Compression de textes en langue naturelle

Claude Martineau

Université de Marne-la-Vallée

Institut Gaspard-Monge

Thèse de Doctorat

Spécialité :Informatique Fondamentale

présentée par

Claude Martineau

pour l'obtention du titre de

Docteur de l'université de Marne-la-Vallée

Compression de textes en langue naturelle

Soutenue le vendredi 7 décembre 2001 devant le jury composé de :

Mr. Maxime Crochemore (Directeur)
Mr. Éric Laporte
Mr. Shmuel Tomi Klein (Rapporteur)
Mr. Denis Maurel (Rapporteur)
Mr. Gérard Plateau (Rapporteur)
Mr. Marc Zipstein

À mes Grands-Parents et à mes Parents.

Remerciements.

Je tiens à remercier tout particulièrement Maxime Crochemore, de m'avoir accueilli avec gentillesse au sein de l'IGM et de m'avoir aidé par ses précieux conseils.

J'adresse tous mes vifs remerciements à Denis Maurel, Gérard Plateau et Shmuel Tomi Klein d'avoir accepté d'être rapporteurs de ma thèse.

Je remercie vivement Éric Laporte d'avoir bien voulu être membre du jury.

Je remercie infiniment Marc Zipstein qui a consciencieusement lu et relu ma thèse.

Je dois remercier également Elsa Sklavounou pour ses encouragements et sans laquelle je n'aurais pu tester mes algorithmes de compression de textes sur le grec.

Enfin, je remercie beaucoup tous ceux qui au sein de l'IGM et en dehors m'ont aidé par leurs encouragements.

Résumé

Nous étudions dans cette thèse les méthodes de compression de données dans le cadre de leur utilisation pour le traitement de textes en langue naturelle. Nous nous intéressons particulièrement aux algorithmes utilisant les mots comme unité de base.

Nous avons développé des algorithmes qui permettent de diviser la taille originale du texte par un coefficient de l'ordre de 3,5 en conservant l'accès direct (via un index) au texte sous forme comprimée.

L'ensemble des mots qui composent un texte (le *lexique*) n'est pas, *a priori* connu. Compresser efficacement un texte nécessite donc de compresser également le lexique des mots qui le constitue. Dans ce but, nous avons mis au point une méthode de représentation des lexiques qui permet, en lui appliquant une compression à base de chaînes de Markov, d'obtenir des taux de compression très importants.

Les premiers algorithmes dédiés à la compression de textes en langue naturelle ont été élaborés dans le but d'archiver de volumineuses bases de données textuelles, pouvant atteindre plusieurs dizaines de gigaoctets, pour lesquelles la taille du lexique est très inférieure à celle des données. Nos algorithmes peuvent s'appliquer aussi aux textes de taille usuelle (variant d'une cinquantaine de Ko à quelques Mo) pour lesquels la taille du lexique représente une part importante de celle du texte.

Abstract

In this Ph.D. thesis we investigate several data compression methods on text in natural language. Our study is focused on algorithms that use the *word* as the basic units, they are usally called word-based text compression algorithms.

We have developped algorithms that allow to divide original size of the text by an average factor of 3.5 and keeps (medium an index) direct access to the compressed form of the text.

The set of words of a text, (the *lexicon*) is not *a priori* known. An efficient compression of the text requires an efficient compression of its lexicon. For this purpose, we have developped a compact representation of the lexicon that allows, by the application of Markov chain based compression algorithms, to get very high compression rates.

The early algorithms dedicated to compress text in natural language have been elaborated to process very large text databases in which the size of the lexicon is very small versus the data one. Our algorithms can be apply also to every day text size (from some fifty Ko up to some Mo) for which the size of the lexicon is an important part of the size of the text.

Table des matières

Liste des tableaux

Table des figures

Introduction

La compression de données a pour objet de réduire la taille des données dans le but de réduire l'espace disque nécessaire à leur mémorisation et le temps de transmission à travers les réseaux. Elle permet d'en faciliter la diffusion par la possibilité d'utiliser des supports magnétiques de faible capacité comme les disquettes, d'améliorer l'archivage de données volumineuses.

Lorsque nous parlons de compression de données, nous ne tenons pas compte de la nature des données traitées : texte, son, photo, dessins vectoriels.

Pendant de nombreuses années, la compression de données a été utilisée pour les besoins propres des informaticiens et s'appliquait donc surtout à des codes sources ou du code objet parfois, des documentations en langue naturelle. Les premiers algorithmes de compression devaient donc traiter des données à une dimension. Peu à peu, on a eu besoin de traiter des images, d'abord en niveaux de gris, puis en couleurs, on a alors développé des algorithmes de compression spécialement adaptés à ces données bi-dimensionnelles.

Les textes en langue naturelle faisant partie des données à une dimension, ils sont encore aujourd'hui, le plus souvent compressés par des algorithmes généralistes de compression de données. Depuis une dizaine d'années seulement sont apparus, quasi uniquement à l'état de prototype, des algorithmes spécialement adaptés au traitement des textes en langue naturelle. L'objet de cette thèse est d'étudier ce type d'algorithmes de compression et d'en proposer des améliorations, de comparer les taux de compression qu'ils permettent d'obtenir avec ceux obtenus avec les algorithmes classiques.

Un texte en langue naturelle étant écrit avec les mots d'une langue naturelle donnée, la plupart des algorithmes adaptés au traitement de ce type de texte considèrent celui-ci comme une suite de mots d'une langue naturelle, séparés par des séquences plus ou moins courtes de caractères de ponctuation que nous appellerons *mots séparateurs* par opposition aux *mots linguistiques*.

Cette façon de considérer le texte oblige à gérer deux niveaux de compressions. D'une part, celui des mots (linguistiques et séparateurs) dont l'unité de base du traitement est le caractère. D'autre part, celui du texte pour lequel les mots deviennent l'unité de base de traitement.

Les algorithmes de compression à base de mots étant le plus souvent des adaptations ou des combinaisons des algorithmes à base de caractères, nous exposerons dans une première partie les algorithmes de compression classiques à base de caractères, puis dans une seconde, les algorithmes de compression à base de mots qui en sont issus.

Dans le premier chapitre, nous rappelons comment un texte est représenté en machine et nous en déduisons ce que l'on entend par compression de texte. Nous présentons un modèle général permettant d'élaborer un algorithme de compression de texte. On définit au cours de ce chapitre, les notions de taux de compression, de code à longueur fixe, de code à longueur variable et celle de décodabilité qui en découle. On définit, également les notions d'information et d'entropie. Toutes ces notions sont étudiées à travers la présentation des codes Morse et Shannon-Fano.

Nous consacrons les deux chapitres suivant aux algorithmes de compression de type statistique. Le chapitre 2 s'intéresse au codage de Huffman [29] statique qui est un codage de type statistique. On y rappelle les notions d'arbre de Huffman et d'arbre de Huffman de hauteur minimale due à Gallager [23]. Nous nous intéressons également à diverses implémentations du codage de Huffman utilisant la notion d'automate qui en ont été faites par Tanaka [61], Simiensky [60] et Zipstein[71]. Dans la mesure où ses performances sont comparables à celles de l'algorithme de Huffman statique, nous ne parlons que brièvement de sa version adaptative.

Le chapitre 3 est consacré à l'étude du codage arithmétique dont le principe est dû à Elias [25] mais dont une première implémentation pratique est due à Witten Neal et Cleary [70]. L'intérêt du codage arithmétique réside dans le fait qu'il permet une nette séparation du modèle de probabilité et du codeur. Ainsi nous présentons les algorithmes de la famille PPM (*Prediction by Partial Matching*) dus notamment à Cleary et Witten [12], qui utilise un modèle fondé sur la notion de chaîne de Markov. Nous y présentons également le modèle d'Abrahamson [1] et l'algorithme de Burrows-Wheeler [10] dont les performances sont proches de celles de PPM.

Le chapitre 4 rappelle le fonctionnement des algorithmes de compressions par facteurs issus des travaux de Ziv et Lempel [36, 37] et habituellement classés en deux familles nommées : type LZ77 et type LZ78. Après avoir décrit les versions originales des algorithmes de type LZ78 à base de dictionnaire et de type LZ77 qui fonctionne à l'aide d'une fenêtre coulissante, nous en présentons des versions améliorées LZW [67] et LZSS [] ainsi que gzip qui est une version améliorée de LZSS due à Jean-Loup Gailly [22].

La seconde partie de cette thèse consacrée aux algorithmes à base de mots, nous amène à distinguer, dans le chapitre 5 les algorithmes qui n'effectuent qu'une unique lecture du texte de ceux qui procèdent à une prélecture. Ces derniers font appel à la construction de lexiques des mots du texte traité. Nous étudions les représentations : listes, arbres, automates de ces lexiques et les méthodes de compression. Nous présentons une méthode compacte de représentation de lexique, fondée sur la structure d'arbre, qui permet d'atteindre des taux de compression de lexique très élevés.

Dans le chapitre 6, nous présentons l'algorithme de Cormack et Horspool [16] qui est une adaptation de l'algorithme de Ziv et Lempel. Nous proposons de lui apporter quelques améliorations concernant la compression des mots et codage des indices des facteurs.

Le chapitre 7 s'intéresse au modèle Word qui est une adaptation du codage arithmétique à la compression à base de mots qui est due à Moffat [41] en 1989. Ce modèle avait été développé, en 1989, alors que les textes à compresser étaient de taille relativement petite ou au vocabulaire réduit. Le modèle Word a donné lieu à une nouvelle implémentaion présentée par Moffat, Neal et Witten [43] en 1995. Elle est fondée notamment sur les travaux de Fenwick [20] qui mit au point une représentation de la liste des fréquences des symboles qui permet leur mise à jour en un temps $O(\log(n))$. Nous avons ajouté à cet algorithme une compression de chaque mot nouveau par Ziv et Lempel ainsi qu'un modèle de découpage du texte qui tente de prédire l'occurrence de mots comportant des lettres majuscules.

Le codage de Huffman à base de mots est étudié dans le chapitre 8. Il fait appel à une forme particulière de codage de Huffman appelé Huffman canonique. Initialement dû à Schwartz et Kallick [56] en 1964 et étudié par Connel [14] en 1973, il a été repris dans le cadre du *Text Retrieval* en 1994 par Moffat Witten et Bell [68] dont une implémentation est connue sous le nom de Huffword. L'algorithme Huff_M$^+$ que nous présentons reprend l'algorithme Huffword en utilisant le traitement des mots comportant des lettres majuscules et une sauvegarde du lexique du texte sous une forme compacte exposée au chapitre 5. D'autre part, en combinant Huff_M avec une version modifiée de Ziv et Lempel nous construisons un algorithme Huff_Co$^+$ qui permet de traiter les cooccurrences de mots linguistiques.

Dans le dernier chapitre, nous présentons brièvement le standard Unicode dont le but est de pouvoir représenter des textes dans de nombreuses langues y compris celles (souvent asiatiques) qui utilisent un nombre élevé de symboles et dont la compression pourrait se rapprocher de la compression à base de mots. Nous proposons certaines améliorations de découpage du texte liées au traitement de la ponctuation ou de certains mots composés. Enfin nous présentons une variante de l'algorithme de Huff_M$^+$ exposé au chapitre précédent qui utilise plusieurs arbres de Huffman.

Tout au long de ce travail, la plupart des algorithmes de compression présentés font l'objet de tests expérimentaux sur six textes en langue naturelle appartenant au Canterbury Corpus [1]

Nom du Fichier	Titre et ou Description	Taille en octets
`asyoulik.txt`	*As You Like It* (pièce de théatre)	125.179
`alice29.txt`	*Alice in Wonderland*	152.089
`lcet10.txt`	*Technical Document*	426.754
`plrabn12.txt`	*Paradise Lost* (poésie anglaise)	481.861
`world192.txt`	*The Cia World Fact Book*	2.473.400
`bible.txt`	*The King James version of the Bible*	4.047.392

Les meilleurs de ces algorithmes ont été également testés sur le Calgary Corpus. De plus, afin de ne pas nous limiter à l'anglais, ils ont aussi été testés sur un corpus de textes français, espagnol, italien et grec.

[1] Adresse web : ftp ://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus

Notations.

On appelle **mot** une suite finie de caractères.

Si A est un ensemble de caractères :

$|A|$ est le nombre d'éléments de A.
A^* est l'ensemble des mots définis sur A.
A^n est l'ensemble des mots de longueur n sur A.
$A^{\leq n}$ est l'ensemble des mots de longueur inférieure ou égale à n sur A.

Si u et v sont des mots :

$|u|$ désigne la longueur de u.
$u \leq v$ veut dire que u est un préfixe de v.
$u < v$ veut dire que u est un préfixe propre de v.
ε représente le mot vide.

Si x est un nombre réel :

$\log_2(x)$ est logarithme en base deux de x.
$\lceil x \rceil$ est l'entier y vérifiant $y - 1 < x \leq y$.

Première Partie

Compression à base de caractères

Chapitre 1

Codes et compression

Dans ce chapitre, après avoir rappelé ce qu'est la représentation usuelle d'un texte électronique, nous décrivons les principes de base sur lesquels sont fondés les algorithmes de compression de données, les notions de code, de décodabilité, ainsi que celles d'information et d'entropie dues à Claude Shannon. Enfin, nous présentons l'un des premiers algorithmes de compression de données, celui de Shannon-Fano.

1.1 Qu'est-ce qu'un texte ?

Pour pouvoir parler de compression de texte, nous devons tout d'abord rappeler comment un texte est codé en machine. De même que les images et les sons, le texte se présente sous la forme de données numériques. Un texte est une suite de caractères qui, en général, sont transcrits à l'aide de mots de *longueur fixe*, sur l'alphabet {0,1}.

Le code des caractères le plus couramment utilisé est le code ASCII[1], qui code un symbole par un octet. Le code ASCII standard utilise les valeurs de 0 à 127 pour représenter, principalement, les lettres non accentuées (majuscules et minuscules), les chiffres et les ponctuations. Les valeurs de 128 à 255 qui n'ont pas d'attribution dans le code ASCII standard sont utilisées, selon les machines et les besoins, pour représenter des caractères accentués (é, û, ï, à, par exemple), des symboles mathématiques, des caractères étrangers (ñ, å, ş, par exemple), voire des caractères semigraphiques. Ainsi, un texte comportant N symboles est représenté au moyen de N octets, soit $N * 8$ bits.

[1] *American Standard Code for Information and Interchange* (voir annexe A.2)

Comment compresser un texte ?

Il faut considérer les symboles un par un et :

- Réduire le nombre moyen de bits par symbole.
- ou considérer certaines suites de symboles, appelés *facteurs*, et les représenter par un nombre moyen de bits inférieur à huit fois la longueur du facteur considéré.

Remarquons que la première approche peut être considérée comme un cas particulier de la seconde avec des facteurs de longueur 1.

Le fonctionnement d'un algorithme de compression de texte peut se représenter par le schéma suivant :

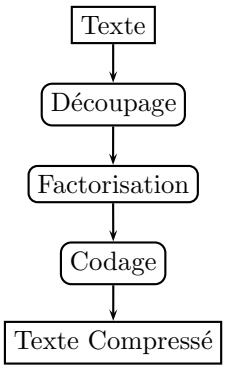


FIG. 1.1 – Modules d'un compresseur

Le découpage consiste à définir quelle est l'unité de traitement : par exemple, le caractère, la syllabe, le mot d'une langue naturelle. Nous utiliserons le terme *symbole* pour désigner l'unité de traitement.

La factorisation a pour but, en regroupant en facteurs des suites fréquentes de symboles, de favoriser une meilleure compression du texte.

Le codage consiste à associer, à chaque facteur, une représentation binaire la plus courte possible.

Les fonctions de découpage et de factorisation sont pour certains algorithmes de compression, triviales ou absentes. Le codage de Shannon-Fano, exposé à la fin de ce chapitre, a pour unité de traitement le caractère et n'utilise aucune factorisation.

On mesure expérimentalement l'efficacité d'un algorithme de compression grâce au *taux de compression* qui est exprimé :

– Soit en pourcentage :

$$\text{taux de compression} = \frac{\text{taille du texte compressé}}{\text{taille du texte}_{ascii}\text{ initiale}} * 100$$

– Soit en bits par symbole :

$$\text{taux de compression} = \frac{\text{taille du texte compressé * 8}}{\text{taille du texte}_{ascii}}$$

Un algorithme de compression qui permet la représentation d'un texte de 1 Mo sur 256 Ko, a un taux de compression de 25% ou de 2 bits par symbole.

1.2 Codes de longueur fixe.

Nous avons vu qu'un texte qui comporte N symboles est codé en ASCII au moyen de $8 * N$ bits, soit N octets. Les codes ASCII supérieurs à 127 étant pour certains textes (en anglais ou en français sans accent par exemple), non utilisés, un moyen simple de compresser ce type de texte consiste à ne coder chaque symbole que sur 7 bits. La taille d'un texte ainsi compressé est donc réduite de $12,5\%$.

Plus généralement, si l'on sait que le texte à compresser s'écrit sur un alphabet connu de q symboles, ces symboles peuvent être codés sur $\lceil \log_2(q) \rceil$ bits. C'est par exemple un moyen simple de compresser un texte décrivant un morceau d'ADN. Le code génétique ne comportant que 4 lettres, `A`, `C`, `G`, `T`, (initiales des bases Adénine, Citosine, Guanine, Thymine), on peut coder chacune sur 2 bits au lieu de 8 et obtenir ainsi facilement, une division par 4 de la taille originale du texte.

Ainsi, si nous codons `A` :00, `C` :01, `G` :10, `T` :11, la chaîne `ACCGTAT` est codée : 00010110110011.

Dans le même domaine, une suite d'acides aminés (au nombre de 20) pourrait être codée au moyen de 5 bits par symbole ($5 = \lceil \log_2(20) \rceil$).

Dans les deux exemples précédents, les alphabets sont prédéterminés. En revanche si l'alphabet utilisé pour écrire un texte est propre à ce texte, on ne peut relire le texte que s'il est accompagné de la liste des q caractères (un sous-ensemble du code ASCII) formant cet alphabet. L'alphabet du texte n'étant ici qu'une simple sélection de caractères dans l'ensemble des caractères ASCII, la présence d'un caractère dans l'alphabet du texte peut se coder sur un bit. La liste des caractères utilisés dans un texte est donc codable au moyen de 256 bits. Le $i^{ème}$ bit est mis à 1 si le $i^{ème}$ caractère ASCII est présent dans l'alphabet du texte. Ainsi peut-on envoyer cette liste grâce à $256/8 = 32$ octets. La taille d'un texte de N caractères ainsi traité est alors $(N * \lceil \log_2(q) \rceil + 32 * 8)$ bits.

Une autre forme de compression utilisant des codes de longueur fixe consiste à utiliser les codes ASCII non utilisés dans un texte pour remplacer certaines suites de deux caractères, appelées *digrammes*, fréquentes dans le texte. Ainsi le coût de ces digrammes est de un octet au lieu de deux. Cette technique de compression a été notamment utilisée dans un algorithme proposé par Udi Manber [39] qui permet d'appliquer les algorithmes classiques de recherche de motif au texte compressé.

1.3 Codes à longueur variable et décodabilité

Tous les exemples précédents permettent de compresser un texte en réduisant le nombre de bits nécessaires à coder l'unité de base d'un texte : le caractère. Tous les caractères de l'alphabet sont codés avec un même nombre de bits. On constate aisément à la lecture d'un texte que la distribution de ces caractères n'est pas uniforme ; ainsi dans un texte écrit en français, la proportion de `e` s'avère plus importante que celle de `n` ou de `k`. Cette simple observation est à la base des codes à *longueur variable* dans lesquels les divers caractères sont codés sur un nombre de bits d'autant plus petit que leur fréquence est grande.

Un code utilisant ce principe de *longueur variable* en fonction de la fréquence du caractère a été mis au point par Samuel F.B. Morse (tableau 1.1). Le code Morse, du nom de son inventeur, code à la base de la télégraphie sans fil, et dont l'idée remonte à 1832, fut présenté pour la première fois en 1837. Il permet de représenter 42 caractères ou signaux à l'aide de trois signes, le point, le trait et l'espace, ce dernier jouant le rôle de séparateur. Le trait dure trois fois plus longtemps que le point. L'espace représente une unité de temps et sera noté `t`. Le point et le trait sont séparés par un espace, les caractères par 3 espaces, les mots par 6 espaces.

Écrivons selon ce code le mot `DE` :

$$\texttt{DE} = -\texttt{t}\cdot\texttt{t}\cdot\texttt{ttt}\cdot$$

Dans le cadre informatique, nous devons utiliser un codage binaire. Supposons que l'on écrive le mot `DE` sans séparateur.

Nous avons : $\texttt{DE} = -\cdot\cdot$

Nous pouvons bien sûr relire aisément le mot `DE`. Cependant le texte codé peut être lu de plusieurs manières : `TEEE`, `TEI`, `TIE`, `TS`, `NEE`, `NI`, `B`.

Cet exemple fait apparaitre la notion de *décodabilité*. En effet, pour relire le mot `DE` et **ne relire que lui**, (unicité du décodage), il faut que l'on sache découper correctement le message en mots du code, puis en symboles.

Car.	Code	Car.	Code
A	·–	1	· – – – –
B	– · ··	2	·· – – –
C	– · –·	3	··· – –
D	– ··	4	···· –
E	·	5	·····
F	·· –·	6	– ····
G	– – ·	7	– – ···
H	····	8	– – – ··
I	···	9	– – – – ·
J	···	0	– – – – –
K	– · –	.	· – · – ·–
L	· – ·	,	– – ·· ––
M	···	:	– – – ···
N	–·	?	·· – – ··
O	– – –	-	– ···· – – · · – ·
P	· – –·	/	· – · · –·
Q	– – ·–	(ou)	· – – · –
R	· – ·	=	– ··· –
S	···	compris	··· – ·
T	–	erreur	····· ··
U	·· –	+	· – · – ·
V	···–	inv	– · –
W	· – –	attente	· – ···
X	– · –	fin de travail	··· – ·–
Y	– · ––	commencement	– · – · –
Z	– – ··	séparateur	· – · · –

TAB. 1.1 – Code Morse

Ce problème de décodabilité provient du fait que contrairement aux codes dont nous avons parlé précédemment, le code Morse est à longueur variable. En effet, lorsqu'un code est de longueur fixe k *bits* par symbole, nous savons que tous les k *bits*, on obtient le code d'un symbole. Dans le cas d'un code à longueur variable, tel le code Morse, il faut savoir à quel instant un symbole a été lu. On dira désormais qu'un ensemble de mots sur un alphabet A est un code si l'unicité du décodage est garantie. Celle du code Morse est obtenue grâce aux espaces séparateurs.

Définition : Soit A un alphabet.

Un code est un ensemble de mots $X \subseteq A^+$ qui vérifie :

$$\forall x_1, x_2, ..., x_n, y_1, y_2, ..., y_m \in X$$
$$x_1x_2...x_n = y_1y_2...y_m \Rightarrow n = m \text{ et } \forall i \; x_i = y_i.$$

Exemple : Considérons les ensembles de 4 mots E_1, E_2, E_3, E_4, E_5 .

Mot	E_1	E_2	E_3	E_4	E_5
m_1	00	0	0	0	0
m_2	01	1	10	01	10
m_3	10	00	110	011	110
m_4	11	11	1110	0111	111

TAB. 1.2 – Ensembles de mots

E_1 est un ensemble de mots de longueur fixe, c'est donc un code.

En ce qui concerne E_2, si nous considérons la suite 100, elle peut se lire de 2 façons : m_2m_3 ou $m_2m_1m_1$. E_2 n'est donc pas un code.

On voit facilement que E_3 est un code. Il suffit de remarquer qu'un nouveau mot a été reconnu dès l'arrivée d'un 0. Le zéro, qui appartient au mot du code reconnu, joue ici le même rôle de séparateur que les *3 unités de temps* du code Morse. Le code E_3, est un code appelé *code instantané*.

De plus, E_4 est, en quelque sorte, le miroir du code E_3. L'occurrence d'un 0 indique, comme dans le cas précédent, qu'un nouveau mot a été lu. Cependant, contrairement au code E_3 le 0 n'appartient pas au mot du code reconnu, mais au début du mot du code suivant. En effet, supposons que nous ayons lu 01, il n'est pas possible à ce stade, de reconnaitre m_2, il faut lire un bit supplémentaire. Si celui-ci est 0, le mot précédent était bien m_2. Si le bit lu est un 1, il faut lire encore un bit pour savoir si le symbole codé est m_3 ou m_4. Ce type de code est dit *non instantané*.

En ce qui concerne E_5, on remarque qu'un mot est reconnu dès l'occurrence d'un 0 ou de trois 1 successifs. E_5 est donc un code intantané.

Définition :

$X \subseteq A^+$ est un code préfixe si et seulement si, aucun mot de X n'est préfixe d'un autre mot de X :

$$\forall x,\ x' \in X\ x \leq x' \Rightarrow x = x'.$$

Une condition nécessaire et suffisante pour qu'un code soit instantané est qu'il soit préfixe.

- Cette condition est suffisante car la lecture d'un mot complet du code entraîne le découpage à la fin de ce mot.

- Cette condition est nécessaire : Si $x < y$, on ne sait pas après avoir lu x si le découpage doit se faire à cette position ou s'il s'agit du début de y.

Si nous revenons à l'étude des ensembles de mots E_3, E_4, E_5, le code E_3 est effectivement un code préfixe. En revanche, pour le code E_4, 0111 a 3 préfixes : 0,01,011, qui appartiennent à ce code.

Enfin, il est clair que le code E_5 est préfixe, et donc instantané.

1.4 Existence d'un code.

Jusqu'à maintenant, nous avons mis en évidence l'existence de codes binaires instantanés.

Nous allons maintenant donner une condition d'existence de tels codes.
Cette condition due à Kraft ne s'applique pas seulement aux codes binaires mais, plus généralement, aux codes n-aires.

Soit E un ensemble de q mots $m_1, m_2, \cdots\cdots, m_q$ sur un alphabet A et $l_1, l_2, \cdots\cdots, l_q$ un ensemble d'entiers tel que $|m_i| = l_i$ $(1 \leq i \leq q)$.

Une condition nécessaire et suffisante à l'existence d'un code instantané est donnée par l'inégalité de Kraft :

$$\sum_{i=1}^{q} n^{-l_i} \leq 1.$$

Dans le cas binaire, qui nous intéresse plus particulièrement, elle s'écrit :

$$K = \sum_{i=1}^{q} 2^{-l_i} \leq 1.$$

Considérons à nouveau les ensembles de mots du tableau 1.2 et donnons pour chacun d'eux les longueurs respectives des mots qui les constituent. Nous obtenons le tableau suivant :

$\mathrm{l}(E_1)$	$\mathrm{l}(E_2)$	$\mathrm{l}(E_3)$	$\mathrm{l}(E_4)$	$\mathrm{l}(E_5)$
2	1	1	1	1
2	1	2	2	2
2	2	3	3	3
2	2	4	4	3

TAB. 1.3 – Longueurs des mots

Calculons pour chacun d'eux la somme K et appellons K_j la somme associée à l'ensemble E_j $(1 \leq j \leq 5)$.

Nous obtenons :

$$\begin{aligned} K_1 &= 2^{-2} + 2^{-2} + 2^{-2} + 2^{-2} = 1 \\ K_2 &= 2^{-1} + 2^{-1} + 2^{-2} + 2^{-2} = \tfrac{3}{2} \\ K_3 &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = \tfrac{15}{16} \\ K_4 &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = \tfrac{15}{16} \\ K_5 &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 1 \end{aligned}$$

D'après K_2 il n'existe aucun code ayant la suite de longueurs $1, 1, 2, 2$ $(\frac{3}{2} > 1)$. L'ensemble E_2, n'est pas un code.

Pour les autres ensembles, l'inégalité de Kraft est vérifiée. Insistons sur le fait que la vérification de cette inégalité si elle nous garantit l'**existence** d'un code instantané, n'énonce nullement que tout code la vérifiant est instantané. Bien que partageant tous deux la même suite de longueurs, le code E_3 est instantané, mais non le code E_4.

Nous avons jusqu'ici défini les notions de codes à longueur fixe, de codes à longueur variable et de décodabilité ainsi que celle de compression de texte comme étant le passage d'un codage ASCII à un codage, le plus souvent à longueur variable et pouvant donc ainsi tenir compte de distributions non uniformes de caractères dans un texte. Ces distributions étant différentes selon le texte considéré, on ne peut se contenter, pour le compresser de façon efficace, d'un codage standard (fondé sur une distribution moyenne de symboles) dans une langue donnée comme le fait le code Morse. La construction du code prendra en compte les fréquences effectives des caractères dans le texte à compresser.

1.5 Information et Entropie.

1.5.1 La Notion d'information.

Considérons l'événement « avoir une température de i degrés », à un instant et en un lieu donnés du globe. Intéressons nous à la température de Tunis à midi, en été. L'occurrence d'une journée avec une température de 5^oC est très peu probable en cette période de l'année, en revanche sa survenue est d'autant plus surprenante et recèle donc beaucoup plus d'information que si le thermomètre affichait 30^oC.

Ainsi la notion d'information due à Claude Shannon [58] qui est en relation avec la notion de surprise varie donc inversement à la probabilité de l'événement.

Considérons un ensemble de q symboles $s_1, s_2, \cdots\cdots, s_q$ de probabilités respectives, $p_1, p_2, \cdots\cdots, p_q$ et appelons $I(s_i)$ l'information associée au symbole s_i. Shannon définit I par :

$$I(s_i) = \log_2(\frac{1}{p_i}).$$

Ainsi l'information contenue dans deux symboles indépendants s_i et s_j est la somme de leurs informations respectives :

$$I(s_i, s_j) = I(s_i) + I(s_j).$$

Car puisque nous savons que la probabilité conjointe de deux événements indépendants est égale au produit de leurs probabilités :

$$p_{i,j} = p_i * p_j.$$

Nous avons bien :

$$I(s_i) + I(s_j) = \log_2(\frac{1}{p_i}) + \log_2(\frac{1}{p_j}) = \log_2(\frac{1}{p_i p_j}) = \log_2(\frac{1}{p_{i,j}}) = I(s_i, s_j).$$

1.5.2 Définition de l'entropie.

Si $I(s_i)$ est la quantité d'information obtenue à la réception du symbole s_i et p_i la probabilité d'obtenir s_i, la quantité moyenne d'information reçue pour chaque symbole s_i est :

$$p_i I(s_i).$$

En considérant l'ensemble des symboles, on définit :

$$H(S) = \sum_{i=1}^{q} p_i I(s_i).$$

Cette quantité est appelée l'*entropie* de la source S émettant les symboles s_i :

$$H(S) = \sum_{i=1}^{q} p_i \log_2(\frac{1}{p_i}).$$

Si l'on considère un code instantané pour l'ensemble des symboles s_i, dont les mots du code sont de longueurs respectives l_i, on démontre, grâce à l'inégalité de Kraft [2] que la fonction H vérifie :

$$H(S) \leq L$$

où L désigne la longueur moyenne des mots du code :

$$L = \sum_{i=1}^{q} p_i l_i.$$

L'entropie H est donc une borne inférieure pour la longueur moyenne L des mots du code.

1.6 Code de Shannon.

Considérons une source S émettant des symboles de l'alphabet $s_1, s_2, \cdots\cdots, s_q$ de probabilités respectives, $p_1, p_2, \cdots\cdots, p_q$. Pour chaque p_i il existe un entier l_i tel que :

$$\log_r(\frac{1}{p_i}) \leq l_i < \log_r(\frac{1}{p_i}) + 1 \qquad (1)$$

où r est la taille de l'alphabet du code.

Considérons le code qui associe à chaque symbole s_i un des mots n_i écrit en base r, de longueur l_i, qui vérifie (1). Montrons que ce code est instantané.

La relation (1) donne :

$$\frac{1}{p_i} \leq r^{l_i} < \frac{r}{p_i}.$$

Soit encore :

$$p_i \geq \frac{1}{r^{l_i}} > \frac{p_i}{r}.$$

Or $\sum p_i = 1$ d'où en sommant pour tous les symboles :

$$1 \geq \sum_{i=1}^{q} (\frac{1}{r^{l_i}}) > \frac{1}{r}.$$

On obtient l'inégalité de Kraft ce qui établit qu'il existe un code instantané ayant les longueurs définies par (1). Un tel code est appelé code de Shannon.

Pour calculer l'entropie du code de Shannon, on multiplie (1) par p_i et on somme :

$$H_r(S) = \sum_{i=1}^{q} p_i \log_r(\frac{1}{p_i}) \leq \sum_{i=1}^{q} p_i l_i < H_r(S) + \sum_{i=1}^{q} p_i.$$

Soit :

$$\boxed{H_r(S) \leq L < H_r(S) + 1}$$

Et pour r = 2 :

$$\boxed{H_2(S) \leq L < H_2(S) + 1}$$

Si nous considérons la distribution de probabilité suivante :

Symbole	*Probabilité*
s_1	$0,25$
s_2	$0,25$
s_3	$0,125$
s_4	$0,125$
s_5	$0,125$
s_6	$0,125$

TAB. 1.4 – Liste des symboles

Nous obtenons la suite de longueurs des mots représentant les symboles. Puis on attribue aux symboles des représentations de manière consécutive. Il y a 2 symboles de longueurs 2 le premier symbole est 00, le second 00 + 1 soit 01. Il y a ensuite 4 symboles de longueur 3. Le premier mot de longueur 3 ne peut avoir ni 00 ni 01 comme préfixe. On choisit donc 10 et s_3 est représenté par 100. On obtient les représentations des 3 autres symboles par incrémentations successives.

On obtient alors :

Symbole	*Longueur*	*Code*
s_1	2	00
s_2	2	01
s_3	3	100
s_4	3	101
s_5	3	110
s_6	3	111

TAB. 1.5 – Le code de Shannon

Le code issu du codage de Shannon peut être calculé grâce à l'algorithme de Robert Fano [21]. Cet algorithme est le suivant :

- Établir la liste triée des symboles selon l'ordre décroissant de leur probabilité.
- Diviser cette liste en deux parties dont les sommes de probabilités sont sensiblement égales.
- Attribuer aux codes des symboles de la première partie un 0 et de la seconde un 1.
- Continuer ainsi récursivement jusqu'à ce que toute partie soit réduite à un singleton.

Si nous appliquons cet algorithme aux symboles du tableau 1.4 nous obtenons le déroulement de l'algorithme décrit ci-dessous et retrouvons les mots du code du tableau 1.5.

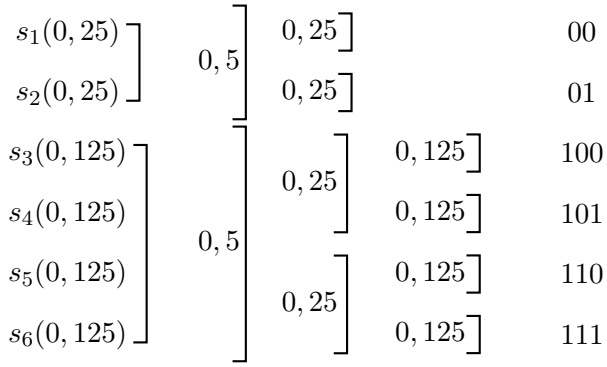


FIG. 1.2 – Algorithme de Fano

Du fait de cette double paternité, le code de Shannon est fréquement appelé code de Shannon-Fano.

Chapitre 2

Code de Huffman

Dans ce chapitre, nous présentons l'algorithme de Huffman et notamment une version de cet algorithme permettant d'obtenir un arbre de Huffman de hauteur minimale. De plus, nous présentons deux algorithmes qui faisant appel à la notion d'automate, améliorent en temps l'implémentation du codage de Huffman.

Le codage de Huffman [29] est la méthode la plus connue des méthodes de compression de données et a été couramment utilisée pendant plus d'une vingtaine d'années. Ce codage a été développé par David Huffman en 1952 alors qu'il était étudiant au MIT et y suivait un cours dirigé par Robert Fano. La mise au point de ce code lui valut d'ailleurs d'être dispensé d'examen terminal. Comme le code de Shannon-Fano, c'est un codage de type statistique qui utilise les fréquences des symboles qui composent le texte pour construire des codes préfixes. Il a donné lieu à de nombreux travaux dont ceux de Robert Gallager [23] qui en a donné une version dynamique ou adaptative ne nécessitant plus qu'une seule lecture du texte traité. Il n'est plus utilisé aujourd'hui directement comme codeur d'un texte, mais il est utilisé en combinaison avec d'autres algorithmes, fondés sur le *codage par facteurs* comme *gzip*[1] ou le *codage à base de mots*[2].

[1] Voir chapitre 4
[2] Voir chapitre 8

2.1 Algorithme de Huffman

L'algorithme de Huffman construit un arbre binaire complet dont les nœuds feuilles portent les symboles que l'on veut représenter. Par convention, l'arête joignant un nœud à son fils droit est étiquetée 1, celle joignant ce nœud à son fils gauche est étiquetée 0. La représentation, ou traduction d'un symbole, est le mot binaire lu en parcourant l'unique chemin menant de la racine à ce symbole. Pour construire cet arbre, appelé arbre de Huffman, on considère la liste des nœuds feuilles et l'on applique l'algorithme suivant :

- On enlève les deux nœuds de plus faible poids de la liste.
- On crée un nouveau nœud ayant pour fils les deux nœuds précédents et pour poids, la somme de leurs poids.
- Ce nouveau nœud est ajouté à la liste.
- Les trois étapes précédentes sont répétées jusqu'à ce qu'il n'y ait plus qu'un nœud dans la liste. Ce nœud est la racine de l'arbre de Huffman.

On démontre que le code préfixe ainsi construit est optimal. La démonstration fait appel aux démonstrations [17] des deux lemmes suivant :

Lemme 1 : Soit A un ensemble dans lequel chaque symbole $s \in A$ a pour fréquence $f(s)$. Soient x et y deux symboles de A ayant les plus petites fréquences. Alors, il existe un code préfixe optimal C pour A dans lequel les représentations de x et y ont la même longueur et ne diffèrent que par leur dernier bit (x et y sont portés par des nœuds frères).

Lemme 2 : Soit T un arbre binaire complet représentant un code préfixe optimal C pour un alphabet A dans lequel la fréquence $f(s)$ est définie pour tout symbole $s \in A$. Pour chaque couple (x, y) de symboles portés par des nœuds frères de T, on introduit un nouveau symbole z, ayant pour fréquence $f(z) = f(x) + f(y)$, porté par le nœud père de x et y. Alors, l'arbre $T' = T - \{x, y\}$ représente un code préfixe optimal C' pour l'alphabet $A' = A - \{x, y\} \sqcup \{z\}$.

Appliquons l'algorithme de Huffman à la distribution de symboles suivante :

Symbole	s_1	s_2	s_3	s_4	s_5	s_6
Fréquence	5	4	2	2	1	1

Nous obtenons le processus suivant, où O symbolise un nœud interne :

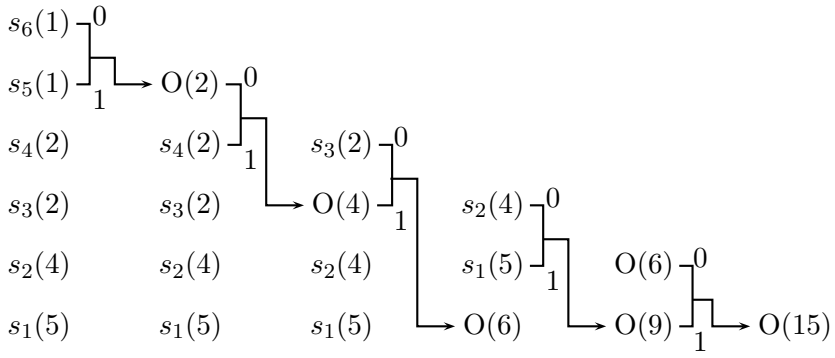
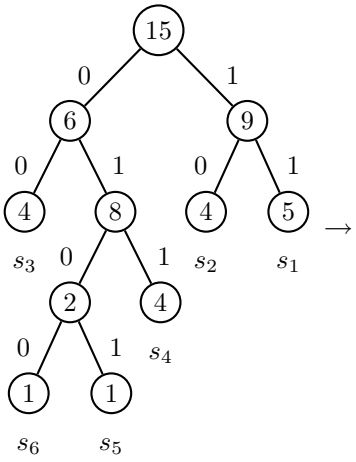


FIG. 2.1 – Construction d'un code.

Cette construction conduit à l'arbre de Huffman et au code suivants :



→

Symbole	*Code*	*Longueur*
s_1	11	2
s_2	10	2
s_3	00	2
s_4	011	3
s_5	0101	4
s_6	0100	4

FIG. 2.2 – Arbre et code de Huffman

Une distribution de symboles donnée peut conduire à plusieurs arbres de Huffman. Ainsi nous pouvons faire aussi la construction de la figure 2.3.

