Développement d’une nouvelle technique de compression pour les codes variables à fixes quasi-instantanés

Mémoire

Fatma Haddad

Maîtrise en informatique
Maître ès sciences (M.Sc.)

Québec, Canada

© Fatma Haddad, 2017

Développement d'une nouvelle technique de compression pour les codes variables à fixes quasi-instantanés

Mémoire

Fatma Haddad

Sous la direction de:

Danny Dubé, directeur de recherche

Résumé

Pas toutes les techniques de compression des données adoptent le principe de dictionnaire pour représenter ses mots de code. Le dictionnaire est un ensemble de mots de code associés aux symboles sources lors de l'opération d'encodage. La correspondance entre le mot de code et le symbole source dépend de l'algorithme de compression adopté.

Généralement, chaque algorithme construit son dictionnaire selon un ensemble de propriétés. Parmi ces propriétés nous avons celle de préfixe. Elle est primordiale pour les codes de type fixe à variable (FV) tels que l'algorithme de Huffman et celui de Shannon-Fano. Par contre, la propriété préfixe est optionnelle pour les codes de longueur variable à fixe (VF). Donc cela peut causer le but de pouvoir construire un dictionnaire plus performant, comme le cas des codes quasi-instantanés.

Dans cette optique, Yamamoto et Yokoo ont éliminé cette condition pour créer un dictionnaire meilleur que celui de Tunstall. Les dictionnaires proposés par Yamamoto et Yokoo sont appelés **les codes VF quasi-instantanés** ou en anglais **almost instantaneous VF codes**. En s'appuyant sur leurs contributions, nous avons déduit que leur technique peut fournir dans certains cas des codes variables à fixes sous-optimaux, d'où notre suggestion de correctifs à leurs algorithmes pour en améliorer l'efficacité. Aussi nous proposons un autre mécanisme pour construire des codes VF en utilisant le principe de la programmation dynamique.

Abstract

Various techniques of data compression use a dictionary to represent their codewords. A dictionary is a set of codewords associated with the source symbols during the encoding operation. The correspondence between the codeword and the symbol source depends on the compression algorithm.

Usually, the prefix property is key for the fixed-to-variable type codes FV as demonstrated in the Huffman and the Shannon-Fano algorithms. However, such a property may be eliminated for fixed-length codes in order to build a more efficient dictionary. In this context, Yamamoto and Yokoo excluded this condition to create a dictionary better than Tunstall. This new dictionary is called **instantaneous variable-to-fixed code**.

Based on their contributions, we have deduced that their technique can provide, in some cases, suboptimal variable-to-fixed codes. Hence, we suggested to improve their algorithms. Also, we proposed another mechanism for building optimal AIVF codes by adopting the principle of dynamic programming.

Table des matières

Résumé **iii**

Abstract **iv**

Table des matières **v**

Liste des tableaux **vii**

Liste des figures **viii**

Dédicace **x**

Remerciements **xi**

1 Introduction **1**

2 Présentation générale **5**
- 2.1 Introduction 5
- 2.2 Les concepts généraux 5
- 2.3 La terminologie 7
- 2.4 Les types de compression 9
- 2.5 Les phases de la compression 10
- 2.6 Les techniques de compression sans perte 10
- 2.7 Conclusion 24

3 Étude préliminaire **25**
- 3.1 Introduction 25
- 3.2 Le code de Tunstall 25
- 3.3 Les codes variables à fixes quasi-instantanés AIVF 29
- 3.4 État de l'art 36
- 3.5 Conclusion 38

4 Amélioration de l'encodage variable à fixe **39**
- 4.1 Introduction 39
- 4.2 Les défauts de construction du dictionnaire de YY 39
- 4.3 La technique de la programmation dynamique en mode multi-arbre 44
- 4.4 Étude comparative 47
- 4.5 Conclusion 50

5 Conclusion **52**

A Les tests de mise en application **54**
A.1 Tunstall 54
A.2 Le mode mono-arbre 55
A.3 Le mode multi-arbre 56

Bibliographie **57**

Liste des tableaux

2.1 Un exemple du mécanise de bit de parité 6
2.2 Fréquence d'apparition des symboles de l'ensemble A 12
2.3 Exemple de code généré par la technique de Huffman [18] 13
2.4 Probabilité d'apparition des symboles 14
2.5 Les bornes des intervalles pour la chaîne M=abca 15
2.6 L'encodage de la chaîne aabbaabcaabbacb en utilisant le principe de LZ77 . . 17
2.7 L'encodage de la chaîne aabbaabcaabbac en utilisant le principe de LZ78 . . . 19
2.8 Les contextes d'ordre (-1, 0, 1, 2 et 3) pour l'alphabet A 20
2.9 Matrice de décalage cyclique pour la chaîne compression 22
2.10 Tri lexicographique de la chaîne compression 22
2.11 L'encodage du $L = nrsoocimpes$ en utilisant le MTF 23

3.1 Le dictionnaire de Tunstall 28
3.2 Le dictionnaire AIVF en mode mono-arbre T_0 34
3.3 Le dictionnaire AIVF en mode multi-arbre T_1 36

Liste des figures

1.1 Statistiques d'utilisation d'Internet et des réseaux sociaux dans le monde [4] . 1
1.2 Croissance estimée du volume mondial des données entre 2013 et 2020 [2] . . 2
1.3 Les types de code 3

2.1 Les éléments d'un arbre 7
2.2 Arbre d'alphabet ternaire complet et arbre d'alphabet ternaire incomplet . . . 8
2.3 L'arbre de Huffman pour l'ensemble A 12
2.4 L'encodage du message M=abca sous forme graphique 16

3.1 L'algorithme de Tunstall où M=7 28
3.2 L'arbre T_0 généré par l'algorithme de Yamamoto et Yokoo [31] 33
3.3 l'arbre T_1 généré par l'algorithme de Yamamoto et Yokoo [31] 35

4.1 Le problème de la racine complète en mode multi-arbre 40
4.2 T_0 en mode mono-arbre selon la technique de Yamamoto et Yokoo [31] 42
4.3 T_0 en mode mono-arbre selon le correctif de DH 43
4.4 Un arbre T_i^N où $i \leq A-2$, $N \geq 2$, $L \geq 1$, $R \geq 1$ et $L+R=N$ 45
4.5 L'arbre T_i^1 pour $i \leq A-2$ 46
4.6 L'arbre T_{A-1}^N 46
4.7 Construction de dictionnaire en utilisant la programmation dynamique . . . 47
4.8 Problème d'une racine complète 48
4.9 Problème de l'exécution complète de l'option II 49
4.10 Construction de l'arbre en mode multi-arbre en utilisant le principe de la programmation dynamique 50

Liste des abréviations

AIFV	Almost Instantaneous Fixed-to-Variable codes ou les codes FV quasi-instantanés
AIVF	Almost Instantaneous Variable-to-Fixe ou les codes VF quasi-instantanés
BWT	Transformée de Burrows-Wheeler ou Burrows-Wheeler Transform
DH	Dubé et Haddad
FF	Encodage Fixe à Fixe
FV	Encodage Fixe à Variable
LZ77	Technique de compression proposée par Lempel et Ziv en 1977
LZ78	Technique de compression proposée par Lempel et Ziv en 1978
LZW	Technique de Lempel-Ziv-Welch proposée par Terry Welch en 1984
MTF	Technique de Move To Front
PPM	Encodage par modélisation de contexte ou Prediction by Partial Matching
VF	Encodage Variable à Fixe
VV	Encodage Variable à Variable
YY	Yamamoto et Yokoo

Dédicace

À mon Père

Je lui dédie ce modeste travail en témoignage de mon grand amour et ma gratitude infinie.

À ma sœur Hela et ma chère amie Salma

Pour le bonheur qu'elles m'apportent, en témoignage de mes sincères reconnaissances pour les efforts qu'elles ont consentis pour l'accomplissement de mes études.

À Marwen Mida et Seifallah Guedouri

En témoignage de l'amitié sincère qui nous a liés et des moments inoubliables passés ensemble. Je vous dédie ce travail en vous souhaitant un avenir prometteur et plein de bonnes promesses.

À tous mes amis

Je vous dirais tout simplement, un grand merci, je vous aime.

FATMA

Remerciements

Il me parait opportun de commencer ce mémoire par des remerciements, à tous ceux qui ont, de près ou de loin, contribué à l'élaboration de ce travail et même à ceux qui ont eu la gentillesse de faire de cette maitrise un moment très profitable.

Je tiens à exprimer toute ma reconnaissance à Prof. **Danny Dubé**, mon directeur de recherche à l'Université Laval, pour son aide, sa disponibilité et son assistance qui ont permis d'assurer le bon déroulement de cette maitrise, aussi pour ses conseils précieux qui ont été d'une très grande qualité et d'un grand réconfort. Merci infiniment **Danny Dubé**.

Mes sincères remerciements s'adressent à Mme **Denise Dubeau** pour l'aide qu'elle m'a accordée durant les moments difficiles et aussi pour tous ses encouragements immenses qu'elle a consentis. Merci infiniment **Denise Dubeau**.

Par la même occasion, je tiens à remercier mes collègues dans le laboratoire de CoDificaTIon compression de données et théorie de l'information **Ahmad Al-Rababa'a**, **Mounir Mechqrane** et **Gabriel Dion-Bouchard** qui ont contribué énormément au bon déroulement de ma maitrise.

Mes vifs et respectueux remerciements s'adressent aussi aux membres du jury pour avoir accepté d'évaluer ce travail.

Chapitre 1

Introduction

Ces dernières décennies, l'Internet est devenu un outil inévitable et nous assistons à son essor phénoménal. En effet, ses utilisateurs se chiffrent en milliards dans un nombre grandissant de pays du monde. Selon un rapport publié en 2015 par l'agence marketing We Are Social, sur une population mondiale de 7.21 milliards de personnes, 3.010 milliards ont un accès à Internet [4]. We Are Social a confirmé aussi que le pourcentage des internautes au niveau mondial était estimé à 35% en 2014. Par contre, ce chiffre a augmenté durant l'année suivante pour atteindre 42% [4].

FIGURE 1.1 – Statistiques d'utilisation d'Internet et des réseaux sociaux dans le monde [4]

Cette utilisation intensive de l'Internet a entraîné une augmentation significative de la taille des données. L'entreprise américaine pour le stockage des données EMC a mentionné dans un article récemment publié [2], que le volume des données numériques peut atteindre 44 milliards de gigaoctets en 2020, 10 fois plus qu'en l'an 2013, tel qu'illustré à la figure 1.2.

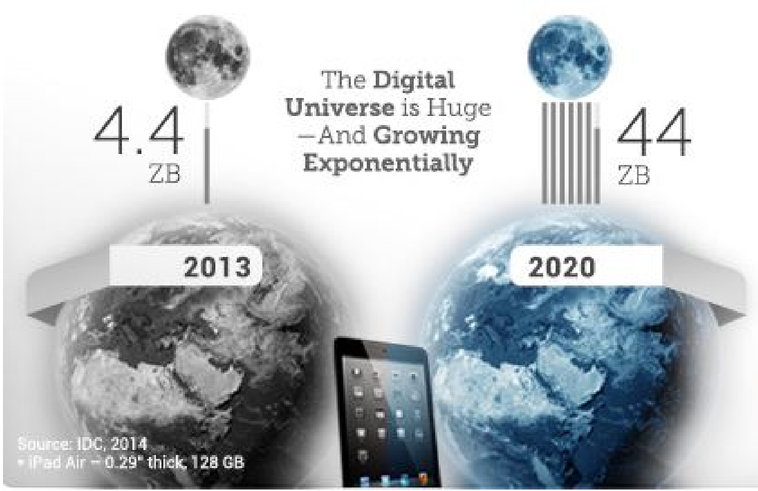

FIGURE 1.2 – Croissance estimée du volume mondial des données entre 2013 et 2020 [2]

L'augmentation significative de la taille de données est due à l'Internet des objets d'abord, mais aussi à l'augmentation de la qualité des fichiers. En effet, ces dernières décennies, nous avons vécu une grande révolution au niveau de qualité des fichiers échangés. Prenons l'exemple des fichiers vidéos où nous sommes passés d'une résolution faible à une résolution de haute définition. En conséquence, une telle quantité d'information a engendré des problèmes liés à la capacité de stockage et au temps de transmission dans les canaux de communication. En pratique, la transmission ou le téléchargement d'une vidéo non compressée et avec une haute qualité peut prendre des heures et parfois des jours.

Pour faire face à ces défis, plusieurs solutions ont été proposées dans le but de diminuer la taille de l'information transmise. Parmi ces solutions, il y a la compression de données. Elle a joué un rôle important pour la minimisation de charge sur les canaux de transmission ainsi que pour diminuer le temps d'attente. La compression de données fait partie de la théorie de l'information. Elle englobe un ensemble de techniques permettant de réduire la taille réelle d'un message afin d'accélérer sa transmission ou de réduire son empreinte sur un support de stockage.

La performance de ces techniques est évaluée selon deux critères. Le premier critère se réfère à **la complexité calculatoire** [22], qui consiste à identifier le temps nécessaire pour exécuter un algorithme de compression. Le second est **le taux de compression** [22], qui correspond à la taille de données après l'opération d'encodage.
Par ailleurs, ces critères sont inspirés par **le facteur de modélisation** [22], dans lequel nous étudions la redondance existante dans un fichier. Aussi influencés par **le facteur d'encodage** [22], qui consiste à chiffrer ces redondances afin de diminuer la taille d'un message.

En outre, l'utilisation de ces techniques dépend du type d'information et de son contexte d'utilisation. Par exemple, dans le domaine de la médecine, il est nécessaire d'utiliser les

techniques de compression sans perte afin de conserver l'intégrité de l'information. La compression sans perte requiert l'utilisation d'un algorithme permettant de reconstruire l'information originale sans altération. Par contre, les techniques avec perte encodent les fichiers sources de façon à ce que les fichiers reconstruits ressemblent à leurs contenus originaux. De ce fait, l'objectif de la compression de données est de diminuer significativement la taille d'un fichier en substituant les symboles redondants dans un fichier par des codes dont la longueur est inférieure ou égale à la taille du symbole original.

Les codes sont classés selon leur taille en quatre catégories : variable à fixe, variable à variable, fixe à variable, fixe à fixe. Chaque catégorie regroupe un ensemble d'algorithmes de compression. La figure 1.3 présente ces différentes catégories de code.

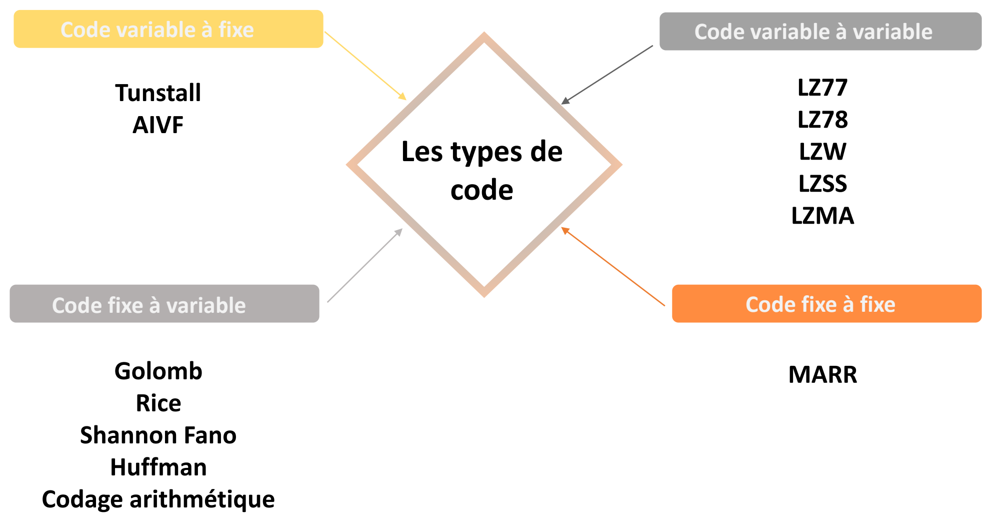

FIGURE 1.3 – Les types de code

Nos travaux de recherche s'inscrivent dans ce contexte, s'attaquant au développement d'une nouvelle technique de compression pour les codes variables à fixes quasi-instantanés présentés dans le chapitre 4. Aussi, nous étudions la performance de l'algorithme de Tunstall [27] et l'algorithme des codes VF quasi-instantanés (AIVF) conçu par Yamamoto et Yokoo [31].

Le présent mémoire s'articule autour de cinq chapitres. Le premier chapitre est la présente introduction. Dans le deuxième chapitre, nous présentons les différentes notions reliées à la compression de données. Le troisième chapitre réfère à l'état de l'art dans lequel nous étudions la technique de Tunstall [27] et la technique AIVF [31]. De plus, nous proposons

un survol des différents travaux reliés à cet axe de recherche. Dans le quatrième chapitre, nous présentons une nouvelle technique de compression dans la catégorie variable à fixe basée sur la programmation dynamique, et nous proposons une amélioration des algorithmes de Yamamoto et Yokoo [27] pour les codes VF quasi-instantanés. Enfin, la conclusion générale présente une synthèse de notre travail, et inclut des éléments de perspective pour des travaux futurs. Nous terminons ce mémoire par des annexes, qui correspondent à nos implantations.

Chapitre 2

Présentation générale

2.1 Introduction

Dans ce chapitre, nous introduisons les différents concepts reliés à la compression de données. Ces concepts sont par la suite utilisés tout au long de ce mémoire.

2.2 Les concepts généraux

La compression de données a été inventée par Shannon-Fano en 1940, et fait partie de la théorie de l'information. Le concept de compression ou de compaction de données suit deux opérations : une opération de compression, qui permet de transformer le fichier original en fichier plus court. La deuxième opération est l'inverse, nommée décompression, qui consiste à reconstituer le fichier original à partir du fichier compressé. Lors de l'opération de compression, nous substituons la suite de symboles sources par une suite de mots de code. La longueur moyenne du document encodé doit être inférieure ou égale à la taille du document original.

