout n, si elle existe, elle est inconnue — nous présentons ici une méthode pour calculer $h_n(l)$ en $O(n^3 \lg l)$ étapes et $O(n^2)$ mémoire. La méthode est similaire à la méthode présentée pour calculer $F_k^{(n)}$. Pour $n = 3$, nous avons la relation

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 \\ 2 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} h_3(k+1) \\ F_{k+3}^{(3)} \\ F_{k+2}^{(3)} \\ F_{k+1}^{(3)} \end{pmatrix}$$

Cette relation se généralise pour tout $n \geqslant 2$. La première rangée calcule la somme de $h_n(k-1)$ et de $F_k^{(n)}$ (grâce à un 1 à la position $n-1$), donnant effectivement $h_n(k) = h_n(k-1) + F_k^{(n)}$. Les autres rangées de la matrice calculent $F_{k+1}^{(n)}$ à $F_{k+n}^{(n)}$ en utilisant la même stratégie que précédemment, avec les rangées de sommation et de décalage. Ici aussi, l'astuce d'exponentiation rapide nous donne un algorithme en $O((n+1)^3 \lg k)$ utilisant une quantité de mémoire $O((n+1)^2)$.

Utiliser un algorithme pour calculer $h_n(l)$ plutôt qu'une forme analytique peut apparaître comme un manque de finesse, mais il ne faut pas oublier que des formes reposant sur une identité impliquant $F_l^{(n)}$ (comme le cas spécial $h_2(l) = F_{l+3} - 1$) nécessiteraient tout de même $O(n^3 \lg l)$ calculs. De plus, il est douteux que nous puissions utiliser des identités qui reposent sur l'utilisation des $c_{(n)}$. Calculer les nombres de Fibonacci en utilisant ϕ et $\sqrt{5}$ précis à 16 décimales (ce qui correspond à la précision des nombres flottants doubles au standard IEEE-754, qui est souvent le maximum supporté par le processeur) ne nous permet pas de dépasser F_{75} sans avoir des erreurs d'arrondi. Les termes des séries $F^{(n)}$ croissent encore plus rapidement, et la précision double est plus rapidement dépassée.

7.4.2.5 Le choix de la longueur du tabou

Si avoir un tabou assez long nous permet de s'approcher de $O(\lg i)$ bits pour coder un très grand entier i, cela nous donne aussi des codes comparativement longs pour les petits entiers. Si on s'attend à trouver plutôt de petits entiers, il est peut-être préférable de choisir un tabou court, par exemple, de longueur deux ou trois, afin d'avoir des codes assez courts en moyenne. Si, au contraire, on s'attend que la distribution soit biaisée vers de grands entiers, peut-être serait-il mieux d'avoir un tabou plus long. Nous avons cependant montré dans une section précédante que le tabou n'a pas à être bien long pour nous donner une bonne constante d'universalité. Déjà, avec une longueur de tabou de 4, on a un code qui est ≈ 1.056-universel.

Les hypothèses d'universalité sont plutôt vagues. Les contraintes imposées sur la distribution de laquelle sont tirés les entiers ne spécifient pas complètement la classe de distribution. Beaucoup de distributions différentes satisfont $P(x) \leqslant P(y)$ si $x \leqslant y$ et $P(x) \neq 0$ pour $x \in \mathbb{N}$ (ou $x \in \mathbb{Z}^*$). Cette description convient aussi bien à la distribution géométrique qu'à la distribution exponentielle. Pour estimer la longueur du tabou qu'il nous faut, il s'agirait en fait d'estimer la

distribution des x et de calculer la longueur moyenne des codes (grâce aux fonctions de longueur de codes que nous avons déjà données) selon les différents tabous et choisir celui qui donne la meilleure longueur moyenne.

7.5 Codes $(Start, Step, Stop)$ et $Start/Stop$

Dans cette section, nous présentons nos contributions sur les codes $(Start, Step, Stop)$, qui furent introduit par Fiala et Greene qui ne présentèrent par d'algorithme d'optimisation [70]. Nous présenterons par la suite une extension aux $(Start, Step, Stop)$, les codes $Start/Stop$ que nous avons présentés à la conférence DCC 2001 en résumé étendu [161]. Les codes $Start/Stop$ généralisent les codes $(Start, Step, Stop)$ en modifiant les contraintes sur les longueurs des codes afin d'offrir plus de souplesse dans la forme du code.

Nous présentons des algorithmes d'optimisation pour l'un et l'autre code et comparerons les résultats à la section 7.5.4. Nous discuterons aussi des modifications qu'il est nécessaire d'apporter aux algorithmes d'optimisation pour obtenir des codes qui seront idéaux pour les implémentations efficaces du codeur et du décodeur. Enfin, nous verrons comment on peut transformer ces codes en codes universels efficaces.

7.5.1 Codes $(Start, Step, Stop)$

Nous avons présenté les codes $(Start, Step, Stop)$ à la section 5.3.5.2. Ces codes sont dûs à Fiala et Greene [70]. Il s'agit de coder des nombres de 0 à $N-1$, distribués selon une loi non croissante. Ces codes sont définis par trois paramètres, $Start$, $Step$ et $Stop$ qui contrôlent la forme et les longueurs des codes. Le code pour un entier i est composé du code unaire tronqué de $k(i)$ (nous reviendrons au calcul de $k(i)$ dans un instant) suivis de $Start + k(i)Step$ bits, jusqu'à une concurrence de $Stop$ bits. Puisque $Stop$ est contraint d'être de la forme $Start + k_{max}Step$, il nous a paru pertinent de changer la paramétrisation du code à $(Start, Step, k_{max})$.

Le préfixe du code est formé du code unaire tronqué du paramètre k étant donné k_{max}, soit $C_{\hat{\alpha}(k_{max})}(\cdot)$, dont nous rappelons la définition :

$$C_{\hat{\alpha}(M)}(i) = \begin{cases} C_\alpha(i) & \text{si } i < M \\ C_{\hat{\alpha}}(i) & \text{si } i = M \end{cases}$$

où $C_\alpha(\cdot)$ est le code unaire normal, et $C_{\hat{\alpha}}(\cdot)$ est le code unaire dont on omet le dernier bit. La longueur moyenne du code, étant donnée une source aléatoire X, est

$$\begin{aligned} \bar{L}_{(Start,Step,k_{max})}(X) &= \sum_{k=0}^{k_{max}} P(l(k) \leqslant X < h(k)) L_{(Start,Step,k_{max})}(x \mid k) \\ &= \sum_{k=0}^{k_{max}} P(l(k) \leqslant X < h(k)) \left(\min(k_{max}, k+1) + Start + k\,Step\right) \\ &= Start + Step \sum_{k=0}^{k_{max}} k\,P(l(k) \leqslant X < h(k)) \\ &\qquad + \sum_{k=0}^{k_{max}} \min(k_{max}, k+1) P(l(k) \leqslant X < h(k)) \\ &= Start + Step\,E[k] + E[k+1] - P(l(k_{max}) \leqslant X < h(k_{max})) \\ &= Start + 1 + (Step+1)E[k] - P(l(k_{max}) \leqslant X < h(k_{max})) \end{aligned} \tag{7.23}$$

où $l(0) = 0$ et

$$l(k+1) = h(k) = \frac{2^{Start+(k+1)Step} - 2^{Start}}{2^{Step} - 1}$$

que l'on a trouvé à partir d'une série géométrique (voir la section 5.3.5.2 pour la démonstration). Nous disions donc que le code pour i est tel que

$$C_{(Start,Step,k_{max})}(i) = C_{\hat{\alpha}(k_{max})}(k(i)) : C_{\beta(Start+k(i)Step)}(i - l(k(i)))$$

où $k(i)$ est calculé de façon à ce que $l(k(i)) \leqslant i$. Nous obtenons $k(i)$ de la façon suivante :

$$\begin{aligned}
\frac{2^{Start+(k+1)Step} - 2^{Start}}{2^{Step} - 1} &\leqslant i \\
2^{Start+(k+1)Step} - 2^{Start} &\leqslant i(2^{Step} - 1) \\
2^{Start+(k+1)Step} &\leqslant i(2^{Step} - 1) - 2^{Start} \\
Start + (k+1)Step &\leqslant \lg\left(i(2^{Step} - 1) - 2^{Start}\right) \\
(k+1)Step &\leqslant \lg\left(i(2^{Step} - 1) - 2^{Start}\right) - Start \\
k + 1 &\leqslant \frac{\lg\left(i(2^{Step} - 1) - 2^{Start}\right) - Start}{Step} \\
k &\leqslant \frac{\lg\left(i(2^{Step} - 1) - 2^{Start}\right) - Start}{Step} - 1
\end{aligned}$$

c'est une manipulation que nous avons déjà vue quelques fois. Enfin, nous calculons le plancher pour respecter $k \in \mathbb{Z}^*$ et l'inégalité, et nous obtenons

$$k(i) = \left\lfloor \frac{\lg\left(i(2^{Step} - 1) - 2^{Start}\right) - Start}{Step} - 1 \right\rfloor$$

L'équation éq. (7.23) suggère que nous en trouvions le minimum grâce aux dérivées

$$\frac{\partial \bar{L}_{(Start,Step,k_{max})}(X)}{\partial\, Start} = 0 \qquad \text{et} \qquad \frac{\partial \bar{L}_{(Start,Step,k_{max})}(X)}{\partial\, Step} = 0$$

mais il ne suffit pas que la fonction de distribution soit non croissante et que la fonction de longueur de code soit non décroissante pour obtenir une fonction concave. La situation est légèrement compliquée par le fait que k_{max} dépende aussi des variables $Start$ et $Stop$. En pratique, il est possible que l'on ne puisse pas allouer exactement N codes, c'est-à-dire satisfaire la contrainte $h(k_{max}) = N$ (où $N-1$ est le plus grand nombre à représenter). Nous allons devoir la statisfaire de façon relâchée, c'est-à-dire en ayant $h(k_{max}) = N + \varepsilon$ pour un $\varepsilon \in \mathbb{Z}^*$ et que l'on veut garder relativement petit.

En fait, on pense d'abord qu'en utilisant l'hypothèse que $P(l(k) \leqslant X < h(k))$ décroît très rapidement en k, on peut laisser ε grandir sans affecter significativement la longueur moyenne du code, et qu'au pire, nous obtenons un code qui pourra représenter quelques nombres de plus. En fait, dû aux contraintes qui exigent que $Start$ et $Step$ soient des entiers non négatifs et surtout à la structure même du code, il est presque certain que les valeurs optimales pour $Step$ et $Start$ qui satisfont $h(k_{max}) \geqslant N$ laisseront des codes libres. Nous verrons à la section des résultats

que dans les fait, le nombre de codes inutilisés est généralement très important.

Si on connaît une expression analytique pour $P(X = x)$, où $x = \{0, 1, \ldots, N-1\}$, on peut calculer exactement l'éq. (7.23) et ses dérivées partielles par rapport à $Start$ et $Step$ si elle est concave et résoudre directement. Or, il est fort à parier que nous n'ayons pas d'expression analytique pour $P(X = x)$, mais seulement un histogramme des fréquences. Nous présenterons, à la section 7.5.3, une méthode d'optimisation pour les codes $(Start, Step, k_{max})$ qui n'utilise que l'information sur l'histogramme pour calculer les valeurs optimales.

La structure du code nous indique que $P(X = x)$ devrait être une fonction de distribution non croissante de façon à minimiser le nombre de bits moyen par code. Quelle serait la fonction de distribution idéale pour cette classe de code ? Nous pouvons la trouver en supposant que la longueur moyenne du code telle qu'énoncée à l'éq. (7.23) est l'entropie de la loi aléatoire inconnue X. Dès lors, on trouve que

$$-\lg P(X = x) = Start + (k(x) + 1)Step + \min(k_{max}, k(x) + 1)$$

et, utilisant la décomposition de Bayes, nous trouvons

$$\begin{aligned} -\lg P(X = x) &= -\lg \left(P\left(X = x \mid l(k(x)) \leqslant X < h(k(x))\right) P\left(l(k(x)) \leqslant X < h(k(x))\right)\right) \\ &= Start + (k(x) + 1)Step + \min(k_{max}, k(x) + 1) \end{aligned}$$

puis

$$\begin{aligned} P(X = x) &= 2^{-(Start + (k(x)+1)Step + \min(k_{max}, k(x)+1))} \\ &= 2^{-(Start + (k(x)+1)Step)}\ 2^{-\min(k_{max}, k(x)+1)} \end{aligned}$$

La meilleure fonction de distribution pour les codes $(Start, Step, k_{max})$ est une distribution uniforme par partie, la taille de chaque partie étant distribuée exponentiellement.

7.5.2 Codes *Start/Stop*

Les codes $(Start, Step, k_{max})$ sont plutôt rigides. À chaque fois que la longueur du code augmente, c'est de $Step$ bits, créant ainsi des intervalles beaucoup plus grands que les précédents. C'est aussi difficile de trouver les paramètres optimaux qui ne créent pas un très grand nombre de nouvelles valeurs représentables. Ces nouvelles valeurs, introduites pour satisfaire la contrainte $h(k_{max}) \geqslant N$, représentent un certain nombre de bits perdus puisque que nous n'avons que N valeurs distinctes à coder.

Comment améliorer les codes $(Start, Step, k_{max})$ de façon à leur donner plus de souplesse ? On peut modifier la règle d'élongation du code en permettant d'avoir un allongement différent et quelconque à chaque fois plutôt que d'être exactement de $Step$ bits. Autrement dit, plutôt que de restreindre les codes à avoir un suffixe de longueur $Start + k\ Step$, on pourrait avoir un code dont les suffixes sont de longueur $m_0 + m_1 + \cdots + m_k$, où tous les $m_j \in \mathbb{Z}^*$. Cela nous permettrait d'optimiser beaucoup plus finement le code pour la fonction de distribution non croissante considérée. Cette généralisation des codes $(Start, Step, Stop)$ est appelée codes *Start/Stop*, et est l'une de nos contributions originales.

Les différences de longueurs qui caractérisent le code *Start/Stop* sont les paramètres du code. Soit $\{m_0, m_1, \ldots, m_{k_{max}}\}$, les paramètres du code *Start/Stop*. Ces paramètres décrivent

complètement le code. Quant à la structure du code, elle est semblable à la structure des codes $(Start, Step, Stop)$. Le code pour $i \in \mathbb{Z}^*$ est composé d'un préfixe qui est le code unaire tronqué de $k'(i)$, suivi de de $\sum_{j=0}^{k'(i)} m_j$ bits. Comme pour les codes $(Start, Step, Stop)$, nous aurons $l'(\cdot)$ et $h'(\cdot)$ définis par la relation

$$l'(0) = 0$$
$$l'(k+1) = h'(k) = \sum_{i=0}^{k} 2^{\sum_{j=0}^{i} m_j}$$

et pour un entier i, nous devrons calculer $k'(i) = \min\{k \mid i < h'(k)\}$. Nous encoderons $k'(i)$ avec le code unaire tronqué à k_{max} suivi de $\sum_{j=0}^{k'(i)} m_j$ bits qui encodent $i - l(k(i))$. Le code pour i est donné par

$$C_{\{m_0,\ldots,m_{k'_{max}}\}}(i) = C_{\dot{\alpha}(k'_{max})}(k'(i)) : C_{\beta(\sum_{j=0}^{k'(i)} m_j)}(i - l'(k'(i)))$$

La longueur moyenne du code, similairement aux codes $(Start, Step, Stop)$, est donnée par

$$\bar{L}_{\{m_0,\ldots,m_{k'_{max}}\}}(X) = \sum_{k=0}^{k'_{max}} P(l'(k) \leqslant X < h'(k)) \left(\min(k+1, k'_{max}) + \sum_{i=0}^{k} m_i \right) \quad (7.24)$$

La forme générale du code est semblable à celle des codes $(Start, Step, Stop)$. Par exemple, le code $\{3, 1, 2, 0\}$ aurait la structure suivante :

Code	Nombres représentés
0 | x_2 x_1 x_0	$0 - 7$
1 0 | x_3 x_2 x_1 x_0	$8 - 23$
1 1 0 | x_5 x_4 x_3 x_2 x_1 x_0	$24 - 55$
1 1 1 | x_5 x_4 x_3 x_2 x_1 x_0	$56 - 87$

Le fait que les m_j soient quelconques (bien que non-négatifs) ne nous permettra pas de découler une expression simple pour $k'(i)$ comme pour les codes $(Start, Step, Stop)$. On peut se consoler en pensant que l'exponentiation se fait à faible coût en arithmétique entière puisqu'on élève 2 à une (petite) puissance et nous disposons d'instructions efficaces pour la calculer. De plus, si le nombre de paramètres est petit, nous pouvons facilement avoir recours à une table que l'on fouille (grâce à la recherche dichotomique, par exemple) pour trouver efficacement $k'(i)$. Pour ce, il suffit de stocker $k'_{max} + 1$ nombres (n'oublions pas que la série commence à zéro), ce qui peut ne pas être négligeable car nous pouvons aussi obtenir un très grand nombre de paramètres — jusqu'à N s'il s'avert qu'ils sont tous zéro, auquel cas nous nous retrouvons avec un encodage unaire tronqué — ce qui pourrait rendre l'utilisation d'une table impossible, ou au moins problématique. Cependant, nous verrons qu'il est plutôt difficile d'obtenir un très grand nombre de paramètres et qu'il est possible de contraindre l'optimisation de façon à n'obtenir qu'un nombre désiré de paramètres.

Nous verrons à la prochaine section le gain que cette nouvelle formulation offre par rapport aux codes $(Start, Step, Stop)$. Nous verrons aussi comment contraindre l'optimisation afin d'obtenir le plus petit nombre de paramètres possible ou comment obtenir des paramètres qui ont

une forme particulière, notamment en vue d'obtenir des implémentations rapides du codeur et du décodeur.

7.5.3 Optimisation des codes $(Start, Step, Stop)$ et $Start/Stop$

Présentons maintenant les algorithmes d'optimisation pour les codes $(Start, Step, Stop)$ et $Start/Stop$. L'optimisation des codes $(Start, Step, Stop)$ est simple et computationnellement peu exigeante. L'optimisation des codes $Start/Stop$ est plus compliquée mais on peut produire des algorithmes efficaces pour trouver les paramètres optimaux sous diverses contraintes. Nous présentons aussi une heuristique qui permet de calculer très efficacement une bonne approximation des paramètres optimaux.

7.5.3.1 Optimisation des codes $(Start, Step, Stop)$

Les codes $(Start, Step, Stop)$ sont faciles à optimiser pourvu que nous puissions calculer $P(l(k) \leqslant X < h(k))$ efficacement. Si nous disposons d'une formulation analytique pour $P(l(k) \leqslant X < h(k))$, nous n'avons besoin d'aucune astuce particulière. Si, par contre, nous n'avons accès qu'à l'histogramme de $P(X = x)$ pour chacune des N valeurs distinctes à coder, nous pouvons nous en tirer au coût de $O(N)$ cases mémoire et en un temps $O(N + k_{max}(\lg N)^3)$.

Définissons $F(x) = \sum_{i=0}^{x} P(x)$, la fonction de masse cumulative (FMC). Calculer la FMC pour tous les x et mettre les résultats dans un tableau coûte au plus $O(N)$, car, puisque chaque valeur pour x est donnée par $F(x-1) + P(x)$, il suffit de propager la somme grâce à cette relation sur tous les éléments du tableau qui maintient la FMC. Le calcul pour $P(a \leqslant X < b)$ devient $F(b-1) - F(a-1)$, ce qui ne requiert un temps constant pour être calculé. Maitenant, nous devons trouver les valeurs de $Start$ et $Step$ qui minimisent $\bar{L}_{Start,Step,k_{max}}(X)$ tout en satisfaisant $h(k_{max}(Start, Step)) \geqslant N$, où $k_{max}(Start, Step)$ dépend des solutions courantes pour $Start$ et $Step$.

Le nombre de valeurs à inspecter pour $Start$ et $Step$ n'est pas très grand. En effet, $Start$ comme $Step$ doit être inférieur ou égal à $\lceil \lg N \rceil$, ce qui fait qu'essayer toutes les combinaisons telles que $0 \leqslant Start \leqslant \lceil \lg N \rceil$ et $0 \leqslant Step \leqslant \lceil \lg N \rceil$ ne coûte au plus que $O((\lg N)^2)$ étapes ; la FMC étant déjà calculée, $P(a \leqslant X < b)$ se calcule en temps constant, de même que $k_{max}(Start, Step)$ et $\bar{L}_{Start,Step,k_{max}(Start,Step)}(X)$ requiert $k_{max}(Start, Step)$ étapes, ce qui est aussi au plus $O(\lg N)$. De plus, on peut élaguer les calculs lorsque $k_{max}(Start, Step) > \lceil \lg N \rceil$ ou lorsque $Start + Step > \lceil \lg N \rceil$. Le coût total de l'algorithme est au plus $O(N + (\lg N)^3)$, ce qui est $O(N)$.

Les résultats obtenus pour quelques distributions par cet algorithme d'optimisation qui calcule les paramètres optimaux sont comparés aux codes de Huffman et au codes $Start/Stop$ à la section 7.5.4.

7.5.3.2 Optimisation des codes $Start/Stop$

Les codes $Start/Stop$ sont considérablement plus compliqués à optimiser que les codes $(Start, Step, Stop)$. D'une part, nous ne connaissons pas *a priori* le nombre de paramètres optimaux. D'autre part, le principe d'optimalité ne s'applique pas, empêchant ainsi de résoudre optimalement d'abord pour m_0, puis de procéder pour les autres paramètres — c'est le même type de problème que nous avons rencontré à la section 6.2.1 avec le code de Shannon-Fano. On

sait que la somme des m_j est inférieure ou égale à $\lceil \lg N \rceil$, ce qui pourrait permettre d'éliminer certaines combinaisons. On peut aussi limiter délibérément le nombre de paramètres que l'on cherche.

7.5.3.2.1 Méthode vorace Cette méthode est la plus simple et la moins coûteuse computationnellement. Nous allons résoudre pour les paramètres $\{m_0, m_1, \ldots, m_k\}$ un à la fois, en commençant par le premier. Le paramètre m_0 est choisi de façon à ce que $P(0 \leqslant X < 2^{m_0}) \approx \frac{1}{2}$, en minimisant la différence à $\frac{1}{2}$. Ayant choisi m_0, on procédera au choix de m_1 de façon à ce que $P(2^{m_0} \leqslant X < 2^{m_0} + 2^{m_0+m_1}) \approx \frac{1}{2}$, et ainsi de suite jusqu'à ce que nous ayons $\sum_{i=0}^{k} 2^{\sum_{j=1}^{k} m_j} \geqslant N$, c'est-à-dire que tous les codes sont représentables, mais en minimisant le nombre de codes superflus introduits.

Le principe d'optimalité ne s'appliquant pas, cette résolution ne peut garantir l'optimalité de la solution. D'une part le choix vorace d'un paramètre basé sur la minimisation immédiate de la différence entre la probabilité obtenue et $\frac{1}{2}$ ne garantit pas de solution optimale car il se peut qu'à l'étape t on ait à choisir une solution localement sous-optimale pour trouver une meilleure solution globale aux étapes $t+1$ et plus. D'autre part, les tailles des partitions obtenues varient de façon exponentielle ; en contraignant les paramètres à être des entiers, on crée des partitions de longueur 2^m. L'algorithme a donc une sensibilité exponentielle à un changement unitaire dans un de ses paramètres.

La complexité de cet algorithme est essentiellement linéaire. Nous trouvons, si nous avons recours à la FMC, une complexité moyenne de $O(N + (\lg N)^2)$. Trouver le premier pivot revient à trouver x tel que $P(X \leqslant x) = \frac{1}{2}$, ce qui se fait en $O(\lg N)$ grâce à une recherche dichotomique sur le tableau contenant la FMC. En étant malchanceux, nous pourrions trouver $x = 0$ et cela nous laisse l'intervalle $1, 2, \ldots, N-1$ à fouiller pour trouver le second pivot, ce qui se fera aussi en $O(\lg N)$. En poursuivant le raisonnement, nous pourrions nous retrouver dans cette situation au plus $O(N)$ fois, ce qui donnera un pire cas de $O(N \lg N)$. Naturellement, cela demanderait une distribution $P(X = x) = 2^{-x}$. Si ce cas se présente, on prendrait soin de limiter le nombre de paramètres à $O(\lg N)$. Si la distribution est de la forme $P(X = x) = 1/N$, nous n'aurons besoin que de $O(\lg N)$ pivots, donnant un meilleur cas de $O((\lg N)^2)$. Quant au cas moyen, il faut considérer toutes les distributions non-croissantes possibles sur 0 à $N-1$; on finit par montrer que le premier pivot se trouve au centre en moyenne, ce qui permet d'espérer une complexité de $(\lg N)^2$, ce qui nous donne aussi notre complexité moyenne.

Nous présentons les résultats de l'algorithme vorace à la section 7.5.4.

7.5.3.2.2 Recherche contrainte Il est bien entendu inconcevable d'essayer toutes les combinaisons de paramètres qui satisfont la contrainte $\sum_{i=0}^{k'_{max}} 2^{\sum_{j=1}^{i} m_j} \geqslant N$. L'algorithme par recherche contrainte procédera à l'examen des solutions ayant au plus M paramètres, ce qui permet d'en limiter considérablement le nombre. En limitant à M le nombre de paramètres et en posant $n = \lceil \lg N \rceil$ et $\sum m_j \leqslant n$, nous trouvons que le nombre de combinaisons de paramètres

que nous avons besoin d'examiner est au plus

$$\begin{aligned}\sum_{i=1}^{M}\binom{i+n-1}{n} &= \sum_{i=1}^{m}\binom{(i-1)+n}{n}\\ &= \sum_{j=0}^{M-1}\binom{j+n}{n}\\ &= \binom{n}{n}+\sum_{j=1}^{M-1}\binom{j+n}{n}\\ &= 1+\binom{(M-1)+n+1}{M-1}-1\\ &= \binom{M+n}{M-1}\end{aligned} \tag{7.25}$$

résultat que l'on dérive en sachant que le nombre de solutions distinctes à l'égalité $\sum_{i=1}^{M} x_i = y$, avec $y, x_i \in \mathbb{Z}^*$, est donné par $\binom{M+y-1}{M}$, puis en utilisant des identités binomiales pour simplifier le tout [87].

La complexité de l'algorithme, si on a besoin de calculer la FMC, est $O(N + \binom{M+n}{M-1}M)$. De par l'éq. (7.25), on a $\binom{M+n}{M-1}$ combinaisons à inspecter, le calcul de la FMC coûte $O(N)$ opérations et l'évaluation la longueur moyenne du code grâce à l'éq. (7.24) coûte $O(M)$ opérations car maintenant $k'_{max} = M$.

La forme même de l'éq. (7.25) réveille en nous les pires craintes. Or, les valeurs obtenues de l'éq. (7.25) sont petites ou modérées pour des M et des N raisonnables. En utilisant l'approximation de Stirling pour la factorielle (voir l'appendice C), on trouve

$$\binom{M+n}{M-1} \approx \frac{1}{\sqrt{2\pi}}(M-1)^{\frac{1}{2}-M}(n+1)^{-\frac{3}{2}-n}(M+n)^{\frac{1}{2}+M+n}$$

dont les deux premiers termes, convergeant rapidement vers zéro, balancent le dernier terme qui croît de façon exponentielle. Par exemple, pour $n = 16$ et $M = 4$ on obtient 1140, pour $n = 32$ et $M = 4$ on obtient 7140, ce qui est sans problème computationnellement.

Comme l'algorithme explore exhaustivement tous les paramètres satisfaisant les contraintes, il est garanti de trouver la solution optimale pour un M donné. Les résultats sont présentés à la section 7.5.4.

7.5.3.3 Considérations de codage et décodage rapides des codes *Start/Stop*

Le but de l'utilisation d'un code *Start/Stop* est de limiter autant que faire se peut la complexité du codeur et du décodeur. Limiter le nombre de paramètres du code permettra d'avoir un petit nombre d'opérations à faire en pire cas pour encoder ou décoder le prochain entier, tout en minimisant le nombre de bits qu'il sera nécessaire de transmettre au décodeur pour décrire le code. Pour les codes *Start/Stop*, la description du code nécessitera $O(\lg k'_{max} + k'_{max} \lg N)$ bits.

Pour réduire à un minimum le nombre d'instructions nécessaires au codage comme au décodage, on pourra prendre soin d'organiser le code de façon à ce que tous les codes aient une longueur multiple d'un certain nombre de bits b qui correspond à une frontière naturelle de la

mémoire, comme un *nibble* ou un octet. Un code basé sur les octets ne requierrait aucune manipulation au niveau des bits individuels. Un code qui permet des longueurs de codes quelconques doit faire des manipulations au niveau des bits pour compenser pour l'alignement du code (qui peut être tout à fait quelconque) dans la séquence compressée, pour extraire et masquer les bits qui ne font pas partie du code, etc. Bien que les instructions qui manipulent les bits aient un coût unitaire sur la plupart des processeurs modernes, elles sont limitées par la taille d'un mot de la machine (16, 32 ou 64 bits) et sur certaines machines (notamment les processeurs RISC) elles ne peuvent opérer que sur les mots alignés avec les frontières de la mémoire : on peut manipuler directement un entier à 32 bits seulement si celui-ci se trouve à une adresse qui est $\equiv 0 (\mathrm{mod}\, 4)$. Cette restriction entraînera une complexification des routines qui récupèrent ou émettent les chaînes de bits dans la séquence compressée.

Les algorithmes d'optimisation que nous avons présentés peuvent être facilement modifiés pour ne considérer que les paramètres satisfaisant une contrainte de la forme

$$\min(k'_{max}, k+1) + \sum_{j=0}^{k} m_j = sb \tag{7.26}$$

avec $s \in \mathbb{N}$ tout en minimisant la différence à l'entropie et respectant $h'(k'_{max}) \geqslant N$. On pourrait même contraindre $k'_{max} \leqslant b$ de façon à pouvoir lire complètement le préfixe d'une seule lecture de b bits. Cette contrainte est satisfaite dès que $M \leqslant b$ puisque $k'_{max} \leqslant M-1$, une conséquence du code unaire tronqué. En posant $b = 4$ dans l'éq. (7.26), on pourrait obtenir les paramètres $\{3, 3, 3, 0\}$. Ce code aurait une forme donnée par

Code	Nombres représentés
$\boxed{0 \;\; x_2 \;\; x_1 \;\; x_0}$	$0 - 7$
$\boxed{1 \;\; 0 \;\; x_5 \;\; x_4 \;\; x_3 \;\; x_2 \;\; x_1 \;\; x_0}$	$8 - 71$
$\boxed{1 \;\; 1 \;\; 0 \;\; x_8 \;\; x_7 \;\; x_6 \;\; x_5 \;\; x_4 \;\; x_3 \;\; x_2 \;\; x_1 \;\; x_0}$	$72 - 583$
$\boxed{1 \;\; 1 \;\; 1 \;\; x_8 \;\; x_7 \;\; x_6 \;\; x_5 \;\; x_4 \;\; x_3 \;\; x_2 \;\; x_1 \;\; x_0}$	$584 - 1096$

où tous les segments sont alignés sur des *nibbles* [4]. Ce code fortement contraint est certainement moins bon qu'un code *Start/Stop* sans la contrainte, mais l'algorithme de recherche contrainte nous garantit que c'est le meilleur code qui respecte toutes les contraintes.

7.5.4 Résultats

Le tableau 7.8 montre les résultats obtenus par les codes $(Start, Step, Stop)$ et les codes $Start/Stop$ dont les paramètres ont été obtenus par l'algorithme vorace et l'algorithme de recherche contrainte. Aux tableaux 7.9, 7.10 et 7.11, on énumère les paramètres obtenus par les algorithmes d'optimisation pour les codes $(Start, Step, Stop)$ et les codes $Start/Stop$. Au tableau 7.12 on compare les résultats obtenus en variant M pour l'algorithme d'optimisation à recherche contrainte aux résultats obtenus par l'algorithme vorace. Toutes les expériences ont été réalisées avec $N = 2^{16} = 65536$.

4 Rappelons que les *nibbles* sont des paquets de 4 bits.

				Start/Stop	
Loi	$\mathcal{H}(X)$	Huffman	$(Start, Step, Stop)$	vorace	$M = 6$
Zipf †	0.6670	1.2483	1.2704	1.2584	1.2675
Exp	1.4935	1.5766	1.8289	1.5766	1.6067
16 Exp ‡	5.4346	5.4637	6.2738	5.5304	5.6028
32 Exp ‡	6.4327	6.4618	6.8279	6.5282	6.6011
$\lvert\mathcal{N}(0,1)\rvert$	1.1080	1.3672	1.6420	1.3672	1.3672
$\lvert\mathcal{N}(0,10)\rvert$	4.3708	4.4028	5.6046	4.5229	4.5524
$\lvert\mathcal{N}(0,100)\rvert$	7.6894	7.7095	8.1513	7.7878	7.7968
Géométrique, $p = \frac{1}{2}$	2.0000	2.0000	2.2593	2.0000	2.0854
Géométrique, $p = \frac{1}{4}$	3.2381	3.2765	3.6281	3.9820	3.3824
Géométrique, $p = \frac{1}{10}$	4.6824	4.7191	5.6969	4.8974	4.8048

TAB. 7.8 – Comparaison des codes $(Start, Step, Stop)$, $Start/Stop$ et Huffman pour quelques distributions. La colonne Huffman correspond à la longueur moyenne du code obtenu grâce à l'algorithme de Huffman. † Loi de Zipf telle que trouvée dans [220], soit $P(Z = z) \approx (z \ln(1.78N))^{-1}$. ‡ C'est-à-dire $\frac{1}{16}X \sim Exp$, $\frac{1}{32}X \sim Exp$.

Les temps d'optimisation n'apparaissent pas aux tableaux comparatifs. Dans le cas de l'algorithme d'optimisation des codes $(Start, Step, Stop)$, le temps se compte dans l'ordre de la milliseconde sur un ordinateur général courant, soit un ordinateur Pentium III à 500 MHz. Pour l'algorithme à recherche contrainte le temps augmente de façon exponentielle avec M, mais pour $M = 6$, nous trouvons encore des temps dans l'ordre de la petite dizaine de millisecondes. L'algorithme vorace s'exécute dans l'ordre de la milliseconde. Pour fins de comparaison, l'algorithme qui calcule les longueurs exactes pour un code de Huffman (l'algorithme de Moffat et Katajainen [146]) est environ cinq fois plus lent que l'algorithme vorace sur les mêmes données. Pour $M = 5$, l'algorithme par recherche contrainte prend à peu près le même temps que l'algorithme de Moffat et Katajeinen pour trouver les paramètres optimaux.

Les implémentations des algorithmes sont relativement efficaces, mais il est possible que des implémentations plus poussées (en assembleur, ou en utilisant une caractéristique spéciale de la machine sous-jacente, par exemple) puissent être légèrement plus rapides, mais les algorithmes ont une complexité qui est dominée par le calcul de la FMC, ce qui fait qu'un algorithme plus rapide devra calculer la FMC plus efficacement. Calculer la FMC de façon nettement plus rapide semble une tâche impossible, du moins très difficile sans utiliser des astuces qui dépendent du matériel.

On remarque, en comparant les tableaux 7.9, 7.10 et 7.11, que le nombre de codes alloués inutilement est très grand dans le cas des codes $(Start, Step, Stop)$. Dans tous les cas, on parle d'un nombre total de codes double de celui désiré ! Nous avions glissé un mot à propos de ce problème à la section 7.5.1, mais on voit, grâce aux résultats numériques, que la situation est problématique. On avait formulé la contrainte $h(k_{max}) \geqslant N$, soit $h(k_{max}) = N + \varepsilon$, que l'on cherchait à satisfaire tout en minimisant $\varepsilon \in \mathbb{Z}^*$. Les codes $Start/Stop$ réussissent beaucoup mieux à cet égard. On voit que les codes $Start/Stop$ générés par l'algorithme vorace et l'algorithme à recherche contrainte réduisent considérablement le nombre de codes superflus.

$(Start, Step, Stop)$		
Loi	Paramètres	Intervalle
Zipf	$(0, 1, 16)$	$0 - 131070$
Exp	$(0, 1, 16)$	$0 - 131070$
16 Exp‡	$(5, 1, 11)$	$0 - 131039$
32 Exp‡	$(5, 1, 11)$	$0 - 131039$
$\lvert\mathcal{N}(0, 1)\rvert$	$(0, 1, 16)$	$0 - 131070$
$\lvert\mathcal{N}(0, 10)\rvert$	$(0, 1, 16)$	$0 - 131070$
$\lvert\mathcal{N}(0, 100)\rvert$	$(6, 1, 16)$	$0 - 131007$
Géométrique, $p = \frac{1}{2}$	$(0, 1, 16)$	$0 - 131070$
Géométrique, $p = \frac{1}{4}$	$(0, 1, 16)$	$0 - 131070$
Géométrique, $p = \frac{1}{10}$	$(0, 1, 16)$	$0 - 131070$

TAB. 7.9 – Les paramètres des codes $(Start, Step, Stop)$ obtenus dans le tableau 7.8. Les paramètres donnés sont $(Start, Step, k_{max})$. On remarque que plusieurs lois diffèrentes recoivent le même ensemble de paramètres, dû à la rigidité du code. On remarque aussi qu'on trouve des paramètres qui, bien que minimisant la longueur moyenne, produisent un grand nombre de codes superflus. ‡ C'est-à-dire $\frac{1}{16}X \sim Exp$, $\frac{1}{32}X \sim Exp$.

$Start/Stop$ vorace		
Loi	Paramètres	Intervalle
Zipf	$\{0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 12\}$	$0 - 65649$
Exp	$\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16\}$	$0 - 65546$
16 Exp‡	$\{3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 13\}$	$0 - 65671$
32 Exp‡	$\{4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12\}$	$0 - 65791$
$\lvert\mathcal{N}(0, 1)\rvert$	$\{0, 0, 0, 0, 0, 16\}$	$0 - 65540$
$\lvert\mathcal{N}(0, 10)\rvert$	$\{2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 14\}$	$0 - 65579$
$\lvert\mathcal{N}(0, 100)\rvert$	$\{6, 0, 0, 0, 0, 0, 10\}$	$0 - 65919$
Géométrique, $p = \frac{1}{2}$	(code unaire)	$0 - 65535$
Géométrique, $p = \frac{1}{4}$	(code unaire)	$0 - 65535$
Géométrique, $p = \frac{1}{10}$	$\{2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 13\}$	$0 - 65619$

TAB. 7.10 – Paramètres obtenus pour les codes $Start/Stop$ du tableau 7.8. On voit que contrairement aux codes $(Start, Step, Stop)$, le nombre de codes superflus introduits pour satisfaire la contrainte $h'(k'_{max}) \geqslant N$ est petit. ‡ C'est-à-dire $\frac{1}{16}X \sim Exp$, $\frac{1}{32}X \sim Exp$.

Start/Stop avec $M = 6$		
Loi	Paramètres	Intervalle
Zipf	$\{0, 0, 1, 2, 3, 10\}$	$0 - 65611$
Exp	$\{0, 0, 0, 0, 2, 14\}$	$0 - 65543$
16 Exp‡	$\{4, 0, 0, 0, 2, 10\}$	$0 - 65663$
32 Exp‡	$\{5, 0, 0, 0, 2, 9\}$	$0 - 65791$
$\lvert\mathcal{N}(0, 1)\rvert$	$\{0, 0, 0, 0, 0, 16\}$	$0 - 65540$
$\lvert\mathcal{N}(0, 10)\rvert$	$\{3, 0, 0, 0, 0, 13\}$	$0 - 65575$
$\lvert\mathcal{N}(0, 100)\rvert$	$\{6, 0, 0, 0, 1, 9\}$	$0 - 65919$
Géométrique, $p = \frac{1}{2}$	$\{0, 0, 0, 1, 1, 14\}$	$0 - 65544$
Géométrique, $p = \frac{1}{4}$	$\{1, 0, 1, 0, 2, 12\}$	$0 - 65563$
Géométrique, $p = \frac{1}{10}$	$\{3, 0, 0, 1, 1, 11\}$	$0 - 65607$

TAB. 7.11 – Paramètres obtenus pour les codes *Start/Stop* avec la contrainte $M = 6$ du tableau 7.8. Ici aussi, peu de codes superflus sont assignés. ‡ C'est-à-dire $\frac{1}{16}X \sim Exp$, $\frac{1}{32}X \sim Exp$.

Start/Stop contraints								
Loi	$M = 3$	$M = 4$	$M = 5$	$M = 6$	$M = 7$	$M = 8$	vorace	$\mathcal{H}(X)$
Zipf	1.4664	1.3371	1.2903	1.2675	1.2565	1.2528	**1.2484**	0.6671
Exp	2.1860	1.8025	1.6600	1.6067	1.5880	1.5810	**1.5766**	1.4935
16 Exp‡	6.1587	5.7916	5.6556	5.6028	5.5708	5.5538	**5.5304**	5.4346
32 Exp‡	7.1514	6.7874	6.6519	6.6011	6.5690	6.5529	**6.5282**	6.4327
$\lvert\mathcal{N}(0, 1)\rvert$	1.6776	1.4083	1.3683	**1.3672**	**1.3672**	**1.3672**	**1.3672**	1.1080
$\lvert\mathcal{N}(0, 10)\rvert$	5.0431	4.6616	4.5673	4.5524	4.5515	4.5493	**4.5229**	4.3708
$\lvert\mathcal{N}(0, 100)\rvert$	8.2926	8.0138	7.8429	7.7968	**7.7878**	**7.7878**	**7.7878**	7.6894
Géométrique, $p = \frac{1}{2}$	2.7111	2.3525	2.1729	2.9854	2.0392	2.0166	**2.0000**	2.0000
Géométrique, $p = \frac{1}{4}$	3.9775	3.6223	3.4711	3.3824	3.3378	**3.3102**	3.9820	3.2381
Géométrique, $p = \frac{1}{10}$	5.4391	5.0440	4.8781	4.8080	4.7726	4.7598	**4.8974**	4.6824

TAB. 7.12 – Dégradations encourrues par le petit nombre de paramètres dans les codes *Start/Stop*. Ici, résultats pour les mêmes lois que le tableau 7.8, mais avec différents M. On voit que plus M est grand, meilleure est la solution. Notez qu'en gras sont indiquées les meilleures solutions. ‡ C'est-à-dire $\frac{1}{16}X \sim Exp$, $\frac{1}{32}X \sim Exp$.

7.5.5 Les codes $(Start, Step, Stop)$ et $Start/Stop$ en codes universels

Les codes $(Start, Step, Stop)$ et $Start/Stop$ peuvent être facilement transformés en codes universels tout en demeurant efficaces pour les petits entiers. Nous avions remarqué que le défaut de certains codes universels est de donner des codes plutôt longs au petits entiers en plus de ne faire que des suppositions judicieusement vagues sur la nature de la fonction de distribution. Or, il est possible de remédier à cette situation en étendant la définition des codes $(Start, Step, Stop)$ et des codes $Start/Stop$.

Les codes $Start/Stop$ peuvent être transformés en codes universels si on accepte un nombre infini de paramètres. L'ensemble des paramètres devient $\{m_0, m_1, \dots, m_k, m_{k+1}, m_{k+1}, \dots\}$, où les $k+1$ premiers paramètres ont été optimisés en fonction de la distribution (ou d'hypothèses sur cette dernière) et le paramètre $m_{k+1} \in \mathbb{N}$. Nous obtiendrons ainsi un code efficace pour les petits entiers, car les paramètres sont optimisés pour la distribution et de façon à satisfaire la contrainte que les $k+1$ premiers paramètres suffisent à représenter l'intervalle des « petits entiers », et un code universel car la longueur du code pour i très grand est $O((1 + \frac{1}{m_{k+1}}) \lg i)$.