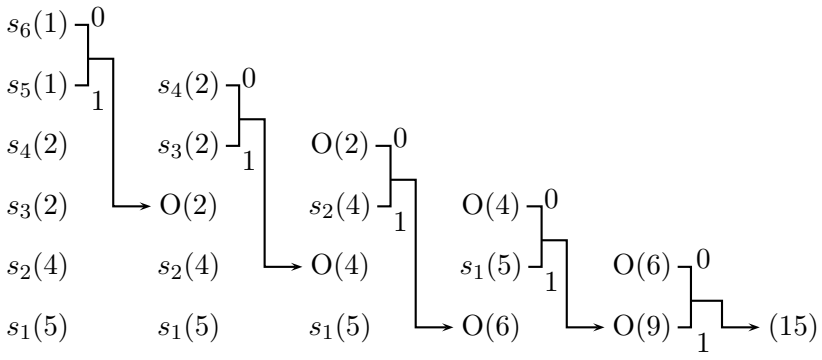
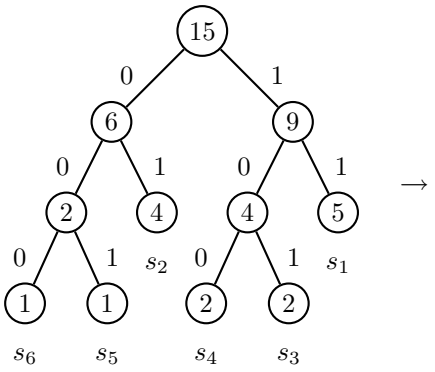


FIG. 2.3 – Construction d'un autre code.

Cette construction conduit à l'arbre de Huffman et au code de la figure 2.4.



→

Symbole	*Code*	*Longueur*
s_1	11	2
s_2	01	2
s_3	101	3
s_4	100	3
s_5	001	3
s_6	000	3

FIG. 2.4 – Arbre de Huffman de hauteur minimale et code.

Pour les deux arbres des figures 2.2 et 2.4 la longueur moyenne de la traduction d'un symbole est $\frac{18}{7}$.

On montre que la longueur moyenne d'un symbole est invariable même si les codes calculés peuvent être différents. On remarque cependant que la longueur L_{max} du plus long mot du code est égale à 3 dans l'arbre de la figure 2.4 alors qu'elle est égale à 4 dans l'arbre de la figure 2.2

En fait, dans la construction du deuxième code, tout nœud nouvellement créé est utilisé après tous les éventuels autres nœuds de même poids. Cette technique appelée *bottom-merging* et due à Eugene S. Schwartz [55], permet de rendre L_{max} minimale.

Pour des raisons de clarté, on a représenté la construction d'un code de Huffman (figures 2.1 et 2.3) de telle façon qu'il faut systématiquement parcourir la liste des nœuds, triée selon leurs poids, pour insérer chaque nœud créé en sorte que la nouvelle liste reste triée. En fait, ce parcours peut être évité grâce à une implémentation utilisant deux files d'attente, l'une contenant les nœuds feuilles, dans l'ordre croissant de leurs fréquences et l'autre les nœuds internes créés au cours de la construction de l'arbre de Huffman. Les deux nœuds de plus petits poids sont toujours situés en tête de l'une ou l'autre des files.

Si l'on appelle F la file contenant les nœuds feuilles et I celle contenant les nœuds internes, l'algorithme de Huffman s'écrit alors :

1. F=file des nœuds feuilles
2. I=file des nœuds internes
3. **Tant que** $|F| + |I| > 1$ **Faire**
4. $u = min(F, I)$
5. $v = min(F, I)$
6. $x = père(u, v)$
7. $poids(x) = poids(u) + poids(v)$
8. mettre x dans I
9. **Fin du Tant que**

La fonction *min* renvoie le nœud de plus petit poids entre F et I et le retire de F ou de I . En cas d'égalité de poids, elle choisit toujours le nœud de F pour respecter la règle de *bottom-merging*.

Examinons la complexité de cet algorithme. La phase de tri peut être faite, grâce à un algorithme efficace, en un temps $O(|A|.\log(|A|))$. La fonction *min* à l'intérieur de la boucle « Tant que » est exécutée en temps constant puisque le nœud de poids mininimal de $F \sqcup I$ est situé parmi les deux premiers nœuds de chaque liste et la boucle qui crée exactement $|A| - 1$ nœuds prend un temps $O(|A|)$ [18]. L'algorithme précédent construit l'arbre de Huffman en un temps $O(|A|.\log(|A|))$.

L'entropie de l'alphabet A est définie par :

$$H(A) = -\sum_{a \in A} p(a).\log(p(a)).$$

Dans le cas du code de Huffman, la longueur moyenne L du code d'un symbole vérifie :

$$H(A) \leq L \leq H(A) + 1.$$

En fait, une borne plus fine a été obtenue par Gallager en 1978 [23]. Si on appelle p_{max} le symbole de plus forte probabilité, L vérifie :

Si $p_{max} \geq 0,5$:

$$H(A) \leq L \leq H(A) + p_{max} + 0.086.$$

Si $p_{max} < 0,5$:

$$H(A) \leq L \leq H(A) + p_{max}.$$

Le fait de devoir connaître préalablement les poids des symboles impose une prélecture du texte traité. D'autre part, la décompression du texte ne peut se faire sans la connaissance de l'arbre de Huffman ou des fréquences qui ont permis sa construction. L'un ou les autres doivent être sauvegardés avec le texte compressé. Les algorithmes de compression et de décompression s'écrivent :

Compression

- Première lecture
  - Calcul des fréquences des symboles.
  - Calcul du code de Huffman.
  - Sauvegarde du code.
- Seconde lecture
  - Substitution symbole/mot du code

Décompression

- Chargement du code de Huffman.
- Substitution mot du code/symbole

La vue de ces algorithmes nous amène à étudier la manière dont on peut mémoriser les codes de Huffman, aspect souvent négligé dans la littérature sur la compression, mais qui se révèle important dans le cas de textes de petite taille ou dans le cas où la taille de l'alphabet est importante[3].

[3]Voir Huffman à base de mots (chapitre 8)

2.2 Sauvegarde du codage

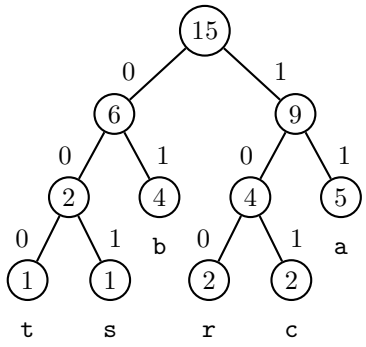
La mémorisation du codage peut se faire soit par sauvegarde des fréquences associées aux symboles, ce qui implique une reconstruction de l'arbre à la décompression, soit par sauvegarde de l'arbre de Huffman.

2.2.1 Sauvegarde de l'arbre de Huffman

La mémorisation de l'arbre de Huffman peut se faire grâce à un mot sur $\{0,1\}^*$ de longueur maximum $2|A|-1$. Il suffit de faire un parcours récursif de l'arbre de Huffman en indiquant un nœud interne par 1 et une feuille par 0, chaque feuille étant suivie de la représentation binaire du code ASCII du symbole qu'elle porte.

Exemple :

En reprenant l'arbre de la figure 2.4 et en considérant que les symboles s_i sont respectivement les caractères ASCII `a`, `b`, `c`, `r`, `s`, `t`, on obtient :



Sa représentation, qui utilise 59 bits est :

1110<u>01110100</u><u>01110011</u><u>00110001</u>0110<u>01100011</u><u>00111001</u><u>00110000</u>1

asc(`t`) asc(`s`) asc(`b`) asc(`r`) asc(`c`) asc(`a`)

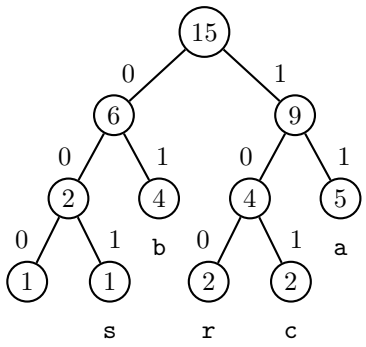
FIG. 2.5 – Représentation binaire d'un arbre de Huffman

2.2.2 Sauvegarde des fréquences

Au lieu de sauvegarder l'arbre, on peut conserver la liste des fréquences des symboles, ce qui oblige à reconstruire l'arbre à la décompression. L'alphabet de base étant l'ensemble des caractères ASCII, ceci obligerait à enregistrer 256 voire 512 octets (si les fréquences sont codées sur deux octets). Dans la pratique, de nombreuses fréquences sont nulles (caractères de contrôle, caractères ASCII supérieurs), on peut se contenter d'indiquer les intervalles de fréquences non nulles.

Une méthode proposée par Mark Nelson [47] consiste à représenter une suite de fréquences de symboles consécutifs, par le code du premier symbole, le code du dernier symbole et la suite des valeurs des fréquences des symboles compris entre ces deux symboles. La dernière suite de fréquences est suivie d'un zéro.

Ainsi, si nous considérons à nouveau l'arbre de la figure 2.5



Nous obtenons la représentation :

$$97, 99, 5, 4, 2, 114, 116, 2, 1, 1, 0$$

$$\texttt{a}\quad \texttt{c}\quad \textit{fréq}\quad \texttt{r}\quad \texttt{t}\quad \textit{fréq}$$

Cette technique de mémorisation implique évidemment de reconstruire l'arbre de Huffman lors de la décompression.

2.3 Codage de Huffman et automates

L'utilisation du codage de Huffman pose des problèmes de structures de données et d'espace mémoire, mais aussi de temps de traitement dus à la l'écriture et à la lecture bit à bit. Nous présentons ici d'une part l'algorithme de Hatsukazu Tanaka [61] et d'autre part celui de Sieminski [60]. Ils font appel tous deux à la structure d'automate.

Définition : On appelle automate $\mathcal{A}$ la donnée de

$$\mathcal{A} =< Q, \Sigma, \delta, q_0, F >$$

où :

Q est l'ensemble des états de $\mathcal{A}$
Σ est l'alphabet d'entrée.
F est l'ensemble des états terminaux,
q_0 est l'état initial.
δ est la fonction de transition de $Q \times \Sigma \rightarrow$ Q

Si $w = w_1 w_2 .... w_n$ est un mot de Σ^* on définit $\delta(q, w) = \delta(\delta(q, w_1), w_2 .... w_n)$. Un mot w est reconnu ou accepté par l'automate A, si et seulement si $\delta(q_0, w) = q_f$, avec $q_f \in F$. (ε désigne le mot vide)

2.3.1 L'algorithme de Tanaka

L'algorithme de Tanaka a été développé dans le cadre du codage de Huffman $n - aire$. Nous l'exposons ici dans le cas binaire qui nous intéresse plus particulièrement.

Considérons un ensemble de q symboles $S = \{s_1, s_2, \cdot \cdot \cdot \cdot, s_q\}$ de probabilités respectives $p_1, p_2, \cdot \cdot \cdot \cdot, p_q$ et un arbre de Huffman, construit sur ces probabilités dont on numérote les nœuds internes de manière croissante en partant de la racine et en descendant par niveau. On numérote également les nœuds feuilles par des nombres négatifs de telle sorte que le symbole s_k soit porté par le nœud numéroté $-k$.

Un arbre peut être vu comme un automate dont les nœuds sont les états, les arêtes les transitions, les feuilles les états finaux et la racine l'état initial. On prolonge cet automate en ajoutant pour tout état final q_f une ε-transition $\delta(q_f, \varepsilon) = q_0$. Cette représentation permet de définir un automate qui traite bit à bit le code d'un symbole.

2.3.1.1 Automate de décodage

L'arbre de Huffman associé à un ensemble de q symboles est un arbre binaire complet ayant $q-1$ nœuds internes. L'automate de décodage est représentable par un tableau de taille $2 \times (q-1)$. On considère donc la matrice M définie $M(i,j) = k$ si la branche de l'arbre reliant le nœud i au nœud k est étiquetée j, $(j \in \Sigma)$.

Si nous considérons $\mathcal{T}_1^{Tanaka}$ l'arbre de Huffman de la figure 2.4 numéroté selon la numérotation de Tanaka, nous avons :

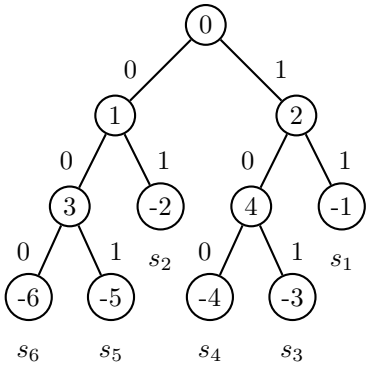


FIG. 2.6 – Arbre numéroté

$\Sigma = \{0,1\}^*$ alphabet d'entrée.
$F = \{-1,-2,-3,-4,-5,-6\}$.
$Q = \{0,1,2,3,4,\} \sqcup \{-1,-2,-3,-4,-5,-6\}$
$q_0 = \{0\}$
$\delta(i,j) = M(i,j) = k$

avec M définie par :

État	0	1
0	1	2
1	3	-2
2	-1	4
3	-6	-5
4	-3	-4

Si nous décodons la séquence binaire 00110, nous obtenons la suite d'états :

$$0\ 1\ 3\ -5\ 0\ 2\ -1$$

Les symboles s_5 et s_1 ont été reconnus.

2.3.1.2 Automate de codage

Le codage est une opération qui nécessite pour chaque mot du code deux nombres, un entier et le nombre de bits de cet entier qu'il faut transmettre. Pour rendre le processus de codage similaire à celui du décodage, Hatsukazu Tanaka propose un algorithme qui consiste à mémoriser dans une pile, les bits portés par les arêtes de l'arbre de Huffman, en le remontant depuis le symbole à coder jusqu'à la racine. Les bits ayant été accumulés dans l'ordre inverse, il suffit pour les transmettre de les dépiler successivement. L'algorithme de Tanaka fait appel à la notion d'arbre parfait.

Définition :

Un arbre binaire parfait est un arbre binaire dont toutes les feuilles sont situées sur deux derniers niveaux seulement ; l'avant-dernier niveau est complet et les feuilles du dernier niveau sont regroupées le plus à gauche.

Les arbres parfaits ont une représentation séquentielle compacte qui est fondée sur la numérotation en ordre hiérarchique total des nœuds dans un parcours par niveaux. On numérote, en ordre croissant à partir de 0, tout nœud de l'arbre à partir de la racine, niveau par niveau, et de gauche à droite sur chaque niveau. L'algorithme de Tanaka considère des arbres binaires de Huffman parfaits.

Considérons $\mathcal{T}_2^{Hp6}$ l'arbre de Huffman parfait numéroté à 6 symboles de la figure 2.7.

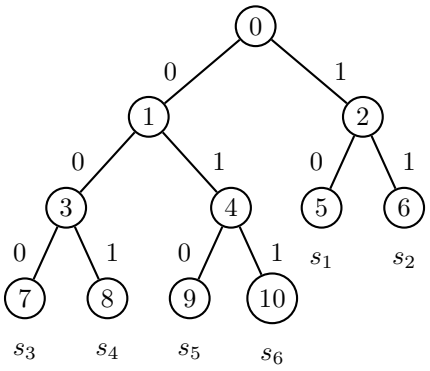


FIG. 2.7 – Arbre de Huffman parfait numéroté

Si un sommet de l'arbre est numéroté i dans la numérotation en ordre hiérarchique total, son fils gauche est numéroté $2i + 1$ et son fils droit $2i + 2$.

Si nous considérons la matrice P représentant la fonction de transition de l'automate associé à l'arbre de Huffman parfait de la figure 2.7, nous avons :

La fonction de transition peut être calculée par :
$P(i, j) = 2 \times i + j + 1$

nœud	0	1
0	1	2
1	3	4
2	5	6
3	7	8
4	9	10

Le codage d'un symbole consiste à remonter le chemin de l'arbre du nœud feuille portant ce symbole jusqu'à la racine en mémorisant les bits portés par les arêtes empruntées.

La numérotation en ordre hiérarchique total nous permet d'écrire pour tout nœud n :

$$père(n) = (n-1)/2 \qquad bit(n) = (n-1) \bmod 2$$

L'algorithme permettant de coder le symbole s_n s'établit ainsi :

n est le numéro de la feuille associée à s_n

1. **Tant que** $n <> 0$ **Faire**
2. Empile($bit(n)$)
3. $n = père(n)$
4. **Fin du tant que**
5. **Tant que** pile non vide **Faire**
6. Écrire(tête(pile))
7. Dépile
8. **Fin du tant que**

Pour utiliser cet algorithme pour un arbre de Huffman quelconque (non parfait) représenté selon la numérotation de Tanaka, nous allons définir comment l'on passe de la numérotation de Tanaka à la numérotation en ordre hiérarchique total, puis une bijection qui associe à tout nœud d'un arbre de Huffman quelconque, un nœud dans l'arbre de Huffman parfait ayant le même nombre de symboles.

Le passage de la numérotation de Tanaka à la numérotation en ordre hiérarchique total se fait aisément, on associe à tout couple $M(i,j) = k$ le couple $M_h(i,j)$ défini par :

$$M_h(i,j) = \begin{cases} k & \text{si } k \text{ est positif} \\ q-2-k & \text{si } k \text{ est négatif} \end{cases}$$

Considérons l'arbre de Huffman parfait numéroté à 6 symboles selon la numérotation de Tanaka et renumérotons-le. On obtient l'arbre de la figure 2.8.

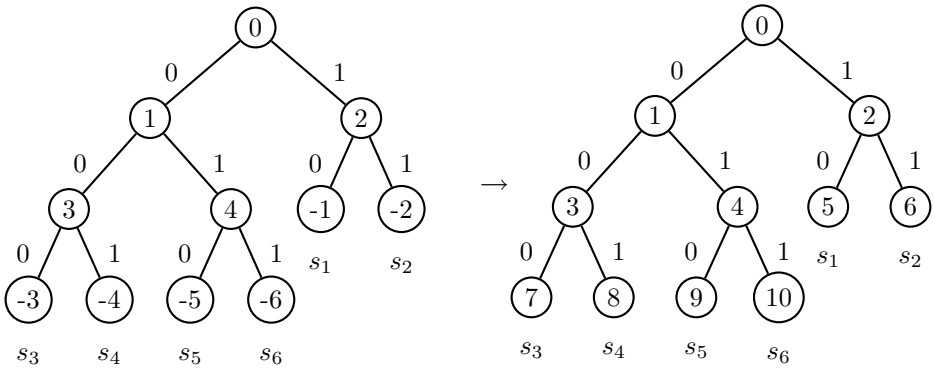


FIG. 2.8 – Arbre parfait à 6 symboles

Considérons à nouveau l'arbre de la figure 2.6 et renumérotons-le. On obtient l'arbre de la figure 2.9.

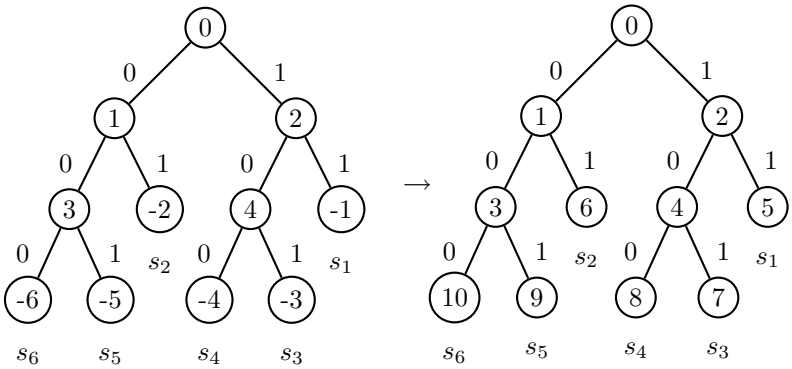


FIG. 2.9 – Arbre renuméroté

On appelle p la bijection qui associe à tout nœud $M_h(i,j)$ d'un arbre de Huffman quelconque le nœud $P(i,j)$ de l'arbre de Huffman parfait portant un nombre de symboles identique.

Si nous considérons l'arbre de la figure 2.9, nous obtenons :

$M_h(i,j)$	0	1
0	1	2
1	3	4
2	5	6
3	7	8
4	9	10

et

P	0	1
0	1	2
1	3	6
2	4	5
3	10	9
4	8	7

$\rightarrow$

n	$p(n)$
1	1
2	2
3	3
4	4
5	5
6	4
7	10
8	9
9	8
10	7

L'algorithme permettant de coder le symbole s_k s'établit alors ainsi :

1. $n = q - 2 - k$
2. **Tant que** $n <> 0$ **Faire**
3. $\quad n = p(n)$
4. $\quad$ Empile(bit(n))
5. $\quad n = père(n)$
6. **Fin du tant que**
7. **Tant que** pile non vide **Faire**
8. $\quad$ Écrire(tête(pile))
9. $\quad$ Dépile
10. **Fin du tant que**

2.3.2 L'algorithme de Sieminski

2.3.2.1 Un transducteur de décodage

Le décodage d'une donnée codée par l'algorithme, qu'il se fasse par un parcours d'arbre ou par l'automate de Tanaka précédemment étudié, utilise une lecture bit à bit qui a pour inconvénient de ralentir le temps de décodage. L'algorithme proposé par Sieminski permet de lire des blocs de bits de longueur fixe (notamment des octets) et ainsi d'améliorer la vitesse de décodage. Il fait appel à un transducteur qui lit des blocs de longueur l et qui écrit les symboles reconnus présents dans ce bloc. Après avoir lu un bloc, le transducteur mémorise dans l'état d'arrivée les bits qui n'ont pu encore être traduits. Ces bits non encore traduits sont appelés contexte. Le nombre de contextes est égal au nombre de préfixes possibles et donc au nombre de nœuds internes de l'arbre de Huffman.

Le transducteur de décodage, traitant des blocs de deux bits, qui correspond au code de la figure 2.4 est en figure 2.10.

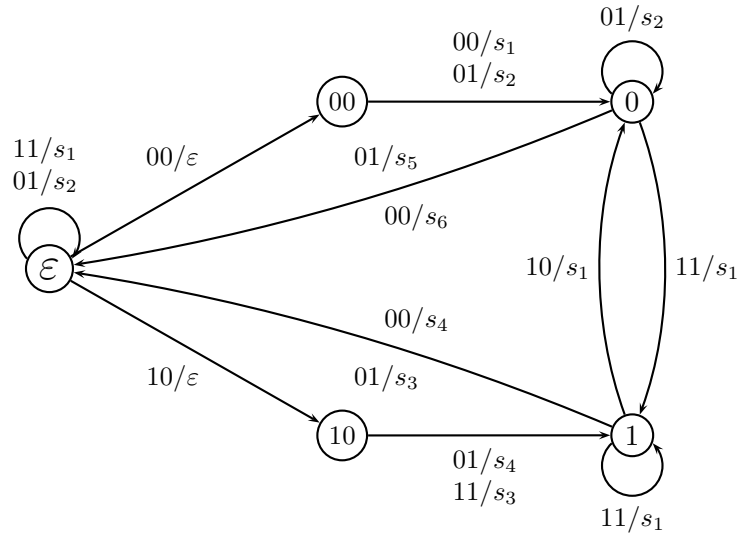


FIG. 2.10 – Transducteur de décodage

2.3.2.2 Un transducteur de codage

On peut construire de façon similaire un transducteur pour le codage. Celui-ci a été proposé, par Marc Zipstein [71]. Dans le cas du codage, le transducteur écrit un nombre entier de blocs de taille l. Les états du transducteur mémorisent les bits qui n'ont pas été encore émis.

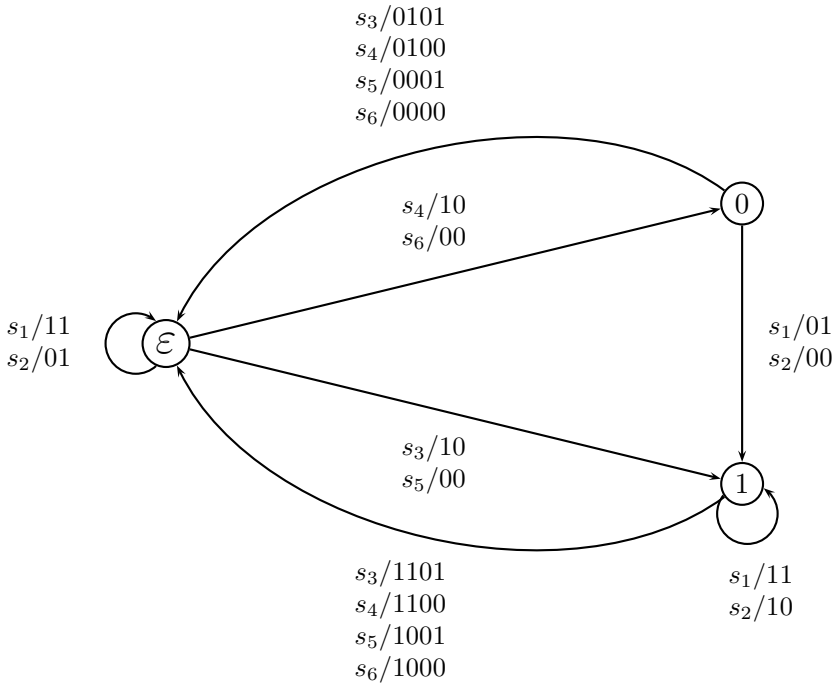


FIG. 2.11 – Transducteur de codage

2.4 L'algorithme de Huffman adaptatif

L'algorithme de Huffman fait appel à deux lectures du texte traité, la première calcule les fréquences des symboles et par la suite leur code de Huffman, la seconde remplace chaque symbole par son code de Huffman. N. Faller en 1973 [19] puis R.G. Gallager en 1978 [23] ont proposé un algorithme *adaptatif* dans lequel les codes des symboles sont calculés en fonction de leurs fréquences dans la partie du texte déjà traité. Ces algorithmes, qui ont été perfectionnés par Knuth [35] et Vitter [65], permettent de ne reconstruire qu'une branche de l'arbre de Huffman. Ces algorithmes permettent d'obtenir des taux de compression semblables à ceux obtenus par l'algorithme de Huffman statique. Ils sont à la base de la commande *compact* de UNIX, comme l'algorithme de Huffman statique est implémenté dans la commande *pack*. Ils ne sont plus guère utilisés aujourd'hui, on leur préfère des algorithmes adaptatifs fondés sur le *codage arithmétique*[4]. Il faut remarquer que si la version adaptative permet de ne faire qu'une lecture du texte, l'algorithme de Huffman statique en attribuant un code unique à chaque symbole permet de conserver l'accès direct au texte.

2.5 Résultats et conclusion

Voici les résultats obtenus sur le corpus de texte présenté dans l'introduction de cette thèse.

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.bible
Taille	125179	152089	426754	481861	2473400	4047392
Huffman	76010	88116	251544	276890	156279	2225674
statique	60,7%	57,9%	58,9%	57,5%	63,3%	55,0%
Huffman	75888	87704	249561	275662	1552734	2211619
adaptatif	**60,6%**	**57,7%**	**58,5%**	**57,2%**	**62,8%**	**54,6%**

On constate ici que les taux de compression obtenus par l'algorithme de Huffman statique et la version adaptative sont semblables. La version adaptative de l'algorithme de Huffman a été faite à l'origine pour éviter une double lecture du texte et ainsi en réduire le temps d'exécution. La vitesse des machines actuelles et l'absence d'accès direct au texte qu'il induit, font qu'il n'est pratiquement plus utilisé aujourd'hui. Si l'algorithme de Huffman statique n'est plus utilisé seul aujourd'hui pour coder directement des caractères, il est largement utilisé comme codeur final dans les algorithmes **gzip** et **bzip2** présentés dans le chapitre 3 et le codage à base de mots que nous présentons dans le chapitre 8.

[4] Voir chapitre suivant

Chapitre 3

Codage arithmétique

Dans ce chapitre, après avoir rappelé le fonctionnement du codage arithmétique et la notion de modèle, nous présentons plusieurs algorithmes fondés sur les chaînes de Markov et d'autres fondés sur des prétraitements du texte.

3.1 Les origines du codage arithmétique

Le *codage arithmétique* a été proposé dans les années soixante par Rissanen [51], puis par Rissanen et Langdon [52] en 1979 et Guazzo [24] en 1980, mais c'est seulement en 1987 qu'a été proposée une implémentation pratique de ce codage qui est due à Witten, Neal et Cleary [70]. Comme le codage de Huffman ou celui de Shannon-Fano, c'est un codage statistique qui représente chaque symbole en fonction de sa probabilité d'apparition. La manière dont les probabilités sont calculées et les paramètres pris en compte pour leur calcul constituent ce que l'on appelle un *modèle de probabilité* . Le codage arithmétique a deux avantages essentiels : le premier est qu'il est capable de coder une information sur un nombre non entier de bits, le second est qu'il permet une nette séparation entre le modèle de probabilité et le codeur. La seule information dont le codeur ou le décodeur doivent disposer est le modèle qui détermine les fréquences des symboles. De nombreux modèles de probabilités ont été proposés. Nous présentons dans ce chapitre plusieurs modèles qui prédisent le caractère courant en fonction des caractères qui le précèdent. Le modèle PPM a été proposé par Cleary et Witten [12] en 1984 perfectionné par Moffat [42] puis PPM* par Cleary, Witten et Teahan [11, 12]. Nous présentons aussi des modèles faisant appel à un prétraitement du texte, ceux d'Abrahamson [1] et de Burrows-Wheeler [10].

3.2 Fonctionnement du codage arithmétique

Le codage arithmétique consiste à représenter le texte compressé par un intervalle inclus dans l'intervalle [0,1[. Cet intervalle est réduit au fur et à mesure que le texte est traité. L'intervalle courant est partitionné en autant d'intervalles qu'il y de caractères potentiels à représenter. La longueur de l'intervalle associé à un caractère est proportionnelle à sa probabilité. Chaque nouveau symbole traité réduit la taille de cet intervalle d'autant moins que ce symbole est fréquent. La partition de l'intervalle pouvant être changée après chaque caractère traité, le *codage arithmétique* est parfaitement adapté pour tirer parti de distributions locales de probabilités.

Exemple : Considérons l'alphabet $A = \{a, b, c\}$, dont les probabilités d'apparition des éléments sont $P(a) = 0.6$, $P(b) = 0.3$ et $P(c) = 0.1$.

Procédons au codage du texte T=`abc`.

Le caractère `a` est représenté par l'intervalle [0,0.6[qui devient le nouvel intervalle de travail, puis on procède au codage du `b` dans ce nouvel intervalle.

Les étapes successives du codage sont illustrées par la figure 3.1.

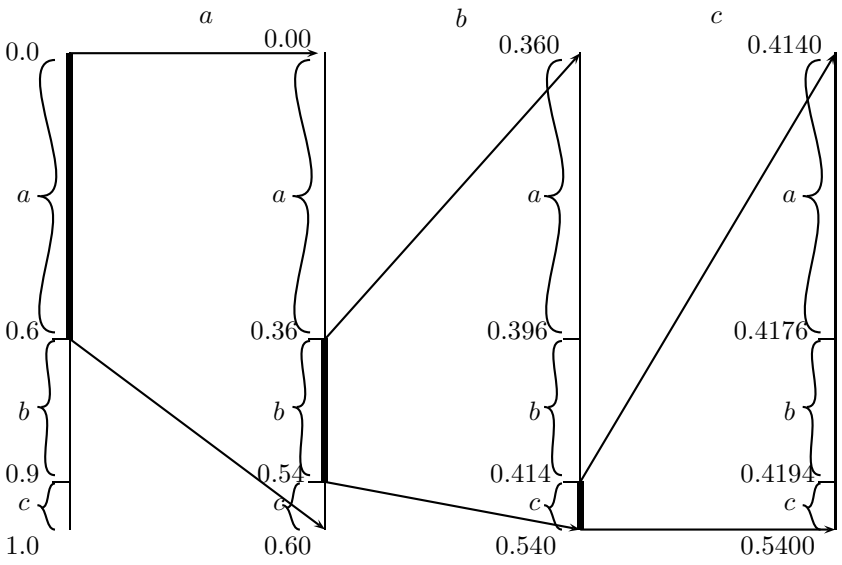


FIG. 3.1 – Les étapes du codage arithmétique

Le décodage s'effectue de manière similaire au codage. Pour décoder un intervalle J $\in$ [0,1[, il suffit de déterminer, à chaque étape, le caractère associé à l'intervalle contenant J, puis de recommencer avec ce nouvel intervalle jusqu'à ce que l'on obtienne exactement J.

Considérons le décodage de l'intervalle $[0.414, 0.540[$.

Le décodeur dispose des probabilités d'apparition des lettres : $P(a) = 0.6$, $P(b) = 0.3$ et $P(c) = 0.1$.

On peut voir les différentes étapes de décodage sur la figure 3.2.

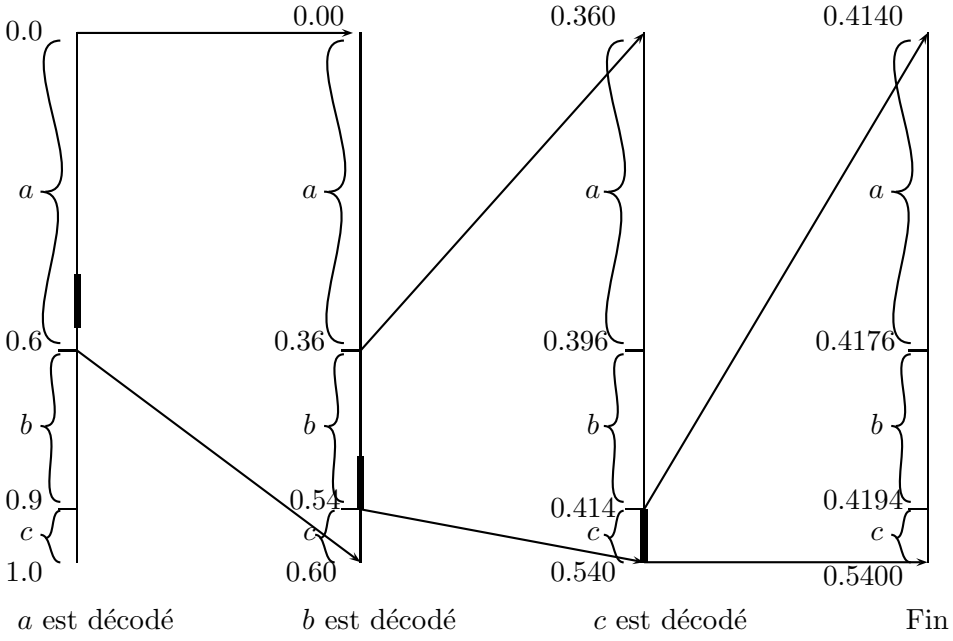


FIG. 3.2 – Les différentes étapes du décodage arithmétique

Nous avons donné une présentation théorique du codage arithmétique. L'algorithme de codage ne produit rien tant que le texte n'a pas été traité en totalité. Dans la pratique, on utilise un algorithme qui est une approximation du modèle théorique précédent, utilisant une fenêtre qui se déplace sur l'écriture des réels manipulés. La longueur N de la fenêtre est égale au nombre maximum de bits utilisables pour écrire un entier sans risque de dépassement de capacité lors du calcul de l'intervalle associé à une lettre. On travaille donc sur des intervalles entiers, l'intervalle initial étant $[0,2^N\text{-}1[$. Si la qualité des taux de compression obtenus dépend de la précision des calculs permise par l'implémentation utilisée, elle dépend aussi et surtout des modèles de probabilités utilisés. Nous présentons maintenant plusieurs modèles dont certains font appel à des prétraitements du texte. Certains modèles calculent la probabilité du caractère courant en fonction des k caractères qui le précèdent. Cet ensemble de k caractères est appelé contexte. On parle alors d'algorithme de compression avec un contexte d'ordre k ou de modèle d'ordre k . Un codage qui ne tient compte d'aucun caractère précédent est donc d'ordre 0.

3.3 Modèle d'ordre 1

Les taux de compression obtenus par un algorithme de compression de type statistique sont tributaires de la précision et de l'exactitude des probabilités utilisées. Si l'on s'intéresse à la fréquence de la lettre `u` en français, on peut dire qu'elle est relativement faible, de l'ordre de 5%. En revanche une occurrence de la lettre `q` est quasiment toujours suivie de celle de la lettre `u`. Ainsi la probabilité d'apparition de la lettre `u` après la lettre `q` est très élevée. La connaissance du caractère qui précède le caractère courant permet d'affiner très nettement l'estimation de la probabilité d'apparition du caractère courant. Un modèle qui permet le codage d'un caractère en fonction du caractère précédent s'appelle modèle d'ordre 1. Si A est l'alphabet du texte, l'implémentation la plus simple d'un tel modèle utilise un tableau de taille $|A|^2$. La case (x, y) du tableau contient la fréquence du caractère y quand il est précédé du caractère x. Tout couple (x, y) de $A \times A$ étant susceptible d'exister dans le texte, on est obligé d'initialiser, à la compression, comme à la décompression, toutes les cases du tableau avec la fréquence 1. Une implémentation permettant de conserver certaines cases nulles nécessiterait l'utilisation d'un système d'échappement qui est utilisé dans un cadre plus général, le modèle d'ordre k.

3.4 Modèle d'ordre k et algorithme PPM

3.4.1 Principe de fonctionnement

Le principe de base de ces algorithmes est de prédire le prochain caractère codé en fonction des k précédents. La généralisation à l'ordre k de l'implémentation précédemment utilisée pour le modèle d'ordre 1 se heurte à la croissance exponentielle de l'espace mémoire nécessaire : $|A|^k$. Pour résoudre ce problème, un algorithme appelé PPM (*Prediction by Partial Matching*) a été initialement proposé en 1984 par Cleary et Witten [5, 12] puis amélioré par Moffat [42] qui en a proposé une implémentation pratique.

L'algorithme PPM utilise un contexte d'ordre maximum k, préalablement fixé. Supposons que k soit égal à 2 et que nous codions la lettre z dans le contexte xy. Plusieurs cas peuvent se présenter.

- Si la lettre z est déjà apparue dans le contexte xy, le modèle utilise le nombre de fois où elle est apparue dans ce contexte pour calculer la probabilité. Le codeur en déduit sa représentation. La fréquence de z est ensuite incrémentée dans les contextes xy, y et ε.

- Si par contre la lettre z n'est pas connue dans ce contexte, on envoie au codeur un caractère spécial appelé code d'échappement et noté Esc qui signifie que l'on essaye maintenant de coder z dans le contexte y (à l'ordre 1) . Si c'est impossible, on essaye à l'ordre 0. Enfin, si la lettre z n'est jamais apparue, on passe à un ordre spécial -1 dans lequel tous les caractères existent avec une fréquence égale à 1.

- Si le contexte xy n'existe pas, on le crée avec le symbole Esc affecté de la fréquence 1, puis l'on tente de coder la lettre z dans le contexte y. On procède ainsi, éventuellement jusqu'à l'ordre -1 où par définition z existe.

3.4.2 Calcul des probabilités des codes d'échappement

Considérons un symbole s appartenant au contexte courant d'ordre o ($-1 \leq o \leq k$) ayant pour fréquence $f_o(s)$ et pour probabilité $p_o(s)$.

On appelle :

- n_o le nombre de caractères apparus dans ce contexte depuis le début du traitement.
$$n_o = \sum_{s \in A} f_o(s)$$
- d_o le nombre de caractères différents apparus dans ce contexte depuis le début du traitement.
- t_1 représente le nombre de caractères différents apparus une fois et une seule dans un contexte depuis le début du traitement).

Plusieurs méthodes ont été proposées pour calculer la probabilité du code d'échappement.

Méthode	$p_o(Esc)$
A	$\frac{1}{n_o+1}$
B	$\frac{d_o-t_1}{n_o}$
C	$\frac{d_o}{n_o+d_o}$

Les méthodes A et B, dues à Cleary et Witten [69, 11], la méthode C par Moffat [42]. Le problème du calcul des probabilités des caractères d'échappement, connu sous le nom de « *zero frequency problem* » [63], est un champ de recherche important pour améliorer les modèles de probabilités.

Exemple : considérons le traitement du texte T=`abracadabra` avec $k = 2$. Le tableau suivant représente l'arbre des contextes après le traitement de `abrac`. Pour chaque ordre et chaque contexte sont indiqués les caractères apparus dans le contexte, suivis de leurs fréquence et probabilité. La méthode C est utilisée pour le traitement des caractères d'échappements.