2.2.1 Les types de code

Soit A un alphabet d'entrée et B un alphabet de sortie. Le dictionnaire d'entrée D est un ensemble de mots sur A. L'ensemble E des mots de code est un ensemble de mots sur B. Habituellement, nous avons $|D| = |E|$. La fonction de codage $C : A^* \to B^*$ est une fonction injective définie sur tous les éléments de D. Tout au long de ce mémoire, nous ignorons la question de la terminaison de la chaîne d'entrée. C'est-à-dire que nous supposons qu'il est toujours possible de décomposer la chaîne d'entrée en une concaténation de mots de D.

Nous classons ces codes selon quatre catégories.

— Les codes variables à fixes (VF) : nous accordons à chaque chaîne énumérée dans un dictionnaire de longueur variée un mot de code de longueur fixe. Nous laissons les

mots dans D ayant des longueurs variables mais nous fixons I comme étant la longueur des mots dans E. L'efficacité de la fonction d'encodage C dépend de la longueur moyenne des mots d'entrée et de I. Pour ce type de code, nous cherchons à maximiser la longueur moyenne des mots d'entrée. Par exemple, l'algorithme de Tunstall [27] est un code VF que nous le présentons dans la section 3.3.

— Les codes variables à variables (VV) : nous associons à chaque chaîne énumérée dans un dictionnaire de longueur variable un mot de code de longueur variable. Plus précisément, la longueur des mots dans D et E est variable. L'efficacité de C dépend de la longueur moyenne des mots d'entrée et de la longueur moyenne des mots de code. Pour ce type de code nous cherchons à maximiser la longueur des mots d'entrée et à minimiser celle des mots de code. Par exemple, LZ77 [34], LZ78 [35], LZW [28] sont des codes VV. Nous détaillons les techniques LZ77 et LZ78 dans la section 2.6.3.

— Les codes fixes à variables (FV) : les chaînes sources de taille fixe sont transformées en mots de code de longueur variable. Autrement dit, nous fixons K comme étant la longueur des mots dans D mais nous laissons les mots dans E ayant des longueurs variables. L'efficacité de la fonction de codage C dépend de k et de la longueur moyenne du mot de code. Généralement, les symboles sont encodés selon leurs probabilités d'apparition. Le symbole le plus probable est encodé avec le mot de code le plus court. Par contre, le moins probable est encodé avec le mot de code le plus long. Pour ce type de code nous cherchons à minimiser la longueur moyenne des mots de code. Par exemple, l'encodage de Huffman [18] est un code FV que nous le présentons dans la section 2.6.2.

— Les codes fixes à fixes (FF) : la chaîne source possède une taille fixe et elle sera encodée avec un mot de code aussi de taille fixe [7]. Autrement dit, la longueur K des mots dans l'ensemble D est égale à la longueur I des mots dans l'ensemble E. Par exemple, le code de parité pour la détection des erreurs. Le bit de parité est un mécanisme utilisé pour détecter les erreurs dans un message. Il s'appuie sur le principe suivant où nous divisons l'information à manipuler en bloc de bits par exemple blocs de 8 bits. Ensuite, nous rajoutons à chaque bloc un bit 1 ou 0 selon la parité du nombre de bits à 1.

Tableau 2.1 – Un exemple du mécanise de bit de parité

Donnée finale	**Nombre de 1**	**Conclusion**	**Donnée initiale**
1111 0000 1	5 bits à 1	Erreur	
1111 0011 0	6 bits à 1	Pas d'erreur	1111 0011
1101 0110 0	5 bits à 1	Erreur	

Nous focalisons nos travaux de recherche sur les codes de longueurs variables à fixes (VF).

2.2.2 Définition du codage

Le codage est une opération qui consiste à transformer une chaîne d'entrée de gauche à droite. Ces sous-chaînes doivent appartenir à un dictionnaire de mots d'entrée. Nous nommons les sous-chaînes d'entée tout en long de ce mémoire par les *mots du dictionnaire*. En revanche, les sous-chaînes produites en sortie sont appelés des *mots de code*. Plus précisément, l'entrée $\mathcal{W}$ est décomposée en une succession $\mathcal{W}_1, \ldots, \mathcal{W}_n$ de mots d'entrée et la sortie est la concaténation $C(\mathcal{W}_1), \ldots, C(\mathcal{W}_n)$ de mots de code, où C est une fonction de codage qui permet de transformer les sous-chaînes d'entrée en une séquence de mot de code.

2.2.3 Définition d'un code préfixe

Un code préfixe est un code possédant la caractéristique de ne comporter aucun mot de code qui soit le préfixe d'un autre mot de code [1]. En particulier, pour les codes FF ou VF, cette propriété découle automatiquement du fait que la fonction de codage C est une fonction injective. L'ensemble $(1, 01, 00)$ est un code préfixe valide. L'ensemble $(0, 1, 01, 00)$ n'est pas un code préfixe valide, car les mots de code 0 et 01 sont tous deux préfixes de la séquence de bits 0100110011.

2.2.4 Définition d'un code exhaustif

Soit S un ensemble de mots. Soit Σ l'alphabet d'où sont tirés les mots dans S. S forme un code exhaustif si, pour tout mot infini $w \in \Sigma^\infty$, il existe un mot dans S qui est un préfixe de w.

2.3 La terminologie

Dans cette partie, nous définissons certains concepts utilisés dans le domaine de la théorie de l'information et de la compression de données.

Arbre : l'arbre est une représentation graphique composée d'un ensemble de nœuds. Un noeud est désigné comme étant la racine de l'arbre. À partir de la racine nous créons d'autres nœuds internes (cercles pleins) ou des feuilles (cercles vides). Chaque nœud interne possède un nombre fini d'enfants.

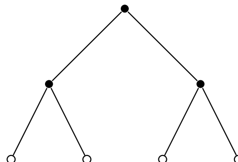

FIGURE 2.1 – Les éléments d'un arbre

Arbre n-aire complet : l'arbre n-aire complet est un arbre dont chaque nœud interne possède n-enfants.

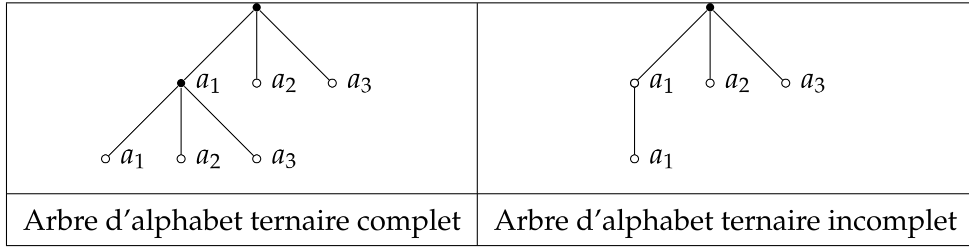

FIGURE 2.2 – Arbre d'alphabet ternaire complet et arbre d'alphabet ternaire incomplet

Profondeur d'un nœud : la profondeur d'un noeud est le nombre de branches parcourues depuis la racine vers un nœud particulier.

Arbre n-aire avec probabilités : l'arbre n-aire avec probabilités est un arbre dont chaque nœud interne possède n branches doté de probabilités variant entre 0 et 1. La probabilité associée à un nœud est égale à la somme des probabilités des feuilles du sous-arbre issu de ce nœud.

Arbre d'encodage : chaque branche dans l'arbre est étiqueté par un bit. Les mots de code assignés à un symbole correspondent à la concaténation des codes binaires attribués à chaque branche allant de la racine vers une feuille.

Distorsion : la distorsion est la différence entre le fichier original et le fichier reconstruit.

Code propre : un code propre correspond à un arbre complet dont toutes les feuilles sont associées à des mots de code.

Longueur moyenne des mots de code : la longueur moyenne d'un mot de code ou d'un mots d'entrée correspond au nombre de bits associés à un symbole. Pour calculer la longueur moyenne, nous utilisons les probabilités assignées aux mots d'entrée. La longueur moyenne est notre unité de mesure pour valider la performance de nos algorithmes.
La longueur moyenne est calculée grâce à la formule suivante [16] :

$$L_{moy} = \sum_{i=1}^{n} P(a_i) * l(a_i)$$

où :

— $P(a_i)$: probabilité de symbole a_i.

— $l(a_i)$: longueur du mot du dictionnaire assigné au symbole a_i.

Information propre : cette notion notée $i(A)$ pour un événemenet A a été définie par Claude Shannon en 1948 dans son article [23] comme étant la relation entre l'information disponible par une source et la probabilité de distribution de cette information. Par exemple, si la probabilité d'un événement A est $P(A) = 1$ alors nous sommes certain que A se réalisera. Par conséquent, nous ne possédons aucune information d'où $i(A) = 0$ [6].
L'information propre est représentée selon la formule suivante [23].

$$i\left(A\right) = \log\left(\frac{1}{P(A)}\right) = -\log\left(P\left(A\right)\right)$$

Entropie d'une source : l'entropie d'une source correspond au nombre moyen minimal de bits nécessaires pour représenter un symbole source généré par une variable aléatoire [26].

$$H\left(X\right) = \sum_{i=1}^{n} P\left(a_i\right) i\left(A_i\right)$$

où :

- $H\left(X\right)$: est l'entropie de la variable aléatoire X, où X comporte n symboles.
- a_i : est la probabilité de chaque symbole a_i.

2.4 Les types de compression

Les techniques de compression de données se basent sur deux types de compression, la compression avec perte et la compression sans perte.
La technique sans perte permet de reconstruire un fichier sans altération après l'opération de décodage. Nous pouvons l'exprimer sous la forme suivante $W = D(C(W))$ [9].
où

- W : désigne le fichier source.
- D : correspond à l'opération de décompression.
- C : correspond à l'opération de compression.

En revanche, si nous appliquons le principe de la compression avec perte, le contenu du fichier original sera modifié, d'où nous ne garantissons pas que $W = D(C(W))$ [9].

2.4.1 La compression sans perte

La compression sans perte, nommée en anglais **lossless compression**, est une technique pour encoder les données sous un format plus compact sans les altérer. Elle est utilisée dans les applications qui ne tolèrent pas la modification du contenu de fichiers sources, dans le but de conserver leur intégrité. De ce fait la nécessité d'utiliser ce type de compression provient de

la nature des fichiers (les fichiers exécutables sont des données dont la nature est inaltérable [9]) ou du contexte d'utilisation (le domaine médical nécessite l'exactitude des données [9]).

2.4.2 La compression avec perte

La compression avec perte ou la compression irréversible nommée en anglais **lossy compression** englobe un ensemble de techniques permettant d'identifier et d'encoder d'une façon unique tous les zones de ressemblance qui ne sont pas détectables à l'œil humain, tel que sur une photographie par exemple. Suite à ce mécanisme, nous obtenons une donnée ressemblante à l'originale et qui semble identique pour les sens de l'humain.

En contrepartie, la compression avec perte se permet d'oublier une partie de l'information. La distorsion ainsi créée nous permet d'obtenir des meilleurs taux de compression comparativement à la compression sans perte.
Les travaux présentés dans ce mémoire concernent les techniques de compression sans perte.

2.5 Les phases de la compression

L'opération de compression citée dans la section 2.2 suit deux phases. La première phase est la modélisation. Cette phase consiste à établir un ensemble de règles afin de gérer la redondance des symboles dans un fichier. Partant de ce fait, plusieurs modèles ont été élaborés. Parmi ces modèles nous citons les suivants.

- Modèle de type dictionnaire : ces modèles reposent sur le principe de remplacement des caractères redondants par des références à leurs apparitions antérieures, tel que les algorithmes LZ77[34] et LZ78 [35].
- Modèles de type statique : généralement, les modèles statiques construisent leurs règles en s'appuyant sur la fréquence d'apparition des symboles, tel que le code proposé par Huffman [18].

Quant à elle, la phase d'encodage succède à l'opération de modélisation. Dans cette phase, nous assignons un mot de code à un symbole selon les règles qui ont été déduites dans la phase de modélisation. Prenons l'exemple de la méthode d'encodage proposée par Huffman [18], qui se base sur le principe de la fréquence d'apparition pour gérer les répétitions des symboles. Ensuite, selon la fréquence d'apparition des symboles nous attribuons le mot de code le plus court au symbole le plus fréquent et le mot de code le plus long au symbole ayant une fréquence faible.

2.6 Les techniques de compression sans perte

La compression sans perte regroupe un ensemble de techniques s'appuyant sur différents principes. Parmi ces techniques nous citons : l'encodage par répétition, l'encodage entro-

pique, l'encodage par dictionnaire, l'encodage par modélisation de contexte et la transformée de Burrows-Wheeler.

2.6.1 L'encodage par répétition

L'encodage par répétition ou par plages nommé en anglais **Run-Length Encoding (RLE)**, est l'algorithme le plus simple dans le domaine de la compression de données. Cette technique est classée dans la catégorie des codes VV. Elle suit le principe du modèle statique pour gérer la redondance dans un fichier. Son principe consiste à substituer une séquence de symboles identiques par le nombre de ses répétitions concaténé avec le symbole lui-même. Prenons l'exemple de la chaîne suivante : BBBVVVVVVVVYYYEE. L'exécution de RLE remplace la chaîne précédente par 3B8V3Y2E. Afin de différencier le message 3B de la séquence 3B dans les données originales, nous transmettons une séquence d'échappement avant d'envoyer le nombre de répétitions d'un symbole [9].
Les algorithmes RLE sont performants lorsqu'ils s'exécutent sur des fichiers contenant beaucoup de répétitions.

2.6.2 L'encodage entropique

Le principe de l'encodage entropique s'appuie sur la fréquence ou la probabilité d'apparition des symboles. En effet, nous associons les mots de code courts aux symboles les plus fréquents et les mots de code longs aux symboles les moins fréquents, tel que proposé par Huffman [18].

L'encodage de Huffman

L'encodage de Huffman est l'un des algorithmes les plus utilisés en compression de données. Il a été publié par David Huffman en 1952 [18]. La technique de Huffman construit un code FV optimal en transformant les symboles individuels en des chaînes de bits. L'algorithme représente le code sous forme d'arbre dont les feuilles correspondent aux différents symboles sources.

L'algorithme de Huffman [18] est le suivant.

- Mettre les symboles dans une liste L et les ordonner par probabilités.
- Retirer les deux symboles s_1 et s_2 ayant les probabilités les plus faibles.
- Créer un nouveau symbole s_i dont la probabilité $P(s_i)$ est $P(s_1) + P(s_2)$. Le symbole s_i est le parent des symboles s_1 et s_2.
- Ajouter le nouveau symbole s_i à la liste L.
- Refaire les étapes précédentes jusqu'à l'obtention d'un seul nœud racine.

Exemple d'encodage de Huffman

Nous considérons le tableau 2.2 contenant nos symboles ainsi que leurs fréquences d'apparition.

Tableau 2.2 – Fréquence d'apparition des symboles de l'ensemble A

Symboles	a	b	c	d	e
Fréquence	4	12	13	7	5

En appliquant le principe de l'algorithme de Huffman nous pouvons générer l'arbre de la figure 2.3.

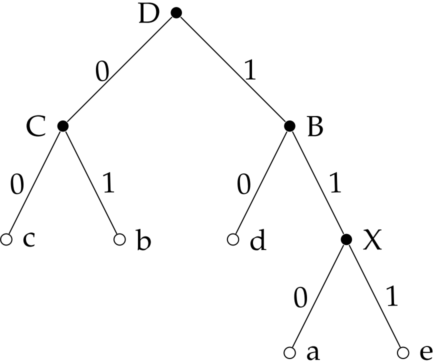

FIGURE 2.3 – L'arbre de Huffman pour l'ensemble A

L'algorithme de Huffman doit relier les symboles ayant les fréquence les plus faibles. Par exemple, nous relions ensemble les symboles e et a selon la notation suivante.

$$X \begin{matrix} \nearrow e \\ \searrow a \end{matrix}$$

Le fait de combiner ces deux symboles et de les remplacer par symbole artificiel X signifie que le mot de code de X sera formé en ajoutant 0 au mot de code e et 1 pour le mot de code a. La fréquence de X est la somme des fréquences de e et de a [26]. Tel qu'il est présenté dans la figure 2.3 la construction de l'arbre de Huffman se base sur la combinaison des différents symboles artificiels.

c : 13, b :12, d :7, e :5, a :4

$$X \begin{matrix} \nearrow e \\ \searrow a \end{matrix}$$

c : 13, b :12, X :9, d :7

$$B \begin{matrix} \nearrow X \\ \searrow d \end{matrix}$$

B : 16, c :13, b :12

$C \nearrow c$, $C \searrow b$

C :25, B :16

$D \nearrow C$, $D \searrow B$

D :41

Tableau 2.3 – Exemple de code généré par la technique de Huffman [18]

Symboles	c	b	d	e	a
Mots de code	00	01	10	111	110

La figure 2.3 présente l'arbre de Huffman pour l'alphabet (A). Nous étiquetons les deux branches issues d'un noeud interne par des bits. Chaque feuille correspond à un symbole de l'alphabet. Afin de récupérer les mots de code assignés aux feuilles, nous parcourons l'arbre de la racine vers la feuille cible. Prenons l'exemple du symbole c de l'alphabet. Nous le présentons par le code binaire 00 (tableau 2.3). Ce code binaire est attribué à la branche qui relie la racine au symbole c. Pour encoder le message M=cdbc nous substituons chaque symbole par le mot de code qui lui correspond dans le tableau 2.3, d'où le message encodé M'=00100100.

L'encodage arithmétique

La technique d'encodage arithmétique constitue une des techniques les plus répandues dans le domaine de compression de données puisqu'elle fournit le meilleur taux de compression par rapport aux différentes techniques d'encodage entropique.

La différence entre l'encodage de Huffman et l'encodage arithmétique réside dans le nombre de bits utilisés pour encoder un symbole. L'algorithme de Huffman [18] ne peut encoder un symbole que sur un nombre entier de bits uniquement. En conséquence, tout symbole doit être codé avec au moins un bit. La technique d'encodage arithmétique présente l'avantage de pouvoir encoder les symboles sur un nombre réel de bits. Prenons l'exemple d'un symbole x qui apparaît 90% du temps dans le texte à compresser. L'information propre du symbole x est 0.15. Hors, l'utilisation de la méthode d'encodage de Huffman [18] assigne au symbole x un mot de code d'au moins 1 bit. Nous constatons que le mot de code assigné au symbole x sera 6 fois plus grand que son information propre.