Démontrons cet énoncé. Négligeant les quelques premiers paramètres, on obtient un code qui est essentiellement i représenté en base $2^{m_{k+1}}$ (pensez aux codes tabou alignés, section 7.4.1) et le préfixe n'est que l'encodage unaire de la longueur de cette représentation en base $2^{m_{k+1}}$, soit une longueur de $\log_{2^{m_{k+1}}} i = \lg i / m_{k+1}$ bits. Le suffixe aura une longueur $O(\lg i)$ car la représentation requiert $\log_{2^{m_{k+1}}} i$ « chiffres », chacun nécessitant $\lg 2^{m_{k+1}}$ bits ; ce qui nous donne $(\log_{2^{m_{k+1}}} i)\ (\lg 2^{m_{k+1}}) = \lg i$ bits. Le préfixe est de longueur $O(\lg i / m_{k+1})$. En additionnant les deux, on obtient, asymptotiquement, une longueur de code $O((1 + \frac{1}{m_{k+1}}) \lg i)$ bits pour i, et pour peu que $m_{k+1} \in \mathbb{N}$, on obtient un code universel avec la constante d'universalité $(1 + \frac{1}{m_{k+1}})$.

On utilisera cette démonstration pour montrer que les codes $(Start, Step, Stop)$ peuvent aussi être transformés en codes universels. Les codes $(Start, Step, Stop)$ ne sont qu'un cas spécial des codes $Start/Stop$ où l'ensemble des $k_{max}+1$ paramètres est donné par $\{Start, Step, \dots, Step\}$. En choissisant $Start \in \mathbb{Z}^*$, $Step \in \mathbb{N}$, et posant $k_{max} = \infty$, c'est-à-dire en ne limitant pas la longueur des codes, on obtient un code $(Start, Step, \infty)$. Ce code est universel. Il suffit de reprendre la démonstration pour les codes $Start/Stop$ en remplançant m_{k+1} par $Step$ pour arriver à la même conclusion pour les codes $(Start, Step, Stop)$. Quant aux petits entiers, on pourra optimiser $Start$ et $Step$ selon la fonction de distribution, mais en ayant la contrainte supplémentaire que $Step \in \mathbb{N}$ plutôt que $Step \in \mathbb{Z}^*$.

7.6 Conclusion

Dans ce chapitre, nous avons présenté plusieurs contributions au codage des entiers. Nous avons d'abord présenté un algorithme modifié pour le calcul des bigrammes (voir section 7.2, p. 141). Cette nouvelle méthode permet d'aller chercher quelques pourcents de plus sur le ratio de compression sans augmenter significativement la complexité du décodeur. Cette nouvelle méthode est basée sur l'observation que des symboles très rares reçoivent quand même un code au détriment de bigrammes ; redonner ces codes à des bigrammes permet d'augmenter la compression, bien qu'au coût d'une légère complexification du décodeur. Les symboles rares sont maintenant introduits dans la séquence décompressée grâce à un symbole spécial réservé à cette fin.

A la section 7.3, p. 143, nous présentons de nouvelles solutions analytiques pour des variations des codes de Golomb en plus d'une nouvelle dérivation du résultat de Gallager et Van Voorhis [77]. Ces nouvelles variations des codes nous dispensent de l'utilisation de codes *phase-in* au profit d'un code naturel. Cette simplification peut être nécessaire dans un environnement où la puissance de calcul est faible et où la compression n'est pas cruciale, car ces généralisations introduisent une certaine perte dans le ratio de compression.

Les codes tabou alignés, section 7.4.1, sont utilisés implicitement depuis toujours, mais nous présentons ici pour la première fois une étude systématique de ces codes. Nous donnons la preuve de leur universalité. Nous présentons ensuite les codes tabou généralisés, où la contrainte imposée par la structure en bloc des codes alignés est remplacée par une structure où la seule contrainte est de ne pas trouver un certain motif de bits dans le code (section 7.4.2). Ces codes sont basés sur une énumération combinatoire de toutes les chaînes de bits d'une longueur donnée ne présentant pas ce motif de bits interdit, le tabou. Nous montrons que ce motif devrait être une série de n zéros de façon à limiter le moins possible le nombre de combinaisons valides. Nous montrons aussi que ces codes tabou sont unversels et que la constante d'universalité décroît en fonction de la longueur du tabou. Nous montrons aussi que les codes de Fibonacci, basés sur la représentation de Zeckendorf sont en fait un cas spécial des codes tabou généralisés.

Nous passons, à section 7.5, p. 166, aux codes $(Start, Step, Stop)$, introduits par Fiala et Greene [70], et aux codes $Start/Stop$. Nous montrons que les codes $(Start, Step, Stop)$ sont un cas spécial des codes $Start/Stop$ qui sont plus généraux. Fiala et Greene ayant omis de préciser les méthodes d'optimisation pour ces codes, nous présentons des algorithmes d'optimisation basés sur la fonction de probabilité cumulative pour les codes $Start/Stop$ et $(Start, Step, Stop)$. Nous présentons aussi un algorithme vorace pour optimiser les codes $Start/Stop$. L'algorithme de solution exacte basé sur la fonction de probabilité cumulative peut être contraint pour ne considérer que les solutions ayant une certaine forme, dictée, par exemple, par des considérations de codage/décodage rapide. Les codes $(Start, Step, Stop)$ et $Start/Stop$ contraints requièrent très peu de paramètres pour être définis (alors qu'avec la méthode de Huffman, on peut avoir un nombre de paramètre proportionnel au nombre de symboles). Les codes dont l'optimisation des paramètres a été contrainte se prêtent bien à des implémentations efficaces du codeur et du décodeur qui minimisent le nombre de manipulations de bits.

Enfin, nous montrons comment les codes $(Start, Step, Stop)$ et $Start/Stop$ peuvent être modifiés pour devenir des codes universels. Non seulement ces codes modifiés seront universels, mais ils pourront être optimisés en fonction d'une distribution connue pour les petits entiers, réduisant ainsi la longueur moyenne du code.

Chapitre 8

Codage Huffman Adaptatif

8.1 Introduction

Dans ce chapitre, nous présentons nos contributions au codage de Huffman adaptatif. Nous commencerons par présenter l'algorithme de Jones qui utilise les *semi-splay trees* pour maintenir un arbre de code efficace [107]. Les *splay trees* tels que présentés par Tarjan [203] s'adaptent trop rapidement et de façon chaotique selon les observations et nous montrerons qu'ils ne peuvent être utilisés pour la compression, alors que la variante où seulement le *semi-splaying* est utilisé donne de relativement bons résultats. Les premières sections seront consacrées à la description des *splay trees* et à l'algorithme de Jones.

Par la suite, nous présenterons nos contributions. De l'algorithme de Jones, nous tirerons l'algorithme W. L'algorithme W nous permettra de trouver des codes environ un demi bit plus courts en moyenne qu'avec l'algorithme de Jones, sans augmenter la complexité ni la quantité de mémoire nécessaire. L'amélioration fondamentale de l'algorithme W est basé sur une meilleure règle de *splaying*. Enfin, nous augmenterons l'algorithme W pour aboutir à l'algorithme M qui donne d'encore meilleurs résultats. Nous comparerons nos résultats avec l'algorithme de Vitter, l'algorithme Λ [216].

❁ ❁ ❁

Nous avons eu Yoshua Bengio comme coauteur pour les communications et les articles publiés sur l'algorithme M [162, 163, 165, 164]. Il nous a apporté une aide précieuse par ses conseils et la révision des manuscrits.

8.2 Algorithme de Jones

Nous avons mentionné l'algorithme de Jones pour le codage adaptatif de Huffman à la section 6.4.4. L'algorithme de Jones utilise un *splay tree* modifié pour maintenir l'arbre de codes. Dans cette variation, le *splaying*, plutôt que de catapulter les symboles à la racine de l'arbre, réduit par deux la profondeur de la feuille en n'appliquant la régle de *splaying* qu'à un nœud sur deux sur le chemin vers la racine. Nous décrirons l'algorithme de base ainsi que l'algorithme modifié dans

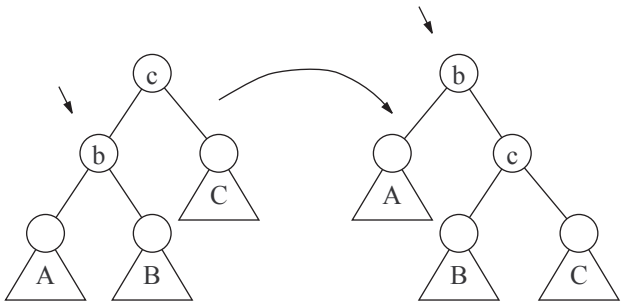


FIG. 8.1 – Un exemple de *splaying* simple. Il y a une opération, qui n'est pas montrée, où l'arbre à transformer est le miroir de celui-ci. La flèche indique le nœud qui est remonté à la racine.

le détail aux sections 8.2.1 et 8.2.2. Nous verrons pourquoi l'algorithme de base, tel que proposé par Tarjan, mène à des situations pathologiques où la longueur des codes devient incontrôlable. La modification à l'algorithme, proposée par Jones pondère l'adaptativité et permet de générer des codes qui sont bons sans toutefois être optimaux [107]. Finalement, nous comparerons les résultats obtenus par l'algorithme de Jones aux résultats obtenus par les algorithmes Λ et de Huffman statique.

8.2.1 Algorithme *splay tree*

L'algorithme du *splay tree* est dû à Tarjan [203]. Alors que les arbres AVL s'efforcent de maintenir la profondeur de l'arbre la plus égale possible, les arbres *splay* vont tenter de maintenir les éléments fréquemment accédés près de la racine de façon à réduire le temps espéré de recherche. L'arbre est transformé de façon à placer la dernière clef accédée à la racine même tout en maintenant les propriété d'arbre de recherche, c'est-à-dire l'ordonnancement stricte des clefs. Nous verrons que pour générer des arbres de codes, il n'est pas nécessaire de satisfaire cette contrainte puisque les symboles se trouvent seulement aux feuilles et les nœuds internes n'ont pour fonction que le maintient de la structure de l'arbre elle-même.

Pour remonter un nœud à la racine, l'algorithme de Tarjan procède par « rotations » successives. Une rotation simple est montrée à la fig. 8.1. On veut ramener le nœud b à la racine. Nous utilisons la convention qu'un sous-arbre à droite ne contient que des clefs inférieures à la clefs qui le domine, alors que le sous-abre de gauche contiendra des clefs qui lui seront supérieures ou égales. En se ramenant à la fig. 8.1, on trouve que le sous-arbre B ne contient que des clefs qui sont supérieures ou égales à la clef b, alors que le sous-arbre A que des clefs strictement inférieures. Le sous-arbre C contient aussi des clefs supérieures ou égales à c, et comme b est strictement inférieur à c, donc supérieures ou égale à b. Dans l'illustration, les positions originales et finales du nœud à faire remonter à la racine sont indiqués par une flèche.

La fig. 8.1 montre une situation où la clef qui subit le *splaying* n'est qu'un niveau plus bas que la racine. La fig. 8.2 montre les étapes successives d'une rotation multiple. L'application successive de la rotation simple, en montant le nœud d'un niveau à la fois, permettra de faire remonter à la racine n'importe quel nœud de l'arbre tout en maintenant l'ordre de ses clefs.

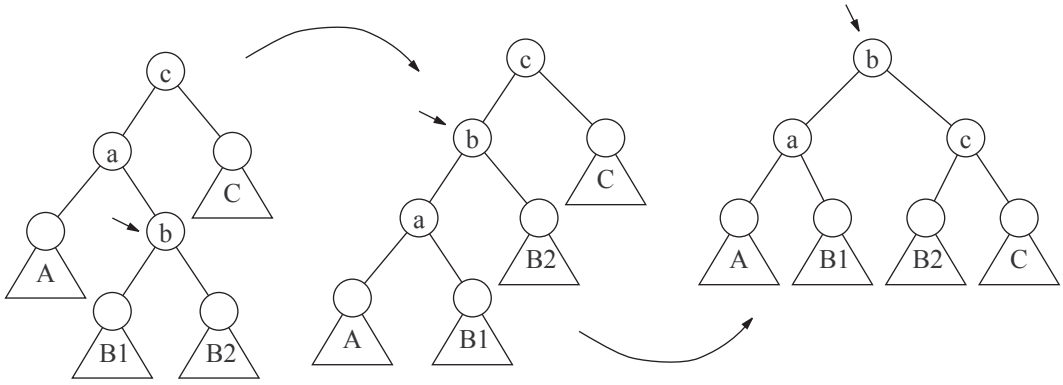


FIG. 8.2 – Un exemple de *splaying* plus complexe. Il y a une opération, qui n'est pas montrée, où l'arbre à transformer est le miroir de celui-ci. Ici aussi, le nœud indiqué par la flèche est remonté à la racine.

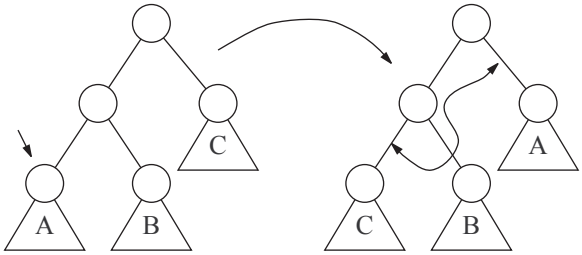


FIG. 8.3 – Un exemple de *splaying* simplifié. Il y a une opération, qui n'est pas montrée, où l'arbre à transformer est le miroir de celui-ci. Plutôt que d'échanger les nœuds de façon à maintenir l'ordre des clefs, on peut se permettre de n'échanger que les pointeurs pour appliquer les rotations. L'ordre des clefs peut ne pas être préservée puisqu'elle n'est pas nécessaire. La flèche indique le nœud qui est remonté à la racine et la double flèche ondulée montre quels pointeurs ont été échangés pour réaliser la rotation.

Alors qu'avec l'algorithme de base il est possible de faire remonter n'importe quel nœud à la racine, les feuilles y compris, l'algorithme modifié ne pourra que faire monter le *parent* de la feuille à la racine. Cela garantit que tous les symboles demeurent dans les feuilles. Cette condition est nécessaire car si nous pouvions adresser des nœuds internes, nous obtiendrions un code ambigu.

Pour fabriquer des codes structurés en arbre, comme les codes de Huffman, il ne nous est pas utile de maintenir un ordonnancement strict des clefs puisque les symboles ne se trouvent qu'au feuilles, celles-ci doivent demeurer des feuilles peu importe la transformation que l'on applique à l'arbre. Maintenir l'ordre des feuilles nous serait au contraire nuisible, puisque dans de nombreux cas — l'alphabet le premier — la valeur numérique des symboles n'a pas de relation évidente avec la fréquence. Enlever la contrainte sur l'ordre des clefs simplifie considérablement l'algorithme, comme le montre la fig. 8.3. L'opération de rotation devient simplement un échange de pointeur entre le nœud à faire monter et son oncle, une opération qui se fait très efficacement.

Comme pour l'algorithme de Huffman, nous obtenons un code structuré en arbre et le code pour un symbole est obtenu en parcourant le chemin depuis la racine jusqu'à la feuille qui le contient. Au début, l'arbre sera un arbre de profondeur la plus égale possible, où toutes les feuilles sont sur au plus deux niveaux différents (pour $N = 2^k + b$ feuilles, $2^k - b$ se trouveront à la profondeur k et $2b$ à la profondeur $k + 1$). Pour compresser la séquence, on émet d'abord le code du prochain symbole, puis on fait remonter le parent de la feuille qui le contient jusqu'à la racine. L'arbre s'ajustera en fonction des symboles qui sont rencontrés dans la séquence. La décompression procédera de la même façon : on décode le prochain code, qui nous mènera à la feuille qui contient le symbole à décoder. Une fois ce symbole décodé, on met à jour l'arbre et on passe au prochain symbole.

La fig. 8.2 porte à croire que les opérations laissent les arbres essentiellement balancés, mais ce n'est pas le cas. Si c'était vraiment le cas, nous serions embêtés car les codes obtenus seraient tous de même longueur, ce qui ne nous donnerait pas de compression du tout ! Considérez la configuration finale de l'arbre à la fig. 8.2. Si le prochain nœud à être remonté à la racine était le nœud a, l'arbre sera nettement débalancé et les feuilles du sous-arbre C recevraient les codes les plus longs — en supposant que les triangles qui figurent des sous-arbres soient d'égales profondeurs.

Cependant, appliqué tel quel sur les fichiers des corpus de Calgary et de Canterbury (voir l'appendice D), cet algorithme donne de très mauvais résultats. En fait, les résultats sont si mauvais que presque tous les fichiers se trouvent à prendre de l'expansion plutôt qu'être compressés !

Que se passe-t-il ? On s'attendrait à voir les symboles les plus fréquents occuper les feuilles près de la racine et les symboles rares relégués à des positions beaucoup plus bas dans l'arbre, comme à la fig. 8.4, et obtenir des taux de compression raisonnables. Or, ce n'est pas du tout ce qui arrive. Par la nature même de l'algorithme, le dernier symbole observé est catapulté près de la racine, déclassant tous les autres symboles : on obtient un arbre tel que l'un des fils de la racine est le dernier symbole observé et l'autre le reste de l'arbre. Bien que les symboles soient tirés selon une distribution pointue, localement, la structure même de la séquence peut empêcher que les symboles fréquents se répètent les uns à la suite des autres. C'est le cas des langages naturels, par exemple le français et l'anglais. Malgré que e soit la lettre la plus fréquente, on ne

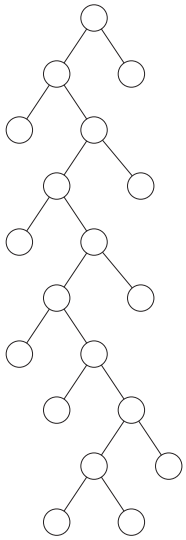


FIG. 8.4 – Un exemple de *splaying* dénéré.

trouve pas de longues séries de e consécutifs, pire, après avoir observé un e, on s'attend plutôt à une série plus ou moins longue d'autres lettres. Ainsi, même si e est la lettre la plus fréquente, elle se trouve déclassée presque après chaque occurence. Les lettres les plus rares sont d'autant plus déclassées qu'elles sont rares ; menant inexorablement à un arbre du type de la fig. 8.4, où les codes deviennent essentiellement des codes unaires du rang des symboles, menant ainsi à la catastrophe.

8.2.2 Algorithme de Jones : *splay tree* modifié

Comme l'algorithme de *splaying* tel que proposé par Tarjan est mauvais sur les sources qui ne sont pas i.i.d., Jones propose d'utiliser non pas le *splaying* simple décrit à la sous-section précédante, mais une variante, le *semi-splaying*. Le *semi-splaying* consiste, plutôt que de monter directement la feuille à la racine, à réduire par un facteur deux sa profondeur [107].

La modification à apporter à l'algorithme de *splaying* est triviale. On applique la première rotation au parent immédiat de la feuille, c'est-à-dire la faire monter d'un seul niveau. Cette opération terminée, on recommencera avec le *nouveau* grand-parent. On recommence encore avec le nouveau grand-parent, jusqu'à ce que nous ayons atteint la racine. Cela aura pour effet de réduire de moitié la profondeur d'une feuille tout en réduisant le nombre de déclassements. En modérant la vitesse à laquelle les feuilles montent à la racine, on réduit considérablement l'effet dégénératif et on empêche l'apparition de cas pathologiques comme celui de la fig. 8.4. On verra les résultats obtenus à la sous-section suivante, où les codes générés par l'algorithme de Jones sont comparés aux codes générés par les algorithmes de Huffman (utilisant la « vraie » distribution) et Λ. Dans ce chapitre, au fur et à mesure que nous présenterons de nouveaux algorithmes, nous augmenterons le tableau comparatif des nouveaux résultats de façon à percevoir l'effet des raffinements successifs.

La complexité computationnelle de l'algorithme de Jones est proportionnelle à $O(\mathcal{H}(X))$ si on fait l'hypothèse que l'algorithme adaptatif converge rapidement vers l'entropie de la source (ce qui semble être approximativement le cas, comme on le verra à la sous-section suivante). En effet, le nombre d'opérations pour une mise à jour d'un symbole dans l'arbre est dominé par la longueur du chemin entre cette feuille et la racine, lequel est approximativement $-\lg P(X = x)$, ce qui donne, sur toutes les feuilles, approximativement $\mathcal{H}(X)$. De plus, nous ne faisons les opérations compliquées qu'un nœud sur deux, ce qui fait que cet algorithme est somme toute beaucoup plus rapide que l'algorithme de *splay* classique.

8.2.3 Résultats

Les résultats de l'algorithme de Jones sont montrés aux tableaux 8.1 et 8.2. Ils y sont comparés aux codes de Huffman (sans le coût de la description du code) et à l'algorithme Λ de Vitter, où on montre la longueur moyenne des codes. On se souviendra que nous avons présenté l'algorithme Λ à la section 6.4.3. L'algorithme de Jones performe considérablement moins bien que l'algorithme statique de Huffman et que l'algorithme de Vitter. Les codes générés par l'algorithme de Jones sont systématiquement plus longs que les codes obtenus par l'algorithme Λ (à la seule exception du fichier *sum*, tableau 8.2), mais seulement d'environ un demi bit, sur la plupart des fichiers. Sur le corpus de Calgary, on observe un écart moyen de 0.61 bits par rapport à l'algorithme Λ, et 0.63 bits par rapport à l'algorithme de Huffman statique. Quant au corpus de Canterbury, les différences sont de 0.55 bits et 0.59 bits, pour les algorithmes Λ et Huffman, respectivement. Toutefois, on constate des écarts aussi grands que 1.26 bits (sur le fichier *bib*, tableau 8.1). Notez que les longueurs moyennes dans tous les tableaux, y compris ceux des sections suivantes, sont données en bits. On y montre aussi les moyennes pour chaque corpus. Toutes les moyennes sont pondérées par le nombre d'octets des fichiers. Cela empêche les petits fichiers qui ne laissent pas le temps aux algorithmes de s'adapter convenablement de hausser artificiellement la moyenne.

La grande simplicité de l'algorithme de *semi-splaying*, en plus de n'appliquer la règle qu'un nœud sur deux sur le chemin vers la racine, nous donne un algorithme très rapide pour la compression. Sur un ordinateur général de puissance modérée (un Pentium II 333 MHz), on peut atteindre plusieurs mégaoctets par seconde, ce qui, malgré les lacunes au niveau de la compression, peut faire de l'algorithme de Jones un algorithme intéressant pour faire de la compression adaptative, voire même la compression temps réel.

8.3 *Splay tree* pondéré : algorithme W

L'algorithme de Jones n'utilise pas explicitement les fréquences des symboles pour maintenir l'arbre. Les fréquences ne sont considérées qu'implicitement à travers la forme de l'arbre et la règle de *semi-splaying* qui est utilisée pour le transformer au fil des observations. Au contraire, les algorithmes de Huffman et Λ utilisent explicitement les fréquences. L'algorithme de Huffman les utilise pour bâtir l'arbre et ne s'en soucie plus par la suite. L'algorithme Λ de Vitter, qui est l'aboutissement des algorithmes de Faller, Gallager et Knuth, maintient la fréquence de chaque symbole dans les feuilles et dans les nœuds internes la somme des fréquences de toutes les feuilles qui en descendent (voir les sections 6.4.2 et 6.4.3 et en particulier les fig. 6.4 et 6.5).

Nous proposons dans cette section un autre algorithme qui utilise à la fois la propriété de *splaying* et les fréquences pour produire des codes efficaces de façon adaptative. Cet algorithme,

Corpus de Calgary			
Fichier	Jones	Λ	Huffman
bib	6.26	5.00	5.23
book1	5.50	4.56	4.56
book2	5.56	4.82	4.82
geo	6.61	5.69	5.67
news	5.88	5.23	5.22
obj1	6.04	6.07	5.97
obj2	6.60	6.30	6.29
paper1	5.73	5.04	5.01
paper2	5.55	4.65	4.63
paper3	5.59	4.71	4.69
paper4	5.58	4.80	4.73
paper5	5.72	5.05	4.97
paper6	5.59	5.07	5.04
pic	1.71	1.66	1.66
progc	5.92	5.26	5.23
progl	5.19	4.81	4.80
progp	5.39	4.92	4.90
trans	5.81	5.58	5.57
moyenne	5.11	4.49	4.49

TAB. 8.1 – Les résultats de l'algorithme de Jones sur le corpus de Calgary, comparés aux résultats obtenus par les algorithmes de Huffman et Λ.

Corpus de Canterbury			
Fichier	Jones	Λ	Huffman
alice29.txt	5.12	4.62	4.62
asyoulik.txt	5.75	4.85	4.85
bible.txt	5.30	4.39	4.35
cp.html	6.18	5.31	5.27
ecoli	2.42	2.25	2.00
fields.c	5.65	5.13	5.04
grammar.lsp	5.38	4.87	4.66
kennedy.xls	3.89	3.60	3.59
kjv.gutenberg	5.19	4.42	4.42
lcet10.txt	5.43	4.70	4.70
plrabn12.txt	5.58	4.58	4.58
ptt5	1.71	1.66	1.66
sum	4.93	5.42	5.37
world192.txt	5.77	5.04	5.04
xargs.1	5.75	5.10	4.92
moyenne	4.46	3.86	3.78

TAB. 8.2 – Les résultats de l'algorithme de Jones sur le corpus de Canterbury, comparés aux résultats obtenus par les algorithmes de Huffman et Λ.

l'algorithme W (pour *weighted splay tree*), maintiendra dans chaque feuille la fréquence du symbole qui lui est associé et, récursivement jusqu'à la racine, chaque nœud qui n'est pas une feuille contient la somme des fréquences de ses deux fils, comme il est montré à la fig. 8.5. L'algorithme de Jones pondère l'adaptativité en n'utilisant que le *semi-splaying* car le *splaying* de base a tendance à désorganiser l'arbre en propulsant le dernier symbole observé à la racine, et à le faire dégénérer, comme à la fig. 8.4. L'algorithme W, sans être optimal s'assure de faire en sorte que les nœuds les plus fréquents se retrouvent plus haut dans l'arbre en utilisant une règle de *splaying* modifiée. Plutôt que d'appliquer le *splaying* inconditionnellement jusqu'à la racine, on va appliquer le *splaying* pour faire remonter un nœud seulement s'il a une plus grande fréquence que son *oncle*.

L'algorithme W procédera de la façon suivante. On émet d'abord le code du prochain symbole dans la séquence. Ensuite, on incrémente la fréquence qui lui est associée dans la feuille qui le contient. Après cela, on procède vers la racine en propageant cet incrément, en appliquant la nouvelle règle de *splaying* lorsque elle s'applique. Comme on doit propager l'incrément, on ne saute plus de grands-parents en grands-parents jusqu'à la racine, mais on remonte les nœuds un à un, en s'assurant que la fréquence du nœud courant soit bel et bien la somme des fréquences de ses fils. Une fois la mise à jour faite, on vérifie si la fréquence du nœud courant excède celle de son oncle (ici, il y a une petite gymnastique à faire pour déterminer de quel côté on est), et si la fréquence du nœud courant excède celle de l'oncle, alors on procède à un *splaying* simplifié, du type décrit à la fig. 8.3. Après le *splaying*, s'il a eu lieu, on recalcule les sommes des fréquences des nœuds qui ont subit la transformation, soit l'ancien parent et l'ancien grand-parent. On monte au nouveau parent, et on recommence la procédure jusqu'à ce que nous atteignions la racine.

Cet algorithme s'assure que les nœuds les plus fréquents (qu'il s'agisse de feuilles ou de nœuds internes) soient plus haut dans l'arbre que les nœuds les moinds fréquents. Cela ne suffit toutefois pas à générer un arbre de code optimal. En effet, la règle de *splaying* ne suffit pas à créer un arbre optimal car elle ne peut pas garantir les deux symboles les moins fréquents partagent le même préfixe, une condition qui, appliquée récursivement à tous les nœuds de l'arbre, permet aux codes de Huffman d'être optimaux (comme nous l'avons vu à la section 6.2.2, où nous trouvons aussi une preuve de cet énoncé). L'algorithme de Jones ne garantit pas que les symboles les plus fréquents soient positionnés plus haut dans l'arbre de code, seulement les plus récemment observés. C'est sans doute cette différence qui fait que l'algorithme W donne des résultats nettement meilleurs que l'algorithme de Jones, comme nous le verrons à la sous-section des résultats.

8.3.1 Algorithme W dans le détail

Le code C++ de l'algorithme de *splaying* est donné à la figure 8.7 et la définition des nœuds utilisés est donnée à la figure 8.6. Il aurait été possible d'implémenter l'algorithme en n'utilisant que des tableaux statiques, comme propose de le faire Jones. Bien que l'utilisation de tableaux nous dispense de l'allocation dynamique des nœuds au moment de l'initialisation de l'algorithme, cela ne sert pas la clarté du code. Jones, par exemple, propose une implémentation où nous avons, pour un alphabet de taille N, un tableau `parent` de $2N-1$ éléments, et deux tableaux `gauche` et `droite`, chacun de $N-1$ éléments. En effet, il n'est pas utile d'assigner des fils aux feuilles, ce qui explique que ces tableaux aient $N-1$ élément plutôt que $2N-1$ comme on aurait pu s'y attendre. Jones estime le nombre de bits de mémoire utilisés pour ces tableaux,

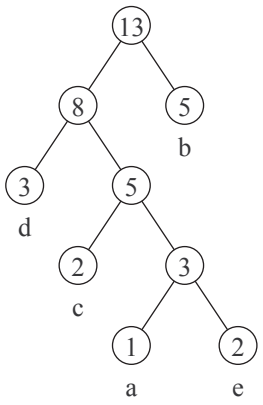


FIG. 8.5 – Un exemple de *splay tree* pondéré pour l'alphabet $\{a, b, c, d, e\}$.

faisant fi des complications extrêmes liés à l'utilisation de tableaux de $2N-1$ éléments de taille $\lceil \lg(2N-1) \rceil$ bits, spécialement lorsque ce nombre de bits est plutôt quelconque par rapport à la taille des mots de la machine [107, p. 1000]. Par exemple, pour $N = 256$, cela nous donnerait des pointeurs de 9 bits. Considérant 9 bits par pointeurs, nous avons $(4N-3) \times 9 = 9189$ bits, soit un peu plus d'un kilo-octet. En utilisant des pointeurs sur 16 bits (qui correspondent en C/C++ au type `short int` sur la plupart des machines), on obtient une consommation de mémoire d'environ deux kilo-octets, une quantité de mémoire (probablement) négligeable. L'algorithme de Vitter, quant à lui, requiert pas moins de 11 tableaux de tailles diverses pour maintenir la structure de données !

Jones propose aussi d'utiliser un compteur de fréquence pour chaque feuille et nœud interne, mais à d'autres fins. Jones propose qu'on les utilise pour obtenir un code arithmétique plutôt qu'un code de Huffman en descendant l'arbre pour générer le code. Ici, nous sommes contraints d'utiliser un entier de la taille maximale permissible pour représenter les fréquences car la séquence peut être très longue.

Si on opte pour une implémentation en arbre utilisant des pointeurs vers des nœuds alloués dynamiquement sur la mémoire libre, on aura besoin d'un autre tableau qui permet de faire l'association entre un symbole et la feuille qui le contient. Il s'agit d'un tableau de N pointeurs de nœuds. Les nœuds sont définis à la fig. 8.6. Les constructeurs et destructeurs n'y sont pas explicités car ils sont très simples. Le constructeur initialise tous les champs du nœud à zéro, ce qui crée une feuille (car les pointeurs aux fils sont nuls). Le destructeur par défaut détruit récursivement les sous-arbres indiqués par les pointeurs aux fils, s'il y a lieu. Pour la mise à jour d'un symbole, on applique la procédure `splay`, décrite à la fig. 8.7, sur la feuille qui contient le dernier symbole observé. Le lecteur aura sans doute remarqué que le type du symbole est `unsigned int`, ce qui permet un très grand nombre de symboles distincts. Nous reviendrons à la section 8.5 sur l'importance de ce choix.

Comme pour l'algorithme de Jones, la complexité computationnelle de l'algorithme W est $O(\mathcal{H}(X))$ puisque qu'une mise à jour pour une feuille implique que nous remontions l'arbre

```
class cWeightedSplayNode
       {
        public: int freq;
                unsigned int symbole;
                cWeightedSplayNode *parent, *gauche, *droite;

        cWeightedSplayNode(); // met tous les champs à zéro
        ~cWeightedSplayNode(); // détruit récursivement le sous-arbre
       };
```

FIG. 8.6 – La définition de la classe `cWeightedSplayNode`, utilisée par l'algorithme de *splaying* pondéré (fig. 8.7).

jusqu'à la racine, et comme le code pour chaque symbole est approximativement de longueur $-\lg P(X = x)$, comme semblent le montrer les résultats (et comme nous le démontrerons à la section 8.4.3), et que chaque symbole est observé avec une probabilité $P(X = x)$, on conclut que pour toute la séquence, le coût est approximativement $n\mathcal{H}(X)$, si n est la longueur de la séquence.

Comme pour l'algorithme de Jones, l'algorithme W est très rapide, bien que quelque peu plus lent, car nous avons quand même à faire le double du nombre d'opérations pour une même séquence. En effet, l'algorithme W ne saute pas de grands-parents en grands-parents, mais remonte l'arbre un étage à la fois et on est susceptible d'appliquer le *splaying* à chaque étape. L'algorithme W est toutefois beaucoup plus rapide que l'algorithme de Tarjan puisque la profondeur moyenne des symboles est beaucoup plus petite. La performance obtenue à partir de l'implémentation en C++ nous permet quand même de compresser les séquences à une vitesse de plusieurs méga-octets par seconde.

8.3.2 Résultats

Les tableaux 8.3 et 8.4 présentent les résultats de l'algorithme W obtenus sur les corpus de Calgary et de Canterbury, comparés aux résultats des algorithmes de Jones, Huffman et Λ. L'algorithme W donne des longueurs de codes moyennes plus grandes que l'algorithme Λ (une exception, le fichier *ecoli*), et plus courtes que l'algorithme de Jones (encore une exception, le fichier *obj1*). L'algorithme W donne des codes en moyenne 0.49 bits plus courts que l'algorithme de Jones sur les deux corpus. L'algorithme W bat donc l'algorithme de Jones par environ un demi bit, ce qui est une amélioration importante. L'algorithme W obtient des codes 0.13 bits plus longs que l'algorithme Λ sur le corpus de Calgary et plus longs de 0.11 bits sur le corpus de Canterbury. La différence avec l'algorithme de Huffman est légèrement plus grande, mais on ne trouve quand même que 0.13 bits pour le corpus de Calgary et 0.19 pour le corpus de Canterbury.

On voit que l'ajout de la règle de *splaying* conditionnelle, qui ne permet le *splaying* que si un nœud est plus fréquemment visité que son oncle, améliore grandement l'algorithme de *splaying* simple — qui était tout à fait désastreux — à un point tel que le nouvel algorithme dépasse même l'algorithme de Jones où c'est une règle de *semi-splaying* qui est utilisée.

8.4 Algorithme M

L'algorithme W donne de meilleurs codes que l'algorithme de Jones car il s'assure de donner les codes les plus courts aux symboles les plus fréquents plutôt qu'aux plus récents. Pour

```
void splay(cWeightedSplayNode * node)
 {
  node->freq++;

  while ((node->parent) && (node->parent->parent))
   {
    cWeightedSplayNode *p  = node->parent;
    cWeightedSplayNode *gp = node->parent->parent;

    if (node->freq > uncle(node)->freq)
         if (gp->gauche== p) // splaying a gauche
              {
               if (p->gauche == node)
                    {
                     swap( p->gauche, gp->droite );
                     p->gauche->parent =  p;
                    }
               else {
                     swap( p->droite, gp->droite );
                     p->droite->parent = p;
                    }
               gp->droite->parent = gp;
               node = gp->droite;
              }
         else // splaying a droite
              {
               if (p->gauche==node)
                    {
                     swap(p->gauche, gp->gauche);
                     p->gauche->parent = p;
                    }
               else
                     swap(p->droite, gp->gauche);
                     p->droite->parent = p;
                    }
               gp->gauche->parent = gp;
               node = gp->gauche;

    else node=p;

    // maintenir les sommes
    p->freq = (p->gauche->freq)+(p->droite->freq);
    gp->freq= (gp->droite->freq)+(gp->gauche->freq);
   }
 }
```

FIG. 8.7 – L'algorithme de *splaying* pondéré. La fonction `uncle`, qui n'est pas décrite ici, retourne le nœud qui est le frère du parent du nœud en argument : son oncle.

Corpus de Calgary				
Fichier	Jones	W	Λ	Huffman
bib	6.26	5.33	5.00	5.23
book1	5.50	4.71	4.56	4.56
book2	5.56	4.98	4.82	4.82
geo	6.61	5.76	5.69	5.67
news	5.88	5.40	5.23	5.22
obj1	6.04	6.19	6.07	5.97
obj2	6.60	6.46	6.30	6.29
paper1	5.73	5.13	5.04	5.01
paper2	5.55	4.78	4.65	4.63
paper3	5.59	4.87	4.71	4.69
paper4	5.58	4.98	4.80	4.73
paper5	5.72	5.15	5.05	4.97
paper6	5.59	5.17	5.07	5.04
pic	1.71	1.68	1.66	1.66
progc	5.92	5.36	5.26	5.23
progl	5.19	4.90	4.81	4.80
progp	5.39	5.09	4.92	4.90
trans	5.81	5.69	5.58	5.57
moyenne	5.11	4.62	4.49	4.49

TAB. 8.3 – Les résultats de l'algorithme W sur le corpus de Calgary, comparés aux résultats obtenus par les algorithmes de Jones, Huffman et Λ.

Corpus de Canterbury				
Fichier	Jones	W	Λ	Huffman
alice29.txt	5.12	4.82	4.62	4.62
asyoulik.txt	5.75	4.96	4.85	4.85
bible.txt	5.30	4.50	4.39	4.35
cp.html	6.18	5.40	5.31	5.27
ecoli	2.42	2.24	2.25	2.00
fields.c	5.65	5.25	5.13	5.04
grammar.lsp	5.38	4.91	4.87	4.66
kennedy.xls	3.89	3.66	3.60	3.59
kjv.gutenberg	5.19	4.68	4.42	4.42
lcet10.txt	5.43	4.85	4.70	4.70
plrabn12.txt	5.58	4.67	4.58	4.58
ptt5	1.71	1.68	1.66	1.66
sum	4.93	5.58	5.42	5.37
world192.txt	5.77	5.19	5.04	5.04
xargs.1	5.75	5.17	5.10	4.92
moyenne	4.46	3.97	3.86	3.78

TAB. 8.4 – Les résultats de l'algorithme W sur le corpus de Canterbury, comparés aux résultats obtenus par les algorithmes de Jones, Huffman et Λ.

certain types de séquences, cette tactique pourrait s'avérer moins bonne que l'approche préconisée par Jones — en particulier les séquences ayant d'assez longues répétitions du même symbole — mais en pratique, considérant les fichiers des deux corpus testés, l'algorithme W bat l'algorithme de Jones par une bonne marge. L'algorithme W demeure sous-optimal car il ne respecte pas l'une des conditions nécessaires des codes optimaux qui stipule que les deux symboles les moins fréquents doivent avoir les codes les plus longs et partager le même préfixe. Cette condition, appliquée récursivement sur les feuilles et les nœuds internes de l'arbre génère un code optimal. Sans chercher à satisfaire complètement cette contrainte — comme l'algorithme Λ — comment peut-on améliorer l'algorithme W ?

On pourrait s'assurer par exemple, que tous les symboles de même fréquence reçoivent des codes de même longueur. Nous avons proposé un tel algorithme, l'algorithme M qui va utiliser des feuilles spéciales où tous les symboles de même fréquence se retrouvent ensemble [162, 163, 165, 164]. Cet algorithme a été présenté en rapports techniques au DIRO [162, 163], à la conférence sur la compression de données de Snowbird 1998 [165] ainsi que dans une revue de programmation grand public [164]. Dans cette section, nous expliquerons dans le détail l'algorithme M et comparerons ses performances aux autres algorithmes présentés jusqu'à maintenant dans ce chapitre.

L'algorithme que nous présentons ici est cependant une version légèrement raffinée de l'algorithme M présenté dans les publications précédentes. La règle de *splaying* a été simplifiée. Cette simplification permet de gagner quelques poussières de bits en compression.

L'algorithme M diffère de l'algorithme W de plusieurs façons importantes. La première différence, c'est que chaque feuille de l'arbre construit par l'algorithme M ne représente plus un seul symbole, mais un *ensemble* de symboles. Cet ensemble contiendra tous les symboles qui ont la même fréquence d'observation. Cela permet de s'assurer que tous les symboles de même fréquence reçoivent des codes de même longueur. La seconde différence se trouve dans la façon de calculer le poids des nœuds. Alors qu'avec l'algorithme W il suffisait d'avoir la fréquence du symbole comme poids de la feuille, avec l'algorithme M on a que le poids de la feuille est donnée par le nombre de symboles dans l'ensemble fois la fréquence de ces symboles. La règle de pondération revient à calculer le poids d'une feuille-ensemble comme s'il s'agissait d'un sous-arbre dont toutes les feuilles auraient un même poids. Pour les autres algorithmes, il suffisait de parcourir le chemin entre la racine et la feuille qui contient le symbole pour générer le code, mais le code généré par l'algorithme M est bipartite. Le préfixe est généré par le chemin entre la racine et la feuille qui contient le symbole, ce qui nous indiquera quelle feuille considérer, mais comme cette feuille représente un ensemble de symboles, nous devons avoir un suffixe qui donne le rang du symbole dans cet ensemble. Il se peut que nous ayons à nous dispenser du suffixe si l'ensemble ne contient qu'un seul symbole, sinon nous utiliserons un code *phase-in* pour encoder l'index (voir section 5.3.4.2). L'utilisation d'un code *phase-in* est tout à fait justifiée, puisque cela correspond à l'utilisation d'un sous-arbre de profondeur la plus égale possible. La longueur d'un code *phase-in* est donnée par l'éq. (5.6).

Avec l'algorithme W, la mise à jour de l'arbre pour un symbole était simple. Il ne s'agissait que d'incrémenter sa fréquence de un et de lancer le *splaying* pour propager l'incrément jusqu'à la racine et appliquer les transformations sur l'arbre au besoin, c'est-à-dire lorsqu'un nœud devient plus fréquent que son oncle. Pour l'algorithme M, l'opération de mise à jour commencera par une *migration* du symbole, d'où le M de l'algorithme M. Peu importe la façon dont l'arbre

aura été initialisé — nous reviendrons sur ce sujet dans un instant — il est tel que tous les symboles ayant une même fréquence d'observation se retrouvent dans la même feuille-ensemble.

Décrivons l'opération de migration. Le prochain symbole à coder, disons a, est observé. Avant l'observation du symbole, nous avons pour a une fréquence de f. Nous émettons le code pour a et procédons à la mise à jour de l'arbre. Dans l'algorithme W, pour la mise à jour il aurait suffit d'incrémenter la fréquence dans la feuille et de procéder au *splaying* pour propager l'incrément jusqu'à la racine tout en appliquant les changements à l'arbre. Dans l'algorithme M, nous allons faire migrer le symbole a de la feuille-ensemble qui contient les symboles de fréquence f à la feuille-ensemble qui contient les symboles de fréquence $f + 1$. On retire a de la première feuille-ensemble, dont le poids est décrémenté par f (car, rappelons-nous, le poids est donné par la fréquence fois la cardinalité de l'ensemble), puis on ajoute a à la feuille-ensemble de fréquence $f + 1$ dont le poids est incrémenté par $f + 1$. Une fois la migration de a accomplie, on procède au *splaying* de la feuille-ensemble de destination, puis à la propagation sans *splaying* du changement de poids de la feuille-ensemble de départ.