Ordre k = 2				Ordre k = 1				Ordre k = 0			Ordre k = -1		
Prédiction		fr	pb	Prédiction		fr	pb	Prédiction	fr	pb	Prédiction	fr	pb
ab	r	1	$\frac{1}{2}$	a	b	1	$\frac{1}{4}$	a	2	$\frac{2}{9}$	A	1	$\frac{1}{\|A\|}$
	Esc	1	$\frac{1}{2}$		c	1	$\frac{1}{4}$	b	1	$\frac{1}{9}$			
					Esc	2	$\frac{1}{2}$	c	1	$\frac{1}{9}$			
br	a	1	$\frac{1}{2}$					r	1	$\frac{1}{9}$			
	Esc	1	$\frac{1}{2}$	b	r	1	$\frac{1}{2}$	Esc	4	$\frac{4}{9}$			
					Esc	1	$\frac{1}{2}$						
ra	c	1	$\frac{1}{2}$										
	Esc	1	$\frac{1}{2}$	r	a	1	$\frac{1}{2}$						
					Esc	1	$\frac{1}{2}$						

TAB. 3.1 – Traitement de `abra`

Par exemple : dans le contexte `ab` (ordre 2), `r` est apparu une fois. Dans le contexte `a` (ordre 1), `b` et `c` sont apparus chacun une fois. A l'ordre 0, nous avons les fréquences des caractères dans le texte `abrac`. Considérons le traitement du `a` suivant et donc du texte `abraca`.

Ordre k = 2				Ordre k = 1				Ordre k = 0			Ordre k = -1		
Prédiction		fr	pb	Prédiction		fr	pb	Prédiction	fr	pb	Prédiction	fr	pb
ab	r	1	$\frac{1}{2}$	a	b	1	$\frac{1}{4}$	a	**3**	$\frac{3}{10}$	A	1	$\frac{1}{\|A\|}$
	Esc	1	$\frac{1}{2}$		c	1	$\frac{1}{4}$	b	1	$\frac{1}{10}$			
					Esc	2	$\frac{1}{2}$	c	1	$\frac{1}{10}$			
ac	**a**	**1**	$\frac{1}{2}$					r	1	$\frac{1}{10}$			
	Esc	**1**	$\frac{1}{2}$	b	r	1	$\frac{1}{2}$	Esc	4	$\frac{4}{10}$			
					Esc	1	$\frac{1}{2}$						
br	a	1	$\frac{1}{2}$										
	Esc	1	$\frac{1}{2}$	**c**	**a**	**1**	$\frac{1}{2}$						
					Esc	**1**	$\frac{1}{2}$						
ra	c	1	$\frac{1}{2}$										
	Esc	1	$\frac{1}{2}$	r	a	1	$\frac{1}{2}$						
					Esc	1	$\frac{1}{2}$						

TAB. 3.2 – Traitement de `abrac`

Le traitement de la lettre `a` a provoqué la création des contextes, `ac` et `c` dans lesquels `a` apparait et l'augmentation de sa fréquence à l'ordre 0.

Après que la totalité du texte T ait été traitée, on obtient l'arbre de contexte :

Ordre k = 2				Ordre k = 1				Ordre k = 0			Ordre k = -1		
Prédiction		fr	pb	Prédiction		fr	pb	Prédiction	fr	pb	Prédiction	fr	pb
ab	r	2	$\frac{2}{3}$	a	b	2	$\frac{2}{7}$	a	5	$\frac{5}{16}$	A	1	$\frac{1}{\|A\|}$
	Esc	1	$\frac{1}{3}$		c	1	$\frac{1}{7}$	b	2	$\frac{2}{16}$			
					d	1	$\frac{1}{7}$	c	1	$\frac{1}{16}$			
ac	a	1	$\frac{1}{2}$		Esc	3	$\frac{3}{7}$	d	1	$\frac{1}{16}$			
	Esc	1	$\frac{1}{2}$					r	2	$\frac{2}{16}$			
				b	r	2	$\frac{2}{3}$	Esc	5	$\frac{5}{16}$			
ad	a	1	$\frac{1}{2}$		Esc	3	$\frac{3}{7}$						
	Esc	1	$\frac{1}{2}$										
				c	a	1	$\frac{1}{2}$						
br	a	2	$\frac{2}{3}$		Esc	1	$\frac{1}{3}$						
	Esc	1	$\frac{1}{3}$										
				d	a	1	$\frac{1}{2}$						
ca	d	1	$\frac{1}{2}$		Esc	1	$\frac{1}{2}$						
	Esc	1	$\frac{1}{2}$										
				r	a	2	$\frac{2}{3}$						
da	b	1	$\frac{1}{2}$		Esc	1	$\frac{1}{3}$						
	Esc	1	$\frac{1}{2}$										
ra	c	1	$\frac{1}{2}$										
	Esc	1	$\frac{1}{2}$										

TAB. 3.3 – Traitement de `abracadabra`

3.4.3 Exclusion de contexte

Ces modèles de probabilités peuvent aussi être affinés en utilisant un principe appelé *exclusion*. Supposons que l'on traite un caractère x dans le contexte d'ordre p, inférieur au contexte d'ordre maximum k. Le fait que x n'ait pu être codé dans les contextes d'ordre i, $(p < i \leq k)$ nous apprend que x est différent de l'ensemble des caractères connus dans ces contextes. Par conséquent, les caractères de cet ensemble présents dans le contexte d'ordre p, peuvent en être exclus pour le calcul de la probabilité de x.

Exemple :

Supposons que la chaîne `abracadabra` soit suivie d'un `c`, `c` est connu dans le contexte `ra` et la probabilité fournie par le modèle étant $\frac{1}{2}$, il peut être codé par un bit. Supposons maintenant que la chaîne `abracadabra` soit suivie d'un `d`, ce caractère n'est pas connu dans le contexte `ra`, il faut donc émettre un *Esc*. Le caractère `d` est connu dans le contexte `a`, nous codons `d` en excluant `c` du calcul des probabilités dans ce contexte car nous savons que le caractère traité ne peut être `c`. Si tel était le cas, il aurait été codé grâce au contexte `ra`.

Le tableau 3.4 montre la suite de symboles émis dans les contextes `ra` (ordre 2), `a` (ordre 1) puis aux ordres 0 et -1 ainsi que le calcul des probabilités avec et sans exclusion des caractères `c`, `d`, `t` si ceux-ci suivent la chaîne `abracadabra`.

caractère	caractères émis	pb. sans exclusion	nombre de bits
c	c	$\frac{1}{2}$	-$\log_2(\frac{1}{2}) = 1bit$
d	Esc, d	$\frac{1}{2}$, $\frac{1}{7}$	-$\log_2(\frac{1}{2} \times \frac{1}{7}) = 3.8bits$
t	Esc,Esc,Esc,ϵ	$\frac{1}{2}$, $\frac{3}{7}$, $\frac{5}{16}$, $\frac{1}{\lvert A\rvert}$	-$\log_2(\frac{1}{2} \times \frac{3}{6} \times \frac{5}{16} \times \frac{1}{256}) = 11.9bits$
caractère	caractères émis	pb. avec exclusion	nombre de bits
c	c	$\frac{1}{2}$	-$\log_2(\frac{1}{2}) = 1bit$
d	Esc, d	$\frac{1}{2}$, $\frac{1}{6}$	-$\log_2(\frac{1}{2} \times \frac{1}{6}) = 3.6bits$
t	Esc,Esc,Esc,ϵ	$\frac{1}{2}$, $\frac{3}{6}$, $\frac{5}{12}$, $\frac{1}{\lvert A\rvert-5}$	-$\log_2(\frac{1}{2} \times \frac{3}{6} \times \frac{5}{12} \times \frac{1}{251}) = 11.2bits$

TAB. 3.4 – Calcul des probabilités avec et sans exclusion

Si l'on regarde la deuxième ligne de la troisième colonne du tableau 3.4, on voit que le codage de `d` consiste à émettre un Esc dans le contexte `ra` suivi d'un `d` dans le contexte `a` privé de `c`.

3.5 PPM avec ordre non borné : PPM*

L'algorithme PPM utilise toujours la même suite de contextes partant du contexte d'ordre maximum et allant éventuellement jusqu'au contexte d'ordre -1. Le fait de partir du contexte d'ordre le plus élevé est particulièrement pénalisant au tout début du traitement du texte. En effet, le seul contexte existant alors est le contexte d'ordre -1 et l'algorithme PPM commence par une phase "d'apprentissage" qui consiste à créer des contextes et à émettre des codes d'échappements.

Cleary, Teahan et Witten [13] proposent un algorithme utilisant un nombre de contextes *a priori* non borné appelé PPM* dans lequel le choix du contexte fait appel à la notion de *contexte déterministe*. Un contexte est dit *déterministe* s'il ne donne qu'une seule prédiction c'est-à-dire s'il n'existe dans ce contexte qu'un caractère possible x, $x \in A$ (et Esc). Ils proposent de choisir le contexte déterministe d'ordre le plus faible. Si il n'y a pas de contexte déterministe, le contexte d'ordre le plus élevé est retenu. Cet algorithme, bien que donnant les meilleurs taux de compression, s'avère très coûteux en temps et en espace mémoire et n'est pas, dans son implémentation actuelle, utilisable dans la pratique.

3.6 Modèles avec transformation du texte

3.6.1 Le modèle d'Abrahamson

David Abrahamson [1] propose un modèle dans lequel la représentation d'un caractère dépend du caractère qui le précède. On considère un alphabet A et une table de taille $A \times A$, notée Δ, dans laquelle la ligne indicée par le caractère x est formée par la liste ordonnée par fréquences d'apparition des caractères qui suivent x dans la partie du texte déjà traitée. Lors du traitement du caractère courant, on considère le couple (caractère précédent, caractère courant) noté (p, c) et l'on remplace le caractère courant par le caractère α tel que :

$$\Delta(p, \alpha) = c$$

On ajoute 1 à la fréquence du caractère $\Delta(p, \alpha)$ et l'on réordonne la liste des successeurs de p. Le caractère α est alors traité par le codage arithmétique. Bien que l'algorithme d'Abrahamson traite le texte en une seule passe, on peut voir ce modèle comme un prétraitement du texte que l'on pourrait appeler « substitution d'ordre 1 », qui traduit un texte T en un texte T' suivi d'un codage arithmétique d'ordre 0 de T'.

3.6.1.1 Exemple

Considérons le traitement du texte T= `abracadabra`.

Initialement, la table se présente ainsi :

		caractère émis						
	Δ	a	b	c	d	...	r	...
								...
	a	a^0	b^0	c^0	d^0	...	r^0	...
	b	a^0	b^0	c^0	d^0	...	r^0	...
caractère	c	a^0	b^0	c^0	d^0	...	r^0	...
précédent	d	a^0	b^0	c^0	d^0	...	r^0	...
	.	a^0	b^0	c^0	d^0	...	r^0	...
	r	a^0	b^0	c^0	d^0	...	r^0	...
	.	a^0	b^0	c^0	d^0	...	r^0	...
	ε	a^0	b^0	c^0	d^0	...	r^0	...

TAB. 3.5 – Initialisation de la table

Les exposants des caractères dénotent leur fréquence d'apparition.

Après le traitement de `abracada`, on obtient : traduction(`abracada`)=`abracada` et la table 3.6. En effet, jusqu'ici il n'y a aucun facteur récurrent dans le texte traité, sa traduction est donc identique à l'original.

Δ	a	b	c	d	...	r	...
a	b^1	c^1	d^1	a^0	...	r^0	...
b	r^1	a^0	b^0	c^0	...	q^0	...
c	a^1	b^0	c^0	d^0	...	r^0	...
d	a^1	b^0	c^0	d^0	...	r^0	...
.	a^0	b^0	c^0	d^0	...	r^0	...
r	a^1	b^0	c^0	d^0	...	r^0	...
.	a^0	b^0	c^0	d^0	...	r^0	...
ε	a^1	b^0	c^0	d^0	...	r^0	...

TAB. 3.6 – Table après le traitement `abracada`

La partie du texte restant à traduire `bra` ayant déjà été rencontrée, le système de substitution lui est appliqué. Le `b` dans le contexte `a` est traduit par `a`, il en est de même pour `r` dans le contexte `b` et le `a` dans le contexte `r`.

A la fin du traitement, nous avons donc :

Δ	a	b	c	d	...	r	...
a	b^2	c^1	d^1	a^0	...	r^0	...
b	r^2	a^0	b^0	c^0	...	q^0	...
c	a^1	b^0	c^0	d^0	...	r^0	...
d	a^1	b^0	c^0	d^0	...	r^0	...
.	a^0	b^0	c^0	d^0	...	r^0	...
r	a^2	b^0	c^0	d^0	...	r^0	...
.	a^0	b^0	c^0	d^0	...	r^0	...
ε	a^1	b^0	c^0	d^0	...	r^0	...

TAB. 3.7 – Table après le traitement `abracadabra`

et T'= traduction(`abracadabra`)=`abracadaaaa`

Cet algorithme qui change la distribution des fréquences des caractères du texte, augmentant celles de certains et diminuant celles d'autres caractères, permet à tout compresseur statistique d'obtenir sur le texte résultant, des taux de compression sensiblement meilleurs que sur le texte initial. Le texte résultant peut être traité par le codage arithmétique ou par celui de Huffman.

3.6.2 Le modèle de Burrows-Wheeler

La méthode de Burrows-Wheeler [10], également appelée *Block-Sorting* ou *Burrows-Wheeler Transform* et notée BWT effectue un traitement du texte par bloc. Elle effectue un prétraitement de chaque bloc avant de le coder par un codeur statistique (Huffman ou codeur arithmétique). Le traitement d'un bloc de longueur N fait appel à trois étapes : transformation du bloc, codage du bloc transformé par une application de l'algorithme *Move-to-Front*, codage statistique des données obtenues.

3.6.3 Transformation d'un bloc

Cette première étape consiste à construire les |B| permutations circulaires du bloc B et à les trier dans l'ordre lexicographique. On mémorise alors la dernière colonne, notée D, de la matrice triée et le numéro noté L de la ligne où le bloc initial B apparaît. Cette étape permet de concentrer dans une région de D des caractères identiques. Cette concentration sera exploitée par une application de l'algorithme *Move-to-Front*.

3.6.3.1 Exemple

Considérons la chaîne : $B =$ `abracadabra`.

Permutations circulaires		Permutations triées
a b r a c a d a b r a		a a b r a c a d a b r
b r a c a d a b r a a		a b r a a b r a c a d
r a c a d a b r a a b		a b r a c a d a b r a
a c a d a b r a a b r		a c a d a b r a a b r
c a d a b r a a b r a	→	a d a b r a a b r a c
a d a b r a a b r a c		b r a a b r a c a d a
d a b r a a b r a c a		b r a c a d a b r a a
a b r a a b r a c a d		c a d a b r a a b r a
b r a a b r a c a d a		d a b r a a b r a c a
r a a b r a c a d a b		r a a b r a c a d a b
a a b r a c a d a b r		r a c a d a b r a a b

FIG. 3.3 – Traitement d'un bloc

On mémorise la chaîne $D =$ `rdarcaaaabb` et la ligne L=2.

Montrons comment on reconstruit la chaîne initiale à partir de D et L.

- Pour reconstruire la première colonne, notée P, de la matrice, il suffit de trier lexicographiquement la chaîne D.

- Pendant ce tri, on construit le tableau T qui indique pour chaque caractère de D, sa position dans P. On a : $D[i] = P[T[i]]$.

- Par construction de la matrice triée, pour chaque ligne i, $D[i]$ précède cycliquement $P[i]$ (voir exemple figure 3.3).

- Notamment dans la ligne L, $D[L]$ précède cycliquement $P[L]$. Or $P[L]$ est le premier symbole de B, $D[L]$ est donc le dernier symbole de B.
  La reconstruction de B s'effectue donc de gauche à droite.

- $D[i]$ précède $P[i]$ dans B pour $i = 0, ..., N-1$. Ainsi $D[T[i]]$ précède $P[T[i]]$ mais par construction de T, $P[T[i]] = D[i]$ donc $D[T[i]]$ précède $D[i]$.

L'algorithme de reconstruction de B est le suivant :

for $i = 0$ to $n-1$ $B[n-1-i] = D[T^i[L]]$ avec $T^0[j] = j$, $T^{i+1} = T[T^i[j]]$

Sur l'exemple de la figure 3.3, on obtient :

i	P	T	D
0	a	9	r
1	a	8	d
2	a	0	a
3	a	10	r
4	a	7	c
5	b	1	a
6	b	2	a
7	c	3	a
8	d	4	a
9	r	5	b
10	r	6	b

FIG. 3.4 –

La reconstruction commence donc par $D[L] = D[2] =$ `a` puis $D[T[I]] = D[0] =$ `a` et ainsi de suite.

3.6.4 L'algorithme *Move-to-Front*

L'algorithme *Move-to-front*, noté *MTF* dû à Sleator, Bentley, Tarjan et Wei [7], permet de tenir compte de distributions locales de probabilités. Il consiste d'abord à initialiser[1] une liste contenant l'ensemble des symboles dans le texte à coder. Ensuite, chaque symbole du texte est représenté par sa position, comptée à partir de 0 dans la liste, puis déplacé en tête de liste. Les symboles les plus fréquents se retrouvent donc à des positions proches de la tête de la liste et sont donc indexés par de petits nombres représentables avec un faible nombre de bits.

Exemple :

Appliquons-le à la chaîne D= `rdarcaaaabb` :

On initialise la liste E = (`a`,`b`,`c`,`d`,`r`).

Codage				Décodage		
D	*E*	*Code*		*Code*	*E*	*D*
r	abcdr	4		4	abcdr	r
d	rabcd	4		4	rabcd	d
a	drabc	2		2	drabc	a
r	adrbc	2	et	2	adrbc	r
c	radbc	4		4	radbc	c
a	cradb	2		2	cradb	a
a	acrdb	0		0	acrdb	a
a	acrdb	0		0	acrdb	a
a	acrdb	0		0	acrdb	a
b	acrdb	4		4	acrdb	b
b	bacrd	0		0	bacrd	b

FIG. 3.5 – *Move to Front*

La chaîne D est traduite par la suite `4,4,2,2,4,2,0,0,0,4,0`.

Les données indices résultant de la phase de traitement par l'algorithme *MTF* sont alors envoyées au codage arithmétique. Dans les implémentations les plus récentes comme **bzip2** due à Julian Seward [57] le codage arithmétique est remplacé par le codage de Huffman.

[1] Cette initialisation qui peut être remplacée par l'utilisation d'un système d'échappement a été utilisée dans un but de clarté

3.7 Résultats et conclusions

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.bible
Taille	125179	152089	426754	481861	2473400	4047392
Huffman statique	76010	88116	251544	276890	156279	2225674
	60,7%	57,9%	58,9%	57,5%	63,3%	55,0%
HuffmanA	75888	87704	249561	275662	1552734	2211619
	60,6%	57,7%	58,5%	57,2%	62,8%	54,6%
Arithmétique	75723	87336	248630	274413	1545470	2201708
ordre 0	**60,4%**	**57,4%**	**58,2%**	**56,9%**	**62,4%**	**54,4%**
Abrahamson	58200	70740	200567	219368	1224609	1765794
	46,4%	46,5%	46,9%	45,5%	49,5%	43,6%
Arithétique	59750	71159	195616	211494	11469594	1666799
ordre 1	47,7%	46,8%	45,8%	43,9%	46,4%	41,2%
PPM_1	54775	66402	188431	204133	1129991	1645129
	43,8%	**43,7%**	**44,2%**	**42,4%**	**45,7%**	**40,7%**
PPM_2	44219	51948	147413	170196	876413	1234682
	35,3%	34,2%	34,5%	35,3%	35,4%	30,5%
PPM_2^e	43607	51318	146181	169004	870949	1230561
	34,8%	33,7%	34,2%	35,0%	35,2%	30,4%
PPM_3	41327	45558	119142	148954	664190	981271
	33,0%	29,9%	27,9%	30,9%	26,8%	24,2%
PPM_3^e	39659	43966	426754	145727	653090	974498
	31,6%	28,9%	27,2%	30,2%	26,4%	24,0%
PPM_4	43057	46228	426754	481861	545857	868870
	34,4%	30,4%	26,5%	30,8%	22,0%	21,5%
PPM_4^e	40117	43379	107476	141605	525957	854247
	32,0%	**28,5%**	**25,1%**	**29,3%**	21,2%	21,1%
PPM_5	46133	48926	118056	159485	514938	846213
	36,9%	32,2%	27,7%	33,1%	20,8%	20,9%
PPM_5^e	42148	44970	109595	147684	486607	818274
	33,6%	29,5%	25,6%	30,6%	**19,7%**	**20,2%**
PPM*	33196	42101	105479	144996	?	?
	26,5%	27,7%	24,7%	30,1%		
bred	48418	44411	131680	174047	692947	1056418
	35,5%	31,8%	30,9%	36,1%	28,0%	26,1%
bzip2	39569	43202	107706	145577	489583	845623
	31,7%	28,4%	26,2%	31,2%	19,8%	21,9%

Nous avons mis en évidence, (en caractères gras) les meilleurs résultats obtenus par la famille d'algorithmes : à l'ordre 0, à l'ordre 1, à l'ordre k, avec ordre non borné, et le modèle BWT. Les résultats concernant deux plus gros textes du corpus n'ont malheureusement pu être obtenus pour PPM* à cause d'un dépassement de capacité.

Nous présentons les résultats sur notre corpus de texte. La version de PPM d'ordre k est notée PPM_k, un signe `e` en exposant indique que cette version utilise le mécanisme d'exclusion. L'algorithme Bred est une implémentation de l'algorithme de Burrows-Wheeler due à David Wheeler.

Ces résultats montrent que, en ce qui concerne l'ordre 0, l'utilisation du codage arithmétique n'apporte pas d'amélioration sensible par rapport au codage de Huffman et que le niveau de compression obtenu est lié au modèle de probabilité utilisé.

On remarque également les différences prévisibles obtenues sur les trois modèles de compression à l'ordre 1.

Le modèle d'Abrahamson donne les moins bons résultats dans la mesure où, bien que le mécanisme de substitution d'ordre 1 rende compte du contexte dans lequel se trouve le caractère courant, le calcul de sa probabilité n'est pas fait avec des fréquences strictement locales à ce contexte, ni restreintes à ce contexte, mais avec des fréquences globales sur le texte résultant de la substitution d'ordre 1 : le nombre de fréquences prises en compte est $|A|$.

Le modèle d'ordre 1 utilisant une table de taille $|A|^2$ donne des résultats un peu meilleurs, mais l'initialisation de la table par des fréquences 1 fait entrer dans le calcul des probabilités des caractères qui n'ont en fait jamais été rencontrés dans le contexte courant. Par exemple, tous les couples de la forme (`q`,x), $x \in A$ ont tous une fréquence 1 alors que dans la pratique le seul couple de cette forme ayant une fréquence non nulle est (`q`,`u`).

Enfin PPM_1 obtient les meilleurs résultats car les seuls caractères pris en compte dans le contexte courant, pour le calcul de la probabilité du caractère courant, sont ceux effectivement rencontrés dans ce contexte.

En ce qui concerne les modèles de type PPM, l'utilisation du système d'exclusion permet de gagner environ 1% sur les taux de compression obtenus.
Pour la famille d'algorithme PPM_k^e on remarque que l'on obtient le plus souvent les meilleurs taux pour $k = 4$. Les taux les meilleurs obtenus avec $k = 5$ correspondent à des textes de taille importante (`world192`, `K.J. Bible`). Ceci s'explique par le fait que plus un facteur est long, plus il est caractéristique d'un mot ou suite de mots de la langue traitée. Il est d'autant plus probable de le rencontrer dans la partie du texte déjà traitée que le texte est long ou que son vocabulaire est réduit.

Si l'algorithme PPM* permet d'obtenir, dans la majorité des cas, les meilleurs taux de compression, il n'est malheureusement que peu utilisable dans la pratique, dans son implémentation actuelle qui nécessite de charger la totalité du texte traité en mémoire.

Chapitre 4

Codage par facteurs

Les algorithmes de compression, le codage de Huffman et le codage arithmétique exposés aux deux chapitres précédents, sont des codages de type statistique. Ils réalisent la compression du texte en réduisant le nombre moyen de bits nécessaires à la représentation de chaque caractère. Les algorithmes que nous présentons dans ce chapitre, tirent parti de la fréquente occurrence de séquences de caractères, appelées *facteurs*. Ils sont issus des travaux de Abraham Ziv et Jacob Lempel publiés en 1977 et 1978. Ils sont classés en deux familles d'algorithmes de compression, LZ77 [36] et LZ78 [37]. Le principe de base de ces algorithmes consiste à remplacer un facteur déjà rencontré dans une partie du texte, par une référence à cette occurrence antérieure. Ils construisent au fur et à mesure, à la compression comme à la décompression, les facteurs dont ils se servent, et n'en mémorisent aucun dans la forme compressée des textes traités.

4.1 Algorithmes de type LZ78

Ces algorithmes parcourent le texte et conservent dans une table, appelée *dictionnaire*, certains de ses facteurs. Ils compressent le texte, en cherchant le plus long facteur courant déjà présent dans le dictionnaire et en le remplaçant par son indice dans ce dictionnaire. Le dictionnaire est construit dynamiquement en mémoire, à la compression comme à la décompression.

4.1.1 Algorithme initial

L'algorithme d'origine émet une suite de couple de la forme (indice , caractère). L'indice se réfère à un facteur déjà présent dans le dictionnaire, le caractère est le premier caractère ne faisant pas partie du facteur reconnu. Le dictionnaire est initialement vide.

Si W est un facteur déjà présent dans la table et a le caractère courant, l'algorithme de codage s'exprime ainsi :

- Si Wa est dans le dictionnaire, on lit le caractère suivant avec $W = Wa$.
- Si Wa n'est pas dans le dictionnaire, on transmet l'indice de W dans le dictionnaire suivi du caractère a.
  On ajoute Wa au dictionnaire et l'on continue la lecture avec $W = \varepsilon$.

Considérons le texte $T =$ `aecdaecdaecaecab` et codons le.

Lettre lue	Couple émis	Dic	Indice
a	0,a	a	1
e	0,e	e	2
c	0,c	c	3
d	0,d	d	4
a			
e	1,e	ae	5
c			
d	3,d	cd	6
a			
e			
c	5,c	aec	7
a			
e			
c			
a	7,a	aeca	8
b	0,b	b	9

Le texte T est codé par la suite : `0, a 0,e 0, c 0, d 1, e 3, d 5, c 7, a 0, b`.

L'algorithme de codage est le suivant :

```
1.  W = caractère-lu
2.  Tant que ((a = Lire()) ≠ Fin de fichier)
3.      Si (Wa est dans la dictionnaire)
4.          W = Wa
5.      Sinon
6.          Ajouter(Wa,dictionnaire)
7.          Écrire(code(W))
8.          Écrire(code(a))
9.          W = ε
10.     Fin du sinon
11. Fin du tant que
12. Écrire(code(W))
```

Le dictionnaire peut être représenté sous forme d'arbre, celui associé au texte T est :

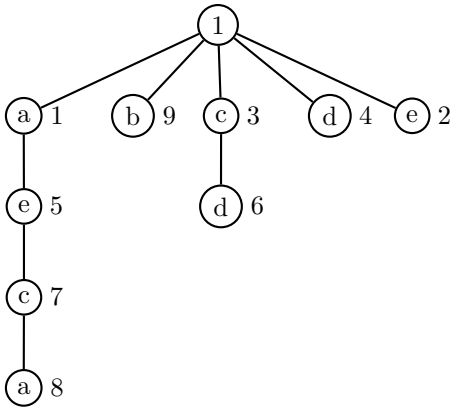


FIG. 4.1 – Dictionnaire

4.1.2 Amélioration de l'algorithme

Le fait d'émettre des couples de la forme (indice , caractère) implique qu'une partie des données n'est pas compressée (une lettre). Miller et Wegman [40] ainsi que Welch [67] ont proposé une méthode qui permet, en émettant seulement des indices, d'éviter cette perte d'efficacité. Le dictionnaire est initialisé avec l'ensemble des caractères de l'alphabet A.

L'algorithme s'écrit alors ainsi :

- Si Wa est dans le dictionnaire, on lit le caractère suivant avec $W = Wa$.
- Si Wa n'est pas dans le dictionnaire, on transmet l'indice de W dans le dictionnaire.
  On ajoute Wa au dictionnaire et l'on continue la lecture avec $W = a$.

Considérons a nouveau le texte T = `aecdaecdaecaecab` et l'alphabet A=`{a,b,c,d,e}` avec lequel le dictionnaire est initialisé.

Lettre lue	Code émis	Dictionnaire	Indice
		a	1
		b	2
		c	3
		d	4
		e	5
a			
e	1	ae	6
c	5	ec	7
d	3	cd	8
a	4	da	9
e			
c	6	aec	10
d			
a	8	cda	11
e			
c			
a	10	aeca	12
e			
c			
a			
b	12	aecab	13
	2		

Le texte est codé par la suite de valeurs : 1,5,3,4,6,8,10,12,2.

Remarquons que le nombre de symboles émis est de 18 avec l'algorithme original alors qu'il n'est que de 9 avec la version améliorée.

Au décodage, on initialise le dictionnaire avec l'alphabet A et l'on procède comme dans le cas de l'algorithme d'origine.

- On lit un code C.
- On écrit le mot W d'indice C dans le dictionnaire.
- On ajoute W suivi de la première lettre du mot suivant.

Cependant, il peut arriver que l'on doive décoder un indice correspondant à un mot non encore complété. Ceci ne se produit que si le mot est celui en cours de création. Il en est ainsi lorsque l'on traduit une chaîne de la forme $xWxWx$ alors que xW est déjà présent dans le dictionnaire.

Code lu	Lettres	Dictionnaire	Indice
		a	1
		b	2
		c	3
		d	4
		e	5
1	a		
5	e	ae	6
3	c	ec	7
4	d	cd	8
6	ae	da	9
8	cd	aec	10
10	aec	cda	11
12	aeca	aeca	12
2	b	aecab	13

Pendant le déroulement de cet algorithme de décompression, nous rencontrons le cas particulier évoqué précédemment avec $x =$`a` et $W =$ `ec`. Le décodage de 12 consiste à considérer le facteur `aec` auquel on ajoute sa première lettre `a`, en obtenant ainsi `aeca`.

Les algorithmes de compression et de décompression s'écrivent comme suit :

Compression

1. Initialiser(*Dictionnaire*)
2. W = Lire()
3. **Tant que** ((a = Lire()) $\neq$ Fin _de_ fichier) **Faire**
4. **Si** (Wa est dans le Dictionnaire)
5. $W = Wa$
6. **Fin du si**
7. **Sinon**
8. Ajouter(Wa,*Dictionnaire*)
9. Écrire(code(W))
10. $W = a$
11. **Fin du sinon**
12. **Fin tant que**
13. Écrire(code(W))

Décompression

1. Initialiser(*Dictionnaire*)
2. Lire(*code*)
3. Mot = Décode($Indice$)
4. Écrire(Mot)
5. **Tant que** (Lire(*code*) $\neq$ Fin_de_fichier) **Faire**
6. **Si** (*code* $\neq$ *dernier*)
7. $Nouveau\_mot$ = Décode($Indice$)
8. Écrire($Nouveau\_mot$)
9. Ajouter(Mot suivit de sa première lettre)
10. $Mot = Nouveau\_mot$
11. **Fin du si**
12. **Sinon**
13. Mot = Mot suivit de sa première lettre
14. Ajouter(Mot)
15. Écrire(Mot)
16. $dernier = dernier + 1$
17. **Fin du sinon**
18. **Fin tant que**

Pour l'algorithme d'origine comme pour la version améliorée, nous remarquons que si un facteur est présent dans le dictionnaire, alors tous ses préfixes le sont aussi. Cette notion de clôture préfixe a permis à Welch [67] de proposer une implémentation compacte du dictionnaire.

Afin d'éviter une perte dans la représentation des indices, Welch utilise un dictionnaire ayant 4096 entrées et une représentation des indices sur 12 bits. Miller et Wegman utilisent une représentation des indices sur $\lceil \log_2(N) \rceil$ bits où N représente le premier indice libre courant dans le dictionnaire. Cette représentation est celle retenue dans la version la plus utilisée de l'algorithme LZW nommée LZC [30] et implémentée dans la commande **UNIX** *compress*. Le dictionnaire étant initialisé avec les caractères ASCII de 0 à 255, le premier code libre est 256. Ainsi les codes sont-ils écrits avec 9 bits, puis 10 bits quand N atteint 512, puis 11 quand il atteint 1024 et ainsi de suite. Dans la plupart des implémentations, le nombre maximum de bits utilisés pour représenter un facteur est 16. Quand le dictionnaire a atteint sa taille maximum, il est réinitialisé et le processus continue en reconstruisant petit à petit un nouveau dictionnaire.

D'autres codages des indices ont été proposés, le *Phased in binary*[1] utilisé dans une variation nommée LZT [64], le codage de Huffman dans une autre nommée LZH [9]. Pour notre part, nous avons testé un autre codage, le codage arithmétique tel qu'il a été implémenté par Moffat, Neal et Witten [43] fondé sur des travaux dus à Fenwick, que nous développerons dans le chapitre 7.

[1] Voir Chapitre 6

4.2 Algorithmes de type LZ77

Cette famille d'algorithmes n'utilise pas de dictionnaire, mais une fenêtre qui coulisse sur le texte à traiter. Cette fenêtre comprend d'une part le texte déjà codé, d'autre part un tampon de prélecture. Le codage s'effectue en cherchant dans la fenêtre, le plus long facteur correspondant au facteur courant du tampon de prélecture. Ce facteur est alors représenté par sa position dans la fenêtre et sa longueur.

4.2.1 Algorithme d'origine

On considère une fenêtre coulissante de taille N comportant un tampon de prélecture de taille F. Il y a donc dans la fenêtre $N - F$ caractères déjà codés. A l'initialisation, les $N - F$ caractères sont des espaces et l'on charge les premiers caractères du texte dans le tampon de prélecture. La compression s'effectue en recherchant le plus long facteur, parmi les $N - F$ caractères déjà traités, correspondant à celui commençant dans le tampon de prélecture. Le facteur le plus long est représenté par le triplet (i,j,a) où i est la position du facteur par rapport au tampon de prélecture, j la longueur du facteur, et a le premier caractère du tampon ne faisant pas partie du facteur. Remarquons que le facteur le plus long peut chevaucher le tampon de prélecture.

Considérons l'exemple suivant :

Fenêtre de lecture

a	e	c	d	a	e	c	d	**a**	**e**	**c**	**a**	**e**	**c**	**a**	b

Texte déjà traité Tampon

Dans cet exemple, le facteur *aeca* sera codé par le triplet $(3, 4, b)$ On remarque que ce facteur chevauche le tampon de prélecture (le dernier a en faisant partie).

La position i peut être représentée par $\lceil \log_2(N - F) \rceil$ bits et la longueur par $\lceil \log_2(F) \rceil$ bits. Ainsi, si A est l'alphabet, la longueur totale représentant le triplet est :

$$L = \lceil \log_2(N - F) \rceil + \lceil \log_2(F) \rceil + \lceil \log_2(|A|) \rceil$$

4.2.2 Algorithme LZSS

L'algorithme LZSS apporte une amélioration similaire à celle qu'ont apportés Miller, Wegman et Welch à LZ78. En effet, les triplets émis par LZ77 peuvent être vus comme une stricte alternance d'un couple $(position, longueur)$ et d'un caractère a qui n'est pas compris dans un facteur. L'algorithme LZSS permet de mélanger des couples et des caractères en émettant un bit qui indique si la donnée suivante doit être interprétée comme un couple ou comme un caractère. Il utilise comme LZ77 une fenêtre de taille N. Soit p le plus petit entier tel que $p \log_2(|A|) > L^2$.

L'algorithme LZSS s'écrit alors ainsi :

Compression LZSS

Tant que (Tampon non vide) {
Construire le pointeur $(décalage, longueur)$ représentant le plus long facteur de la fenêtre coïncidant avec le facteur du tampon de prélecture.

Si $longueur > p$
Écrire un bit 1
Écrire $(décalage, longueur)$.
Décaler la fenêtre de $longueur$ caractères.
Sinon
Écrire un bit 0
Écrire le premier caractère du tampon.
Décaler la fenêtre de 1 caractère.
}

Pour simplifier l'implémentation, on peut numéroter les caractères du texte modulo N circulairement depuis le début du tampon comme on le montre ci-dessous :

Fenêtre de lecture

5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
a	e	c	d	a	e	c	d	a	e	c	a	e	c	a	b

Texte déjà traité Tampon

[2]Voir page suivante

Ainsi le premier paramètre d'un couple indique une position dans le tableau $[0..N-1]$. Les F premières valeurs qui sont inutilisées car elles correspondent au tampon de prélecture, peuvent servir à représenter des symboles spéciaux comme $Fin\_de\_fichier$. De plus F étant très inférieure à N, la perte éventuelle due à cette non utilisation est négligeable. La position peut être représentée par $\lceil \log_2(N-F) \rceil$ bits et la longueur par $\lceil \log_2(F) \rceil$ bits. Ainsi, si A est l'alphabet, la longueur totale représentant le facteur est :

$$L = \begin{cases} 1 + \lceil \log_2(|A|) \rceil & \text{si il n'y pas de facteur reconnu} \\ 1 + \lceil \log_2(N-F) \rceil + \lceil \log_2(F) \rceil & \text{sinon} \end{cases}$$

4.2.3 Algorithme Gzip

Qu'il s'agisse d'algorithmes de la famille LZ77 ou de la famille LZ78, ils se composent toujours de deux phases qui sont d'une part la factorisation, d'autre part le codage d'un indice ou plus généralement d'un pointeur. Le soin apporté à chacune de ces phases contribue à la qualité de la compression obtenue. L'algorithme Gzip dû à Jean-Loup Gailly [22] en est une parfaite illustration. Il fait partie de la famille LZ77 et est une version très améliorée de LZSS.

Gzip traite le texte par bloc de 64K, en calculant des couples de la forme $(l, \textit{décalage})$. Dans ce couple, le paramètre l représente soit une longueur, soit un littéral. Dans le second cas, le paramètre *décalage* est absent. Le fait d'émettre le paramètre l en premier permet de savoir s'il y a lieu ou non de lire un second paramètre et évite d'envoyer un bit de contrôle comme le fait LZSS. Les paramètres l et *décalage* présents dans le bloc sont représentés par des codes de Huffman. Il y a un arbre de Huffman pour coder les longueurs et littéraux et un arbre de Huffman pour coder les décalages. Ces arbres sont représentés sous forme compacte au début de chaque bloc.

De plus, pour tenter d'améliorer la compression, la recherche du plus long facteur est faite à partir du caractère courant, mais aussi à partir de celui qui le suit. Si le plus long facteur trouvé commence au caractère qui suit le caractère courant, le caractère courant est émis sous la forme d'un littéral.

4.3 Résultats

Dans le tableau suivant, nous donnons les taux de compression obtenus sur notre corpus par les algorithmes Compress, LZSS, gzip (avec les paramètres 1 et 9) ainsi que l'algorithme LZ78 avec codage des indices par le codage arithmétique.

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.bible
Taille	125179	152089	426754	481861	2473400	4047392
Huffman statique	76010	88116	251544	276890	156279	2225674
	60,7%	57,9%	58,9%	57,5%	63,3%	55,0%
LZW	54990	62247	163147	196963	920163	1377093
(compress)	43,9%	40,9%	65,4%	40,9%	37,2%	34,2%
LZSS	58825	64689	173679	235541	947854	1431364
	47,0%	42,5%	40,7%	48,9%	38,3%	35,4%
gzip-1	65144	56813	174132	228778	917896	1490042
	42,8%	45,3%	40,8%	47,4%	37,1%	36,8%
LZW+Ari	54168	61435	160990	481862	788454	1238445
	43,3%	40,4%	37,7%	40,6%	31,9%	30,6%
gzip-9	48829	54191	144429	194277	721413	1176649
	39,0%	**35,6%**	**33,8%**	**40,3%**	**29,1%**	**29,0%**
PPM_1	54775	66402	188431	204133	1129991	1645129
	43,8%	43,7%	44,2%	42,4%	45,7%	40,7%
PPM_2	44219	51948	147413	170196	876413	1234682
	35,3%	34,2%	34,5%	35,3%	35,4%	30,5%
PPM_2^e	43607	51318	146181	169004	870949	1230561
	34,8%	33,7%	34,2%	35,0%	35,2%	30,4%
PPM_3	41327	45558	119142	148954	664190	981271
	33,0%	29,9%	27,9%	30,9%	26,8%	24,2%
PPM_3^e	39659	43966	426754	145727	653090	974498
	31,6%	28,9%	27,2%	30,2%	26,4%	24,0%
PPM_4	43057	46228	426754	481861	545857	868870
	34,4%	30,4%	26,5%	30,8%	22,0%	21,5%
PPM_4^e	40117	43379	107476	141605	525957	854247
	32,0%	**28,5%**	**25,1%**	**29,3%**	21,2%	21,1%
PPM_5	46133	48926	118056	159485	514938	846213
	36,9%	32,2%	27,7%	33,1%	20,8%	20,9%
PPM_5^e	42148	44970	109595	147684	486607	818274
	33,6%	29,5%	25,6%	30,6%	**19,7%**	**20,2%**
bred	48418	44411	131680	174047	692947	1056418
	31,8%	35,5%	30,9%	36,%1	28,0%	26,1%
bzip2	39569	43202	107706	145577	489583	845623
	31,7%	**28,4%**	**26,2%**	**31,2%**	**19,8%**	**21,9%**

On remarque que les méthodes de codage par facteurs permettent d'obtenir des taux de compression nettement meilleurs que ceux obtenus par l'algorithme de Huffman. En revanche, ils sont inférieurs à ceux obtenus par les meilleurs algorithmes de la famille PPM. Cependant la lenteur relative de ceux-ci et la mémoire nécessaire à la création des arbres de contexte expliquent que l'algorithme Gzip, qui combine une recherche soignée du plus long facteur et un codage compact des indices, soit l'un des plus couramment utilisé ces dernières années. Il tend peu à peu à être remplacé par bzip2.