Dans ce contexte, le code arithmétique a été développé pour corriger cette lacune. Il encode une chaîne M, à l'aide de la subdivision de l'intervalle $I = [0, 1[$ selon les probabilités des symboles.

Les différentes itérations de l'encodage arithmétique permettent de mettre à jour les bornes de l'intervalle $I = [L, U[$ en utilisant les équations suivantes :

$$U^{(n)} = L^{(n-1)} + (U^{(n-1)} - L^{(n-1)}) * F_{(X)}(x_{(n)})$$
$$L^{(n)} = L^{(n-1)} + (U^{(n-1)} - L^{(n-1)}) * F_{(X)}(x_{(n-1)})$$

où :

— $F_{(X)}$: désigne la fonction de distribution cumulative de la variable X.

— X : soit X une variable aléatoire hypothétique qui a servi pour chaque symbole de M.

— $U^{(n)}$: limite supérieure.

— $L^{(n)}$: limite inferieure.

— n : le nombre d'observation.

La fonction de distribution cumulative ou (CDF) est la suivante $0 \leq F_{(X)} = Pr(X \leq x) \leq 1$ [14]. Elle est utilisée pour la création du code arithmétique.
Le principe de l'algorithme arithmétique est le suivant [9] :

— Initialiser l'intervalle à $[0, 1[$

— Tant qu'il y a des symboles à encoder :

  — Diviser l'intervalle courant selon la probabilité des symboles.

  — Lire le prochain symbole a encodé.

  — Restreindre l'intervalle courant en un autre sous-intervalle.

— Répéter les étapes précédentes jusqu'à ce que tous les symboles de la chaîne source soient encodés.

Exemple d'encodage arithmétique

Considérons la source aléatoire telle que présentée dans le tableau 2.4. Nous voulons encoder la chaîne source M=abca en utilisant l'encodage arithmétique.

Tableau 2.4 – Probabilité d'apparition des symboles

Symboles	a	b	c
Probabilités	0.6	0.3	0.1

À partir des probabilités ci-dessus, nous avons : $F_X(0) = 0$, $F_X(a) = 0.6$, $F_X(b) = 0.9$, $F_X(c) = 1$. Pour permettre d'utiliser F_X nous inventons un symbole artificiel nommé $F_X(0) = 0$ que nous l'utilisons avant les autres.

Tableau 2.5 – Les bornes des intervalles pour la chaîne M=abca

Itérations	Symboles	Bornes des intervalles
_	_	[0,1[
1	a	[0,0.6[
2	b	[0.36, 0.54[
3	c	[0.522, 0.54[
4	a	[0.522, 0.532[

Les valeurs présentées dans le tableau 2.5 décrivent les intervalles pour encoder le message M. Les différentes itérations effectuent le calcul suivant.

État initial :

$$U^{(0)} = 1$$
$$L^{(0)} = 0$$

Itération 1 :

$$U^{(1)} = L^{(0)} + (U^{(0)} - L^{(0)}) * F_{(X)}(a) = 0 + (1-0) * 0.6 = 0.6$$
$$L^{(1)} = L^{(0)} + (U^{(0)} - L^{(0)}) * F_{(X)}(0) = 0 + (1-0) * 0 = 0$$

Itération 2 :

$$U^{(2)} = L^{(1)} + (U^{(1)} - L^{(1)}) * F_{(X)}(b) = 0 + (0.6-0) * 0.9 = 0.54$$
$$L^{(2)} = L^{(1)} + (U^{(1)} - L^{(1)}) * F_{(X)}(a) = 0 + (0.6-0) * 0.6 = 0.36$$

Itération 3 :

$$U^{(3)} = L^{(2)} + (U^{(2)} - L^{(2)}) * F_{(X)}(c) = 0.36 + (0.54-0.36) * 1 = 0.54$$
$$L^{(3)} = L^{(2)} + (U^{(2)} - L^{(2)}) * F_{(X)}(b) = 0.36 + (0.54-0.36) * 0.9 = 0.522$$

Itération 4 :

$$U^{(1)} = L^{(0)} + (U^{(0)} - L^{(0)}) * F_{(X)}(a) = 0.522 + (0.54-0.522) * 0.6 = 0.532$$
$$L^{(1)} = L^{(0)} + (U^{(0)} - L^{(0)}) * F_{(X)}(0) = 0.522 + (0.54-0.522) * 0 = 0.522$$

La figure 2.4 présente graphiquement les bornes des intervalles des quatre itérations effectuées pour encoder M = abca. Les différentes itérations présentées ci-dessus génèrent une étiquette (point central de l'intervalle). Cette étiquette sera transmis au décodeur afin de pouvoir reconstruire le message M.

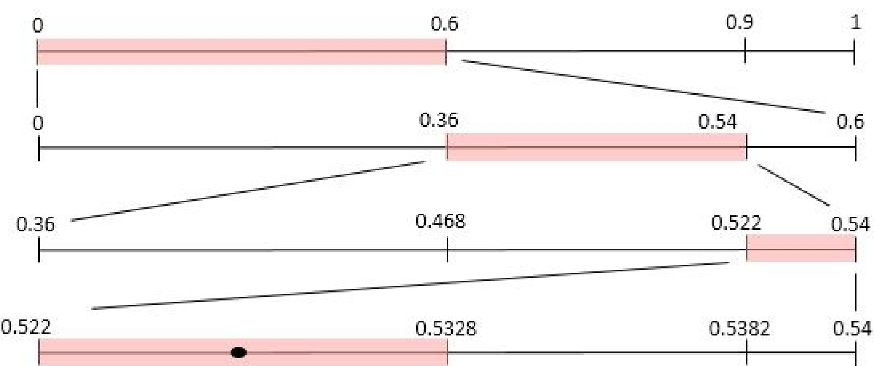

FIGURE 2.4 – L'encodage du message M=abca sous forme graphique

Prenons l'exemple précédent : notre étiquette est égale à 0.527, soit le point central de l'intervalle [0.522, 0.532). La stratégie utilisée par le décodeur consiste à vérifier si l'étiquette se situe dans l'intervalle $[F_X(k-1), F_X(k))$. En effet, à la première itération, nous déduisons que l'étiquette 0.527 se situe dans l'intervalle $[0, 0.6]$. En conséquence, le décodeur retrouve a. À l'étape suivante nous mettons à jour nos bornes d'intervalle pour que le décodeur déduise le nouveau symbole. Nous suivons ce principe jusqu'à ce que le message M soit entièrement décodé.

Malgré le fait que cette technique produise un meilleur taux de compression par rapport aux autres algorithmes de compression, elle demeure toujours moins utilisée parce qu'elle est plus coûteuse à mettre en application.

2.6.3 L'encodage par dictionnaire

En 1977 et 1978, Jacob Ziv et Abraham Lempel ont présenté les techniques LZ77 [34] et LZ78 [35]. Ce sont les deux algorithmes les plus connus au niveau de l'encodage par dictionnaire. Les algorithmes de compression par dictionnaire débutent par une recherche de similitudes entre le texte courant (non encodé) et le texte déjà encodé. Si une similitude est trouvée, le texte courant sera substitué par une référence à son emplacement précédent.

Plusieurs variantes ont été proposées pour améliorer les techniques LZ77 et LZ78. Par exemple le LZW énoncé par Terry Welch dans l'article [28]. En dépit de leurs faiblesses, les algorithmes de Jacob Ziv et Abraham Lempel procurent un meilleur taux de compression comparativement à l'encodage entropique.

L'algorithme LZ77

La technique LZ77 utilise le principe de la fenêtre coulissante pour repérer les motifs redondants. La fenêtre coulissante est définie comme étant une mémoire tampon qui contient la partie du texte récemment encodé. Autrement dit, si les prochains caractères de la chaîne d'entrée apparaissent dans la fenêtre coulissante nous pouvons remplacer ceux-ci par une référence à leurs emplacements dans la fenêtre coulissante. Ces références sont sous formes des triplets <O,L,C>.

— (O)ffset : pointeur de début.

— (L)ongueur : nombre maximal de symboles consécutifs.

— (C)ode : le prochain caractère.

Dans le même contexte, plusieurs autres variantes ont été proposées. Elles adoptent aussi le principe de la fenêtre coulissante que LZSS [25].
Prenons l'exemple de la chaîne aabbaabcaabbacb, nous voudrons encoder ce message en utilisant LZ77 tous en respectant les conventions citées ci-dessous.

— Convention 1 : trouver une copie de longueur maximale, soit L cette longueur.

— Convention 2 : s'il y a plusieurs copies de longueur maximale, nous choisissons celle qui a le décalage (O) le petit.

Le tableau 2.6 décrit les différentes étapes pour encoder le message aabbaabcaabbacb. Nous utilisons le symbole | pour séparer le texte déjà encodé du celui à encoder. Plus précisément, nous considérons que la fenêtre coulissante est à gauche du symbole | . Cette fenêtre contient le texte déjà encodé. En revanche, le texte qui se trouve à droite du symbole | c'est le texte a encodé.

Tableau 2.6 – L'encodage de la chaîne aabbaabcaabbacb en utilisant le principe de LZ77

Itération	chaîne	Triplet
1	|aabbaabcaabbacb	<0,0,a>
2	a|abbaabcaabbacb	<1,1,b>
3	aab|baabcaabbacb	<1,1,a>
4	aabba|abcaabbacb	<4,2,c>
5	aabbaabc|aabbacb	<8,5,c>
6	aabbaabcaabbac|b	<0,0,b>
7	aabbaabcaabbacb|	

À l'itération 1, l'encodeur envoie le triplet <0,0,a> pour indiquer qu'il n'y a aucune correspondance de caractère a dans le bus de recherche. C'est-à-dire notre bus de recherche ne

contient pas le caractère a. À l'itération 2, nous avons encore le caractère a à encoder, le compresseur envoie le triplet <1,1,b> pour indiquer qu'il a trouvé le symbole a à l'offset 1, sa longueur est égale à 1 et le symbole qui suit a est un b. À l'itération 3, le compresseur envoie le triplet <1,1,a> puisqu'il a trouvé le symbole b à l'offset 1, sa longueur est égale à 1 et le caractère qui suit b est un a. À l'itération 4, nous voulons encoder le caractère a, le compresseur envoie le triplet <4,2,c> puisqu'il a trouvé la plus longue chaîne du symbole ab à l'offset 4, sa longueur est égale à 2 et le caractère qui suit b est un c. À l'itération 5, nous voulons encoder encore une fois le caractère a, le compresseur envoie le triplet <8,5,c> puisqu'il a trouvé la plus longue chaîne du symbole aabba à l'offset 8, sa longueur est égale à 5 et le caractère qui suit a est un c. Nous procédons de cette façon jusqu'à ce que tout le message soit encodé.

L'algorithme LZ78

L'algorithme LZ78 a été présenté l'année suivante par les mêmes auteurs [35]. La technique LZ78 a été inventée pour corriger la faiblesse amenée par l'utilisation de la fenêtre coulissante. En effet, la fenêtre coulissante limite la distance à laquelle nous pouvons faire référence aux symboles déjà rencontrés, ce qui fait que l'algorithme LZ77 est incapable de traiter le texte qui est en dehors de cette fenêtre.

Dans ce contexte, LZ78 a éliminé l'utilisation de la fenêtre coulissante, il se sert d'un dictionnaire global pour encoder le message. Ce dictionnaire est construit d'une manière progressive. Les symboles sont ajoutés au fur et à mesure dans le dictionnaire et nous assignons à chaque nouveau symbole un indice.
Les entrées du dictionnaire sont substituées par le couple <i,c> avec :

— i : pour identifier l'indice de l'entrée maximale dans le dictionnaire. L'indice 0 sert à repérer les nouveaux symboles ajoutés au dictionnaire.

— c : le prochain symbole à encoder.

Par exemple, nous désirons encoder la chaîne aabbaabcaabbac avec l'algorithme LZ78 (tableau 2.7). À l'itération 1, l'encodeur envoie le couple <0,a> car aucune correspondance n'a été détectée dans le dictionnaire lors de l'encodage du caractère a. À l'itération 2, il émet le couple <1,b> où 1 est l'indice du caractère a dans le dictionnaire, et b est le caractère suivant. À l'itération 3, l'encodeur émet le couple <0,b> car aucune correspondance n'a été détectée dans le dictionnaire lors de l'encodage du caractère b. À l'itération 4, l'encodeur émet le couple <1,a> où 1 est l'indice du caractère a dans le dictionnaire, et a est le caractère suivant. À l'itération 5, l'encodeur émet le couple <3,c> où 3 est l'indice du caractère b dans le dictionnaire, et c est le caractère suivant. Nous répétons les itérations précédentes jusqu'à ce que toute la chaîne ait été traitée lue par l'encodeur.

Tableau 2.7 – L'encodage de la chaîne aabbaabcaabbac en utilisant le principe de LZ78

<table>
<tr><th>Indice</th><th>Chaîne</th><th>Couple</th><th>Entrée</th></tr>
<tr><td>0</td><td></td><td></td><td></td></tr>
<tr><td>1</td><td>| aabbaabcaabbac</td><td><0,a></td><td>a</td></tr>
<tr><td>2</td><td>a | abbaabcaabbac</td><td><1,b></td><td>ab</td></tr>
<tr><td>3</td><td>aab | baabcaabbac</td><td><0,b></td><td>b</td></tr>
<tr><td>4</td><td>aabb | aabcaabbac</td><td><1,a></td><td>aa</td></tr>
<tr><td>5</td><td>aabbaa | bcaabbac</td><td><3,c></td><td>bc</td></tr>
<tr><td>6</td><td>aabbaabc | aabbac</td><td><4,b></td><td>aab</td></tr>
<tr><td>7</td><td>aabbaabcaab | bac</td><td><3,a></td><td>ba</td></tr>
<tr><td>8</td><td>aabbaabcaabba | c</td><td><0,c></td><td>c</td></tr>
<tr><td>9</td><td>aabbaabcaabbac |</td><td></td><td></td></tr>
</table>

2.6.4 L'encodage par modélisation de contexte

Introduit en 1984, par John Cleary et Ian Witten [12]. Le principe de l'algorithme de modélisation par contexte, en anglais **Prediction by Partial Matching (PPM)**, repose sur deux étapes.

- Une étape de modélisation de contexte : elle consiste à encoder le nouveau symbole en utilisant l'historique sauvegardé.
- Une étape d'encodage : l'étape d'encodage se fait en utilisant la technique de codage arithmétique.

Les algorithmes de modélisation de contexte permettent de prédire la probabilité du prochain symbole à encoder à partir d'un ensemble de contextes. Chaque contexte contient la fréquence d'apparition des différents symboles déjà vus. Nous identifions un contexte par son numéro d'ordre. Prenons l'exemple d'un PPM d'ordre 3 qui permet de prédire le nouveau symbole en utilisant deux symboles vus précédemment.

Le mécanisme du PPM suit la démarche suivante pour prédire un nouveau symbole jamais rencontré. Dans le cas où il est impossible de prévoir ce symbole à partir d'un contexte N, nous envoyons d'abord un symbole d'échappement ESC pour indiquer que le nouveau symbole n'existe pas dans le contexte courant et ensuite, nous utiliserons le contexte $N-1$. Nous répétons ces étapes jusqu'à atteindre le contexte d'ordre 0. Lorsqu'il s'avère que le nouveau symbole n'existe pas dans le contexte 0, nous optons pour un modèle équiprobable qui ne dépend d'aucun contexte, c'est le modèle d'ordre -1.

L'algorithme présenté ci-dessous décrit les différentes étapes d'exécution d'un PPM . Nous nous servirons des variables suivantes pour présenter notre algorithme où :

— M : est la taille de l'alphabet.

— C : est le contexte courant.

— S : désigne le nouveau symbole à encoder.

— N : désigne le numéro d'ordre du contexte.

L'algorithme PPM fonctionne selon le principe suivant [15].

— Si le nouveau symbole S n'existe pas dans le contexte C et $N > -1$ alors

  — Envoyer le symbole d'échappement ESC pour indiquer qu'il y a un nouveau symbole c'est-à-dire ce symbole n'existe pas dans le contexte courant.

  — Décrémenter N.

  — Mettre à jour C.

  — Répéter les instructions précédentes jusqu'à $N = -1$.

— Si $N = -1$ alors

  — Encoder S avec une probabilité $Pr = 1/M$, où M est la taille de l'alphabet.

— Si $N > -1$ alors

  — Procéder à l'étape d'encodage en utilisant le code arithmétique.

  — Mettre à jour le poids du symbole S dans les N contextes.

Testons l'algorithme de PPM sur l'exemple suivant. Nous considérons que le texte baccabac est déjà encodé, le tableau 2.8 illustre les modèles d'ordre -1, 0, 1, 2 et 3 réunis pour l'alphabet A={a, b, c, d}. Cet exemple a été extrait du cours IFT-7023 compression de données, offert durant la session hiver 2016.

Tableau 2.8 – Les contextes d'ordre (-1, 0, 1, 2 et 3) pour l'alphabet A

Ordre -1			**Ordre 0**			**Ordre 1**			**Ordre 2**			**Ordre3**		
C	***S***	***N***	***C***	***S***	***N***	***C***	***S***	***N***	***C***	***S***	***N***	***C***	***S***	***N***
ϵ	a	1	ϵ	a	3	a	b	1	ab	a	1	aba	c	1
	b	1		b	2		c	2		ESC	1		ESC	1
	c	1		c	3		ESC	1	ac	c	1	acc	a	1
	d	1		ESC	1	b	a	2		ESC	1		ESC	1
							ESC	1	ba	c	2	bac	c	1
						c	a	1		ESC	1		ESC	1
							c	1	ca	b	1	cab	a	1
							ESC	1		ESC	1		ESC	1
									cc	a	1	cca	b	1
										ESC	1		ESC	1

Supposons que la chaîne baccabac est déjà encodée, résultant en contextes ci-dessus et que nous voulons encoder le symbole suivant b en s'appuyant sur ces contextes.
Pour encoder b nous suivons ces étapes :

- Encodage du symbole ESC dans le contexte bac d'ordre 3 avec une probabilité 1/2.
- Échappement vers le contexte ac d'ordre 2.
- Encodage du symbole ESC dans le contexte ac d'ordre 2 avec une probabilité 1/2.
- Échappement vers le contexte c d'ordre 1.
- Encodage du symbole ESC dans le contexte c d'ordre 1 avec une probabilité 1/3.
- Échappement vers le contexte d'ordre 0.
- Encodage du symbole b dans le contexte 0 avec une probabilité 2/9.
- Incrémentation du poids de symbole b dans les contextes bac, ac, c et 0.