Le *splaying* de l'algorithme M est essentiellement identique au *splaying* de l'algorithme W, à la différence que ce sont les poids qui sont mis à jour plutôt que seulement les fréquences. Les poids, rappelons-le, simulent les sommes des fréquences des feuilles que l'on aurait eut si ce n'eût été de l'utilisation des feuilles-ensembles. Les poids correspondent donc aux fréquences que nous aurions obtenues si chaque feuille-ensemble était remplacée par un sous-arbre de profondeur la plus égale possible.

Revenons à la configuration initiale de l'arbre. Dans les algorithmes de Jones et W, nous commençons avec un arbre complet ayant exactement N feuilles. Cet arbre est de profondeur la plus égale possible. Avec l'algorithme M, chaque feuille est un ensemble de symboles. Nous pouvons initialiser l'arbre avec une seule feuille-ensemble de fréquence zéro qui contient tous les symboles de l'alphabet, et c'est la configuration que nous avons utilisé pour les tests aux tableaux 8.5 et 8.6. En n'utilisant qu'une seule feuille-ensemble au départ, cela revient à la tactique des algorithmes FGK et Λ où un symbole spécial est réservé pour introduire les symboles qui n'avaient pas été encore observés jusqu'alors. Le code spécial d'introduction de nouveaux symboles est simplement le code de la feuille qui contiendra tous les symboles qui n'ont pas été observés. Quand cette feuille disparaît parce que tous les symboles ont été observés au moins une fois, le code réservé qui y menait disparaît aussi, l'éliminant automatiquement lorsqu'il n'est plus nécessaire. On peut aussi considérer initialiser l'algorithme M avec deux feuilles, avec des fréquences précalculées, disons l'une de fréquence 1 avec les lettres de l'alphabet et quelques symboles de ponctuation et une autre feuille de fréquence 0 avec tous les autres symboles. On peut aussi penser à l'utilisation d'un arbre plus compliqué, précalculé à partir d'une distribution quelconque, comme à la fig. 8.8. L'utilisation d'un autre arbre initial ne nécessite pas forcément la transmission d'un grand nombre de bits au décodeur. S'il s'agit d'un arbre partagé par le codeur et le décodeur, nous n'avons pas à transmettre d'information du tout ; s'il s'agit d'un parmi n arbres possibles, nous n'avons qu'à transmettre $\lceil \lg n \rceil$ bits, ce qui est possiblement une quantité triviale d'information, surtout comparée à la longueur de la séquence compressée qui résultera de la compression.

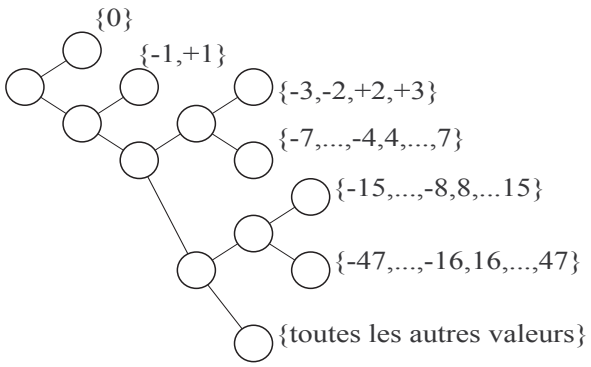


FIG. 8.8 – Une configuration alternative de l'arbre de départ. Cette figure est adaptée de la fig. 6. de [162].

8.4.1 Algorithme M dans le détail

L'algorithme M utilise donc l'opération de migration pour déplacer un symbole d'une feuille-ensemble à une autre afin de s'assurer que les symboles de fréquences égales reçoivent des codes de la longueur la plus égale possible. Dans cette section, nous allons présenter des détails d'implémentation ainsi que les algorithmes de migration et de *splaying*.

La fig. 8.9 décrit le contenu d'un nœud utilisé dans l'arbre de l'algorithme M. Les ensembles sont représentés par un `cSet`. Le `cSet` est une superclasse qui est dérivée pour obtenir la meilleure représentation selon la taille de l'alphabet à coder. Lorsque l'alphabet est de taille inférieure ou égale à 256 symboles, on instancie un `cByteSet` qui utilise un vecteur de bits pour représenter les ensembles, selon la méthode que nous exposerons à la section 8.4.2. Si nous avons plus de 256 symboles possibles, on instancie un `cIntSet`, qui utilise un arbre coagulant pour représenter les ensembles. Les arbres coagulants sont aussi présentés à la section 8.4.2. Cette tactique de détermination automatique de la classe à utiliser correspond au *design pattern* de type *factory*, où une superclasse décide au dernier instant possible le quel de ses descendants instancier, possiblement en utilisant une tierce classe [78]. Outre pour l'ensemble, qui n'est d'ailleurs instancié que pour les feuilles, ces nœuds ne diffèrent des nœuds utilisés dans l'algorithme W que par la présence du champ `poids` qui maintient la somme pondérée des feuilles qui descendent d'un nœud.

Pour la migration, il suffit de trouver l'ensemble de fréquence $f+1$ où sera transféré un symbole x de fréquence f. La feuille-ensemble qui contient x est facilement trouvée grâce à un tableau `to_leaf[ ]` qui utilise directement le symbole comme clef. La feuille-ensemble destination, de fréquence $f+1$ est trouvée grâce à une mémoire associative, ici, un arbre AVL, qui associe fréquences et feuilles. La fig. 8.10 montre les structures d'adressage `to_leaf[ ]` et `hash` qui servent à trouver efficacement les feuilles-ensembles source et destination. Si la feuille-ensemble de fréquence $f+1$ n'est pas trouvée (car elle peut ne pas exister), elle est créée dans l'arbre comme la sœur de la feuille-ensemble de fréquence f. Une fois les deux feuilles identifiées, le transfert de x d'une feuille-ensemble à l'autre ne consiste qu'à retirer x l'ensemble de la feuille originale pour l'ajouter à celui de la feuille de destination et à mettre à jour les poids

```
class cMcodecNode
       {
        private:

        public:  int poids;
                 int freq;

                 cSet *ensemble;
                 cMcodecNode *parent, *gauche, *droite;

        cMcodecNode();
        ~cMcodecNode();
       };
```

FIG. 8.9 – La définition de la classe `cMcodecNode`, utilisée par l'algorithme M.

respectifs des feuilles. Si les deux feuilles sont sœurs, l'algorithme de *splaying* mettra à jour correctement les poids lorsqu'on le lancera sur la feuille de destination. Sinon, il faut propager le changement de poids de la feuille source par la méthode `reweight( )` avant le *splaying* qui ne fait que propager les changements de poids, sans appliquer de transformations sur l'arbre, jusqu'à la racine. Une fois le transfert complété on lance le *splaying* sur la feuille destination. La méthode de *splaying* est identique à la méthode utilisée pour l'algorithme W à la différence qu'on propage les changements sur les poids plutôt que sur les fréquences (c'est à dire qu'on remplace `freq` par `poids` dans la fig. 8.7). L'algorithme de migration est donné à la fig. 8.11.

Pour obtenir le code pour un symbole, on procède de façon semblable aux autres algorithmes. Le préfixe est donné par le chemin entre la racine et la feuille qui contient le symbole. Le suffixe est donné par le code *phase-in* de l'index du symbole dans l'ensemble considéré. Si l'ensemble ne contient qu'un seul élément le suffixe requiert zéro bits et zéro bits seront émis. Si l'ensemble contient s symboles et si c'est le i^{e} symbole que l'on doit émettre, alors on émet le code $C_{\phi(s)}(i)$ pour ce symbole (voir section 5.3.4.2). Les codes *phase-in* sont nécessaires puisqu'on simule la présence de sous-arbres de profondeurs la plus égale possible par les ensembles attachés aux feuilles. Avoir un sous-arbre de profondeur la plus égale possible n'implique pas que les chemins soient tous de longueur égale ; souvenons-nous qu'un arbre *complete*[1] ne peut avoir des feuilles que sur au plus deux profondeurs différentes. Le décodage procèdera de la même façon, à l'inverse : on lit les bits jusqu'à ce que nous arrivions à une feuille, cette feuille nous dit combien de bits il faut lire pour obtenir le suffixe.

8.4.2 Complexité, implémentation, vitesse et performance

La vitesse de l'algorithme M est moindre mais comparable à la vitesse de l'algorithme W. La complexité est essentiellement la même mais la vitesse de l'implémentation repose entièrement sur le choix des méthodes utilisées pour représenter les feuilles-ensembles et pour retrouver la feuille qui correspond à une fréquence donnée. Comme il est probable qu'il n'y ait aucun rapport évident entre la valeur numérique du symbole et sa fréquence, il est à parier que les symboles qui occupent une feuille soient quelconques et que nous ayons affaire à un ensemble ténu (*sparse set*).

1 Soit un arbre dont chaque nœud est soit une feuille ou a exactement deux fils et dont toutes les feuilles sont sur au plus deux niveaux différents.

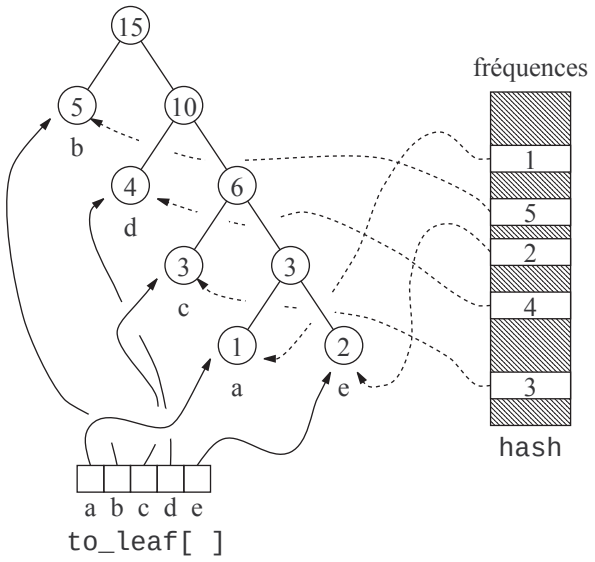


FIG. 8.10 – Les principales structures de données de l'algorithme M.

Si on a affaire à un petit nombre de symbole, disons des octets, il est probable que la meilleure façon de procéder soit d'utiliser des tableaux de valeurs booléennes. L'insertion et le retrait d'éléments se font en temps constant grâce à des opérations très simples. Si on veut sauver un peu de mémoire, on peut utiliser des vecteurs de bits, où le bit x est à 1 si x est élément de l'ensemble. Ici aussi, les opérations d'ajout et de retrait se font en temps constant et ne requièrent que des opérations élémentaires de la machine. Les opérations qui consistent à trouver le i^{e} élément ou à trouver l'index d'un élément se font en temps linéaire en la taille de l'alphabet, mais la constante multiplicative peut être réduite par un facteur 8 ou plus trivialement. Bien que nous ayons un vecteur de bits, nous ne sommes pas obligé de les compter un à un jusqu'à ce que nous arrivions au i^{e} élément. Nous pouvons fort bien compter le nombre de bits sur les octets qui précèdent l'octet qui contient le i^{e} bit, ce qui ne demande qu'une petite table qui maintient le nombre de bits à 1 pour chaque octet possible. C'est la méthode que nous avons choisi pour les `cByteSet`.

La situation se complique considérablement lorsque nous avons à gérer un nombre potentiellement très grand de symboles distincts. Si on peut se permettre d'utiliser beaucoup de mémoire pour représenter les ensembles, on peut opter pour des implémentations efficaces mais brutales d'arbres de recherche tels que les arbres AVL ou encore à des tables à adressage dispersé. Les arbres AVL tentent de maintenir un arbre de recherche de profondeur la plus égale possible [3, 196]. Les arbres AVL sont potentiellement adéquats dans ce cas puisque tous les symboles contenus dans un ensemble ont une même fréquence. Si nous sommes concernés par l'utilisation de la mémoire, il faut recourir à des représentations plus astucieuses pour les ensembles.

Dans le *Doctor Dobb's Journal* [164], nous avons présenté une implémentation complète de l'algorithme M qui utilise une simple liste chaînée de pages pour représenter les ensembles. Chaque page contient un nombre fixe de cases qui contiennent soit des éléments seuls ou des intervalles (on peut représenter les intervalles contigus en ne représentant que les extremums)

```
void cMcodec::update(int x)
 {
  cMcodecNode *feuille = to_leaf[x]; // pointeur à la feuille qui contient x
  cMcodecNode *nouvelle_feuille = hash.getContents(feuille->freq + 1);

  if ((nouvelle_feuille==0) && (feuille->ensemble->cardinal()==1))
       {
        // migration paresseuse...

        hash.remove(feuille->freq);
        feuille->freq++;
        hash.add(feuille->freq,feuille);

        feuille->poids = feuille->freq;

        splay(feuille);
       } // migration rapide
  else {
        feuille->ensemble->remove(x);
        feuille->poids -= feuille->freq;

        if (nouvelle_feuille==0)
         {
          // ajoute comme soeur de l'autre
          nouvelle_feuille = ajouterFeuille(feuille);
          nouvelle_feuille->freq = feuille->freq+1;

          if (alphabet_size<=256)
               nouvelle_feuille->ensemble = new cByteSet(false);
          else nouvelle_feuille->ensemble = new cWordSet(false);

          hash.add(nouvelle_feuille->freq, nouvelle_feuille);

          if (root==feuille) root=feuille->parent;
         }
        else reweight(feuille); // progage les changements de poids

        nouvelle_feuille->ensemble->add(x);
        nouvelle_feuille->poids += nouvelle_feuille->freq;

        to_leaf[x]=nouvelle_feuille;

        splay(nouvelle_feuille);

        if (feuille->ensemble->cardinal()==0)
            {
             hash.remove(feuille->freq);
             deleterFeuille(feuille);
             feuille=0;
            }
        } // migration normale
 }
```

FIG. 8.11 – L'algorithme de migration dans M.

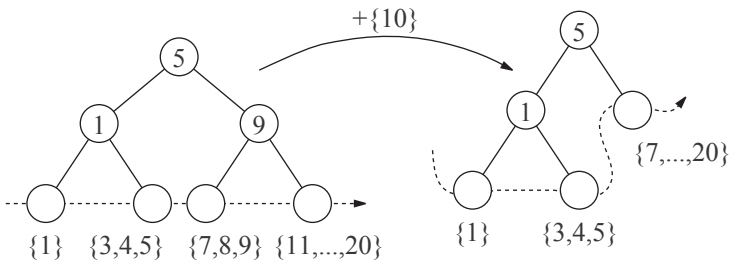


FIG. 8.12 – L'ajout de l'élément 10 dans un arbre coagulant. La flèche pointillée représente la liste chaînée entre les feuilles de l'arbre. Le nombre inscrit dans les nœuds internes représente la plus grande valeur à gauche du nœud.

maintenus en ordre croissant. Cette méthode mène à un temps de recherche en pire cas de $O(n/k + \lg k)$ pour n éléments et des pages de k éléments. Les temps d'ajout et de retrait d'un élément sont $O(n/k + k)$. Ces performances ne sont acceptables que lorsque n est petit et k modéré. C'est heureusement le cas pour n, du moins si nous tirons nos conclusions à partir des tests réalisés sur les corpus de Calgary et Canterbury lorsque ces séquences sont considérés en tant que suite d'octets. Sur les fichiers du corpus et avec $k = 16$, la plupart des ensembles ne requièrent qu'une seule page. Le temps pour les diverses opérations dégénère donc en $O(\lg k)$ pour la recherche et en $O(k)$ pour les ajouts, ce qui tout à fait acceptable.

Cette solution n'est pas particulièrement safisaisante, car lorsque les séquences sont considérées en tant que suite de mots de 16 bits on se retrouve avec plus d'une page. Après avoir cherché des représentations alternatives qui soient à la fois économes en mémoire et efficaces en temps de calcul, nous somme arrivés à concevoir une solution apparemment inédite pour la représentation des ensembles : les *arbres coagulants*. Les arbres de recherche communs ne maintiennent qu'une seule clef par nœud (feuille ou nœud interne). Cela requiert un nombre de nœuds égal au nombre d'éléments qui y sont maintenus, et pour de grands ensembles, cela mène à un gaspillage éhonté de la mémoire. Chaque nœud utilise des pointeurs vers le parent et les deux fils (sans compter les autres informations nécessaires au balancement de l'arbre) ce qui fait que pour un symbole qui tient sur deux octets, on fini par en utiliser 16. Les arbres coagulants sont un compromis entre la liste chaînée de pages et les arbres de recherche non contraints. Ils permettent une recherche en $O(\lg n)$. Comme les arbres AVL, les arbres coagulants maintiennent dans chaque nœud la plus grande profondeur du sous-arbre dominé par ce nœud. La clef dans chaque nœud interne représente la plus grande clef qui se trouve à gauche du nœud, selon la méthode de visite en profondeur (*depth first search*). À la différence des arbres de recherche classique, les feuilles des arbres coagulants sont soit des nombres seuls soit des intervalles continus. Les intervalles sont représentés par leur extrémums.

Cette structure permet de déterminer efficacement l'appartenance d'un élément, de procéder à l'extraction du i^{e} élément ainsi que la détermination de i pour un élément. Ce qui est nouveau de cette structure, c'est la possibilité qu'ont les feuilles de se fusionner. Précisons que toutes les feuilles sont chaînées entre elles en ordre numérique croissant des éléments qu'elles contiennent. Suivre cette chaîne nous permet d'énumérer tous les éléments sans fouiller l'arbre. Considérez la fig. 8.12. Lorsque nous ajoutons un élément (dans la figure, on ajoute 10), on

vérifie si deux feuilles voisines dans la liste chaînée ne formeraient plus qu'un intervalle continu en tenant compte du nouvel élément. Si c'est le cas, on fusionne les intervalles et on élimine une feuille de l'arbre. Après l'élimination de la feuille, on procède au rebalancement AVL. Le test de coagulation se fait en temps constant. Le rebalancement AVL coûte $O(\lg n)$ opérations. L'ajout et le retrait d'éléments coûtent donc eux aussi $O(\lg n)$ opérations. L'arbre coagulant est une très bonne structure pour maintenir les ensembles utilisés dans l'algorithme M. Dans la version simplifiée de l'algorithme M, celle qui est présentée dans ce chapitre, nous avons réimplémenté les ensembles `cIntSet` par cette méthode.

Les résultats numériques obtenus par l'algorithme M (voir la section 8.4.4) semblent indiquer que les longueurs de codes produits sont très près de l'entropie de la source, soit que la longueur des codes est $\approx -\lg P(X = x)$. En fait, nous montrerons à la section 8.4.3 que la longueur du code pour x est effectivement $O(-\lg P(X = x))$. Cela nous donne, comme pour les autres algorithmes présentés une complexité en temps de $O(\mathcal{H}(X))$. L'algorithme des arbres coagulants donne des temps de $O(\lg n)$ pour les opérations de retraits, d'ajouts et de calcul d'index pour un ensemble de n éléments (ce qui est consistant avec le fait que les symboles d'un même ensemble soient équiprobables), et les longueurs des préfixes donnés par l'algorithme M convergent vers $O(-\lg P(X \in S_f))$, où S_f est l'ensemble des symboles de fréquence f. Puisque la complexité de maintenir l'ensemble S_f est proportionnel à $O(\lg |S_f|)$, nous avons une complexité combinée de $O(-\lg P(X = x))$ pour la mise à jour d'un symbole x. En effet, si nous assumons que chaque symbole dans une feuille est conditionnellement équiprobable, nous trouvons que $-\lg P(X = x \mid x \in S_f) = \lg |S_f|$, et que $-\lg P(X \in S_f) - \lg P(X = x \mid x \in S_f) = -\lg P(X = x)$. Enfin, puisque chaque symbole est observé avec une probabilité $P(X = x)$, nous trouvons que la complexité est $O(\mathcal{H}(X))$.

8.4.3 Bornes sur la longueur des codes

Qu'en est-il des longueurs des codes générés ? Posons $\bar{L}_M(X)$, la longueur moyenne des codes générés par l'algorithme M pour la source X. Avec l'algorithme M, nous décomposons le code pour x en préfixe et suffixe. Nous pouvons exprimer la longueur du code pour x de la façon suivante :

$$L_M(x) = L_M(S_x) + L_{\phi(|S_x|)}(I(x, S_x))$$

où $L_{\phi(n)}(i)$ est le code *phase-in* de $0 \leqslant i < n$, S_x est l'ensemble qui contient x, et $I(x, S_x)$ est l'index de x dans S_x, c'est-à-dire que x est le $I(x, S_x)^{\text{e}}$ élément de S_x. $L_M(S_x)$ est la longueur du code qui mène à la feuille-ensemble S_x.

Pour ce qui est du suffixe, nous avons montré que l'utilisation d'un code *phase-in*, sous l'hypothèse de tirage uniforme, mène à une perte d'au plus ≈ 0.0860713 bits. Nous avons démontré cet énoncé à la section 5.3.4.2, page 103. Nous avons

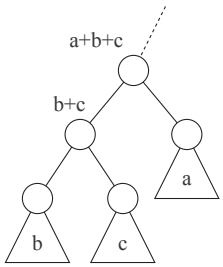
$$\bar{L}_{\phi(|S_x|)}(I(\,\cdot\,, S_x)) \leqslant \lg |S_x| + 0.0860713 \tag{8.1}$$

Nous allons montrer que les préfixes respectent

$$-\lg P(S = S_x) \leqslant L_M(S_x) \leqslant -2 \lg P(S = S_x)$$

où S est la source aléatoire pour les différents S_x. Nous démontrerons ce résultat en deux parties. La première partie consistera à montrer combien de bits au plus sont gaspillés dans la région

interne de l'arbre, c'est-à-dire entre la racine et le dernier nœud interne avant la feuille. La seconde partie consistera à montrer combien de bits sont perdus dans le cas où nous avons affaire à la fin du préfixe, soit à une paire de feuilles. Considérez la situation suivante, à un endroit quelconque de l'arbre :



où a, b et c représentent les poids des sous-arbres adjoints. Par simplicité, supposons que nous avons a,b et c tels que $a+b+c=1$, puisqu'il s'agit de probabilités. À tout endroit de l'arbre on a l'invariant $a \geqslant \max(b,c)$. Nous avons obligatoirement $b \leqslant a$ et $c \leqslant a$, autrement il y aurait eu *splaying*. En respectant la contrainte de l'invariant, la plus grande différence entre a et $b+c$ est trouvée lorque $b+c=\varepsilon$ et $a=1-\varepsilon$, pour $\varepsilon > 0$. Autrement dit, on peut faire en sorte que a soit essentiellement 1 et $b+c$ essentiellement zéro. Puisque nous allouons un bit complet pour l'embranchement entre a et $b+c$, le nombre de bits gaspillé est de

$$1 - (-a \lg a - (b+c) \lg(b+c))$$

et

$$\lim_{\varepsilon \to 0} 1 - (-(1-\varepsilon) \lg(1-\varepsilon) - \varepsilon \lg \varepsilon) = 1$$

d'où on déduit que l'on gaspille au plus 1 bit par embranchement. Cela termine la partie de la preuve quant à la longueur des préfixes.

Combinant la longueur du préfixe avec la longueur du suffixe, l'éq. (8.1), nous trouvons enfin

$$\begin{aligned} -\lg P(X=x) &\leqslant L_M(x) \\ &= L_M(S_x) + L_{\phi(|S_x|)}(I(x,S_x)) \\ &\leqslant -2 \lg P(S=S_x) + \lg |S_x| + \gamma \end{aligned}$$

où $\gamma = 0.0860713$. En pondérant par les probabilités d'occurence, on trouve

$$\begin{aligned} -\sum\nolimits_x P(X=x) \lg P(X=x) &\leqslant \sum\nolimits_x P(X=x) L_M(x) \\ &\leqslant \sum\nolimits_x P(X=x) \left(-2 \lg P(S=S_x) + \lg |S_x| + \gamma\right) \end{aligned}$$

c'est-à-dire

$$\mathcal{H}(X) \leqslant \bar{L}_M(x) \leqslant \gamma + \sum\nolimits_x P(X=x) \left(-2 \lg P(S=S_x) + \lg |S_x|\right)$$

d'où on découle

$$
\begin{aligned}
\bar{L}_M(X) &\leqslant \gamma + \sum\nolimits_x P(X=x)\,(-2\lg P(S=S_x) + \lg|S_x|) \\
&\leqslant \gamma + \sum\nolimits_x P(X=x)(-2\lg P(S=S_x) - \lg P(X=x \mid S=S_x)) \\
&\leqslant \gamma - \sum\nolimits_x P(X=x \mid S=S_x)P(S=S_x)(2\lg P(S=S_x) + \lg P(X=x \mid S=S_x)) \\
&\leqslant \gamma - \sum\nolimits_x P(X=x \mid S=S_x)P(S=S_x)(\lg(P(X=x \mid S=S_x)P(S=S_x)) + \lg P(S=S_x)) \\
&\leqslant \gamma - \sum\nolimits_x P(X=x \mid S=S_x)P(S=S_x)\lg(P(X=x \mid S=S_x)P(S=S_x)) + \\
&\qquad - \sum\nolimits_x P(X=x \mid S=S_x)\lg P(S=S_x)) \\
&\leqslant \gamma + \mathcal{H}(X) - \sum\nolimits_x P(X=x \mid S=S_x)\lg P(S=S_x) \\
&\leqslant \gamma + \mathcal{H}(X) + \mathcal{H}(S) \\
&\leqslant \gamma + 2\mathcal{H}(X)
\end{aligned}
$$

ce qui nous donne que la longueur moyenne des codes générés par l'algorithme M est bornée par

$$\mathcal{H}(X) \leqslant \bar{L}_M(X) \leqslant 2\mathcal{H}(X) + \gamma$$

Cette démonstration de la performance de l'algorithme M s'applique aussi à l'algorithme W dans la mesure où l'invariant que nous avons utilisé pour M existe aussi pour l'algorithme W. Avec l'algorithme W les codes ne comportent pas de suffixes *phase-in*, ce qui fait que la borne est véritablement $2\mathcal{H}(X)$ plutôt que $\mathcal{H}(X)+\mathcal{S}+\gamma$. Or comme $\mathcal{H}(X)+\mathcal{S}+\gamma \leqslant 2\mathcal{H}(X)$, les deux algorithmes sont essentiellement équivalents.

Par comparaison, Vitter montre que pour l'algorithme FGK, la longueur de la séquence compressée est d'au plus $2H+t$, où H est le coût de codage de la séquence grâce à la méthode de Huffman statique et t la longueur de la séquence [216]. Pour l'algorithme Λ, Vitter montre que la longueur de la séquence compressée par l'algorithme Λ n'est jamais plus longue que $H+t$, ce qui revient approximativement à dire que les codes individuels sont tout au plus un bit plus long que l'optimal. Nous obtenons une longueur maximale plus élevée pour les algorithmes M et W, soit $\approx t(2\mathcal{H}(X)+\gamma)$.

8.4.4 Résultats

Nous trouvons aux tableaux 8.5 et 8.6 les résultats obtenus par l'algorithme M. L'algorithme M se situe entre l'algorithme Λ et l'algorithme W en performance. La configuration initiale de l'arbre consistait d'une feuille-ensemble unique contenant tous les symboles.

On voit que le partage de préfixes permet de gagner 0.04 bits par rapport à l'algorithme W sur le corpus de Calgary, et 0.05 sur le corpus de Canterbury. En valeurs relatives, cela ne compte que pour une amélioration d'environ 1 à 2%. On ne pouvait vraiment s'attendre à une plus grande amélioration car l'algorithme W donne déjà des résultats de l'ordre de 0.1 bits plus grands que l'algorithme Λ. L'algorithme M réduit quand même l'écart par un facteur 2, ce qui n'est pas négligeable. L'augmentation de complexité de l'algorithme M (en termes d'implémentation) ne vaut peut-être pas le gain en compression obtenu. L'algorithme W est en effet

Corpus de Calgary					
Fichier	Jones	W	M	Λ	Huffman
bib	6.26	5.33	5.41	5.00	5.23
book1	5.50	4.71	4.66	4.56	4.56
book2	5.56	4.98	4.93	4.82	4.82
geo	6.61	5.76	5.83	5.69	5.67
news	5.88	5.40	5.36	5.23	5.22
obj1	6.04	6.19	6.18	6.07	5.97
obj2	6.60	6.46	6.36	6.30	6.29
paper1	5.73	5.13	5.14	5.04	5.01
paper2	5.55	4.78	4.68	4.65	4.63
paper3	5.59	4.87	4.81	4.71	4.69
paper4	5.58	4.98	4.87	4.80	4.73
paper5	5.72	5.15	5.11	5.05	4.97
paper6	5.59	5.17	5.12	5.07	5.04
pic	1.71	1.68	1.67	1.66	1.66
progc	5.92	5.36	5.37	5.26	5.23
progl	5.19	4.90	4.90	4.81	4.80
progp	5.39	5.09	5.07	4.92	4.90
trans	5.81	5.69	5.65	5.58	5.57
moyenne	5.11	4.62	4.58	4.49	4.49

TAB. 8.5 – Les résultats de l'algorithme M sur le corpus de Calgary, comparés aux résultats obtenus par les autres algorithmes.

beaucoup plus simple et beaucoup plus rapide que les algorithmes M et Λ. Cependant, il nous reste à considérer le cas où l'alphabet contient un très grand nombre de symboles possibles. En effet, les résultats des tableaux 8.5 et 8.6 montrent les résultats obtenus en considérant les fichiers comme des séries d'octets. Nous verrons à la section 8.5 que l'algorithme M est mieux conçu pour les très grands alphabets que les algorithmes de Jones, W et Λ en ce qui a trait au nombre de nœuds créés. Puisque l'algorithme M, comme l'algorithme de Vitter, ne crée les nœuds que sur demande seulement, on pourra gagner beaucoup sur la mémoire utilisée, en autant que nous ayons une représentation des ensembles qui ne requiert que peu de mémoire.

Enfin, le tableau 8.7 montre les résultats cumulatifs obtenus par les différents algorithmes. À la section 8.5, nous présenterons les résultats obtenus avec les fichiers considérés comme des séquences de bigrammes, soit des symboles sur 16 bits. Nous verrons que les feuilles-ensembles de l'algorithme M permettent de créer beaucoup moins de feuilles que les autres algorithmes. Cela peut avoir un avantage quant à la quantité de mémoire utilisée pour maintenir l'arbre.

8.5 Algorithmes adaptatifs sur de très grands alphabets

Les algorithmes de Jones, W et Λ créent beaucoup de nœuds pour maintenir les symboles. Les algoritmes de Jones et W créent l'arbre complet au moment de l'initialisation, ce qui les rends difficiles à utiliser lorsque nous considérons un grand alphabet, par exemple sur 16 bits. Les algorithmes M, FGK et Λ utilisent un code spécial pour introduire de nouveaux symboles

Corpus de Canterbury					
Fichier	Jones	W	M	Λ	Huffman
alice29.txt	5.12	4.82	4.70	4.62	4.62
asyoulik.txt	5.75	4.96	5.01	4.85	4.85
bible.txt	5.30	4.50	4.42	4.39	4.35
cp.html	6.18	5.40	5.43	5.31	5.27
ecoli	2.42	2.24	2.25	2.25	2.00
fields.c	5.65	5.25	5.22	5.13	5.04
grammar.lsp	5.38	4.91	4.93	4.87	4.66
kennedy.xls	3.89	3.66	3.62	3.60	3.59
kjv.gutenberg	5.19	4.68	4.58	4.42	4.42
lcet10.txt	5.43	4.85	4.84	4.70	4.70
plrabn12.txt	5.58	4.67	4.78	4.58	4.58
ptt5	1.71	1.68	1.67	1.66	1.66
sum	4.93	5.58	5.54	5.42	5.37
world192.txt	5.77	5.19	5.13	5.04	5.04
xargs.1	5.75	5.17	5.15	5.10	4.92
moyenne	4.46	3.97	3.92	3.86	3.78

TAB. 8.6 – Les résultats de l'algorithme M sur le corpus de Canterbury, comparés aux résultats obtenus par les autres algorithmes.

Algorithme	Calgary	Canterbury
Jones	5.11	4.46
W	4.62	3.97
M	4.58	3.92
Λ	4.49	3.86
Huffman	4.49	3.78

TAB. 8.7 – Comparaisons des algorithmes, sur les octets, pourles deux corpus.

et peuvent être initialisés avec un arbre vide. Cette stratégie permet de ne créer que les feuilles nécessaires, réduisant ainsi grandement la taille de l'arbre, car, même si l'alphabet contient un très grand nombre de symboles, il se peut que seule une relativement petite partie de ces symboles soient observée dans une séquence.

L'algorithme M réduit le nombre de feuilles en utilisant des feuilles-ensembles plutôt que des feuilles qui ne contiennent qu'un symbole. Les tableaux 8.8 et 8.9 montrent les longueurs de codes obtenues et le nombre de feuilles créées. Les scores sont normalisés de façon à pouvoir comparer directement les résultats obtenus par les algorithmes avec les symboles à 16 bits aux résultats des algorithmes avec les symboles sur des octets. Les résultats devraient être multipliés par deux pour représenter les scores réels obtenus pour 16 bits.

Utiliser des mots de 16 bits plutôt que des octets revient à compresser la séquence en bigrammes, mais où chaque code représente obligatoirement un bigramme, contrairement à l'algorithme que nous avons présenté à la section 5.3.2.4. Supposer que les paires d'octets sont fortement correlés est parfaitement raisonnable pour de nombreuses classes de séquences comme les textes et les exécutables. La dépendance des octets paire à paire est moins claire pour d'autre types de séquences, en particulier les fichiers d'images, sur lesquels nous reviendrons dans des chapitres ultérieurs.

De plus, avec la mondialisation, nous voyons émerger de nouveaux standards pour représenter des alphabets étendus. En particulier, le standard absolu en la matière est le système Unicode, proposé par le Consortium Unicode, un organisme à but non lucratif. Le système Unicode permet de représenter environ 2^{32} symboles, tirés de tous les alphabets connus.

Le Consortium propose un algorithme de compression pour les séquences codées au standard Unicode [224]. Cet algorithme est une variation de la méthode de Karlgren (voir section 5.3.2.5) où les symboles sont encodés sur des octets. Les octets de 0 à 127 (hormis quelques codes de contrôle) sont assignés aux codes ASCII de base, et les codes 128 à 255 sont choisis selon une palette courante de 128 symboles contingus dans Unicode. Les codes de contrôle sont réservés pour changer la palette courante, contrôler le changement de ligne ou introduire un seul caractère qui ne fait pas partie de la palette courante. Les caractères introduits de cette façon sont représentés grâce à un code de longueur variable alignés sur des octets. La structure du code n'est pas sans rappeler les codes de Pike (section 5.3.2.7) ou les codes *Start/Stop* contraints aux octets (section 7.5.3.3). Cette méthode de compression n'est pas vraiment une méthode de compression car il est fort à parier que, pour un texte écrit avec l'alphabet latin par exemple, la majorité des symboles non ASCII retrouvés se retrouvent tous dans la même palette de 128 symboles, hormis peut-être quelques symboles typographiques. Cette méthode revient donc à utiliser des octets pour tous les symboles, ce qui ne donne aucune compression.

Les codes de Huffman adaptatifs feraient mieux, peut-être, que cette méthode rudimentaire de compression. Les algorithmes qui sont modifiés pour représenter un très grand nombre de symboles distincts, comme l'algorithme M ou encore l'algorithme Λ permettraient probablement d'obtenir de meilleurs résultats sans payer le prix d'un arbre complet en mémoire — rappelons-nous que l'alphabet Unicode contient environ 2^{32} symboles représentables ! La gestion explicite des palettes de symboles étendus deviendrait inutile avec un algorithme qui donne des codes de longueur quelconques, mais optimisés en fonction de la distribution des symboles. L'algorithme M crée un nombre réduit de feuilles, ce qui peut permettre, à condition d'avoir une représentation

Corpus de Calgary										
		Jones		W		M		Λ		
Fichier	$\|\Sigma\|$	Bits	Feuilles	Bits	Feuilles	Bits	Feuilles	Bits	Feuilles	Huffman
bib	1324	5.26		4.60		4.59	213	4.49		4.29
book1	1633	4.91		4.21		**4.10**	438	4.11		4.07
book2	2739	4.97		4.45		4.43	419	4.36		4.28
geo	2042	5.56		5.03		5.03	142	4.94		4.64
news	3686	5.37		4.91		4.87	358	4.81		4.65
obj1	3064	5.49		6.97		**6.09**	50	6.79		4.58
obj2	6170	4.42		5.00		**4.85**	243	4.87		4.46
paper1	1354	5.12		4.85		4.76	142	4.74		4.32
paper2	1121	4.93		4.39		4.35	186	4.29		4.06
paper3	1011	5.05	2^{16}	4.59	2^{16}	4.52	142	4.47	$\|\Sigma\|+1$	4.11
paper4	705	5.10		5.04		**4.87**	67	4.92		4.07
paper5	812	5.27		5.44		**5.19**	58	5.30		4.21
paper6	1219	5.04		4.98		**4.82**	117	4.83		4.30
pic	2321	1.27		1.28		**1.26**	138	1.27		1.19
progc	1443	5.15		5.11		**4.98**	112	4.99		4.40
progl	1032	4.46		4.32		4.29	159	4.24		4.00
progp	1255	4.62		4.56		4.46	113	4.45		4.02
trans	1791	4.93		4.88		4.80	175	4.77		4.45
moyenne		4.39		4.07		4.02		3.98		3.80

TAB. 8.8 – Les résultats de l'algorithme M sur le corpus de Calgary, comparés aux résultats obtenus par les autres algorithmes. La colonne $|\Sigma|$ donne le nombre de symboles différents observés dans les fichiers. Les longueurs de codes en bits sont normalisées par rapport aux octets, les vraies longueurs de codes pour les symboles sur 16 bits ont été divisées par deux. L'algorithme Λ génère toujours $|\Sigma|+1$ feuilles, puisque nous avons toujours le code réservé à l'introduction de symboles qui n'ont pas encore été observés. Les nombres en gras soulignent les endroits où l'algorithme M obtient de meilleurs résultatas que l'algorithme Λ.

compacte des ensembles, de sauver de la mémoire par rapport à l'algorithme Λ. En effet, on voit, en examinant les tableaux 8.8 et 8.9, que l'algorithme M produit parfois aussi peu que 10% des feuilles créées par l'algorithme Λ et obtient parfois de meilleurs taux de compression sur certains fichiers. Ces fichiers où M bat Λ sont montrés en gras aux tableaux.

8.6 Conclusion

Nous avons montré que l'algorithme de Jones est très simple et demande peu de ressources. Nous avons aussi montré que la compression qu'on obtient est moindre que celle obtenue grâce aux autres algorithmes. Nous avons montré à la section 8.3.2 que déjà l'algorithme W, qui est une modification de l'algorithme de Jones, donne des codes en moyenne un demi bit plus courts, sans augmentation de ressources et sans véritable complexification de l'algorithme. L'algorithme W s'approche d'environ 0.1 bits des longueurs des codes données par l'algorithme Λ. Cette différence représente une différence relative d'environ seulement 5%. Nous avons par la suite modifié l'algorithme W pour nous assurer que les symboles de même fréquences aient des codes de même longueur, ce qui nous permet de réduire par un facteur deux la différence entre la longueur des codes générés par l'algorithme W et l'algorithme Λ. L'Algorithme M donne des codes qui ne sont en moyenne que 0.05 bits plus long que ceux générés par l'algorithme Λ. L'arbre de code généré par l'algorithme M utilise des feuilles spéciales, qui ne représentent

Corpus de Calgary										
		Jones		W		M		Λ		
Fichier	$\|\Sigma\|$	Bits	Feuilles	Bits	Feuilles	Bits	Feuilles	Bits	Feuilles	Huffman
alice29.txt	1133	4.81		4.23		4.20	234	4.13		4.00
asyoulik.txt	1044	4.97		4.38		4.36	213	4.27		4.12
bible.txt	1121	4.53		3.91		3.89	586	3.82		3.82
cp.html	1193	5.40		5.25		**5.07**	91	5.11		4.33
ecoli	16	2.37		2.07		2.05	17	2.02		2.00
fields.c	645	4.86		5.03		**4.84**	58	4.90		3.96
grammar.lsp	355	4.77		5.34		**4.92**	32	5.19		3.66
kennedy.xls	1655	3.37	2^{16}	3.42	2^{16}	3.24	133	3.22	$\|\Sigma\|+1$	3.19
kjv.gutenberg	1607	4.50		3.95		3.89	679	3.84		3.84
lcet10.txt	1714	4.81		4.27		4.24	343	3.58		4.10
plrabn12.txt	1089	4.81		4.09		4.09	360	4.00		3.96
ptt5	2321	1.27		1.28		**1.26**	138	1.27		1.19
sum	2389	4.35		5.32		**4.95**	82	5.17		4.18
world192.txt	2779	5.11		4.48		4.45	724	4.36		4.35
xargs.1	443	5.36		5.82		**5.43**	31	5.67		4.00
moyenne		3.93		3.46		3.42		3.36		3.35

TAB. 8.9 – Les résultats de l'algorithme M sur le corpus de Canterbury, comparés aux résultats obtenus par les autres algorithmes. Ici aussi les scores ont été normalisés (voir la légende de la fig. 8.8).

plus des symboles individuels, mais des ensembles de symboles. Ces ensembles de symboles contiennent chacun des symboles qui ont la même fréquence d'observation, élagant ainsi l'arbre de façon importante. À la section 8.5, nous avons montré que l'algorithme M sur de très grands alphabets crée parfois aussi peu que 10% des feuilles créées par l'algorithme Λ, cela représente environ 500 fois moins de feuilles que les algorithmes de Jones et W.

Chapitre 9

LZW avec perte

9.1 Introduction

Dans ce chapitre, nous présentons des modifications à l'algorithme LZW appliquées à la compression d'image. En particulier, nous nous intéresserons aux images à base de palette où chaque pixel est représenté par un index dans une table de couleurs, la palette. Bien que nous ayons déjà décrit l'algorithme LZW à la section 4.2.1.2, nous reviendrons sur cet algorithme en présentant les structures de données nécessaires avec plus de détail. Nous verrons comment cet algorithme peut être utilisé pour la compression d'image, en particulier avec le protocole GIF. Nous ne décrirons pas en grand détail le protocole GIF, bien que nous nous en soyons servi pour établir nos résultats. Nous étudierons les effets de la réduction du nombre de couleurs dans les images ainsi que du *dithering* qui réduit les effets négatifs liés à cette réduction. Enfin, nous présenterons deux modifications à l'algorithme LZW qui permettent de faire des concordances approximatives — alors que l'algorithme original exige des concordances exactes — pour augmenter les taux de compression obtenus tout en minimisant les dommages sur l'image reconstruite. Nous présenterons un algorithme semblable tiré de la littérature aux fins de comparaisons. Nous verrons que nos deux algorithmes sont supérieurs, en termes de taux de compression comme de qualité d'image, à l'algorithme de référence.

9.2 LZW : L'algorithme de Welch

Nous avons brièvement décrit l'algorithme de Welch à la section 4.2.1.2 sans rentrer dans les détails. Dans cette section, nous revenons sur l'algorithme et présentons les structures de données nécessaires. Nous décrirons la structure de donnée qui sera utilisée pour le maintient du dictionnaire. Cette structure, c'est la treille [74].