Seconde Partie

Compression à base de mots

Chapitre 5

Du caractère au mot

Dans la première partie de cette thèse, le caractère était l'unité de base de traitement. Dans cette seconde partie, nous nous intéressons aux algorithmes tirant partie du fait qu'un texte en langue naturelle est essentiellement formé de mots de la langue. Bien que cette approche implique, le plus souvent, que l'unité de base de traitement soit le mot, quelques algorithmes utilisent la notion de mot tout en codant le texte traité lettre par lettre. Nous commençons par décrire brièvement ces algorithmes, puis nous nous intéressons à la représentation des lexiques utilisés par les algorithmes qui procèdent à une prélecture du texte et qui le codent mot par mot.

5.1 Algorithmes utilisant la notion de mot

5.1.1 Une adaptation de l'algorithme de Huffman

L'algorithme de Huffman, comme nous l'avons vu, utilise la fréquence d'apparition des caractères du texte traité. Si l'on s'intéresse aux mots d'une langue, on constate que la distribution des lettres de l'alphabet n'est pas la même selon la position de la lettre dans le mot. Ainsi, en français, la proportion de mots commençant par un `z` est faible ; de même le `u`, bien que fort fréquent, l'est moins en première position. Cette observation est à la base de l'algorithme proposé par D. Basu [4] en 1991. Il utilise un arbre de Huffman par position de lettre dans le mot, ce qui permet de tenir compte de distributions de fréquences « locales » et d'améliorer ainsi le taux de compression.

5.1.2 L'algorithme DZ

L'algorithme DZ dû à D. Revuz et M. Zipstein [50] fait appel au codage arithmétique et à un modèle fondé sur la connaissance de la langue dans laquelle est écrit le texte à compresser (ici le français). Un texte étant essentiellement constitué de mots de cette langue, le modèle utilisé est un transducteur construit sur un lexique de 650000 mots du français, le L.E.L.A.F. (lexique électronique des formes fléchies du L.A.D.L.[1]) qui mémorise la fréquence d'apparition des lettres.

De manière à pouvoir traiter n'importe quelle suite de lettres, plusieurs types de transitions ont été ajoutés à l'automate acyclique qui représente le L.E.L.A.F. Tout d'abord, on ajoute à chaque état une $\varepsilon - transition$ retournant à l'état initial. Ces transitions ne sont utilisées que si la lettre lue n'est l'étiquette d'aucune transition sortante de l'état courant. On ajoute à l'état initial toutes les transitions de la forme (I, a, I) où a désigne une lettre qui n'est pas la première lettre d'un mot du français. L'automate privilégie ainsi les chemins correspondant à des mots du français.

De plus, le L.E.L.A.F. ne contenant que des mots en minuscules, on ajoute à l'état initial la transition (I, A, q) si la transition (I, a, q) existe, de manière à prendre en compte les majuscules en début de mot.

Enfin, le traitement de la ponctuation est fait en créant un état P. On ajoute alors à chaque état terminal les transitions de la forme (T, x, P) où x est un caractère séparateur différent de l'espace. Tout état terminal est également porteur d'une transition de la forme $(T, espace, I)$.

L'initialisation du transducteur consiste à associer une fréquence 1 à chaque transition.

Le codage de chaque lettre est effectué simultanément au parcours de l'automate. Les probabilités associées aux lettres sont déterminées localement en fonction de l'état courant. Si l'état courant est E et la lettre lue est l, deux cas peuvent se présenter. Si l est l'étiquette d'une transition de l'état E, elle est codée en fonction des fréquences portées par les transitions de E. Sinon, on code le mot vide et l'on retourne à l'état I. On continue alors le codage avec le couple (I, l). Le décodeur effectue un traitement équivalent.

Ainsi un mot connu du lexique donne lieu à un parcours ininterrompu conduisant de l'état 1 à un état final. Un mot inconnu est vu comme une suite de préfixes de mots connus du lexique, séparés par des mots vides.

[1] Laboratoire d'Automatique et de Documentation Linguistique

L'alphabet de caractères est A ={`a`,`d`,`e`,`i`,`n`,`s`,`u`}

L'alphabet des séparateurs est B ={',','.','espace'}

et le langage reconnu est L ={`de`,`des`,`du`,`la`,`le`,`les`,`un`,`une`}.

L'automate obtenu est :

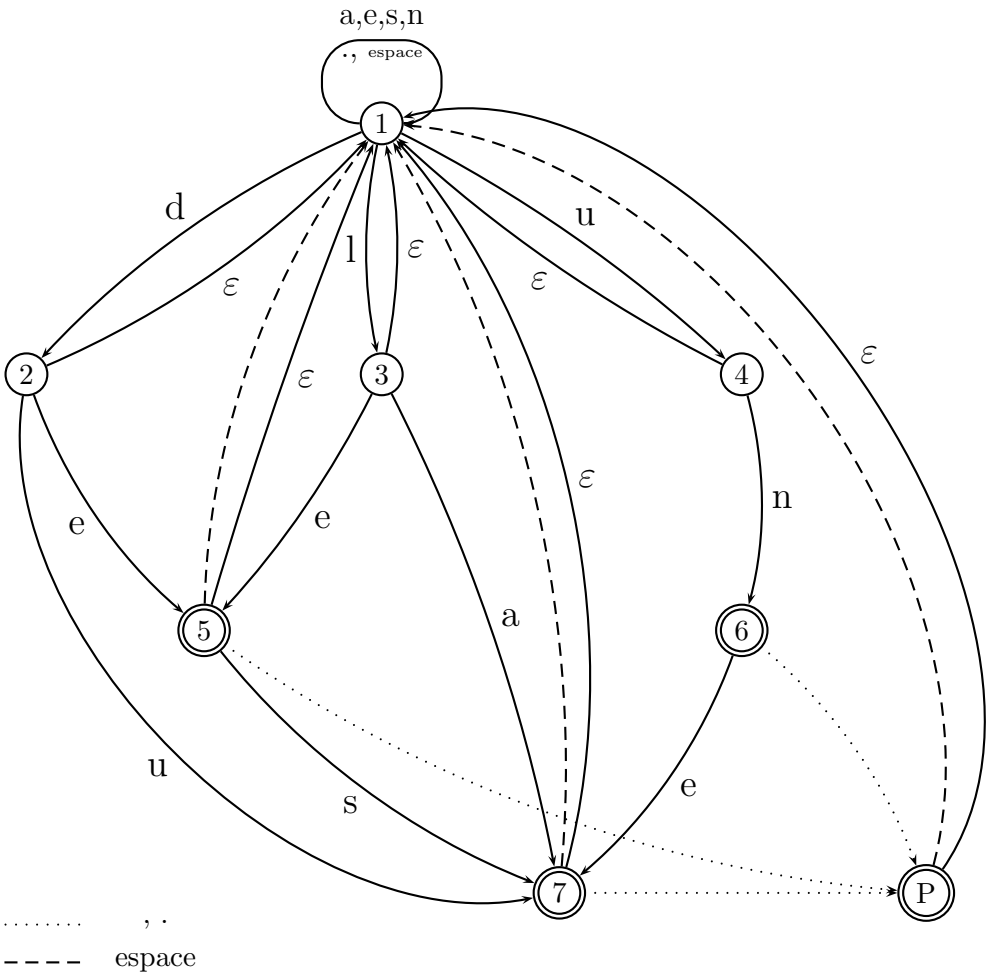


FIG. 5.1 – Modèle de DZ

La lecture de « la lune » correspond au parcours :

$1 - \texttt{l} - 3 - \texttt{a} - 7 - \textit{espace} - 1 - \texttt{l} - 3 - \varepsilon - 1 - \texttt{u} - 4 - \texttt{n} - 6 - \texttt{n} - \texttt{e} - 7.$

5.2 Introduction aux algorithmes à base de mots

Les algorithmes dont l'unité de base de traitement est le mot considèrent le texte comme une suite alternée de mots d'une langue naturelle (que nous appelons mots linguistiques) et de mots séparateurs appelés aussi inter-mots . Ainsi un texte est-il écrit à l'aide de deux ensembles de mots.

5.2.1 Spécificité des algorithmes à base de mots

L'élaboration d'algorithmes à base de mots, bien qu'étant souvent une adaptation de ceux qui existent pour des alphabets de caractères, se heurte à deux problèmes :

- Alors que pour les algorithmes à base de lettres, l'alphabet, l'ensemble des caractères ASCII est connu, pour ceux à base de mots, les ensembles de mots ne sont pas connus *a priori*.

- La taille de ces ensembles de mots peut varier de quelques centaines à plusieurs milliers, voire plusieurs dizaines de milliers. Celle de l'ensemble des mots séparateurs est dans la majorité des cas de beaucoup plus petite taille.

Pour résoudre le premier problème, on pourrait supposer que l'on dispose d'un lexique de la langue dans laquelle le texte est écrit. Ainsi un mot pourrait être représenté par son indice dans ce lexique comme un caractère l'est par son code ASCII. Cependant, un tel lexique ne pouvant être exhaustif, de nombreux mots tels, des noms géographiques, des noms propres, des néologismes, des prénoms pourraient ne pas y figurer. Il serait donc nécessaire d'avoir un traitement différent selon qu'un mot appartient ou non au lexique. De plus, cette approche restreindrait l'utilisation de ce type d'algorithme aux textes écrits dans des langues pour lesquelles de tels lexiques auraient été élaborés. Aussi pour être indépendant de tout lexique extérieur, l'approche la plus couramment employée consiste à joindre à la forme compressée du texte, l'ensemble des mots qui le constitue. Le problème, soulevé par la taille des alphabets mis en jeu, est résolu par la croissance de la capacité en terme de mémoire et de vitesse des ordinateurs, ainsi que par l'élaboration d'algorithmes spécialement conçus pour traiter une masse aussi importante de données.

5.2.2 Algorithmes avec ou sans prélecture

De même que pour les algorithmes de compression à base de caractères, les algorithmes à base de mots peuvent faire appel ou non à une prélecture du texte. Ce qui conduit à deux formes d'algorithmes très différents.

5.2.2.1 Algorithmes avec prélecture

Dans ce type d'algorithmes, on utilise la première lecture du texte traité pour construire les lexiques de mots linguistiques et séparateurs. Tout mot du texte est alors parfaitement défini par son indice dans l'un des deux lexiques construits, de la même manière qu'un caractère est parfaitement défini par son code ASCII. Les mots du texte étant désormais connus, on peut, lors d'une seconde lecture, effectuer la compression proprement dite du texte en appliquant une adaptation des algorithmes à base de caractères à des alphabets de grande taille. Il faut bien entendu joindre au texte ainsi compressé une forme compressée des lexiques de mots qui le constituent.

5.2.2.2 Algorithmes avec une seule lecture

Ici le lexique se construit au fur et à mesure de la lecture. Nous devons donc faire appel à une méthode permettant de traiter les mots inconnus. Ce système appelé échappement, est similaire à celui qui permet de changer de contexte dans l'algorithme PPM[2]. Considérons le mot courant w, si celui-ci a déjà été rencontré dans la partie du texte traitée, il appartient au lexique et peut donc être représenté par son code, sinon on émet un code spécial, appelé code d'échappement (noté *ESC*), suivi de la suite de lettres formant le nouveau mot et on l'ajoute au lexique, et on lui attribue un numéro (le plus souvent son indice dans un lexique construit en mémoire). La suite de lettres formant le nouveau mot est éventuellement codée avec un algorithme de compression sur les caractères.

La compression consiste alors à appliquer un algorithme de type *adaptatif* aux indices des mots.

[2]Voir chapitre 3

5.2.3 Le découpage du texte

Considérons l'ensemble A des caractères ASCII. Nous appellons L le sous-ensemble de A comprenant les caractères alphanumériques, et S son complémentaire. Un mot linguistique est alors défini comme une suite de caractères alphanumériques, apparaissant dans le texte, encadrée de caractères non alphanumériques, (ou en tête et fin de fichier). Un mot séparateur est une suite de caractères non alphanumériques encadrée de caractères alphanumériques. Les lexiques sur lesquels nous travaillons sont donc inclus dans L^* et S^*. Si on appelle l et s des éléments de L^* et S^*, un texte peut être décrit par l'expression rationnelle :

$$T = (l + \varepsilon)(sl)^*(s + \varepsilon)$$

En fait, dans la pratique nous devons fixer une longueur limite de mot linguistique et de mot séparateur. Ainsi il est plus exact de dire que les mots que nous considérons appartiennent à $L^{\leq k}$ et $S^{\leq r}$ où k et r sont les longueurs maximales permises pour les mots linguistiques et pour les mots séparateurs. On ajoute le mot vide aux deux ensembles. Si un mot linguistique ou séparateur plus long est rencontré, il est décomposé en plusieurs mots linguistiques ou séparateurs, séparés par des mots vides séparateurs ou linguistiques.

On notera ε_l le mot vide ajouté à l'ensemble L et ε_s celui ajouté à l'ensemble S.

Ainsi si $k = 5$ le mot : « compression »

est décomposé en la suite de mots : compr ε_s essio ε_s n

Plus généralement, un texte T peut être découpé sous la forme :

$$T = ((l + \varepsilon_l)(s + \varepsilon_s))^*$$

Ceci permet de traiter le cas des mots très longs mais aussi de considérer que tout texte commence par un mot linguistique (éventuellement vide).

5.3 Représentation et compression de lexiques

Nous avons vu que les algorithmes avec prélecture (5.2.2.1) font appel à la construction de lexiques. Nous nous intéressons maintenant aux façons de les représenter et de les compresser.

Définissons ce que nous appelons lexique et dictionnaire.

- Un lexique est une suite de mots triée selon un ordre préalablement défini sur les mots. Cet ordre est lui-même défini à partir d'un ordre défini sur les lettres.

- Un dictionnaire est un lexique dans lequel chaque entrée peut être munie d'informations, *a priori*, indépendantes de celles des autres entrées.

Considérons, par exemple, la suite de mots :

a, *air*, *avis*, a*visé*, *avocat*, *avouer*, *bal*, *bien*, *biere*, *brise*, *roue*, *route*, *rue*,
et les informations lexicales associées.

Nous avons :

Dictionnaire	
Lexique	Informations
a	V :pre3s
air	N :ms
avis	N :ms
avisé	PP
avocat	N :ms
avouer	V :inf
bal	N :ms
bien	N :fs
bière	N :fs
brise	N :fs
roue	N :fs
route	N :fs
rue	N :fs

Tab. 5.1 – Dictionnaires et Lexiques

On cherche une représentation du lexique peu coûteuse en place.

5.3.1 Représentation par liste

5.3.1.1 Utilisation d'un séparateur

La façon la plus simple de fournir un lexique est de donner la liste des mots le constituant en marquant la fin de chaque mot à l'aide d'un séparateur (par exemple :).

Une variante consiste à ne pas utiliser de séparateur et à faire précéder chaque mot du nombre N_l de lettres qui le composent ($0 \leq N_l \leq 256$).

Mot
a :
air :
avis :
avisé :
avocat :
avouer :
bal :
bien :
bière :
brise :
roue :
route :
rue :

$\longrightarrow$

N_l	Mot
1	*a*
3	*air*
4	*avis*
5	*avisé*
6	*avocat*
6	*avouer*
3	*bal*
4	*bien*
5	*bière*
5	*brise*
4	*roue*
5	*route*
3	*rue*

TAB. 5.2 – Lexique : représentation par liste

5.3.1.2 La méthode du *Front-Coding*

Une technique fondée sur le *codage différentiel* permet d'obtenir une représentation plus compacte. Le *codage différentiel* consiste à coder une suite de symboles par la *différence* de deux symboles consécutifs. Ainsi codera-t-on, par exemple, une suite de dates par le nombre de jours qui les sépare, les pixels voisins d'une image par leur différence. Dans le cas de lexiques, le codage différentiel est fondé sur le fait que de nombreux mots partagent les mêmes préfixes . Cette technique de codage appelée alors *front-coding* consiste à représenter tout mot du lexique par la longueur L_p du préfixe qu'il a en commun avec son prédécesseur, la longueur L_s du suffixe restant, et ce suffixe lui-même.

Si nous appliquons le *front-coding* à notre lexique, nous obtenons le tableau suivant.

Mot
a :
air :
avis :
avisé :
avocat :
avouer :
bal :
bien :
bière :
brise :
roue :
route :
rue :

$\longrightarrow$

L_p	L_s	$Suffixe$
0	1	*a*
1	2	*ir*
1	3	*vis*
4	1	*é*
2	4	*ocat*
3	3	*uer*
0	3	*bal*
1	3	*ien*
2	3	*ère*
1	4	*rise*
0	4	*roue*
3	2	*te*
1	2	*ue*

TAB. 5.3 – Lexique « Front-codé »

5.3.2 Représentation par arbre

La représentation par arbre tire parti, comme la technique de *front coding*, de l'abondance de mots partageant les mêmes préfixes. Les dictionnaires sont représentés sous la forme d'un automate en arbre multibranche où chaque nœud comporte une lettre avec l'indication éventuelle que ce nœud est terminal. Tout nœud feuille étant terminal, seuls les nœuds terminaux internes sont indiqués.

L'arbre qui représente notre liste de mots est le suivant :

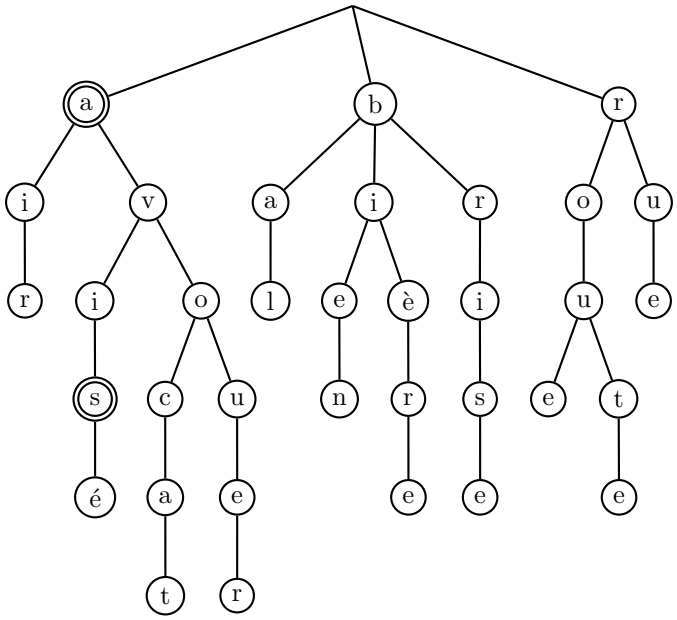


FIG. 5.2 – Lexique Arbre

Cet arbre est alors « mis à plat », grâce à un parcours préfixe, sous une forme compacte dans laquelle un signe + indique que la lettre qui le précède est suivie de plusieurs suffixes, un signe - indique la fin d'un suffixe et qu'il faut remonter dans l'arbre au dernier « embranchement ». Le symbole | indique que l'état atteint par la lettre qui le précède est terminal.

Ainsi pour l'arbre de la figure 5.2 nous obtenons :

+a|+ir-v+is|é-o+cat-uer----b+al-i+en-ère--rise--r+ou+e-te--ue--

Cette forme peut être quelque peu simplifiée. En effet, en ordonnant alphabétiquement dans l'arbre de la figure 5.2 les nœuds frères, les fils de la racine sont dans l'ordre `a`, `b`, `r`, de même les fils de `a` sont `i`, `v`. Ceci se retrouve, bien sûr, sur la forme compacte. Cette remarque permet de supprimer quelques signes `-`.

En effet, considérons le signe `-` qui suit le suffixe commençant par le caractère c_x d'un mot. Celui-ci indique qu'il faut remonter au nœud père de celui qui porte le caractère c_x. Si ce nœud père porte un caractère c_y inférieur ou égal au caractère c_x, le signe `-` peut être supprimé, car si c_y était porté par un nœud frère de celui portant c_x, nous aurions par construction de l'arbre : $c_y > c_x$.

Considérons le `b` de `bal`, il est précédé de 4 signes `-`.

Par construction de l'arbre :

- `b` ne peut être porté par un nœud frère de `u` (`b` < `u`).
- de même `b` ne peut être porté par un nœud frère de `o` (`b` < `o`).
- Enfin `b` ne peut être porté par un nœud frère de `v` (`b` < `v`).

Ces trois remarques nous permettent de supprimer trois signes `-`. En appliquant ce principe sur la totalité de la forme compacte, elle devient :

```
+a|+ir-v+is|é-o+cat-uer-b+al-i+en-ère-rise-r+ou+e-te--ue--        (2)
```

Le gain expérimental obtenu sur le taux de compression des lexiques par passage de la forme (1) à la forme (2) est de l'ordre de 1%.

5.3.3 Représentation par automate acyclique

Cette représentation permet de mettre en commun non seulement des préfixes de mots, comme le font les arbres, mais aussi des suffixes et plus généralement, des facteurs. Cette représentation a été largement utilisée dans le domaine de l'informatique linguistique, tant au plan lexical que syntaxique, ainsi que pour représenter les dictionnaires électroniques de plusieurs langues européennes.

L'automate acyclique correspondant à la liste de mots du tableau 5.1 est :

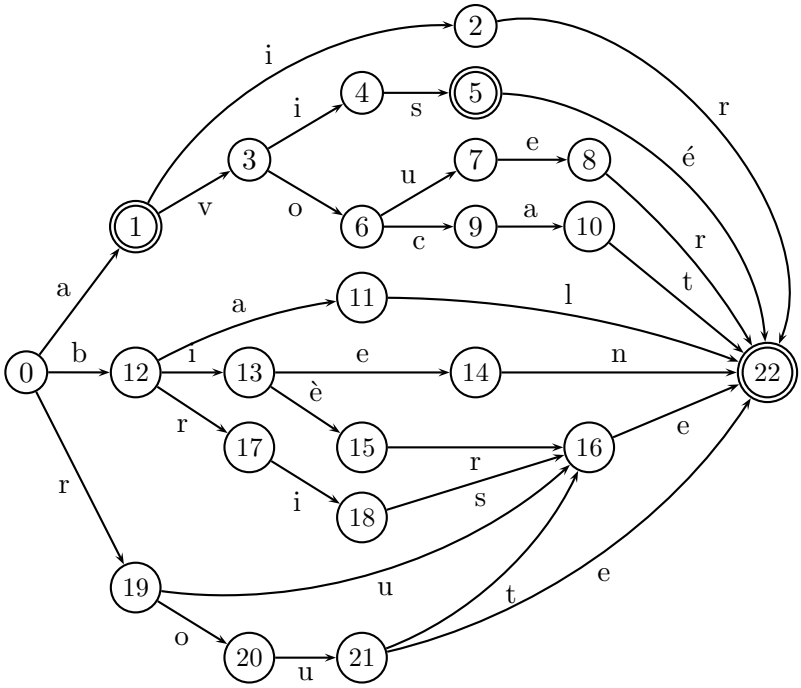


FIG. 5.3 – Représentation d'un dictionnaire

5.3.4 Compression des lexiques

La compression des lexiques va donc consister à effectuer la compression de l'une des représentations de lexiques précédemment décrites. Les représentations les plus compactes sont celles par arbres et par automates. Ces représentations des lexiques doivent être compressées par des algorithmes spécialement adaptés à chacunes d'elles.

Pour la représentation par arbre, nous compressons la forme compacte de l'arbre par l'algorithme PPM à l'ordre 2.

Pour la représentation par automate, nous considérons la liste des transitions telle qu'elle est représentée dans le tableau 5.4. La liste des transitions étant ordonnée par numéro d'état, il suffit d'indiquer pour chaque état le nombre de transitions qu'il comporte, suivi par la liste de ses transitions qui sont des couples de la forme (*lettre*, *numéro d'état*). La place des données dans la liste des états étant clairement définie, nous pouvons compresser cette liste en utilisant trois arbres de Huffman dont les alphabets sont les entiers et les caractères : un pour le nombre de transitions, un pour les lettres, un pour les numéros d'états.

L'automate de la figure 5.3 est représenté par la liste de transitions de la table 5.4.

0 :	a→1	4 :	r→5	11 :	l→22	16 :	e→22
	b→12	5 :	e→22	12 :	a→11	17 :	e→22
	r→19	6 :	u→7		i→13	18 :	s→16
1 :	i→2		c→9		r→17	19 :	u→16
	v→3	7 :	e→8	13 :	e→14		o→20
2 :	r→22	8 :	r→11		e→15	20 :	u→21
3 :	i→4	9 :	a→10	14 :	n→22	21 :	t→16
	o→6	10 :	t→22	15 :	r→16		e→22

TAB. 5.4 – Liste des transitions

5.4 Résultats expérimentaux

Pour la représentation par arbre, nous donnons dans le premier tableau pour chaque lexique :

- La taille du lexique sous forme de liste lexicographiquement triée.
- La taille de cette liste compressée par l'algorithme PPM à l'ordre 3 ou 4 (le meilleur des deux ordres est cité).
- La taille de la forme compacte.
- La taille de la forme compacte simplifiée.
- La taille de la forme compacte compressée par PPM à l'ordre 2.
- La taille de la forme compacte simplifiée compressée par PPM à l'ordre 2.

En dessous de chaque taille est indiqué le taux de compression obtenu par rapport au lexique sous forme de liste. Pour la forme compacte simplifiée compressée, nous donnons aussi le nombre moyen de bits par mot (nbb/m).

Pour la représentation par automates, nous indiquons dans un tableau, pour chaque lexique :

- Le nombre d'états.
- Le nombre de transitions de l'automate.

De plus, ces transitions ont été compressées en effectuant un Huffman sur les lettres et un Huffman sur les numéros d'états. Les résultats sont indiqués respectivement sur les lignes *lettres* et *états* du second tableau.

Représentation par Arbres

Lexique	asyoulik	alice29	lcet10	plrabn12	world192	K.J.Bible
taille	20695	23304	55378	76098	175274	101284
nombre de mots	2781	3215	6245	9364	22189	12483
liste compressée	8785	9960	21380	29665	50927	79660
	42,4%	42,7%	38,6%	38,9%	45,4%	39,3%
forme compacte	13233	14853	32190	44259	104641	55790
	63,9%	63,7%	58,1%	58,1%	59,7%	55,0%
forme compacte	12684	14122	30680	41892	99783	52832
simplifiée	61,2%	60,5%	55,4%	55,1%	56,9%	52,1%
forme compacte	5536	6274	13114	16858	46513	20891
compressée	26,7%	26,9%	23,6%	22,1%	26,5%	20,6%
forme compacte	5360	6052	12671	16257	44924	20117
simplifiée compressée	**25,8%**	**22,8%**	**22,5%**	**21,3%**	**26,5%**	**19,8%**
nb bits/mot	15,42	15,06	16,23	13,89	16,20	12,89

Représentation par Automates

Lexique	asyoulik	alice29	lcet10	plrabn12	world192	K.J.Bible
taille	20695	23304	55378	76098	175274	101284
nombre de mots	2781	3215	6245	9364	22189	12483
nombre d'états	2738	2867	7044	7005	19784	8444
nombre transitons	4706	5274	11402	13522	36723	16565
Huffman statique	11578	12652	30558	33834	99717	41350
taux	55,9%	54,3%	55,1%	44,4%	56,9%	40,8%

Il est clair que la compression par arbre est bien meilleure. Ce type de compression étant faite pour sauvegarder les lexiques sous la forme d'une donnée « non parcourable », il n'est pas surprenant qu'elle soit meilleure que la compression par automate qui, elle, permet la reconnaissance de mots. Ceci s'explique par le fait que le langage L qui structure la forme compacte de l'arbre est très réduit : $L = \{(,|,)\}$ alors que la structure d'automate utilise de nombreux numéros d'états.

Le passage par la forme compacte de l'arbre permet d'obtenir des gains de compression très importants par rapport au lexique sous forme de liste. Pour étudier plus amplement l'efficacité de ce processus de compression de lexiques, nous l'avons testé sur des lexiques comportant de très nombreux mots. Ces lexiques sont ceux contenant les formes fléchies du français, de l'anglais et de l'italien.

Représentation par Arbres

Lexique	Français	Anglais	Italien
taille	6721958	1305270	6559083
nombre de mots	575994	129704	561189
forme compacte simplifiée	1982951	485194	2010857
taux	29,5%	37,2%	30,6%
forme compacte simplifiée compressée	172653	120842	176799
taux	**2,57%**	**9,26%**	**2,70%**
nombre de bit par mot	2,40	7,45	2,52

Représentation par Automates

Lexique	Français	Anglais	Italien
taille	6721958	1305270	6559083
nombre de mots	575994	129704	561189
nombre d'états	59065	52519	57584
nombre de transition	176500	137180	174458
Huffman	603429	360266	437486
taux	8,9%	27,6%	6,6%

Nous observons que les forts taux de compression obtenus sur des lexiques de textes se vérifient et s'amplifient sur les lexiques très longs des formes fléchies de langues naturelles. Précisons que ces très gros lexiques ont été compressés par l'algorithme PPM à l'ordre 4 pour l'anglais, et à l'ordre 7 pour l'italien et le français. Les fichiers obtenus par compression des automates sont de l'ordre du triple de ceux obtenus par compression des arbres.

Chapitre 6

Compression à base de mots avec LZW

L'algorithme LZW que nous avons présenté au chapitre 2, a été adapté à un découpage par mots par Cormak et Horspool en 1992 [16]. Dans ce chapitre, après avoir exposé l'algorithme de Cormak et Horspool, nous proposons quelques améliorations qui font appel à une meilleure compression des mots formant les lexiques du texte traité. L'algorithme comportant l'ensemble de ces améliorations permet d'obtenir des taux de compression sensiblement meilleurs que ceux obtenus avec l'algorithme original.

6.1 L'algorithme de Cormack et Horspool.

Dans le but de garder à l'algorithme LZW la rapidité de la version à base de caractères, Cormack et Horspool proposent un algorithme ne nécessitant qu'une seule lecture du texte traité. Cependant le fait de travailler avec deux lexiques de mots (linguistiques et séparateurs) *a priori* inconnus rend impossible une initialisation préalable des tables de Ziv et Lempel équivalente à celle pratiquée avec l'ensemble des codes ASCII dans l'algorithme original à base de caractères.

Cormack et Horspool utilisent un mécanisme d'échappement (évoqué en 5.2.2.2) semblable à celui utilisé dans l'algorithme PPM[1]. Lors de l'occurrence d'un nouveau mot, un code d'échappement est émis, suivi par la suite des caractères formant le mot traité ainsi qu'un caractère spécial qui indique la fin du mot. Dans le but d'obtenir un meilleur taux de compression, la suite des caractères formant le mot est elle-même compressée par un LZW à base de caractères.

Le fait qu'un texte en langue naturelle soit considéré comme une suite alternée de mots linguistiques et de mots séparateurs permet à Cormack et Horspool de considérer deux tables, l'une (notée T_L) contenant les facteurs commençant par un mot linguistique, l'autre (notée T_S) contenant les facteurs commençant par un mot séparateur.

Si l et s représentent un mot linguistique ou un mot séparateur quelconque, ε_l et ε_s les mots vides linguistique et séparateur, les tables contiennent les facteurs de la forme :

$$T_L \ : (l+\varepsilon_l)(s+\varepsilon_s)^*((l+\varepsilon_l)+\varepsilon)$$

$$T_S \ : (s+\varepsilon_s)(l+\varepsilon_l)^*((s+\varepsilon_s)+\varepsilon)$$

Le découpage du texte ainsi obtenu n'est pas ambigu. Lors de la décompression du texte, il suffit de regarder par quel type de mot (linguistique ou séparateur) le facteur courant traité se termine pour savoir dans quelle table interpréter le facteur suivant. Le fait que deux tables soient utilisées permet de faire augmenter lentement la valeur des indices. Des indices identiques peuvent faire référence à des facteurs différents des deux tables et améliorer ainsi le taux de compression du texte.

[1] Voir chapitre 3

Nous donnons ci-dessous, l'algorithme original de Cormack et Horspool.

Soit m un mot linguistique ou séparateur et ε le mot vide.

1. $W = \varepsilon$
2. **Répeter**
3. lire(m)
4. **Si** nouveau(**m**)
5. Si $W \neq \varepsilon$ Écrire($indice(W)$)
6. Écrire(ESC)
7. Écrire(la suite compressée des lettres de m)
8. Écrire(Fin-de-mot)
9. $W = \varepsilon$
10. **Fin du si**
11. **Sinon**
12. **Si** ($W[type]$.**m** $\in$ table[$Début$]) $W[Début] = W[Début].m$
13. **Sinon**
14. Écrire($indice(W)$)
15. Ajouter($W, table[Début]$)
16. $Début\_linguistique = type$
17. $W = m$
18. **Fin du sinon**
19. **Fin du sinon**
20. $type$ = not $type$
21. **Jusqu'a Fin-de-Fichier**

FIG. 6.1 – Algorithme de Cormack et Horspool

6.2 LZW en « parallèlle »

Dans l'algorithme original les facteurs considérés sont des alternances de mots linguistiques et séparateurs. Nous proposons de considérer d'une part les facteurs uniquement constitués de mots linguistiques, d'autre part ceux constitués uniquement de mots séparateurs. L'algorithme que nous proposons consiste donc à compresser « en paralèlle » les mots linguistiques et les mots séparateurs. Le contenu des tables est donc de la forme :

$$T_L \ : (l + \varepsilon_l)^*$$
$$T_S \ : (s + \varepsilon_s)^*$$

Ceci implique évidemment qu'il faut réalterner correctement les mots lors de la décompression en utilisant un traitement par file d'attente car les facteurs de mots séparateurs sont souvent plus longs que ceux de mots linguistiques.

Nous donnons ci-dessous la forme de notre algorithme qui s'applique « en paralèlle » sur les mots linguistiques et les mots séparateurs. La variable *type* prend alternativement la valeur **linguistique** ou **séparateur**.

Soit m un mot linguistique ou séparateur et ε le mot vide.

1. **Si** nouveau(m)
2. **Si** $W[type] \neq \varepsilon$ Écrire($indice(table[type]), W[type]$))
3. Écrire(**ESC**)
4. Écrire(la suite compressée des lettres de m)
5. Écrire(Fin_de_Mot)
6. Ajouter($table[type], m$)
7. $W[type] = \varepsilon$
8. **Fin du si**
9. **Sinon**
10. **Si** ($W[type].m \in table[type]$) $W[type] = W[type].m$
11. **Sinon**
12. Ajouter-table($W[type].m$)
13. Écrire($indice(table, [type]), W[type]$)
14. $W[type] = m$
15. **Fin du sinon**
16. **Fin du sinon**

FIG. 6.2 – LZW-Mot en parallèlle

Cette version de LZW en parallèlle permet d'obtenir des taux de compression légèrement supérieurs à la version originale (« entrelacée ») de l'algorithme. Nous proposons maintenant deux améliorations supplémentaires qui portent sur la compression des mots et le codage des indices.

6.3 Une meilleure compression des mots

La deuxième amélioration de l'algorithme C.H. que nous proposons concerne le codage des mots lors de la rencontre de leur première occurrence dans le texte. C'est particulièrement important car le nombre de mots d'un texte est de l'ordre de quelques milliers. Nous avons vu que chaque mot nouveau est aussi compressé par LZW, fonctionnant alors sur les lettres du mot. Ceci veut dire que, en quelque sorte, chaque mot est considéré comme un fichier de lettres et donc que chaque fin de mot est considérée comme une fin de fichier. Cette fin de mot est très coûteuse à représenter car elle compte pour un facteur et est donc codée comme tout autre facteur sur au moins 9 bits, puis 10, 11, etc... Ainsi dans l'algorithme de Cormack et Horspool, si k est le nombre de facteurs d'un mot, chaque mot nouveau nécessite l'envoi de la séquence de codes suivante :

$$ESC code(fact_1).....code(fact_k)Fin\_de\_mot$$

Nous pouvons, grâce à une bufferisation, compter le nombre de facteurs du mot traité. Ce nombre qui est borné du fait que la longueur des mots est limitée, pourrait être écrit avec un petit nombre de bits après le code ESC. Chaque mot nouveau donnerait donc lieu à la suite de codes :

$$ESC\mathbf{k}code(fact_1).....code(fact_k)$$

En fait nous proposons d'intégrer **k** au code d'échappement en créant autant de codes d'échappements qu'il y a de longueurs de facteur possibles. Ainsi la séquence émise pour chaque mot nouveau est-elle :

$$ESC_{\mathbf{k}}code(fact_1).....code(fact_k)$$

Le coût de l'ajout d'un code d'échappement est très faible. On économise donc la longueur du code de fin de mot par mot nouveau. Si A et P sont respectivement les lexiques de mots linguistiques et de mots séparateurs, le gain obtenu est au minimum de $9 * (|A| + |P|)$ bits.

6.4 Codage *Phased Binary*

Enfin nous proposons une troisième amélioration qui consiste à changer le codage des facteurs. Au lieu de coder les facteurs au moyen de $l = \lceil \log_2(N) \rceil$ bits, N étant le premier indice libre dans la table, on peut utiliser une autre méthode, appelée *Phased binary*, qui permet de coder éventuellement les facteurs au moyen de $l - 1$ bits. Elle repose sur l'observation suivante ; si N n'est pas une puissance entière de 2, $2^{l-1} < N < 2^l$ au lieu de n'utiliser que les codes inférieurs à N et de coder tout nombre au moyen de l bits, on utilise tous les codes inférieurs à 2^l et l'on peut ainsi coder certains nombres au moyen de $l-1$ bits. Cette méthode a été utilisée dans une version de LZW à base de lettres connue sous le nom de LZT [5].

Elle est définie de la manière suivante :

Si i est l'indice à coder, soient $l = \lceil \log_2(N) \rceil$ et $f(i)$ le code de i alors :

$$f(i) = \begin{cases} i & \text{écrit en binaire avec } l-1 \text{ bits si } i < 2^l - N \\ i + 2^l - N & \text{écrit en binaire avec } l \text{ bits sinon} \end{cases}$$

Voici un exemple de ce codage pour $N = 10$ (et donc $l = 4$).

Entier	Code	Entier	Code
0	000	6	1100
1	001	7	1101
2	010	8	1110
3	011	9	1111
4	100		
5	101		

Pour décoder un nombre écrit avec ce type de code, on commence par lire $l-1$ bits, puis on compare le nombre x obtenu à $2^l - N$. Si x est supérieur ou égal à $2^l - N$, on lit un bit supplémentaire et on effectue le calcul $x = x - (2^l - N)$, sinon x est la valeur codée.

Avec le code du tableau précédent, si on lit la séquence de bits 111, elle est égale à 7 qui supérieur à 6 ($6 = (2^4 - 10)$), on lit donc un bit supplémentaire. Admettons que ce soit un 0, nous obtenons la séquence 1110, soit la valeur 14 à laquelle on soustrait 6. Ce qui donne bien $x = 8$.

6.5 Résultats expérimentaux

Dans les résultats qui suivent, CH0 désigne la version originale de l'algorithme de Cormack et Horspool, CH1 désigne la version « en parallèle » de l'algorithme de Cormack et Horspool , CH2 correspond à CH1 avec l'amélioration concernant la compression des mots. Enfin CH3 reprend CH2 en y remplaçant le codage usuel par le code *Phased Binary*.

Résultats

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.bible
Taille	125179	152089	426754	481861	2473400	4047392
gzip-9	48829	54191	144429	194277	721413	1176649
	39,0%	35,6%	33,8%	40,3%	29,1%	29,0%
bzip2	39569	43202	107706	145577	489583	845623
	31,7%	**28,4%**	**26,2%**	**31,2%**	**19,8%**	**21,9%**
PPM_1	54775	66402	188431	204133	1129991	1645129
	43,8%	**43,7%**	**44,2%**	**42,4%**	**45,7%**	**40,7%**
PPM_2^e	43607	51318	146181	169004	870949	1230561
	34,8%	33,7%	34,2%	35,0%	35,2%	30,4%
PPM_3^e	39659	43966	426754	145727	653090	4047392
	31,6%	28,9%	27,2%	30,2%	26,4%	24,0%
PPM_4^e	40117	43379	107476	141605	525957	854247
	32,0%	**28,5%**	**25,1%**	**29,3%**	21,2%	21,1%
PPM_5^e	42148	44970	109595	147684	486607	818274
	33,6%	29,5%	25,6%	30,6%	**19,7%**	**20,2%**
CH0	56293	58091	146106	205571	681177	1020738
	44,9%	38,1%	34,2%	42,6%	27,5%	25,2%
CH1	55745	57703	141264	203974	659071	1032547
	44,5%	37,9%	33,1%	42,3%	26,6%	25,5
CH2	50135	52627	129334	185334	615237	1009467
	40,0%	34,6%	30,3%	38,4%	24,8%	24,9%
CH3	47628	50090	123349	176318	591106	972893
	38,0%	**32,9%**	**28,9%**	**36,5%**	**23,8%**	**24,0%**

Dans ce tableau, on donne, pour chaque texte, la taille du texte compressé et le taux de compression obtenu.