2.6.5 La transformée de Burrows-Wheeler

La transformée de Burrows-Wheeler, en anglais **Burrows-Wheeler Transform** (BWT), a été présentée par David Wheeler et Michael Burrows en 1994 [10].
La BWT s'appuie sur le principe de restructuration des données avant l'opération d'encodage. La procédure d'encodage relative à la BWT se résume comme suit, étant donné que N est la taille d'un message :

- Créer la matrice $N * N$ à partir du message original
- Créer $N - 1$ séquences obtenues par un décalage cyclique du message original.
- Organiser la matrice en ordre lexicographique.
- Récupérer la dernière colonne L de la matrice.
- Transmettre les lettres accumulées de la colonne L.
- Transmettre le rang de la séquence originale dans la matrice, soit le numéro de la ligne relative à la séquence originale après l'opération de tri.

Nous appliquons le principe de BWT sur la chaîne compression.

Tableau 2.9 – Matrice de décalage cyclique pour la chaîne compression

Matrice de rotation										
c	o	m	p	r	e	s	s	i	o	n
o	m	p	r	e	s	s	i	o	n	c
m	p	r	e	s	s	i	o	n	c	o
p	r	e	s	s	i	o	n	c	o	m
r	e	s	s	i	o	n	c	o	m	p
e	s	s	i	o	n	c	o	m	p	r
s	s	i	o	n	c	o	m	p	r	e
s	i	o	n	c	o	m	p	r	e	s
i	o	n	c	o	m	p	r	e	s	s
o	n	c	o	m	p	r	e	s	s	i
n	c	o	m	p	r	e	s	s	i	o

Après la création d'un décalage cyclique, nous trions les rangées de la matrice en ordre lexicographique.

Tableau 2.10 – Tri lexicographique de la chaîne compression

Numéro de rangée	Matrice de rotation triée										
1	c	o	m	p	r	e	s	s	i	o	n
2	e	s	s	i	o	n	c	o	m	p	r
3	i	o	n	c	o	m	p	r	e	s	s
4	m	p	r	e	s	s	i	o	n	c	o
5	n	c	o	m	p	r	e	s	s	i	o
6	o	m	p	r	e	s	s	i	o	n	c
7	o	n	c	o	m	p	r	e	s	s	i
8	p	r	e	s	s	i	o	n	c	o	m
9	r	e	s	s	i	o	n	c	o	m	p
10	s	i	o	n	c	o	m	p	r	e	s
11	s	s	i	o	n	c	o	m	p	r	e

Après l'opération de tri nous notons le couple (L, N)

— L : est la dernière colonne de la matrice triée.

— N : est le numéro de ligne de la séquence originale.

Dans notre exemple, l'encodeur envoie le couple suivant $(L, n) = (nrsoocimpes, 1)$.
En termes d'analyse, nous déduisons que la BWT n'ajoute aucun avantage à la compression puisqu'il ne réduit pas la taille de données. En revanche, nous remarquons que nous avons besoin d'un nombre supplémentaire pour envoyer le rang de la séquence originale. Le bénéfice de cette technique est révélé lorsque nous appliquons ce principe sur un long texte. Dans ce cas, l'algorithme BWT tend à produire un L où il y a des lettres identiques qui se répètent fréquemment.

Dans ce contexte, les chercheurs David Wheeler et Michael Burrows [10] ont suggéré de combiner la BWT avec l'algorithme Move To Front (MTF). MTF substitue chaque symbole par un entier qui indique depuis combien d'étape grosso modo nous avons vu ce symbole. Le principe de MTF se présente comme suit [15].

— Trier l'alphabet A de la colonne L dans un ordre lexicographique.

— Attribuer la valeur 0 au premier caractère, 1 au deuxième, ainsi de suite.

— Encoder le nouveau symbole tout en envoyant le numéro qui lui correspond.

— Déplacer ce symbole en tête de liste.

Considérons l'exemple de la chaîne compression, nous désirons appliquer le principe de MTF sur le message (nrsoocimpes,1).
$L = nrsoocimpes$ et $A = \{ceimnoprs\}$.

Tableau 2.11 – L'encodage du $L = nrsoocimpes$ en utilisant le MTF

0 :c	0 :n	0 :r	0 :s	0 :o	0 :o	0 :c	0 :i	0 :m	0 :p	0 :e
1 :e	1 :c	1 :n	1 :r	1 :s	1 :s	1 :o	1 :c	1 :i	1 :m	1 :p
2 :i	2 :e	2 :c	2 :n	2 :r	2 :r	2 :s	2 :o	2 :c	2 :i	2 :m
3 :m	3 :i	3 :e	3 :c	3 :n	3 :n	3 :r	3 :s	3 :o	3 :c	3 :i
4 :n	4 :m	4 :i	4 :e	4 :c	4 :c	4 :n	4 :r	4 :s	4 :o	4 :c
5 :o	5 :o	5 :m	5 :i	5 :e	5 :e	5 :e	5 :n	5 :r	5 :s	5 :o
6 :p	6 :p	6 :o	6 :m	6 :i	6 :i	6 :i	6 :e	6 :n	6 :r	6 :s
7 :r	7 :r	7 :p	7 :o	7 :m	7 :m	7 :m	7 :m	7 :e	7 :n	7 :r
8 :s	8 :s	8 :s	8 :p	8 :p	8 :p	8 :p	8 :p	8 :p	8 :e	8 :n

Le code à envoyer est : $C = 47870467886$. Suite à l'utilisation de MTF nous pouvons utiliser l'algorithme de Huffman [18] pour encoder C.

2.7 Conclusion

Dans ce chapitre, nous avons présenté les principes généraux de la compression de données. Nous avons commencé par définir la différence entre la compression avec perte et la compression sans perte. Ensuite, nous avons présenté certaines définitions et certaines formules mathématiques qui sont fréquemment utilisées au niveau de la théorie de l'information. La dernière partie de ce chapitre a été affectée à la présentation de quelques techniques de compression sans perte. Le chapitre suivant sera dédié à l'état de l'art dans lequel nous allons focaliser sur les travaux de recherche concernant les codes variables à fixes.

Chapitre 3

Étude préliminaire

3.1 Introduction

Dans la première partie de ce chapitre nous étudions d'abord la technique de Tunstall [27]. Cette technique permet de créer un code variable à fixe (VF) optimal, qui respecte la contrainte d'être préfixe. Ensuite, nous présentons l'algorithme de Yamamoto et Yokoo [31], qui construit un code VF meilleur que celui de Tunstall. Nous concluons ce chapitre par un survol rapide des différents travaux reliés aux codes variables à fixes (VF).

3.2 Le code de Tunstall

L'algorithme de Tunstall appartient à la famille des codes variables à fixes (VF). Il adopte le principe de dictionnaire pour construire son code. Ce dictionnaire contient un ensemble de mots, appelé *mots du dictionnaire* qui sont associés aux différents symboles sources. Le nombre du mots de code dépend de la taille du dictionnaire. Entre autres, nous pouvons accroître le nombre de mots du dictionnaire tant que nous n'avons pas atteint la taille maximale du dictionnaire.

Tout au long de ce mémoire, nous présentons le dictionnaire des codes (VF) sous la forme d'arbres. Comme nous travaillons sur des codes (VF), tous les mots de code ont la même longueur et habituellement, nous ne donnerons pas explicitement les mots de code.

3.2.1 Principe du code de Tunstall

En 1967, Brian Parker Tunstall a inventé un nouvel algorithme de compression basé sur les codes variables à fixes. Cet algorithme a été présenté dans sa thèse intitulée **Synthesis of noiseless compression codes** [27].

La technique de Tunstall est classée parmi les techniques d'encodage entropique les plus utilisées pour les codes variables à fixes (VF). Le dictionnaire de Tunstall respecte la pro-

priété préfixe. C'est-à-dire, aucun mot de code ne peut pas être le préfixe d'un autre mot de code. Cette propriété n'est pas obligatoire pour construire un code VF. L'algorithme de Tunstall crée un arbre complet, où chaque mot de code est assigné à une feuille de l'arbre. La construction de l'arbre de Tunstall s'appuie sur les formules suivantes :

$$M = A + (A - 1)K$$

où :

— M : le nombre de mots de code dans l'arbre.

— A : la taille de l'alphabet.

— K : un entier {1,2,3,...}.

et

$$Q(n_j) = Q(n) \times P(a_j)$$

où :

— n_j : l'enfant du noeud parent n.

— $Q(n)$: la probabilité du nœud n. Par convention, la probabilité de la racine est $Q(n_0) = 1$.

— $Q(n_j)$: la probabilité du nœud n_j.

3.2.2 L'algorithme de Tunstall

Les algorithmes 1 et 2 décrivent la procédure de construction du code de Tunstall [27], où :

— M : le nombre désiré de mots de code dans l'arbre.

— A : la taille de l'alphabet.

— V : contient la liste des nœuds incomplets dans l'arbre T c'est-à-dire les feuilles.

— n_{max} : correspond au nœud le plus probable présent dans V.

— W : correspond aux chemins vers les nouveaux noeuds issus de n_{max}.

— $T + W$: réfère à l'ajout dans l'arbre T de tous les nouveaux noeuds dont les chemins sont donnés par W.

— π : correspond à la concaténation du noeud le plus probable n_{max} avec les noeuds issues de n_{max}.

— $\#T_n$: correspond au nombre de mots de code consommé dans l'arbre T_n.

Algorithme 1 : Option I

ENTRÉE: $M \geq A$
ENTRÉE: $T \leftarrow \{\pi(\text{racine}).a \mid a \in A\}$
1: $V \leftarrow$ noeuds incomplets dans T sauf la racine
2: $n_{max} \leftarrow argmax_{n \in V}\ Q(n)$
3: $W \leftarrow \{\pi(n_{max}).a \mid a \in A\}$ {Les enfants de n_{max}}
4: **retourner** $T + W$

Algorithme 2 : Procédure de Tunstall avec M mots de code

ENTRÉE: $M \geq A$
1: $T_n \leftarrow racine$
2: **répéter**
3: $T_o \leftarrow T_n$
4: $T_n \leftarrow$ Option I (T_o)
5: **jusqu'à** $\#T_n > M$
6: **retourner** T_o

Tout d'abord, nous construisons la racine n_0. Celle-ci a une probabilité de $Q(n_0) = 1$. Ensuite, nous créons les A noeuds $j = \{1, 2, 3, \ldots, A\}$ à partir de la racine. Nous attribuons à chaque enfant créé sa probabilité et un mot de code. De ce fait, chaque itération crée A nœuds à l'arbre mais nous ajoutons $A - 1$ feuilles. Autrement dit, si un nœud possède A enfants, nous pouvons créer le A-ième enfant en utilisant $A - 1$ mots de code, en déplaçant le mot de code qui était assigné au noeud parent vers l'A-ième enfant créé.

Suite à la création de A noeuds depuis la racine, nous choisissons la feuille n la plus probable et nous créons tous ces enfants. Puis nous assignons à chaque nouveau nœud créé sa probabilité $Q(n_j) = Q(n) \times P(a_j)$. Nous répétons ces itérations jusqu'à ce que la taille maximale du dictionnaire soit atteinte.

3.2.3 Exemple

Prenons l'exemple suivant pour illustrer l'algorithme de Tunstall. Cet exemple a été extrait de l'article de Yamamoto et Yokoo [31].
Soit A notre alphabet source $A = \{a_1, a_2, a_3\}$ et soient $p(a_1) = 0.6$, $p(a_2) = 0.3$, $p(a_3) = 0.1$ les probabilités associées à chaque symbole. Supposons que la taille du dictionnaire est fixée à $M = 7$.

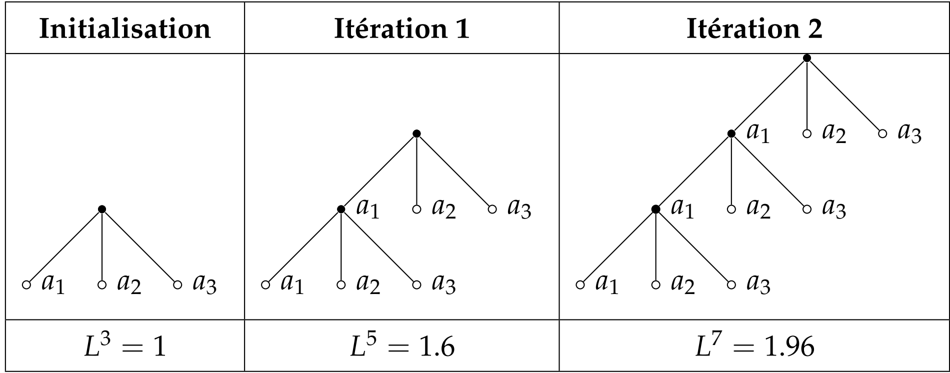

FIGURE 3.1 – L'algorithme de Tunstall où M=7

— L^N : longueur moyenne des mots du dictionnaire chez l'arbre T^N.
— N : indique le nombre de mots de code consommés chez l'arbre T^N.

Tableau 3.1 – Le dictionnaire de Tunstall

Mots du dictionnaire	**Probabilités**	**Mots de code**
$a_1a_1a_1$	0.216	A
$a_1a_1a_2$	0.108	B
$a_1a_1a_3$	0.036	C
a_1a_2	0.18	D
a_1a_3	0.06	E
a_2	0.3	F
a_3	0.1	G
_	_	H

La figure 3.1 illustre les différentes itérations de l'algorithme de Tunstall pour construire l'arbre des mots du dictionnaire. À chaque itération, nous choisissons la feuille a_1 puisqu'elle est la plus probable et nous créons tous ses enfants. Comme l'arbre doit être complet, chaque nœud dans l'arbre peut avoir 0 ou 3 enfants. Il n'est pas possible d'avoir 1 ou 2 enfants. En outre, les 7 mots de code sont associés aux différentes feuilles de l'arbre. Nous présentons les feuilles qui possèdent les mots de code par des cercles vides. En revanche, les nœuds internes sont représentés par des cercles noirs. Ces noeuds ne possèdent pas de mots de code.

Nous constatons que le code de Tunstall crée un arbre complet et propre. Aussi la propriété de préfixe, nous a permis de construire un code où aucun mot du dictionnaire n'est le préfixe

d'un autre. Par conséquent, il n'y a toujours qu'un et un seul mot du dictionnaire qui peut être combiné avec une entrée de dictionnaire

L'inconvénient de la technique de Tunstall réside dans le fait qu'il faut créer 0 ou 3 enfants pour un alphabet ternaire. Dans ce cas, nous pouvons nous retrouver face à des situations où il y a des entrées du dictionnaire inutilisées, comme serait le cas du code H si nous nous permettrions d'utiliser jusqu'à 8 mots de code, tel qu'il est présenté au tableau 3.1.

Supposons que nous voulons encoder la chaîne $X = a_1a_3a_2a_1a_3$ en utilisant l'algorithme de Tunstall. La phase d'encodage repose sur la recherche des mots du dictionnaire les plus longs qui correspondent à un préfixe de X. Nous remplaçons ces mots du dictionnaire par les mots de code qui leur correspondent. Par conséquent, la chaîne X est encodée par la concaténation des mots de code suivant $X' = EFE$.

3.3 Les codes variables à fixes quasi-instantanés AIVF

3.3.1 Principe des codes AIVF

Telle qu'elle est présentée dans la section ci-dessus, la technique de Tunstall produit un code optimal, si nous nous restreignons à des codes VF préfixes. En revanche, si nous enlevons cette restriction, cette technique n'est pas optimale.

Plusieurs travaux de recherche ont démontré que la propriété de préfixe n'est pas obligatoire pour les codes VF. Plus précisément, si nous éliminons l'obligation de respecter cette propriété, nous pouvons créer un dictionnaire qui contient des mots qui sont les préfixes d'autres mots. Ceci permet parfois de construire des codes plus performant que celui de Tunstall. Parmi ces travaux, nous citons celui de Savari [21], ainsi que les travaux de Yamamoto et Yokoo [31].

En 2001, Hirosuke Yamamoto et Hidetoshi Yokoo ont présenté dans leur article un nouveau type de code nommé les codes VF quasi-instantanés ou en anglais *Almost Instantaneous VF* (AIVF) . C'est-à-dire que les nœuds internes incomplets et les feuilles sont associés à des mots de code. L'arbre AIVF définit par Yokoo et Yamamoto ne respecte pas nécessairement la propriété de préfixe. Pour cette raison, leur technique a permis de surpasser la longueur moyenne par rapport à la technique de Tunstall dans certaines situations.

La technique de Yamamoto et Yokoo propose deux algorithmes pour construire des codes dans deux modes différents. Le mono-arbres ou l'arbre par défaut est nommé T_0 tout au long de ce mémoire. Cet arbre ne dépend d'aucune condition d'entrée. C'est-à-dire, il ne dispose d'aucune connaissance sur le contenu à venir de la chaîne source. En revanche, les multi-arbres sont spécialisés pour les contextes où nous savons que certains symboles sont impossibles. Plus précisément, nous pouvons se retrouver dans des situations où nous sa-

vons que le prochain symbole de l'entrée ne peut pas être a_1, a_2, ... ,a_i. Dans un tel cas, nous utilisons l'arbre T_i. Cette connaissance se traduit par l'interdiction de certaines branches. Si nous savons que le prochain symbole de la chaîne source n'est pas a_1 alors c'est avantageux d'utiliser l'arbre T_1. En d'autres termes, T_1 désigne l'arbre où nous interdisons une seule branche, c'est la branche a_1. En effet, i l'indice de l'arbre T_i nous dévoile une information concernant le nombre de symboles interdits dans l'arbre d'analyse.
où :

— T_0 : correspond à l'arbre où nous ne disposons aucune connaissance sur le prochain symbole d'où tous les symboles sont autorisés à partir de la racine et aucune branche n'est interdite.