L'algorithme de Welch est une variation des algorithmes de la classe LZ78 qui utilisent un dictionnaire pour maintenir leurs contextes plutôt qu'une fenêtre coulissante sur la séquence [221]. Avec les algorithmes de la classe LZ78, la séquence à compresser est découpée incrémentalement en sous-séquences qui sont emmagasinées dans le dictionnaire. Ces algorithmes diffèrent entre eux par la façon dont la séquence est découpée et par le nombre de sous-séquences accessibles dans le dictionnaire. L'algorithme LZW permet d'avoir accès à toutes les chaînes dans le dictionnaire ainsi qu'à tous leurs préfixes — ce qui est différent d'avoir accès à toutes les

sous-chaînes possibles.

Certains algorithmes de la classe LZ78 utilisent, après avoir trouvé la plus longue concordance D_i dans le dictionnaire, des tuples de la forme $\langle i, \sigma \rangle$, où i est l'index de la concordance dans le dictionnaire D et σ le premier symbole à ne pas concorder entre la séquence à compresser et les sous-séquences du dictionnaire. Ces algorithmes mettent à jour leur dictionnaire en y ajoutant la séquence $(D_i : \sigma)$. L'algorithme de Welch utilise une astuce qui nous dispense de l'utilisation de ces tuples. L'algorithme de Welch n'utilise en effet que les index, car il n'est pas nécessaire de connaître immédiatement σ pour encoder ou décoder la prochaîne sous-séquence.

Voici comment procède l'algorithme de Welch. Supposons que nous parcourrions la séquence à compresser à la recherche de la plus longue concordance avec l'une des entrées du dictionnaire. Après un certain nombre de symboles, disons que nous trouvions une concordance (de longueur quelconque) dont l'index est i. Plutôt que d'émettre i et le symbole fautif σ encodés en tuple, nous allons plutôt émettre i et reprendre la recherche de concordance en commençant à la position de σ dans la séquence originale. Après un certain autre nombre de symboles concordants, nous trouvons que la plus longue concordance se trouve à l'index j dans le dictionnaire. Nous savons que l'entrée D_j commence par σ, car c'est le premier symbole de la concordance, mais c'est aussi le symbole qui a mis fin à la concordance avec D_i. Puisque nous connaissons D_j, nous connaissons σ et nous pouvons déduire $(D_i : \sigma)$. La nouvelle séquence $(D_i : \sigma)$ est ajoutée au dictionnaire après l'émission de j.

Au départ, le dictionnaire est initialisé avec $|\Sigma|$ entrées, chacune étant l'un des symboles $\sigma \in \Sigma$, pour un alphabet de séquence Σ. Cette méthode nous permet aussi de nous dispenser de codes spéciaux réservés pour l'introduction de nouveaux symboles. Quand le dictionnaire est rempli, il peut être remis à zéro ou encore on cesse d'ajouter les nouvelles sous-séquences. Certains algorithmes dérivés de LZW utilisent des techniques plus sophistiquées pour décider quand réinitialiser le dictionnaire en surveillant, par exemple, le taux de compression.

Dans son article, Welch proprose d'utiliser des codes de longueur fixe pour les index [221, p. 14]. Welch préconise l'utilisation d'un dictionnaire de 4096 entrées, nécessitant ainsi 12 bits par code ; en général une implémentation ayant un dictionnaire de taille N demandera $\lceil \lg N \rceil$ bits par index.

9.2.1 La treille comme dictionnaire

Welch propose aussi d'utiliser une table à adressage dispersé pour maintenir le dictionnaire. Nous nous proposons plutôt d'utiliser une treille, une structure de données qui permet de faire des recherches de concordance en temps linéaire en la longueur de la plus longue concordance et des ajouts en temps constant. Par comparaison, la table à adressage dispersée demandera un temps linéaire en la longueur de la séquence ajoutée. La table à adressage dispersée demandera aussi une table qui a un nombre d'entrées beaucoup plus grand que le nombre d'entrées dans le dictionnaire pour assurer les accès en temps constant (modulo le temps linéaire nécessaire à la comparaisons entre les entrées et la séquence). Les tables à adressage dispersé ont d'autres inconvénients, en particulier la façon dont on doit procéder pour le calcul d'adresse. Dans une situation ordinaire, on a une chaîne s dont la longueur est connue. Si on veut tester l'appartenance de s dans le dictionnaire, on calcule son adresse $h(s)$ grâce à la fonction de dispersion $h(\cdot)$ et on vérifie si la table, à cette adresse, contient s ou non (laissons de côté pour l'instant le problème

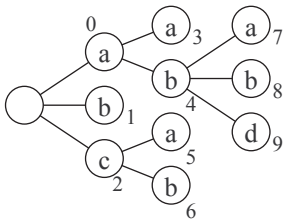


FIG. 9.1 – Une treille représentée par un arbre m-aire. Ici, la treille maintient les chaînes a, aa, aba, abb, abc, b, c, ca et cb. La racine, à laquelle correspondrait la chaîne vide $\perp$, n'a pas reçu d'adresse car la chaîne vide n'est pas utilisée dans cet algorithme.

des collisions). Or, dans le cas qui nous intéresse, on ne connaît pas d'avance la longueur de s, la plus longue concordance dans le dictionnaire. Une des façons possibles de procéder, c'est de calculer l'adresse en utilisant seulement les n premiers symboles de s (une stratégie qui rapelle celle de LZRW), réduisant le temps de calcul de l'adresse mais résultant en des recherches plus longues puisque nous sommes susceptibles d'avoir de très nombreuses collisions : toutes les chaînes dont le préfixe est le même que le préfixe de s se retrouvent dans la même case de départ.

Une structure de donnée qui permet des recherches et des ajouts efficaces, c'est la treille [74]. La figure 9.1 montre une version simple d'une treille. Une treille n'est qu'un arbre m-aire où chaque nœud maintient une donnée, ici un symbole de Σ. Les séquences sont maintenues indirectement : elles sont représentées par la séquences de nœuds visités depuis la racine jusqu'à une feuille donnée. Ainsi, à la fig. 9.1, la séquence représentée par le nœud 8 est abb, qui est obtenue en parcourant la treille en partant de la racine et en descendant jusqu'à ce nœud. Cette représentation est très efficace en temps. Trouver une concordance se fait en temps linéaire en la longueur de la concordance car il suffit de choisir le i^{e} fils d'un nœud pour vérifier la concordance entre la treille et un mot $(\omega : \sigma_i)$, en considérant que la chaîne ω nous a mené à ce nœud. L'ajout se fait en temps constant : si on a pas de fils pour σ_i pour le nœud courant, on le crée et ainsi on crée $(\omega : \sigma_i)$. Le désavantage, c'est que chaque nœud doit réserver $O(|\Sigma|)$ pointeurs pour ses fils, puisqu'*a priori* tout symbole de l'alphabet peut être ajouté à une séquence.

On peut réduire la quantité de pointeurs d'une treille significativement. Plutôt que d'avoir $|\Sigma|$ pointeurs par nœud, on peut utiliser seulement trois pointeurs par nœuds, pour un total de $3N$ pointeurs plutôt que $N\,|\Sigma|$, et comme $|\Sigma| \gg 3$ (typiquement $|\Sigma| = 256$), cela mène à une grande économie de mémoire. Ces trois pointeurs sont les pointeurs vers le parent, le premier fils et le frère du nœud courant. On obtient une structure telle qu'illustrée à la fig. 9.2. Plutôt que de retrouver dans chaque nœud une multitude de pointeurs à chacun des fils, on ne retrouve qu'un pointeur vers le premier des fils, et les autres sont accessibles seulement en suivant le pointeur horizontal qui lie les frères les uns aux autres. La recherche d'une concordance dans une telle structure demeure en temps espéré linéaire, bien qu'elle puisse dégénérer en temps $O(|\Sigma|\ |s|)$ pour une concordance s. La recherche ne dégénère que lorsque les séquences sont essentiellement aléatoire, où chaque sous-séquence a tous les symboles de l'alphabet comme suffixe. Autrement, le temps de recherche demeure $O(|s|E[b])$, où $E[b]$ est le facteur de branchement moyen.

Cette organisation pour la treille permet aussi de n'utiliser que quatre tableaux pour main-

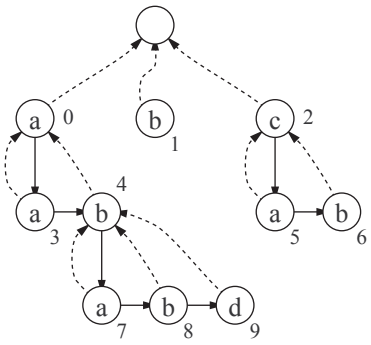


FIG. 9.2 – La structure d'une treille linéaire. Les flèches pleines vers le bas représentent le lien d'un nœud vers son premier fils. Les flèches pleines horizontales maintiennent les liens entre les nœuds frères. Les flèches pointillées représentent le lien vers le parent. Avec une telle structure, il est aisé de remonter vers la racine à partir de n'importe quel nœud.

tenir la structure. Le premier tableau maintient le symbole contenu dans chaque nœud, les trois autres les pointeurs. Comme en général la taille du dictionnaire est limitée, dans notre cas à 4096 entrées, il n'est pas nécessaire d'opter pour des pointeurs sur 32 bits, des entiers sur 16 bits suffisent amplement. Une telle configuration ne demanderait que 28 kilo-octets pour maintenir la treille, rendant ainsi possible la réalisation d'un compresseur/décompresseur sur une machine de faible puissance.

Nous avons déjà considéré le problème de recherche de concordance. Pour trouver la plus longue concordance entre la séquence à compresser et les chaînes contenues dans la treille, il suffit de commencer à la racine et de descendre la treille jusqu'à ce que nous trouvions un nœud qui est une feuille ou dont aucun fils ne correspond au prochain symbole dans la séquence. L'index de ce nœud donne l'index de la plus longue concordance entre les chaînes contenues dans le dictionnaire et la séquence à compresser. Pour extraire une chaîne de la treille, nous commençons au nœud indiqué par l'index et nous remontons à la racine en suivant les pointeurs vers les parents. Cela nous donne la chaîne inversée, et nous laisse avec essentiellement deux options. Soit nous utilisons une pile auxiliaire où la chaîne inversée est copiée avant d'être remise à l'endroit dans la séquence décompressée, soit nous maintenons dans chaque nœud sa profondeur et nous copions directement la chaîne dans la séquence décompressée. Si on connaît la profondeur d'un nœud, on connaît donc la longueur de la chaîne qu'il représente ; donc on peut savoir où commencer la copie à reculons et se dispenser de l'utilisation d'une pile.

9.2.2 Performance de LZW

Welch rapporte des taux de compression obtenus jusqu'à 6 : 1 pour des fichiers tests qu'il a choisi (et auxquels nous n'avons pas accès). Des améliorations dans la façon d'encoder les index permettent d'atteindre des taux entre 4 : 1 et 5 : 1 pour les textes anglais (alors que Welch rapporte environ 2 : 1), et jusqu'à 10 : 1 sur des fichiers modérément ténus. On a montré que l'algorithme de Welch est asymptotiquement optimal. En effet, il code toute séquence suffisamment longue avec un nombre de bits moyen par symbole près de l'entropie de la source, étant

donné un dictionnaire contenant un grand nombre d'entrées [130, 182].

Les vitesses de décompression comme de compression sont aussi intéressantes. Une implémentation que nous avons réalisée utilisant une treille linéaire pour maintenir le dictionnaire permet de compresser les données de test à un rythme allant jusqu'à 5 méga-octets par seconde et de décompresser à un rythme allant jusqu'à 10 méga-octets par seconde, en comptant le temps écoulé à accéder aux fichiers en lecture et en écriture. Ces tests ont été fait sur notre machine de référence, un Pentium II à 333MHz. Cette implémentation utilise aussi les codes récursivement *phase-in* tel que suggérés par Acharya et Já Já pour encoder les index [1, 2]. Nous avons présenté ces codes à la section 5.3.4.2. Comparativement, les implémentations de LZW que l'on retrouve dans différents compresseurs GIF (dont celui de XView et Thumbs+) ne compressent qu'au rythme de quelques méga-octets par seconde et décompressent les données légèrement plus rapidement.

9.3 LZW appliqué à l'image

L'algorithme LZW peut être appliqué à la compression d'essentiellement n'importe quel type de données. Une implémentation typique de l'algorithme découpe la séquence au niveau de l'octet et ne fait aucune supposition quant à la nature de la source aléatoire qui génère la séquence. Cette quasi absence de suppositions sur la source rend l'algorithme universel. De plus, quelque que soit la source stationnaire, l'algorithme compressera une séquence suffisamment longue avec un nombre de bits par symboles moyen s'approchant de l'entropie de la source.

Avant d'appliquer LZW aux images, il convient de présenter les deux méthodes principales de représentation des images. Ces deux méthodes sont la représentation *true color* et la représentation à base de palette de couleur.

9.3.1 Les images *true color*

Les images dites *true color* (on trouve aussi *truecolor* et *high color*) sont composés de pixels qui sont des tuples $\langle R, G, B \rangle$, où chaque composante comporte un certain nombre de bits. Le modèle de couleur avec les composantes primaires rouge, verte et bleue (d'où RGB) est le plus couramment supporté par le matériel, bien qu'il en existe d'autres (voir l'appendice E à ce sujet). Typiquement, le matériel est capable d'afficher des images où chacune des composantes comporte de 5 à 8 bits, parfois moins, parfois plus. Ainsi, un mode d'affichage où chaque composante est représentée par 8 bits permet d'afficher environ 17 millions de couleurs[1]. Nous trouvons aussi d'autres modes, par exemple où les trois composantes sont représentées ensemble sur 16 bits, dont 5 bits sont alloués à la composante R, 6 bits à la composante G et les 5 bits restants à la composante B.

Peu importe le modèle de représentation de couleurs utilisé, un grand nombre de bits par composante fait en sorte qu'il soit fort probable que les bits de poids faible de chaque composante soient essentiellement aléatoires. Sur 8 bits, par exemple, il y a fort à parier que les deux ou trois bits de poids faible soient fortement bruités. Avec ce bruit, même en séparant l'image en trois plans (un pour chaque composante de couleur) et en appliquant l'algorithme LZW sur chacun des plans séparément, il est peu probable qu'on réussisse à extraire suffisamment de sous-séquences

1 24 bits permettent de représenter 16 777 216 combinaisons différentes.

répétées pour obtenir une compression intéressante. Cela réduit l'intérêt de l'utilisation LZW sur ce type d'image. Nous verrons, à la section 9.3.4 comment la randomisation des valeurs provoquée par le *dithering* affecte la compression obtenue par l'algorithme LZW.

9.3.2 Les images à palettes de couleur

S'il est impossible que chaque pixel puisse être décrit par son propre tuple $\langle R, G, B \rangle$, on peut avoir recours au mode de palette de couleur [2]. Avec cette représentation, chaque pixel de l'image devient un index dans un tableau, généralement de taille très limitée, où chaque entrée est un tuple $\langle R, G, B \rangle$. Ce tableau, que l'on nomme la palette, comporte N couleurs adéquatement choisies pour représenter l'image. Nous reviendrons sur le choix des couleurs grâce à un algorithme qui les choisira de façon à minimiser l'erreur moyenne de reconstruction. Typiquement, nous aurons de $N = 16$ à $N = 256$, de façon à ce que chaque pixel puisse tenir sur un octet ou moins.

Lorsque l'image est reconstruite à l'affichage, chaque index est remplacé par la couleur qu'il indique dans la palette. Cette conversion est faite soit par le matériel lorsque c'est le seul mode disponible, soit en logiciel lorsque le mode courant d'affichage est le mode *true color*. La situation est illustrée à la fig. 9.3, où on voit, à gauche la représentation de l'image en série d'index, au centre, la palette et à droite l'image reconstruite.

La résolution *effective* des composantes de couleurs des entrées de la palette dépend du matériel bien que par le protocole on puisse insister pour que chaque composante ait un nombre quelconque de bits. Le protocole peut supporter un nombre quelconque de bits, mais on voudra utiliser la résolution maximale permise, qui est typiquement de 8 bits par composante, bien que certains protocoles permettent jusqu'à 16 bits par composante. Si les couleurs sont choisies de façon à assurer la meilleure reconstruction possible de l'image, il est possible que les couleurs dans la palette puissent être plutôt quelconques ; et le choix des couleurs peut faire en sorte que deux entrées qui se suivent dans palette représentent des couleurs très différentes. Puisque l'algorithme LZW travaillera exclusivement sur les index, les couleurs elles-même n'entreront pas en compte dans la qualité de la compression obtenue.

9.3.3 Le protocole GIF

Le protocole GIF, introduit par Compuserve en 1987, révisé en 1989, est l'un des formats de fichiers d'images les plus communs sur le web [54]. Ce format permet de transporter des images à base de palettes d'au plus 256 couleurs et des animations grâce à un mode multi-image. La possibilité de représenter des animations est probablement ce qui a permi aux images GIF de survivre aussi longtemps en occupant une niche bien particulière sur le web, c'est-à-dire les banderolles publicitaires animées. Ces banderolles GIF seront progressivement remplacées par des animations basées sur le système Flash Macromédia, qui est un environnent multimédia complet.

Alors, pourquoi utiliser le protocole GIF ? D'abord parce que l'algorithme de compression de GIF est l'algorithme LZW légèrement modifié. Ensuite parce que les décodeurs GIF sont omniprésents : tous les navigateurs web sont capables d'afficher ces images. De plus, nous utiliserons GIF pour comparer nos résultats sur un pied d'égalité avec des résultats obtenus par d'autres auteurs [46]. Enfin, le système GIF n'est pas excessivement complexe et se prête facilement à

2 On dit aussi parfois *pseudocolor mode*.

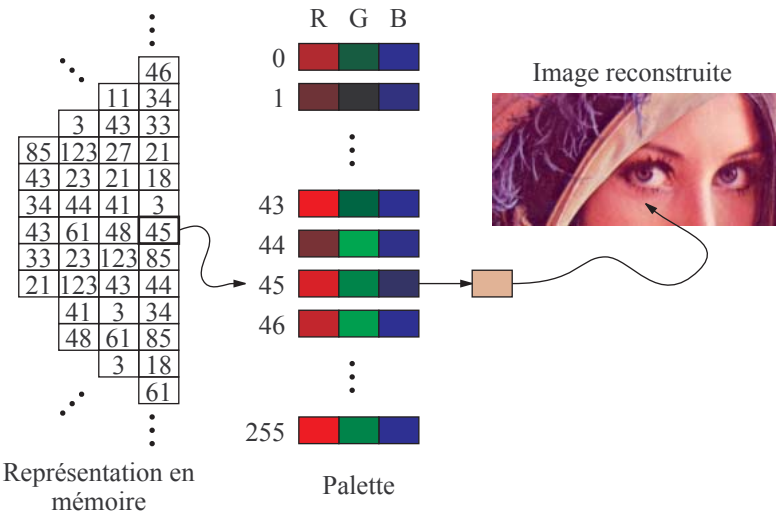


FIG. 9.3 – Les images à palette de couleur. L'image est représentée en mémoire par une matrice de nombres qui font référence aux couleurs prédéfinies de la palette de couleur. La palette elle-même n'est qu'un tableau où sont définies les couleurs, le plus souvent par des triplets RGB.

une implémentation complète du protocole. Ces deux raisons font en sorte qu'il est facile de construire l'environnement de test pour LZW avec perte et de voir les images résultantes. Nos implémentations des algorithmes que nous présentons sont compatibles au protocole GIF. Nos encodeurs prennent des décisions différentes de l'algorithme LZW de base, mais les séquences compressées sont compatibles avec les décodeurs LZW de base.

Comme l'utilisation du protocole GIF sert principalement à la comparaison des résultats, nous n'en décrirons pas les détails. Que vous chaut où sont les bits qui définissent le nombre d'entrées dans la palette ? Que les données compressées soient organisés en petits blocs de 256 octets ne porte aucun intérêt en soi, si ce n'est de savoir qu'à tous les blocs de 256 octets un octet est gaspillé pour indiquer la longueur du bloc. La spécification (presque) complète du protocole se trouve dans le document de Compuserve [54]. Le seul détail de GIF sur lequel nous insisterons, c'est la façon dont les index sont encodés.

Alors que Welch préconise l'utilisation de codes de taille fixe, par exemple 12 bits, la version GIF va utiliser des codes dont la taille varie en fonction du nombre d'entrées occupées dans le dictionnaire. En comptant les deux codes spéciaux (fin de séquence et remise à zéro du dictionnaire) qui occupent deux entrées dans le dictionnaire, la longueur des codes est toujours $\lceil \lg |D| \rceil$, où $|D|$ est le nombre d'entrées occupées dans le dictionnaire et non sa taille maximale. Comme le dictionnaire est initialisé avec l'alphabet de la séquence à compresser, plus les deux codes réservés, les codes commencent à $\lceil \lg(|\Sigma| + 2) \rceil$ bits. Dans le protocole GIF, Σ doit avoir un nombre de symboles qui est une puissance de deux. Si l'alphabet comporte un nombre de symboles qui n'en est pas un, il est artificiellement augmenté à la prochaine puissance de deux, et ces nouvelles entrées sont perdues. Malgré l'augmentation de l'alphabet, les longueurs variables de codes permettent de sauver jusqu'à $12 - \lceil \lg(|\Sigma| + 2) \rceil$ bits pour un dictionnaire de taille 4096,

qui est standard avec le protocole GIF [3].

9.3.4 Réduction de couleurs et *dithering*

Le protocole GIF ne permettant pas d'avoir plus de 256 couleurs, il se peut que nous ayons à réduire le nombre de couleurs présentes dans l'image. Une image naturelle peut comporter quelques milliers de couleurs distinctes, dépendamment de la résolution originale des couleurs et de l'image, et nous devrons en choisir au plus 256 de façon à avoir la meilleure reconstruction possible. Cependant, la réduction de couleur peut introduire des artefacts comme les faux contours et les pertes de relief.

Les faux contours apparaissent lorsqu'une région en dégradé se trouve représentée par un relativement petit nombre de couleurs. Plutôt que d'avoir une courbe relativement lisse nous nous retrouvons avec un escalier, ce qui provoque un effet très visible. Pour la même raison, nous pouvons avoir une surface qui était perçue en relief grâce à un dégradé subtil nous apparaître plate puisqu'elle n'est plus représentée que par quelques couleurs distinctes. Pour remédier à ces artefacts, nous allons avoir recours au *dithering* qui utilise une modulation spatiale des couleurs pour obtenir des tons qui approximent bien la couleur originale, du moins sur une région. L'effet de *dithering* fonctionne bien lorsque la résolution est très élevée ou la distance de visualisation est suffisamment grande.

Dans cette section nous présenterons brièvement les méthodes de réduction de couleurs et les méthodes de *dithering*. Nous verrons aussi quels sont les effets du *dithering* (qui est essentiellement une randomisation des pixels) sur la compression LZW.

9.3.4.1 Méthodes de réduction de couleurs

La réduction de couleur est essentiellement un problème de discrétisation vectorielle. Nous avons déjà présenté quelques méthodes à la section 4.1.2. Le problème particulier de la discrétisation de couleur a donné naissance à des solutions qui se distinguent légèrement des méthodes de discrétisation de vecteurs quelconques. On retrouve par exemple les algorithmes de popularité, de coupe médiane et par *octree*.

L'algorithme de popularité est très simple et donne aussi les pires résultats. L'algorithme de popularité consiste simplement à trouver les k valeurs les plus fréquentes lorsque l'on veut k valeurs. Cet algorithme représente bien les couleurs les plus fréquentes mais est très désavantageux pour les couleurs rares. En effet, les couleurs les plus fréquentes viennent en général en plusieurs variantes subtiles perceptuellement très près de la couleur principale, ce qui a pour effet d'augmenter artificiellement le nombre de couleurs populaires. Les couleurs rares qui correspondent aux petits détails ne font jamais partie des k plus populaires et sont ainsi détruites et remplacées par une des k couleurs, ce qui ne garantit nullement les résultats.

Les algorithmes de coupe médiane découpent récursivement l'espace de couleurs en cuboïdes de poids égaux [93, 94, 226]. On commence par construire un histogramme en trois dimensions, où les fréquences de chaque tuple (r, g, b) est comptabilisé. Cet algorithme comporte plusieurs variantes, dont les coupes alternantes et les coupes en variances maximales. La variante par

3 Il existe de meilleures stratégies pour encoder les index, comme l'utilisation des codes récursivement *phase-in*, tels que suggérés par Acharya et Já Já. Nous avons présenté ces codes à la section 5.3.4.3.

coupes alternantes découpe le cube RGB en trouvant d'abord la médiane sur l'axe R, puis au niveau d'en dessous sur l'axe G, puis sur l'axe B jusqu'à ce que le nombre de cuboïdes désiré soit atteint. La variante par coupes en variances maximales coupe, à chaque étape, sur l'axe qui a la plus grande variance (bien que la médiane minimise une norme L_1). Cela demande de tester les contenus des cuboïdes avant de procéder à la coupe. Il existe des astuces pour accélérer les calculs des variances [4] [226]. Ces deux variantes donnent de très bons résultats.

Enfin viennent les méthodes à base d'*octree* [82]. La méthode à base d'*octree* découpe récursivement le cube RGB en octants, jusqu'à ce que le nombre de régions obtenues corresponde au nombre de couleurs désirées. Une façon de faire, c'est de construire un arbre d'ordre 8 où les clefs sont construites à partir des bits des composantes de couleurs. Si on a n bits par composante, les clefs contenues dans les nœuds à la profondeur i sont formés par la composition de chacun des $(n-i-1)^{\text{e}}$ bits des composantes. Si r_j donne le j^{e} bit de la composante rouge (pareillement pour g_j et b_j), les feuilles d'un nœud à la profondeur j sont indexées par $r_{n-j-1}g_{n-j-1}b_{n-j-1}$ considéré en tant que nombre. On procède donc à l'ajout successif de chaque couleur de l'image dans l'arbre en maintenant la fréquence des feuilles et les fréquences cumulatives des sous-arbres dans les nœuds internes. Une fois chaque pixel de l'image cumulé, on procède à l'élagage de l'arbre en commençant par les nœuds terminaux (ceux qui ne supportent que des feuilles) les moins fréquents. On les remplace par une seule feuille qui maintient le centroïde pondéré du cube représenté. On répète ce procédé jusqu'à ce qu'il ne reste que le nombre de feuilles désiré. Puisque cette méthode divise récursivement le cube RGB en octants, les régions sont obligatoirement des cubes, et non plus des cuboïdes ou des régions convexes quelconques, ce qui empêche l'algorithme de trouver des coupes qui pourrait être meilleures.

Pour les expériences, nous avons utilisé l'algorithme de coupe médiane. La plupart des logiciels de traitement d'image offrent la réduction de couleur grâce à cet algorithme, ce qui en fait un bon choix de départ. Cela permettra aussi de comparer directement nos résultats aux résultats d'autres auteurs.

9.3.4.2 Méthodes de *dithering*

La technique de *dithering* consiste à moduler spatialement les couleurs et les tons de gris de façon à créer, sur une petite régionde l'image, la couleur moyenne désirée. On voit des exemples de cette technique dans les images imprimées dans les journaux, où les tons de gris sont approximés par des points plus ou moins denses. On peut utiliser une modulation selon une grille régulière ou une modulation pseudo-aléatoire. La modulation selon une grille régulière $n \times n$ permet d'approximer $O(n^2)$ tons de gris, mais forme un motif distinctif qui est très visible si la densité n'est pas assez élevée. La modulation pseudo-aléatoire brise tout motif et est visuellement moins distrayante que la modulation régulière, surtout à basse résolution.

L'approche habituelle pour produire une modulation pseudo-aléatoire consiste à propager l'erreur faite sur un pixel sur les pixels voisins. L'erreur dans ce contexte, c'est la différence entre la couleur désirée et la plus proche couleur disponible. Sur une grande région, on espère que l'erreur moyenne sera réduite, au moins visuellement. Pour un pixel de couleur x, on trouve

4 Ces astuces sont basées sur le fait que la variance est une mesure aggrégative. On utilise le théorème de Huygens $Var(X) = E[X^2] - E^2[X]$ qui permet de briser le calcul des espérances sur les cuboïdes en utilisant la méthode d'inclusion-exclusion. Il suffit d'avoir deux autres tableaux qui calculent les sommes cumulatives des x^2 et les sommes cumulatives des x pour chaque tuple (r, g, b), c'est-à-dire qu'à chaque case (r, g, b) on trouve la somme des x^2 et des x de 0 à r, de 0 à g, de 0 à b (ou toutes autres bornes adéquates).

Floyd-Steinberg

	X	7
3	5	1

Stucki

		X	8	4
2	4	8	4	2
1	2	4	2	1

Burkes

		X	8	4
2	4	8	4	2

Jarvis, Judice, Ninke

		X	7	5
3	5	7	5	3
	3	5	3	

FIG. 9.4 – Les grilles de propagation d'erreur. Les diviseurs pour les grilles de Floyd-Steinberg : 16, Burkes : 32, Stucki : 42 et Jarvis, Judice, Ninke : 46

la couleur $\hat{x}$ de la palette qui est la plus semblable. Une erreur $x - \hat{x}$ est encourue. On propage alors une fraction de cette erreur sur les pixels voisins (habituellement en ordre de balayage de l'image, voir la fig. 9.4). Ainsi, un pixel y voisin du pixel x sera approximé par $\hat{y}$, mais où $\hat{y} \approx y + \varepsilon(x)(x - \hat{x})$, soit le vrai pixel y modulé par l'erreur reçue du pixel x et où $\varepsilon(x)$ est un paramètre qui dépend de la grille de propagation. En général, chaque pixel peut recevoir des erreurs de plusieurs autres pixels. Cumulées, et en supposant que la palette soit appropriée, ces erreurs permettent d'améliorer grandement la qualité perçue de l'image.

La fig. 9.4 donne les plus simples que l'on trouvera dans les ouvrages de référence comme [71, 174]. Ces méthodes ne diffèrent entre elles que par la forme de la grille de propagation et les coefficients ; la tactique reste la même. Certaines de ces grilles de propagation d'erreur ont des coefficients qui font en sorte qu'il soit facile de n'utiliser que l'arithmétique en nombre entiers pour effectuer les calculs de propagation d'erreur. Ces coefficients sont choisis de façon *ad hoc*.

D'autres méthodes de *dithering* sont basées sur le *blue noise*. Le bruit blanc (*white noise*) est un bruit dont la caractéristique spectrale est telle que la puissance de toutes les fréquences est égale. Pour le bruit bleu, les fréquences sous un certain seuil ont une puissance nulle et les fréquences au dessus ont une puissance constante [212]. Le bruit bleu est donc un bruit blanc qui a subit un filtre passe-haut. Ulichney propose de moduler les coefficients de dispersion d'erreur (ces mêmes coefficients que l'on retrouve dans les méthodes de la fig. 9.4) grâce à une fonction de bruit bleu de façon à détruire toute longue chaîne de propagation d'erreur. Il suggère aussi d'utiliser un balayage boustrophédonique sur l'image de façon à inverser la direction générale de la propagation d'erreur à chaque ligne. L'effet global de l'utilisation du bruit bleu est de réduire les correlations de pixels qui pourraient faire apparaître un motif. Pour le bruit bleu, ses propriétés spectrales et ses applications au *dithering*, on consultera [142, 144, 143, 212]. Pour le *dithering* en général, on consultera plutôt [5, 9, 14, 79, 201].

Nous avons retenu les méthodes de Floyd-Steinberg, de Burkes, de Stucki et de Jarvis, Judice et Ninke parce qu'elles se retrouvent dans plusieurs logiciels de traitement d'image. Il est donc vraisemblable qu'un utilisateur aura recours à l'une d'elle lorsqu'il réduira le nombre de couleur de l'image avant la compression. À la prochaine sous-section, nous comparerons leurs effets sur la compression LZW appliquée aux images.

9.3.4.3 Impacts du *dithering* sur la compression

Le *dithering* améliore l'aspect des images en introduisant une modulation qui est essentiellement du bruit. Si le *dithering* rend les images plus plaisantes, il les rend aussi plus difficiles à compresser. L'algorithme LZW va réussir à obtenir une certaine compression en exploitant les sous-chaînes de pixels semblables dans l'image. Or, le *dithering* randomise significativement les pixels, ce qui réduit grandement nos chances de trouver de longues sous-chaînes redondantes.

La fig. 9.5 montre quelques expériences avec les images du corpus Kodak. On y voit des images dont les couleurs sont réduites à 32, 64, 128 ou 256 couleurs, et les taux de compression obtenus avec ces image sans *dithering* et avec quatre méthodes de *dithering*. Plus la méthode de *dithering* produit des pixels (localement) bien randomisés, moins le taux de compression est bon. En examinant la fig. 9.5, on voit que les méthodes de Floyd-Steinberg et de Burke sont les plus nocives à la compression. Les méthodes de Stucki et de Jarvis, Judice et Ninke donnent des résultats similaires, bien que ce soit la méthode de Jarvis, Judice et Ninke qui nuise le moins à la compression.

Alors, si ces méthodes randomisent les pixels, comment expliquer une telle différence dans les taux de compression ? Considérez la fig. 9.6, où on montre les résultats des quatres méthodes sur une même région de l'image test *parrots*. La méthode Floyd-Steinberg produit des points qui semblent très aléatoires. Déjà, la méthode de Burkes produit des points qui semblent former des chaînes et on peut voir de plus longues répétitions. Les méthodes de Stucki et Jarvis, Judice et Ninke produisent des motifs et des chaînes de pixels qui sont plus facilement utilisés par l'algorithme de compression, même si aucun motif n'est apparent. Le résultat net, sur les 15 images tests utilisées (dont les résultats de seulement six sont montrés à la fig. 9.5) c'est que les méthodes de Stucki et Jarvis, Judice et Ninke sont essentiellement comparables, bien que la seconde méthodes donne des résultats légèrement meilleurs en termes de compression.

Puisque c'est la méthode de *dithering* de Jarvis, Judice et Ninke qui aide le plus (ou nuit le moins à) la compression, c'est cette méthode de *dithering* qui a été retenue pour nos tests. Remarquez que les images *Lena* et *airplane* qui sont utilisées aux fins de comparaisons avec les résultats présentés par Chiang et Po, n'ont pas subi le processus de *dithering*. Ces auteurs ne parlent en effet d'aucune méthode de *dithering*, nous laissant supposer qu'ils n'en ont pas utilisé.

9.4 Mesures de qualité sur les images

À l'appendice E nous présentons les espaces de couleurs les plus communs. Les espaces de couleurs les plus intéressants représentent les couleurs avec trois composantes : la luminance, la teinte et la saturation, ou pureté de la couleur. Parmi ces espaces, on retrouve HSV et $L^*a^*b^*$. L'espace HSV représente la teinte par un angle sur une « roue de couleur » où on retrouve les six couleurs élémentaires, ce qui le rend particulièrement intuitif à utiliser ; c'est d'ailleurs pourquoi on le retrouve souvent dans les interfaces à l'usager. L'espace $L^*a^*b^*$ est moins intuitif mais possède une propriété très intéressante : il est perceptuellement uniforme. C'est-à-dire que la région autour d'une certaine couleur définie par la *just noticable difference* est sphérique. Les transformées de RGB vers HSV ou $L^*a^*b^*$ sont non-linéaires. Elles impliquent des fonctions trigonométriques ou des extractions de racines, ce qui les rends moins attrayantes pour les applications où la vitesse de calcul importe.

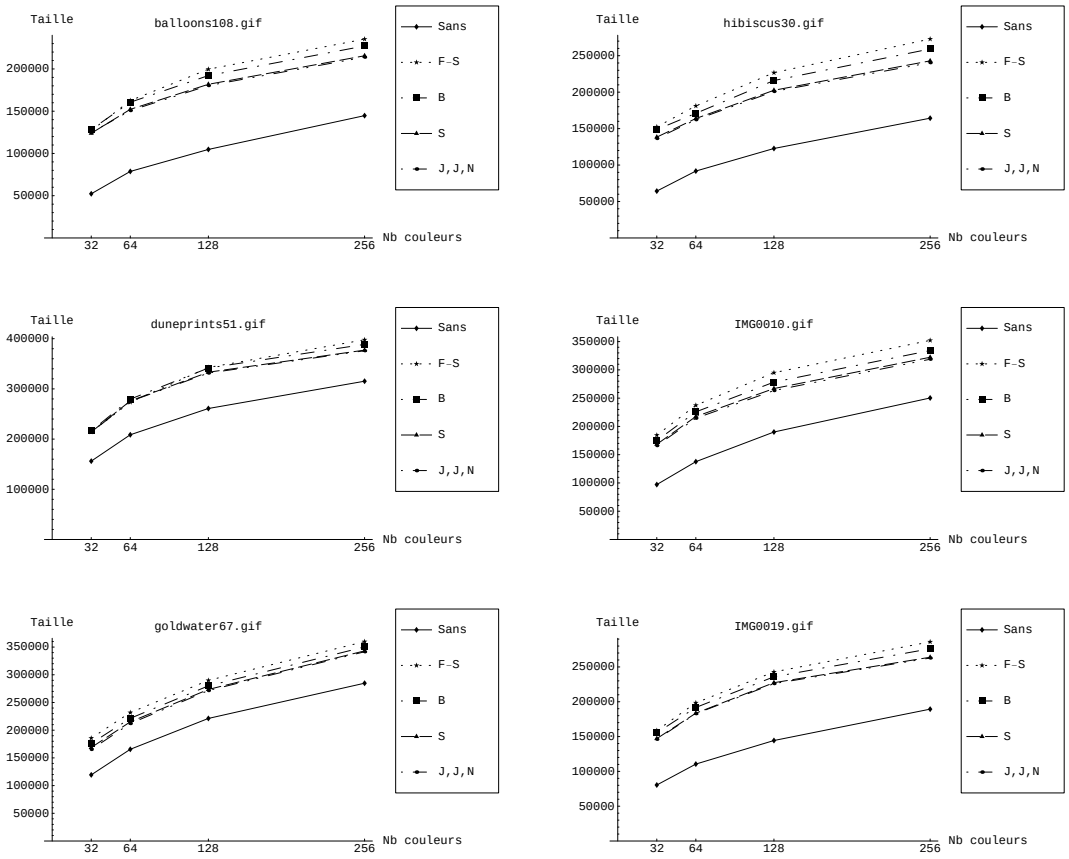


FIG. 9.5 – L'effet du *dithering* sur la compression LZW. Les images sont tirées du corpus Kodak. Les méthodes de *dithering* illustrées ici sont les méthodes de Floyd-Steinberg, de Burkes, de Stucki et de Jarvis, Judice et Ninke. On y compare aussi les taux de compressions obtenus sans le dithering. Nous avons fait les tests avec 32, 64, 128 et 256 couleurs pour chaque image.

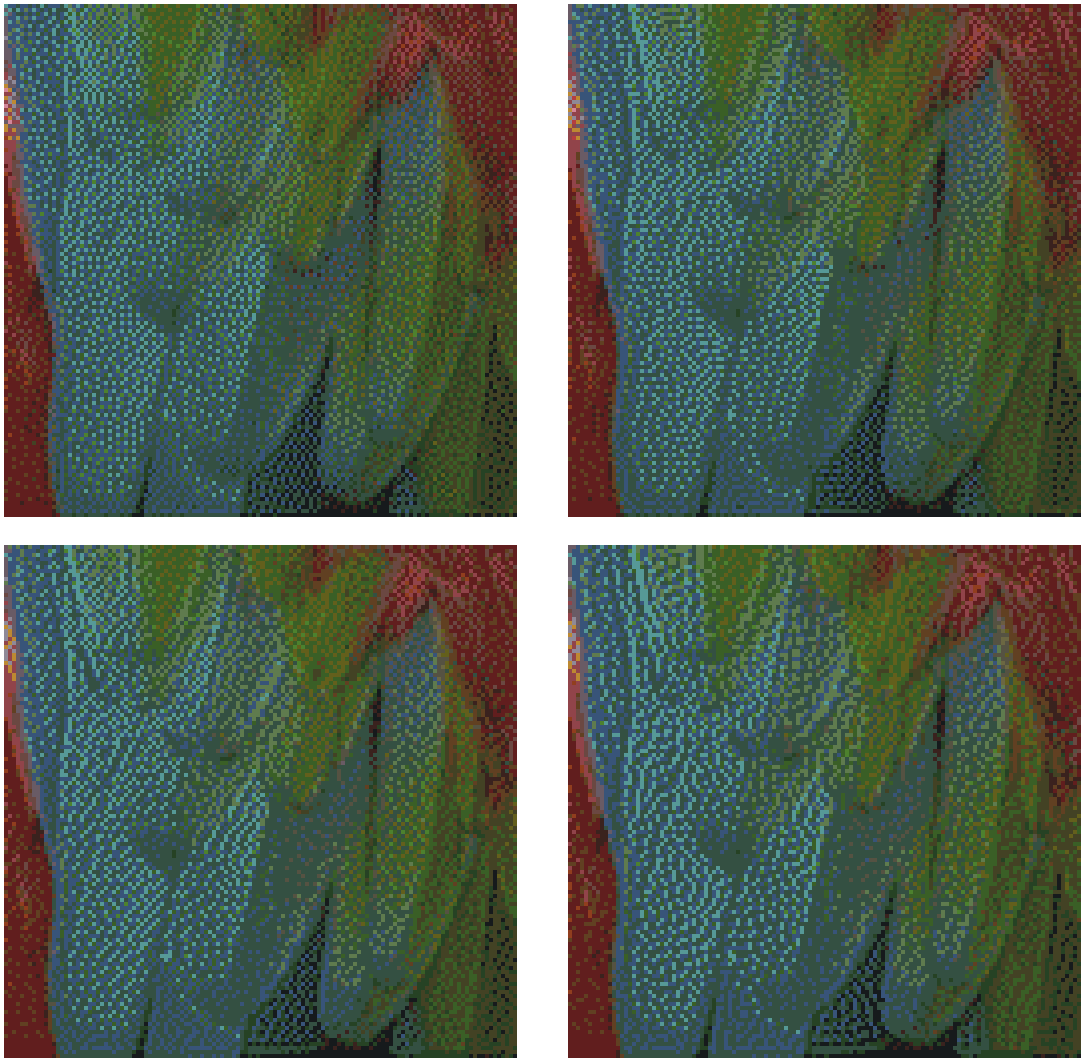


FIG. 9.6 – Les effets des techniques de *dithering* sur l'image *parrots* (voir la figure D.1). De gauche à droite : les méthodes de Floyd-Steinberg, de Burkes, de Stucki et de Jarvis, Judice et Ninke.

Cependant, pour des raisons computationnelles, on se rabbat sur les espaces de couleurs pour lesquels il existe une transformation linéaire depuis et vers RGB. Parmi ceux-ci, on retrouve les espaces dont la composante principale encode la luminance (la luminosité perçu) de la couleur : YUV, YIQ, YC_rC_B, etc. Ces espaces tiennent compte de la notion heuristique selon laquelle l'œil est beaucoup plus sensible aux variations dans la luminance que dans le teinte ou la pureté des couleurs. Il est cependant difficile d'établir des ratios entre ces différentes sensibilités et on aura recours à des pondérations heuristiques puisque les deux autres composantes ne correspondent habituellement pas vraiment ni à la teinte ni à la pureté ; ces espaces sont contraints, par la transformation linéaire, à n'être que des rotations et des mises à l'échelle du cube RGB.