6.6 Conclusion

Les résultats expérimentaux montrent que les améliorations de l'algorithme original de Cormack et Horspool que nous proposons, permettent d'augmenter sensiblement les taux de compression des textes. Le passage du traitement « entrelacé » au traitement en « parallèlle » permet d'obtenir une légère amélioration, de 0,45%, du taux de compression moyen. En revanche, l'augmentation du taux de compression est, en moyenne, de 4,3% entre CH1 et CH3 ; le passage du codage binaire strict au code *Phased Binary* représentant quant à lui un gain moyen de 1,3%. Ceci montre que l'essentiel de l'amélioration du taux de compression est dû à la meilleure compression des mots. Cette meilleure compression des mots apparaît de manière plus ou moins évidente dans le taux de compression du texte (qui est un taux global sur l'ensemble du texte) selon que le lexique du texte est important ou qu'il occupe une part importante de la taille du texte. On peut remarquer, par exemple, que la différence de taux de compression pour le texte `world192` est de 1,8% entre CH1 et CH2 alors qu'elle n'est que de 0.6 % pour `K.J.Bible`. Ceci s'explique par le fait que le lexique de `world192` comporte un nombre de mots de l'ordre du double de celui de `K.J.Bible` pour une taille de texte presque deux fois moindre.

Chapitre 7

Codage arithmétique sur les mots

Durant les années 70 et le début des années 80, le codage arithmétique était largement vu comme une curiosité académique, *a fortiori* celui à base de mots. Les premiers travaux effectués à propos du codage arithmétique à base de mots sont dus à Alistair Moffat [41] en 1987. Les textes électroniques disponibles à cette époque et traités dans ces travaux étaient souvent de type informatique et comportaient un nombre de mots peu élevé. Les textes en langues naturelles, de plus en plus nombreux et largement diffusés depuis quelques années notamment grâce à internet, ont une taille et un vocabulaire importants qui requièrent des outils de traitement adaptés. En 1994, Edward Fenwick [20] a proposé une nouvelle implémentation du calcul des fréquences cumulées. Son travail a été repris dans une nouvelle implémentation du codage arithmétique par A.Moffat, R.Neal, et I.H. Witten [43] en 1995 et connue sous le nom de DCC95. Dans ce chapitre, nous exposons cette implémentation et montrons qu'une amélioration de la compression des mots permet d'obtenir une amélioration du taux de compression des textes.

7.1 Les spécificités du codage à base de mots

La première implémentation du codage arithmétique proposée en 1987 par Witten, Neal et Cleary [70], largement diffusée et connue sous le nom de CACM package, était destinée à la compression à base de caractères. En effet, la taille de l'alphabet était limitée et fixe. De plus la fréquence maximale d'un symbole ne pouvait dépasser la valeur de 32768. Ces restrictions ont conduit Moffat, Neal, et Witten à construire un environnement répondant aux exigences suivantes :

- L'utilisation d'une structure de données permettant de traiter des ensembles de symboles de taille importante.
- L'utilisation de plusieurs contextes.
- L'existence d'un système d'échappement pour changer de contexte.
- L'utilisation de structures de données dynamiques, la taille des ensembles de symboles n'étant pas a priori bornée.

L'observation de ces exigences leur a permis d'obtenir un environnement performant et facilement adaptable et modifiable.

7.2 Une implémentation en trois modules

Le codage arithmétique se caractérisant par une nette séparation entre le modèle de probabilité et le codeur fait appel habituellement à une implémentation en deux modules *Modèle* et *Codeur*. L'implémentation présentée ici se distingue par l'utilisation de trois modules : *Modèle*, *Statistiques*, *Codeur*.

Le module *Statistiques* gère les structures de données qui mémorisent les fréquences. A partir d'un identificateur de symbole, il renvoie la fréquence cumulée de l'ensemble des symboles présents et effectue l'incrémentation et la mise à jour des fréquences. Ni le modèle, ni le codeur ne connaissent la structure de données utilisée ou la manière dont les fréquences sont maintenues à jour. De plus, le modèle ne connaît nullement la probabilité associée à chaque symbole, de même que le codeur n'a pas connaissance des identificateurs des symboles, ni du nombre des symboles.

Nous présentons dans le tableau suivant les fonctions des modules *Statistiques* et *Codeur* auxquelles le module *Modèle* fait appel.

Module	Codeur	Décodeur
Statistiques	$C \leftarrow création\_de\_contexte()$	$C \leftarrow création\_de\_contexte()$
	$coder(C, s)$	$s \leftarrow décoder(C)$
	$installer\_symbole(C, s)$	$installer\_symbole(C, s)$
Codeur	$initialise\_codeur()$	$initialise\_décode()$
	$arith\_codeur(l, h, t)$	$but \leftarrow décoder\_but(t)$
		$arith\_décoder(l, h, t)$
	$fin\_codeur()$	$fin\_décoder()$

TAB. 7.1 – Module Interface

Le modèle Word utilise six contextes, trois pour les mots linguistiques et trois pour les mots séparateurs. Pour chacun de ces types de mots, il y a un contexte pour les mots (M), un contexte pour les caractères (C) constituant les mots nouveaux et un contexte pour la longueur des mots (L). Lors de l'occurrence d'un mot nouveau, un symbole d'échappement est émis permettant de quitter le contexte M. On émet alors la longueur du mot nouveau dans le contexte L suivi de la suite des caractères qui le constituent dans le contexte C. La figure suivante illustre ce mécanisme.

```
1.  M ← création_de_contexte()
2.  C ← création_de_contexte()
3.  L ← création_de_contexte()
4.  Tant que fichier non vide Faire
5.      m ← mot_suivant().
6.      Si m appartient au lexique coder(M, Indice(m))
7.      Sinon
8.          coder(M, ESC)
9.          coder(L, longueur(m))
10.         Pour chaque caractère c de m coder(C, c)
11.         Ajouter m au lexique
12.     Fin du Sinon
13. Fin du Tant que
14. coder(M, Fin_de_message).
15. fin_codeur().
```

FIG. 7.1 – Modèle Word

7.3 Calcul des fréquences

Si on considère un contexte C dans lequel les symboles sont ordonnés selon un ordre connu et un symbole s, le codage du symbole s est effectué par un appel à $arith\_encode(l_s, h_s, t)$ où l_s, h_s et t sont respectivement les fréquences cumulées des symboles strictement inférieurs à s, des symboles inférieurs ou égaux à s et de la totalité des symboles. Le symbole s a pour fréquence $h_s - l_s$ et est représenté par l'intervalle $[l_s/t, h_s/t]$.

L'implémentation du calcul des fréquences présente dans CACM nécessite un temps de calcul $O(n)$ pour le codage d'un symbole. Une structure de données proposée par Fenwick[20] en 1994 permet de mettre à jour l et h en un temps $O(\log(n))$.

Cette structure utilise un tableau F de taille n qui représente implicitement un arbre. Considérons s tel que $1 \leq s \leq n$. On définit $back(s)$ comme étant l'entier obtenu en soustrayant de s le bit 1 le plus à droite. Par exemple la représentation binaire de 13 est 1101 et $back(13) = 1100_2$ de même, 26 est représenté par 11010 et $back(26) = 11000_2$. De façon similaire, on définit $forward(s) = s + 2^i$ où i est la position du bit 1 le plus à droite. Ainsi $forward(13) = 14$ et $forward(16) = 32$.

Le tableau F est défini par $F[s] = h_s - h_{back(s)}$ avec $h_0 = 0$. Le calcul de h_s et l'incrémentation de la fréquence de s sont effectués par l'algorithme suivant :

1. $i \leftarrow s$ and $h_s \leftarrow 0$.
2. **Tant que** $i <> 0$ Faire
   $h_s \leftarrow h_s + F[i]$.
   $i \leftarrow back(i)$.
3. $i \leftarrow s$.
4. **Tant que** $i \leq n$ Faire
   $F[i] \leftarrow F[i] + incr$.
   $i \leftarrow forward(i)$.
5. Return h_s.

FIG. 7.2 – Calcul de h_s

Par exemple, h_{13} est calculée en effectuant la somme $F[13] + F[12] + F[8]$ c'est-à-dire $(h_{13} - h_{12}) + (h_{12} - h_8) + (h_8 - h_0) = h_{13}$. L'incrémentation de la fréquence est faite par les étapes 3 et 4.

7.4 Modification du codeur

Dans l'implémentation CACM (figure 7.3), l'état interne du codeur est défini par les deux variables entières L et R, représentées sur b bits, où L désigne la borne inférieure de l'intervalle courant et R sa taille, qui prennent leurs valeurs sur $[0,2^{b-1}]$.

initialise_coder() :
    $L \leftarrow 0$ $R \leftarrow 2^{b-1} - 1$.
arith_coder(l, h, t) :
1. $T \leftarrow (R \times l)$ div t.
2. $L \leftarrow L + T$.
3. $R \leftarrow ((R \times h)$ div $t) - T$.
4. **Tant que** $R < 2^{b-2}$ Faire
        Arrondir R et écrire un bit.

FIG. 7.3 – Codage version CACM

Cette implémentation a deux inconvénients. Le premier concerne la lenteur des opérations multiplication et division par rapport à l'addition et la soustraction. Le second inconvénient est celui du dépassement de capacité. Les machines étant encore limitées à une arithmétique sur 32 bits (la taille d'un entier est $w = 32$) pour que les multiplications « $R \times l$ » et « $R \times h$ » il faut avoir $2^w - 1 \geq (2^{b-1} - 1)2^f$ soit $w \geq b - 1 + f$. De plus, pour qu'à l'étape 3 il n'y ait pas de dépassement de capacité et que r ne soit pas nul il faut $R \geq t$ soit $2^{b-2} \geq 2^f$. Nous avons les deux contraintes : $w \geq b - 1 + f$ et $b - 2 \geq f$ qui conduisent à $w \geq 2f + 1$. Pour $w = 32$ nous avons le valeur maximale $f = 15$. Ainsi, dans cette implémentation, la fréquence cumulée d'un symbole ne peut dépasser la valeur 32768.

Moffat, Neal et Witten utilisent l'implémentation présentée figure 7.4 dans laquelle au lieu de calculer « $(R \times l)$ div t » on calcule, « $(R$ div $t) \times l$ ». Ils montrent que le fait d'arrondir le rapport « R div t » à un entier n'induit qu'une très faible détérioration du taux de compression et permet, en outre, d'obtenir des contraintes beaucoup plus faibles sur f. Pour $w = 32$ la valeur de f alors permise est $f = 30$.

initialise_encode(l, h, t) :
1. /* $2^{b-2} \leq R < 2^{b-1}$ */
        $r \leftarrow R$ div t.
2. $L \leftarrow L + r \times l$.
3. **Si** $h < t$ $R \leftarrow R - r \times (h - l)$.
        Sinon $R \leftarrow R - r \times l$.
4. **Tant que** $R < 2^{b-2}$ **Faire**
        Arrondir R et écrire un bit.

FIG. 7.4 – Codage version DCC95

7.5 Compression des mots

Pour améliorer la compression des mots (linguistiques et séparateurs) nous ajoutons une étape intermédiaire qui consiste à utiliser une compression LZW. Chaque mot est traité par LZW en utilisant les facteurs des mots précédemment rencontrés.

En ce qui concerne l'aspect général du programme de compression, le contexte des caractères (C) est remplacé par le contexte facteurs (F). De plus, une partie du gain de compression est obtenue au niveau du contexte L qui ne mémorise plus le nombre de caractères du mot traité, mais celui de ses facteurs. Ces nombres, au fur et à mesure de la compression, tendent à se situer dans un ensemble d'entiers très restreint : $\{1, 2, 3, 4\}$ voire $\{1, 2, 3\}$.

La figure 7.1, donne la version modifiée de l'algorithme Word.

```
1.  M ← création_de_contexte()
2.  F ← création_de_contexte()
3.  L ← création_de_contexte()
4.  Initialiser_codeur().
5.  Tant que fichier non vide
6.      m ← mot_suivant().
7.      Si m appartient au lexique coder(M, Indice(m))
8.      Sinon
9.          coder(M, esc),
10.         nb_facteurs = LZW_code(m)
11.         coder(L, nb_facteurs)
12.         Pour chaque facteur f de m Faire
13.         coder(F, f),
14.         Ajouter m au lexique
15.     Fin du Sinon
16. Fin du Tant que
17. coder(M, Fin_de_message).
18. fin_codeur().
```

FIG. 7.5 – Algorithme Word_{zl}

7.5.1 Une prédiction des mots avec majuscules

L'implémentation DCC95 permet de rajouter facilement un ou plusieurs contextes. Nous proposons de créer un contexte qui permet de traiter la plupart des mots commençant par une lettre majuscule. Le principe consiste à mettre dans le contexte Maj (et non dans M) tout mot précédé par un mot séparateur contenant un point, un point d'interrogation ou un point d'exclamation.

7.6 Résultats expérimentaux

Les algorithmes Word, Word$_{zl}$, Word$_{zl}^{Maj}$, désignent respectivement les versions originales du modèle Word, celle avec traitement des mots par Ziv et Lempel et celle qui inclut en plus un traitement des majuscules.

Résultats

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.Bible
Taille	125179	152089	426754	481861	2473400	4047392
gzip-9	48829	54191	144429	194277	721413	1176649
	39,0%	35,6%	33,8%	40,3%	29,1%	29,0%
bzip2	39569	43202	107706	145577	489583	845623
	31,7%	**28,4%**	**26,2%**	**31,2%**	**19,8%**	**21,9%**
PPM$_2^e$	43607	51318	146181	169004	870949	1230561
	34,8%	33,7%	34,2%	35,0%	35,2%	30,4%
PPM$_3^e$	39659	43966	426754	145727	653090	4047392
	31,6%	28,9%	27,2%	30,2%	26,4%	24,0%
PPM$_4^e$	40117	43379	107476	141605	525957	854247
	32,0%	**28,5%**	**25,1%**	**29,3%**	21,2%	21,1%
PPM$_5^e$	42148	44970	109595	147684	486607	818274
	33,6%	29,5%	25,6%	30,6%	**19,7%**	**20,2%**
CH3	47628	50090	123349	176318	591106	972893
	38,0%	32,9%	28,9%	36,5%	23,8%	24,0%
Word	42495	46502	117086	154968	695193	992871
	34,0%	30,6%	27,4%	32,2%	28,1%	24,5%
Word$_{zl}$	41608	45597	111316	149228	663119	984095
	33,24%	29,98%	26,08%	30,97%	26,81%	24,31%
Nb mots du Lexique	3568	3154	7080	11084	23914	13465
Word$_{zl}^{Maj}$	41321	45268	110377	148960	659859	969349
	33.01%	**29.76%**	**25.86%**	**30.91%**	**26.68%**	**23.95%**
Nb mots du Lexique	3531	3066	6745	11014	23150	13222
Nb mots total	23303	27124	62850	80684	341824	767834
Lexique Maj	146	194	604	320	1599	747
Nb mots traités	959	977	2430	1304	9753	25438

Les performances obtenues par le modèle Word sont nettement meilleures que celles obtenues avec la meilleure version de gzip. Les améliorations contenues dans la version Word$_{zl}^{Maj}$ permettent d'obtenir un gain de compression de l'ordre de 1%. Les différences de tailles de lexiques entre Word$_{zl}$ et Word$_{zl}^{Maj}$ montrent que certains mots sont spécifiques au contexte Maj et permettent de réduire quelque peu la taille du lexique général. Les améliorations de compression entre Word et Word$_{zl}$ portant sur une meilleure compression des mots, les améliorations de taux de compression qui en résultent dépendent de la taille du lexique des textes traités par rapport à la taille des textes eux-mêmes. Par exemple, les textes `plrabn12.txt` et `K.J.Bible` ont des lexiques de taille proche alors que la taille du second est quasiment neuf fois celle du premier. Il en découle une différence de taux de compression entre Word et Word$_{zl}$ de 1,23 % pour `plrabn12.txt` et de 0,19 % pour `K.J.Bible`.

Chapitre 8

Huffman canonique

Dans ce chapitre, nous nous intéressons à l'adaptation de l'algorithme de Huffman à des lexiques de mots. Cette adaptation nous amène à considérer la notion de fréquence normalisée dans le but d'obtenir une meilleure compression du texte. Le code de Huffman obtenu avec les fréquences normalisées est similaire à celui connu sous le nom de code de Huffman canonique. Bien que ce dernier soit issu des travaux de Schwartz et Kallick [56] en 1964 et B.Connel [14] en 1973, il n'a donné lieu à de nouveaux travaux dus notamment à Moffat, Bell, Witten et Turpin [68, 45] que très récemment dans le cadre de la recherche sur le « Text retrieval system ». Après avoir présenté le codage de Huffman canonique à base de mots, nous en proposons une version améliorée, en considérant des groupes de mots consécutifs qui sont traités grâce à l'algorithme de Ziv et Lempel.

8.1 Huffman sur les mots

L'algorithme de Huffman sur les mots s'applique de la même manière que celui sur les lettres. Chaque mot est remplacé par son code de Huffman. Cependant, du fait de la stricte alternance mot linguistique, mot séparateur, on peut utiliser sans ambiguité un même code pour un mot de la langue naturelle et un mot séparateur. Ainsi nous utilisons deux arbres de Huffman, l'un pour les mots linguistiques, l'autre pour les mots séparateurs. L'algorithme de Huffman nécessitant deux lectures du texte fait appel au premier schéma de compression défini en 5.2.2.1. Ainsi au cours d'une première lecture du texte, on construit les dictionnaires des mots linguistiques et séparateurs. On mémorise dans ces dictionnaires la fréquence d'apparition des mots. Dans une seconde phase le texte est relu et chaque mot linguistique est remplacé par le mot du code de Huffman calculé grâce aux fréquences. De manière à pouvoir traiter l'occurrence éventuelle de mots très longs, les deux dictionnaires sont initialisés avec le mot vide. La compression des dictionnaires fait appel à la méthode exposée en 5.3.2, qui s'est révélée très efficace, et qui construit une forme compacte à laquelle on applique l'algorithme PPM.

L'algorithme de codage s'établit ainsi :

- Première lecture

  Construction des deux lexiques.

  Calcul des fréquences des mots (linguistiques et séparateurs).

  Calcul des deux codes de Huffman.

  Compression et sauvegarde des deux lexiques.

  Compression et sauvegarde des deux listes de fréquences.

- Seconde lecture

  Substitution mot/code

8.1.1 Sauvegarde des fréquences

La sauvegarde des codes de Huffman ne peut être faite sous la forme utilisée pour les alphabets de caractères. En effet, cette méthode consistait à sauvegarder l'arbre de Huffman avec ses feuilles, les caractères à représenter. Ici les feuilles portent des mots qui sont, par ailleurs, compressés ensemble sous la forme d'une liste lexicographiquement triée selon la méthode décrite en 5.3.2. De ce fait, au lieu de sauvegarder les arbres de Huffman, il est préférable de sauvegarder les fréquences des mots, ce qui implique une reconstruction des arbres à la décompression.

Du fait de la taille importante des lexiques de mots linguistiques (le nombre de mots peut varier de quelques centaines à plusieurs milliers) l'ensemble des valeurs prises par ces fréquences peut être de taille importante. De plus, certaines de ces valeurs peuvent être très élevées (considérons la fréquence des mots anglais *the*, *in* ou des mots français *de*, *le*, *la*). Ainsi, à la différence des alphabets de caractères, la sauvegarde des fréquences occupe une place non négligeable dans la forme compressée du texte.

Considérons à nouveau la liste de mots utilisée en (5.3.2) en affectant une fréquence à chaque mot, puis appliquons lui l'algorithme d'Huffman :

Mot	Fréquence
a	5
air	3
avis	3
avisé	1
avocat	1
avouer	2
bal	7
bien	4
bière	2
brise	1
roue	5
route	2
rue	4

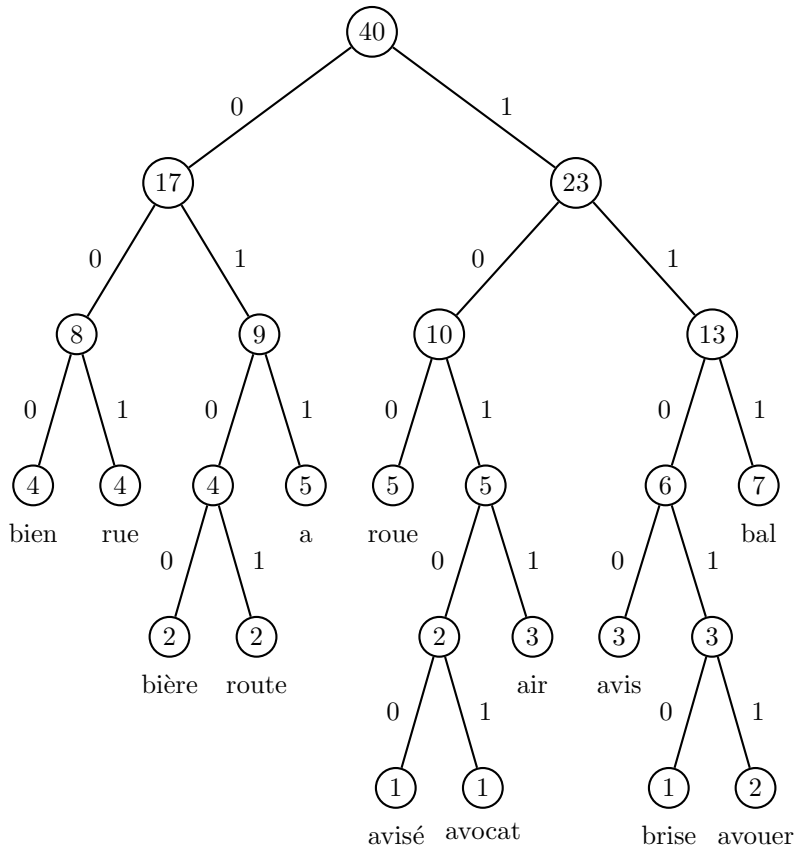
TAB. 8.1 – Liste de mots

Liste

Mot	Fréquence
a	5
air	3
avis	3
avisé	1
avocat	1
avouer	2
bal	7
bien	4
bière	2
brise	1
roue	5
route	2
rue	4

Liste triée

Mot	Fréquence
avisé	1
avocat	1
brise	1
avouer	2
bière	2
route	2
air	3
avis	3
bien	4
rue	4
a	5
roue	5
bal	7



Mot	Code
a	011
air	1011
avis	1100
avisé	10100
avocat	10101
avouer	11011
bal	111
bien	000
bière	0100
brise	11010
roue	100
route	0101
rue	001

FIG. 8.1 – Arbre de Huffman sur les mots

Reprenons l'arbre de Huffman de la figure 8.1 en écrivant à côté de chaque nœud la valeur 2^{h-n} où n est le niveau du nœud (*i.e.* la longueur de l'unique chemin partant de la racine arrivant à ce nœud) et h la hauteur de l'arbre (i.e. la longueur du plus long chemin de la racine à une feuille). L'arbre obtenu est le suivant :

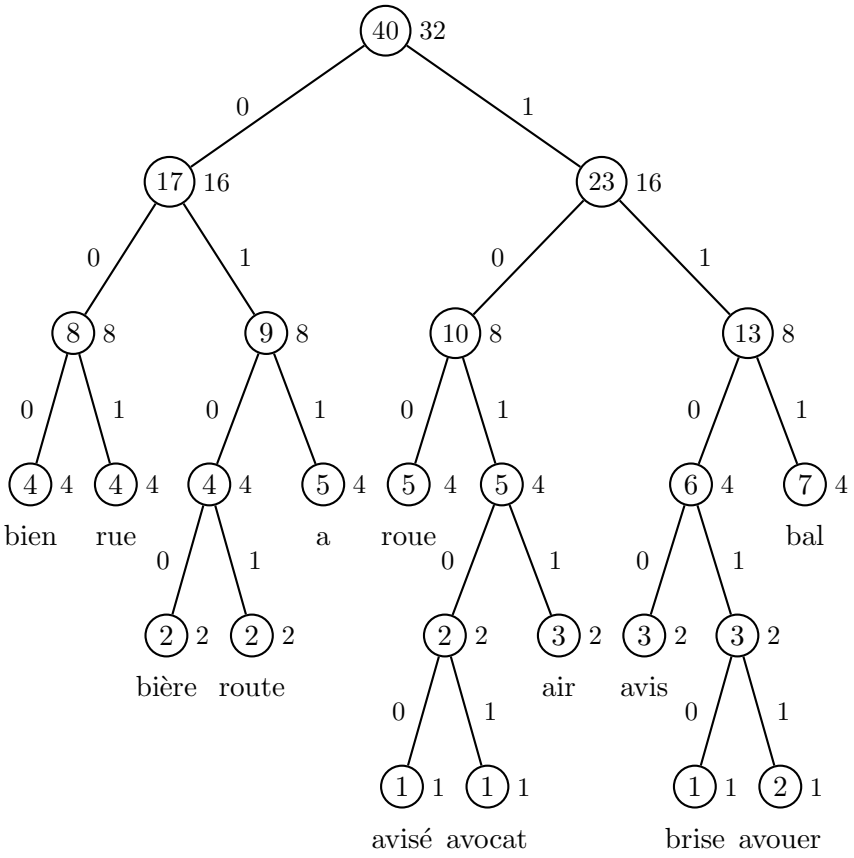


FIG. 8.2 – Fréquences normalisées

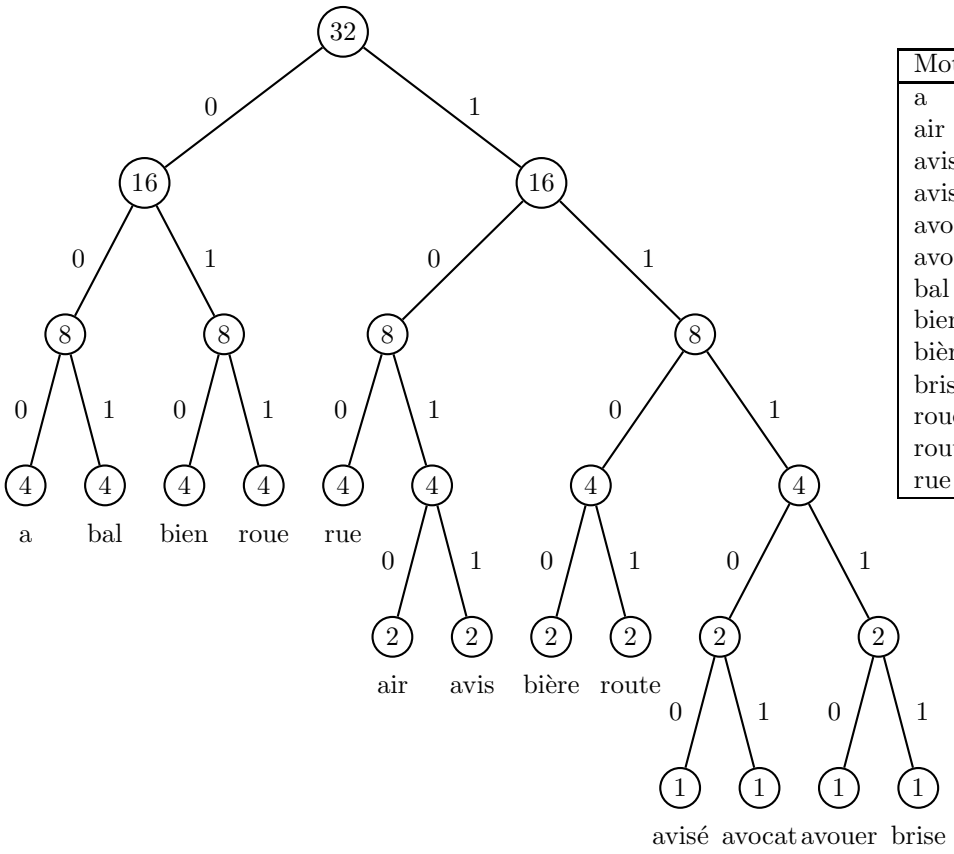
Dans le but de limiter la place mémoire à la transition des fréquences, on associe à chaque mot la fréquence 2^{h-n}. Le code de Huffman obtenu est équivalent (même valeur pondérée) mais sa mémorisation est plus compacte.

Liste

Mot	Fréquence
a	4
air	2
avis	2
avisé	1
avocat	1
avouer	1
bal	4
bien	4
bière	2
brise	1
roue	4
route	2
rue	4

Liste triée

Mot	Fréquence
avisé	1
avocat	1
avouer	1
brise	1
air	2
avis	2
bière	2
route	2
a	4
bal	4
bien	4
roue	4
rue	4



Mot	Code
a	000
air	1010
avis	1011
avisé	11100
avocat	11101
avouer	11111
bal	001
bien	010
bière	1100
brise	11110
roue	011
route	1101
rue	100

FIG. 8.3 – Arbre de Huffman normalisé.

Ce nouvel arbre est aussi efficace que celui de la figure 8.2 pour compresser les données ; $\sum f_i.l_i$ est inchangée. Il est parfaitement défini par la suite 2,1,1,0,0,0,2,2,1,0,2,1,2, qui sont les logarithmes des fréquences normalisées.

L'ensemble des valeurs entières à mémoriser appartiennent à l'intervalle $[0, h-1]$. La valeur de h étant le plus souvent limitée à 32 (la longueur d'un mot mémoire) le passage des *fréquences réelles* aux logarithmes des *fréquences normalisées* réduit très sensiblement la place prise par la sauvegarde de l'arbre de Huffman. Même si le dictionnaire comporte un million de mots, le million de valeurs à mémoriser sont des nombres inférieurs à 32.

Dans notre algorithme, l'ensemble de ces valeurs est compressé, comme les lexiques, par l'algorithme PPM. Insistons sur le fait qu'il faut donc, dans notre algorithme, ne compresser le texte qu'avec l'arbre issu des *fréquences normalisées* et donc construire systématiquement deux arbres.

En fait, on peut se passer de cette reconstruction d'arbre et l'obtenir directement. Si nous considérons l'ensemble des mots du code de la figure 8.3 et que nous les trions selon leurs longueurs, (tableau 8.2) nous remarquons que les mots du code de même longueur représentent des entiers consécutifs. Nous retrouvons ici une propriété du codage de Shannon. En fait, le code de Huffman que nous avons construit est connu sous le nom de code de Huffman Canonique.

Mot	Code
a	000
bal	001
bien	010
roue	011
rue	100
air	1010
avis	1011
bière	1100
route	1101
avisé	11100
avocat	11101
avouer	11110
brise	11111

TAB. 8.2 – Code de Huffman canonique

8.2 Le code de Huffman Canonique

Le code de Huffman canonique a été proposé en 1964 par Bruce Kallick et Eugene S. Schwartz [56] puis J.B.Connel [14] l'a proposé à nouveau en 1973, sous le nom de « Huffman-Shannon-Fano code » et a montré son intérêt en ce qui concerne le temps et l'espace nécessaires à la décompression d'un texte. Enfin, récemment Witten, Moffat et Bell [68] ont utilisé le codage de Huffman canonique pour compresser de grandes bases de données textuelles. Le codage canonique évite de conserver un arbre de Huffman en mémoire et permet d'obtenir des temps de décompression très rapides.

La construction du code canonique procède en deux temps. On commence par calculer les longueurs des mots du code grâce à l'algorithme de Huffman, puis un second algorithme, prenant ces longueurs comme données, calcule effectivement les mots du code. La phase de tri des fréquences peut être faite très rapidement, en un temps $O(n \log(n))$ grâce à l'algorithme Quicksort de Bentley-McllRoy [6] qui tire parti de la présence de paquets de fréquences égales. Le calcul des longueurs des mots du code est faisable sur place, en un temps $O(n)$ à l'aide de l'algorithme de Moffat et Katajainen [44].

Considérons un ensemble M, lexicographiquement trié, de mots linguistiques m_i de fréquence f_i. Trions-les selon leurs fréquences croissantes et appelons r_i leur indice dans cette liste triée. Calculons les longueurs des mots du code grâce à l'algorithme de Huffman et appelons l_{r_i} la longueur du mot du code associé au mot linguistique d'indice r_i. En reprenant l'exemple de la figure 8.1, nous obtenons les tableaux suivants :

i	1	2	3	4	5	6	7	8	9	10	11	12	13
m_i	a	air	avis	avisé	avocat	avouer	bal	bien	bière	brise	roue	route	rue
f_i	5	3	3	1	1	2	7	4	2	1	5	2	4

↓

r_i	1	2	3	4	5	6	7	8	9	10	11	12	13
m_{r_i}	avisé	avocat	brise	avouer	bière	route	air	avis	bien	rue	a	roue	bal
l_{r_i}	5	5	5	5	4	4	4	4	3	3	3	3	3

FIG. 8.4 – Longueurs des mots du code

Le code de Huffman canonique vérifie la propriété dite « séquence additive » du code de Shannon qui signifie que les mots du code de même longueurs représentent des entiers consécutifs. Il suffit donc de pouvoir calculer les entiers correspondant au premier mot du code de chaque longueur, noté $premier(l)$.

On appelle L la longueur du plus long mot du code, $nombre(l)$ le nombre de mots du code de longueur l pour $1 < l < L$, $position(l)$ l'indice du mot linguistique représenté par le premier mot du code de longueur l.

Les valeurs de $premier(l)$ sont calculées par :

$premier(L) = 0$
Pour $l = L - 1$ à 1 **Faire**
$premier(l) = \lceil (nombre(l+1) + premier(l+1))/2 \rceil$

Sur l'exemple de la figure 8.4 nous obtenons :

l	1	2	3	4	5
$nombre(l)$	0	0	5	4	4
$premier(l)$	2	4	3	2	0
$position(l)$	0	0	9	5	1

↓

r_i	1	2	3	4	5	6	7	8	9	10	11	12	13
l_{r_i}	5	5	5	5	4	4	4	4	3	3	3	3	3
b_{r_i}	00000	00001	00010	00011	0010	0011	0100	0101	011	100	101	110	111

FIG. 8.5 – Mots du code

Un code canonique est parfaitement défini par la suite de nombres $nombre(l)$, appelée *index* du code. Pour le tableau précédent, nous avons $index = 0, 0, 5, 4, 4$.

Cette manière de mémoriser le code ne permet plus de conserver les mots linguistiques dans l'ordre lexicographique mais dans celui résultant du tri. Les mots linguistiques étant compressés dans l'ordre lexicographique, la mémorisation du code est faite en conservant la liste des longueurs des mots du code associés aux mots linguistiques.

Les algorithmes de codage et de décodage s'expriment ainsi :

Codage Canonique

1. Dérterminer le plus petit l tel que $position[l] \leq r_i$
2. $code \leftarrow r_i - position[l] + premier[l]$
3. Écrire le $l^{ième}$ bit le moins significatif de *code*.

Décodage Canonique

1. $code \leftarrow$ le bit suivant

   $l \leftarrow 1$
2. **Tant que** $code < premier[l]$ **Faire**
   (a) $code \leftarrow decalage\_gauche(code, 1)$, et insérer le bit suivant
   (b) $l \leftarrow l + 1$
3. $r_i \leftarrow code - premier[l] + pos[l]$

Considérons le codage et le décodage du mot *brise* :

l	1	2	3	4	5
$premier(l)$	2	4	3	2	0
$position(l)$	0	0	9	5	1

r_i	1	2	3	4	5	6	7	8	9	10	11	12	13
m_{r_i}	avisé	avocat	brise	avouer	bière	route	air	avis	bien	rue	a	roue	bal

Le mot *brise* a pour indice 3. Le plus petit l tel que $position[l] \leq 3$ est $l = 5$. La représentation de *brise* est donc 00010, les 5 bits représentant le nombre 3 - 1 + 0 = 2.

Au décodage, les bits sont lus un à un jusqu'à $2 > premier[5]$, le rang du symbole recherché est 2 - 0 + 1 = 3. Le mot *brise* est retrouvé.

Aucun des mots du code de Huffman canonique n'a besoin d'être mémorisé. Les seules données utilisées pour le calcul ou la reconnaissance d'un mot du code sont les tableaux *premier* et *position*.

Le codage de Huffman Canonique, surtout dans le cadre de son utilisation pour le traitement de grande bases de données textuelles, permet un décodage très rapide grâce à des algorithmes dus à A. Moffat et A. Turpin [45] et Shmuel T. Klein [34].

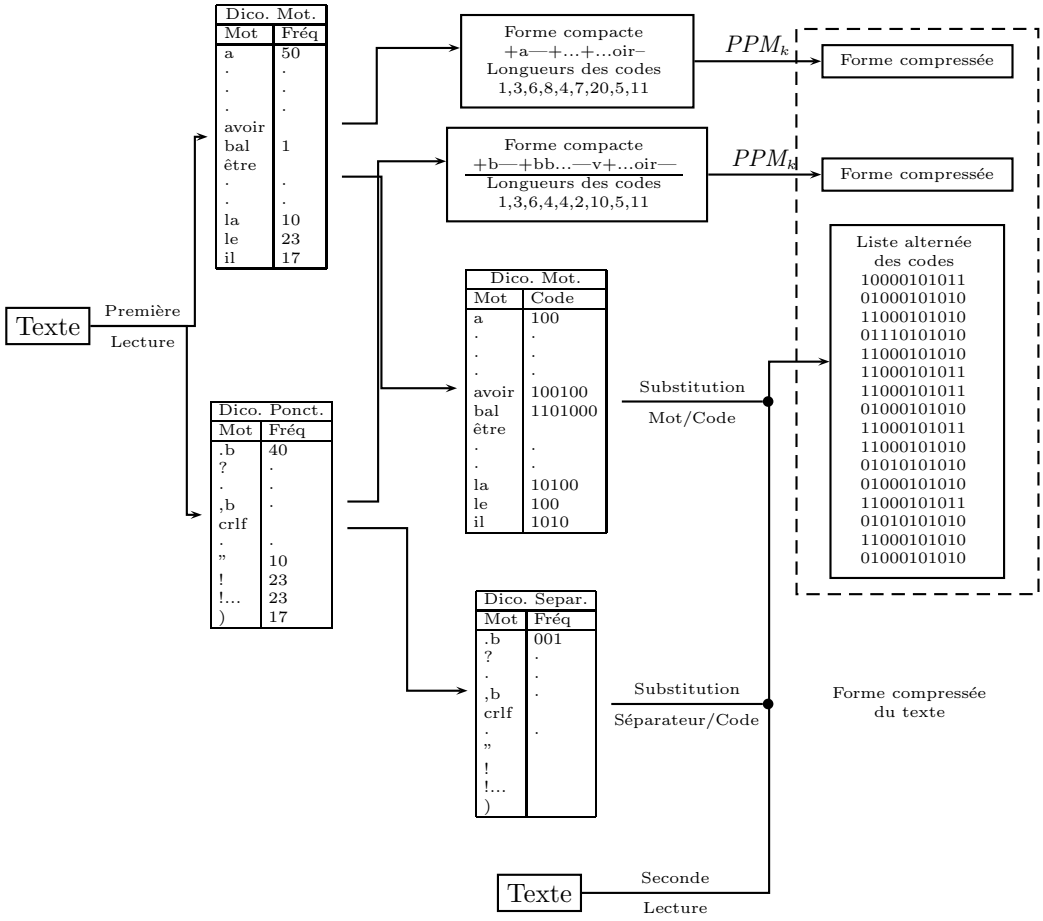


FIG. 8.6 – Compression de Huffman avec dictionnaire

La figure permet de visualiser le traitement nécessaire à la compression d'un texte dont les étapes sont rappelées ci-dessous.

- Première lecture

  Construction des lexiques et calcul des fréquences des mots.

  Calcul des deux codes de Huffman.

  Compression de la forme compacte des deux lexiques par PPM_k.

  Compression des listes de longueurs de code par PPM_k.

- Seconde lecture

  Substitution mot/code

8.3 Intégration de « mots composés »

Le taux de compression important obtenu par le codage de Huffman à base de mots, très supérieur au codage de Huffman à base de lettres, tire parti du codage par blocs. On aimerait bien pouvoir étendre cette notion à un niveau supérieur. En effet on voudrait passer du codage par blocs de lettres (les mots) au codage par blocs de mots (les mots composés). Cependant, mémoriser dans le dictionnaire le fait que deux mots simples ou plus forment un mot composé s'avère fort coûteux. Ainsi ce qui serait gagné sur la compression des données serait aussitôt perdu dans le dictionnaire. Étant donné que nous ne pouvons nous permettre de mémoriser un dictionnaire de mots composés, nous le construisons dynamiquement à la compression comme à la décompression grâce à une adaptation de l'algorithme de Ziv et Lempel.