— T_1 : correspond à l'arbre où nous disposons une information partielle sur le prochain symbole de l'entrée a_1 d'où nous interdisons la branche a_1.

— T_2 : correspond à l'arbre où nous disposons une information partielle sur les deux prochains symboles de l'entrée a_1 et a_2 d'où nous interdisons les deux branches a_1 et a_2.

— T_i : correspond à l'arbre où où nous disposons une information partielle sur les i prochains symboles $a_1, a_2, \ldots, a_i$ d'où nous interdisons i branches $a_1, a_2, \ldots, a_i$.

3.3.2 L'algorithme AIVF

Les algorithmes 3 et 4 décrivent la procédure de construction des codes VF quasi-instantanés d'une façon itérative en s'appuyant sur deux stratégies. Ces stratégies proposent simultanément des ajouts à l'arbre du dictionnaire dans le but de choisir la meilleure des deux propositions.

La première stratégie ou l'option I (Algorithme 3), son principe ressemble à la technique de Tunstall qui a été présenté dans la section 3.2.2. Elle développe tous les enfants du nœud le plus probable. Étant donné que les codes VF quasi-instantanés peuvent comporter des noeuds internes incomplets alors l'option I peut suggérer de compléter ces nœuds, pas seulement les feuilles. L'algorithme ci-dessous décrit cette stratégie, où :

— V : contient la liste des nœuds incomplets dans l'arbre T.

— n_{max} : correspond au nœud le plus probable présent dans V.

— W : correspond aux chemins vers les nouveaux noeuds issus de n_{max}.

— $T + W$: réfère à l'ajout dans l'arbre T de tous les nouveaux noeuds dont les chemins sont donnés par W.

— π : correspond à la concaténation du noeud le plus probable n_{max} avec les noeuds issues de n_{max}.

Algorithme 3 : Option I

ENTRÉE: $M \geq A$
ENTRÉE: $T \leftarrow \{\pi(\text{racine}).a \,|\, a \in A\}$
1: $V \leftarrow$ noeuds incomplets dans T sauf la racine
2: $n_{max} \leftarrow argmax_{n \in V}\ p(n)$
3: $W \leftarrow \{\pi(n_{max}).a \,|\, a \in A\}$ {Les enfants de n_{max}}
4: **retourner** $T + W$

La deuxième stratégie, appelée l'option II (Algorithme 4) propose comme prochain nœud celui qui sera le plus probable parmi tous les enfants pas encore créés. Nous présentons les différents opérateurs et variables qui seront utilisées dans l'algorithme 4, où :

— U : contient les chemins vers tous les nœuds.
— V : correspond à l'extension des chemins de tous les noeuds.
— W : contient les chemins des enfants les plus probables.
— n_{max} : correspond à l'enfant le plus probable présent dans W.
— $T + n_{max}$: l'ajout du nœud le plus probable dans l'arbre T.
— $U.A$: corresponds à la concatenation d'un ensemble de la chaînes avec un ensemble de symboles.

Algorithme 4 : Option II

ENTRÉE: $M \geq A$
ENTRÉE: $T \leftarrow \{\pi(\text{racine}).a \,|\, a \in A\}$
1: $U \leftarrow \{\pi(n) \mid \text{noeud } n \text{ dans } T\}$
2: $V \leftarrow U.A$
3: $W \leftarrow V - U$
4: $n_{max} \leftarrow argmax_{n \in W}\ p(n)$
5: **retourner** $T + n_{max}$ {n_{max} n'ajoute qu'un seul noeud à T}

Enfin, nous présentons l'algorithme principal (Algorithme 5), où nous comparons répétitivement ces deux options et nous gardons celle qui possède la meilleure longueur moyenne.

Algorithme 5 : Comparaison des deux stratégies

ENTRÉE: $0 \leq i \leq A - 2$ { i : nombre de symboles interdits}
ENTRÉE: $M \geq A - i$ { M : taille du dictionnaire}
1: $T_{new} \leftarrow$ racine $+\{a_{i+1}, \cdots, a_A\}$
2: **répéter**
3: $T_{old} \leftarrow T_{new}$
4: $T_I \leftarrow$ Option I (T_{old}) {Quelques mots de code ajoutés}
5: $T_{II} \leftarrow$ Option $\text{II}^{\#T_I - \#T_{old}}$ (T_{old}) {Le nombre de mots de code de T_{II} dépend de T_I}
6: $T_{new} \leftarrow$ L meilleur arbre entre T_I et T_{II}
7: **jusqu'à** $\#T_{(new)} > M$
8: **retourner** Option $\text{II}^{M - \#T_{old}}$ (T_{old})

D'après les algorithmes présentés ci-dessus, nous remarquons que la technique AIVF crée son arbre de code selon trois phases.

— Phase d'initialisation consiste à créer la racine et tous les enfants permis.

— Phase d'exécution des deux stratégies en exécutant les deux options, nous pouvons agrandir l'arbre jusqu'à atteindre notre condition d'arrêt. Les deux stratégies s'exécutent d'une façon equitable en consommant le même nombre de mots de code (voir ligne 5 de l'algorithme 5). Notre condition d'arrêt est reliée à la taille M du dictionnaire.

— Phase d'exécution de la stratégies II jusqu'à la taille M.

Après l'initialisation de la racine avec tous ses enfants, nous commençons l'exécution des deux options. Dans le but d'avoir une comparaison équitable entre les deux propositions, chaque option doit consommer le même nombre de mots de code. Si l'option I cause l'ajout de K mots de code alors l'option II doit s'exécuter K fois avec elle-même. Le choix entre les deux stratégies est basé sur la meilleure longueur moyenne chez les arbres retournés par chacune.

Au premier coup d'oeil, nous pourrions être tentés de croire que l'option II est plus compétitive que l'option I puisqu'elle crée seulement les enfants les plus probables, tandis que l'option I nous oblige à créer tous les enfants issus du noeud le plus probable, même si certains enfants sont peu probables. Toutefois, si un nœud interne possède $A - 1$ enfants, la première stratégie nous suggère de créer l'enfant manquant, tout en déplaçant le mot de code qui était assigné au parent vers cet enfant. Le fait de créer le A-ième enfant sans consommer de mot de code, rend la première stratégie plus avantageuse que la deuxième à certaines occasions. De cette façon, nous améliorons la longueur moyenne sans avoir besoin d'un mot de code supplémentaire.

3.3.3 Exemple

Prenons l'exemple suivant. Soit A notre alphabet source $A = \{a_1, a_2, a_3\}$ et soient $p(a_1) = 0.6$, $p(a_2) = 0.3$, $p(a_3) = 0.1$ les probabilités associées à chaque symbole. supposons que la taille du dictionnaire est fixée à $M = 7$. La figure 3.2 illustre les différentes étapes effectuées pour construire l'arbre T_0 en mode mono-arbre selon la méthode de Yamamoto et Yokoo, c'est-à-dire $i = 0$.

À l'initialisation, nous créons la racine avec tous ses enfants. À l'itération 1, nous commençons l'exécution des deux options. L'option I propose de créer tous les enfants du noeud a_1 puisqu'il est le plus probable. Nous créons les noeuds $\{a_1a_1, a_1a_2, a_1a_3\}$ (figure 3.2). Par contre, nous ne consommons que deux mots de code, en déplaçant le mot de code assigné au noeud a_1 vers le noeud a_1a_3. Aussi, l'option II s'exécute deux fois d'une façon consécutive afin de consommer le même nombre de mots de code que l'option I, d'ou elle propose de prolonger deux fois la branche a_1. Suite à l'exécution des deux stratégies d'une façon équi-

table, nous procédons au choix de celle qui possède la plus grande longueur moyenne. Dans notre exemple, c'est l'option I, sa longueur moyenne est égale à $L_0^5 = 1.60$. Dans les itérations suivantes, nous continuons d'accroître l'arbre T_0 jusqu'à atteindre la taille maximale du dictionnaire.

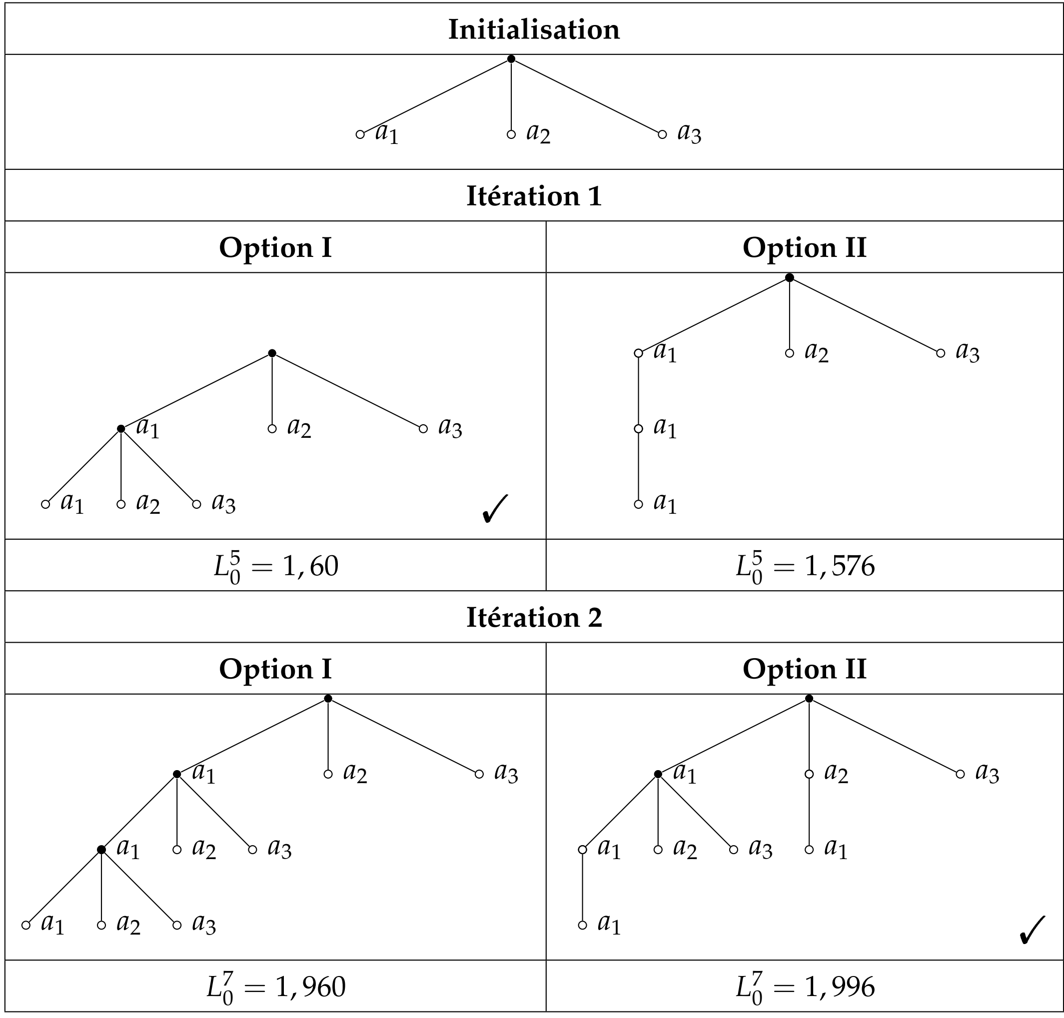

FIGURE 3.2 – L'arbre T_0 généré par l'algorithme de Yamamoto et Yokoo [31]

L'arbre T_0 de la figure 3.2 est différent de l'arbre de Tunstall de la figure 3.1. Nous remarquons que T_0 a la meilleure longueur moyenne $L_0^7 = 1.996$ (figure 3.2) comparativement à l'arbre de Tunstall, qui a $L^7 = 1.960$ (figure 3.1).

Le tableau 3.2 correspond à la liste des mots du dictionnaire, les probabilités de l'entrée ainsi que les mots de code générés par l'arbre T_0. Notons que le dictionnaire associé à cet arbre ne satisfait pas la propriété d'être préfixe. Cette propriété a été abandonnée au niveau des nœuds $\{a_1a_1, a_1a_1a_1\}$, $\{a_2, a_2a_1\}$. Le fait d'abandonner la propriété de préfixe peut entrainer

plusieurs mots du dictionnaire qui peuvent encoder un préfixe de l'entrée.

Tableau 3.2 – Le dictionnaire AIVF en mode mono-arbre T_0

Mots du dictionnaire	**Probabilités de l'entrée**	**Mots de code**
$a_1a_1a_1$	0.216	A
a_1a_2	0.18	B
a_1a_3	0.06	C
a_1a_1	0.36	D
a_2a_1	0.18	E
a_2	0.3	F
a_3	0.1	G

Prenons l'exemple de la chaîne $X = a_1a_1a_3a_2a_1a_3$ que nous voulons encoder en utilisant T_0. L'encodage de X repose sur la recherche des mots du dictionnaire les plus longs qui correspondent à la chaîne X. Ensuite, nous remplaçons ces mots du dictionnaire par les mots de code qui lui conviennent. Nous commençons par substituer a_1a_1 par D, puis a_3 par G. Une fois parvenus à a_2 nous nous retrouvons face à deux possibilités : encoder seulement a_2 ou encoder a_2a_1. Nous choisissons a_2a_1 puisqu'il réfère au mot du dictionnaire le plus long. Nous remplaçons a_2a_1 par E. Dans une dernière étape, nous substituons a_3 par G. Par conséquent, la chaîne encodée en utilisant la technique de Yamamoto et Yokoo est $X' = DGEG$.

Yamamoto et Yokoo [31] ont proposé une autre construction de code en mode multi-arbre. Cette construction est utilisée lorsque nous possédons certaines informations à propos du prochain symbole de l'entrée. L'encodage de la chaîne a_1a_1 en mode multi-arbre débute par l'utilisation de l'arbre T_0. Nous décrivons les étapes d'encodage comme suit : nous considérons que le premier symbole de la source est a_1. Le second symbole aussi est a_1. Par contre, le troisième peut être un des trois symboles suivants soit : $\{a_1, a_2, a_3\}$. Si le troisième symbole est a_1 alors il est plus avantageux de choisir le mot de code assigné au noeud $a_1a_1a_1$, sinon nous émettons le mot de code attribué au nœud a_1a_1. L'utilisation du mot de code de a_1a_1, nous dévoile certaines informations à propos du prochain symbole. nous savons qu'il ne peut être que a_2 ou a_3. Forts de cette connaissance, nous pouvons spécialiser notre prochain arbre d'analyse. Dans ce contexte, nous utiliserons l'arbre T_1. Dans celui-ci, nous interdisons la branche a_1 à partir de la racine et nous autorisons tous les symboles dans les niveaux inférieurs tel qu'il est proposé dans la figure 3.3.

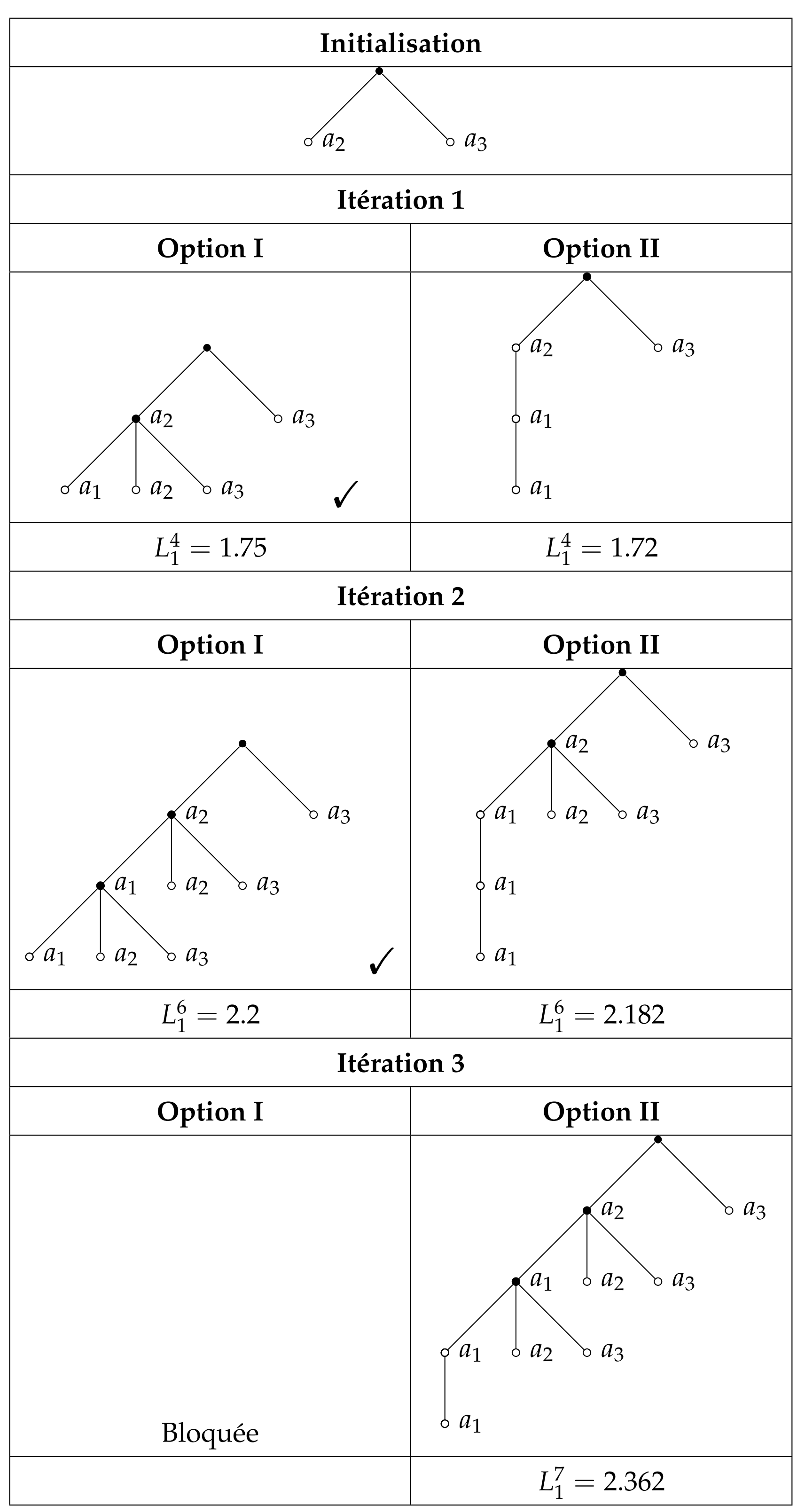

FIGURE 3.3 – l'arbre T_1 généré par l'algorithme de Yamamoto et Yokoo [31]

Le tableau 3.3 correspond à la liste des mots du dictionnaire, les probabilités ainsi que les mots de code générés par l'arbre T_1.