Par exemple, la luminosité d'un pixel (r, g, b) peut être calculée par

$$\|(r, g, b)\| = \sqrt{\gamma_r^2\, r^2 + \gamma_g^2\, g^2 + \gamma_b^2\, b^2}$$

où $\gamma_r = 0.299$, $\gamma_g = 0.587$ et $\gamma_b = 0.114$ si on considère les phosphores des anciens moniteurs ou des télévisions, ou encore $\gamma_r = 0.2126$, $\gamma_g = 0.7152$ et $\gamma_b = 0.0722$ si on utilise plutôt les caractéristiques des nouveaux phosphores. La première variante correspond au Y de plusieurs espaces linéaires. Ce type de mesure ne permet pas de tenir compte explicitement des variations de teinte ou de saturation — d'autant plus que ces composantes n'existent pas vraiment dans ces espaces — ce n'est qu'une mesure de la luminosité perçue. Pour comparer deux couleurs, on pourrait utiliser

$$\|(r_1, g_1, b_1) - (r_2, g_2, b_2)\| = \sqrt{\gamma_r^2\, (r_1 - r_2)^2 + \gamma_g^2\, (g_1 - g_2)^2 + \gamma_b^2\, (b_1 - b_2)^2} \tag{9.1}$$

ce qui revient à pondérer différemment les axes du cube RGB selon leur contribution à la luminosité perçue. Or, cette mesure n'est pas véritablement adéquate parce que la différence entre deux tons de bleu n'a pas besoin d'être $0.299/0.114$ fois une différence entre deux tons de rouge pour produire la même différence perçue. Chiang et Po proposent de transformer les tuples RGB en YUV (grâce à la transformée eq. (E.11), en appendice E) et d'utiliser la métrique

$$\|(y_1, u_1, v_1) - (y_2, u_2, v_2)\| = \sqrt{\frac{5}{8}\,(y_1 - y_2)^2 + \frac{3}{16}\,(u_1 - u_2)^2 + \frac{3}{16}\,(v_1 - v_2)^2} \tag{9.2}$$

qui pondère explicitement (mais heuristiquement) les composantes [46]. Si nous utilisions plutôt $L^*a^*b^*$, les coefficients de pondération seraient inutiles puisque l'espace de couleur compenserait déjà pour ceux-ci ; la mesure dégénérerait en une mesure euclidienne. Cependant, utiliser l'éq. (9.1) ou l'éq. (9.2) ne semble pas affecter significativement ni la compression obtenue ni la qualité des images reconstruites. Pour nos test, encore à des fins de comparaison, nous utiliserons l'éq. (9.2).

Une fois la métrique choisie pour mesurer les différences entre les pixels, il nous reste à établir une mesure générale pour l'image. Nous avons essentiellement deux alternatives : le SNR ou le PSNR. Le SNR (ou *Signal to Noise Ratio*) calcule une distortion moyenne sur les données. Le PSNR (*Peak Signal to Noise Ratio*) calcule l'erreur moyenne par rapport à la magnitude de la plus grande valeur dans les données originales.

Pour des données à une dimension, le SNR est défini par

$$SNR = \frac{\sum_i \hat{x}_i^2}{\sum_i (x_i - \hat{x}_i)^2}$$

où les x_i sont les données originales et les $\hat{x}_i$ sont les données reconstruites. Pour obtenir une version en décibels du SNR, on utilise l'identité

$$SNR(dB) = 10 \log_{10} SNR$$

Appliqué aux pixels, la formule de SNR devient

$$SNR = \frac{\sum_i \|\hat{x}_i\|^2}{\sum_i \|x_i - \hat{x}_i\|^2}$$

où $\|x\|$ est la métrique utilisée. Le PSNR, quant à lui, est défini par

$$PSNR = \frac{(\max\{x_i\})^2}{\frac{1}{N}\sum_i (x_i - \hat{x}_i)^2} \tag{9.3}$$

et cette formule se généralise aux pixels de la même façon que pour le SNR. L'utilisation du PSNR est comparativement rare dans la littérature de la compression. Nous utiliserons donc l'éq. (9.4) convertie en dB et l'éq. (9.3) pour comparer nos résultats.

9.5 Variations avec perte de l'algorithme LZW

Présentons donc enfin nos contributions. Dans cette section, nous présenterons nos deux variantes de LZW avec perte, G-LLZW et P-LLZW que nous comparerons à la variante proposée par Chiang et Po que nous nommerons CP-LLZW [46]. L'algorithme G-LLZW a été présenté dans le journal étudiant l'interactif [159] et l'algorithme P-LLZW a été présenté à la conférence sur la compression de données de Snowbird 2001 [160]. À la section 9.6, nous présenterons nos résultats.

9.5.1 Variation vorace : G-LLZW

L'algorithme LZW insiste pour trouver la plus longue concordance exacte entre la séquence à compresser et les sous-séquences contenues dans le dictionnaire. Nous avons décrit l'utilisation de la treille comme structure de donnée pour le dictionnaire. Cette structure nous permet de trouver les concordances de longueur maximale en temps essentiellement linéaire en la longueur de la concordance. L'algorithme commence la recherche de concordance à la racine de la treille, qui est le nœud courant initial — la racine correspondant à la chaîne vide. Pour le prochain symbole dans la séquence, on va chercher un fils du nœud courant dont la clef correspond au prochain symbole dans la séquence. Si ce fils existe, il devient le nouveau nœud courant et la recherche de concordance se poursuit. S'il n'existe pas de fils correspondant au prochain symbole, la plus longue concordance se termine au nœud courant, et l'index du nœud courant est émis.

L'algorithme original est vorace : il fait le seul choix possible un symbole à la fois. La concordance se termine lorsqu'à une étape aucun des fils du nœud courant ne correspond au prochain symbole dans la séquence. Si faire une concordance approximative ne fait aucun sens pour des données comme du texte, une concordance approximative peut être acceptable lorsqu'on parle de pixels. Substituer un pixel d'une certaine couleur par un autre pixel d'une couleur semblable ne nuit pas considérablement à l'image et aide à la compression si cet autre pixel est déjà dans la treille dans le contexte courant.

L'algorithme G-LLZW (pour *greedy lossy LZW*) va modifier la façon de trouver les concordances en ne cherchant plus des concordances exactes mais en trouvant des concordances *tolérables*. En utilisant un paramètre θ qui dépend de la métrique utilisée mais qui est fourni par l'usager, nous définirons une concordance entre deux pixels x et y comme étant θ-tolérable si

$$\|x - y\| \leqslant \theta$$

Les nouvelles concordances sont trouvées en cherchant, à chaque étape, la meilleure concordance qui est θ-tolérable. Ainsi, l'algorithme procède d'essentiellement la même façon pour trouver les concordances, sauf qu'au lieu d'exiger qu'un nœud ait un fils qui soit égal au prochain symbole dans la séquence, nous allons examiner ses fils et choisir celui qui minimise la métrique tout en étant θ-tolérable. Si aucun des fils n'est θ-tolérable, la concordance se termine au nœud courant. Le reste de l'algorithme demeure inchangé. La mise à jour du dictionnaire se fait de la même façon ; même chose pour l'émission et le décodage des index.

Si l'algorithme augmente la longueur moyenne des concordances, il ne garantit pas de trouver la concordance qui minimise les différences entre les pixels de la chaîne contenue dans le dictionnaire et la séquence à compresser. Comme il procède de façon vorace, un mauvais score local peut cacher une chaîne de pixels qui est à la fois plus longue et plus fidèle à la séquence à compresser. Cet algorithme n'est donc pas optimal, mais, comme nous le verrons à la section 9.6, donne de très bons résultats.

9.5.2 Variation Chiang-Po : CP-LLZW

Chiang et Po proposent une variation de cet algorithme en remplaçant la θ-tolérance par une mesure qui dépend de la variance locale dans l'image [46]. Sur une région 3×3 centrée sur le pixel d'intérêt, on calcule la variance σ_Y^2 de la luminosité en Y. On calcule le facteur

$$K = m\sigma_Y^2 + b$$

où m et b sont des constantes choisies. En connaissant $\bar{Y}$, la luminosité moyenne de la région, on calcule la plus petite différence perceptible (en anglais, JND ou *just noticeable difference*) grâce à la formule de Weber[5]. Une fois la JND calculée, on impose la tolérance T telle que

$$T = JND(\bar{Y})K \approx \Delta\bar{Y}(m\sigma_Y^2 + b)$$

Alors que notre première variation utilise un paramètre constant θ qui est donné par l'utilisateur, l'algorithme de Chiang et Po va utiliser pour chaque pixel un seuil T différent calculé grâce à la JND locale et $m\sigma_Y^2 + b$.

Les échelles de m et b dépendent des valeurs de Y obtenues. Si Y est normalisé de façon à être entre 0 et 1, on peut s'attendre à d'assez petites valeurs pour ces deux paramètres. Si au contraire Y est entre 0 et 255, ces paramètres pourront prendre de plus grandes valeurs. Chiang et Po proposent d'utiliser $m = 0.027$ et $b = 0.9$ (pourquoi ? aucune précision à ce sujet) dans [46, tableau 1].

5 La formule de Weber est une observation heuristique qui dit que si ΔI est la plus petite variation perceptible sur un fond de luminosité I, alors $\Delta I/I \approx 0.01$. La constante 0.01 peut aller jusqu'à 0.05 selon les auteurs.

9.5.3 Variation optimisée : P-LLZW

Cette seconde variante va chercher à optimiser un critère global pour trouver la meilleure concordance entre la séquence à compresser et les entrées dans le dictionnaire. Plutôt que de chercher à trouver la meilleure concordance (qui est à la fois la plus longue tout en étant localement la moins différente à chaque étaple) de façon vorace, cette variante va fouiller exhaustivement le dictionnaire de sous-séquences pour trouver celle qui maximise une fonction objectif fournie par l'usager.

Cette fonction objectif peut être utilisée pour choisir les chaînes θ-tolérables les plus longues, ou les chaînes les plus fidèles, ou un compromis entre les deux. La recherche de concordance cherche la chaîne qui maximise la valeur de la fonction objectif tout en exigeant que chaque pixel de la concordance soit θ-tolérable par rapport au pixel correspondant de la séquence à compresser. Puisque la fonction objectif doit être évaluée pour chaque chaîne de pixels contenue dans le dictionnaire, on ne peut plus procéder de façon vorace et n'explorer qu'un chemin dans la treille.

Le nouvel algorithme n'est pas excessivement dispendieux à calculer. On pourrait penser *a priori* qu'il nous faut examiner chaque chaîne dans le dictionnaire et calculer la fonction objectif. Pour un dictionnaire de d entrées, on peut avoir des chaînes d'au plus d symboles de long, laissant entrevoir un coût $O(d^2)$ pour calculer la fonction objectif (en assumant qu'elle prenne un temps linéaire). Si on utilise une fonction objectif séparable, c'est-à-dire qui est telle que

$$f(a : b) = c_1 f(a) + c_2 f(b)$$

pour deux constantes c_1 et c_2 et deux chaînes a et b de longueurs quelconques, on peut calculer la fonction objectif pour toutes les entrées du dictionnaire en temps $O(d)$ grâce à une fouille en profondeur modifiée. Cela nous donne un algorithme de type programmation dynamique.

Il suffit de commencer à la racine et d'accumuler dans les nœuds visitiés les valeurs de $f(\cdot)$ (la fonction objectif) cumulées depuis la racine. À chaque nœud, on verifie si la valeur cumulative de $f(\cdot)$ sur ce chemin est meilleure que la meilleure valeur objectif obtenue jusqu'à présent, et si c'est le cas, on conserve ce nouveau meilleur score et l'index de ce nœud. Après avoir fouillé la treille en profondeur, on a trouvé le meilleur score selon la fonction objectif séparable ; donc la meilleure séquence dans le dictionnaire sous la fonction objectif. Cela résulte en $O(d)$ calculs pour une treille avec d nœuds, donc contenant d entrées.

Une fois cette meilleure chaîne (sous la mesure de la fonction objectif) trouvée, les opérations qui suivent demeurent inchangées : les ajouts de nouvelles chaînes et l'émission des index procèdent de la même façon. On remarque que les deux deux algorithmes proposés (et probablement la variation de Chiang et Po, bien que cela ne soit pas précisé par les auteurs) n'affectent pas la façon dont la version compressée de la séquence est décodée. En effet, rappelons-nous de la façon dont procède LZW pour ajouter de nouvelles chaînes dans le dictionnaire. Le symbole qui sera ajouté à une chaîne D_i est le premier symbole de la chaîne suivante D_j. Or, comme nous attendons de décoder j, donc D_j, avant d'ajouter un symbole à D_i, on s'assure de ne pas propager d'erreurs dûes aux concordances approximatives trouvées par l'un ou l'autre des algorithmes, c'est-à-dire que le symbole ajouté à D_i est toujours bel et bien le premier symbole de D_j.

9.6 Résultats

Pour les tests nous avons utilisés des fichiers provenant du corpus Kodak réduits à 256 couleurs et un *dithering* de Jarvis, Judice et Ninke. Les fichiers *airplane* et *lena* proviennent du corpus USC-SIPI (voir l'appendice D à ce sujet). Nous nous sommes contentés de les réduire à 256 couleurs, sans *dithering*. Ces deux fichiers se retrouvent dans [46] où il n'est fait allusion qu'à la méthode de réduction de couleur (la coupe médiane) et nullement à la méthode *dithering*, d'où on déduit qu'ils n'en ont utilisé aucune.

Le tableau 9.1 donne les tailles originales des fichiers en version GIF choisis pour les tests. Le tableau 9.2 montre les résultats obtenus avec l'algorithme G-LLZW, la variation vorace. L'algorithme optimisant a été testé avec une variété de fonctions objectif. Une fonction objectif devrait équilibrer la longueur de la concordance et la distortion induite ; la longueur comptant positivement et la distortion négativement. En définissant la distortion accumulée par

$$d = \sum_{i=1}^{l} \|x_i - \hat{x}_i\|$$

où l est la longueur de la concordance sous examen. Puisque la la concordance n'est valide que si $\|x_i - \hat{x}_i\| \leqslant \theta$, alors $d \leqslant \theta l$. Mais θ dépend de l'échelle de l'espace de couleur. Pour des composantes $0 \leqslant c < 256$, (où c est r, g ou b), nous avons que $0 \leqslant \theta \leqslant 255\sqrt{\gamma_r^2 + \gamma_g^2 + \gamma_b^2}$. En utilisant YUV, nous arrivons à des bornes différentes, mais que nous utilisions RGB ou YUV, les valeurs utiles de θ vont approximativement de 2 à 10. Nous posons la fonction objectif

$$f(l, d) = l^2 - \frac{1}{10}d$$

de façon à favoriser les longues concordances tout en minimisant la distortion pour des valeurs raisonnables de θ. Le tableau 9.3 présente les résultats obtenus avec la version optimisante. Pour comparer les résultats avec Chiang et Po, nous avons assez peu de données. Chiang et Po présentent leur résultats en $PSNR(dB)$ [46]. Pour *Lena*, version couleur, ils obtiennent une longueur compressée de 147887 octets et un $PSNR$ de 33.59 dB. Pour le même fichier, et pour une longueur compressée semblable, soit 147151 octets (avec $\theta = 4$), nous obtenons un $PSNR$ de 45.00 dB. Pour le fichier *airplane*, ils obtiennent 87891 octets et un $PSNR$ de 34.90 dB alors que l'algorithme P-LLZW donne 86156 octets (avec $\theta = 5$) et un $PSNR$ de 42.50 dB ! Des différences de ≈ 12 dB et $\approx$ $8dB$ sont *très* significatives ! L'algorithme P-LLZW donne une qualité d'image (très) supérieure à l'algorithme CP-LZZW pour une taille compressée comparable.

Les temps de compression sont très raisonnables. L'algorithme $P - LLZW$ est légèrement plus lent que l'algorithme $G - LLZW$ mais les temps de compression sont comparables. L'algorithme $P - LLZW$ pourrait fouiller tout le dictionnaire si on posait $\theta = \infty$, mais puisque la valeur utile de θ est inférieure à 20, on se trouve à élaguer significativement la recherche et n'examiner qu'un relativement petit nombre de chaînes contenues dans le dictionnaire. La compression par l'algorithme $P - LLZW$ est environ deux fois plus lente que la compression par l'algorithme $G - LLZW$.

Comment se comparent les résultats obtenus par les algorithmes G-LLZW et P-LLZW ? Comme les résultats pour les mêmes θ sont différents (autant à la compression qu'en termes de la qualité obtenue) nous allons utiliser la fonction de score

$$s(l, l', q) = q\frac{l}{l'}$$

où l est la longueur originale, l' la longueur compressée et q la qualité en dB. On suppose aussi que $l \approx l'$. Cette mesure donne une idée de la qualité obtenue en fonction du taux de compression. Les résultats sont présentés au tableau 9.4. Notez que les taux de compression n'y sont pas donnés par rapport à la taille de l'image brute originale mais par rapport à la taille du fichier GIF de départ, soit la version en couleurs décimées à 256 couleurs et avec le *dithering* de Jarvis, Judice et Ninke. Voir le tableau 9.1 pour les tailles compressées. L'algorithme P-LLZW permet de gagner sur le taux de compression (d'environ 3%) tout en augmentant la qualité moyenne des images. En effet, la mesure de score comporte une composante logarithmique (les dB) ce qui fait la petite différence de ≈ 0.62 correspond à un effet exponentiel et n'est en fait pas négligeable.

❁ ❁ ❁

La qualité de la reconstruction, mesurée en décibels, ne compte que la qualité de reconstruction point par point et ne tient pas compte de la perception d'une région de l'image. Or, ce que nous observons avec le *dithering*, c'est que les concordances approximatives peuvent quand même produire une modulation des pixels qui apparaît à l'observateur tout à fait raisonnable même si la mesure en décibels laisserait sous-entendre que la qualité est très réduite. Par exemple, les images *duneprints51*, *goldwater67*, *img0010*, *img0019*, *img0031*, *img0056*, *kodim05*, *kodim09* et *kodim23* peuvent subir une compression très élevée avant que la perte de qualité ne soit désastreuse. Considérez l'image *img0056* (fig. 9.7), qui ne montre pas la dégradation à laquelle on eût pu s'attendre en sachant que la qualité de la reconstruction est de 24.30 dB. Elle montre cependant une dégradation, en particulier un bruit de faute fréquence mais de faible amplitude, mais elle ne nuit pas à la qualité de l'image; les hiéroglyphes demeurent tout à fait lisibles et les contours ne soint pas amoindris.

9.7 Conclusion

Dans ce chapitre, nous avons présenté deux variations de l'algorithme LZW qui admettent des pertes. Ces algorithmes, plutôt que de chercher les plus longues concordances exactes entre la séquence à compresser et les sous-séquences contenues dans le dictionnaire, vont accepter d'utiliser des concordances approximatives, en espérant trouver de plus longues concordances et améliorer le taux de compression sans être trop dommageable sur la qualité de l'image. Le premier de ces algorithmes, G-LLZW, procède à la recherche de cette concordance de façon vorace. Pour chaque symbole de la concordance, on vérifie si le pixel de la séquence à compresser correspond à un des pixels dans la treille en admettant au plus une distortion θ spécifiée par l'utilisateur. Cette constante θ dépend de l'espace de couleur utilisé et de l'échelle des composantes de la couleur. Il empêche d'avoir des pixels très mauvais mais limite la longueur des concordances trouvées. Le second algorithme, P-LLZW, optimise un critère fourni par l'usager sur toutes les sous-séquences contenues dans le dictionnaire afin de déterminer la quelle de ces sous-séquences choisir. Cet algorithme, pour chaque concordance, prend un temps linéaire au nombre d'entrées dans le dictionnaire. Ce second algorithme contraint aussi la distortion maximale permissible pour un pixel individuel grâce au paramètre θ.

Nous avons comparé nos deux algorithmes à l'algorithme de Chiang et Po pour montrer qu'ils donnaient des résultats supérieurs, autant en terme de compression que de qualité d'image. Nous avons aussi montré que l'algorithme P-LLZW donne une meilleure qualité d'image que l'algorithme G-LLZW pour le même taux de compression. Les taux de recompression (c'est-à-dire

FIG. 9.7 – L'image *img0056*. À droite, l'image originale. À gauche, la reproduction à 24.30 dB.

Fichier	Taille
airplane	188 630
lena	204 608
balloons108	213 929
duneprints51	376 074
goldwater67	341 537
hibiscus30	240 417
img0010	318 794
img0019	262 812
img0031	348 294
img0036	292 009
img0056	453 443
img0077	262 514
kodakbus93	303 971
kodim05	373 599
kodim09	291 566
kodim23	253 984
leaves40	298 390

TAB. 9.1 – Les fichiers du corpus Kodak utilisés et leurs tailles originales.

par rapport à l'image compressée grâce à la modification GIF de LZW) pour un paramètre θ modéré va à plus de 2 :1, tout en conservant une qualité d'image supérieure à 35 dB.

Nous avons aussi construit une implémentation complète de l'algorithme qui produit des versions compressées décodable par un décodeur GIF standard bien que l'algorithme d'encodage prenne des décisions différentes. En effet, nous avons montré que notre modification de l'algorithme ne changeait pas la façon dont le dictionnaire était mis à jour ni la façon dont sont émis les index. Bien que cela puisse paraître n'être qu'une simple curiosité, le fait d'être compatible au protocole GIF permet de créer des images et de les distribuer sans pour autant fournir un logiciel spécial pour le visionnement. Tous les navigateurs et tous les logiciels de visionnement savent comment décoder et afficher les images GIF. Cela permet à l'algorithme d'être tout à fait applicable pour les applications web.

	$\theta = 5$	$\theta = 10$	$\theta = 15$	$\theta = 20$
Fichier	Taille dB	Taille dB	Taille dB	Taille dB
airplane	102 454	71 069	57 740	49 094
	41.15	34.69	31.54	29.31
lena	137 144	100 653	79 809	68 389
	38.44	31.59	27.88	25.65
balloons108	125 830	81 315	60 489	50 368
	37.02	30.47	27.25	25.42
duneprints51	248 401	197 159	164 158	142 437
	37.99	31.27	27.36	24.93
goldwater67	261 333	203 108	171 834	149 070
	36.84	29.05	25.30	22.76
hibiscus30	168 971	119 947	96 480	82 020
	34.64	27.34	23.74	21.49
img0010	200 533	146 763	120 832	105 554
	35.64	28.62	25.20	23.03
img0019	148 108	105 103	85 351	73 488
	32.45	26.51	23.44	21.12
img0031	244 190	182 497	150 682	129 179
	38.02	31.07	27.61	25.26
img0036	151 677	97 301	74 809	64 203
	38.35	32.51	29.81	28.01
img0056	315 006	256 692	222 364	197 102
	38.43	31.31	27.54	24.90
img0077	196 548	139 829	108 652	90 096
	35.83	28.27	24.55	22.25
kodakbus93	212 675	160 937	133 095	114 449
	36.32	29.52	25.68	23.19
kodim05	283 509	227 408	192 875	169 935
	36.41	28.75	24.75	22.27
kodim09	167 839	115 646	89 662	75 492
	38.31	32.01	28.69	26.61
kodim23	184 967	130 354	101 992	82 529
	38.15	30.22	26.78	24.32
leaves40	206 079	151 921	123 263	104 983
	37.08	30.04	26.56	24.18

TAB. 9.2 – Les résultats de l'algorithme G-LLZW sur diverses images.

	$\theta = 5$	$\theta = 10$	$\theta = 15$	$\theta = 20$
Fichier	Taille dB	Taille dB	Taille dB	Taille dB
airplane	96 471	65 526	53 061	45 597
	40.17	33.89	30.76	28.72
lena	134 688	95 886	74 157	63 246
	37.86	30.91	27.22	25.09
balloons108	119 754	76 627	57 137	47 938
	36.25	29.86	26.73	24.03
duneprints51	244 681	190 434	156 319	134 632
	37.30	30.54	26.58	24.14
goldwater67	260 529	198 678	164 218	140 336
	36.52	28.61	24.72	22.11
hibiscus30	164 258	114 519	90 730	76 577
	34.08	26.88	23.16	20.93
img0010	195 891	140 449	114 914	100 452
	35.05	28.02	24.64	22.40
img0019	141 938	99 705	80 664	69 623
	31.69	25.90	22.83	20.49
img0031	239 618	174 288	141 563	121 386
	37.45	30.36	26.89	24.59
img0036	139 995	89 054	70 239	59 417
	37.30	31.77	29.17	27.38
img0056	312 376	251 182	214 923	189 925
	37.69	30.67	26.85	24.30
img0077	194 760	132 506	99 304	80 901
	35.50	27.64	23.84	21.56
kodakbus93	209 418	156 421	125 732	106 930
	35.81	29.00	25.02	22.53
kodim05	281 994	222 350	185 695	163 527
	35.81	28.25	24.13	21.72
kodim09	159 849	107 236	83 189	70 476
	37.42	31.27	28.01	26.01
kodim23	182 137	124 101	93 870	75 311
	37.65	29.71	26.14	23.69
leaves40	203 032	146 076	117 181	98 292
	36.61	29.52	25.96	23.62

TAB. 9.3 – Les résultats de l'algorithme P-LLZW sur diverses images.

$\theta = 5$	Scores		Ratio	
Fichier	vorace	optimisé	vorace	optimisé
airplane	75.69	78.54	1.84	1.96
lena	57.32	57.51	1.49	1.52
balloons108	62.94	64.76	1.70	1.79
duneprints51	51.52	57.33	1.51	1.54
goldwater67	48.15	47.88	1.31	1.31
hibiscus30	49.30	49.88	1.42	1.46
img0010	56.66	57.04	1.59	1.63
img0019	57.58	58.68	1.77	1.85
img0031	54.28	54.44	1.43	1.45
img0036	73.83	77.80	1.93	2.09
img0056	55.82	54.71	1.44	1.45
img0077	47.86	47.85	1.34	1.35
kodakbus93	51.91	51.98	1.43	1.45
kodim05	47.98	47.45	1.32	1.33
kodim09	66.55	68.25	1.74	1.82
kodim23	52.39	52.05	1.37	1.40
leaves40	53.69	53.80	1.45	1.47
moyennes	56.99	57.61	1.53	1.58

TAB. 9.4 – Les scores et les ratios de recompressions obtenus.

Chapitre 10

Les *pseudo-ondelettes* binaires

10.1 Introduction

Les techniques de décomposition de signaux utilisées pour la compression d'image, comme les transformées trigonométriques ou ondelettes sont conçues pour les images en tons continus et éprouvent des problèmes lorsque le nombre de valeurs que le signal peut prendre est très restreint. En fait, ces méthodes sont à peu de choses près totalement inutiles lorsque l'on doit considérer un signal binaire, par exemple une image textuelle.

Dans l'article *Binary Pseudowavelets and Applications to Bilevel Image Processing*, nous présentons une solution possible à la décomposition de signaux binaires par un mécanisme semblable aux ondelettes mais n'impliquant que des opérations binaires très simples [166]. Ces « pseudo-ondelettes » partagent plusieurs propriétés avec les « vraies » ondelettes, en particulier les propriété de multirésolution et de support compact des bases. Nous avons vu au chapitre 4, à la section 4.1.1.4, ce qu'étaient les ondelettes définies sur les signaux continus et quelques unes de leurs propriétés. Ce chapitre est donc consacré à cette contribution.

La motivation première de cette innovation est de fournir un nouvel outil pour la décomposition multirésolution des images monochromes aux fins de compression et de transmission progressive.

Les images monochromes sont habituellement codées dans le domaine spatial. Soit les répétitions de pixels sont codés grâce à un code de Golomb ou un code de Huffman modifié, comme l'encodage facsimilé CCITT G4 [37], soit les pixels sont soit prédits à l'aide d'un contexte limité, comme avec JBIG où les séries de prédictions correctes et les prédictions erronées sont encodées par un code géométrique mais où les bits sont générés par un codeur arithmétique plutôt que de Huffman [53]. Enfin, il existe les méthodes où l'image (que l'on suppose être du texte) est découpée en régions et compressée en utilisant une concordance approximative et des dictionnaires de glyphes (comme JBIG 2). La méthode des pseudo-ondelettes a pour but de fournir une méthode naturelle de décomposition de l'image et ainsi pouvoir appliquer la compression dans le domaine transformé plutôt que le domaine spatial. Une méthode ayant la propriété de multirésolution ouvre la voie aux méthodes de compression avec perte qui dégradent l'image de façon imperceptible, ou du moins tolérable.

Les pseudo-ondelettes sont définies sur les vecteurs binaires alors que les ondelettes classiques sont définies soit sur l'espace des fonctions réelles soit, en version discrète, sur les vecteurs de réels. Les bases pseudo-ondelettes sont composés de vecteurs de bits que l'on compose non pas avec les opérations arithmétiques classiques mais avec les opérations bit à bit comme le *et* et le *ou-exclusif*. L'utilisation de ces opérations simples mènent à des algorithmes d'une très faible complexité et à des implémentations logicielles ou matérielles d'une grande simplicité nécessitant très peu de ressources.

Nous montrerons comment calculer les transformées en pseudo-ondelettes et présenterons quelques résultats sur des images monochromes. Nous indiquerons que les méthodes pseudo-ondelettes peuvent avoir leur utilité dans le cas des images d'imprimeries produites en *halftoning*. Nous montrerons que les implémentations logicielle et matérielle des transformées requièrent très peu de ressources et pourraient facilement être intégrés dans une future génération d'appareils de télécopie.

Le lecteur remarquera que nous ne donnons aucune référence à des travaux semblables et antérieurs, réalisés par d'autres auteurs. La raison en est qu'il ne semble tout simplement pas y en avoir !

❁ ❁ ❁

Ces contributions ont été présentées en séance plénière de la conférence annelle sur la compression de données, à Snowbird, 1999. Yoshua Bengio, notre coauteur, nous a apporté une aide précieuse par ses conseils, principalement en révisant les manuscrits de l'article [166].

10.2 Définitions

Alors que les ondelettes discrètes sont définies sur $\mathbb{R}^N$, les pseudo-ondelettes sont définies sur $\mathbb{B}^N$, c'est-à-dire les vecteurs de bits de longueur N. La propriété la plus intéressante des ondelettes est sans conteste la propriété de multirésolution qui garantit que, à chaque fois que nous ajoutons un coefficient ondelette, la qualité de la reconstruction obtenue augmente uniformément (voir section 4.1.1.4 pour la définition de multirésolution). Dans le cas des pseudo-ondelettes, comme pour les ondelettes classiques, les vecteurs de base ont des supports d'une longueur qui est inversement proportionnelle à la fréquence associée. Pour les pseudo-ondelettes, cela implique — nous verrons pourquoi dans un instant — que les vecteurs contiennent de moins en moins de uns au fur et à mesure que la fréquence augmente. Le vecteur associé à la fréquence zéro (qui correspond au terme DC) peut être remplis de uns, mais tous les autres vecteurs en contiendront de moins en moins, jusqu'à n'en contenir que $O(\frac{1}{N})$ pour les vecteurs associés aux plus hautes fréquences.

Comme pour la transformée de Haar, la fréquence du f^{e} vecteur est proportionnelle à p, selon la décomposition $f = 2^p + q - 1$, donnée par l'éq. (4.12), page 49. Comme pour les transformées discrètes, nous utiliserons un produit matrice-vecteur pour obtenir les coefficients pseudo-ondelettes, mais un produit d'un type bien particulier que nous définirons sous peu. D'abord, présentons les définitions et conventions nécessaires.

Définition 10.2.1 Et logique. Pour $a, b \in \mathbb{B}$, $a \wedge b$ est le *et* logique de a et b. L'opérateur $\wedge$ est calculé selon la grille suivante :

$\wedge$	0	1
0	0	0
1	0	1

Définition 10.2.2 Ou-exclusif. Pour $a, b \in \mathbb{B}$, $a \oplus b$ est le *ou exclusif* logique de a et b. L'opérateur $\oplus$ génère la table suivante :

$\oplus$	0	1
0	0	1
1	1	0

Définition 10.2.3 Vecteur binaire. Pour $\vec{v} \in \mathbb{B}^N$, $\vec{v} = (v_0, v_1, \ldots, v_{N-1})$ tel que $v_i \in \mathbb{B}$, $\vec{v}$ est un vecteur binaire.

Définition 10.2.4 Opérateur compteur. Pour $\vec{v} \in \mathbb{B}^N$, $|\vec{v}|_1$ donne le nombre de uns contenus dans $\vec{v}$, pareillement $|\vec{v}|_0$ donne le nombre de zéros, et $|\vec{v}|_0 + |\vec{v}|_1 = N$. Nous avons donc :

$$|\vec{v}|_x = \sum_{i=0}^{N-1} (v_i = x)$$

où $(a = b)$ est une fonction indicatrice qui vaut 1 si $a = b$ et 0 sinon.

Définition 10.2.5 Opérateur transition. Pour $\vec{v} \in \mathbb{B}^N$, $|\vec{v}|_T$ donne le nombre de transitions de bits dans $\vec{v}$, soit le nombre de fois que les bits passent de 0 à 1 ou vice-versa. Les extrémités ne comptant pas pour des transitions, le nombre de transitions est donné par

$$|\vec{v}|_T = \sum_{i=1}^{N-1} (v_i \neq v_{i-1})$$

où $(a \neq b)$ est une fonction indicatrice qui vaut 1 si $a \neq b$ et 0 sinon.

Définition 10.2.6 Matrice binaire. Une matrice binaire $W \in \mathbb{B}^{N \times N}$ est une collection de N vecteurs de $\mathbb{B}^N$. Une matrice binaire est composée de vecteurs-rangées $\vec{w}_i$, $i = 0, \ldots, N-1$.

Définition 10.2.7 Opérations sur les vecteurs. Soient $\vec{a}, \vec{b} \in \mathbb{B}^N$ et $c \in \mathbb{B}$. Alors :

- $\vec{a} \wedge c = (a_0 \wedge c, a_1 \wedge c, \ldots, a_{N-1} \wedge c)$
- $\vec{a} \oplus c = (a_0 \oplus c, a_1 \oplus c, \ldots, a_{N-1} \oplus c)$
- $\vec{a} \wedge \vec{b} = (a_0 \wedge b_0, a_1 \wedge b_1, \ldots, a_{N-1} \wedge b_{N-1})$
- $\vec{a} \oplus \vec{b} = (a_0 \oplus b_0, a_1 \oplus b_1, \ldots, a_{N-1} \oplus b_{N-1})$

Définition 10.2.8 Opérations sur les matrices. Pour $V, W \in \mathbb{B}^{N \times N}$ et $\vec{c} \in \mathbb{B}^N$, nous avons :

- $V * \vec{c} = (\vec{v}_0 \wedge c_0) \oplus (\vec{v}_1 \wedge c_1) \oplus \cdots \oplus (\vec{v}_{N-1} \wedge c_{N-1})$
- $V * W = (\vec{v} * \vec{W}_0, \vec{v} * \vec{W}_1, \ldots, \vec{v} * \vec{W}_{N-1})$

À partir de ces définitions d'opérations de base sur les vecteurs et les matrices binaires, nous somme prêts à énoncer les théorèmes suivants :

Théorème 10.2.1 Existence de bases. Une matrice $V \in \mathbb{B}^{N\times N}$ est une base sur $\mathbb{B}^N$ sous l'opérateur $*$, si, et seulement si, $\forall\, \vec{x} \in \mathbb{B}^N, \exists!\, \vec{x}' \in \mathbb{B}^N$ t.q. $V * \vec{x}' = x$.

Théorème 10.2.2 Existence de l'inverse. Si $V \in \mathbb{B}^{N\times N}$ est une base sur $\mathbb{B}^N$, alors son inverse est donné par $V^{-1} \in \mathbb{B}^{N\times N}$, tel que pour $\vec{v}_i^{-1}$, $i = 0, \ldots, N-1$, $V * \vec{v}_i^{-1} = I_i$, où I_i est la i^{e} rangée de la matrice identité (en respectant la convention que la numérotation des rangées commence à zéro). V^{-1} est aussi une base sur $\mathbb{B}^N$ sous l'opérateur *. V^{-1} est bel et bien l'inverse de V car $V^{-1} * V = I$ et $V^{-1} * (V * \vec{x}) = (V^{-1} * V) * \vec{x} = I * \vec{x} = \vec{x}$. Cela se démontre à partir des définitions déjà énoncées.

Corollaire 10.2.1 $V \in \mathbb{B}^{N\times N}$ est une base, si, et seulement si, $\exists V^{-1} \in \mathbb{B}^{N\times N}$ tel que $V^{-1} * V = I$.

Définition 10.2.9 Fréquences d'un vecteur binaire. Deux vecteurs de même longueur $\vec{v}$ et $\vec{w}$ sont de même fréquence si, et seulement si, $|\vec{v}|_T = |\vec{w}|_T$.

Définition 10.2.10 Base pseudo-ondelette. Une matrice W est une base sur $\mathbb{B}^N$ et est une base pseudo-ondelette si W^{-1} existe et satisfait :

1. Tous les vecteurs-rangées de W^{-1} sont *localisés*, sauf possiblement le vecteur de fréquence zéro. Un vecteur $\vec{w}$ est localisé si $|\vec{w}|_T \leqslant 2$.
2. Tous les vecteurs-rangées ont un nombre de uns proportionnel à la fréquence associée selon la décomposition de l'éq. (4.12).
3. La base possède la propriété de multirésolution. Soit $H(\vec{v}, \vec{w}) = |\vec{v} \oplus \vec{w}|_1$, la distance de Hamming, soit le nombre d'endroits où deux vecteurs $\vec{v}$ et $\vec{w}$ de longueur égale N diffèrent. Soit $\vec{m}_i$, le vecteur débutant par i uns suivis de $N - i$ zéros, le vecteur « masque ». Soit $\vec{x}' = W * \vec{x}$, la transformée de $\vec{x}$. Une base est multirésolution, au sens des pseudo-ondelettes, si, et seulement si, $\forall\, \vec{x}$, nous avons

   $$N \geqslant H\left(W^{-1} * (\vec{x}' \wedge \vec{m}_1), \vec{x}\right) \geqslant H\left(W^{-1} * (\vec{x}' \wedge \vec{m}_2), \vec{x}\right) \cdots \geqslant H\left(W^{-1} * (\vec{x}' \wedge \vec{m}_N), \vec{x}\right) = 0$$

   c'est-à-dire qu'à chaque coefficient ajouté à la reconstruction, la qualité de celle-ci n'empire pas ; elle s'améliore progressivement jusqu'au moment où, tous les coefficients étant présents, on retrouve la reconstruction exacte.

La dernière contrainte de la définition 10.2.10 peut être relaxée dépendamment du type de reconstruction progressive que l'on désire. Il n'est peut-être pas nécessaire d'avoir une base qui est strictement multirésolution. On pourrait par exemple utiliser une autre fonction que $H(\cdot,\cdot)$ pour calculer l'erreur de reconstruction, une fonction qui tiendrait compte du contexte dans lequel la reconstruction est faite. Pour une fonction F adéquate, nous pourrions exiger que

$$cN \geqslant F\left(W^{-1} * (\vec{x}' \wedge \vec{m}_1), \vec{x}\right) \geqslant F\left(W^{-1} * (\vec{x}' \wedge \vec{m}_2), \vec{x}\right) \cdots \geqslant F\left(W^{-1} * (\vec{x}' \wedge \vec{m}_N), \vec{x}\right) \geqslant 0$$

pour une constante $c \in \mathbb{R}^*$ qui dépend de F. Cette condition pourrait être même relaxée davantage pour devenir

$$cN \geqslant \mathrm{E}\left[F\left(W^{-1} * (\vec{x}' \wedge \vec{m}_1), \vec{x}\right)\right] \geqslant \mathrm{E}\left[F\left(W^{-1} * (\vec{x}' \wedge \vec{m}_2), \vec{x}\right)\right] \cdots \geqslant \mathrm{E}\left[F\left(W^{-1} * (\vec{x}' \wedge \vec{m}_N), \vec{x}\right)\right] \geqslant 0$$

où $E[\cdot]$ est bien entendu l'espérance, en prenant soin de poser des hypothèses raisonnables sur la distribution des $\vec{x}$.

Pour calculer une matrice W nous commençons habituellement par trouver la matrice inverse, W^{-1}. Puisque c'est la propriété de multirésolution qui nous intéresse le plus, on commence par choisir le type de reconstruction qui nous sied le mieux, c'est-à-dire les vecteurs-rangées de W^{-1}, puis on calcule W. Le théorème 10.2.2 suggère peut-être l'utilisation d'une exploration combinatoire pour trouver la matrice inverse (dans notre cas, on s'intéresse à l'inverse de l'inverse, soit la transformée originale). Or, une méthode naïve mène immédiatement à des algorithmes $O(N2^N)$ qui sont parfaitement impraticables même pour de petits N. Or, une modification à l'algorithme d'élimination Gauss-Jordan nous donne un algorithme pour calculer l'inverse en $O(N^3)$.

Les modifications à l'algorithme Gauss-Jordan sont minimales. On initialise l'algorithme en utilisant deux matrices, la matrice de gauche contiendra W^{-1} et la matrice de droite contiendra la matrice identité d'ordre N. La soustraction d'un vecteur par un autre y est remplacée par le *ou-exclusif* entre deux vecteurs-rangées, et le t^{e} pivot (en commençant à $t = 0$) est choisi de façon à ce que le vecteur le contenant soit le vecteur de la matrice de gauche contenant le plus de uns, et tel que le premier un de ce vecteur soit à la position t. Si un autre vecteur-rangée a un 1 à la position t on lui « soustrait » le vecteur-rangée contenant le pivot ; de même pour la matrice de droite. Comme pour l'algorithme de Gauss-Jordan, l'inverse est calculée (ou une permutation de celle-ci) lorsque la matrice de gauche contient l'identité (ou une permutation de celle-ci). Il suffit alors de réorganiser les matrices pour obtenir à gauche l'identité en permutant les rangées des deux matrices simultanément. La matrice de droite contiendra alors l'inverse de W^{-1}, soit W.

10.3 Exemples d'analyses et de reconstructions

Présentons maintenant quelques exemples d'analyses et de reconstructions calculées grâce aux transformées pseudo-ondelettes. Pour garder les choses simples, nous limiterons N à 8. Bien que le choix des bases dépende de l'application considérée, nous définissons les matrices W et

W^{-1} :

$$W = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad W^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Notez la structure de W^{-1}. Les vecteurs de base contiennent de moins en moins de uns, et la résolution double à chaque groupe de vecteurs. Cette base nous permet d'obtenir des approximations à un $\frac{1}{8}^{\text{e}}$ de la résolution, à un $\frac{1}{4}^{\text{e}}$ de la résolution, à une demie et enfin à pleine résolution. La première rangée de la matrice inverse nous permet d'avoir un premier bit transformé qui donne la luminosité « moyenne » du vecteur. Les autres vecteurs apportent des raffinements successifs. Voyons un exemple concret. Posons $\vec{x} = (0,1,0,0,0,1,1,1)$. Calculons $W * \vec{x}$:

$$W * \vec{x} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{array}{cccccccccc} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \oplus & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \oplus & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ \oplus & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline = & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \vec{x}'$$

Le vecteur résultant est $\vec{x}' = (0,0,1,0,0,1,0,0)$. De la même façon, nous lui faisons subir la transformation inverse, $W^{-1} * \vec{x}' = x$:

$$W^{-1} * \vec{x}' = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{array}{cccccccccc} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \oplus & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \oplus & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline = & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \vec{x}$$

Le concept de multirésolution s'explique peut-être mieux lorsqu'on considère une progression dans le temps. Supposons que pour les besoins de la démonstration, le receveur et l'émetteur se soient mis d'accord pour une base pseudo-ondelette quelconque, disons W. Au temps $t = 0$ l'émetteur calcule $W * \vec{x} = \vec{x}'$. Au temps $t = 1$, l'émetteur envoie le bit $\vec{x}'_0$ au receveur qui calcule une première approximation de $\vec{x}$, $W^{-1} * (\vec{x}' \wedge m_1)$. Au temps $1 < t \leqslant N$, le receveur aura une approximation $W^{-1} * (\vec{x}' \wedge m_t)$, à partir des t bits qu'il connaît de $\vec{x}'$. Le vecteur qu'il connaît contient les t premiers bits de $\vec{x}'$ et les $N - t$ bits suivants sont à zéro. Les bits à zéro font en

sorte que les vecteurs de base qui leur sont associés ne participent pas à la reconstruction. Enfin, lorsque $t = N$, $W^{-1} * (\vec{x}' \wedge m_N) = \vec{x}$ et le receveur peut reconstruire exactement $\vec{x}$.