Il nous faut préciser un peu la notion de mots composés. Les mots composés, au sens habituel ou linguistique du terme, s'avèrent peu nombreux ou plus exactement leur fréquence n'est pas suffisante pour qu'elle permette d'améliorer sensiblement le taux de compression. En revanche, certaines suites de mots séparés par l'espace telles que : *de la, alors que, il y a, c'est, ne pas le*, sont très fréquentes. Les linguistes appelent de telles suites de mots séparés par l'**espace** *cooccurrences*.

Les cooccurrences, comme les autres mots, sont traitées par l'algorithme de Huffman. Ceci nécessite donc un double passage par Ziv et Lempel, le premier construisant et comptant les cooccurrences, le second n'étant en quelque sorte qu'une simulation du premier pendant la phase de relecture. En effet, la table de Ziv et Lempel a été construite à la première lecture, il est inutile de la reconstruire, il suffit de simuler sa reconstruction en utilisant des marqueurs pour indiquer jusqu'où, à un instant donné, les cooccurrences sont de nouveau générées.

Nous donnons à la page suivante l'algorithme Huffman-LZW-cooccurrences qui permet de construire les cooccurrences lors de la première lecture du texte.

```
1.  W = Mot linguistique
2.  Tant que ( (p = Mot Séparateur ) ou (a = Mot )) ≠ Fin de fichier)
3.  Si (p = espace )
4.      Si (Wpa est dans le dictionnaire) W = Wpa
5.      Sinon
6.          Ajouter(Wpa,dictionnaire)
7.          Incrémenter(fréquence(W))
8.          Incrémenter(fréquence(p))
9.          W = a
10.     Fin du Sinon
11. Sinon
12.         Incrémenter(fréquence(W))
13.         Incrémenter(fréquence(p))
14.         W = a
15. Fin du Sinon
16. Fin du Tant que
17. Incrémenter(fréquence(W))
```

FIG. 8.7 – Huffman-LZW-cooccurrences

On remarque que les cooccurrences construites ne sont pas toutes utilisées. Seules celles apparaissant au moins deux fois le sont. Ce sont les seules pour lesquelles un mot du code de Huffman est calculé. Il faut cependant mémoriser toutes les cooccurrences construites pour que la décompression puisse se faire correctement. Les cooccurrences non utilisées sont mémorisées simplement en leur associant une longueur de mot du code nulle.

A la décompression, lors de la reconstruction du code de Huffman, on obtient des mots du code dont on sait déjà à quels mots ils correspondent (les mots simples du lexique) et d'autres, dont on ne connaît pas encore ce qu'ils représentent, qui seront attribués à des cooccurrences dans le même ordre où ils l'ont été pendant la compression.

8.4 Un prétraitement des mots du texte

Lors du traitement du texte, tout nouveau mot linguistique implique la création d'un chemin dans l'arbre qui représente le lexique, même s'il ne diffère que par la présence d'une ou plusieurs lettres en majuscule. La présence de majuscules peut provenir de plusieurs raisons : début de phrase, nom propre, sigles, etc...

Ceci nous conduit à effectuer un traitement par couple de mots consécutifs (mot séparateur, mot linguistique). Nous mémorisons les lettres majuscules présentes dans le mot linguistique en adjoignant un code spécial au mot séparateur qui le précède.

Dans ce but, trois codes opérateurs ont été définis : M, C, S.

M : la première lettre du mot suivant est mise en majuscule.
C : toutes les lettres du mot suivant sont mises en majuscule.
S : suivi d'une suite d'octets où les bits à 1 indiquent les lettres en majuscule.

Exemple :

Compression DE TeXtE naturel

est codé : (M)compression (C)de (S)$_{10101000}$texte ()naturel

Lors de la décompression, les codes spéciaux, qui ne sont pas des caractères pouvant appartenir à un mot séparateur, sont enlevés des mots séparateurs qui les contiennent et l'opération qu'ils représentent est appliquée au mot linguistique qui les suit.

Ce traitement permet de diminuer le nombres de mots linguistiques et donc le nombre de mots binaires dans le code de Huffman qui les représente. En revanche, la taille du lexique des mots séparateurs est quelque peu augmentée. Cependant, ce traitement permet d'abaisser le nombre global de mots contenus dans les deux lexiques (linguistique et séparateur).

8.5 Résultats

Dans le tableau ci-dessous Huff_M désigne la première version de l'algorithme de Huffman à base de mots, Huff_Co désigne la seconde qui fait appel aux cooccurrences. Les algorithmes Huff_M$^+$ et Huff_Co$^+$ sont les versions respectives de Huff_M et Huff_Co qui utilisent le traitement des majuscules.

Résultats

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.bible
Taille	125179	152089	426754	481861	2473400	4047392
gzip-9	48829	54191	144429	194277	721413	1176649
	39,0%	35,6%	33,8%	40,3%	29,1%	29,0%
bzip2	39569	43202	107706	145577	489583	845623
	31,7%	**28,4%**	**26,2%**	**31,2%**	**19,8%**	**21,9%**
PPM_2^e	43607	51318	146181	169004	870949	1230561
	34,8%	33,7%	34,2%	35,0%	35,2%	30,4%
PPM_3^e	39659	43966	426754	145727	653090	974498
	31,6%	28,9%	27,2%	30,2%	26,4%	24,0%
PPM_4^e	40117	43379	107476	141605	525957	854247
	32,0%	**28,5%**	**25,1%**	**29,3%**	21,2%	21,1%
PPM_5^e	42148	44970	109595	147684	486607	818274
	33,6%	29,5%	25,6%	30,6%	**19,7%**	**20,2**
Huff_M	41185	45455	109675	147193	654920	1012106
	33,1%	29,8%	25,6%	30,5%	26,4%	25,0%
Huff_M$^+$	40307	44457	107876	139807	680798	1016898
	32,1%	29,2%	25,2%	29,0%	27,5%	25,1%
Huff_Co	40420	42870	105316	145562	560967	856339
	32,2%	28,1%	24,6%	30,2%	**22,6%**	**21,1%**
Huff_Co$^+$	39756	41818	103856	584163	560967	860229
	31,6%	**27,4,%**	**24,3%**	**28,7%**	23,6%	21,2%

Les résultats obtenus avec Huff_M sont très proches de ceux de PPM_3 avec cependant l'avantage d'une plus grande rapidité de traitement et la possibilité de l'accès direct (via un index) au texte compressé. Ceux obtenus avec Huff_Co sont voisins de ceux de PPM_4 (parfois légèrement supérieurs).

8.6 Conclusion

Le fait d'utiliser une méthode de compression des lexiques très efficace permet de rendre performant l'algorithme de Huffman à base de mots sur de petits textes (de quelques dizaines de Ko à quelques Mo) et de ne plus le limiter au traitement de données textuelles très volumineuses (parfois plusieurs Go) qui était son utilisation première.

Les taux de compression obtenus avec de Huff_M sont légèrement moins bons que ceux obtenus avec Huff_Co. Cependant l'algorithme Huff_M permet, moyennant la construction d'un index, un accès direct au texte. Avec Huff_Co, on dispose d'un accès direct avec perte d'information car la lecture ne commençant pas au début du fichier, certains mots du code de Huffman représentent des cooccurrences que l'algorithme de Ziv et Lempel ne peut reconstruire.

Chapitre 9

Langue et compression

Jusqu'ici, les algorithmes que nous avons présentés étaient dédiés à des textes dont les lexiques étaient formés de mots écrits à l'aide d'une partie de l'ensemble des 256 caractères ASCII. De plus, nous n'avons pas tenu compte de la langue du texte traité (implicitement, qu'il s'agissait de textes en français ou en anglais).

Nous allons aborder dans ce chapitre deux questions souvent ignorées dans la littérature sur la compression de textes. La première concerne l'universalité du compresseur ; tout algorithme est-il capable de compresser un texte écrit en français, en russe, en japonais ou en chinois ou *a fortiori* un texte multilingue ? La seconde consiste à se demander dans quelle mesure, nous pouvons profiter des connaissances que nous avons sur la langue dans laquelle le texte traité est écrit pour en améliorer la compression.

9.1 Universalité du compresseur

9.1.1 Langues et symboles de base

Dans le chapitre 2 (intitulé *Codes et Compression*), nous avons défini un texte comme une suite de symboles appartenant à un ensemble de 256 éléments (standard de 0 à 127, ASCII supérieurs de 128 à 255). Malheureusement ce nombre de symboles se révèle parfois insuffisant pour certaines langues non alphabétiques (chinois, japonais, coréen). Examinons brièvement les systèmes d'écritures des langues les plus connues.

Il faut tout d'abord distinguer les langues écrites au moyen d'un alphabet, de taille relativement réduite, des langues non alphabétiques, notamment asiatiques.

9.1.1.1 Langues utilisant l'alphabet latin

En ce qui concerne les langues écrites avec un alphabet, le découpage correct du texte en mots nécessite de connaître parfaitement les codes attribués aux caractères spéciaux utilisés dans la langue traitée. Un certain nombre de langues ont pour base l'alphabet latin utilisé en anglais moderne (lettres de *a* à *z* majuscules et minuscules reprises dans le standard ASCII). Certaines langues utilisent en plus des caractères formés à partir d'un caractère standard auquel on ajoute un ou plusieurs signes (un accent, une cédille , un tréma, etc...) appelés signes diacritiques. Certains de ces caractères spécifiques à certaines langues appelés donc caractères diacritiques sont donnés dans le tableau suivant.

Langues	Caractères
Allemand	ä, ë, ö, ß, ü
Danois	ä, ë, ö, ü
Espagnol	á, ch, é, í, ll, ñ, ó, ú, ¿, ¡
Français	à, â, ä, ç, é, è, ê, ë, î, ï, ô, ö, œ, ù, û, ü
Italien	á, à, é, è, ì, ò, ù
Hongrois	á, cs, é, gy, ny, ó, ö, ő, sz, ty,ú, ü, ű, zs
Néerlandais	ä, ë, ö, ü
Norvégien	ä, å, ö
Polonais	á, ą, ć, ę, ł, ń, ó, ś, ú, ù, û, ü, ż, ź
Portugais	á, ã, ç, é, í, ó, õ ú
Roumain	ă, â, î, ş
Serbocroate	ć, č, lj, nj, š, ž
Slovaque	á, ä, č, ď, dz, dž, é, í, ch, ľ, ň, ó, š, ť, ú, ý
Suédois	ä, å, ö
Tchèque	á, ď, ě, í, ň, ř, š, ť, ž
Turque	ç, ğ, ö, ş, ü

Tab. 9.1 – Caractères diacritiques

On remarque dans ce tableau certains caractères particuliers, juxtaposition de deux caractères simples. Bien qu'étant souvent tapés comme deux caractères, ils sont considérés comme une lettre notamment pour les algorithmes de tri lexicographique. Ainsi, en espagnol le mot *crisis* est avant le mot *chico* car *ch* est une lettre à part entière et $c < ch$. Il faut remarquer la présence dans certaines langues de caractères de ponctuation spécifiques comme par exemple le point d'interrogation espagnol ¿.

9.1.1.2 Langues n'utilisant pas l'alphabet latin

Si un grand nombre de langues partagent l'alphabet latin, de nombreuses langues ont leur alphabet propre ou n'utilisent pas un système alphabétique.

- **Le grec.** L'alphabet grec moderne comporte lettres majuscules et minuscules accentées. Les signes diacritiques appelés *esprits* ne sont plus utilisés dans le grec actuel.

- **Le russe.** Le russe utilise l'alphabet cyrillique qui comporte 31 lettres. L'alphabet cyrillique est utilisé pour écrire le russe, le bulgare et aussi le serbocroate. Remarquons que le serbocroate s'écrit aussi avec une variante de l'alphabet latin (appelée latinica) ; une langue deux alphabets. Notons qu'il existe aussi certaines variations entre les cyrilliques russe, bulgare et serbocroate.

- **L'arabe.** L'arabe est une langue qui s'écrit de droite à gauche au moyen de 28 caractères dont la forme varie suivant leur place dans le mot : lettre initiale, médiane, finale, isolée. L'arabe utilise également un certain nombre de signes diacritiques qui ont la particularité de pouvoir s'appliquer à quasiment toutes les lettres de base donnant ainsi un nombre de caractères diacritiques élevé.

- Le **chinois**. Le chinois s'écrit au moyen d'idéogrammes construits grâce à 214 clés.

- **Le japonais.** Le japonnais utilise deux syllabaires, le *hiragana* et le *katakana* qui comportent respectivement 69 et 68 symboles ainsi que les idéogrammes chinois, appelés *kanji*. Le syllabaire *hiragana* qui est une écriture cursive est le plus utilisé. Le *katakana* est utilisé principalement pour l'écriture des mots étrangers. Selon les recommandations du gouvernement japonais, le nombre de symboles, kana et kanji, à connaître est de l'ordre 2000 (très exactement 1945 kanji).

- **Le coréen.** Le coréen utilise un alphabet de lettres qui sont regroupées par caractères appelées Hangul au nombre de 7000.

Nous venons de voir que la limitation à 256 symboles est très largement insuffisante pour certaines langues. Il faut ajouter que même si l'on se limite aux langues alphabétiques, les caractères propres à chacunes d'elles, caractères latins accentués, caractères grecs ou cyrilliques sont souvent codés en utilisant les codes ASCII supérieurs. Ceux-ci étant en nombre limité, un ñ espagnol pourrait avoir le même code qu'un β grec.

L'utilisation d'un même code pour représenter les caractères de langues différentes rend donc difficile, voire impossible, la rédaction de texte bilingue et à fortiori multilingue. De plus, le codage des caractères n'est pas standardisé, il est dépendant de la machine utilisée. Nous pouvons illustrer ce dernier point en considérant les codes des caractères diacritiques français selon différentes machines.

Caractère	VMS	Mach
à	224	213
â	226	215
ä	228	217
ç	231	219
è	233	220
é	232	221
ê	234	222
ë	235	223
î	238	228
ï	239	229
ñ	241	231
ô	244	238
ö	246	239
ù	249	242
û	251	244
ü	252	246

TAB. 9.2 – Codes et Machines

Toutes ces raisons ont contribué à la naissance d'une nouvelle norme de codage appelée *Unicode* permettant de coder sans difficulté des textes en de nombreuses langues ainsi que des textes multilingues.

9.1.1.3 Le standard Unicode

Dans le standard Unicode, tout symbole est codé sur **deux octets**. Le passage du code ASCII à Unicode consiste à ajouter 00 comme poids fort. Ainsi, la lettre majuscule `A` codée 41_{16} en ASCII sera codée 0041_{16} selon Unicode et notée $:U+0041$.

Remarquons tout de suite que ceci implique d'exprimer le taux de compression en bits par symbole. Nous n'avons plus l'équivalence un caractère un octet, un texte français verrait son taux de compression divisé par deux. La mise en œuvre de ce standard qui remonte à 1988 permet dans sa version 3.0 parue en 2000 le codage de près de 50000 symboles. Cette nouvelle norme de codage a pour but de faciliter l'écriture de textes en langues naturelles multilingues et l'échange de données de type textuelles à travers les réseaux. Il est notamment intégré au langage Java.

La norme Unicode respecte un certain nombre de principes de base qui sont :

– Tout caractère Unicode est codé sur 16 bits. Cependant deux formats (UTF8, UTF7), ont été définis de façon à pouvoir traiter, sans perte d'information, les données de type Unicode dans des environnements travaillant sur 8 ou 7 bits.

– Toute séquence de 16 bits correspond à un caractère Unicode. Dans la version 3.0 de cette norme, 49194 symboles sont actuellement répertoriés sur les 2^{16} théoriquement possibles. En fait cette limite de 2^{16} peut être dépassée par un système appelé *surrogate* permettant de coder alors plus d'un million de symboles.

– Unicode code des caractères et non des dessins. Ainsi un A de police Times ou Old English aura le même code.

– Unicode code du texte (en anglais *plain text*) et non du texte enrichi comme le font SGML, HTML, LaTeX, RTF.

– Unicode permet de gérer les sens d'écriture, de gauche à droite (latin, grec, cyrillique), de droite à gauche (arabe, hébreux), les écritures verticales (chinois, japonais) et le mélange de ces sens au sein d'un même texte.

– Le standard Unicode ne permet pas de déterminer la langue du texte. Un e peut être le e français, le e espagnol. Unicode définit des *scripts* qui sont des ensembles de symboles partagés par un certain nombre de langues. Au sein d'un même script, les caractères de langues différentes ont un code identique. Nous donnons dans le tableau A.1 un ensemble de scripts gérés par Unicode.

– Un caractère diacritique peut être codé de plusieurs manières : soit sous la forme d'un caractère « précalculé » qui n'utilise qu'un seul code, soit sous la forme d'une suite de codes comprenant le code du caractère de base suivi des codes des signes diacritiques qu'il comporte.

Ainsi ą́ peut s'exprimer sous trois formes :

ą́

a + ’ + ˛

a + ˛ + ’

9.1.1.4 Le codage UTF8

Ce codage permet de transformer un texte Unicode (codage sur 16 bits) sous la forme d'une suite d'octets comme indiqué dans le tableau 9.3.

Valeur	UTF16	1^{er} octet	2^{ieme} octect	3^{ieme} octect	4^{ieme} octect
000000000xxxxxxx	000000000xxxxxxx	0xxxxxxx			
00000yyyyyxxxxxx	00000yyyyyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyyyyxxxxxx	zzzzyyyyyyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	

TAB. 9.3 – Codage UTF8

Ce tableau nous permet de voir que tout symbole faisant partie du code ASCII standard est représenté par un octet dont le bit 7 est 0. Tout symbole ne faisant pas partie des symboles ASCII standards est représenté par une suite d'octets dont le premier comporte dans ces bits de plus forts poids, écrit en unaire (0 bit de stop) le nombre d'octets utilisés pour son codage. Ainsi, un texte en français qui est habituellement représenté avec le codage ASCII étendu, subit, en UTF8, une très légère augmentation de sa taille due à la présence des caractères accentués représentés par deux octets. Si le codage UTF8 est une représentation plus compacte que l'unicode UTF16 pour les langues écrites à l'aide des scripts latins, il est en revanche plus gourmand que l'UTF16 si l'on doit, par exemple, coder un texte japonais. Il permet néanmoins pratiquer dans un texte latin, des insertions de textes asiatiques à moindre coût.

9.1.2 Compresseurs et taille de l'alphabet

Au regard de la taille des alphabets utilisés, les implémentations habituelles des algorithmes classiques ne peuvent efficacement traiter des textes chinois, japonais ou coréens, non seulement parce qu'ils ne traitent que 256 caractères, mais aussi parce que la gestion de l'initialisation des symboles dans ces compresseurs n'est pas dynamique. En effet, le plus souvent, les algorithmes classiques initialisent la totalité de l'alphabet ; ainsi le codage arithmétique initialise toutes les fréquences des caractères à 1, l'algorithme de Ziv et Lempel initialise la table des facteurs avec les 256 caractères ASCII. Il est impensable de procéder ainsi avec 10000 idéogrammes chinois ou 7000 Hanguls coréens alors que seule une partie d'entre eux sont présents dans le texte traité.

La gestion de ces très gros « alphabets » s'apparente beaucoup plus à celle des mots dans les algorithmes à base de mots exposés dans les chapitres précédents. Ainsi ces algorithmes sont facilement adaptables au traitement de langues ayant un nombre de symboles de base élevé en prenant l'équivalence : un caractère chinois, japonais, ou un Hangul = un mot.

En revanche, utiliser la notion de mot pour ces langues n'est pas aussi aisé que pour les langues européennes dans la mesure où elles ne vérifient pas toujours l'alternance mot linguistique mot séparateur, car il n'y a pas toujours de séparateur entre les mots, notamment pas d'espace. Il faut donc ici posséder un dictionnaire électronique de la langue traitée pour pouvoir découper correctement le texte.

9.2 Linguistique et compression

La plupart des algorithmes de compression ignorent la langue dans laquelle le texte traité est écrit. On peut tenter d'utiliser certaines connaissances linguistiques sur la langue du texte pour en améliorer la compression. Nous exposons ici plusieurs manières de s'adapter à la langue du texte traitée.

9.2.1 Amélioration et choix du découpage

Certaines améliorations de découpages peuvent faire appel à des informations de types linguistiques supplémentaires.

9.2.1.1 Traitement de l'apostrophe

En français comme en italien, on peut accrocher l'apostrophe au mot qui la précède. Ainsi pourra-t-on découper :

« l'élève » en « l' » + « élève » au lieu de « l » + « ' » + « élève »

« dell'ochio » en « dell' » + « ochio » au lieu de « dell » + « ' » + « ochio »

On remarquera que lors de la décompression, il suffit de vérifier s'il existe un caractère de ponctuation à la fin du mot lu. Si tel est le cas, le prochain code lu sera à nouveau un code mot linguistique au lieu d'un code de mot de séparateur, comme l'alternance habituelle nous y obligerait.

En revanche, si l'on considère l'anglais où l'apostrophe n'indique plus une élision mais une contraction, celle-ci doit être attachée au mot qui la suit.
Ainsi :

« She's » en « She » + « 's » au lieu de « She » + « ' » + « s »

Cependant ici, la règle d'alternance n'est plus respectée et nous ne pouvons pas savoir qu'il faut lire deux fois de suite un mot linguistique. Nous pouvons résoudre ce problème en inversant dans la forme compressée les deux mots linguistiques.

On écrira donc :

« She's » en « 's » + « She »

Ainsi à la décompression, l'existence d'un caractère de ponctuation au début du premier mot décodé nous indique qu'il faut lire à nouveau un mot linguistique qui sera alors écrit avant le mot déjà lu.

9.2.1.2 Découpage à l'intérieur d'un mot

Si l'on veut identifier clairement un mot par un code unique, notamment dans le but d'indexer le texte, il peut être utile de découper un mot lu.

En effet, en espagnol par exemple, on attache parfois un ou deux pronoms à la forme verbale. Il faut donc, soit garder la forme attachée, soit découper la forme verbale rencontrée, soit introduire un ou plusieurs mots vides de ponctuation.
Ainsi on pourra écrire :

« Cantame » ou bien « Canta » + « ε » + « me »

« Cantala ou bien « Canta » + « ε » + « la »

« Digamelo » ou bien « Diga » + « ε » + « me » + « ε » + « lo »

En allemand la forte production de mots composés pose des problèmes comparables. Si nous considérons le mot :

Kartoffelsalat

Nous pouvons le traiter de la même manière :

« Kartoffel » + « ε » + « salat »

Dans le cas d'une indexation du texte, la recherche du mot *salat* ne consiste alors qu'à rechercher un code dans le flux des codes qui représentent le texte. En revanche dans la première représentation, il faux considérer tous les mots composés dans lesquels le mot *salat* peut se trouver.

9.2.1.3 Traitement de la ponctuation

Nous pouvons constater expérimentalement que les « petits mots », mots de longueur inférieure à 5 (en français), sont le plus souvent suivis d'un espace. On peut alors, tout comme pour l'apostrophe, coller cet espace au « petit mot » qui le précède. La décompression, comme nous l'avons vu (9.2.1.1), ne pose pas de problème.

9.2.2 Traitement affiné des cooccurrences

Nous avons déjà utilisé la notion de cooccurrences au chapitre 7 en utilisant une modification de l'algorithme de Ziv et Lempel afin de tirer parti de l'effet « codage par bloc » de l'algorithme de Huffman.

En fait l'algorithme de Ziv et Lempel permet, notamment, de traiter 3 types de cooccurrences :

1. De vrais mots composés comme :
   Président de la République
   pomme de terre
   taxe d'habitation.
2. Des cooccurrences de « petits mots » (souvent des particules préverbales) tels que :
   de le
   c'est
   il ne le
   sur la
   pour que.
3. Des cooccurrences aux allures de mots composés comme :
   l'agriculture russe
   la pêche espagnole
   le prix des agrumes.

Dans ces cooccurrences, celles de type 1 pourraient être traitées par un filtrage du texte par un dictionnaire de mots composés tel le DELACF[1]. Les cooccurrences de type 2 peuvent être captées par une heuristique simple consistant à considérer la plus longue suite de « petit mots » qui sont souvent des particules préverbales. Seules les cooccurrences de type 3 nécessitent un système dynamique de type Ziv et Lempel.

Remarquons que lorsque les cooccurrences sont traitées par filtrage par un lexique extérieur, la première occurrence est codée par un seul code alors que dans un traitement par Ziv et Lempel la première occurrence est codée par autant de codes qu'il y a de facteurs déjà présents dans la table.

L'algorithme de Ziv et Lempel capte également des cooccurrences dont on peut penser qu'elles ne sont ni fréquentes (non utiles pour la compression) ni très signifiantes (pas d'intérêt pour l'indexation automatique). Si nous considérons la phrase :

La voiture rouge roulait lentement dans la direction d'un petit village ...

Les facteurs « rouge roulait » et « lentement dans » ont peu de chance d'être fortement récurrents dans un texte. En revanche les facteurs « dans la direction » et « petit village » sont sans doute plus intéressants à conserver.

Il faut donc mettre en œuvre des systèmes de filtrage qui permettent de sélectionner certains facteurs et d'en ignorer d'autres. Cette sélection peut s'effectuer soit par prétraitemement du texte, soit à la volée lors de la compression, en considérant les catégories lexicales des mots rencontrés pour capter par exemple, des groupes nominaux en découpant le texte en de bons endroits.

[1]Dictionnaire électronique des formes fléchies des mots composés du LADL

9.2.3 Modèle de langue

La mise en œuvre d'un modèle de chaîne de Markov d'ordre 1 sur les mots est coûteuse à cause des ressources en mémoire nécessaires. En effet, si nous nous proposons de coder le mot y suivant un mot x qui a déjà été traité, il faut connaître les fréquences de tous les successeurs passés de x, ce qui s'avère fort coûteux vu la taille des lexiques de textes.

Au lieu de nous servir de connaissances liées au texte, nous pouvons nous servir de nos connaissances concernant la langue dans laquelle il est écrit. Nous pouvons nous intéresser aux mots n'apparaissant jamais après le mot x. Ainsi, si A est l'alphabet du texte et X_{int} les mots de la langue qui ne peuvent suivre x, y appartient à $(A - X_{int})$. Le calcul du code de y se fait en mettant temporairement à 0 les fréquences des éléments de $A \cap X_{int}$. Ce procédé rappelle celui connu sous le nom de *tableau des scores* utilisé dans l'algorithme PPM.

On peut considérer, comme pour les chaînes de Markov, des interdits d'ordre 1,2,..k. En effet, plus le contexte est grand, plus le filtrage est important.

Considérons la phrase :

Il ne lui en x.

Si la prise en compte d'un contexte d'ordre 3 permet de dire que x est une forme verbale, seul l'ordre de 4 (prenant en compte le *Il*) nous indique qu'il s'agit d'une forme verbale à la troisième personne du singulier.

Les tests expérimentaux qui ont été réalisés en utilisant ce modèle avec l'algorithme Huff_M$^+$ à l'ordre 1 ont fait appel à une structure de données qui associe à chaque mot du dictionnaire de référence[2] un vecteur binaire indiquant l'appartenance de ce mot (bit à 1 ou 0) à une catégorie lexicale donnée. Les résultats obtenus n'ont montré qu'une légère amélioration du taux de compression en regard des ressources linguistiques et informatiques nécessaires à sa mise en œuvre.

Les notions de mots *permis* ou *interdits* contenues dans ce modèle peuvent être exploitées d'une autre manière qui nécessite beaucoup moins de ressources linguistiques. Si nous considérons certains petits mots grammaticaux, tels que *de*, *à*, *pour*, *the*, *of*, ils ont, en plus d'être très fréquents, la particularité d'opérer une forte sélection sur les mots qui les suivent. En français, par exemple, le mot *de* n'est jamais suivi d'une forme conjuguée d'un verbe, le mot *la* n'est jamais suivi d'un mot masculin.

Ceci nous permet de mettre en œuvre un algorithme de Huffman sur les mots qui utilise un ordre 1 restreint à ces mots grammaticaux.

[2]Pour les besoins des test le dictionnaire est issu d'un traitement du texte par le logiciel Intex du à Max Silberztein : http ://www.ladl.jussieu.fr

On utilise pour chaque langue une liste préétablie de mots grammaticaux fréquents qui opèrent une forte classification des mots qui les suivent.
Pour chaque mot de cette liste on considère un arbre de Huffman qui mémorise les fréquences de ses successeurs et permet donc d'en calculer les représentations binaires. Chaque mot de la liste constitue un contexte d'ordre 1. Ainsi tout mot du texte est codé soit à l'ordre 1 s'il est précédé d'un mot appartenant à la liste, soit à l'ordre 0. De plus, notre algorithme (Huff_M$^+$) utilisant un codage particulier des majuscules, un arbre est réservé pour le traitement des mots qui en comportent.

Si nous considérons une liste L de contextes comportant k mots, l'algorithme Huff_M* utilisera un ensemble de $k+3$ arbres de Huffman dont les affectations sont :

- Arbre 0 : mots séparateurs.
- Arbre 1 : mots linguistiques à l'ordre 0 (dans le contexte ε).
- Arbre 2 : mots précédés de mots comportant des majuscules.
- Arbre i+3 : mots précédés du $i^{ième}$ mot de la liste ($1 \leq i \leq k$).

De plus il utilise l'heuristique évoquée en 9.2.1.3 qui consiste à attacher à tout mot de longueur inférieure à 4 l'espace qui le suit. L'apostrophe (en français ou italien) est également collée au mot précédent.

On peut étendre cet algorithme en ne considérant pas seulement des contextes réduits à un mot, mais considérer des ensembles de mots. Par exemple en français, on peut considérer le contexte des mots élidés [*c*, *d*, *j*, *l*, *m*, *qu*, *s*, *t*] qui sont tous suivis de mots commençant par une voyelle.

9.2.4 Indexabilité

L'index d'un texte comprend pour chaque mot du texte la liste de ses positions. Dans le cas d'un texte non compressé, cette position est le nombre d'octets depuis le début du texte. Dans le cas d'un fichier compressé par codage de Huffman (donc à longueur variable), cette position est un nombre de bits par rapport au début du texte exprimé sous la forme d'un nombre d'octets suivi d'un nombre de bits b ($0 \leq b \leq 7$) par rapport au début de l'octet suivant.

Il est clair que l'algorithme Huff_M$^+$ permet l'indexation du texte compressé.
Pour Huff_M* la représentation d'un mot peut dépendre du mot qui le précède. Dans ce cas, le mot recherché est le premier à afficher. On l'écrit et il constitue le contexte permettant de décompresser les mots suivants. L'index doit contenir non plus la liste des positions du mot recherché mais la liste des positions de ses successeurs.

Ce type d'index permet de conserver l'accès direct avec des textes compressés à l'ordre 1, sans que des informations supplémentaires aient besoin d'y être rajoutées. En revanche, le traitement d'un texte à un ordre supérieur impliquerait la construction d'un index enrichi.

L'algorithme Huff_M* étant fondé sur l'utilisation de contextes contenant des mots grammaticaux, sa généralisation à un ordre supérieur à 1 qui conserve l'indexabilité du texte compressé, peut être faite, en considérant des *k-mots* grammaticaux. En français, par exemple, le *bimot* `à le` est suivi d'un verbe à l'infinitif.

9.3 Résultats expérimentaux

L'algorithme Huff_M* a été testé sur les textes en langue naturelle du Canterbury Corpus, que nous avons utilisés tout au long de cette thèse et aussi sur des textes en français, espagnol, italien, et grec qui sont issus notamment du projet Gutemberg.

Nous donnons pour chaque langue les contextes utilisés et pour chaque texte, le nombre de mots qu'il comporte et le nombre de mots traités dans un contexte différent de ε. Les choix de contextes effectués pour ces tests sont faits à titre d'exemples. Les choix de contextes effectués ont pour objet de séparer les mots qui suivent ces contextes par genre et par nombre pour les langues latines et le grec. Pour l'anglais qui est la langue ayant le plus faible nombre de flexions, on a, par exemple, isolé les formes verbales de la troisième personne du singulier. Une étude plus poussée permettrait certainement d'obtenir de meilleurs choix de contextes. On pourrait envisager que ce choix ne soit pas fait *a priori*, mais issu d'un prétraitement du texte considéré.

Anglais

Texte	asyoulik	alice29	lcet10	plrabn12	world192	K.J.bible
Taille	125179	152089	426754	481861	2473400	4047392
gzip-9	48829	54191	144429	194277	721413	1176649
	39,0%	35,6%	33,8%	40,3%	29,1%	29,0%
bzip2	39569	43202	107706	145577	489583	845623
	31,7%	28,4%	26,2%	31,2%	19,8%	21,9%
Huff_M$^+$	40307	44547	107876	139807	680805	1016898
	32,1%	29,2%	25,2%	29,0%	27,5%	25,1%
Huff_M*	39870	43381	105563	139662	651867	922479
	31,8%	28,5%	24,7%	28,9%	26,3%	22,7%
Nombre de mots	23303	27100	62690	80661	341803	767835
Nombre de mots ordre 1	5765	7271	15203	14527	27518	210132

Contextes utilisés : a, an, at, from, for, in, is, that, the, of, to, {I, you, we, they}, {he, she, it}.

Les taux de compresson obtenus avec Huff_M* sont légèrement meilleurs que ceux obtenus avec Huff_M$^+$ et sont semblables à ceux obtenus avec bzip2.

Français

Texte	Verne	AFP	Droit	Bible
Taille	363631	744794	1705435	4150190
gzip-9	135166 37,1%	284135 38,1%	565627 33,1%	1435026 34,5%
bzip2	104924 28,8%	211972 28,5%	413614 24,3%	1073992 25,9%
Huff_M$^+$	108839 29,9%	233757 31,3%	449219 26,3%	1194774 28,7%
Huff_M*	101979 28,0%	220025 29,5%	402195 23,5%	1061752 25,5%
Nombre de mots	62377	120468	287734	775481
Nombre de mots à l'ordre 1	22231	36120	103113	278502

Contextes utilisés : à, au, aux, de, pour, dans, en, y, pas, {je, tu, il, elle, nous, vous, ils, elles, ne}, {le, un, mon, ton, son, ce, cet}, {la, une, ma, ta, sa, cette}, {les, des, mes, tes, ses, leurs, ces}, {est, était}.

Espagnol

Texte	Celestina	Mundo
Taille	689152	4776976
gzip -9	263377 38,3%	1824618 38,7%
bzip2	211668 31,3%	1407505 29,5%
Huff_M$^+$	221806 32,1%	1374912 28,7%
Huff_M*	207879 30,1%	1295985 27,1%
Nombre de mots	106784	783768
Nombre de mots à l' ordre 1	15968	192714

Contextes utilisés : a, de, por, para, en, el, la, los, las, {un, ese, este, aquel}, {una, esa, esta, aquella}.

En espagnol, les déterminants démonstratifs distinguent trois niveaux de proximité `ici`, `là`, `là-bas` qui se traduisent par l'emploi de `ese`, `este`, `aquel` au masculin et `esa`, `esta`, `aquella` au féminin.

Les taux de compressions obtenus en français et espagnol avec Huff_M* sont sensiblement meilleurs que ceux obtenus avec Huff_M$^+$ et montrent qu'un petit nombre de contextes permet de traiter à l'ordre 1 une part importante des mots du texte.

Italien

Texte	Elogio	Polder	Favole	Comedia	Chichote
Taille	203534	209245	393416	565977	1029438
gzip -9	78912	80624	160489	225624	384476
	38,8%	38,6%	40,8%	39,9%	37,4%
bzip2	63600	63873	125037	176418	286695
	31,4%	30,5%	31,8%	31,1%	27,9%
Huff_M$^+$	63030	63805	125769	183515	290955
	30,9%	30,4%	31,9%	32,4%	28,2%
Huff_M*	61861	61756	122054	179118	274583
	30,3%	29,5%	31,0%	31,6%	26,6%
Nombre de mots	34206	36203	70709	104135	176805
Nombre de mots à l'ordre 1	8381	8023	16765	23139	41736

Contextes utilisés : a, per, di, ne, che, {l, all, dell, dall, nell, coll}, {lo, allo, dello, dallo, nello, collo}, {il, al, del, dal,nel, col}, {la, alla, della, dalla, nella, colla}, {i, ai, dei, dai, nei, coi}, {le, alle, delle, dalle, nelle, colle}, {io, tu, lui, lei, noi, voi, loro}.

En italien, l'article défini masculin est `il` devant les mots commençant par une consonne sauf {`s+consonne`, `z`, `gn`, `ps`} et `lo` devant les mots commençant par `s+consonne`, `z`, `gn`, `ps`. L'article défini féminin est `la` devant les mots commençant par une consonne. Il prend la fome élidée `l` au masculin et au féminin devant les mots commençant par une voyelle. L'italien comporte un article défini masculin pluriel `i` et défini féminin pluriel `le`. Tous les articles définis se combinent avec les cinq prépositions `a` (a), `di` (de), `da` ,`nel` (dans), `col` (avec). Au masculin pluriel, par exemple, on obtient `ai`, `dei`, `dai`, `nei`, `coi`.

Grec

Texte	keros	athlitika	astin	apopsis	agelies
Taille	56218	208974	210714	390115	1401449
gzip -9	16755	86068	79757	156946	511867
	29,8%	41,2%	37,9%	40,3%	36,6%
bzip2	15142	68232	63558	120229	168211
	27,0%	32,7%	30,2%	30,8%	22,0%
Huff_M$^+$	16745	70866	65003	120570	345322
	29,7%	33,9%	30,8%	30,9%	24,6%
Huff_M*	16439	70481	64440	119276	328214
	29,2%	33,7%	30,5%	30,5%	23,4%
Nombre de mots	8434	31449	30637	57716	189649
Nombre de mots à l'ordre 1	617	2284	2151	4479	1598

Contextes utilisés : o (le), η (la), τo (le neutre), $o\iota$ (les), $\tau\alpha$ (les neutre).

Les différences de taux de compression entre Huff_M$^+$ et Huff_M* obtenues pour l'italien et le grec sont plus faibles que celles obtenues pour le français et l'espagnol. En ce qui concerne le grec, les résultats les meilleurs sont obtenus avec bzip2. Ceci pourrait être dû au fait que la longueur moyenne des mots grecs est plus importante que celles des autres langues traitées. Les lexiques des textes grecs seraient alors moins compressés que ceux des autres langues.

9.4 Application à la musique

Les algorithmes de compression de textes que nous avons présentés pourraient être adaptés à la compression de textes musicaux. Comme dans les textes en langue naturelle, on peut retrouver dans un texte musical des notions d'alternance (note, durée), et de mots (suite ascendante de notes de même durée, suite de durées de notes caractérisant un rythme).

Dans une première approche, on peut s'intéresser à un texte représentant la partition d'un instrument monophonique.

On peut considérer que cette partition est formée de séquences de notes définies par leur hauteur, durée, altération, accentuation, ornementation et de signes musicaux définissant la structure du morceau. Les formats musicaux sont principalement de deux types : numériques ou symboliques.

- Les formats numériques conservent des séquences de notes à jouer caractérisées par l'instrument utilisé, leur hauteur, leur durée. Le plus connu de ces formats est le format **MIDI**.
- Les formats symboliques permettent de mémoriser une partition telle qu'elle est écrite.

Considérons les ensembles H de hauteurs, D de durées, A_l d'altérations, A_c d'accents, O d'ornements, S de structures.

$$H = \{\text{do}_1, \text{ré}_1, \text{mi}_1, \text{fa}_1, \text{sol}_1, \text{la}_1, \text{si}_1 \ldots\ldots \text{do}_9\}$$

$$D = \{ \text{𝅝}, \text{𝅗𝅥}, \text{♩}, \text{♪}, \text{𝅘𝅥𝅯}, \text{𝅘𝅥𝅰}, \text{𝅘𝅥𝅱}, \text{𝄻}, \text{𝄼}, \text{𝄽}, \text{𝄾}, \text{𝄿}, \text{𝅀} \}$$

$$A_l = \{ \text{𝄫}, \text{♭}, \text{♮}, \text{♯}, \text{𝄪} \}$$

$$A_c = \{ \text{𝄐}, \text{𝄑}, \text{𝄒}, \text{𝄓}, \text{𝄔} \}$$

$$O = \{ tr, \text{𝆕}, \text{𝆖}, \text{𝆗}, \text{𝆘}, \text{𝆙}, \text{𝆚}, \text{𝆛}, \text{𝆜}, \text{𝆝}, \text{𝆞} \}$$

$$S = \{ \text{𝄞}, \text{𝄢}, \text{𝄡}, \text{𝄵}, \text{𝄴}, \text{𝄶}, \text{𝄷}, \ldots \}$$

Si on appelle h, d, a_l, a_c, o, s les éléments respectifs des ensembles H, D, A_l, A_l, O, S, un texte musical peut s'écrire sous la forme :

$$T_M = (h \cdot d(\varepsilon + a_l)(\varepsilon + a_c)(\varepsilon + o)(\varepsilon + s))^*$$

La plupart des formats symboliques sont des formats propriétaires et évoluent avec les versions des logiciels qui les utilisent. De ce fait, le format MIDI est devenu le principal format d'échange de données musicales. Le texte musical issu d'un fichier de type MIDI est obtenu par des algorithmes d'évaluation et de quantification qui associent une durée à un symbole de durée (𝅝, 𝅗𝅥, ♩, ♪,...). Ces algorithmes ne permettent pas de restituer un texte musical identique à la partition originale.