Tableau 3.3 – Le dictionnaire AIVF en mode multi-arbre T_1

Mots du dictionnaire	**Probabilités de l'entrée**	**Mots de code**
$a_2a_1a_1$	0.27	A
$a_2a_1a_2$	0.13	B
$a_2a_1a_3$	0.04	C
a_2a_2	0.22	D
a_2a_3	0.075	E
a_3a_1	0.15	F
a_3	0.25	G

Prenons l'exemple de la chaîne $X = a_2a_2a_1a_2$ que nous voulons encoder en mode multi-arbre. Nous utilisons l'arbre T_0 (tableau 3.2) pour commencer l'encodage vu que nous ne disposons aucune connaissance sur le début de la chaîne. À partir du tableau 3.2, nous déduisons qu'il n'existe pas de meilleur mot du dictionnaire en terme de longueur que a_2, d'où l'encodeur envoie le mot de code F. L'encodage du symbole a_2 nous donne un indice à propos du deuxième symbole celui-ci ne peut pas être a_1. Par contre, il peut être soit a_2 ou a_3. En s'appuyant sur cette connaissance, nous nous servirons de T_1 (tableau 3.3) pour encoder la suite de la chaîne X. Nous choisissons le mot de code B pour encoder $a_2a_1a_2$. L'encodage de la chaîne X est $X' = FB$.

3.4 État de l'art

La technique de Tunstall [27] se situe parmi les premiers algorithmes proposés pour les codes VF. Elle associe une chaîne de taille variable à un mot de code de taille fixe. Son principe s'appuie sur la création d'un dictionnaire de mots de code sous forme d'arbre. Le dictionnaire de Tunstall fournit un code complet obéissant à la propriété de préfixe. L'algorithme de Tunstall [27] construit son arbre en se basant sur la création de tous les nœuds issus de celui qui est le plus probable. Nous continuons à agrandir l'arbre jusqu'à atteindre la taille maximale du dictionnaire. En revanche, dans la catégorie des codes FV, nous trouvons la technique de Huffman [18] qui permet de construire un code FV optimal. L'algorithme de Huffman s'appuie sur la probabilité d'apparition des symboles pour encoder une source. Autrement dit, les symboles les plus probables sont encodés sur moins de bits et les symboles les moins probables sont encodés sur plus de bits.

Depuis plusieurs décennies, divers travaux ont contribué à améliorer la performance de ces

techniques dans le but de les optimiser. Chan [11] et Golin [17] ont choisi d'utiliser le principe de la programmation dynamique pour les codes fixes à variables (FV), plus précisément au niveau de l'algorithme de Huffman [18]. En adaptant cette méthode algorithmique les deux chercheurs ont réussi à réduire sa complexité.

Dans le même contexte des méthodes de programmation, Michael Baer [8] a proposé une nouvelle stratégie de programmation pour la technique de Tunstall [27]. Cette stratégie s'appuie sur le principe de la programmation en temps linéaire. L'algorithme de Tunstall [27] construit son dictionnaire élément par élément. Cette idée a été considérée par Michael Baer [8] comme étant une faiblesse qui augmente le niveau de complexité de la technique de Tunstall [27]. De ce fait, il a proposé une solution en utilisant le principe des files d'attente pour gérer le dictionnaire des mots de code.

Dans d'autres travaux, Yamamoto, Tsuchihashi et Honda [30] ont utilisé le principe de la programmation linéaire en nombres entiers pour améliorer la performance des codes fixes à variables. La technique proposée par ces chercheurs s'appuie sur la création des codes fixes à variables quasi-instantanés (AIFV) dans laquelle nous assignons des mots de code aux noeuds internes incomplets et aussi aux différentes feuilles de l'arbre. En effet, la solution de Yamamoto, Tsuchihashi et Honda [30] a permis de créer des arbres binaires et n-aires ou des mono-arbres et des multi-arbres possédant un taux de compression meilleur que celui de l'algorithme de Huffman. Dans le même axe de recherche, Yamamoto et Iwata [29] ont proposé d'utiliser la méthode de la programmation dynamique au niveau des codes fixes à variables quasi-instantanés (AIFV) pour améliorer leurs performances lors de l'encodage et résoudre les problèmes de stockage. Le principe des files d'attente, la programmation dynamique ou la programmation linéaire en nombres entiers restent toujours des méthodes algorithmiques que nous pouvons utiliser pour optimiser le temps d'exécution d'une technique.

En revanche, certains travaux ont mis en place des solutions techniques évolutives. Savari [21] a présenté une solution pour optimiser la technique de Tunstall [27]. La solution proposée élimine la propriété préfixe pour les codes variables à fixes (VF). Cette amélioration a permis de construire un dictionnaire meilleur que celui de Tunstall [27].

Dans le même contexte, Yamamoto et Yokoo [31] ont construit un code VF quasi instantané. Le principe de la technique de Yamamoto et Yokoo [31] repose sur l'utilisation de deux stratégies qui ont été cité dans la section 3.3.
Yoshida et Kida [32] ont proposé une amélioration pour la technique de Yamamoto et Yokoo [31]. Cette amélioration se base sur la fusion des multiples arbres en un seul arbre compact dans le but de minimiser le temps d'encodage et décodage et de résoudre le problème du stockage. Leur idée s'appuie sur l'observation que plusieurs noeuds dans le mode multi-arbre sont communs et nous pouvons les multiplexer afin de résoudre le problème cité ci-dessus. Ils ont montré aussi, en appliquant des preuves mathématiques, que leur technique fournit

un arbre ayant un nombre de noeuds inférieur à celui produit par l'arbre de Yamamoto et Yokoo [31]. En outre, ils ont prouvé que leur méthode est trois fois plus rapide que celle de Yamamoto et Yokoo [31] lors de l'opération d'encodage et de décodage [32].

À l'instar de Yoshida et Kida [32], il nous a poussés à proposer une nouvelle solution pour résoudre les faiblesses de la technique de Yamamoto et Yokoo [31]. Notre solution s'appuie sur le principe de la programmation dynamique et elle est présentée dans le chapitre 4. Ce principe a été adopté dans différents travaux [11, 29, 17] pour les codes fixes à variables et ces travaux [11, 29, 17] ont montrés que cette méthode de programmation diminue la complexité de leur algorithme.

3.5 Conclusion

Dans ce chapitre, nous avons présenté deux algorithmes pour construction des codes variables à fixe. L'algorithme de Tunstall repose sur la création d'un dictionnaire respectant la propriété d'être préfixe. Par contre, l'algorithme proposé par Yamamoto et Yokoo crée un dictionnaire meilleur que celui de la technique de Tunstall. Ce dictionnaire ne satisfait pas nécessairement la propriété préfixe. Bien que cette solution fournisse un bon taux de compression, elle possède certains inconvénients comme le problème relié à la vitesse d'encodage et de décodage, ainsi que des problèmes liés au stockage. Dans le chapitre suivant, nous présentons les faiblesses de la technique de Yamamoto et Yokoo et nous proposons certains correctifs pour rendre cette technique plus performante.

Chapitre 4

Amélioration de l'encodage variable à fixe

4.1 Introduction

Ce chapitre comporte deux grandes sections. Dans la première, nous présentons les défauts existants dans la technique de Yamamoto et Yokoo, puis nous proposons des solutions pour les corriger. Dans la deuxième section, nous suggérons une nouvelle technique pour construire des codes VF quasi-instantanés plus performants.

4.2 Les défauts de construction du dictionnaire de YY

4.2.1 Le problème de la racine complète dans le mode multi-arbre

La technique de Yamamoto et Yokoo [31] exige la création d'une racine complète lors de l'initialisation. Plus précisément, la racine complète est une nécessité en mode mono-arbre. Par contre, elle était considérée par Yamamoto et Yokoo comme étant une obligation en mode multi-arbre. Cette obligation est du au fait que la technique de Yamamoto et Yokoo permet de construire un code exhaustif. Dans ce contexte, nous avons proposé d'éliminer cette obligation dans le but d'obtenir un code VF meilleur en mode multi-arbre.

Notre amélioration consiste à enlever l'obligation d'utiliser l'**option I** pour créer les enfants de la racine. L'exemple ci-dessous démontre l'existence de cette faiblesse dans la technique de Yamamoto et Yokoo. Nous voulons construire l'arbre T_0 en mode multi-arbre avec l'alphabet $A = \{a, b, c\}$ et la probabilité associée à chaque symbole étant la suivante $p(a) = \alpha$, $p(b) = \beta$, $p(c) = \gamma$ sachant que $\alpha + \beta + \gamma = 1$ sans perte de généralité, nous supposons $\alpha \geq \beta \geq \gamma$ et $M = 4$ est la taille du dictionnaire.

	État initial	Itération 1	Itération 2
YY	•	a b c	a b c a
	$L_0^1 = 0$	$L_0^3 = 0.99$	$L_0^4 = 1.49$
DH	•	a a	a a a
	$L_0^1 = 0$	$L_0^3 = 1.19$	$L_0^4 = 1.53$

FIGURE 4.1 – Le problème de la racine complète en mode multi-arbre

La figure 4.1 illustre les différentes itérations pour construire un dictionnaire de taille 4 en utilisant la technique de Yamamoto et Yokoo ainsi que notre correctif, en faisant certaines hypothèses sur les probabilités des symboles. Les colonnes intitulées YY montrent le résultat suivant l'algorithme de Yamamoto et Yokoo et les colonnes intitulées DH correspondent aux résultats obtenus en appliquant notre algorithme. Le N dans T^N représente le nombre de mots de code consommés dans l'arbre T^N. Pour que la construction se fasse telle qu'illustrée à la figure 4.1 et obtenir une longueur moyenne supérieure à celle de Yamamoto et Yokoo, nous avons besoin d'ajouter certaines conditions sur le choix des probabilités de l'alphabet source avant de construire notre dictionnaire. Ces conditions sont les suivantes.

- $\Omega = \sqrt[3]{\frac{\sqrt{31}}{2} + 3 * \frac{\sqrt{3}}{2}}$
- $\Gamma = \frac{\Omega - \frac{1}{\Omega}}{\sqrt{3}}$
- $\alpha > \Gamma$
- $\Gamma \approx 0,68233$

Supposons que la taille de notre alphabet $A = 3$, $\epsilon = \{a, b, c\}$ où $p(a) = 0.7$, $p(b) = 0.2$ $p(c) = 0.1$. D'après la figure 4.1, nous déduisons que les deux méthodes consomment le même nombre de mots de code $M = 4$.

Notons qu'à l'état initial, T_{YY}^1 n'existe pas réellement dans la technique de Yamamoto et Yokoo. Elle est illustrée uniquement pour maintenir la symétrie de la disposition de la figure 4.1. À l'itération 1, la technique de Yamamoto et Yokoo [31] construit l'arbre en créant tous les enfants depuis la racine. En revanche, notre amélioration propose d'étendre le nœud a

puisqu'il est beaucoup plus probable que les autres symboles. À la fin de l'itération 1, les deux techniques proposées consomment le même nombre de mots de code, soit 3.
La deuxième itération de notre correctif suggère de prolonger le nœud *aa* pour obtenir T_{DH}^4. Notre proposition permet de créer T_{DH}^4, lequel est différent de T_{YY}^4. D'ailleurs, la longueur moyenne de T_{DH}^4 est $L_0^4 = 1.53$. Tandis que celle de T_{YY}^4 est $L_0^4 = 1.49$. Par conséquent, cet arbre possède l'allure d'une ligne, dans laquelle nous favorisons la prolongation du symbole le plus probable dans notre alphabet.

De plus, nous remarquons selon la figure 4.1 que la technique de YY permet d'assigner des mots de code aux noeuds (a, aa, b, c). En revanche, notre technique assigne des mots de code aux noeuds $(racine, a, aa, aaa)$. Le mot de code attribué à la racine permet d'encoder les entrées du dictionnaire qui débute par le symbole b ou c.

4.2.2 Le problème de l'exécution complète de l'option II

Nous avons identifié une autre lacune au niveau de la technique de Yamamoto et Yokoo [31]. Cette lacune concerne l'exécution complète de la proposition faite par l'option II. En effet, l'option II propose d'étendre seulement le nœud le plus probable. Le nombre de répétitions de l'exécution de l'option II dépend du nombre de mots de code consommés par l'option I. Nous savons que la technique de Yamamoto et Yokoo agrandit le dictionnaire de mots de code d'une façon équitable entre les deux stratégies.

Cette contrainte sur le nombre de répétitions de l'exécution des deux options a forcé la deuxième stratégie à créer des nouvelles branches. Ces branches ont des probabilités décroissantes. Comme nous le démontrons juste après, l'exécution complète de la deuxième stratégie rend la technique de Yamamoto et Yokoo sous-optimale. En conséquence, nous avons suggéré une rectification à leur technique. Cette rectification consiste à ne pas exécuter la séquence complète de l'option II. Nous ajoutons seulement la branche la plus probable. L'exemple ci-dessous confirme l'existence de cette deuxième lacune dans l'algorithme de YY et le fait que notre correctif augmente la longueur moyenne par rapport à la technique de Yamamoto et Yokoo. Supposons que nous voulons construire T_0 en mode mono-arbre. Soient $\epsilon = \{a, b, c, d, e\}$ et les probabilités des symboles $p(a) = \frac{1}{3}$, $p(b) = \frac{1}{4}$, $p(c) = \frac{1}{6}$, $p(d) = \frac{3}{20}$, $p(e) = \frac{1}{10}$. Soit $M = 10$ la taille désirée du dictionnaire.

La figure 4.2 illustre les différentes itérations pour construire l'arbre de mots du dictionnaire selon la technique de Yamamoto et Yokoo. À l'initialisation, l'algorithme en mode mono-arbre exige que la racine soit complète. Plus précisément, à l'état initial nous devons créer tous les enfants depuis la racine. À partir de l'itération 1, nous commençons l'exécution des deux options d'une façon équitable. Ensuite, nous gardons l'arbre qui possède la longueur moyenne maximale $L_0^9 = 1.3402\overline{7}$ [1]. Par contre à l'itération 2, nous remarquons qu'il ne reste

1. La barre au-dessus d'un chiffre indique sa partie décimale périodique.

qu'un seul mot de code disponible. Cette situation provoque l'adoption par défaut de la deuxième stratégie.

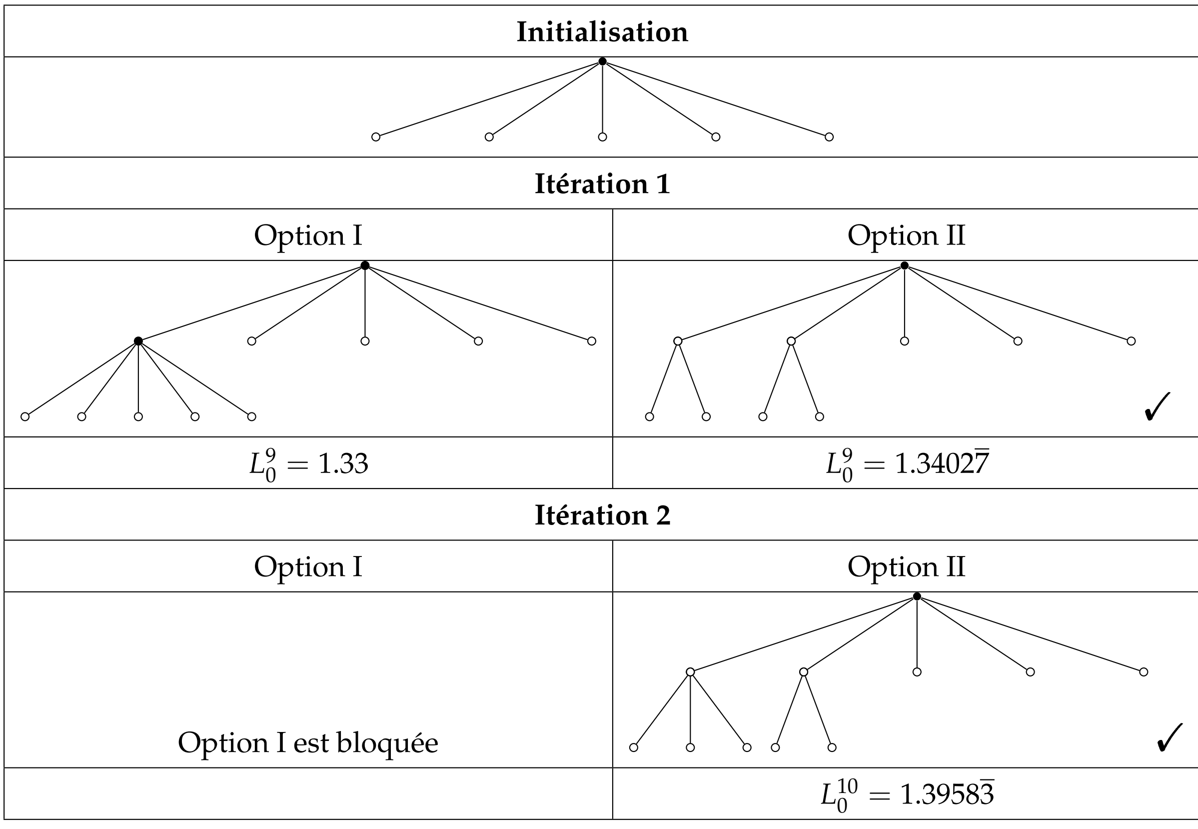

FIGURE 4.2 – T_0 en mode mono-arbre selon la technique de Yamamoto et Yokoo [31]

La figure 4.3 présente l'arbre de mot de code généré selon notre correctif. Au cours des différentes itérations l'option II peut proposer plusieurs branches à la fois. La plus probable est illustrée avec un cercle. En revanche, les autres sont illustrées par des étoiles rouges. À cet égard, notre technique propose de garder seulement la branche la plus probable, soit celle en noir. Nous voyons que l'exécution complète de la deuxième stratégie construit un arbre sous-optimal comparativement à notre correctif. En effet, l'itération finale présentée dans la figure 4.3 construit un dictionnaire possédant une forme d'arbre différente comparée à la technique Yamamoto et Yokoo et aussi une longueur moyenne plus élevée $L_0^{10} = 1.41\overline{6}$ (figure 4.3) par rapport à celle de Yamamoto et Yokoo $L_0^{10} = 1.39583\overline{3}$ (figure 4.2).