La base pseudo-ondelette peut être choisie de façon à satisfaire des contraintes spéciales selon l'application. Par exemple, lorsqu'on voudra compresser les images monochromes, on voudra que la reconstruction soit visuellement plaisante. Pour les images, qui sont en deux dimensions, on aura besoin de transformées à deux dimensions. À cette fin, l'image sera découpée en tuiles de $N \times N$ sur lesquelles on appliquera la transformée. Soit on opte pour un tenseur de transformation pour deux dimensions, soit on opte pour une transformée séparable. Une transformée séparable est une version de la transformée où, plutôt que d'utiliser un tenseur, on applique la transformée d'abord sur les colonnes de la matrice puis sur les rangées de la matrice composée des colonnes transformées (ou vice-versa) comme on le fait avec les transformées de Hartley ou la DCT (voir les sections 4.1.1.2 et 4.1.1.3 à ces sujets). Plusieurs schèmes de compression d'image utilisent cette astuce pour se dispenser d'avoir une transformée trop complexe. Si $P \in \mathbb{B}^{N \times N}$ est une tuile de l'image, alors la tranformée de P est donnée par $U * (U * P)^{\mathsf{T}}$, où A^{T} est la transposée de A. Le produit $U * P$ est défini par le produit rangée par rangée de P par U (voir définition 10.2.8).

La base U est définie par

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

La fig. 10.1 montre l'homme de Vitruvius transformé et reconstruit progressivement avec la base U en deux dimensions et une transformée séparable (on ne voit que la portion du visage obtenue par seuillage à partir d'une image originale en couleur, voir fig. 4.6, p. 53).

Les bases pseudo-ondelettes pourraient être calculées pour être compatibles avec les images en tons simulés (*halftoning*) selon le même procédé que l'on retrouve avec les photographies et autres images imprimés dans les journaux ou sur les affiches. Alors que la représentation d'un document en *halftones* qui a été numérisé pose plusieurs problèmes (comme estimer le pas entre les points [63, 64]) on peut espérer qu'en moyenne l'image ne sera pas trop affectée si la longueur d'onde des ondelettes ne correspond pas au pas de *halftoning*. Cependant, dans les applications d'impression haute qualité, on a souvent la chance de générer le *halftoning* donc d'en connaître tous les paramètres, comme l'orientation, la forme des points et la distance entre ceux-ci. On peut donc calculer la longueur d'onde adéquate plus facilement. Par exemple, aux figs. 10.2 et 10.3, on trouve une image en *halftones* numérisée à 300 points par pouce. Cette image est reconstruite progressivement grâce à la base W. Bien que la base W n'ait pas été spécialement conçue pour les grilles de points de *halftoning*, la dégradation de l'image semble minimale.

10.4 Généralisations et recherche de nouvelles bases

Alors qu'une taille de tuile 8×8 est fort pratique, on peut quand même vouloir travailler sur des tuiles plus larges, voire même de dimension arbitraire. Pour ce faire, il faut obtenir les

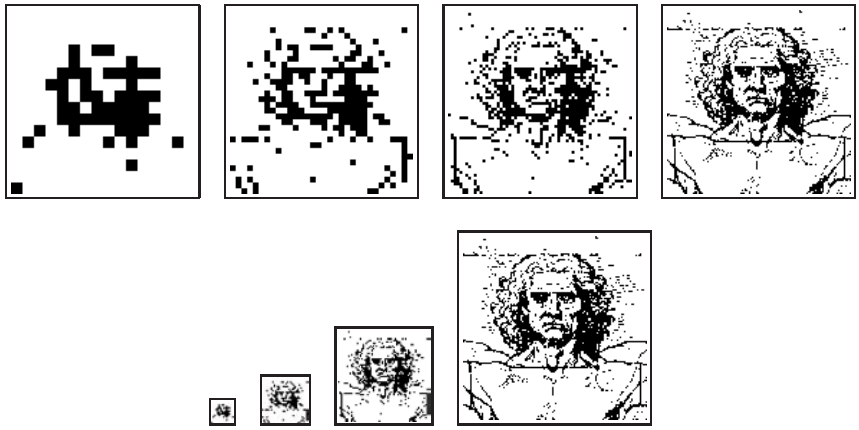


FIG. 10.1 – La figure « homme de Vitruvius » reconstruite avec la base U en deux dimensions (en version séparable), avec, de gauche à droite, 1, 4, 16 et tous les coefficients. Découpée en tuiles de 8×8, l'image transformée aura 64 coefficients. La rangée du bas montre les mêmes images, mais aux échelles correspondantes.

générateurs des matrices de transformation. Comme nous l'avons fait pour la matrice de Haar (voir encore la section 4.1.1.4 et les équations (4.12) et (4.13), nous utiliserons ces fonctions pour générer des matrices de la taille désirée. Pour la matrice U, par exemple, nous trouvons

$$U_n(f,t) = \begin{cases} 1 & \text{si } (n-1-t) \wedge (f \bmod n = \frac{1}{2}(2-n+t)n) \\ 1 & \text{si } (t \bmod 2^{\alpha(n,n-1)-\alpha(n,t)} = f \text{ div } 2^{\alpha(n,t)+1}) \wedge (f \bmod 2^{\alpha(n,t)} = 0) \\ 0 & \text{sinon} \end{cases}$$

et

$$U_n^{-1}(f,t) = U_n^{-1}((p,q),f) = \begin{cases} 1 & \text{si } (f=1) \wedge t \geqslant \frac{1}{2}n \\ 1 & \text{si } (f \neq 1) \wedge \left(\frac{nq}{2^p} \leqslant t < \frac{n(q+1)}{2^p}\right) \\ 0 & \text{sinon} \end{cases}$$

avec p et q tels que $2^p + q = f$, où p est le plus grand entier non-négatif satisfaisant l'égalité, avec les exceptions $f = 0 \to p = 0, q = 0$ et $f = 1 \to p = 0, q = 1$. La fonction $\alpha(n,t)$ est donnée par

$$\alpha(n,t) = \begin{cases} 0 & \text{si } 0 \leqslant t < \frac{n}{2} \\ 1 & \text{si } \frac{n}{2} \leqslant t < \frac{3n}{4} \\ 2 & \text{si } \frac{3n}{4} \leqslant t < \frac{7n}{8} \\ \vdots & \\ \lceil \lg n \rceil - 1 & \text{si } \frac{(n/2-1)n}{n/2} \leqslant t < \frac{(n-1)n}{n} \end{cases}$$

Cette formulation, quoique plus laborieuse, ressemble à ce que nous avions pour les ondelettes de Haar. Cela nous permet de trouver des matrices U_n (pour n une puissance de deux) de la taille désirée. Dans [166] nous présentons les matrices U_{16} et U_{16}^{-1}.

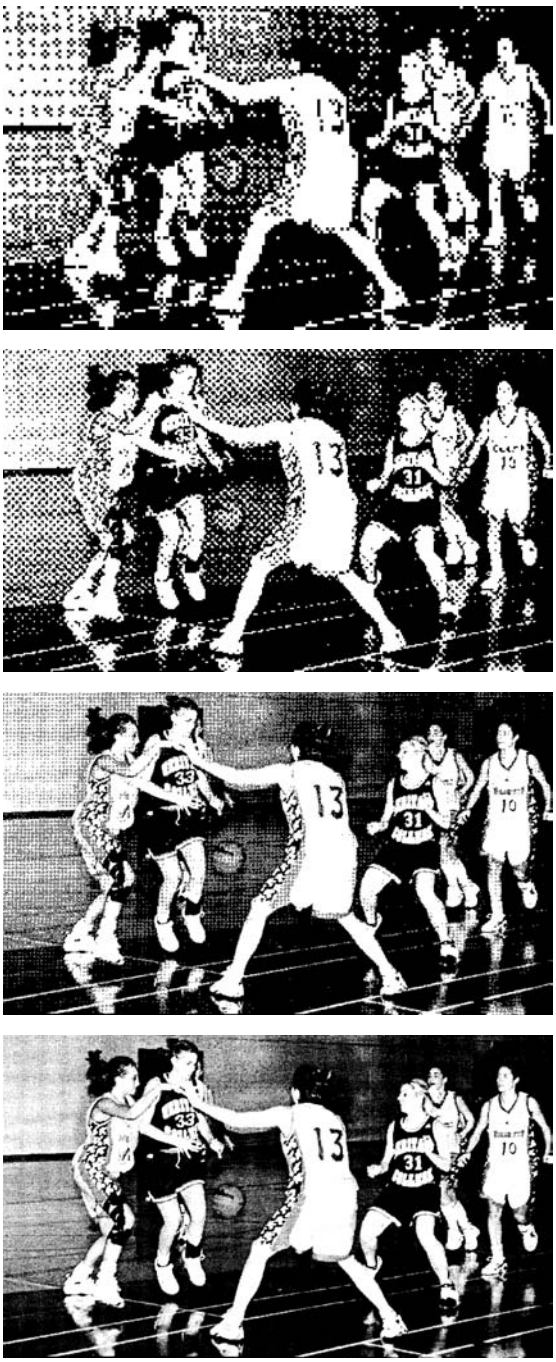


FIG. 10.2 – L'image « basket » reconstruite avec la base W en deux dimensions, avec, de haut en bas, 1, 4, 16 et tous les coefficients. En tuiles de 8×8, l'image comporte 64 coefficients.

FIG. 10.3 – L'image « basket » reconstruite avec la base W en deux dimensions, avec, de gauche à droite, 1, 4, 16 et tous les coefficients. En tuiles de 8×8, l'image comporte 64 coefficients. Cette figure montre les images aux échelles correspondantes.

しかしながら、１９５６年９月に敷設された大西洋横断電話ケーブルは、大陸間電話通信の自動化および半自動化への技術的可能性を与え、ＣＣＩＴＴがこの問題を取り上げるに及び、ＣＣＩＴＴの性格は漸次、汎世界的色彩を実質的に帯びるに至った。この汎世界的性格は第２次世界大戦後目ざましくなったアジア・アフリカ植民地の独立に伴ってＩＴＵの構成員の中にこれらの国が加わり、ＩＴＵの中に新しい意見が導入されたことにも起因して、技術面、政治面の双方から導入されてき

しかしながら、１９５６年９月に敷設された大西洋横断電話ケーブルは、大陸間電話通信の自動化および半自動化への技術的可能性を与え、ＣＣＩＴＴがこの問題を取り上げるに及び、ＣＣＩＴＴの性格は漸次、汎世界的色彩を実質的に帯びるに至った。この汎世界的性格は第２次世界大戦後目ざましくなったアジア・アフリカ植民地の独立に伴ってＩＴＵの構成員の中にこれらの国が加わり、ＩＴＵの中に新しい意見が導入されたことにも起因して、技術面、政治面の双方から導入されてき

FIG. 10.4 – L'image test #7 du CCITT, reconstruite avec la base W en deux dimensions, avec, de gauche à droite, 1, 4, 16 et tous les coefficients. En tuiles de 8×8, l'image comporte 64 coefficients. Cette figure montre les images aux échelles correspondantes.

❁ ❁ ❁

Une autre question intéressante survient : comment calculons-nous de nouvelles bases pseudo-ondelettes ? Une façon serait de placer les bits à la main dans une matrice de la taille désirée et de faire en sorte de vérifier qu'il s'agit bel et bien d'une matrice de base pseudo-ondelettes. Un programme peut aisément vérifier si la base respecte toute les contraintes, et s'il est intéractif, il peut fournir en temps réel à l'utilisateur des informations sur la matrice, par exemple quels bits rendent le calcul de l'inverse impossible. Nous avons monté rapidemenet une petite application qui permet de construire une matrice 8×8. L'application indique immédiatement à l'utilisateur lorsque sa matrice forme une base ou, contrairement, lorsqu'un changement fait en sorte qu'elle cesse de l'être. Cette méthode semble pénible mais en fait il devient facile après un moment de trouver des bases pseudo-ondelettes parce que l'application répond immédiatement. En effet, on comprend assez rapidement comment les bases sont structurées et il devient aisé d'en construire. La fig. 10.5 montre l'application interactive *BasisFinder* que nous avons programmé à cette fin.

Une autre façon serait de procéder à une exploration combinatoire de l'espace des bases. La recherche brutale n'est envisageable que pour des N très petits mais on peut limiter significativement le nombre de combinaisons examinées en présélectionnant les vecteurs de base en imposant des conditions qui limitent grandement leurs nombres. Essayer toutes les combinaisons de bits nous forcera à examiner $O(2^{N^2})$ combinaisons, essayer de trouver des matrices en changeant un vecteur complet à la fois regardera $O(\binom{2^N}{N})$ combinaisons, ce qui est déjà beaucoup mieux. Mais choisir $m \propto N$ vecteurs nous donne $O(\binom{m}{N})$ combinaisons à examiner. L'algorithme final est $O(\binom{m}{N}N^3)$ car il suffit de vérifier que l'inverse existe pour valider l'ensemble de vecteurs. En fait, on peut davantage élaguer la recherche en vérifiant au fur et à mesure si les vecteurs qu'on ajoute à la matrice sont linéairement indépendants, ou encore que deux vecteurs de même fréquences se chevauchent, etc. Une fois que la recherche a produit un ensemble de bases pseudo-ondelettes, il ne reste qu'à choisir la quelle parmi celles-ci sera utilisée.

Cette méthode de recherche suggère un protocole d'échange de bases. Supposons que l'émetteur et le receveur doivent se transmettre une image codée avec les pseudo-ondelettes. Plutôt que d'avoir un grand nombre k de bases connues, il se seraient mis d'accord sur un ensemble de m vecteurs de base. Or, comme mN bits est vraisemblablement beaucoup plus petit que kN^2 bits, il est profitable de ne stocker que cette table de m vecteurs plutôt que les bases elles-mêmes. Pour identifier quelle base sera choisie, l'émetteur transmet un code pour une des $\binom{m}{N}$ combinaisons possibles, grâce à un code énumératif qui ne nécessite que $O(\lg \binom{m}{N})$ bits (voir section 5.3.4.4, sur le sujet et l'appendice C pour estimer $\lg \binom{n}{k}$). Le code énumératif est plus efficace qu'envoyer un vecteur de m avec N bits à un car $\lg \binom{m}{N} \leqslant m$.

10.5 Implémentations efficaces en logiciel et en matériel

Si on prévoit utiliser une taille de tuile réduite, par exemple $N = 8$ ou $N = 16$, on peut se dispenser d'une implémentation logicielle qui calcule les transformées individuellement. On peut remplacer la transformation d'un vecteur de N bits par une simple référence dans une table de 2^N éléments où on aura précalculé, à l'adresse i, la valeur $B * \vec{i}$, où B est la matrice de transformationet $\vec{i}$ est le « vecteur » binaire i. Si par malchance la longueur des vecteurs est trop grande pour l'utilisation de tables, on peut toujours trouver l'algorithme de transformée

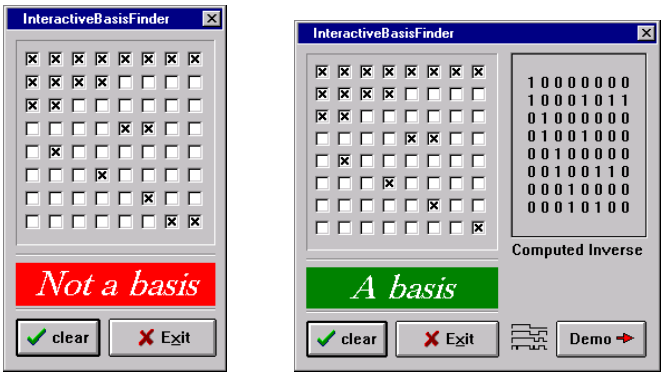


FIG. 10.5 – L'application *BasisFinder* qui permet de trouver interactivement de nouvelles bases pseudo-ondelette. L'usager est informé immédiatement lorsqu'une nouvelle base est découverte ou, au contraire, si les changements effectués font en sorte que la matrice cesse d'être une base. On voit les deux états de l'application. Quand une base est trouvée, l'application donne l'inverse.

rapide pour notre matrice.

Une transformée rapide peut être obtenue en logiciel mais c'est plus difficile qu'obtenir une transformée efficace en matériel. Si l'ordinateur qui exécute le code peut traiter des mots de p bits, il est facile d'obtenir une transformée qui demande $O(\frac{1}{p}N^2)$ opérations. Lorsque $p = N$, la transformée se calcule en temps linéaire. Lorsque p est plus petit que N, nous devons vérifier si la matrice est suffisamment ténue pour extraire individuellement les $O(N \lg N)$ bits et utiliser les mêmes astuces que la simplification d'expressions arithmétiques telle que réalisée par les compilateurs. Lorsque la matrice est très ténue, et que N est très grand, on peut arriver à un calcul en $O(N \lg N)$.

En ce qui concerne les implémentations en matériel, les circuits nécessaires pour la transformée et son inverse vont demander un nombre de portes logiques au plus $O(N \lg N)$ et auront une profondeur maximale $O(\lg \lg N)$. Puisque nous avons affaire à des bits comme éléments des vecteurs plutôt que des entiers ou des nombres en virgule flottante, tout le circuit peut être construit à partir d'un seul type de portes logiques : le *ou-exclusif*. En effet, les *et* logiques peuvent être éliminés trivialement. Il est inutile de calculer le et logique d'un bit toujours à un et d'un bit de l'entrée ; le résultat du calcul est entièrement déterminé par le bit d'entrée. Ainsi, à chaque entrée d'une porte ou-exclusif, on trouve directement le bit d'entrée plutôt qu'une cascade à travers un et logique. Comme les et logiques n'interviennent qu'au niveau des entrées des ou-exclusifs, ils peuvent être complètement éliminés.

Si on admet que la matrice de transformation contient $O(N \lg N)$ bits à un, on peut supposer qu'une colonne contient $O(\lg N)$ bits en moyenne. Pour calculer le *ou-exclusif* de cette colonne, on aura besoin d'un circuit de $O(\lg N)$ portes mais d'une profondeur de $O(\lg \lg N)$ portes. Pour calculer la série de *ou-exclusifs* $a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_k$, on peut utiliser un parenthésage du type $(\cdots((a_0 \oplus a_1) \oplus (a_2 \oplus a_3)) \oplus \cdots \oplus ((a_{k-3} \oplus a_{k-2}) \oplus (a_{k-1} \oplus a_k))\cdots)$, ce qui nous donne une profondeur de parenthèses de $O(\lg k)$. Un tel ensemble de parenthèses correspond à un

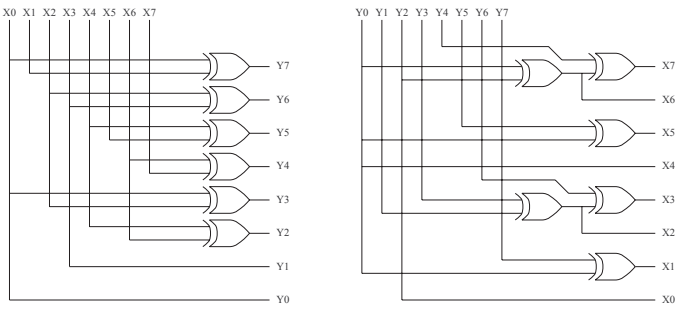


FIG. 10.6 – Diagrammes des circuits logiques pour la transformée U (à gauche) et son inverse U^{-1} (à droite) étant donnée une longueur $N = 8$.

arbre plein qui a toutes ses feuilles sur au plus deux profondeurs différentes. De plus, il n'est pas exclu que le circuit puisse réutiliser des sous-expressions communes, comme on a fait à la fig. 10.6.

L'extrême simplicité des circuits résultants permettrait d'implémenter un circuit qui calcule directement la transformée d'une tuile 8×8 (ou 16×16) à même un microprocesseur, sans changer significativement le nombre total de transistors. En fait, il est à parier que le nombre de portes logiques qui seraient nécessaires pour charger les registres et récupérer les valeurs sera beaucoup plus grand que le nombre de portes logiques contenues dans le circuit de transformation lui-même. Pour le cas des images en format fax (encore, [37]) chaque ligne comporte 1728 pixels, et un circuit qui transformerait une ligne complète demanderait ≈ 20000 portes logiques, ce qui n'est rien selon les standards courants. Il serait très concevable d'inclure cette fonctionnalité à un télécopieur, soit grâce à un microprocesseur augmenté, soit à un circuit de type EPLD ou FPGA s'il le faut vraiment.

10.6 Directions futures

Il reste plusieurs points à éclaircir. Par exemple, si on veut utiliser des pseudo-ondelettes pour analyser une image en *halftoning*, il nous faut connaître la grosseur et les distances entre les points pour construire une base ondelette adéquate. C'est nécessaire pour qu'une reconstruction progressive corresponde à des versions en *halftones* progressivement raffinées de la même image. Si on a accès à l'image originale avant le processus de *halftoning*, on peut calculer exactement tous les paramètres nécessaires pour les pseudo-ondelettes. Si, par contre, on se retrouve dans la situation où les documents à compresser sont numérisés, on doit estimer les paramètres du *halftoning*, comme l'orientation de la grille et sa densité, la taille des points, etc. Certains auteurs se sont penchés sur le sujet, mais dans le cadre d'une compression prédictive [63, 64].

Les pseudo-ondelettes binaires fourniraient peut-être une façon naturelle de transmettre des images monochromes en tenant compte de la bande passante disponible ou désirée, dans un schème de type QOS (*Quality of Service*) où le client décide, en fonction des capacités de son matériel — ou de son portefeuille. Les PDA (*personal digital assistant*) et les téléphones cellulaires ne disposent, la plupart du temps, que d'écrans monochromes, d'assez faible résolution, de technologie de type cristaux liquides. Les PDA Palm, par exemple, ont un écran de 160×160

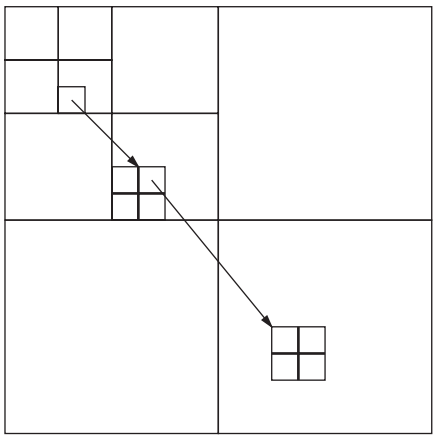


FIG. 10.7 – Les bandes de coefficients obtenues par les transformées ondelettes. Les coefficients sont naturellement organisés de façon à ce qu'un coefficient de fréquence t soit le parent d'un certain nombre de coefficients de fréquence $2t$. Dans la figure, on voit une lignée de ces coefficients, depuis les coefficients de basse fréquence jusqu'aux coefficients de très haute fréquence.

pixels. Un téléphone cellulaire pourait avoir un écran de peut-être 80×24 pixels. Les images qu'ils affichent doivent être conçues ou adaptées en conséquence. Avec un schème pseudo-ondelette, l'un pourrait reconstruire l'image avec moins de niveaux que l'autre ; donnant ainsi une image satisfaisante sur les deux appareils.

Les méthodes de compression d'image à partir de décomposition en fréquences, qu'il s'agisse de transformées dites trigonométriques ou ondelettes, obtiennent des taux de compression intéressants car ils discrétisent les coefficients résultants de la transformation et jettent des bits à la poubelle. Si une transformée trigonométrique nous retourne des coefficients réels, une transformée pseudo-ondelette ne nous donne que des bits individuels en guise de coefficients. On peut difficilement parler de discrétisation ; il s'agit de déterminer quels bits détruire et comment encoder l'information contenue dans les bits restant. Il est évident qu'une technique de compression des bits basée sur un code de Huffman, par exemple, est vouée à l'échec (voir, par exemple, la section 6.3.3 pour comprendre pourquoi). Il devient nécessaire d'utiliser le codage arithmétique et des prédicteurs sur les coefficients pour parvenir à compresser l'image, même si on décide de garder tous les bits.

Les coefficients pseudo-ondelettes sont correlés de façon pyramidale, comme les coefficients des autres transformées, on peut extraire de l'information de contexte à partir de la valeur du parent des coefficients. Considérez la fig. 10.7. La magnitude d'un coefficients obtenu d'une transformée ondelette classique est fortement correlé avec la magnitude de son coefficient parent et de ses coefficients descendants : il en va de même pour les pseudo-ondelettes. Par exemple, avec la base W, si un bit est à 1, c'est qu'au moins deux de ses descendants sont aussi des 1, un résultat dû à la propriété de stricte multirésolution. Sans connaître les probabilités exactes que les coefficients descendants soient à 0 ou 1, on a quand même une certaine quantité d'information. On pourra aussi exploiter les coefficients voisins des parents et des descendants pour obtenir de

meilleures estimations.

10.7 Conclusion

Les images aux figures 10.2 et 10.3 semblent indiquer que les images en *halftones* peuvent être traités grâce aux pseudo-ondelettes binaires. La figure 10.1 semble indiquer que certaines bases pseudo-ondelettes peuvent aussi servir sur des images monochromes et la fig. 10.4 semble montrer que la technique s'applique aussi aux images de type fax. La grande simplicité des implémentations logicielles et matérielles rendent tout à fait vraisemblable l'utilisation des pseudo-ondelettes dans les systèmes de traitement d'image, soit à l'imprimerie, soit dans le cadre de la transmission de document par un moyen similaire au protocole facsimilé, soit encore dans les applications où le QOS (*quality of service*) peut varier.

Chapitre 11

Conclusion

La thèse est organisée de façon linéaire. Les chapitres 1 à 6 présentent le matériel introductif nécessaire à la bonne compréhension de nos contributions. Ces chapitres sont assez longs, mais nous avons préféré présenter toute la matière nécessaire plutôt que de laisser le soin au lecteur de rechercher toutes les références. Nous pensons que toute la matière qui est introduite est utilisée par la suite, si ce n'est que pour soutenir un raisonnement. Les quatre chapitres qui suivent, *Contributions au codage des entiers*, *Contributions au codage Huffman*, *LZW avec perte* et *Pseudo-ondelettes binaires* présentent l'essentiel de nos contributions ; d'autres contributions se trouvant dans un chapitre introductif. Ces quatres chapitres sont consacrés exclusivement à nos contributions. Nous y présentons les algorithmes, leurs analyses et les résultats numériques.

Cachés dans le matériel introductif, nous trouvons deux résultats nouveaux. Le premier résultat, présenté à la section 5.3.4.2, est la longueur moyenne des codes *phase-in* ainsi que la borne supérieure sur la différence par rapport à l'entropie, étant donnée une distribution uniforme. Curieusement, ce résultat semble inédit, malgré sa relative simplicité. Plus loin dans le même chapitre, nous présentons une nouvelle fonction de pairage qui ne fait intervernir que des manipulation de bits. Cette nouvelle fonction de pairage, présentée en détail à la section 5.3.6.3, nous dispense de l'utilisation d'opérations arithmétiques plus complexe comme la division entière et le modulo ; de plus, nous montrons que cette fonction de pairage a des propriété fractales intéressantes.

Dans le chapitre 7, *Contributions au codage des entiers* nous avons présenté plusieurs contributions. Nous avons d'abord présenté un algorithme modifié pour le calcul des bigrammes (voir section 7.2, p. 141). Cette nouvelle méthode permet d'aller chercher quelques pour-centièmes de plus sur le ratio de compression sans augmenter significativement la complexité du décodeur. À la section 7.3, p. 143, nous présentons de nouvelles solutions analytiques pour des variations des codes de Golomb en plus d'une nouvelle dérivation du résultat de Gallager et Van Voorhis [77]. Ces nouvelles variations des codes permettent de remplacer l'utilisation de codes *phase-in* par un code naturel. Cette simplification pourrait être nécessaire dans un environnement où la puissance de calcul est faible et lorsqu'on sera prêt à sacrifier la performance en compression.

Les codes tabou alignés, section 7.4.1, sont présentés ici pour la première fois. Nous présentons une étude systématique de ces codes. Nous donnons la preuve de leur universalité, c'est-à-dire leur capacité à encoder tout entier n avec $O(c \lg n)$ bits, où la constante c dépend de la taille du

bloc utilisé. Nous présentons ensuite les codes tabou généralisés, où la contrainte imposée par la structure en bloc des codes alignés est remplacée par une structure où la seule contrainte est de ne pas trouver un certain motif de bits dans le code (section 7.4.2). Ces codes sont basés sur une énumération combinatoire de toutes les chaînes de bits d'une longueur donnée ne présentant pas ce motif de bits interdit, le tabou. Nous montrons que ce motif devrait être une série de n zéros de façon à limiter le moins possible le nombre de combinaisons valides. Nous montrons aussi que les codes tabou généralisés sont unversels et que la constante d'universalité décroît en fonction de la longueur du tabou. Nous montrons aussi que les codes de Fibonacci, basés sur la représentation de Zeckendorf ne sont qu'un cas spécial des codes tabou généralisés.

Nous passons, à la section 7.5, p. 166, aux codes $(Start, Step, Stop)$, introduits par Fiala et Greene [70], et aux codes $Start/Stop$. Nous montrons que les codes $(Start, Step, Stop)$ sont un cas spécial des codes $Start/Stop$, une contribution, qui sont plus généraux. Fiala et Greene ayant omis de préciser les méthodes d'optimisation pour ces codes, nous présentons des algorithmes d'optimisation basés sur la fonction de probabilité cumulative pour les codes $Start/Stop$ et $(Start, Step, Stop)$. Nous présentons aussi un algorithme vorace pour optimiser les codes $Start/Stop$. L'algorithme de solution exacte basé sur la fonction de probabilité cumulative peut être contraint pour ne considérer que les solutions ayant une certaine forme, dictée, par exemple, par des considérations de codage/décodage rapide. Enfin, nous montrons comment les codes $(Start, Step, Stop)$ et $Start/Stop$ peuvent être modifiés pour devenir des codes universels. Non seulement ces codes modifiés seront universels, mais ils pourront être optimisés en fonction d'une distribution connue pour les petits entiers, réduisant ainsi la longueur moyenne du code.

Au chapitre 8, nous présentons nos contributions au codage des entiers grâce à des techniques de type Huffman. Nous y montrons que l'algorithme de Jones est très simple et demande peu de ressources mais que la compression qu'on obtient est moindre que celle obtenue grâce aux autres algorithmes, dû à un problème de stabilité du *splay tree*. Nous avons montré à la section 8.3.2 que l'algorithme W, qui est une modification de l'algorithme de Jones, donne des codes en moyenne un demi bit plus courts, sans augmentation de ressources et sans véritable complexification de l'algorithme. L'algorithme W utilise une règle d'adaptativité différente de celle de l'algorithme de Jones, réglant en grande partie le problème de stabilité du *splay tree*. L'algorithme W s'approche d'environ 0.1 bits des longueurs des codes données par l'algorithme Λ qui est optimal. Cette différence représente une différence relative d'environ seulement 5% sur les fichiers que nous avons utilisés. Nous avons par la suite modifié l'algorithme W pour nous assurer que les symboles de même fréquence reçoivent des codes de même longueur, ce qui nous permet de réduire par un facteur deux la différence entre la longueur des codes générés par l'algorithme W et l'algorithme Λ. L'Algorithme M donne des codes qui ne sont en moyenne que 0.05 bits plus long que ceux générés par l'algorithme Λ. L'arbre de code généré par l'algorithme M utilise des feuilles qui ne représentent plus des symboles individuels mais des ensembles de symboles. Chaque ensemble contient tous les symboles qui ont la même fréquence d'observation, élagant ainsi l'arbre de façon importante. À la section 8.5, nous avons montré que l'algorithme M appliqué sur de très grands alphabets crée parfois aussi peu que 10% des feuilles créées par l'algorithme Λ. Cela représente environ 500 fois moins de feuilles que les algorithmes de Jones et W.

Au chapitre 9, *LZW avec perte*, nous avons présenté deux variations de l'algorithme LZW qui admettent des pertes. Ces algorithmes, plutôt que de chercher les plus longues concordances exactes entre la séquence à compresser et les sous-séquences contenues dans le dictionnaire, vont accepter d'utiliser des concordances approximatives, en espérant trouver de plus longues

concordances afin d'améliorer le taux de compression tout en n'étant pas trop dommageable sur la qualité de l'image. Le premier de ces algorithmes, G-LLZW, procède à la recherche de cette concordance de façon vorace. Pour chaque symbole de la concordance, on vérifie si le pixel de la séquence à compresser correspond à un des pixels dans la treille en admettant au plus une distortion θ spécifiée par l'utilisateur. Cette constante θ dépend de l'espace de couleur utilisé et de l'échelle des composantes de la couleur. On pourrait travailler directement dans l'espace RGB mais d'autres espaces, comme YC_rC_b et YUV offrent des métriques plus raisonnables (voir l'appendice E à ce sujet). Le seuil θ empêche d'avoir des pixels très différents ce qui peut limiter la longueur des concordances trouvées, mais assure qu'une certaine qualité d'image est atteinte. Le second algorithme, P-LLZW, optimise un critère fourni par l'usager sur toutes les sous-séquences contenues dans le dictionnaire afin de déterminer la quelle de ces sous-séquences choisir. Cet algorithme, pour chaque concordance, prend un temps linéaire au nombre d'entrées dans le dictionnaire. Ce second algorithme contraint aussi la distortion maximale permissible pour un pixel individuel grâce au paramètre θ.

Nous avons comparé nos deux algorithmes à un algorithme existant, celui de Chiang et Po, montrant ainsi qu'ils donnent des résultats nettement supérieurs, autant en terme de compression que de qualité d'image. Nous avons aussi montré que l'algorithme P-LLZW donne une meilleure qualité d'image que l'algorithme G-LLZW pour le même taux de compression. Les taux de recompression (c'est-à-dire par rapport à l'image compressée grâce à la modification GIF de LZW) pour un paramètre θ modéré va à plus de 2 :1, tout en conservant une qualité d'image supérieure à 35 dB.

Nous avons aussi montré que nos implémentations des algorithmes G-LLZW et P-LLZW étaient compatibles au standard GIF. Bien que cela puisse ne sembler qu'une curiosité, il est en fait surprenant de voir qu'un algorithme aussi strict que LZW puisse admettre plusieurs algorithmes de compression pour le même algorithme de décompression. Les algorithmes de compression prennent des décisions très différentes de celles prises par l'algorithme de base et pourtant on peut décompresser le résultat sans problème avec l'algorithme standard. Cela a pour principal avantage de nous permettre d'utiliser des implémentations existantes du décodeur, par exemple, celles que l'on trouve dans les logiciels de navigation Internet.

Enfin, le chapitre 10, *Les pseudo-ondelettes binaires*, présente une méthode tout à fait inédite d'analyse d'image. Les pseudo-ondelettes binaires partagent la plupart de leurs propriétés essentielles avec les ondelettes classiques, bien qu'elles soient conçues pour opérer sur des vecteurs de bits plutôt que sur des vecteurs de réels. Alors que les schèmes classiques de compression d'images binaires utilisent de la prédiction dans le domaine spatial, ou encore des concordances approximatives de glyphes, les pseudo-ondelettes binaires permettent d'approcher le problème par la multirésolution.

Les tranformées pseudo-ondelettes sont calculées exclusivement grâce aux opérations logiques *et* et *ou-exclusif* plutôt qu'avec des opérations arithmétiques. Cela permet d'implémenter efficacement en matériel comme en logiciel les transformées rapides. Cette grande simplicité des implémentations rend tout à fait vraisemblable l'utilisation de pseudo-ondelettes dans les systèmes de traitement d'image, d'imprimerie, dans le cadre de la transmission de document par un moyen similaire au protocole facsimilé, soit encore dans les applications où le QOS (*quality of service*) peut varier selon la demande de l'utilisateur.

❁ ❁ ❁

Les contributions présentées dans cette thèse sont diversifiées. Nous nous sommes attaqué au codage des entiers avec les codes universels et les codes de type Huffman. Nous nous sommes attaqués à la compression d'image avec deux méthodes très différentes, à savoir une variation de l'algorithme LZW qui admet des concordances approximatives et une toute nouvelle classe d'ondelettes, les pseudo-ondelettes binaires qui permettent l'analyse multirésolution des images binaires.

Appendice A

Notation pour les codes

$C_\alpha(i)$	Code unaire de i, c'est-à-dire i uns suivi de zero.
$C_{\hat{\alpha}}(i)$	Code unaire tronqué de i, soit $i-1$ uns.
$C_{\hat{\alpha}(N)}(i)$	Code unaire conditionellement tronqué de i, voir section 5.3.5.2.
$C_\beta(i)$	Code « naturel » de i sur $\lceil \lg i \rceil$ bits ; indécodable si on ne connaît pas $\lceil \lg i \rceil$.
$C_{\hat{\beta}}$	Code naturel de i, sans le bit le plus significatif (donc $\lceil \lg i \rceil - 1$ bits de long).
$C_{\beta(n)}(i), C_{\hat{\beta}(n)}(i)$	Code naturel sur $n \in \mathbb{N}$ bits, de i, de i sans le bit le plus significatif.
$C_\gamma(i)$	Code entrelacé ; chaque bit de $C_\beta(i)$, est suivi de 1 si il reste des codes à lire ou de 0 si c'est le dernier bit, voir section 5.3.3.2.
$C'_\gamma(i)$	Le code formé par $(\, C_\alpha(\lceil \lg i \rceil) : C_{\hat{\beta}}(i) \,)$, voir section 5.3.3.2.
$C_\omega(i)$	Code récursif d'Elias, voir section 5.3.3.3.
$C_{\phi(N)}(i)$	Codes *phase-in* pour $0 \leqslant i < N$, voir section 5.3.4.2
$C_{\rho(N)}(i)$	Codes récursivement *phase-in* pour $0 \leqslant i < N$, voir section 5.3.4.3
$C'_{\rho(N)}(i)$	Codes récursivement *phase-in* inversés pour $0 \leqslant i < N$, voir section 5.3.4.3
$C_{\binom{n}{m}}(t)$	Codes énumératifs, voir section 5.3.4.4.
$C_{fib}(i)$	Codes de Fibonacci, voir section 5.3.3.5.

Appendice B

Démontration du théorème de Shannon

Démontrons que l'éq. (3.1) du théorème 3.3.1, page 37, satisfait en effet les trois énoncés. La démonstration présentée ici est essentiellement celle trouvée dans Ash [6]. Rappelons les trois contraintes que doit satisfaire une mesure d'information :

1. Elle est continue en p_i.

2. Si tous les p_i sont égaux, soit $p_i = \frac{1}{n}$, alors elle est monotone croissante en n.

3. Si on peut grouper les choix, alors elle est la somme pondérée des mesures pour les groupes, plus l'information propre au choix du groupe.

Démonstration. Le premier énoncé est satisfait, car

$$\frac{\partial \mathcal{H}(X)}{\partial P(X = x_i)} = 1 + \lg P(X = x_i)$$

qui est continu en $P(X = x_i)$. Le second énoncé stipule que

$$\mathcal{H}(\underbrace{\tfrac{1}{n}, \ldots, \tfrac{1}{n}}_{n \text{ fois}}) < \mathcal{H}(\underbrace{\tfrac{1}{n+1}, \ldots, \tfrac{1}{n+1}}_{n+1 \text{ fois}})$$

Il suffit de faire l'expansion des expressions et de simplifier :

$$\begin{aligned}
\mathcal{H}(\tfrac{1}{n}, \ldots, \tfrac{1}{n}) &< \mathcal{H}(\tfrac{1}{n+1}, \ldots, \tfrac{1}{n+1}) \\
-\sum_1^n \tfrac{1}{n} \lg \tfrac{1}{n} &< -\sum_1^{n+1} \tfrac{1}{n+1} \lg \tfrac{1}{n+1} \\
-\lg \frac{1}{n} &< -\lg \frac{1}{n+1} \\
\lg n &< \lg(n+1)
\end{aligned}$$

ce qui est toujours vrai pour $n \in \mathbb{N}$. Cela conclut la preuve pour le second énoncé. Le troisième énoncé requiert une reformulation. Soient $B_1, B_2, \ldots, B_k$ une partition des évènements de X. Rappelons qu'une partition d'un ensemble X est un ensemble de sous-ensembles $B_i \subseteq X$, tels que $\bigcup_i B_i = X$ et $\forall\, i, j \in \{1, 2, \ldots, k\}$, $B_i \cap B_j = \varnothing$. On doit montrer que

$$\mathcal{H}(X) = \mathcal{H}(B_1, B_2, \ldots, B_k) + \sum_{i=1}^{k} P(B = B_i)\mathcal{H}(B_i)$$

En supposant que le nombre de symboles soit n, et que tous les B_i aient m éléments (donc que $n = km$), on a

$$\begin{aligned}\mathcal{H}(\tfrac{1}{n}, \ldots, \tfrac{1}{n}) &= \mathcal{H}(\tfrac{m}{n}, \ldots, \tfrac{m}{n}) + \sum_{i=1}^{k} \tfrac{m}{n}\, \mathcal{H}(\tfrac{1}{m}, \ldots, \tfrac{1}{m}) \\ &= \mathcal{H}(\tfrac{m}{n}, \ldots, \tfrac{m}{n}) + \mathcal{H}(\tfrac{1}{m}, \ldots, \tfrac{1}{m})\end{aligned}$$

En posant $m^s = n$, on obtient

$$\mathcal{H}(\tfrac{1}{m^s}, \ldots, \tfrac{1}{m^s}) = \mathcal{H}(\tfrac{1}{m^{s-1}}, \ldots, \tfrac{1}{m^{s-1}}) + \mathcal{H}(\tfrac{1}{m}, \ldots, \tfrac{1}{m})$$

Posons $g(t) = \mathcal{H}(\frac{1}{t}, \ldots, \frac{1}{t})$. Alors l'équation précédante devient

$$g(m^s) = g(m^{s-1}) + g(m)$$

et $g(m^s) = s\, g(m)$, ce qui montre que $g(\cdot)$ a des propriétés de type logarithmique. Puisque $\mathcal{H}$ est strictement croissante,

$$g(m^s) < g(m^{s+1})$$

et

$$s\, g(m) < (s+1) g(m)$$

ce qui montre que $g(\cdot)$ est obligatoirement non négative. En choisissant $r, s, t \in \mathbb{N}$ tels que

$$m^s \leqslant r^t < m^{s+1} \tag{B.1}$$

et sachant que $g(\cdot)$ est strictement croissante, on peut poser

$$g(m^s) \leqslant g(r^t) < g(m^{s+1})$$

et

$$s\, g(m) \leqslant t\, g(r) < (s+1)\, g(m)$$

que l'on réorganise en

$$\frac{s}{t} \leqslant \frac{g(r)}{g(m)} < \frac{s+1}{t} \tag{B.2}$$

Si on pose $g(\cdot) = \lg(\cdot)$, les inégalités tiennent encore :

$$s\ \lg m \leqslant t\ \lg r < (s+1)\ \lg m$$

tout comme

$$\frac{s}{t} \leqslant \frac{\lg r}{\lg m} < \frac{s+1}{t}$$

que l'on combine avec l'éq. (B.2) pour obtenir

$$-\frac{1}{t} \leqslant \frac{g(r)}{g(m)} - \frac{\lg r}{\lg m} < \frac{1}{t}$$

Or, le choix de t est arbitraire, dans la mesure où r, s, et t sont choisis pour satifaire l'éq. (B.1). On peut alors choisir t arbitrairement grand, ce qui rend $\frac{1}{t}$ arbitrairement petit ; donnant ainsi

$$-\varepsilon < \left| \frac{g(r)}{g(m)} - \frac{\lg r}{\lg m} \right| < \varepsilon$$

ce qui nous permet de conclure que $g(\cdot) = \lg(\cdot)$, terminant ainsi la preuve du théorème. ■

Appendice C

Formules de Stirling pour $n!$ et $\lg n!$

L'approximation de $n!$, dûe à James Stirling (1692–1770) [220], est donnée par

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} - \frac{571}{2488320n^4} + \cdots\right)$$

Nous avons donc cette décomposition (aussi dûe à Stirling) :

$$\begin{aligned}
\lg n! = \sum_{i=1}^{n} \lg i &= \lg\left(\left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \frac{1}{12n} + \cdots\right)\right) \\
&= \lg\left(\frac{n}{e}\right)^n + \lg\sqrt{2\pi n} + \lg\left(1 + \frac{1}{12n} + \cdots\right) \\
&= n\lg n - n\lg e + \frac{1}{2}\left(\lg 2\pi + \lg n\right) + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} + \cdots \\
&= \left(n + \frac{1}{2}\right)\lg n - n\lg e + \frac{1}{2}\lg 2\pi + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} + \cdots \\
&\approx \left(n + \frac{1}{2}\right)\lg n - 1.442695n + 1.3257480\ldots \\
&\in O\left(\left(n + \frac{1}{2}\right)\lg n\right)
\end{aligned}$$

Cela nous permettra d'estimer

$$\begin{aligned}
\lg\binom{n}{k} &= \lg\left(\frac{n!}{k!(n-k)!}\right) \\
&= \lg n! - \lg k! - \lg(n-k)! \\
&\approx \left(n + \frac{1}{2}\right)\lg n - \left(k + \frac{1}{2}\right)\lg k - \left(n - k + \frac{1}{2}\right)\lg(n-k)
\end{aligned}$$

et

$$\begin{aligned}\lg \binom{n}{n_1\ n_2 \cdots n_k} &= \lg \left(\frac{n!}{n_1!\, n_2! \cdots\ n_k!} \right) \\ &\approx \left(n + \frac{1}{2} \right) \lg n - \sum\nolimits_{i=1}^{k} \left(n_i + \frac{1}{2} \right) \lg n_i\end{aligned}$$

Appendice D

Les Corpus

D.1 Un corpus en guise d'étalon

Une des bases de la méthode scientifique, c'est la possibilité de répéter les expériences. Dans diverses sciences, cette exigeance est satisfaite par divers protocoles qui régissent les expériences, dans notre cas, c'est beaucoup moins compliqué : il ne s'agit que de refaire les expériences sur les mêmes données. La façon privilégiée de procéder, c'est de mettre un corpus de données disponible publiquement, par exemple par le biais de serveurs FTP ou web.