Quelques formats symboliques s'élaborent peu à peu et constituent des formats d'échanges possibles. On compte parmi ceux-ci les formats MusiXML (orienté web), MusiXTeX dû à Daniel Taupin [62] (module de Tex[3]) qui sont des formats enrichis qui gèrent non seulement la structure musicale mais aussi sa présentation (polices, espacement des notes, etc...). Dans le but de nous limiter à la structure (sans enrichissement) de présentation, nous considérons le format ABC dû à Chris Walshaw [66].

Considérons l'extrait de la figure 9.1

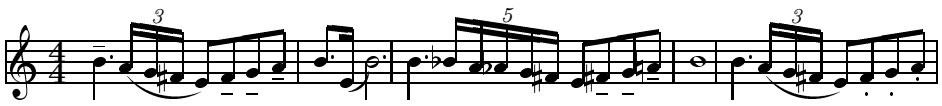


FIG. 9.1 – Mesures 1 à 5 de l'étude n^o 4 pour flûte ou violon solo d'Astor Piazzola

Sa représentation au format ABC est :

```
M:4/4
L:1/8
K:C
MB3 ((3:2:1A/(3:2:1G/(3:2:1^F/ E)MF MGMA | MB3/2(E/ B6) |
B3 (5:2:1_B/(5:2:1A/(5:2:1_A/(5:2:1G/(5:2:1_F/ EMF MGM=A | B8 |
B3 ((3:2:1A/(3:2:1G/(3:2:1^F/ E).F.G.A |]
```

Le format ABC utilise la notation anglo-saxone pour les notes de la gamme : C,D,E,F,G,A,B. Les symboles $\flat, \natural, \sharp$, . , -, sont respectivement représentés par :_ ,= ,^ ,. ,**M**. La durée des notes est exprimée par coefficient multiplicateur de la durée de base notée L (1/8 *i.e* une croche dans l'exemple précédent). Ainsi B3 est un *Si noire pointée* et ^F/ est un *Fa*$\sharp$ *double croche*. Les parenthèses ouvrante et fermante représentent respectivement le début et la fin de liaison d'un ensemble de notes. Une parenthèse ouvrante suivie de chiffres représente un $n-uplet$ de notes.

Par comparaison le source (ci-dessous) de cet extrait en MusiXTeX est beaucoup plus complexe et peu lisible car il doit gérer non seulement la structure musicale mais aussi sa présentation (par exemple les pentes des liaisons des croches).

```
\begin{music}
\nobarnumbers
\parindent10mm
\generalmeter{\meterfrac44}
\startextract
\notes\ust j\qlp i\enotes
\notes \islurd0 h\ibbu0h{-3}\qb0 h\butext0\qb0 g\tbu0\qb0{^f}\tslur0{e}\enotes
\notes \ibu0f3\qb0e{\lst e}\qb0f{\lst e}\qb0g{\lst h}\tbu0\qb0h\enotes\bar
\notes \ibu0i{-2}\qbp0i\tbbu0\tbu0\slur eid1\qb0e\hlp i\enotes\bar
\def\txt{\eightit 5}%
\notes \qlp i\ibbu0i{-3} \qb0{_i}\qb0h\butext0\qb0{_h}\qb0g\tbu0\qb0{^f}\enotes
\notes \ibu0f3\qb0e{\lst e}\qb0{^f}{\lst e}\qb0g{\lst h}\tbu0\qb0{=h}\enotes\bar
\notes \wh i\enotes\bar
\notes \qlp i\enotes
\def\txt{\eightit 3}%
\notes \islurd0 h\ibbu0h{-3}\qb0 h\butext0\qb0 g\tbu0\qb0{^f}\tslur0{e}\enotes
\notes \ibu0f3\qb0e{\lpz e}\qb0f{\lpz e}\qb0g{\lpz h}\tbu0\qb0h\enotes
\endextract
\end{music}
```

[3]Utilisé pour la rédaction de cette thèse

La notation musicale comporte, elle-même, un certain nombre d'outils de compression.

- Les altérations mises à la clé permettent d'éviter de les mettre sur un certain nombre de notes de la portée.
- Les barres de reprises permettent de répéter tout ou partie d'une suite de mesures.
- Certains symboles dont les ornements représentent un ensemble de notes calculable à partir de la note qui les portent (figure 9.2).

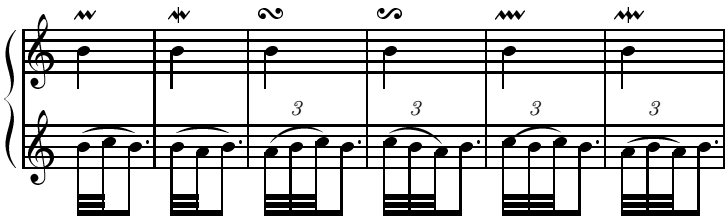


FIG. 9.2 – Ornement et effet produit

Considérons à nouveau l'extrait de la figure 9.1 sans tenir compte des liaisons (dans un but de simplification). Appelons mot une suite de notes de même durée munies de leurs altérations mais non de leur accentuation. Appelons $m_{i,j}$ le $i^{ème}$ mot de la mesure j. Intéressons-nous aux mesures 1, 3 et 5 toutes trois composées de trois mots.

Nous avons : $m_{1,1} = m_{1,3} = m_{1,5}$, $m_{2,1} = m_{2,5}$, $m_{3,1} = m_{3,5}$

Posons : $M_1 = m_{1,1}$, $M_2 = m_{2,1}$, $M_3 = m_{3,1}, M_4 = m_{2,3}$

On peut appliquer au texte musical un procédé analogue à celui utilisé pour le traitement des majuscules que nous avons proposé pour la compression de texte en langue naturelle. On considère des "mots modifieurs" ou foncteurs qui s'appliquent aux mots musicaux qui les suivent. En utilisant ce principe, on peut écrire :

$$Mesure_1 = (^{-})M_1M_2(\varepsilon,^{-},^{-},^{-})M_3$$
$$Mesure_5 = M_1M_2(\varepsilon,\cdot,\cdot,\cdot)M_3$$

En ce qui concerne la mesure 3, on peut considérer un mot modifiant l'altération et écrire :

$$Mesure_3 = M_1M_4(\sharp,\varepsilon,\varepsilon,\natural)(\varepsilon,^{-},^{-},^{-})M_3$$

Remarquons qu'il est possible d'utiliser une notion de contexte musical comparable à celle de contexte linguistique (par exemple, accord déterminant nom, mots interdits). En se rappelant, qu'une altération portée par une note affecte toutes les notes identiques de la mesure courante, on pourrait écrire :

$$Mesure_3 = M_1M_4(\sharp,\varepsilon,\varepsilon,\varepsilon)(\varepsilon,^{-},^{-},^{-})M_3$$

En effet, la dernière note de M_3 est un *Fa naturel*, comme la dernière note de la mesure 3. Le symbole bécarre porté par cette note n'est présent que pour annuler l'effet du bémol précédent et pourrait être calculé par une analyse du contexte. Si l'on étend les outils à l'utilisation de ceux concernant la recherche de motifs approchés, on peut remarquer que $M_4 = [si\flat, \_, la\flat, \_, \_]M_2$ avec insertion de deux notes.

Le court extrait musical de la figure 9.1 nous a permis de remarquer une proximité de traitement entre les textes en langue naturelle et les textes musicaux qu'il serait intéressant de développer. Ceci sera d'autant plus réalisable que les formats symboliques de représentation de textes musicaux sont de plus en plus complets et qu'un ensemble de symboles musicaux vient d'être intégré à la norme Unicode (version 3.1) facilitant ainsi la diffusion de ce type de documents *via* Internet.

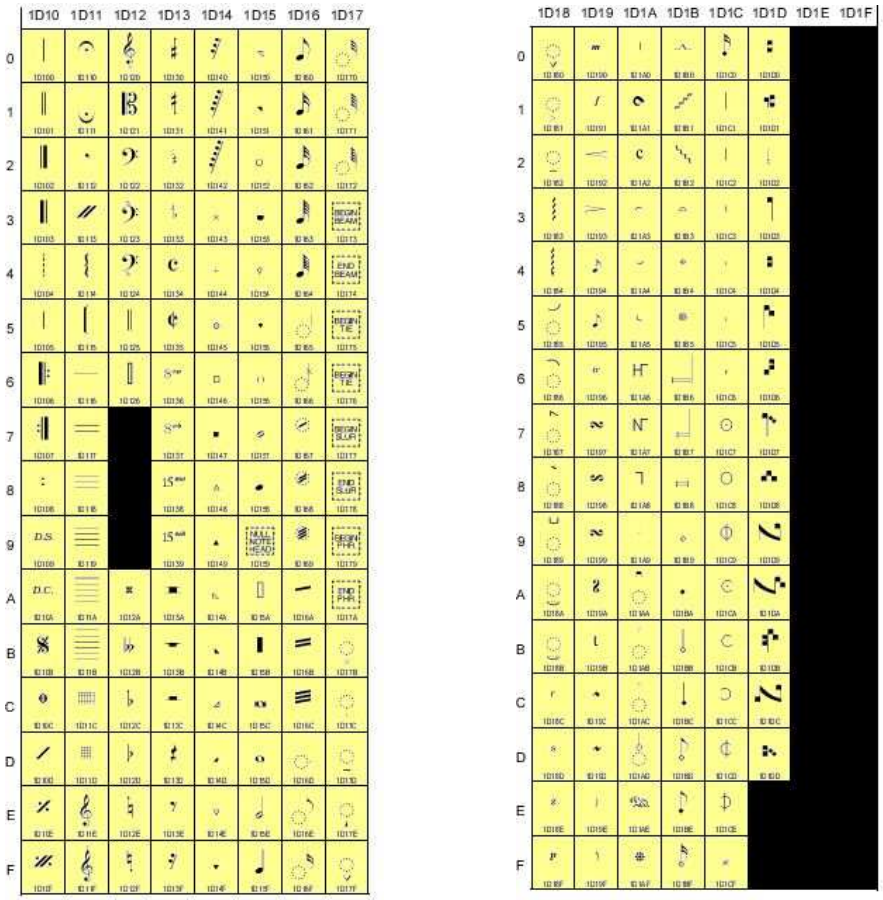


FIG. 9.3 – Script Musical : Notation Classique et Grégorienne

Conclusion

Nous avons tout au long de cette thèse présenté et comparé de nombreux algorithmes de compressions de textes. La comparaison entre les méthodes à base de mots et celles à base de caractères montre que les taux de compressions obtenus par les meilleurs algorithmes de ces deux familles sont semblables et permettent de diviser la taille originale du texte par un facteur compris entre 3,5 et 4.

Les méthodes fondées sur l'utilisation de chaîne de Markov (de type PPM) permettent d'obtenir les meilleurs taux de compressions. Elles nécessitent de nombreux calculs dus à l'utilisation du codage arithmétique et d'importantes ressources en mémoire dues à la création des arbres de contextes ce qui implique un temps d'exécution élevé dans le cas d'un nombre de contextes borné et se révèle inutilisable dans celui où le nombre de contextes est non borné. De ce fait, les algorithmes de type PPM sont peu utilisés en pratique.

Les algorithmes fondés sur la transformation de Burrows-Wheeler dont l'implémentation bzip2 utilise un codage de Huffman canonique permettent d'allier vitesse d'exécution et taux de compression élevé, très proches de ceux obtenus par la famille PPM.

Les résultats que nous avons obtenus montrent que le soin apporté à la compression des unités de bases du traitement que sont les mots, permet d'apporter une amélioration sensible du taux de compression des textes. Ceci est particulièrement remarquable dans le cas d'algorithmes qui rassemblent ces mots sous la forme de lexiques dont les représentations compactes permettent une compression très importante. Nos algorithmes permettent de traiter des textes de taille réduite dans lesquels la taille du lexique représente une part importante de celle du texte.

Parmi les algorithmes de compressions à base de mots que nous avons étudiés, ce sont ceux qui utilisent l'algorithme de Huffman qui s'avèrent les plus performants. Ils permettent d'obtenir les taux de compression parmi les meilleurs tout en conservant un accès direct à la forme compressée du texte.

L'utilisation de connaissances linguistiques permet d'affiner les taux de compression obtenus. La notion de contexte linguistique comme dans l'algorithme Huff_M* s'avère une approche utilisable en pratique. Le développement de tels

algorithmes nécessiterait de pouvoir interfacer des outils de type linguistique comme INTEX avec des outils de compression permettant par exemple d'effectuer un choix dynamique des contextes utilisés.

Annexes

Annexe A

Unicode

A.1 Scriptes Unicode

- Arabe
- Arménien
- Bengali
- Bopomofo
- Cherokee
- CJK
- Cyrillique
- Éthiopien
- Devenagari
- Gujarati
- Gurmukhi
- Géorgien
- Gothique
- Grec
- Gurmukhi
- Han
- Hangul Jamo
- Hébreu
- Hiragana
- Kannada
- Katakana
- Khmer
- Latin
- Lao
- Malayalam
- Mongol
- Myanmar
- Ogham
- Oriya
- Phonétique
- Runique
- Sinhala
- Syriaque
- Tamoul
- Telugu
- Thaana
- Thai
- Tibétain
- Yi

Tab. A.1 – Scriptes gérés par Unicode

A.2 Exemples de scriptes

Nous donnons dans les pages suivantes en exemple, les scriptes de quelques familles de langues.

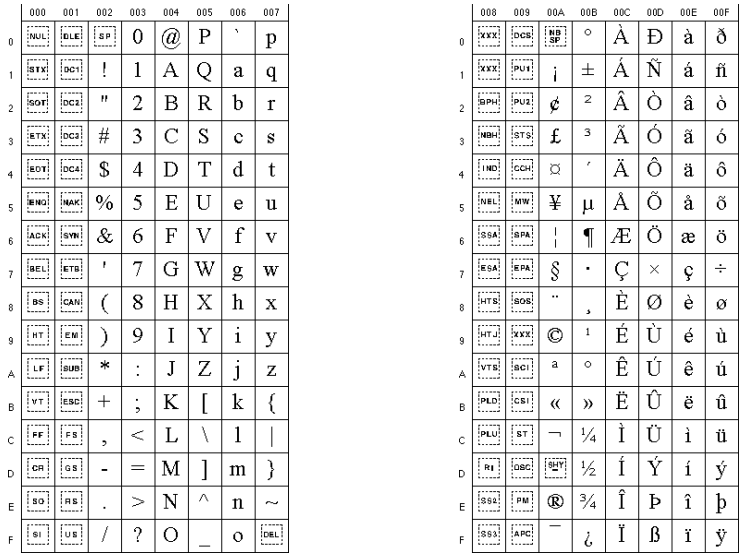


FIG. A.1 – Scriptes ASCII simple et étendu

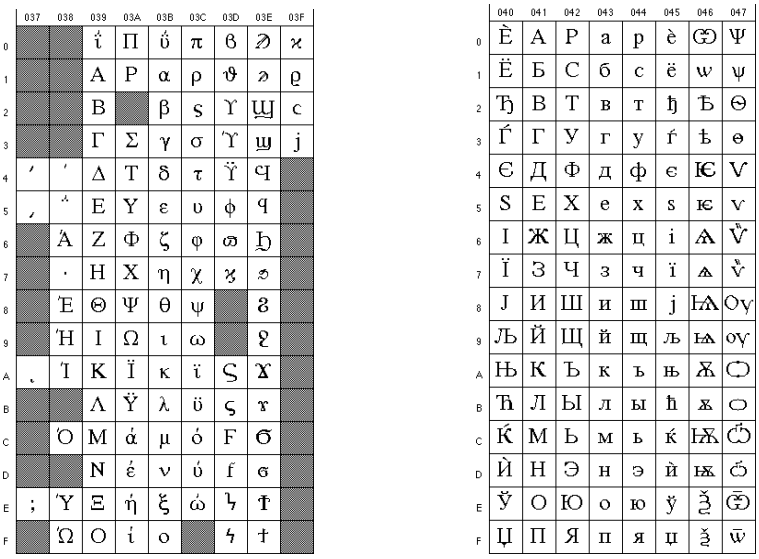


FIG. A.2 – Scriptes Grec et Cyrillique

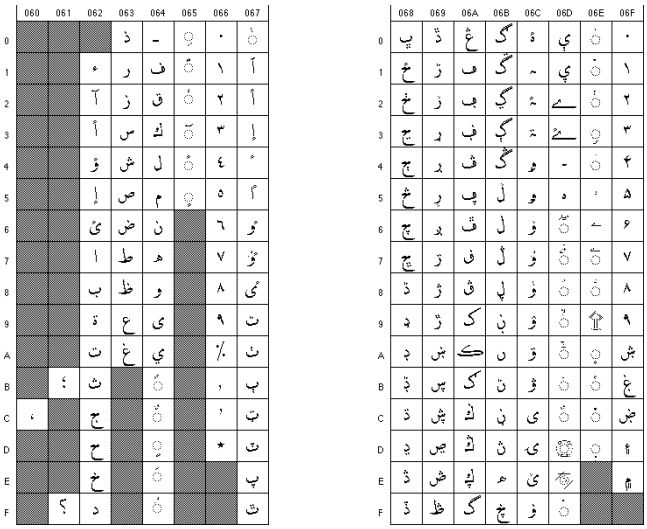


FIG. A.3 – Scriptes Arabes simple et étendus

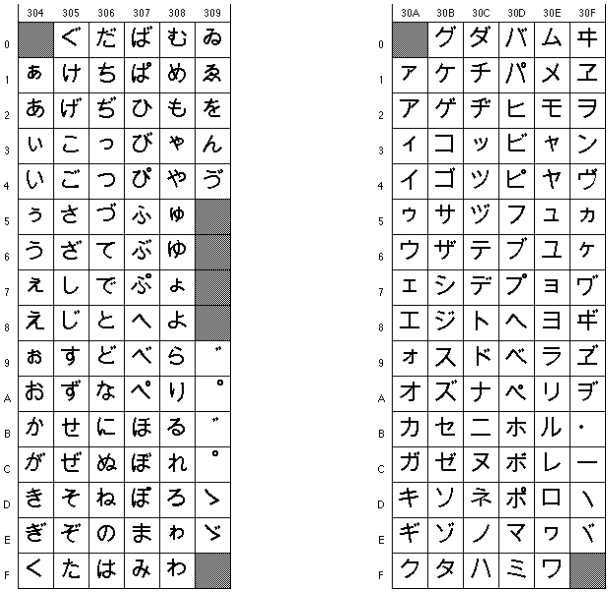


FIG. A.4 – Scriptes Japonais Hiragana et Katakana

Annexe B

Ressources Internet

Nous donnons ici une liste d'adresses internet qui concernent la compressions de données en générale mais aussi les ressources utilisées dans le cadre de cette thèse.

http ://www.internz.com/compression-pointers.html

http ://www.hn.is.uec.ac.jp/ arimura/compression_links.html

http ://sources.redhat.com/bzip2/index.html

ftp ://ftp.cl.cam.ac.uk/users/djw3/

ftp ://ftp.cs.waikato.ac.nz/pub/compression/ppm/

http ://corpus.canterbury.ac.nz/

ftp ://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/

http ://www.cs.brandeis.edu/ dcc/

Annexe C

Index des algorithmes

Nous récapitulons ici les principes de fontionnement des algorithmes étudiés. Les algorithmes dont les noms commencent par `CH` ou `Word` sont des algorithmes qui effectuent une lecture unique du texte et traitent les mots nouveaux par un système d'échappement. Les algorithmes dont les noms commencent par `Huff` utilisent le codage de Huffman et font appel à une prélecture du texte qui permet de construire un lexique des mots linguistiques et un lexique des mots séparateurs. La forme comprimée du texte comporte alors la forme comprimée des lexiques et la suite alternée des représentations de Huffman associées aux mots linguistiques ou séparateurs.

- **CH0** : Algorithme dû à Cormack et Horspool qui est fondé sur celui de Ziv et Lempel et utilise deux tables de facteurs. Une des tables contient les facteurs commençant par un mot linguistique, l'autre ceux commençant par un mot séparateur. Cet algorithme effectue un traitement dit « entrelacé » dans la mesure où les deux tables contiennent des mots linguistiques et des mots séparateurs.

- **CH1** : CH1 est une version de CH0 qui utilise un traitement « en parallèlle ». Il utilise une table pour les mots linguistiques et l'autre pour les mots séparateurs.

- **CH2** : CH2 reprend CH1 et améliore la compression des mots.

- **CH3** : CH3 reprend CH2 en utilisant le codage *Phased Binary* pour obtenir un codage plus compact des indices.

- **Word** : Est une implémentation très efficace et pratique du codage arithmétique sur les mots due à Moffat, Neal et Witten fondée sur une implémentation du calcul des fréquences cumulées due à Fenwick.

- **Word**$_{zl}$: Amélioration de la compression des mots nouveaux qui sont traités par Ziv et Lempel. Ici le code d'échappement n'est plus suivi par le nombre de caractères du mot, mais par le nombre de ses facteurs.

- **Word**$_{zl}^{Maj}$: Ajoute à Word$_{zl}$ une prédiction des mots qui commence par une majuscule en traitant un contexte spécial *Maj* les mots linguistiques précédés par des mots séparateurs comportant les caractères : point, virgule, ou point d'exclamation.

- **Huff_M** : Effectue un codage de Huffman à base de mots qui utilise un arbre de Huffman pour les mots linguistiques et un second pour les mots séparateurs.

- **Huff_M$^+$** : Ajoute à Huff_M un traitement particulier des majuscules qui consiste à mettre tous les mots en minuscules et à mémoriser les lettres majuscules par un code ajouté au mot séparateur qui précède le mot traité.

- **Huff_Co** : Fonctionne de la même manière que Huff_M mais utilise non seulement des mots mais aussi des suites de mots séparés par les caractères **espace** ou **apostrophe** qui sont construites dynamiquement à la compression comme à la décompression par une adaptation de l'algorithme de Ziv et Lempel.

- **Huff_Co$^+$** : Ajoute à Huff_M un traitement particulier des majuscules qui consiste à mettre tous les mots en minuscules et à mémoriser les lettres majuscules par un code ajouté au mot séparateur qui précède le mot traité.

- **Huff_Co$^+$** : Fonctionne comme Huff_Co et utilise le traitement des majuscules présent dans Huff_M$^+$.

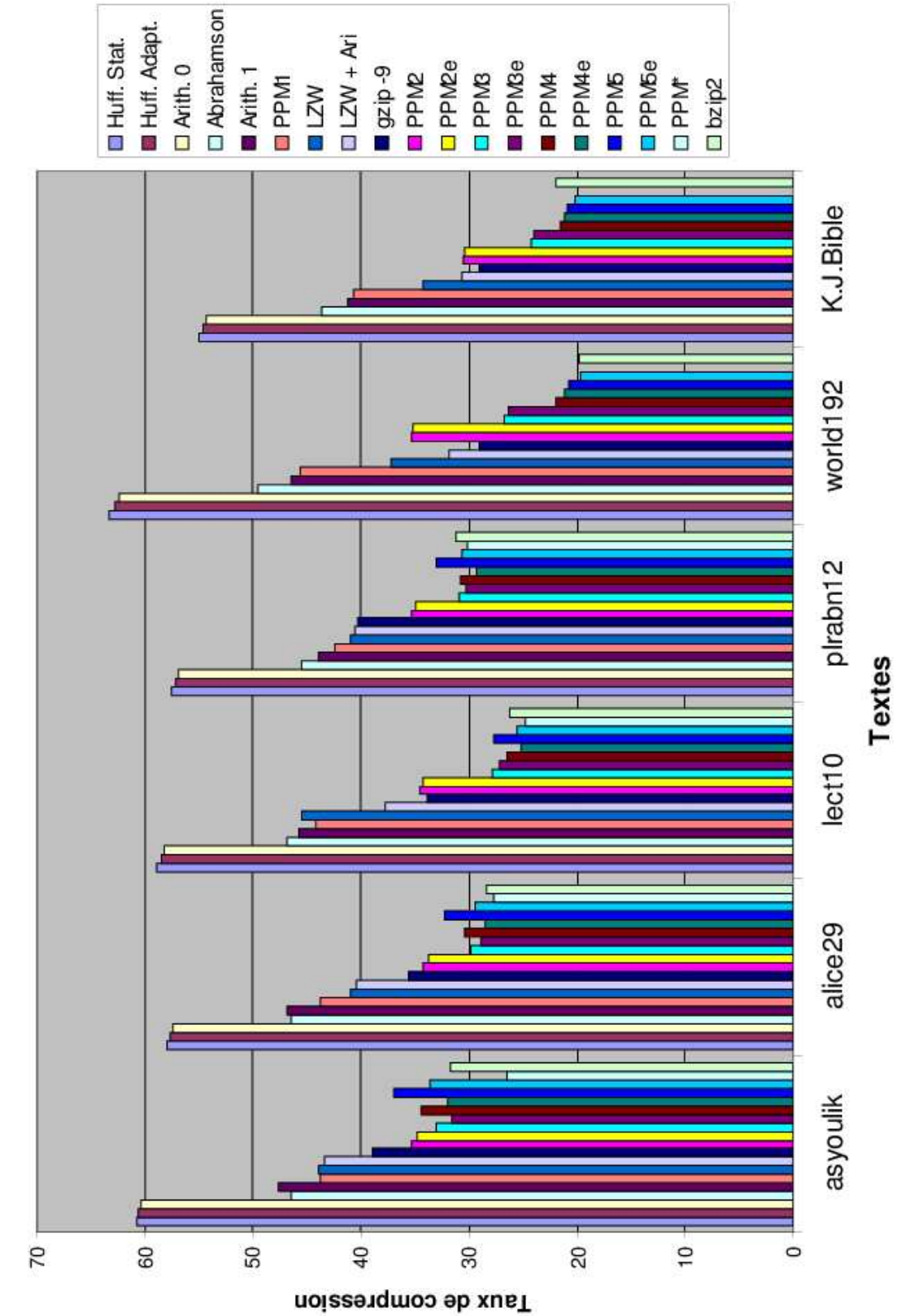
- **Huff_M*** : Fait appel à la notion de contexte linguistique qui utilise le fait que certains mots comme les mots grammaticaux opèrent une forte sélection sur la classe grammaticale des mots qui les suivent. Un contexte peut être formé d'un ou plusieurs mots. L'ensemble des mots successeurs d'un contexte sont traités par un arbre de Huffman exclusivement associé à ce contexte.

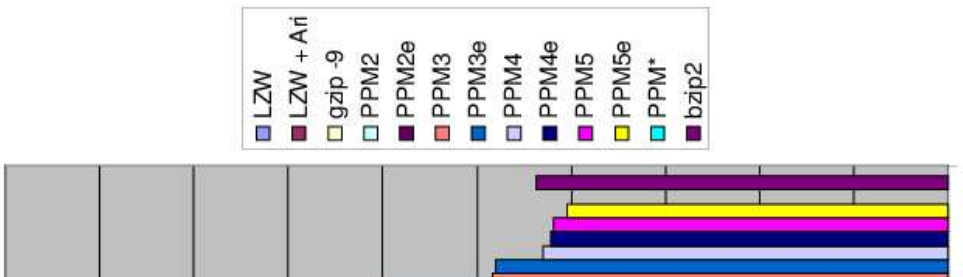
Annexe D

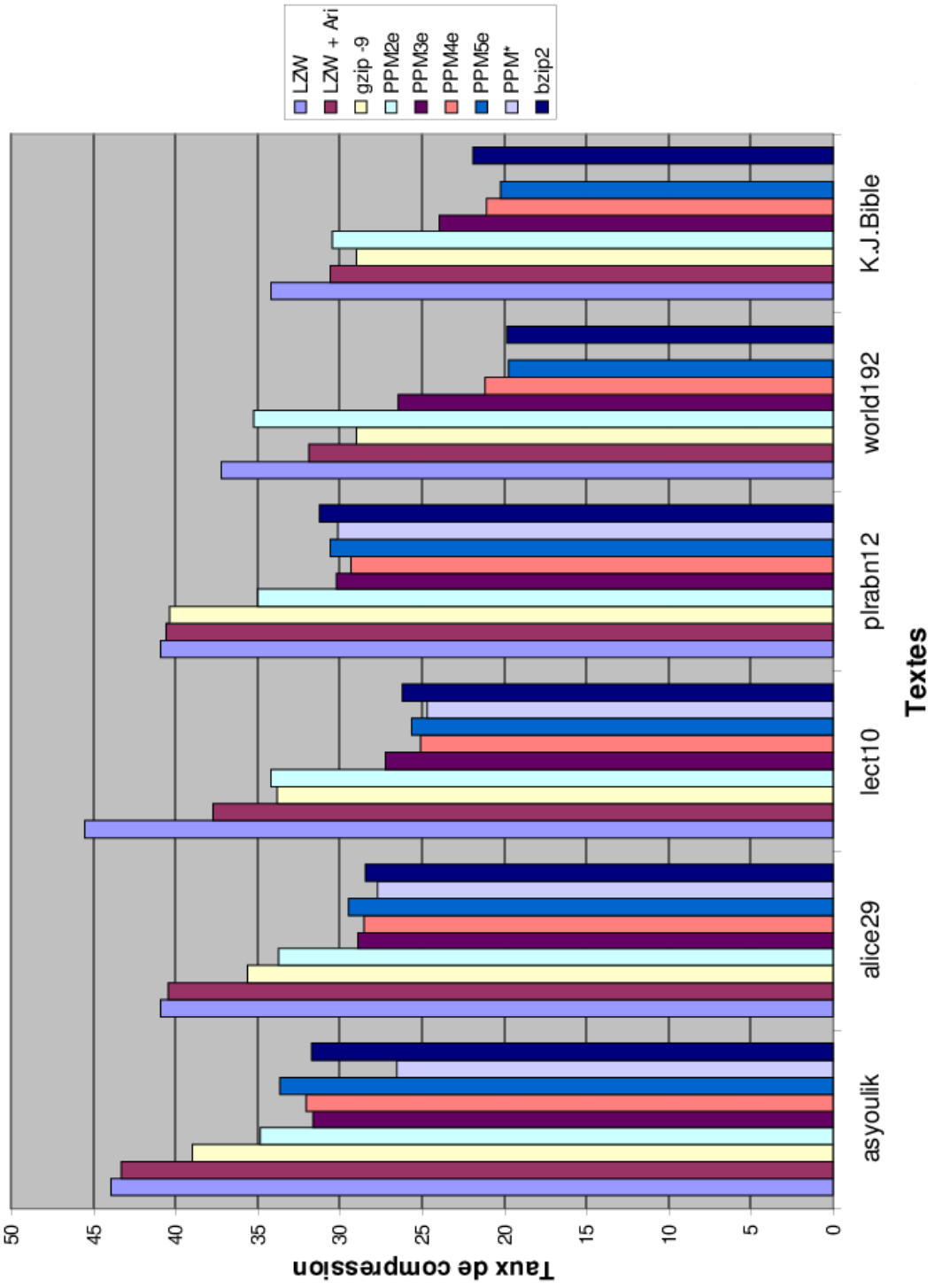
Histogrammes qui représentent les taux de compressions obtenus par chacun des algorithmes de compression sur les textes du Canterbury Corpus

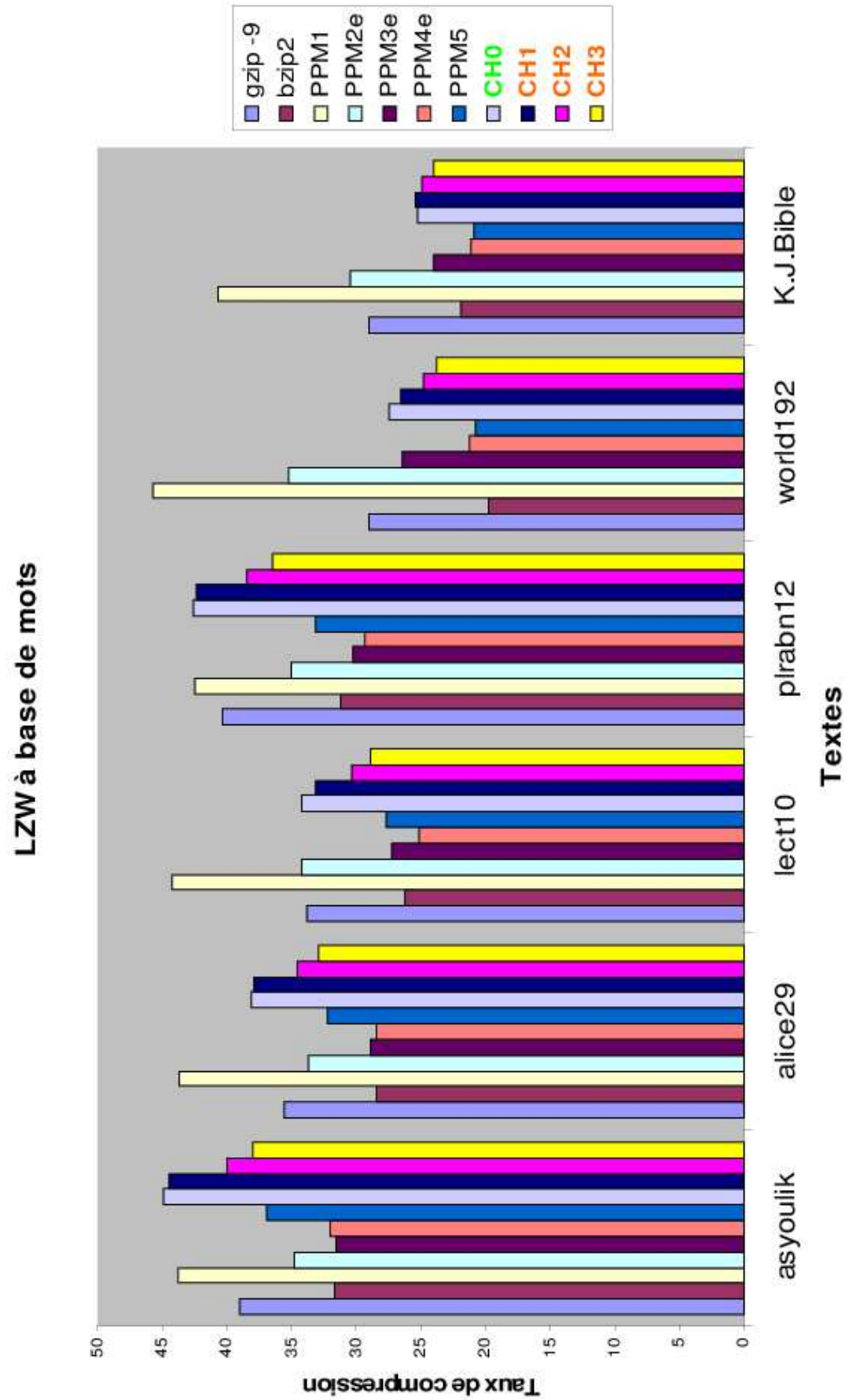
Nous donnons dans cette annexe les graphiques qui représentent pour chacun des six textes issus du Canterbury Corpus, les taux de compression obtenus pour l'ensemble des algorithmes présentés dans cette thèse.

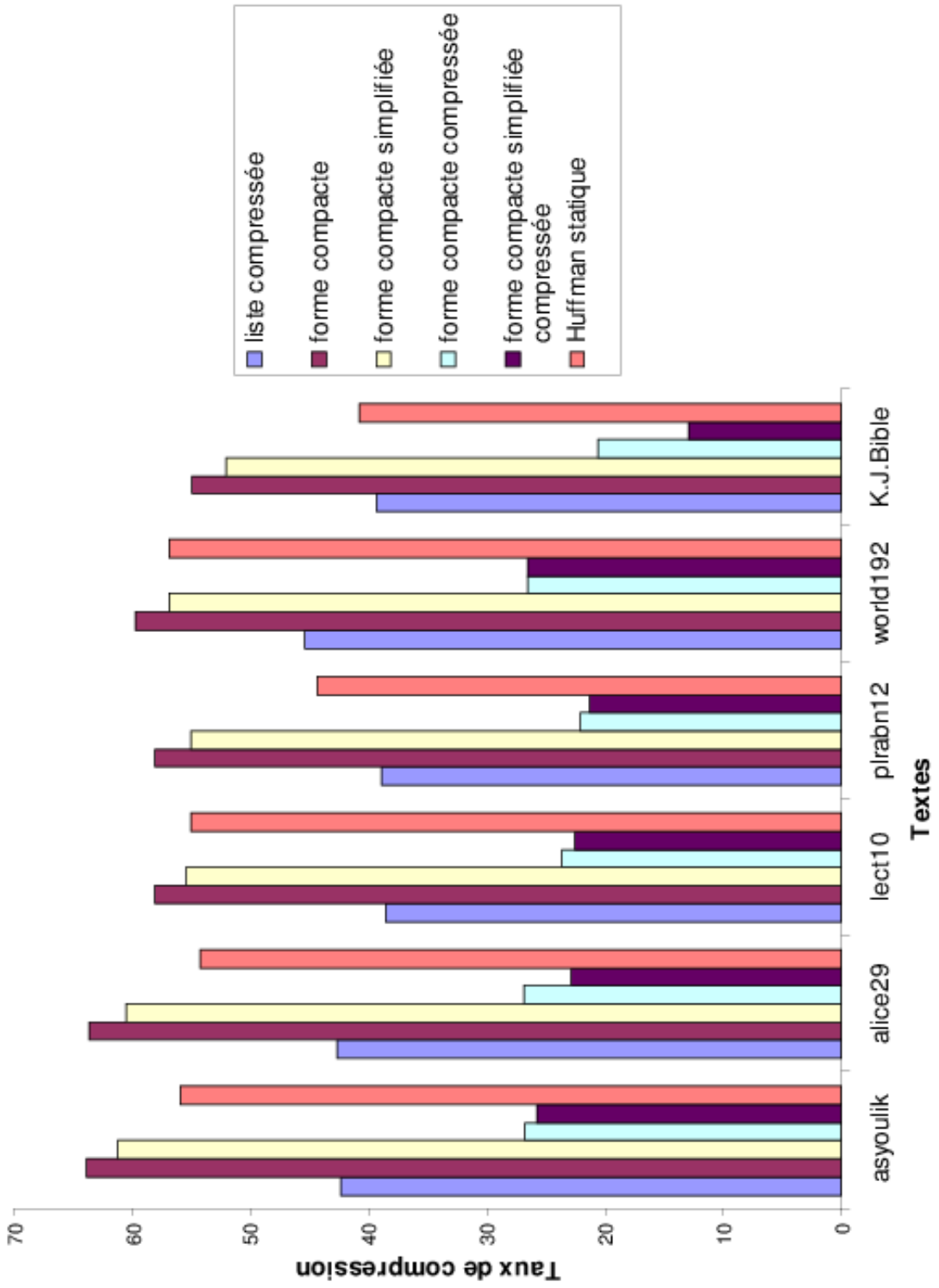
Pour chaque graphique, le taux de compression obtenu est indiqué en ordonnée. L'axe des abscisses comporte la série des textes traités par l'ensemble des algorithmes utilisés.