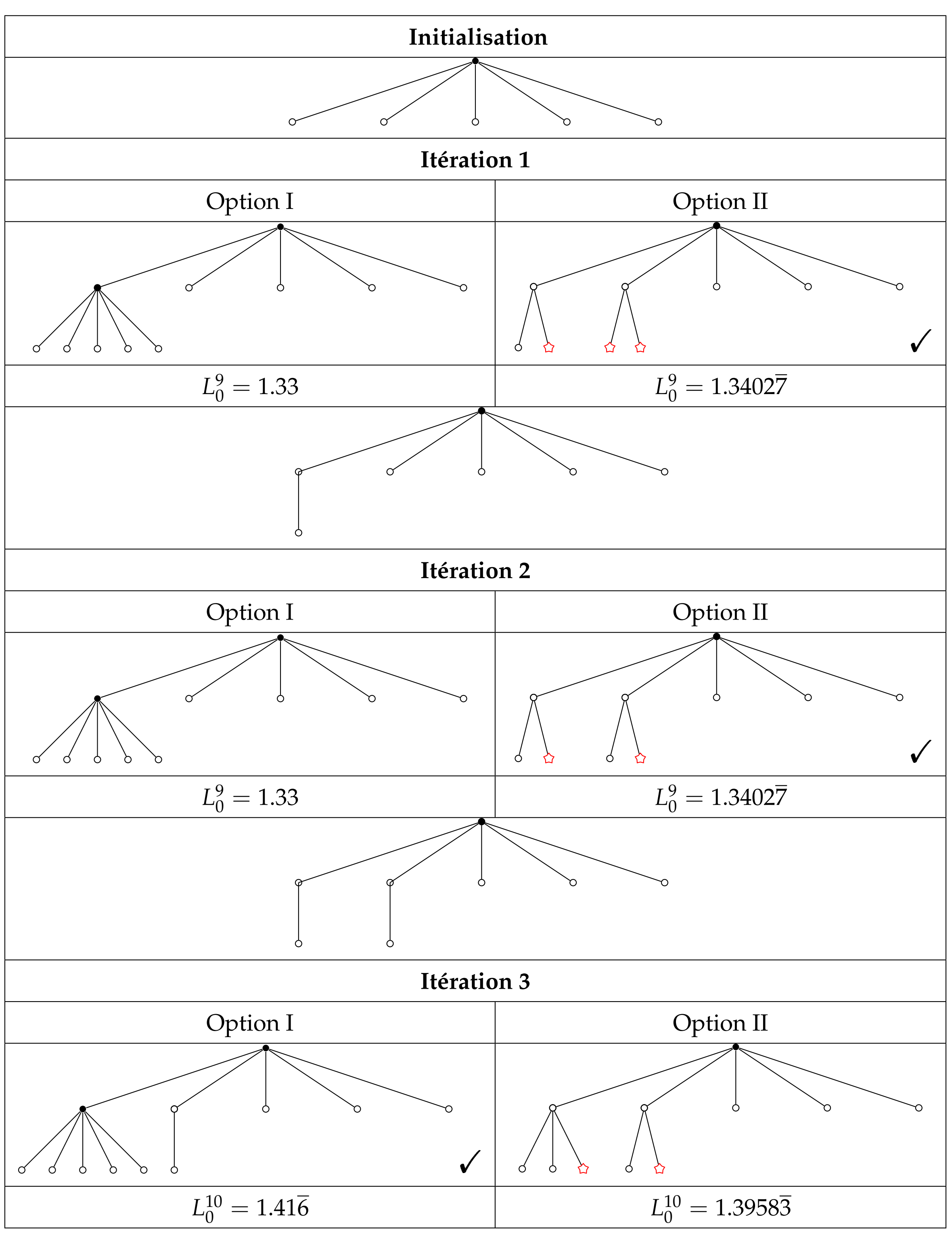

FIGURE 4.3 – T_0 en mode mono-arbre selon le correctif de DH

4.3 La technique de la programmation dynamique en mode multi-arbre

Dans cette section, nous proposons une nouvelle technique pour la construction des codes VF en mode multi-arbre [2]. En effet, notre technique s'appuie sur le principe de la programmation dynamique pour construire des codes quasi-instantanés optimaux. Le dictionnaire généré ressemble à celui de la technique de Yamamoto et Yokoo [31] mais il est construit différemment. Avant d'aborder la présentation de notre technique nous présentons les notations qui sont utilisées dans cette section.

— T_i^N : un arbre T_i avec N mots de code.

— L_i^N : longueur moyenne des mots du dictionnaire chez l'arbre T_i^N.

— i : indique le nombre de symboles interdits.

L'idée principale derrière notre technique s'appuie sur l'exemple suivant. Nous voulons construire l'arbre T_i^N à partir de l'information suivante : $\epsilon = \{a_1, a_2, a_3\}$ désigne notre alphabet et les probabilités associées à ces symboles sont $p(a_1) = 0.6$, $p(a_2) = 0.3$ et $p(a_3) = 0.1$. D'abord, notons que la racine possède nécessairement une branche a_i. Nous pouvons décomposer T_i^N en deux sous-arbres T' et T''.

— T' : désigne le sous-arbre en-dessous de la branche la plus probable a_i.

— T'' : réfère à l'arbre restante si nous éliminons la branche la plus probable a_i et T'.

Nous assignons à chaque sous-arbre un nombre de mots de code. L désigne le nombre de mots de code assignés à l'arbre T' lequel est noté sous ce format T'^L, où $L \geq 1$. R est le nombre de mots de code assignés à l'arbre T'' lequel est noté sous ce format T''^R, où $R \geq 1$. La somme de nombre des mots de code assignés à T' et à T'' doit correspondre au nombre total de mots de code relatifs à l'arbre T_i^N, d'où $N = L + R$. Lorsque nous observons les deux sous-arbres, nous constatons que T' est un arbre qui ne bénéficie d'aucune information sur le prochain symbole source. Donc, T' est un arbre de forme T_0. En revanche, T'' est un arbre qui sait que le premier symbole de la chaîne de source n'est pas a_{i+1}. T'' est un arbre de forme T_{i+1}.

L'objectif de notre approche est de construire l'arbre T_i^N en se basant sur le fait que nous savons construire des arbres T'^L et T''^R optimaux, ce qui est vrai par induction pour T_i^N.
Suite à ce principe, nous présentons nos algorithmes.

2. Pas seulement en mode multi-arbre. Elle peut aussi servir en mode mono-arbre grâce à des modifications mineures.

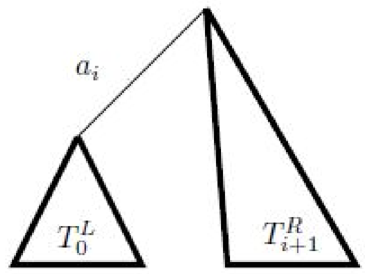

FIGURE 4.4 – Un arbre T_i^N où $i \leq A-2$, $N \geq 2$, $L \geq 1$, $R \geq 1$ et $L + R = N$

Algorithme 6 : Création de l'arbre

ENTRÉE: $N \geq 1$
ENTRÉE: $i \leq A-1$
1: **si** $i = A-1$ **alors**
2: **retourner** T_{A-1}^N {où T_{A-1}^N a la forme présentée dans la figure 4.7}
3: **sinon**
4: **si** $i \leq A-2$ **alors**
5: **si** $N = 1$ **alors**
6: **retourner** T_i^1 {où T_i^1 a la forme présentée dans la figure 4.5}
7: **sinon**
8: **retourner** $T_i^N \leftarrow argMax_{t \in \mathcal{T}_i^N}$ LongMoy (t)
9: $\mathcal{T}_i^N = \{T_0^L \oplus T_{i+1}^R | L + R = N$, où $L \geq 1, R \geq 1\}$
10: **finsi**
11: **finsi**
12: **finsi**

Algorithme 7 : Programmation dynamique

ENTRÉE: $M \geq 1$
1: **pour** $N = 1$ **à** M **faire**
2: **pour** $i = 0$ **à** $A-1$ **faire**
3: $T_i^N \leftarrow$ Création de l'arbre (N, i)
4: **fin pour**
5: **fin pour**

L'algorithmes 6 et 7 décrivent les différentes étapes de création d'un arbre en utilisant la méthode de programmation dynamique. Nous remarquons qu'ils se basent sur la condition d'arrêt relatif au nombre de mots de code restant N pour construire l'arbre T_i^N.
En effet, l'algorithme 6 s'appuie sur deux conditions :

— $i \leq A-2$: nous disposons d'une information partielle à propos du prochain symbole,

ce symbole peut être l'un de $a_{i+1} \ldots a_A$. Ceci nous permet d'identifier deux structures d'arbre.

- — $N = 1$: réfère à un arbre avec seulement un nœud racine, tel qu'il est illustré dans la figure 4.5. Cette structure d'arbre reconnaît un manque de connaissances vis-à-vis du le prochain symbole.

$\circ$

FIGURE 4.5 – L'arbre T_i^1 pour $i \leq A - 2$

- — $N > 1$: c'est le cas récursif où nous devons tester toutes les possibilités pour construire l'arbre gauche T_0^L et l'arbre droit T_{i+1}^R.

— $i = A - 1$: Nous disposons d'une information totale à propos du prochain symbole. Ce symbole ne peut être que a_A. Ceci permet de créer un nœud racine avec une seule branche, c'est l'arbre T_{A-1}^N où nous interdisons $A - 1$ branches à partir du nœud racine. La structure de l'arbre correspond à la figure 4.6 où nous avons une connaissance parfaite du prochain symbole.

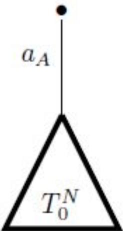

FIGURE 4.6 – L'arbre T_{A-1}^N

Pour clarifier le principe de l'algorithme, nous reprenons l'exemple vu précédemment pour présenter les différentes itérations effectuées par notre technique afin de construire l'arbre T_i^N. Supposons que notre alphabet est $A = \{a_1, a_2, a_3\}$ et les probabilités associées à ces symboles sont $p(a_1) = 0.6$, $p(a_2) = 0.3$ et $p(a_3) = 0.1$. Notre dictionnaire est de taille 3.

La figure 4.7 montre les différentes itérations pour la création de l'arbre de mots de code en utilisant la technique de programmation dynamique. En effet, T_i^1 , T_i^2, T_i^3 désigne l'arbre T_i avec le nombre indiqué de mots de code. En revanche, la notation T_1^N réfère à l'arbre où la branche a_1 est interdite à partir de la racine. T_2^N correspond à l'arbre où les branches a_1 et a_2 sont interdites. L'arbre T_2^1 est décrit par un seul mot de code puisqu'il s'agit d'une connaissance parfaite du prochain symbole de notre source. Autrement, au niveau de l'arbre

T_2^1 nous interdisons les branches a_1 et a_2 à partir de la racine alors notre mot de code sera assigné au noeud du sous-arbre. En revanche, l'arbre T_0^1 identifie qu'il y a un manque de connaissances sur le prochain symbole. Étant donné que nous avons un seul mot de code disponible et que nous avons besoin d'avoir un dictionnaire complet, nous attribuons ce mot de code au nœud racine. Finalement, les arbres T_1^3 et T_0^3 nous présentent les cas qui nécessitent une recherche pour identifier les différentes possibilités de construction des sous-arbres avec les mots de code disponibles et nous gardons en définitive l'arbre qui possède la meilleure longueur moyenne.

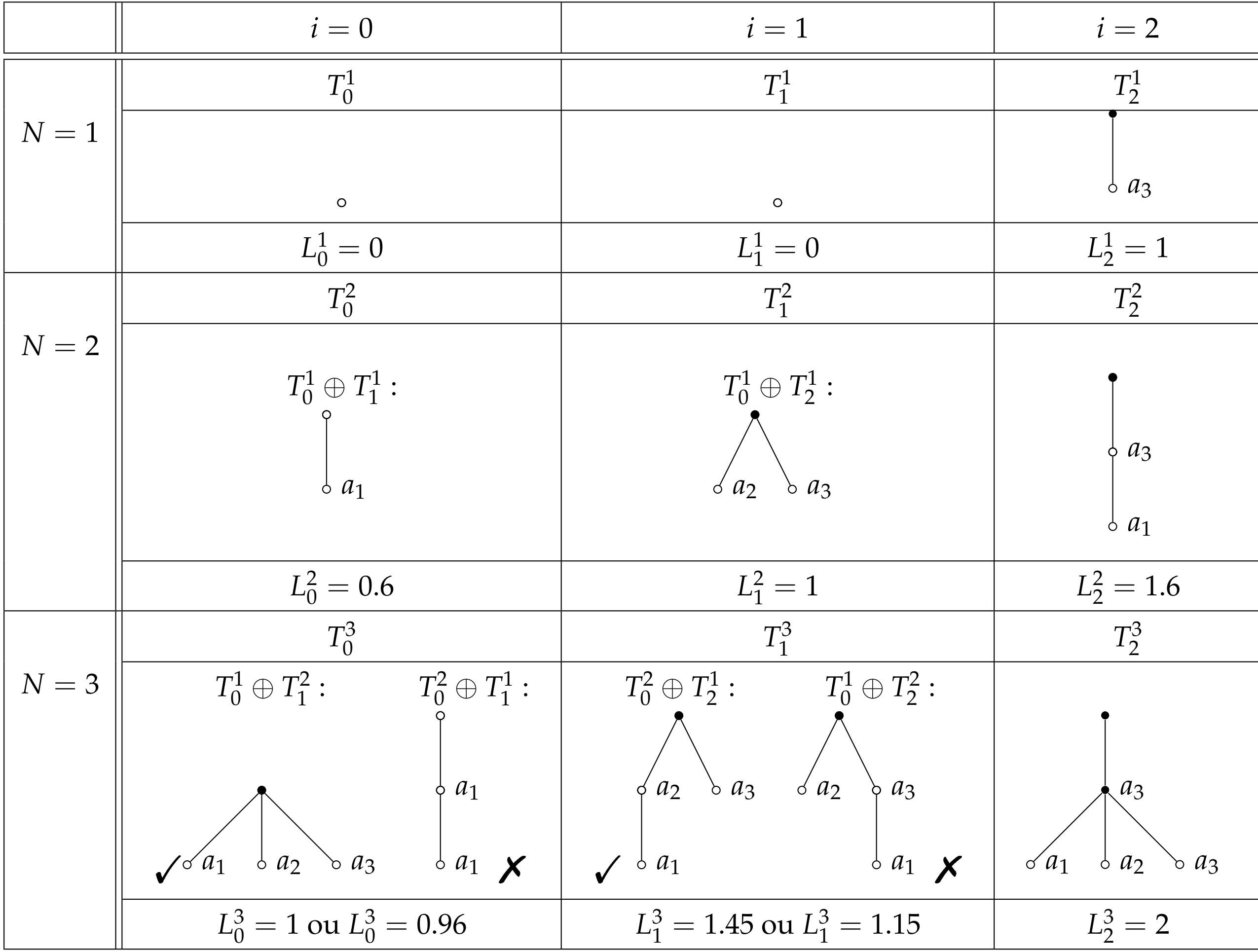

FIGURE 4.7 – Construction de dictionnaire en utilisant la programmation dynamique

4.4 Étude comparative

Dans cette section, nous présentons une étude comparative entre notre solution et la solution de Yamamoto et Yokoo [31] afin de valider l'efficacité de nos correctifs.

4.4.1 Le problème d'une racine complète

Notre correctif relatif au problème de la racine complète propose d'enlever l'obligation d'utiliser l'option I dans la création d'une racine complète en mode multi-arbre, en ajoutant certaines hypothèses sur les probabilités des symboles. Par contre, la solution de Yamamoto et Yokoo impose la création d'une racine complète lors de l'initialisation. Afin de valider l'efficacité de notre technique, nous proposons l'exemple ci-dessous où l'alphabet source est le suivant $A = \{a, b, c\}$ où $p(a) = 0.8$, $p(b) = 0.1$ et $p(c) = 0.1$ et la taille du dictionnaire est $M = 4$.

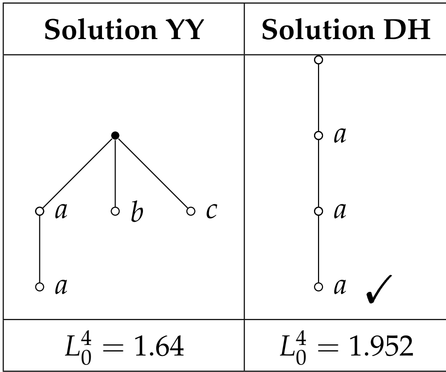

FIGURE 4.8 – Problème d'une racine complète

En éliminant cette obligation, nous remarquons à la figure 4.8 que notre solution maximise la longueur moyenne puisqu'elle favorise la prolongation de la branche la plus probable.