Le corpus doit être composé de données qui sont adéquates au champ d'application considéré, qu'il s'agisse d'images, de sons ou de textes. Dans chaque cas, on cherchera à choisir des données qui sont représentatives, au moins *a priori*, des données qu'on devra compresser. Le nombre et la variété des séquences, même à l'intérieur d'une même classe générale, garantira la robutesse et l'utilité des méthodes testées. Un grand nombre de fichiers permettra de vérifier les propriétés de l'algorithme de façon statistique, et la variété permettra de cibler les cas pathologiques ou au moins identifier les cas qui donnent un peu plus de fil à retordre.

Les corpus que nous examinerons dans cet appendice sont le corpus de Calgary, le corpus de Canterbury, la suite d'images publiques Kodak et le corpus d'image USC-SIPI. Les corpus de Calgary et de Canterbury sont principalement orientés vers les données textuelles plus ou moins enrichies (html, troff, vaguement formatées), mais ils contiennent quelques fichiers d'autres types. Ces corpus ne sont pas les seuls corpus essentiellement textuels disponibles. Le corpus du *Gutenberg Project* regroupe plus de deux cents fichiers textes qui sont la transcription manuelle (réalisées par des bénévoles) de textes classiques, comme la Bible, les œuvres des poètes anglais du XIX^{e} siècle, en général des œuvres dont les droits sont expirés et tous en anglais. Le corpus de l'association des bibliophiles universels (ABU) fait écho au *Gutenberg Project*. En effet, le corpus de l'ABU ressemble beaucoup au corpus du *Gutenberg Project* : principalement de vieux textes en français dont les droits sont échus depuis belle lurette, encodés en ISO latin-1 et minimalement formatés.

Les corpus d'images sont plus intéressants. Le corpus d'image Kodak, par exemple, consiste d'images dites « photographiques », c'est-à-dire des scènes naturelles, telles que captées par un appareil photo — Kodak, on suppose. Kodak pousse l'utilisation d'un standard maison qui vise la reproduction de très haute qualité des images, et ce pour une large gamme d'applications, allant

de l'album souvenir de la famille aux presses haute résolution de l'industrie de la reprographie. Le corpus USC-SIPI offre des images d'origines diverses (images satellites, radar, photographiques, etc.) avec des résolutions variant de faibles à très élevées.

D.2 Le corpus de Calgary

Le corpus de Calgary a été introduit dans le livre de Bell, Cleary et Witten, *Text Compression* [12] pour comparer les algorithmes de compression qui y étaient présentés. Le corpus, élaboré avant 1990, est consistué de fichiers de tailles variables, de natures différentes, ces fichiers ont été choisis de façon à être représentatifs des fichiers que nous étions susceptibles de rencontrer au début des années '90. Bell et Witten décrivent le choix des fichiers dans le *readme* :

> *Nine different types of text are represented, and to confirm that the performance of schemes is consistent for any given type, many of the types have more than one representative. Normal English, both fiction and non-fiction, is represented by two books and papers (labeled book1, book2, paper1, paper2, paper3, paper4, paper5, paper6). More unusual styles of English writing are found in a bibliography (bib) and a batch of unedited news articles (news). Three computer programs represent artificial languages (progc, progl, progp). A transcript of a terminal session (trans) is included to indicate the increase in speed that could be achieved by applying compression to a slow line to a terminal. All of the files mentioned so far use ASCII encoding. Some non-ASCII files are also included : two files of executable code (obj1, obj2), some geophysical data (geo), and a bit-map black and white picture (pic). The file geo is particularly difficult to compress because it contains a wide range of data values, while the file pic is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros.*

De toute évidence, ce corpus est fortement biaisé vers les séquences représentant du texte, avec quelques « impuretés » destinées à vérifier la robustesse et la généralité des algorithmes. Le tableau D.1 décrit les fichiers, leur taille et leur type.

D.3 Le corpus de Canterbury

Le corpus de Calgary se faisant vieux, on a voulu le remplacer par ce nouveau corpus. Le corpus de Canterbury est aussi dû aux auteurs du corpus de Calgary. Cependant, en examinant le tableau D.2 on remarque d'abord l'inflation de la taille des fichiers. Cette inflation reflète la tendance courante au gigantisme des fichiers. Le corpus de Canterbury est plus varié que le corpus de Calgary. Plutôt que de trouver exclusivement des fichiers textes — bien qu'ils s'y trouvent encore en bon nombre — on trouve aussi des fichiers exécutables et des fichiers d'applications comme une séquence génomique et un fichier de chiffrier.

D.4 Le corpus Kodak

Kodak met à la disposition du public environ 250 images provenant de diverses caméras digitales ou classiques. Kodak en permet l'utilisation à des fins non commerciales, de recherche ou privées. Ces images sont regroupées sous le nom de *Kodak Digital Image Offering*, et sont

Corpus de Calgary		
Fichier	Taille	Description
bib	111 261	Texte (bibliographie), anglais
book1	768 771	Texte (troff), anglais
book2	610 856	Texte (troff), anglais
geo	102 400	Données binaires
news	377 109	Texte (format NNTP), anglais
obj1	21 504	Exécutable
obj2	246 814	Exécutable
paper1	53 161	Texte (troff), anglais
paper2	82 199	Texte (troff), anglais
paper3	46 526	Texte (troff), anglais
paper4	13 286	Texte (troff), anglais
paper5	11 954	Texte (troff), anglais
paper6	38 105	Texte (troff), anglais
pic	513 216	Image, noir et blanc
progc	39 611	Programme, C
progl	71 646	Programme, Lisp
progp	49 379	Programme, Pascal
trans	93 695	Texte enrichi (session de terminal)

TAB. D.1 – Les fichiers du corpus de Calgary.

Corpus de Canterbury		
Fichier	Taille	Description
Alice29.txt	152 089	Texte, *Alice's Adventures in Wonderland*
arj241a.exe	223 856	Exécutable DOS
asyoulik.txt	125 179	Texte, *As You Like It*
bible.txt	4 047 392	Texte, la Bible
cp.html	24 603	Hypertexte, "Compression Pointers"
ecoli	4 638 690	Séquence génomique
fields.c	11 150	Programme, C
gramma.lsp	3 721	Programme, Lisp
kennedy.xls	1 029 744	Fichier de chiffier Excel
kjv.gutenberg	4 846 137	Texte (formaté), la Bible
lcet10.txt	426 754	Texte (formaté), *Workshop on Electronic Texts*
plrabn12.txt	481 861	Texte, *Paradise Lost*
ptt5 †	513 216	Image, noir et blanc
sum	38 240	Exécutable au format ELF (Linux)
worlds192.txt	2 473 400	Texte (formaté), *The World Fact Book 1992*
xargs.1	4 227	Texte, man page

TAB. D.2 – Les fichiers du corpus de Canterbury. † Ce fichier est le même que le fichier *pic* du corpus de Calgary.

FIG. D.1 – Une des images du corpus Kodak.

disponibles sur leurs serveurs FTP et web. La fig. D.1 donne un exemple de ces superbes images que l'on trouve dans ce corpus. Le format d'image proposé par Kodak, PhotoCD, emmagasine l'image dans chaque fichier en (jusqu'à) six résolutions différentes (192×128, 384×256, 768×512, 1536×1024, 3072×2048 et 6144×4096) en 24 bits, chaque destinée à être extraite en fonction du médium d'affichage. Chaque sous-image est restituée exactement par l'algorithme de compression, qui est sans perte. Les fichiers obtenus sont énormes (de 4 à 10 mégaoctets), mais cela n'est pas véritablement un inconvénient, étant donné que le support préféré est le CD-ROM.

Pour les expériences que nous avons menées au chapitre 9, nous avons utilisé la résolution 768×512 (ou, en « portrait », 512×768), ce qui nous a paru raisonnable. La résolution 768×512 correspond plus ou moins à la résolution utilisée pour l'application principale, soit l'affichage sur l'écran de télévision. Même à cette résolution, le corpus demande quand même 300 méga-octets de stockage.

D.5 Le corpus USC-SIPI

Le corpus USC-SIPI est maintenu par le *Signal & Image Processing Institute* de l'université de Californie du sud. Ce corpus éclectique s'est constitué, contrairement aux corpus de Calgary et de Canterbury, sur une longue période par ajouts successifs. Le corpus contient des photographies aériennes, radar, des textures de toutes sortes, des scènes bucoliques et même quelques séquences vidéo (sans le son). La résolution des images varie grandement mais elle sont presque

toutes carrées (256×256, 512×512, 1024×1024, etc.). La résolution en couleur va de 8 bits en tons de gris jusqu'à 24 bits couleur. La fig. D.2 montre quelques exemples des images de ce corpus.

L'image la plus familière de ce corpus — du moins dans la communauté du traitement et de la compression d'image — est sans nul doute la photographie de Lena, fig. D.2 a). Lena Sodeberg, née Sjööblom, apparaît à la page centrale du magasine *Playboy* de novembre 1972. L'image entre dans le corpus dès 1972, numérisée par un inconnu grâce à un équipement tout aussi inconnu. L'image est restée dans la communauté et a fini par se retrouver dans un très grand nombre de communications, articles et livres. Munson justifie, *a posteriori*, l'utilisation de l'image Lena [152]. Cette image, dit-il, présente un bon mélange de régions planes, texturées, dégradées et contenant des détails fins, ce qui la rend intéressante pour vérifier les divers algorithmes de traitement d'image et de compression. Il s'empresse d'ajouter que le fait qu'il s'agisse de la photographie d'une séduisante jeune femme a certainement joué un rôle dans la popularité de l'image. Bien que l'utilisation de cette image soit une violation du droit d'auteur, *Playboy* n'a jamais tenté de sévir, bien qu'il semble que la société ait été au courant de la situation depuis au moins 1992.

a) c)

b) d)

FIG. D.2 – Des images du corpus USC-SIPI. En a) Lena (voir le texte), en b) l'image Goldhill, c) Barabara et d) Baboon.

Appendice E

Les espaces de couleur additifs

Les façons de représenter les couleurs sont nombreuses. Dans cet appendice, nous en présentons quelques unes. Nous commençons par l'espace RGB, qui est en quelque sorte notre espace de référence. Cet espace RGB est un espace vectoriel qui peut subir des transformations linéaires ou non linéaires pour nous amener dans un autre espace de couleurs. Comme chaque élément de l'espace RGB est un vecteur (r, g, b), il suffit d'une matrice de 3×3 pour définir une transformation linéaire. Ces transformations seront des compositions de rotations, de mise à l'échelle, de translation, de miroir, etc. Les transformations linéaires que nous présentons nous amènerons dans les espaces Kodak 1, Kodak YCC, XYZ, Xerox YES, YIQ, YUV, YC_rC_b, YP_rP_b, et Ohta 1 et 2. Toutes ces transformations sont linéaires. Nous présenterons aussi des transformations non linéaires entre l'espace RGB et les espaces HSV, CIÉ $L^*a^*b^*$, NCS et Munsell.

Notons que nous nous intéressons seulement aux systèmes additifs, où les couleurs sont créées en ajoutant des sources lumineuses. Cela diffère des systèmes soustractifs, où la lumière blanche réfléchie est filtrée par une série de pigments qui absorbent des bandes de fréquences données. Les systèmes soustractifs correspondent aux plastiques, aux peintures, aux encres et aux teintures qui sont utilisés dans le monde physique de tous les jours pour donner couleur aux objets. Le système additif procède par l'ajout successif de sources lumineuses pour créer les couleurs. Les couleurs émises par votre moniteur sont composées par des sources rouge, verte et bleue qui, grâce à des intensités variées qui se combinnent, permettent de former une large gamme de couleur. Il faut être conscient que les couleurs représentables par ce système ne sont en fait qu'un sous-ensemble des couleurs que l'œil est capable de percevoir.

Comme nous présentons les transformées de l'espace RGB vers ces autres espaces, il est important de réaliser que les solides montrés sont en fait l'intersection entre les couleurs représentables grâce au cube RGB et les différents espaces de couleurs. Certains des espaces de couleurs sont capables de représenter des couleurs qu'il est impossible d'afficher sur un moniteur. Les solides montrés aux figures de cet appendice doivent être compris comme l'intersection de l'espace RGB et de chacun des espaces individuels.

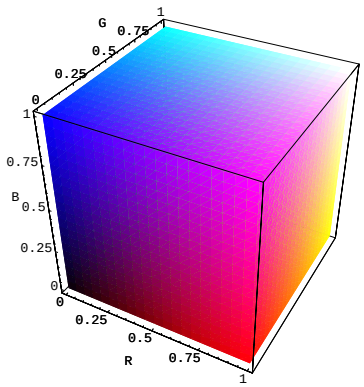


FIG. E.1 – L'espace de couleur RGB.

E.1 Espace *RGB*

L'espace de couleur *RGB* sera notre espace de référence. Cet espace de couleur, étant directement supporté par le matériel, est particulièrement simple d'utilisation. Les composantes R, G et B sont les bases d'un cube où toutes les couleurs représentables — par un matériel typique comme un moniteur vidéo — sont confinées. Idéalement, nous pouvons concevoir ce cube comme étant unitaire, c'est-à-dire de côtés de longueur 1. Selon la représentation de la couleur et le matériel utilisé, on utilisera un nombre fini de valeurs possibles pour chaque composante, typiquement 256. Certains équipements permettent jusqu'à 16 bits par composantes, d'autres n'en permettront que 6. Les matrices de transformation qui sont données dans cet appendice sont normalisées, ce qui nous permet d'utiliser des coordonnées mises à l'échelle de la façon que l'on préfère.

La « transformation » de *RGB* vers *RGB* est donnée par :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Comme l'espace *RGB* nous sert d'espace de référence, toutes les autres matrices de transformation permettront de convertir des coordonnées *RGB* vers un autre espace et vice-versa. La fig. E.1 montre le cube *RGB*.

Cette appendice ne contient pas toutes les conversions possibles de tous les espaces de couleur vers tous les espaces de couleur. Cependant, on peut s'en sortir assez facilement en combinant les matrices de transformation. Supposons que nous voulions convertir d'un espace A à un espace B sans avoir une transformation allant directement de A à B. Il suffit de transformer A vers *RGB*, puis de *RGB* vers B.

E.2 Espaces Kodak

Deux espaces de couleurs sont utilisés par Kodak. L'espace Kodak 1 est une simple transformation qui concentre l'information de luminance — la quantité de lumière présente dans la couleur — dans la première composante. La seconde composante est une différence jaune-bleue : les composantes rouges et vertes forment les tons de jaunes dans le système additif. La troisième composante est donnée par une différence rouge-cyan, car les composantes vertes et bleues forment les tons de cyan. Les deux matrices de transformations sont données par

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & -1 & 1 \\ 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} K_1 \\ K_2 \\ K_2 \end{pmatrix} \tag{E.1}$$

et

$$\begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} K_1 \\ K_2 \\ K_2 \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.2}$$

La seconde transformée, Kodak YCC, est utilisée dans le protocole de compression d'image sans perte Photo CD. C'est une version normalisée de la première. Les composantes rouge, verte et bleue du signal contribuent respectivement 0.299, 0.587 et 0.114 de la luminance perçue pour une couleur. La première composante devient alors la luminance normalisée. Les deux autres composantes s'interprètent toujours comme étant des différences jaune-bleu et rouge-cyan. Les transformées sont données par

$$\begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ C_1 \\ C_2 \end{pmatrix} \tag{E.3}$$

et

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & -0.194 & 0.509 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} Y \\ C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.4}$$

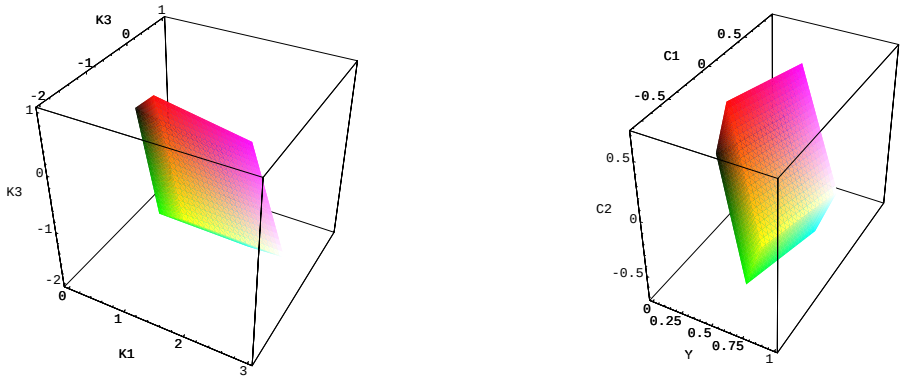
ce qui est très facile à calculer.

La figure E.2 illustre les deux transformations effectuées sur le cube RGB selon les transformées Kodak 1 et YCC.

E.3 Espace XYZ

L'espace XYZ nous vient de la Commission Internationale de l'Éclairage (la CIÉ) [50]. Alors que dans la plupart des espaces de couleurs, la composante Y représente la luminance pour les phosphores du type que l'on retrouve dans les écrans de téléviseurs, la composante Y de l'espace XYZ est calibrée pour les écrans de moniteurs d'ordinateurs et en fonction des phosphores dans les téléviseurs haute résolution. Le X correspond à un axe vert-jaune et le Z à un axe bleu. Les transformées sont données par

$$\begin{pmatrix} 0.431 & 0.342 & 0.178 \\ 0.222 & 0.707 & 0.071 \\ 0.020 & 0.130 & 0.939 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{E.5}$$



a) b)

FIG. E.2 – Les espaces de couleur Kodak. En a), l'espace Kodak 1 et eb b), l'espace Kodak YCC.

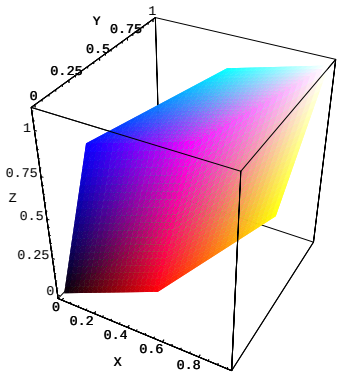


FIG. E.3 – L'espace de couleur XYZ.

et

$$\begin{pmatrix} 3.060 & -1.392 & -0.475 \\ -0.968 & 1.875 & 0.042 \\ 0.069 & -0.230 & 1.069 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.6}$$

et la fig. E.3 montre comment le cube RGB est transformé en XYZ.

E.4 Espace Xerox YES

Cet espace de couleur a été introduit par Xerox dans le but de créer des systèmes à haute fidélité de reproduction des couleurs. On remarquera que le Y de YES est donné par les mêmes coefficients que le Y de XYZ. La composante E correspond à une différence mauve moins vert et la composante S à une différence bleu moins jaune. On remarquera que, contrairement à

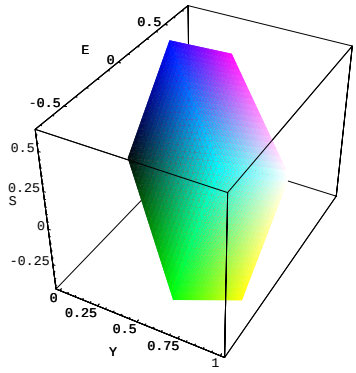


FIG. E.4 – L'espace de couleur YES.

la transformée XYZ, aucun coefficient n'est vraiment petit. Le cube RGB représenté dans le système YES est montré à la fig. E.4.

Les transformées sont données par

$$\begin{pmatrix} 0.222 & 0.707 & 0.071 \\ 0.351 & -0.705 & 0.387 \\ -0.201 & -0.206 & 0.605 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ E \\ S \end{pmatrix} \tag{E.7}$$

et

$$\begin{pmatrix} 1.166 & 1.488 & -1.089 \\ 0.976 & -0.500 & 0.205 \\ 0.719 & 0.324 & 1.361 \end{pmatrix} \begin{pmatrix} Y \\ E \\ S \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.8}$$

E.5 Espace YIQ

L'espace YIQ est utilisé pour encoder les images selon le protocole NTSC (*National Television Standards Committee*). Les télévisions nord américaine et japonaise utilisent le standard NTSC pour la diffusion des émissions. Puisque la télévision couleur devait obligatoirement être compatible avec la télévision noir et blanc, le signal de luminance seule, Y, a été augmenté de deux autres composantes qui correspondent approximativement à la teinte (I) et à la saturation (Q). Les trois signaux sont multiplexés en fréquences pour la diffusion. Le protocole NTSC ne prévoit pas que chaque pixel de l'image puisse avoir ses trois composantes distinctes. Les composantes I et Q sont transmises l'une à la suite de l'autre, et non pas simultanément, ce qui fait que deux pixels consécutifs partagent soit le même I, soit le même Q. Comme l'œil est particulièrement sensible à la luminance et comparativement peu à la teinte et à la saturation, cela permet quand même d'obtenir des images de qualité satisfaisante.

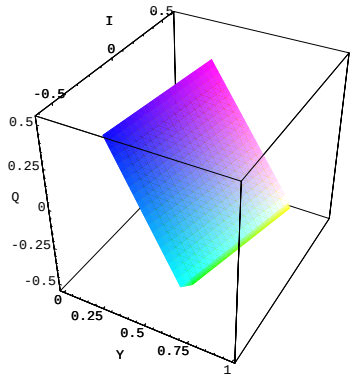


FIG. E.5 – L'espace de couleur YIQ.

Les matrices de transformation sont données par

$$\begin{pmatrix} 0.289 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.322 \\ 0.212 & -0.523 & 0.311 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ I \\ Q \end{pmatrix} \tag{E.9}$$

et

$$\begin{pmatrix} 1.011 & 0.964 & 0.628 \\ 1.010 & -0.261 & -0.641 \\ 1.009 & -1.097 & 1.709 \end{pmatrix} \begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.10}$$

E.6 Espace YUV

Les systèmes de télévision européens utilisent l'encodage PAL ou SÉCAM (respectivement, *Phase Alternating Line* et Séquentiel Couleur à Mémoire[1]) dont l'encodage de couleur est réalisé grâce à l'espace YUV. Ici encore, Y encode la luminance du signal. Les composantes U et V correspondent ici aussi vaguement à la teinte et à la saturation. Les transformées sont données par

$$\begin{pmatrix} 0.289 & 0.587 & 0.114 \\ -0.147 & 0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ U \\ V \end{pmatrix} \tag{E.11}$$

et

$$\begin{pmatrix} 1.010 & -0.000 & 1.151 \\ 1.309 & -0.511 & -0.737 \\ -0.527 & 2.632 & 0.877 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.12}$$

alors que la forme de l'espace YUV est donnée par la fig. E.6.

1 Et non comme on eut pu penser : Système Essentiellement Contre les AMéricains.

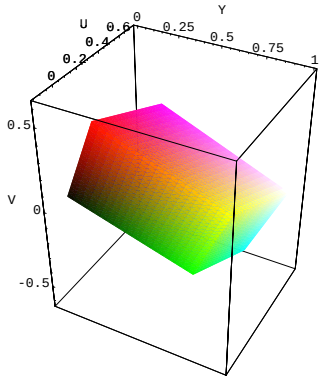


FIG. E.6 – L'espace de couleur YUV.

E.7 ★ Espaces YC_rC_B

L'espace YC_rC_b est utilisé dans plusieurs protocoles de compression d'image, notemment JPEG, MPEG, plusieurs formats à base d'ondelettes dont Déjà Vu. La composante Y est encore la luminance. La composante C_r correspond à la « composante rouge » et C_b à la « composante bleue », bien qu'on aurait pu penser qu'il s'agisse des différences bleue et rouge, respectivement. Les transformées sont

$$\begin{pmatrix} 0.289 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.418 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ C_r \\ C_b \end{pmatrix} \tag{E.13}$$

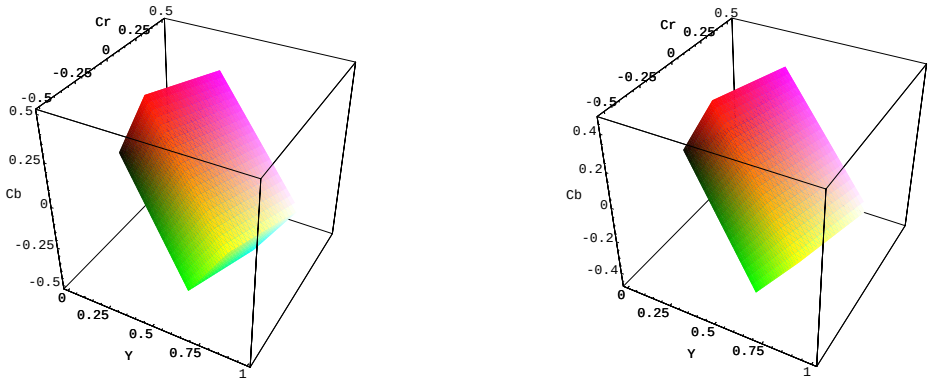
et

$$\begin{pmatrix} 1.009 & -0.000 & 1.417 \\ 1.011 & -0.344 & -0.701 \\ 1.010 & 1.772 & 0.015 \end{pmatrix} \begin{pmatrix} Y \\ C_r \\ C_b \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.14}$$

Alors que nous travaillions au projet Déjà Vu, lors d'un de nos séjours aux États Unis, avons dû modifier légèrement l'espace YC_rC_b. Ce nouvel espace, légèrement différent, $\tilde{Y}\tilde{C}_r\tilde{C}_b$ a été conçu pour avoir une transformée inverse particulièrement rapide à calculer. Le but étant de nous dispenser d'avoir recours aux opérations en points flottants pour le calcul de la transformée inverse, nous avons changé les paramètres de façon à ce qu'il soit facile de calculer les produits grâce à des décalages et des additions seulement. La transformée *inverse* est donnée par

$$\begin{pmatrix} 1 & 0 & \frac{3}{2} \\ 1 & -\frac{1}{4} & -\frac{3}{4} \\ 1 & \frac{7}{4} & 0 \end{pmatrix} \begin{pmatrix} \tilde{Y} \\ \tilde{C}_r \\ \tilde{C}_b \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.15}$$

On remarquera que tous les coefficients sont des fractions simples. Par exemple, $\frac{3}{2}x = x + \frac{1}{2}x$; la division par deux se calculant par un décalage à droite d'un bit. De même, $\frac{7}{4}x = 2x - \frac{1}{4}x$, à l'instar de la division par deux, la division par quatre se calcule par un décalage à droite de deux bits ; la multiplication par deux se fait par un décalage à gauche d'un bit. Une fois la matrice



a) b)

FIG. E.7 – Les espaces de couleur YC_rC_b. En a), l'espace original et eb b), l'espace modifié pour Déjà Vu.

inverse établie — notez que les nouveaux coefficients ressemblent quand même aux coefficients de l'inverse pour YC_rC_b — il ne nous reste qu'à calculer l'inverse de l'inverse, résultant en la transformée

$$\begin{pmatrix} 0.304 & 0.609 & 0.087 \\ -0.174 & -0.348 & 0.522 \\ 0.464 & -0.406 & -0.058 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \tilde{Y} \\ \tilde{C}_r \\ \tilde{C}_b \end{pmatrix} \tag{E.16}$$

Les transformées YC_rC_b et $\tilde{Y}\tilde{C}_r\tilde{C}_b$ sont comparés à la figure E.7.

E.8 Espace YP_rP_b

L'espace YP_rP_b peut être considéré comme une mise à jour de YC_rC_b pour les nouveaux phosphores. L'espace YP_rP_b est utilisé dans les protocoles de télévision numérique ainsi que pour la télévision haute définition. La fig. E.8 montre l'espace obtenu et les transformations sont données par :

$$\begin{pmatrix} 0.212 & 0.701 & 0.086 \\ -0.116 & -0.383 & 0.500 \\ 0.500 & -0.445 & -0.055 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ P_r \\ P_b \end{pmatrix} \tag{E.17}$$

et

$$\begin{pmatrix} 1.001 & 0.001 & 1.576 \\ 1.001 & -0.225 & -0.477 \\ 0.999 & 1.828 & 0.000 \end{pmatrix} \begin{pmatrix} Y \\ P_r \\ P_b \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.18}$$

E.9 Espaces Ohta

La vraie transformation de couleur Otha demande que l'espace soit obtenu par une analyse par composantes principales du nuage de couleurs présentes dans l'image [153]. On peut

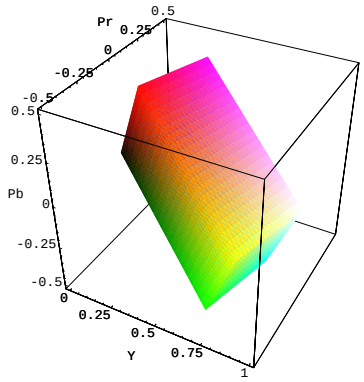


FIG. E.8 – L'espace de couleur YP_rP_b.

obtenir ces vecteurs de base soit en utilisant la transformée de Karhunen-Loève ou encore la décomposition en valeurs singulières. On obtient ainsi un espace de couleur optimal (sous une transformation linéaire) pour les couleurs d'une image donnée. Bien qu'on puisse calculer cette base optimale relativement facilement et qu'il coûte peu de la transmettre au décodeur, on peut utiliser les versions simplifiées suggérées par Ohta, données par ces transformées :

$$\begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{2} & 1 & -\frac{1}{2} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} \tag{E.19}$$

et

$$\begin{pmatrix} 1 & 1 & -\frac{1}{3} \\ 1 & 0 & \frac{2}{3} \\ 1 & -1 & -\frac{1}{3} \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.20}$$

puis

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} I'_1 \\ I'_2 \\ I'_3 \end{pmatrix} \tag{E.21}$$

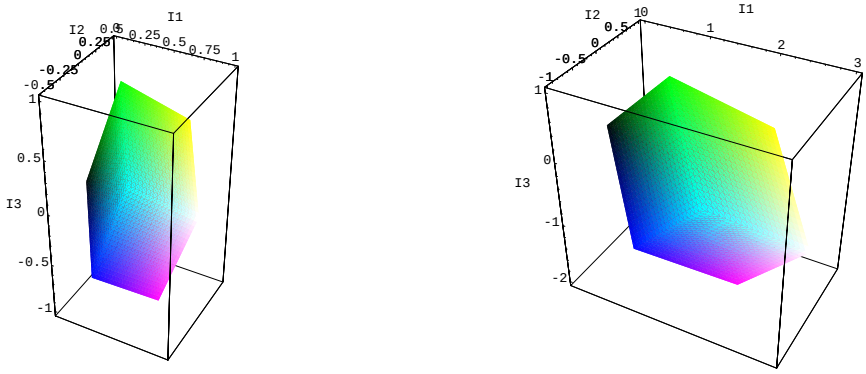
et

$$\begin{pmatrix} \frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{4} & -\frac{1}{2} & -\frac{1}{4} \end{pmatrix} \begin{pmatrix} I'_1 \\ I'_2 \\ I'_3 \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{E.22}$$

Nous trouvons ces deux espaces comparés à la fig. E.9.

E.10 Espace *HSV*

Le système de couleur *HSV* (*hue*, *saturation* et *value*, soit teinte, saturation et luminance) est un espace de couleur non-linéaire. Deux des paramètres, h et s sont des coordonnées polaires.



a) b)

FIG. E.9 – Les espaces de couleur Otha simplifiés. En a) la première transformée donnée dans le texte, en b) la seconde.

Le paramètre h, souvent exprimé en degrés, correspond à la teinte. Les couleurs primaires sont séparées de 60° : rouge correspond à 0°, jaune à 60°, vert à 120°, cyan à 180°, bleu à 240° et enfin magenta à 300°. Le paramètre s, la saturation ou pureté de la couleur, correspond à la distance au pôle. Lorsque $s = 1$, la couleur est très pure, très vive, alors que lorsque s est près de zéro, la couleur est un ton de gris de luminosité v. Ce dernier paramètre, v donne l'intensité lumineuse de la couleur, de 0, parfaitement noir à 1, très lumineux mais pas nécessairement blanc.

Ce système ne forme cependant pas un espace cylindrique. Plutôt, les paramètres sont contraints de façon à former un cône. La fig E.10 montre le cône de couleur HSV. L'espace généré est un cône car une couleur très peu lumineuse ne peut pas être très pure ; elle ne peut être qu'un ton de gris plutôt foncé avec une légère teinte. Seule une couleur suffisamment lumineuse peut être fortement saturée.

Ce système est intuitif car il correspond aux cercles de couleurs des artistes, en version additive, où les six couleurs primaires se suivent, à la différence que le système de couleur est additif plutôt que soustractif. De nombreux logiciels utilisent ce système pour les dialogues de choix de couleurs.

E.11 Espace CIÉ $L^*a^*b^*$

La Commission Internationale de l'Éclairage (la CIÉ) a défini ce standard de représentation de couleur en 1976. Dans ce modèle de couleur, l'axe L^* correspond à la luminance, soit l'intensité lumineuse de la couleur. L'axe a^* est l'axe vert-rouge, et b^* l'axe bleu-jaune. Une valeur a^* négative contribue une composante verte, une value positive une composante rouge. Si la valeur est près de zéro, très peu de couleur est contribuée. La saturation en couleur est d'autant plus grande que l'on est éloigné du zéro. La même chose est valide pour l'axe b^*, à différence qu'il s'agit de l'axe bleu-jaune.

La propriété intéressante de cet espace est qu'il perceptuellement uniforme. C'est-à-dire

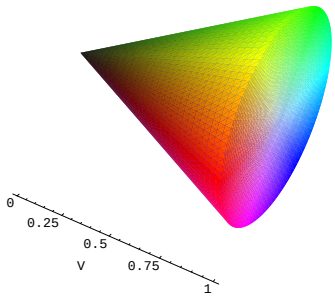


FIG. E.10 – L'espace de couleur HSV. L'axe indiquée correspond à v. Les paramètres h et s ne sont pas indiqués sur le graphique. Le paramètre s correspond à la distance à l'axe v du cône. Le paramètre h donne, en degrés, l'angle que fait la couleur avec le point rouge, en sens horaire. Les couleurs primaires sont séparées de 60° : rouge correspond à 0°, jaune à 60°, vert à 120°, cyan à 180°, bleu à 240° et enfin magenta à 300°.

qu'étant donné un point x dans l'espace $L^*a^*b^*$, tous les points qui sont à une distance euclidienne c de celui-ci seront perçus comme également différent de x. Autrement dit, la région définie par la *just noticable difference* autour d'une couleur donnée est sphérique.

Les formules de conversions de RGB vers $L^*a^*b^*$ sont plutôt laborieuses. Commençons par calculer les coordonnées (x, y, z) à partir des valeurs (r, g, b) grâce à l'éq. (E.5) :

$$\begin{pmatrix} 0.431 & 0.342 & 0.178 \\ 0.222 & 0.707 & 0.071 \\ 0.020 & 0.130 & 0.939 \end{pmatrix} \begin{pmatrix} r \\ g \\ b \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Les valeurs (L^*, a^*, b^*) sont données par

$$\begin{aligned} L^* &= 116\,(\sqrt[3]{y} - 16) \\ a^* &= 500\,\left(\sqrt[3]{x} - \sqrt[3]{y}\right) \\ b^* &= 200\,\left(\sqrt[3]{y} - \sqrt[3]{z}\right) \end{aligned}$$

E.12 Espace NCS

On doit ce système au *Scandinavian Colour Institute* de Stockholm en Suède. Le *Natural Colour System*, ou NCS, est basé sur six composantes primitives : le noir, le blanc, le jaune, le rouge, le bleu et le vert. Ce système vise plutôt les reproductions de couleurs sur papier ou pour les plastiques. L'espace NCS peut être représenté comme un octaèdre, dont les coins équatoriaux sont occupés par les couleurs primaires jaune, bleu, rouge et vert, et dont les coins polaires sont occupés par le noir et le blanc (voir fig. E.12). Le système de notation ne se prête pas de suite à une représentation pratique. Par exemple, la couleur S2060-B30G, représente une couleur ayant 20% de noir, une saturation de 60%, et compte 30% de vert sur une base bleue. Ce système n'est pas sans rappeler l'espace $L^*a^*b^*$.

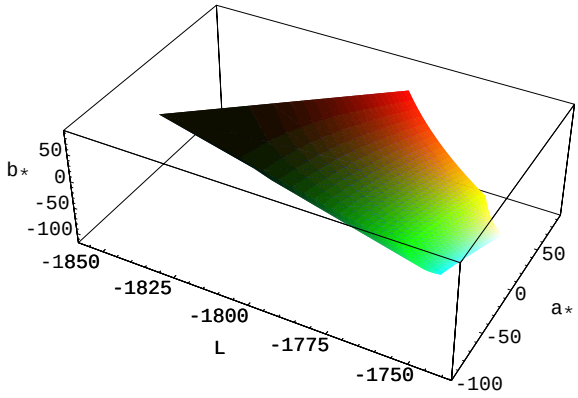


FIG. E.11 – L'espace de couleur $L^*a^*b^*$.

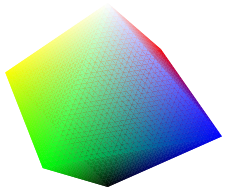


FIG. E.12 – L'espace de couleur NCS.

E.13 Espace de Munsell

Cet espace de couleur a été introduit par Albert H. Munsell (1858 – 1918) [150, 151]. Munsell était peintre de métier (on lui doit par exemple un portrait d'Hellen Keller, 1892) et il s'est très tôt intéressé à la systématisation de la description des couleurs. Alors qu'avec le système HSV, on retrouve six axes principaux correspondant aux couleurs primaires, le système de Munsell en comporte cinq, correspondant aux couleurs rouge, jaune, vert, bleu et mauve. Les axes principaux sont espacés de 72°, avec des axes secondaires aux 36°. Les axes sont donnés par

$$\begin{aligned} R &= 0^\circ \\ YR &= 36^\circ \\ Y &= 72^\circ \\ YG &= 108^\circ \\ G &= 144^\circ \\ BG &= 180^\circ \\ B &= 216^\circ \\ PB &= 252^\circ \\ P &= 288^\circ \\ PR &= 324^\circ \end{aligned}$$

La luminance est donnée par un nombre entre 0 et 10, et la saturation dépend de l'angle. Curieusement, l'espace de Munsell n'est pas un solide élégant : il a une forme irrégulière, comme on peut le voir à la fig. E.13. Le solide de la figure a été construit grâce aux données mise à la disposition du public par Munsell Inc. Ces données représentent une correspondance entre la notation de Munsell et l'espace de couleur XYZ, facilement convertible en RGB. La figure montre cet espace reconstruit avec les couleurs RGB mais aux coordonnées prévues par le système de Munsell.

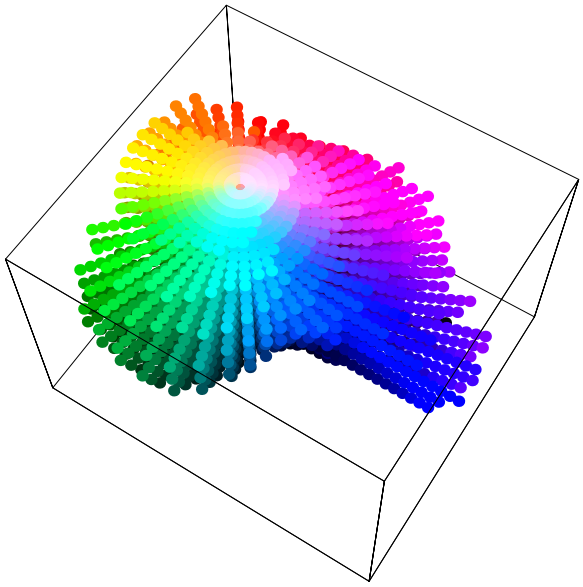


FIG. E.13 – L'espace de couleur de Munsell.

Bibliographie

[1] Acharya (Tinku) et Já Já (Joseph). – *Enhancing LZW Coding Using a Variable-Length Binary Encoding.* – Rapport technique n° TR-95-70, Institute for Systems Research, 1995.

[2] Acharya (Tinku) et Já Já (Joseph). – An online variable length binary encoding of text. *Informatics & Computer Science*, vol. 94, 1996, pp. 1–22.

[3] Adel'son-Vel'skii (G. M.) et Landis (E. M.). – An algorithm for the organization of information. *Dokl. Acad. Nauk. SSSR Math.*, vol. 146, n° 2, 1962, pp. 263–266. – Traduit à l'anglais dans *Sov. Math. Dokl.*, # 3, pp. 1259–1262.