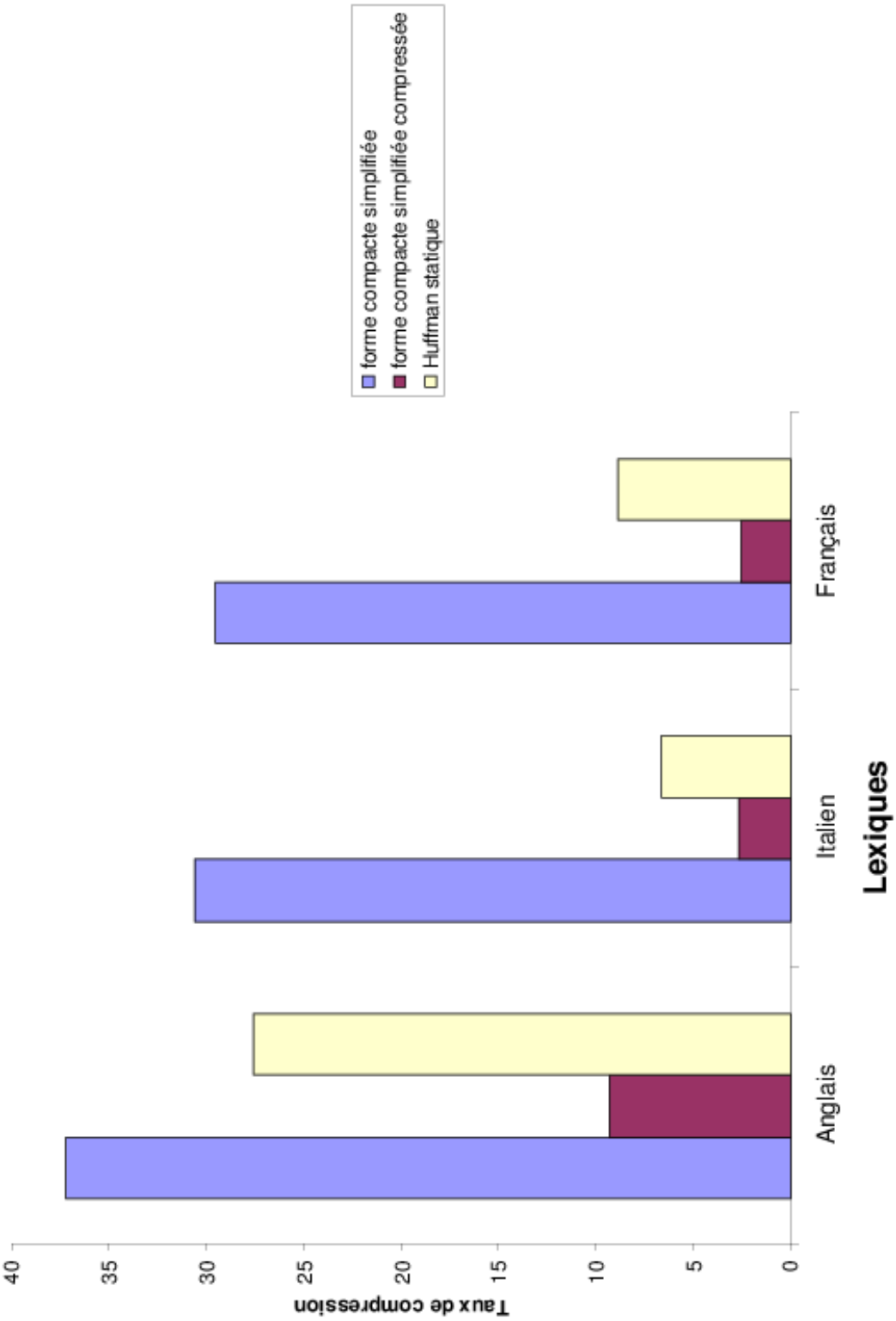

Huff. Stat.
Huff. Adapt.
Arith. 0
Abrahamson
Arith. 1
PPM1
LZW
LZW + Ari
gzip -9
PPM2
PPM2e
PPM3
PPM3e
PPM4
PPM4e
PPM5
PPM5e
PPM*
bzip2
Taux de compression
70
60
50
40
30
20
10
0
asyoulik
alice29
lect10
plrabn12
world192
K.J.Bible
Textes

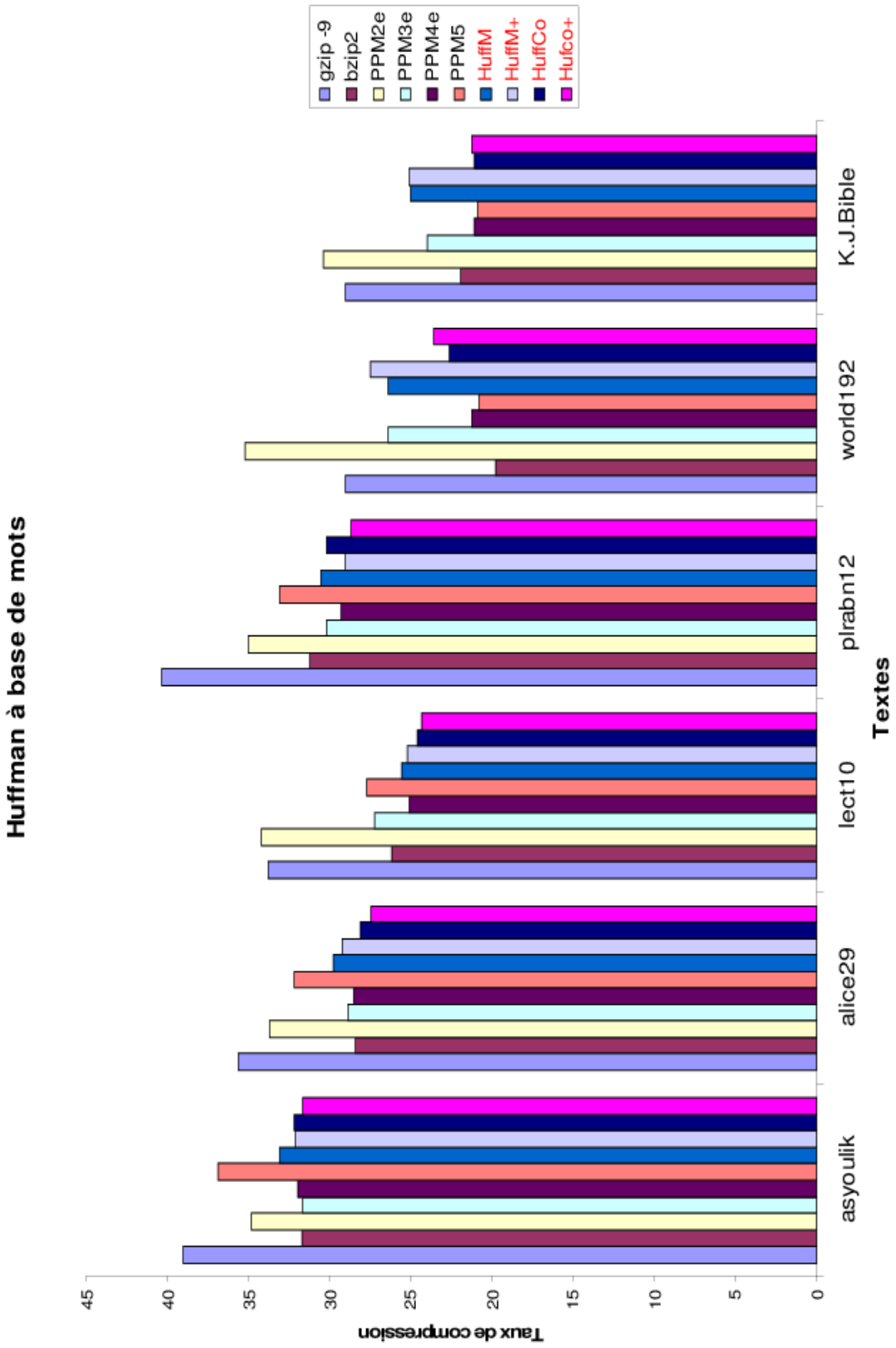

LZW
LZW + Ari
gzip -9
PPM2
PPM2e
PPM3
PPM3e
PPM4
PPM4e
PPM5
PPM5e
PPM*
bzip2
Bible

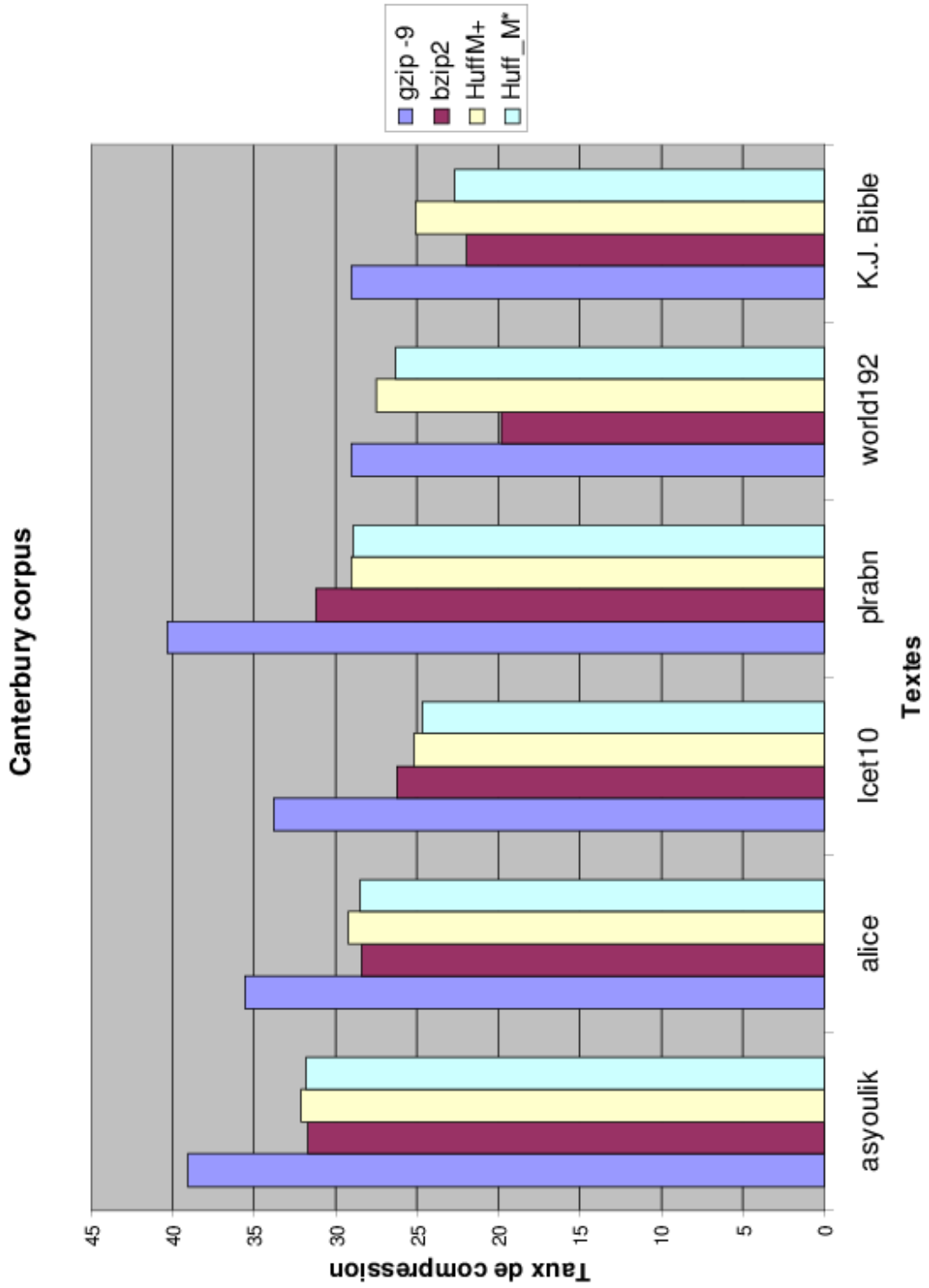

LZW
LZW + Ari
gzip -9
PPM2e
PPM3e
PPM4e
PPM5e
PPM*
bzip2
Taux de compression
0
5
10
15
20
25
30
35
40
45
50
asyoulik
alice29
lect10
plrabn12
world192
K.J.Bible
Textes

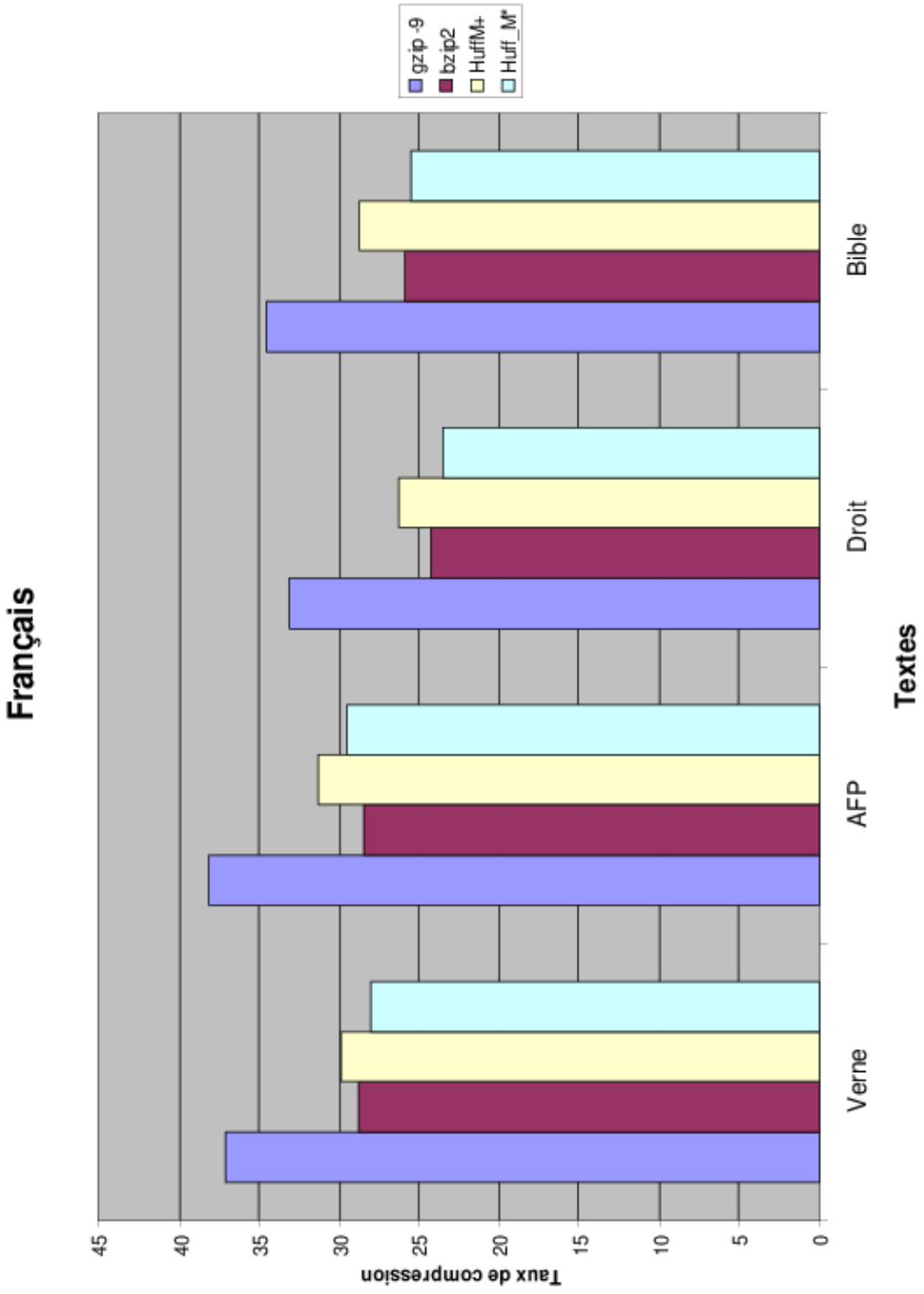

gzip -9
bzip2
PPM1
PPM2e
PPM3e
PPM4e
PPM5
CH0
CH1
CH2
CH3
LZW à base de mots
Taux de compression
50
45
40
35
30
25
20
15
10
5
0
asyoulik
alice29
lect10
plrabn12
world192
K.J.Bible
Textes

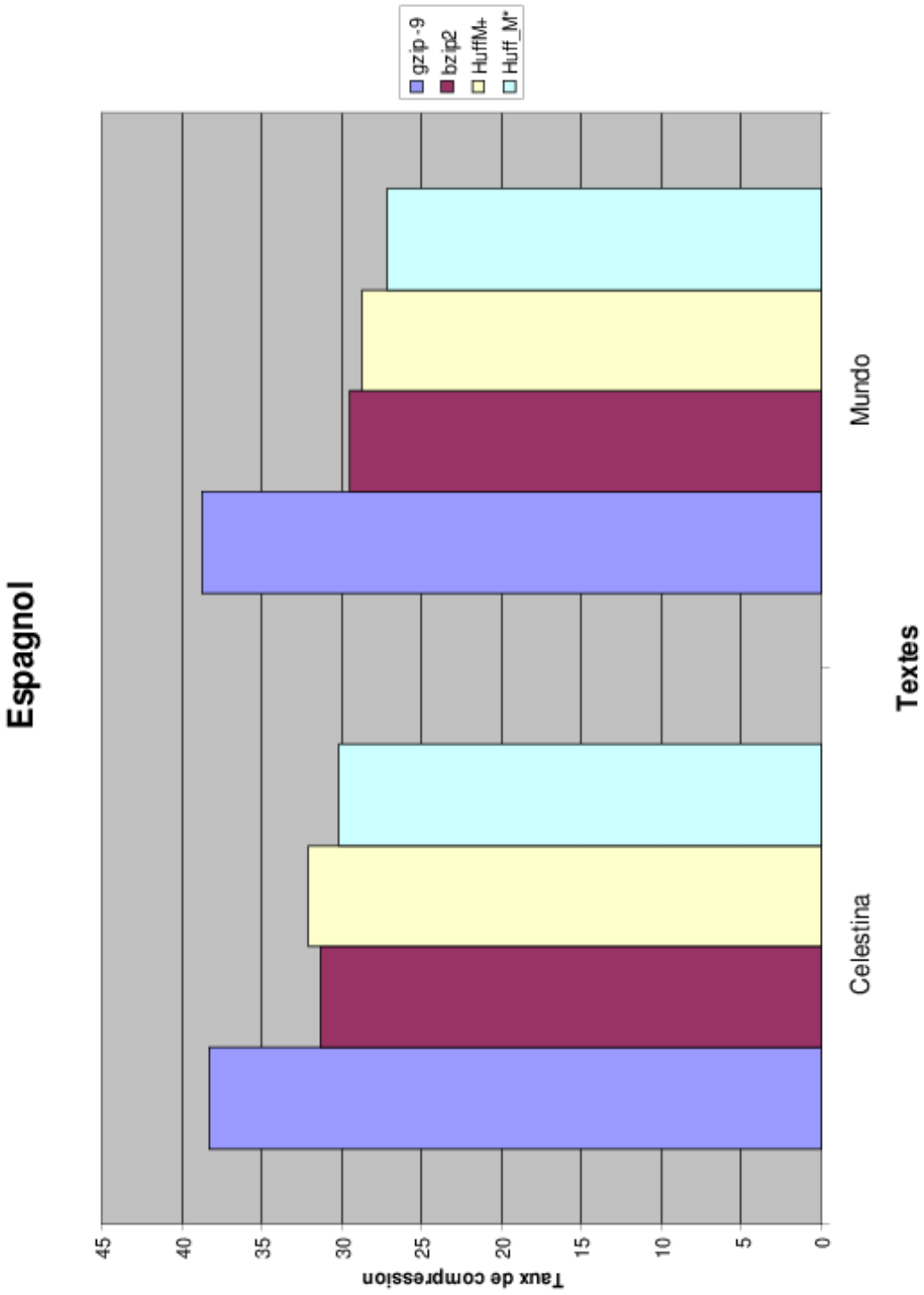

liste compressée
forme compacte
forme compacte simplifiée
forme compacte compressée
forme compacte simplifiée compressée
Huffman statique
Taux de compression
0
10
20
30
40
50
60
70
asyoulik
alice29
lect10
plrabn12
world192
K.J.Bible
Textes

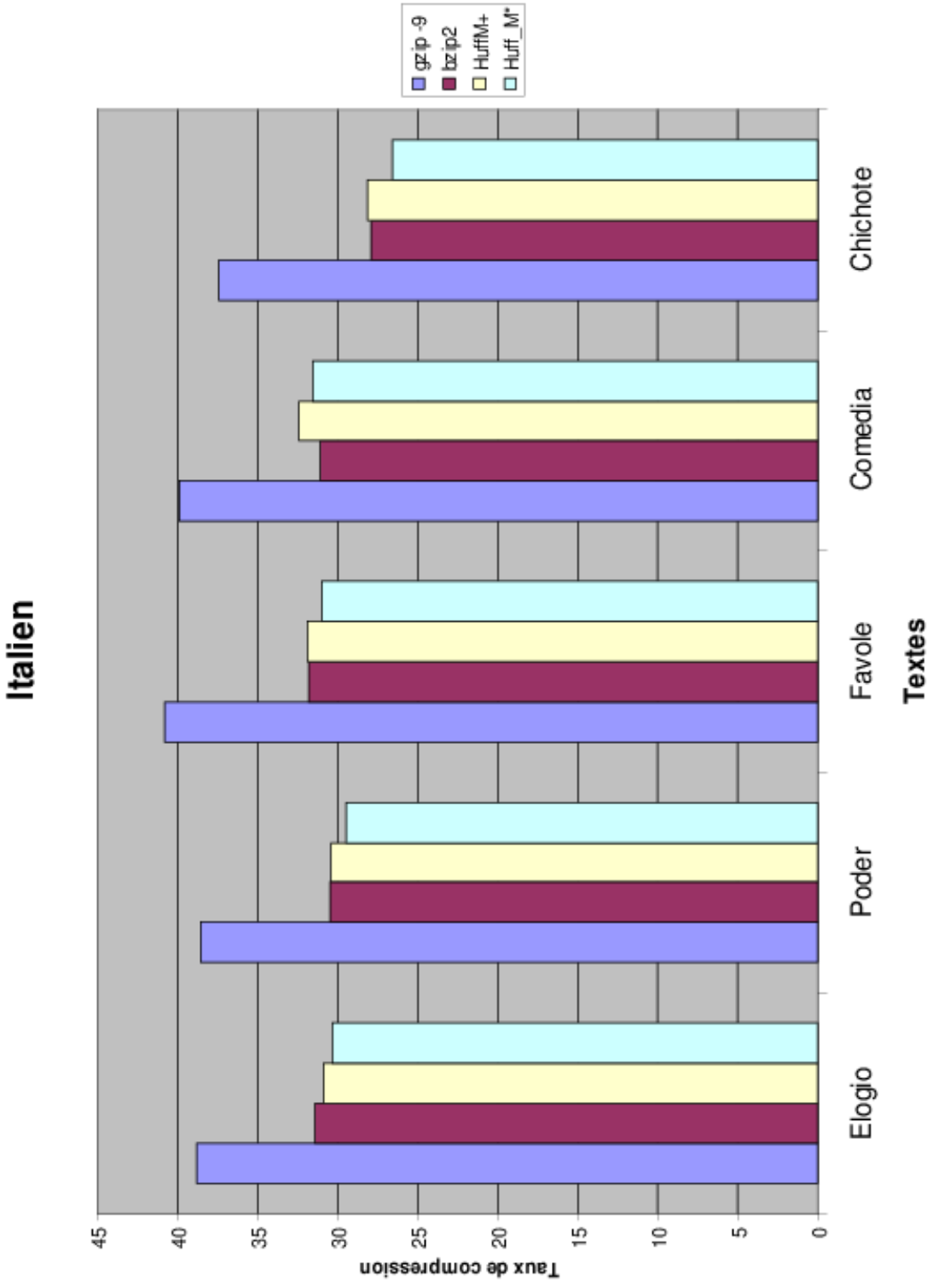

forme compacte simplifiée
forme compacte simplifiée compressée
Huffman statique
Taux de compression
0
5
10
15
20
25
30
35
40
Anglais
Italien
Français
Lexiques

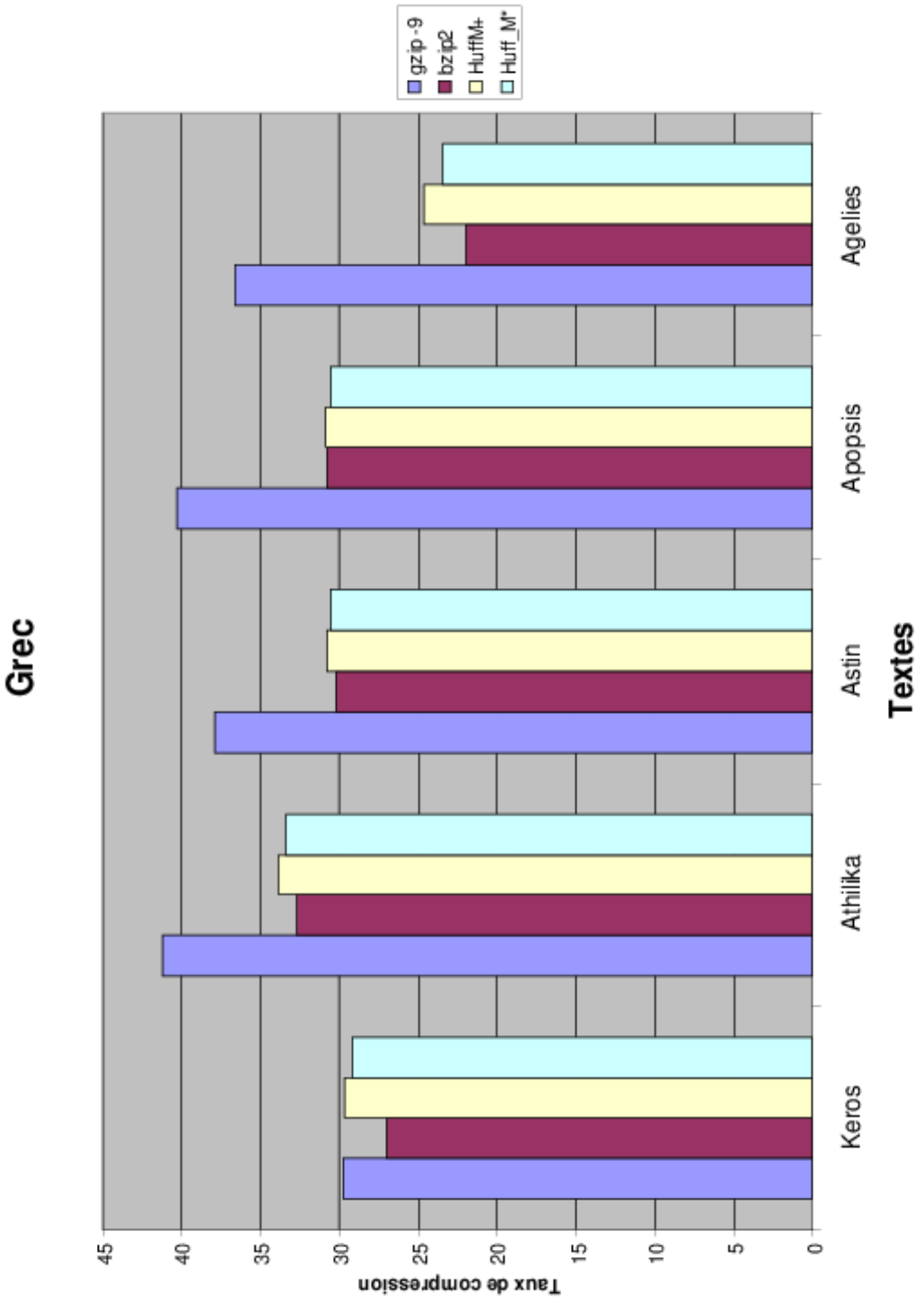

Huffman à base de mots
gzip -9
bzip2
PPM2e
PPM3e
PPM4e
PPM5
HuffM
HuffM+
HuffCo
Hufco+
asyoulik
alice29
lect10
plrabn12
world192
K.J.Bible
Textes
Taux de compression
0
5
10
15
20
25
30
35
40
45

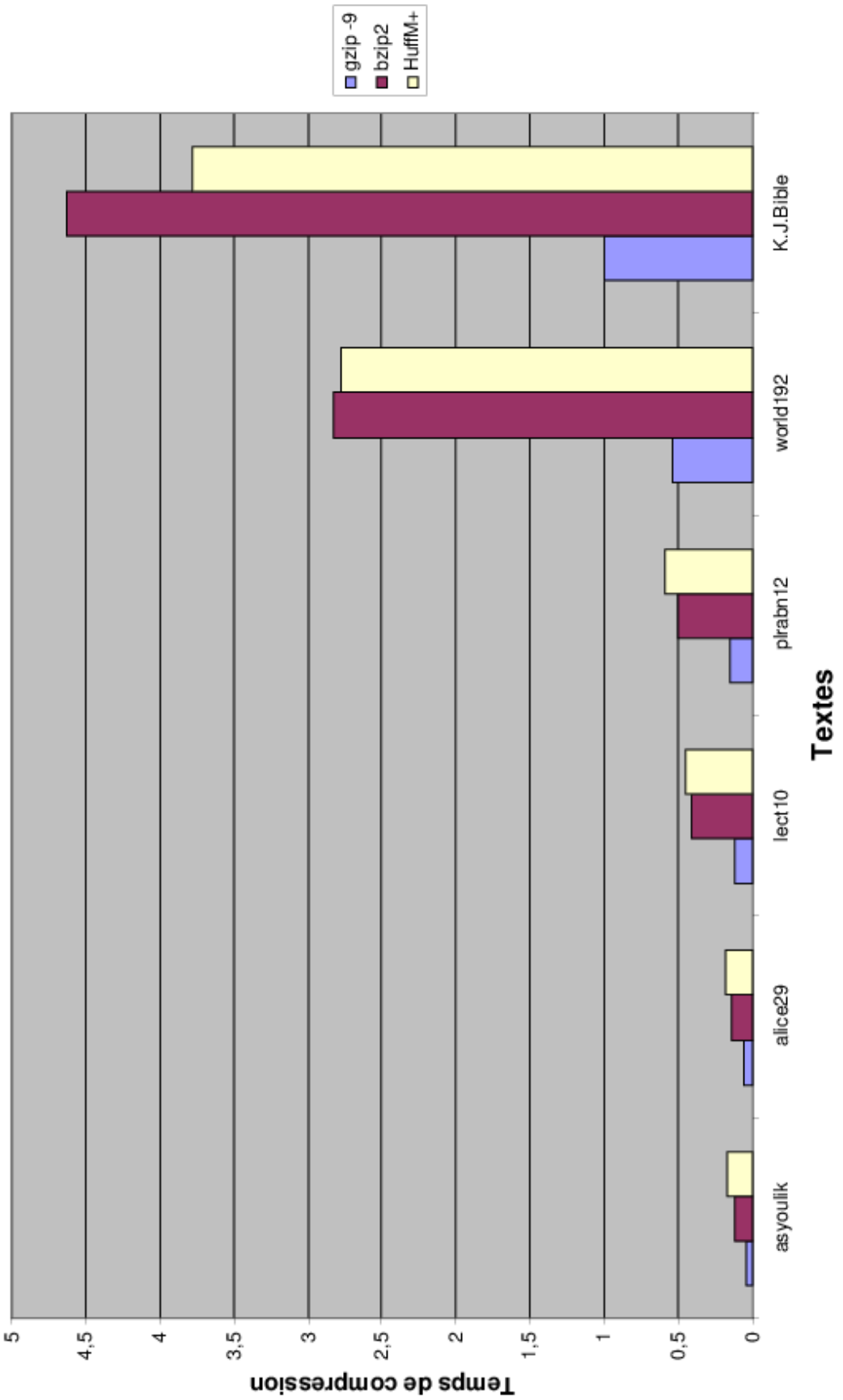

gzip -9
bzip2
HuffM+
Huff_M*
Canterbury corpus
Taux de compression
45
40
35
30
25
20
15
10
5
0
asyoulik
alice
lcet10
plrabn
world192
K.J. Bible
Textes


gzip -9
bzip2
HuffM+
Huff_M*
Français
Textes
Bible
Droit
AFP
Verne
Taux de compression
0
5
10
15
20
25
30
35
40
45


gzip -9
bzip2
HuffM+
Huff_M*
Espagnol
Mundo
Celestina
Textes
Taux de compression
0
5
10
15
20
25
30
35
40
45


Italien
gzip -9
bzip2
HuffM+
Huff_M*
Taux de compression
0
5
10
15
20
25
30
35
40
45
Elogio
Poder
Favole
Comedia
Chichote
Textes


Grec
gzip -9
bzip2
HuffM+
Huff_M*
Agelies
Apopsis
Astin
Athilika
Keros
Textes
Taux de compression
0
5
10
15
20
25
30
35
40
45


gzip -9
bzip2
HuffM+
K.J.Bible
world192
plrabn12
lect10
alice29
asyoulik
Textes
Temps de compression
0
0,5
1
1,5
2
2,5
3
3,5
4
4,5
5

Bibliographie

[1] D.M. Abrahamson. An adaptive dependency source model for data compression. *Commununications of the ACM*, 32(1) :77–83, january 1989.

[2] N. Abramson. *Information Theory and Coding*. Electronic Sciences Series. McGraw-Hill, Englewood Cilffs, New Jersey, 1963.

[3] Y. Bar-Ness and C. Peckman. Word-based data compression schemes. *IEEE.*, pages 300–303, 1989.

[4] D. Basu. Compression with several Huffman trees. *Data Compression Conference*, March 1991.

[5] J.C. Bell, T.C. Cleary and I.H. Witten. *Text Compression*. Advanced Reference Series. Prentice-Hall, Englewood Cilffs, New Jersey, 1990.

[6] J.L. Mc EIlroy M.D. Bentley. Engeeniring sorting function. *Software Practice and Experience*, 23(11) :1249–1265, November 1993.

[7] J.L. Sleator D.D Tarjan Bentley and Wei V.K. A locally adaptative data compression scheme. *Communications of ACM*, 29(4) :320–330, Apr 1986.

[8] J.L. Sleator D.D Tarjan Bentley and Wei V.K. A locally adaptative data compression scheme. *Communications of ACM*, 29(4) :320–330, Apr 1986.

[9] Bred. Burrows. *Communications of ACM*, 29(4) :320–330, Apr 1986.

[10] D.J. Wheeler Burrows, M. A block-sorting lossless data compression algorithme. Technical Report 124, Digital Equipement Corporation, May 1994.

[11] J.G. Cleary and W.J. Teahan. *IEEE Trans. Comm.*, it24(5) :396–402, March 1997.

[12] J.G. Cleary and I.H. Witten. Data Compression Using Adaptative Coding and Partial Matching. *IEEE Trans. Comm.*, it24(5) :396–402, March 1984.

[13] W.J. Cleary, J.G. Teahan and I.H. Witten. Unbounded Lenth Contexts for PPM. *Data Compression Conference*, March 1995.

[14] J.B. Connell. A Huffman-Shannon-Fanno code. *Data Compression Conference*, 61 :1046–1047, July 1973.

[15] G.V. Cormack and R.N. Horspool. Constructing Word-based Text Compression Algorithms. *The computer journal*, 30(6) :541–550, 1987.

[16] G.V. Cormack and R.N. Horspool. Constructing Word-based Text Compression Algorithms. *Data Compression Conference*, March 1992.

[17] C.E. Rivest R.L. Cormen, T.H. Leiserson. *Introduction to Algorithms*. MIT Press, 1998.

[18] M. Crochemore and R. Rytter. *Text Algoritms*. Oxford University Press, 1994.

[19] N. Faller. An adaptative system for data compressio. *IEEE Transaction on Information Theory*, it24(5) :668–674, November 1973.

[20] A. Fenwick. A new data structure for cumulative frequency tables. *Software-Practice and Experience*, 24(3) :327–336, 1994.

[21] A. Fenwick. A new data structure for cumulative frequency tables. *Software-Practice and Experience*, 24(3) :327–336, 1994.

[22] J.-L Gailly. Documentation for gzip. *IEEE Transaction on Information Theory*, August 1993.

[23] R.G Gallager. Variations on a theme by Huffman. *IEEE Transaction on Information Theory*, it24(5) :668–674, November 1978.

[24] M. Guazzo. A general minimum-redudancy source coding algorithm. *IEEE Transaction on Information Theory*, it26(1) :15–25, 1980.

[25] R.W. Hamming. *Coding and Information Theory*. Advanced Reference Series. Prentice-Hall, Englewood Cilffs, New Jersey, 1980.

[26] D. Hirschberg and D. Lelewer. Data Compression. *Computing Surveys*, 19 :261–296, September 1987.

[27] D.S. Hirschberg and D.A. Lelewer. Efficient decoding of prefix codes. *Commununications of the ACM*, 33 :449–459, April 1990.

[28] R.N. Horspool. Improving lzw. *Data Compression Conference*, March 1991.

[29] D.A. Huffman. A method for the construction on minimum redundancy codes. *Proc. IRE*, 40 :1098–1101, 1952.

[30] Thomas S.W. McKie J. Davies S. Turkovsky K. Wood J.A. and Orost J.W. Compress (version 4.0) program and documentation. 1985.

[31] J. Jiang and Jones S. Word-based dynamic algorithms for data compression. *IEE. Proceedings*, 139(6) :582–586, 1992.

[32] A. Katajainen, J. Moffat and A. Turpin. A fast and space-economical algorithm for length-limited coding. *Data Compression Conference*, pages 393–402, March 1996.

[33] A. Katajainen, J. Moffat and A. Turpin. A fast and space-economical algorithm for length-limited coding. *Data Compression Conference*, pages 393–402, March 1996.

[34] Shmuel T. Klein. Space and Time-Efficient Decoding with Canonical Huffman Trees. *CPM*, pages 65–75, 1997.

[35] Knuth. Dynamic Huffman coding. *Journal od Algorithms*, pages 163–180, 1985.

[36] A. Lempel and J. Ziv. Universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3) :337–343, may 1977.

[37] A. Lempel and J. Ziv. Compression of individual sequences via variable rate coding. *IEEE Transactions on Information Theory*, 24(5) :530–536, september 1978.

[38] C. Liu and C. Yu. Data compression using word encoding with Huffman codes. *Journal of the American Society for Information Science*, pages 685–697, october 1991.

[39] U. Manber. A text compression scheme that allows fast searching in the compressed file. *Software-Practice and Experience*, 19(2) :11–124, 1993.

[40] V.S. Miller and Wegman M.N. Variations on a theme by Ziv and Lempel. *Combinatorial algorithms on words*, F12 :131–140, 1984.

[41] A. Moffat. Word-based text compression. *Software-Practice and Experience*, 19(2) :185–198, 1989.

[42] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11) :1917–1921, 1990.

[43] R.M. Moffat, A. Neal and I. H. Witten. Arithmetic coding revisited. *Data Compression Conference*, pages 202–211, 1995.

[44] J. Moffat A. Katajainen. In-place calculation of the minimum redudancy codes. *Data Compression Conference*, pages 393–402, March 1996.

[45] A. Moffat A. Turpin. On the implementation of the minimum redudancy prefix codes. *Data Compression Conference*, pages 170–179, March 1996.

[46] J. Moffat A. Zobel. Word-based text compression. *Software-Practice and Experience*, 25(8) :891–903, August 1995.

[47] M. Nelson. *The Data Compression Book*. M&T Books, 1992.

[48] E. Norwood. The number of different possibles compact codes and its application to efficient codin and decoding. *IEEE Transaction on Information Theory*, pages 613–616, october 1967.

[49] D. Revuz. *Dictionnaires*. PhD thesis, Université Paris 7, March 1989.

[50] D. Revuz and M. Zipstein. Dz an universal compression algorithm specialized in text. *Data Compression Conference*, March 1992.

[51] J. Rissanen. D. *D*, March 196.

[52] J. Langdon G.G Rissanen. Arithmetic coding. *IBM Journal of Reasearch and Developpement*, 29(2) :149–162, March 1979.

[53] E. Roches. *Analyse syntaxique*. PhD thesis, Université Paris 7, 1993. Soutenue le 3 de'cembre 1993.

[54] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, San Francisco, California, 1996.

[55] E.S. Schwartz. An optimum encoding with minimum longest code and total number of digits. *Commununications of the ACM*, 7 :166–169, 1964.

[56] E.S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Commununications of the ACM*, 7 :166–169, 1964.

[57] Julian Seward. Documentation for bzip2. *http ://sourceware.cygnus.com/bzip2/*, 2000.

[58] C.E. Shannon. Prediction and entropy of printed english. *Bell Systeme Technical Journal*, pages 379–423 et 623– 656, 1951.

[59] C.E. Weaver W. Shannon. A mathematical theory of communication. *Bell Systeme Technical Journal*, 1948.

[60] A. Sieminski. Fast decoding of the Huffman codes. *Information Processing Letters*, 26 :237–241, January 1988.

[61] H. Tanaka. Data structure of Huffman codes and its application to efficient codin and decoding. *IEEE Transaction on Information Theory*, it33(1) :155–160, 1987.

[62] Daniel Taupin. Musixdoc.ps. *http ://www.gmd.de/Misc/Music/musixtex/*, 2001.

[63] W.J. Teahan. *Modelling English Text*. PhD thesis, University of Waikato, 1998.

[64] P. Tischer. A modified Lempel-Ziv-Welch data compression scheme. *Australian Computer Science*, 9(1) :261–272, 1987.

[65] J.S. Vitter. A technique for high-performance data compression. *Journal of ACM*, 34(4) :825–845, 1985.

[66] C. Walshaw. Format ABC. *http ://www.gre.ac.uk/ c.walshaw/*, 2001.

[67] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6) :8–19, 1984.

[68] I. H. Moffat A. Witten and T.C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Advanced Reference Series. Van Nostrand Reinhold, New York, 1994.

[69] J.C. Witten, I.H. Cleary. The zero frequency problem : Estimating the probabilities of novel events in adaptative text compression. *IEEE Transaction on Information Theory*, 37(4) :1085–1094, July 1991.

[70] R.M. Witten, I. H. Neal and J.C. Cleary. Arithmetic coding for data compression. *Commununications of the ACM*, 30(6) :520–540, 1987.

[71] M. Zipstein. *Les methodes de compression de texte algorithmes et performances*. PhD thesis, Université Paris 7, 1990. Soutenue le 3 de'cembre 1993.

Index