[4] Ahlswede (Rudolf), Han (Te Sun) et Kobayashi (Kingo). – Universal coding of integer and unbounded search trees. *IEEE Trans. Inf. Theory*, vol. 43, n° 2, 1997, pp. 669–682.

[5] Allebach (J. P.) et Liu (B.). – Analysis of halftone dot profile and aliasing in the discrete binary representation on images. *J. Opt. Soc. Amer.*, vol. 67, 1977, pp. 1147–1154.

[6] Ash (Robert B.). – *Information Theory.* – Dover, 1965.

[7] Barnard (G. A.). – Statistical calculations of word entropies for four western languages. *IEEE. Trans. Inf. Theory*, vol. 1, n° 1, 1955, pp. 49–53.

[8] Bassiouni (Mostafa A.) et Mukherjee (Amar). – Efficient decoding of compressed data. *J. Amer. Soc. Inf. Sci.*, vol. 46, n° 1, 1995, pp. 1–8.

[9] Bayer (B. E.). – An optimum method for two level rendition of continuous-tone pictures. *In : International Conference on Communications 1973.* pp. 26–11–26–15. – Procs. IEEE.

[10] Bell (T. C.). – Better OPM/L text compression. *IEEE Trans. Comm.*, vol. 34, n° 12, 1986, pp. 1176–1182.

[11] Bell (T. C.). – *A unifying Theory and Improvements for Existing Approaches to Text Compression.* – Thèse de PhD, University of Canterbury, 1986.

[12] Bell (T. C.), Cleary (J. G.) et Witten (I. H.). – *Text Compression.* – Prentice-Hall, 1990. QA76.9 T48B45.

[13] Bellman (R.). – *Adaptive Control Processes : A Guided Tour.* – Princeton University Press, 1961.

[14] Billotet-Hoffman (C.) et Bryngdahl (O.). – On the error diffusion technique for electronic halftoning. *Proceedings of the Society for Information Display*, vol. 24, n° 3, 1983, pp. 253–258.

[15] Boissonat (J.D) et Yvinec (M.). – *Algorithmic Geometry.* – Cambridge, 1998.

[16] Boncompagni (Baldassare). – *Tre Scritti Inediti di Leonardo Pisano : pubblicati da Baldassare Boncompagni, secondo la lezione di un codice della Biblioteca ambrosiana di Milano.* – Tipografia Galileina di M. Cellini, 1852.

[17] Boncompagni (Baldassare). – *Della vita e delle opere di Leonardo Pisano.* – Rome (?), 1854.

[18] Boncompagni (Baldassare). – *Intorno ad alcune opere di Leonardo Pisano, matematico del secolo decimoterzo, notizie raccolte de Baldassare Boncompagni.* – Tipografia delle belle arti, 1854.

[19] Bookstein (Abraham) et Fouty (Gary). – A mathematical model for estimating the effectiveness of bigram coding. *Information Processing and Management*, vol. 12, 1976, pp. 111–116.

[20] Bottou (Léon), Howard (Paul G.) et Bengio (Yoshua). – The Z-coder adaptive binary coder. *In : Data Compression Conference 1998*, éd. par Storer (James A.) et Cohn (Martin). pp. 13–22. – IEEE Computer Society Press.

[21] Boyer (R. S.) et Moore (J. S.). – A fast string search algorithm. *Comm. of the ACM*, vol. 20, n° 10, 1977, pp. 762–772.

[22] Bracewell (Norman R.). – Discrete Hartley transform. *J. Opt. Soc. Amer.*, vol. 73, n° 12, 1983.

[23] Bracewell (Norman R.). – The fast Hartley transform. *Proceedings of the IEEE*, vol. 72, n° 8, 1984, pp. 1010–1018.

[24] Bracewell (Norman R.). – *The Hartley Transform.* – Oxford Publicity Press, 1986.

[25] Bracewell (Norman R.), Buneman (O.) et Hao (H.). – Fast two dimensional Hartley transform. *Proceedings of the IEEE*, vol. 74, n° 9, 1986, pp. 1282–1283.

[26] Brassard (Gilles) et Bratley (Paul). – *Algorithmique : conception et analyse.* – Masson, Les Presses de l'Université de Montréal, 1987.

[27] Breiman (L.), Freidman (J. H.), Olshen (R. A.) et Stone (C. J.). – *Classification and Regression Trees.* – Wadsworth, 1984.

[28] Brent (R. P.). – A linear algorithm for data compression. *Australian Computer Journal*, vol. 19, n° 2, 1987, pp. 64–68.

[29] Broden (R.). – RFC #468 FTP data compression, 1973.

[30] Bruckman (Paul S.). – The generalized Zeckendorf theorems. *Fibonacci Quarterly*, vol. 27, 1989, pp. 338–347.

[31] Buro (Michael). – On the maximum length of Huffman codes. *Information Processing Letters*, vol. 45, 1993, pp. 219–223.

[32] Burrows (M.) et Wheeler (D. J.). – *A block sorting data compression algorithm.* – Rapport technique n° SRC-124, Digital Systems Research Center, 1994.

[33] Butterman (Lee) et Memon (Nasir D.). – Error resilient block sorting. *In : Data Compression Conference 2001*, éd. par Storer (James A.) et Cohn (Martin). p. 487. – IEEE Computer Society Press.

[34] Calderbank (Robert), Forney (G. David) et Moayeri (Nader) (édité par). – *Coding and Quantization.* – American Mathematical Society, 1993. DIMACS #14.

[35] Campos (Arturo San Emeterio). – When Fibonacci and Huffman met, 2000. http ://www.arturocampos.com/ac_fib_Huffman.html.

[36] Capocelli (R. M.), Giancarlo (R.) et Taneja (I. J.). – Bounds on the redundancy of Huffman codes. *IEEE Trans. Inf. Theory*, vol. 32, n° 6, 1996, pp. 854–857.

[37] CCITT Study Group XIV. – Draft recommendation T.4 - standardization of group 3 facsimile apparatus for document transmission. http ://www.cis.ohio-state.edu/htbin/rfc/rfc804.html.

[38] Chaitin (Gregory J.). – On the length of programs for computing finite binary sequences. *Journal of the ACM*, vol. 13, 1966, pp. 547–569.

[39] Chaitin (Gregory J.). – On the length of programs for computing finite binary sequences : statistical considerations. *Journal of the ACM*, vol. 16, 1969, pp. 145–159.

[40] Chaitin (Gregory J.). – On the simplicity and speed of programs for computing infinite sets of natural numbers. *Journal of the ACM*, vol. 16, 1969, pp. 407–422.

[41] Chaitin (Gregory J.). – On the difficulty of computations. *IEEE Trans. Info. Theory*, vol. IT-16, 1970, pp. 5–9.

[42] Chaitin (Gregory J.). – A theory of program size formally identical to information theory. *Journal of the ACM*, vol. 22, 1975, pp. 329–340.

[43] Chaitin (Gregory J.). – *Algorithmic Information Theory*. – Cambridge University Press, 1987. QA76.6.

[44] Chen (W.), Smith (C. H.) et Fialick (S. C.). – A fast computational algorithm for the discrete cosine transform. *IEEE Trans. Communications*, vol. 25, 1977, pp. 1004–1009.

[45] Cheung (Gene), McCanne (Steve) et Papadimitriou (Christos). – Software synthesis of variable-length code decoder using a mixture of programmed logic and table lookups. *In : Data Compression Conference 1999*, éd. par Storer (James) et Cohn (Martin). pp. 121–130. – IEEE Computer Society Press.

[46] Chiang (S. W.) et Po (L. M.). – Adaptive lossy LZW algoritm for palettized image compression. *IEEE Letters*, vol. 32, n° 10, 1997, pp. 852–854.

[47] Chou (P. A.), Lookabaugh (T.) et Gray (R. M.). – Optimal pruning with applications to tree structured source coding and modelling. *IEEE Trans. Inf. Theory*, vol. 35, n° 1, 1989, pp. 31–42.

[48] Choueka (Y.), Klein (S. T.) et Perl (Y.). – Efficient variants of Huffman codes in high level languages. *In : Proceedings of the 8th ACM-SIGIR Conference, Montreal*, pp. 122–130.

[49] Church (Alonzo). – On the concept of a random squence. *Bull. Amer. Math. Soc.*, vol. 46, 1940, pp. 130–135.

[50] CIÉ. – *Colorimetry*. – Bureau Central de la CIÉ, 1986, deuxième édition.

[51] Cleary (J. G.) et Teahan (W. J.). – Unbounded length context for PPM. *The Computer Journal*, vol. 40, 1997, pp. 30–74.

[52] Cleary (J. G.) et Witten (I. H.). – Data compression using adaptive coding and partial string matching. *IEEE Trans. Comm.*, vol. 32, n° 4, 1984, pp. 396–402.

[53] Committee (JBIG). – *Final Draft FD14492 : JBIG Final Draft*. – Rapport technique, ITU, 1999.

[54] Compuserve. – *The Graphic Interchange File Format, v89a*. – Rapport technique, Compuserve Incorporated, Columbus, Ohio, 1989.

[55] Cooley (J. W.) et Tukey (O. W.). – An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, vol. 19, 1965, pp. 297–301.

[56] Cormack (G. V.) et Horspool (R. N. S.). – Data compression using dynamic Markov modelling. *The Computer Journal*, vol. 30, 1987, pp. 541–550.

[57] Cormack (Gordon V.) et Horspool (R. Nigel). – Algorithms for adaptive Huffman codes. *Information Processing Letters*, vol. 18, 1984, pp. 159–165.

[58] Cover (Thomas M.). – Enumerative source encoding. *IEEE Trans. Inf. Theory*, vol. 19, n° 1, 1973, pp. 73–77.

[59] Crandall (Richard E.). – *Topics in Advanced Scientific Computation*. – Springer-Verlag, 1996.

[60] Daubechies (Ingrid). – *Ten Lectures on Wavelets*. – SIAM, 1992.

[61] Davis (Harry F.). – *Fourier Series and Orthogonal Functions*. – Dover, 1989.

[62] Delorme (André). – *Psychologie de la perception*. – Études vivantes, Montréal, 1982. BF 311 D44.

[63] Denecker (K.), De Neve (P.) et Lemahieu (I.). – Improved lossless halftone image coding using a fast adaptive context template scheme. *In : Data Compression Conference 1998*, éd. par Storer (James A.) et Cohn (Martin). p. 541. – IEEE Computer Society Press.

[64] Denecker (K.), De Neve (P.) et Lemahieu (I.). – Improved lossless halftone image coding using a fast adaptive context template scheme. *In : Signal Processing Symposium*. pp. 75–78. – IEEE Benelux.

[65] Duhamel (Pierre) et Vetterli (Martin). – Improved Fourier and Hartley transform algorithms : Application to cyclic convolution of real data. *IEEE Trans. Acoust. Speech, Signal Processing*, vol. 35, n° 4, 1987, pp. 818–824.

[66] Elias (P.). – Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, vol. 21, n° 2, 1975, pp. 194–203.

[67] Even (S.) et Rodeh (M.). – Economical encoding of commas between strings. *Comm. of the ACM*, vol. 21, n° 4, 1978, pp. 315–317.

[68] Faller (N.). – An adaptive system for data compression. *In : Records of the 7th Asilomar Conference on Circuits, Systems & Computers*, pp. 393–397.

[69] Fano (R. M.). – *The Transmission of Information*. – Rapport technique n° 65, Research Laboratory of Electronics, MIT, 1944.

[70] Fiala (E. R.) et Greene (D. H.). – Data compression with finite windows. *Comm. of the ACM*, vol. 32, n° 4, April 1989, pp. 490–505.

[71] Foley (James D.), van Dam (Andries), Feiner (Steven K.) et Hughes (John F.). – *Computer Graphics : Principles and Practice*. – Addison-Wesley, 1990.

[72] Fourier (Joseph). – *Théorie Analytique de la Chaleur*. – Firmin-Didot, Père et Fils, Paris, 1822.

[73] Fraenkel (A. S.) et Klein (S. T.). – Bounding the depth of search trees. *The Computer Journal*, vol. 36, n° 7, 1993, pp. 668–678.

[74] Fredkin (E.). – TRIE memory. *Comm. of the ACM*, vol. 3, n° 9, 1960, pp. 490–499.

[75] Friend (R.) et Monsour (R.). – RFC #2395 IP payload compression using LZS, 1998.

[76] Gallager (R. G.). – Variations on a theme by Huffman. *IEEE trans. Inf. Theory*, vol. 24, n° 6, novembre 1978, pp. 668–674.

[77] Gallager (R. G.) et Voorhis (D.C Van). – Optimal source codes for geometrically distributed integer alphabets. *IEEE Trans. Inf. Theory*, vol. 21, n° 3, 1975, pp. 228–230.

[78] Gamma (Erich), Vlissides (John), Johnson (Ralph) et Helm (Richard). – *Design Patterns : Elements of reusable Object Oriented Sofware*. – Addison Wesley Longman, 1994.

[79] Gardos (T.), Arce (G. R.) et Allebach (J. P.). – The optimal ordered dither cell for the binary representation of continuous tone images on a hexagonal lattice. *In : Procs of the 1986 Conf. on Inf. Science & Systems.*

[80] Gödel (Kurt). – On formally undecidable propositions of *Principia Mathematica* and related systems. *Monatshefte für Mathematik und Physik*, vol. 38, 1931, pp. 173–178.

[81] Gersho (A.) et Gray (R. M.). – *Vector Quantization and Signal Compression.* – Kluwer Academic Publishers, 1991.

[82] Gervautz (M.) et Purgathofer (W.). – A Simple Method for Color Quantization : Octree Quantization. *In : New Trends in Computer Graphics*, éd. par Magnenat-Thalmann (N.) et Thalmann (D.), pp. 219–231. – New York, NY, Springer-Verlag, 1988.

[83] Gibson (Jerry D.), Berger (Toby), Lookabaugh (Tom), Lindbergh (Dave) et Baker (Richard L.). – *Digital Compression for Multimedia, Principles and Standards.* – Morgan Kaufmann, 1998.

[84] Gilbert (E. N.) et Moore (E. F.). – Variable length binary encodings. *Bell Systems Technical Journal*, vol. 38, 1959, pp. 933–968.

[85] Golomb (Solomon W.). – Run length encodings. *IEEE Trans. Inf. Theory*, vol. 12, n° 3, 1966, pp. 399–401.

[86] Graf (Siegfried) et Luschgy (Harald). – *Foundations of Quantization for Probability Distributions.* – Springer, 2000. LNM #1730.

[87] Graham (Ronald), Knuth (Donald E.) et Patashnik (Oren). – *Concrete Mathematics : A Foundation for Computer Science.* – Adisson-Wesley, 1994.

[88] Grimm (R. E.). – The autobiography of Leonardo Pisano. *Fibonacci Quarterly*, vol. 11, 1973, pp. 199–104.

[89] Haar (Alfred). – Zur theorie der orthogonalen funktionsystem. *Math. Annal.*, vol. 69, 1910, pp. 331–371.

[90] Hartley (R. V. L.). – Transmission of information. *Bell System Technical Journal*, vol. 7, 1928, pp. 535–563.

[91] Hartley (R. V. L.). – A more symmetrical Fourier analysis applied to transmission problems. *Proc. I.R.E*, vol. 30, n° 3, 1942.

[92] Haykin (Simon). – *Neural Networks.* – Macmillan, 1994.

[93] Heckbert (Paul). – *Color Image Quantization for Frame Buffer Display.* – Rapport technique, Machine Architecture Group, MIT, 1980.

[94] Heckbert (Paul). – Color image quantization for frame buffer display. *In : SIGGraph 82*, pp. 297–307.

[95] Henri (Pierre). – *La vie et l'œuvre de Louis Braille, inventeur de l'alphabet des aveugles, 1809–1852.* – Les Presses Universitaires de France, 1952. 920.96177 H518v.

[96] Hilbert (D.). – Über die stetige abbildung einer linie auf ein flachenstücl. *Math. Ann.*, vol. 38, 1891, pp. 459–460.

[97] Hirschberg (Daniel S.) et Lelewer (Debra A.). – Efficient decoding of prefix codes. *Comm. of the ACM*, vol. 33, n° 4, 1990, pp. 449–459.

[98] Hoggatt (Verner E.), Cox (Nanette) et Bicknell (Majorie). – A primer for the Fibonacci numbers : Part XII. *Fibonacci Quarterly*, vol. 11, 1973, pp. 317–331.

[99] Hopcroft (John. E.) et Ullman (Jeffrey D.). – *Introduction to automata theory, languages, and computation.* – Addison-Wesley, 1979. QA267.H56.

[100] Hu (T. C.) et Tucker (A. C.). – Optimum computer search trees and variable length alphabetic codes. *SIAM*, vol. 22, 1971, pp. 225–234.

[101] Huffman (David). – A method for the construction of minimum redundancy codes. *Proceedings of the I.R.E.*, vol. 40, n° 9, 1952, pp. 1098–1101.

[102] IBM Corporation. – HASP-II system manual, 1971.

[103] Ifrah (Georges). – *Histoire Universelle des Chiffres.* – Robert Laffont, 1981.

[104] Jackobsson (M.). – Compression of character strings by an adaptive dictionary. *BIT*, vol. 25, n° 4, 1985, pp. 593–603.

[105] Jayant (Nuggehally S.). – Adaptive quantization with one-bit memory. *Bell System Technical Journal*, vol. 49, 1970, pp. 321–340.

[106] Jayant (Nuggehally S.). – Adaptive quantization with one word memory. *Bell System Technical Journal*, vol. 52, 1973, pp. 1119–1144.

[107] Jones (D. W.). – Application of splay trees to data compression. *Comm. of the ACM*, vol. 31, 1988, pp. 996–1007.

[108] Kahn (David). – *La guerre des codes secrets : des hiéroglyphes à l'ordinateur.* – Interéditions, 1980.

[109] Karlgren (H.). – Representation of text strings in binary computers. *Bit*, vol. 3, 1963, pp. 52–59.

[110] Katona (Gyula O. H.) et Nemetz (Tibor O. H.). – Huffman codes and self-information. *IEEE Trans. Inf. Theory*, vol. 22, n° 3, 1976, pp. 337–340.

[111] Kimberling (Clark). – Zeckendorf number systems and associated partitions. *Fibonacci Quarterly*, vol. 29, 1991, pp. 120–123.

[112] Kiyohara (Junya) et Kawabata (Tsutomu). – A note on Lempel-Ziv-Yokoo algorithm. *IEICE Trans. Fundamentals*, vol. E79-A, n° 9, 1996, pp. 1460–1463.

[113] Knuth (D. E.), Morris (J. H.) et Pratt (V. R.). – Fast pattern matching in strings. *SIAM Journal on Computing*, vol. 6, n° 2, 1977, pp. 240–267.

[114] Knuth (Donald E.). – Dynamic Huffman coding. *Journal of Algorithms*, vol. 6, 1985, pp. 163–180.

[115] Kolmogorov (Andrey N.). – *Grundbegriffe der Wahrscheinlicjkeitsrechnung.* – Springer-Verlag, 1939. Traduit en anglais, *Foundation of the Theory of Probability*, Chelsea, 1956.

[116] Kolmogorov (Andrey N.). – On tables of random numbers. *Sankhyā, The Indian Journal of Statistics, Series A*, vol. 25, 1963, pp. 369–376.

[117] Kolmogorov (Andrey N.). – Three approaches to the quantitative defintion of information. *Problems of Information Transmission*, vol. 1, n° 1, 1965, pp. 1–7.

[118] Kolmogorov (Andrey N.). – Logical basis for informaiton theory and probability theory. *IEEE Trans. Inf. Theory*, vol. IT-14, 1968, pp. 662–664.

[119] Kolmogorov (Andrey N.). – Some theorems on algorithmic entropy and the algorithmic quantity of information. *Uspekhi Mat. Nauk*, vol. 23, n° 2, 1968, p. 201.

[120] Kolmogorov (Andrey N.). – On the logical foundations of information theory and probability theory. *Problems on Information Transmission*, vol. 5, 1969, pp. 1–4.

[121] Kraft (L. G.). – *A device for Quantizing, Grouping and Coding Amplitude Modulation Pulses*. – Thèse, Massachusetts Institute of Technology, 1949.

[122] Körner (T. W.). – *Fourier Analysis*. – Cambridge University Press, 1986. QA403.5.K67 1986.

[123] Kwong (C. P.) et Shiu (K. P.). – Structured fast Hartley transform algorithms. *IEEE Trans. Acoust. Speech, Signal Processing*, vol. 34, n° 4, 86, pp. 1000–1002.

[124] Larmore (Lawrence L.) et Hirschberg (Daniel S.). – A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, vol. 37, n° 3, July 1990, pp. 464–473.

[125] Lehmer (D. H.). – Teaching combinatorial tricks to a computer. *In : Symposium Applied Mathematics, vol. 10*. – AMS.

[126] Lempel (Abraham) et Gavish (Amnon). – Match length functions for data compression. *IEEE Trans. Inf. Theory*, vol. 42, n° 5, 1996, pp. 1375–1380.

[127] Li (M.) et Vitányi (P.). – *Introduction to Kolmogorov Complexity and its Applications*. – Springer, 1997, 2nd édition. QA267.7.L5 1997.

[128] Linde (Y.), Buzo (A.) et Gray (R. M.). – An algorithm for vector quantiser design. *IEEE Trans. Comm.*, vol. 28, n° 1, 1980, pp. 84–95.

[129] Lloyd (Stuart P.). – Least means square quantization in PCM. *IEEE Trans. Inf. Theory*, vol. 28, n° 3, 1982, pp. 127–135.

[130] Louchard (Guy) et Szpankowski (Wojciech). – On the average redundancy rate of the Lempel-Ziv code. *IEEE Trans. Inf. Theory*, vol. 43, n° 1, 1997, pp. 2–8.

[131] Mabee (Carleton). – *American Leonardo : A Life of Samuel F. B. Morse*. – Hippocrene Books, 1969.

[132] Mallat (Stéphane). – *A Wavelet Tour of Signal Processing*. – Academic Press, 1997.

[133] Manber (Udi). – *A text compression scheme that allows fast searching directly in the compressed file*. – Rapport technique n° 93–07, Dept. of Computer Science, University of Arizona, 1993.

[134] Max (Joel). – Quantizing for minimum distortion. *I.R.E. Trans. Inf. Theory*, vol. 6, n° 1, 1960, pp. 7–12.

[135] McIntyre (David R.) et Pechura (Micheal A.). – Data compression using static Huffman code-decode tables. *Comm. of the ACM*, vol. 28, n° 6, June 1985, pp. 612–616.

[136] McMillan (B.). – Two inequalities implied by unique decypherability. *I.R.E Transf. Inf. Theory*, vol. 2, 1955, pp. 115–116.

[137] Memon (Nasir D.) et Wu (Xiaolin). – Recent developments in context-based predictive techniques for lossless image compression. *The Computer Journal*, vol. 40, 1997, pp. 127–136.

[138] Meyer (Yves). – *Wavelets : Algorithms and Applications*. – SIAM, 1994. Traduit à l'anglais et augmenté par Robert D. Ryan.

[139] Milidiú (Ruy Luiz), Pessoa (Artur Alves) et Laber (Eduardo Sany). – In-place length-restricted prefix coding. *In : String Processing and Information Retrieval*, pp. 50–59.

[140] Miller (V. S.) et Wegman (M. N.). – Variation on a theme by Ziv and Lempel. *In : Combinatorial Algorithms on Words*, éd. par Apostolico (A.) et Galil (Z.). pp. 131–140. – Springer-Verlag.

[141] Mitchell (J. L), Pennebaker (W. B.), Fog (C. E.) et LeGall (D. J.). – *MPEG Video Compression Standard.* – Chapman & Hall, 1997.

[142] Mitsa (Theophano) et Parker (Kevin J.). – Digital halftoning using a blue noise mask. *In : ICASSP '91 : 1991 International Conference on Acoustics, Speech, and Signal Processing.* IEEE, pp. 2809–2812. – Toronto, Canada, mai 1991.

[143] Mitsa (Theophano) et Parker (Kevin J.). – Digital halftoning using a blue-noise mask. *Journal of the Optical Society of America A*, vol. 9, n° 11, novembre 1992, pp. 1920–1929.

[144] Mitsa (Theophano) et Parker (Kevin J.). – Power-spectrum shaping of halftone patterns and its effect on visual appearance. *In : ICASSP-92 : 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing.* IEEE, pp. 193–196. – San Francisco, California, mars 1992.

[145] Moffat (Alistair). – Implementing the PPM data compression scheme. *IEEE Trans. Comm.*, vol. 38, n° 10, 1990, pp. 1917–1921.

[146] Moffat (Alistair) et Katajainen (Jyrki). – In place calculation of minimum redundancy codes. *In : Proceedings of the Workshop on Algorithms and Data Structures.* pp. 303–402. – Springer Verlag. LNCS 955.

[147] Moffat (Alistair) et Turpin (Andrew). – Efficient construction of minimum redundancy codes for large alphabets. *In : Proceeding of the Data Compression Conference 1995*, éd. par Storer (James A.) et Cohn (Martin). pp. 170–179. – IEEE Computer Society Press.

[148] Moffat (Alistair) et Turpin (Andrew). – On the implementation of minimum redundancy prefix codes. *IEEE Trans. Comm*, vol. 45, n° 10, October 1997, pp. 1200–1207.

[149] Morrison (D. R.). – PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, vol. 15, n° 4, 1968, pp. 514–534.

[150] Munsell (A. H.). – *A Color Notation.* – Boston, 1905.

[151] Munsell (A. H.). – *Atlas of the Munsell Color System.* – Boston, 1915.

[152] Munson (David C.). – Editorial. *IEEE Trans. Image Processing*, vol. 5, n° 1, 1996.

[153] Ohta (Y.). – *A region-oriented image-analysis system by computer.* – Thèse de PhD, Kyoto University, 1980.

[154] Olson (Harry F.). – *Music, Physics and Engineering.* – Dover, 1967, deuxième édition. Portait le titre de *Musical Engineering* dans les éditions précédentes.

[155] O'Neill (Mark). – Faster than fast Fourier. *Byte*, no4, 1988, pp. 293–298(?). – Réédité dans *The Best of Byte*, aux pages 117–122.

[156] Pall (G.). – RFC #2118 Microsoft point-to-point compression (MPPC) protocol, 1997.

[157] Pennebaker (W. B.) et Mitchell (J. L.). – *JPEG Still Image Data Compression Standard.* – Van Nostrand Reinhold, New York, 1993.

[158] Pigeon (Steven). – *A Fast Image Compression Method based on the Fast Hartley Transform.* – Rapport technique n° HA6156000-961220-01, Speech & Image Processing Lab 6, AT&T Research, 1996.

[159] Pigeon (Steven). – Flatland, ou comment réduire une image GIF en modifiant l'algorithme LZW. *Journal l'Interactif*, mars 1996.

[160] Pigeon (Steven). – An optimizing lossy generalization of LZW. *In : Data Compression Conference 2001*, éd. par Storer (James A.) et Cohn (Martin). p. 509. – IEEE Computer Society Press.

[161] Pigeon (Steven). – Start/stop codes. *In : Data Compression Conference*, éd. par Storer (James A.) et Cohn (Martin). p. 511. – IEEE Computer Society Press.

[162] Pigeon (Steven) et Bengio (Yoshua). – *A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols.* – Rapport technique n° 1081, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, 1997.

[163] Pigeon (Steven) et Bengio (Yoshua). – *A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, Revisited.* – Rapport technique n° 1095, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, 1997.

[164] Pigeon (Steven) et Bengio (Yoshua). – Memory-efficient adaptive Huffman coding. *Doctor Dobb's Journal*, no290, 1998, pp. 131–135.

[165] Pigeon (Steven) et Bengio (Yoshua). – A memory-efficient Huffman adaptive coding algorithm for very large sets of symbols. *In : Proceeding of the Data Compression Conference 1998*, éd. par Storer (James A.) et Cohn (Martin). p. 568. – IEEE Computer Society Press.

[166] Pigeon (Steven) et Bengio (Yoshua). – Binary pseudowavelets and applications to bilevel image processing. *In : Data Compression Conference 1999*, éd. par Storer (James A.) et Cohn (Martin). pp. 364–373. – IEEE Computer Society Press.

[167] Pike (J.). – Text compression using a 4-bit coding system. *Computer Journal*, vol. 24, n° 4, 1981, pp. 324–330.

[168] Pisano (Léonardo). – *Liber Abaci.* – (éditeur inconnu), 1228, seconde édition.

[169] Polybe. – *Histoire.* – La Pléïade, Gallimard, NRF, 1985.

[170] Pratt (William K.). – *Digital Image Processing.* – John Wiley & Sons, 1991, deuxième édition.

[171] Press (W. H.), Teukolsky (S. A.), Vetterling (W. T.) et Flannery (B. P.). – *Numerical Recipes in C.* – Cambridge University Press, 1995.

[172] Ramabadran (Tenkasi V.). – A coding scheme for m-out-of-n codes. *IEEE Trans. Comm.*, vol. 38, n° 8, 1990.

[173] Rao (K. R.) et Yip (P.). – *Discrete cosine Transform.* – Academic Press, 1990.

[174] Rimmer (Steve). – *Bit Mapped Graphics.* – Windcrest/McGraw-Hill, 1990.

[175] Rocach (Arie). – Optimal computation, on computer, of Fibonacci numbers. *Fibonacci Quarterly*, vol. 34, n° 5, 1996, pp. 436–437.

[176] Rodeh (M.), Pratt (V. R.) et Even (S.). – Linear algorithm for data compression via string matching. *Journal of the ACM*, vol. 28, n° 1, 1981, pp. 16–24.

[177] Roman (Steven). – *Coding and Information Theory.* – Springer-Verlag, 1992.

[178] Sadakane (K.). – A fast algorithm for making suffix arrays and for the Burrows-Wheeler transform. *In : Data Compression Conference 1998*, éd. par Storer (James A.) et Cohn (Martin). pp. 129–138. – IEEE Computer Society Press.

[179] Sadakane (Kunihiko) et Imai (Hiroshi). – Improving the speed of LZ-77 compression by hashing and suffix sorting. *IEICE Trans. Fundamentals*, vol. E83-A, n° 12, 2000, pp. 2689–2698.

[180] Sadeh (Ilan). – Universal data compression algorithm based on approximate string matching. *Probability in the Engineering and Informational Science*, vol. 10, n° 4, 1996, pp. 465–486.

[181] Salomon (David). – *Data Compression : The Complete Reference.* – Sprigner-Verlag, 2000.

[182] Savari (Serap A.). – Redundancy of the Lempel-Ziv incremental parsing rule. *IEEE Trans. Inf. Theory*, vol. 43, n° 1, 1997, pp. 9–21.

[183] Savari (Serap A.). – Redundancy of the Lempel-Ziv-Welch code. *In : Data Compression Conference 1997*, éd. par Storer (James A.) et Cohn (Martin). pp. 191–200. – IEEE Computer Society Press.

[184] Sayood (Khalid). – *Introduction to Data Compression.* – Morgan-Kaufmann, 2000, deuxième édition.

[185] Schalkwijk (J. Pieter). – An algorithm for source coding. *IEEE Trans. Inf. Theory*, vol. 18, n° 3, 1972, pp. 395–399.

[186] Seroussi (Gadiel) et Weinberger (Marcelo J.). – On adaptive strategies for an extended family of golomb-type codes. *In : Proceedings of the Data Compression Conference 1997*, éd. par Storer (James A.) et Cohn (Martin). pp. 131–140. – IEEE Computer Society Press.

[187] Seward (Julian). – On the performance of BWT sorting algorithms. *In : Data Compression Conference 2000*, éd. par Storer (James A.) et Cohn (Martin). pp. 173–182. – IEEE Computer Society Press.

[188] Shannon (Claude E.). – A mathematical theory of communication I. *Bell System Technical Journal*, vol. 27, n° 7, 1948, pp. 379–423.

[189] Shannon (Claude E.). – A mathematical theory of communication II. *Bell System Technical Journal*, vol. 27, n° 10, 1948, pp. 623–656.

[190] Siemiński (Andrzej). – Fast decoding of the Huffman codes. *Information Processing Letters*, vol. 26, January 1988, pp. 237–241.

[191] Singh (Simon). – *The Code Book.* – Anchor Books, 1999. Z103.S56 1999.

[192] Slepian (David). – *Key Papers in The Development of Information Theory.* – IEEE Press, 1974.

[193] Solomonoff (Ray J.). – *A Preliminary Report on a General Theory of Inductive Inference.* – Rapport technique n° V-131, Zator Co., Cambridge, Mass, 1960.

[194] Solomonoff (Ray J.). – A formal theory of inductive inference I and II. *Information and Control*, vol. 7, 1964, pp. 1–22,224–254.

[195] Solomonoff (Ray J.). – Complexity-based induction systems : Comparisons and convergence theorems. *IEEE Trans. on Inf. Theory*, vol. IT-24, 1978, pp. 422–432.

[196] Standish (Thomas A.). – *Data Structures Techniques.* – Addison-Wesley, 1980.

[197] Stein (Sherman K.). – *Mathematics : The Man-Made Universe.* – McGraw-Hill, 1999.

[198] Stockdale (Jim) et Stockdale (Sybil). – *In Love and War.* – Naval Institute Press, 1990.

[199] Storer (J. A.) et Syzmanski (T. G.). – Data compression via textual substitution. *Journal of the ACM*, vol. 29, 1982, pp. 928–951.

[200] Stout (Quentin F.). – Improved prefix encodings of the natural numbers. *IEEE Trans. Inf. Theory*, vol. 26, 1980, pp. 607–609.

[201] Stucki (P.). – *Digital Halftoning.* – MIT Press, 1987.

[202] Tanaka (Hatsukazu). – Data structure of Huffman codes and its application to efficient coding and decoding. *IEEE Trans. Inf. Theory*, vol. 33, n° 1, January 1987, pp. 154–156.

[203] Tarjan (Robert Endre). – *Data Structures and Network Algorithms.* – SIAM, 1983, *Regional Conference Series in Applied Mathematics.*

[204] Terletskii (Ya P.). – *Physique Statistique.* – Presses de l'Université du Québec, 1975. (Traduit du Russe).

[205] Teuhola (J.) et Raita (T.). – Application of a finite state model to text compression. *The Computer Journal*, vol. 36, n° 7, 1984, pp. 306–315.

[206] Thomas (S.W), McKie (J.), Davis (S.), Turkowski (K.), Woods (J. A) et Orost (J. W.). – Compress v.4.0. program and documentation. Manuel de l'utilisateur, documentation online.

[207] Tischer (P.). – A modified Lempel-Ziv-Welch data compression scheme. *Australian Computer Journal*, vol. 9, n° 1, 1987, pp. 262–272.

[208] Tropper (Richard). – Binary-coded text : A text compression method. *Byte*, vol. 7, n° 4, 1982, pp. 398–413.

[209] Tunstall (B. P.). – *Synthesis of Noiseless Compression Codes.* – Thèse de PhD, Georgia Institute of Technology, September 1967.

[210] Turing (Allan Mathison). – On computable numbers, with an application to the *Entscheidungsproblem. Proceedings, London Mathematical Society*, vol. 42, 1937, pp. 230–265.

[211] Turner (J. C.) et Robb (T. D.). – Generalizations of the dual Zeckendorf integer representation theorems : Discovery by Fibonacci trees and word patterns. *Fibonacci Quarterly*, vol. 28, 1990, pp. 230–239.

[212] Ulichney (Robert A.). – Dithering with blue noise. *Proceedings of the IEEE*, vol. 76, n° 1, janvier 1988, pp. 56–79.

[213] Varn (Ben). – Optimal variable length codes (arbitrary symbol cost and equal word probability. *Information and Control*, vol. 19, 1971, pp. 289–301.

[214] Villasenor (John). – Alternatives to the discrete cosine transform for irreversible tomographic image compression. *IEEE Trans. Medical Imaging*, vol. 12, n° 4, décembre 1993.

[215] Villasenor (John). – Tomographic image compression using multidimensional transforms. *J. Inf. Processing and Management*, vol. 30, 1994, pp. 817–824.

[216] Vitter (Jeffrey Scott). – Design and analysis of dynamic Huffman codes. *Journal of the ACM*, vol. 34, n° 4, October 1987, pp. 823–843.

[217] Vitter (Jeffrey Scott). – Algorithm 673 : Dynamic Huffman coding. *ACM Trans. Math. Software*, vol. 15, n° 2, June 1989, pp. 158–167.

[218] Wallace (G. K.). – The JPEG still picture compression standard. *Comm. of the ACM*, vol. 34, n° 4, 1991, pp. 31–44.

[219] Weinberger (Marcelo J.), Seroussi (Gadiel) et Sapiro (Guillermo). – LOCO-I : A low complexity, context-based, lossless image compression algorithm. *In : Data Compression Conference 1996*, éd. par Storer (James A.) et Cohn (Martin). pp. 140–149. – IEEE Computer Society Press.

[220] Weisstein (Eric). – *CRC concise encyclopedia of Mathematics.* – Chapman & Hall/CRC, 1999. QA5.W45 1998.

[221] Welch (Terry A.). – A technique for high performance data compression. *Computer*, June 1984, pp. 8–19.

[222] Williams (Ross). – *Adaptive Data Compression.* – Kluwer Academic Publishers, 1991. *Ce livre est une version de la thèse de doctorat de l'auteur.*

[223] Williams (Ross). – An extremely fast Ziv-Lempel data compression algorithm. *In : Data Compression Conference 1991*, éd. par Storer (James A.) et Cohn (Martin). pp. 351–360. – IEEE Computer Society Press.

[224] Wolf (Misha), Whistler (Ken), Wicksteed (Charles), Davis (Mark) et Freytag (Asmus). – *A Standard Compression Scheme for Unicode.* – Rapport technique n° 6, Unicode Consortitium, 2000.

[225] Wood (Derick). – *Theory of Computation.* – John Wiley & Sons, 1987.

[226] Wu (Xiaolin). – *Efficient Statistical Computations for Optimal Color Quantization*, pp. 126–133. – Morgan Kaufmann, 1991.

[227] Wu (Xiaolin) et Memon (Nasir D.). – CALIC : A context-based adaptive lossless image coding scheme. *IEEE Trans. Comm.*, vol. 45, n° 5, 1996, pp. 437–444.

[228] Wu (Xiaolin), Memon (Nasir D.) et Sayood (Khalid). – A context based adaptive lossless/nearly lossless coding scheme for continuous tone images, 1995. ISO Working Document ISO/IEC SC29/WG1/N256.

[229] Yamamoto (Hirosuke). – A new recursive universal code of the positive integers. *IEEE Trans. Inf. Theory*, vol. 46, n° 2, 2000, pp. 717–723.

[230] Yeung (Raymond W.). – Alphabetic codes revisited. *IEEE Trans. Inf. Theory*, vol. 37, n° 3, May 1991, pp. 564–572.

[231] Yokoo (Hidetoshi). – Improved variations relating the Ziv-Lempel and Welch-type algorithms for sequential data compression. *IEEE Trans. Inf. Theory*, vol. 38, n° 1, 1992, pp. 73–81.

[232] Zipf (George Kinsley). – *Selective studies and the principle of relative frequency in language.* – Harvard University Press, 1932.

[233] Zipf (George Kinsley). – *Psycho-biology of languages.* – Houghton-Mifflin, 1935.

[234] Zipf (George Kinsley). – *Human Behavior and the Principle of Least Effort.* – Addison-Wesley, 1949.

[235] Ziv (J.) et Lempel (A.). – A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, vol. 23, n° 3, 1977, pp. 337–343.

[236] Ziv (J.) et Lempel (A.). – Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, vol. 24, n° 5, September 1978, pp. 530–536.

Index

Vitæ

Steven Pigeon a fait ses études du baccalauréat à l'université de Montréal, d'où il gradua en 1995. Ses études de maîtrise, aussi faites à l'université de Montréal, portaient sur l'utilisation des ordinateurs à logique programmable dans les applications de traitement d'image et d'apprentissage automatique grâce aux réseaux de neurones. Il obtint sa maîtrise en 1996. Ses recherches doctorales portent sur la compression de données, en particulier le codage des entiers et la compression d'image. Il reçu son Ph.D. de l'université de Montréal en 2002.

Depuis l'acquisition d'une propriété champêtre sise sur quelques acres de forêt, il aime se retirer de la vie mondaine dans son « hermitage ». Il y poursuit ses intérêts pour la lecture, l'égyptologie et l'astronomie, un passe temps adéquat pour un oiseau de nuit.

Liste des publications

Jocelyn Cloutier, Éric Cosatto, Steven Pigeon, François R. Boyer, Patrice Y. Simard — VIP : an FPGA-based Processor for Image Processing and Neural Networks — *Proceedings of the fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Lausane, Suisse, Fév. 1996.

Steven Pigeon — A Fast Image Compression Method based on the Fast Hartley Transform — rapport technique n° HA6156000-961220-01, AT&T Research, Speech & Image Processing Lab 6, Holmdel, 1996

Steven Pigeon — Flatland, ou comment réduire une image GIF en modifiant l'algorithme LZW — Journal l'Interactif, mars 1996

Steven Pigeon, Yoshua Bengion — A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols — Rapport technique n° 1081, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1997

Steven Pigeon, Yoshua Bengion — A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, Revisited — Rapport technique n° 1095, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1997

Steven Pigeon, Yoshua Bengion — A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols — *Proceedings of the Data Compression Conference 1998*, IEEE Computer Press, p. 568

Steven Pigeon, Yoshua Bengion — Memory-Efficient Adaptive Huffman Coding — Doctor Dobb's Journal, n° 290, 1998, p. 131–135

Steven Pigeon, Yoshua Bengion — Binary pseudowavelets and Applications to Bilevel Image Processing — *Data Compression Conference 1999*, IEEE Computer Society Press, 1999, p. 364–373

Steven Pigeon — An Optimizing Lossy Generalization of LZW — *Data Compression Conference 2001*, IEEE Computer Society Press, 2001, p. 509

Steven Pigeon, Léon Bottou — Masked Wavelets : Applications to image compression — *Data Compression Conference 2001*, IEEE Computer Society Press, 2001, p. 510

Steven Pigeon — Start/Stop Codes — *Data Compression Conference 2001*, IEEE Computer Society Press, 2001, p. 511

Steven Pigeon — Unconstrained vector lengths in Fast Wavelet Transforms — *Data Compression Conference 2001*, IEEE Computer Society Press, 2001, p. 512

Steven Pigeon — Image compression with wavelets — Doctor Dobb's Journal, n° 302, Août 1999, pp. 111–115

Steven Pigeon — Sorting lists and the Radix Sort — Doctor Dobb's Journal, n° 336, Mai 2002, pp. 89–94

Soumis ou à paraître

Le chapitre 3 sur le codage de Huffman, in *The Handbook of Lossless Data Compression*, Khalid Sayood, éd. Academic Press, 2002. *A paraître*

Steven Pigeon — Optimizing C/C++ code for speed — *à paraître dans* C/C++ User's Journal, 2002.

Steven Pigeon — Optimal Golomb Coding — *soumis aux* Information Processing Letters.

Steven Pigeon — Phase-in Codes Revisited — *soumis aux* Information Processing Letters.

Steven Pigeon — Taboo Codes : New classes of Universal Codes — *soumis au* SIAM Journal of Computing.

Brevets

US patent #6,058,214. Léon Bottou, Steven Pigeon. *Compression of Partially Masked Images*, émis à AT&T Research, 2 Mai 2000. *Une version de ce brevet est sous étude au bureau européen des brevets.*