4.4.2 Le problème de l'exécution complète de l'option II

Nous avons identifié une autre faiblesse dans la technique de Yamamoto et Yokoo [31] celle qui est relative à l'exécution complète de la deuxième stratégie. La deuxième stratégie peut proposer la création de plusieurs branches. Ces branches sont classées dans un ordre décroissant. En s'appuyant sur cette analyse, nous avons proposé de garder seulement la branche la plus probable.

Nous appliquons ce principe (figure 4.9) sur l'alphabet $A = \{a, b, c, d, e\}$ et les probabilités $p(a) = \frac{1}{3}$, $p(b) = \frac{1}{4}$, $p(c) = \frac{1}{6}$, $p(d) = \frac{3}{20}$, $p(e) = \frac{1}{10}$ avec $M = 10$ comme taille de dictionnaire en mode mono-arbre. Nous déduisons que notre solution augmente la longueur moyenne lorsqu'elle est comparée à celle de Yamamoto et Yokoo.

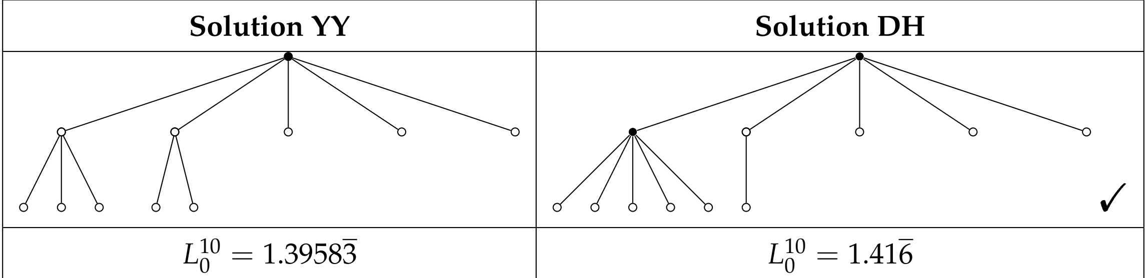

FIGURE 4.9 – Problème de l'exécution complète de l'option II

4.4.3 La technique de la programmation dynamique en mode multi-arbre

Kida et Yoshida [32] ont présenté certains inconvénients relatifs à la technique de Yamamoto et Yokoo [31]. Ces inconvénients réfèrent au temps d'exécution lors des opérations d'encodage et de décodage, et aussi aux problèmes de stockage. Dans ce contexte, nous avons proposé une amélioration s'appuyant sur le principe de la programmation dynamique afin de corriger ces faiblesses. La programmation dynamique a été employée dans divers travaux [11, 17, 29] afin d'optimiser le temps d'exécution des algorithmes.

La figure 4.10 illustre les différentes étapes de construction de T_i^3 en mode multi-arbre où l'alphabet est le suivant : $A = \{a_1, a_2, a_3\}$, les probabilités associées aux symboles sont $p(a_1) = 0.6$, $p(a_2) = 0.3$ et $p(a_3) = 0.1$ et notre dictionnaire est de taille 3. Selon le résultat présenté à la figure 4.10, nous remarquons qu'il n'y a pas de différence entre notre solution et la solution de Yamamoto et Yokoo puisque la longueur moyenne des deux méthodes est égale [31]. Notre technique minimise effectivement l'espace de stockage requis étant donné que nous sauvegardons au maximum un entier pour représenter l'arbre T_i^N (la valeur de L ou de R).

En analysant notre algorithme nous déduisons qu'il existe deux cas :

— $i = A - 1$: T_{A-1}^N est formé d'une racine avec une seule branche a_A, qui nous mène à un sous-arbre T_0^N. Dans cette situation nous n'avons rien à stocker pour mémoriser la forme de l'arbre car tous les détails non-triviaux se trouvent dans T_0^N.

— $i \leq A - 2$: est composé de deux sous-cas.

  — $N = 1$: l'arbre est composé seulement d'un nœud racine, dans ce cas nous n'avons rien à mémoriser pour stocker la forme de l'arbre.

  — $N \geq 2$: l'arbre est composé d'un sous-arbre de gauche T_0^L et d'un reste d'arbre droit T_{i+1}^R. Pour mémoriser la forme de T_i^N, il faut stocker L ou R. En effet, nous avons besoin d'un seul des deux entiers pour représenter T_i^N, pour chaque combinaison de N et de i.

Selon cette analyse, la complexité en espace de notre algorithme est $O(M * A)$, qui réfère au nombre de combinaisons possibles sachant que seulement certaines d'entre elles sont stockées pour décrire récursivement l'arbre T_i^N. En revanche, nous évaluons la complexité en temps à $O(M^2 * A)$.

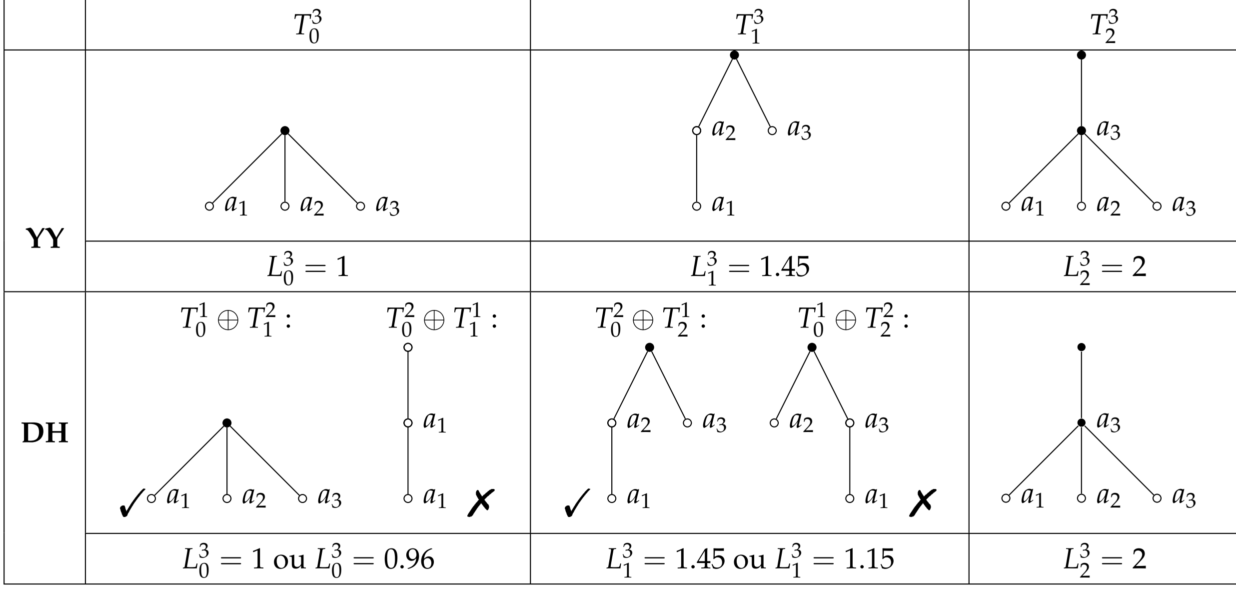

FIGURE 4.10 – Construction de l'arbre en mode multi-arbre en utilisant le principe de la programmation dynamique

4.5 Conclusion

Dans le présent chapitre, dans lequel nous avons présenté les faiblesses de la technique de Yamamoto et Yokoo aussi nous proposons une nouvelle technique de construction des codes AIVF. La première faiblesse correspond à l'obligation de créer une racine complète en mode multi-arbre. En éliminant cette condition nous avons pu construire un dictionnaire meilleur que celui de Yamamoto et Yokoo. La deuxième faiblesse correspond à l'exécution complète de l'option II. En analysant son principe de fonctionnement, nous avons déduit que l'option II propose des branches avec des probabilités décroissantes. Afin de proposer un dictionnaire plus performant, nous avons suggéré de garder seulement la branche la plus probable, et la figure 4.3 a permis de démontrer que notre proposition est meilleure comparée à celle de Yamamoto et Yokoo. Ensuite, nous avons présenté une nouvelle solution simple pour la construction des codes VF quasi-instantanées basée sur la programmation dynamique dans le but d'optimiser le temps d'exécution de la technique de Yamamoto et Yokoo et l'espace de stockage.

En termes des travaux futurs, nous envisageons de valider nos solutions par des preuves

mathématiques dans le but de confirmer leur optimalité.

Chapitre 5

Conclusion

> *"All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value."* **Carl Sagan**

La compression des données est l'art de réduire le nombre de bits requis pour stocker ou transmettre de l'information. Sur ce principe, plusieurs algorithmes de compression, avec ou sans perte ont vu le jour. Parmi ces algorithmes, nous retrouvons la technique d'encodage pour les codes VF quasi-instantanés qui a été inventé par Yamamoto et Yokoo [31]. En effet, la technique Yokoo et Yamamoto [31] repose sur la construction d'un dictionnaire complet, qui ne respecte pas nécessairement la propriété d'être préfixe. Partant de ce fait, Yamamoto et Yokoo [31] ont réussi à fournir un code VF meilleur que celui de Tunstall [27].

La technique de Yamamoto et Yokoo crée le dictionnaire sous forme d'arbre. Leur technique propose deux modes. Le mode mono-arbre est adapté dans les contextes où nous ne disposons d'aucune connaissance sur le contenu du prochain symbole. Par contre, le mode multi-arbre est défini pour les contextes où certaine information est connue sur le contexte du prochain symbole. L'algorithme de Yamamoto et Yokoo utilise deux stratégies pour les deux modes d'arbres. La première stratégie crée tous les nœuds issus du nœud le plus probable, soit le même principe que celui de l'algorithme de Tunstall. Cette stratégie débute toujours l'exécution de l'algorithme de Yamamoto et Yokoo. Par contre, la deuxième stratégie développe seulement le nœud le plus probable parmi les nœuds pas encore créés. Les deux options s'exécutent d'une façon équitable dans le but de consommer le même nombre de mots du dictionnaire. Le choix entre ces deux stratégies est basé sur le critère de la longueur moyenne des mots du dictionnaire.

En s'appuyant sur leur contribution, nous avons fait ressortir que leur technique fournit des codes VF sous-optimaux dans certaines situations. Dans cet axe de recherche, nous avons proposé des correctifs dont l'objectif est d'améliorer la longueur moyenne par rapport à la technique de Yamamoto et Yokoo. Dans le premier correctif nous avons découvert que la

construction obligatoire d'une racine complète est une faiblesse. Partant de cette constatation, nous avons suggéré d'enlever l'obligation d'utiliser l'option I pour créer les enfants de la racine.
Pour avoir un choix équivalent entre les deux stratégies, Yamamoto et Yokoo ont imposée que la deuxième stratégie doit s'exécuter autant de fois que le nombre de mots de code consommés par la première. Ainsi la deuxième stratégie peut proposer l'ajout de plusieurs branches, ces branches ont des probabilités décroissantes. Dans le deuxième correctif, nous avons suggéré de garder seulement la branche la plus probable. Il a été démontré dans le chapitre 4 que ces deux correctifs fournissent une longueur moyenne supérieure comparée à la technique de Yamamoto et Yokoo [31]. De plus, nous avons présenté une nouvelle technique complètement différente de celle de Yamamoto et Yokoo. Cette technique emploie le principe de la programmation dynamique pour créer des arbres optimaux.

Afin de valider ces nouveaux codes, nous visons à fournir des preuves mathématiques pour confirmer que nos correctifs permettent de produire des longueurs moyennes supérieures par rapport à la technique de Yamamoto et Yokoo. Aussi, parmi nos objectifs futurs, nous désirons étudier l'efficacité de notre nouvelle technique pour le mode multi-arbre.

Annexe A

Les tests de mise en application

A.1 Tunstall

L'algorithme de Tunstall avec $A = \{a, b, c\}$, les probabilités des symboles sont $p(a) = 0.6$, $p(b) = 0.3$, $p(c) = 0.1$ et la taille du dictionnaire est $M = 7$.

```
Saisir la taille de l'alphabet :
3
Saisir le nombre de mots de code :
7
Saisir un symbole de l'alphabet :
a
Saisir sa probabilité :
0,6
Saisir un symbole de l'alphabet :
b
Saisir sa probabilité :
0,3
Saisir un symbole de l'alphabet :
c
Saisir sa probabilité :
0,1
Données :
Le symbole : a  probabilité : 0.6
Le symbole : b  probabilité : 0.3
Le symbole : c  probabilité : 0.1
Le nombre de mots de code : 7
Le taille de l'alphabet : 3
################# Arbre final #################
################# La longueur moyenne=1.9600000000000002
Le mot du dictionnaire : aaa  probabilité : 0.216
Le mot du dictionnaire : aab  probabilité : 0.108
Le mot du dictionnaire : aac  probabilité : 0.036
Le mot du dictionnaire : ab  probabilité : 0.18
Le mot du dictionnaire : ac  probabilité : 0.06
Le mot du dictionnaire : b  probabilité : 0.3
Le mot du dictionnaire : c  probabilité : 0.1
```

A.2 Le mode mono-arbre

L'algorithme de Yamamoto et Yokoo [31] en mode mono-arbre avec l'alphabet $A = \{a, b, c\}$, les probabilités des symboles sont $p(a) = 0.6$, $p(b) = 0.3$ et $p(c) = 0.1$ et la taille du dictionnaire est $M = 7$.

```
Saisir la taille de l'alphabet :
3
Saisir le nombre de mots de code :
7
Saisir un symbole de l'alphabet :
a
Saisir sa probabilité :
0,6
Saisir un symbole de l'alphabet :
b
Saisir sa probabilité :
0,3
Saisir un symbole de l'alphabet :
c
Saisir sa probabilité :
0,1
Le symbole : a  probabilité : 0.6
Le symbole : b  probabilité : 0.3
Le symbole : c  probabilité : 0.1
Le nombre de mots de code : 7
Le taille de l'alphabet : 3
 Longueur moyenne de l'option I : 1.6000000000000003
 Longueur moyenne de l'option II : 1.576
 Nous avons choisi  l'option I
 Longueur moyenne de l'option I : 1.9600000000000002
 Longueur moyenne de l'option II : 1.996
 Nous avons choisi  l'option II
################## Arbre final ################
Le mot du dictionnaire : aa  probabilité : 0.36
Le mot du dictionnaire : aaa  probabilité : 0.216
Le mot du dictionnaire : ab  probabilité : 0.18
Le mot du dictionnaire : ac  probabilité : 0.06
Le mot du dictionnaire : b  probabilité : 0.3
Le mot du dictionnaire : ba  probabilité : 0.18
Le mot du dictionnaire : c  probabilité : 0.1
```

A.3 Le mode multi-arbre

L'algorithme de Yamamoto et Yokoo [31] en mode multi-arbre avec l'alphabet $A = \{a, b, c\}$, les probabilités des symboles sont $p(a) = 0.6$, $p(b) = 0.3$ et $p(c) = 0.1$ et la taille du dictionnaire est $M = 7$.

```
Saisir la taille de l'alphabets :
3
Saisir le nombre de mots de code :
7
Saisir un symbole de l'alphabet :
a
Saisir sa probabilité
0,6
Saisir un symbole de l'alphabet :
b
Saisir sa probabilité
0,3
Saisir un symbole de l'alphabet :
c
Saisir sa probabilité
0,1
Données :
Le symbole : a  probabilité : 0.6
Le symbole : b  probabilité : 0.3
Le symbole : c  probabilité : 0.1
Le nombre de mots de code7
La taille de l'alphabet : 3
 Longueur moyenne de l'option I : 1.6000000000000003
 Longueur moyenne de l'option II : 1.576
 Nous avons choisi  l'option I
 Longueur moyenne de l'option I : 1.9600000000000002
 Longueur moyenne de l'option II : 1.996
 Nous avons choisi  l'option II
 Longueur moyenne de l'option I : 1.7499999999999996
 Longueur moyenne de l'option II : 1.7299999999999997
 Nous avons choisi  l'option I
 Longueur moyenne de l'option I : 2.2999999999999993
 Longueur moyenne de l'option II : 2.182
 Nous avons choisi  l'option I
 Longueur moyenne de l'option I : 0.0
 Le nombre de mots de code est insuffisant
 Longueur moyenne de l'option II : 2.362
 Nous avons choisi  l'option II
 Longueur moyenne de l'option I : 1.9999999999999998
 Longueur moyenne de l'option II : 2.199999999999997
 Nous avons choisi  l'option II
 Longueur moyenne de l'option I : 1.9999999999999998
 Longueur moyenne de l'option II : 1.9600000000000002
 Nous avons choisi  l'option I
 Longueur moyenne de l'option I : 2.6
 Longueur moyenne de l'option II : 2.576
 Nous avons choisi  l'option I
 Longueur moyenne de l'option I : 2.9600000000000004
 Longueur moyenne de l'option II : 2.9960000000000004
 Nous avons choisi  l'option II
#################### Arbre final ####################
****** Arbre T_0^7  Longueur moyenne = 1.996
Le mot du dictionnaire : aa  probabilité : 0.36
Le mot du dictionnaire : aaa  probabilité : 0.216
Le mot du dictionnaire : ab  probabilité : 0.18
Le mot du dictionnaire : ac  probabilité : 0.06
Le mot du dictionnaire : b  probabilité : 0.3
Le mot du dictionnaire : ba  probabilité : 0.18
Le mot du dictionnaire : c  probabilité : 0.1
****** Arbre T_1^7  Longueur moyenne = 2.362
Le mot du dictionnaire : baa  probabilité : 0.26999999
Le mot du dictionnaire : baaa probabilité : 0.162
Le mot du dictionnaire : bab  probabilité : 0.13499995
Le mot du dictionnaire : bac  probabilité : 0.04499999
Le mot du dictionnaire : bb
probabilité : 0.224999995
Le mot du dictionnaire : bc  probabilité : 0.075
Le mot du dictionnaire : c  probabilité : 0.25
Le mot du dictionnaire : c  probabilité : 0.15
****** Arbre T_2^7  Longueur moyenne = 2.9960000000004
Le mot du dictionnaire : caa  probabilité : 0.36
Le mot du dictionnaire : caaa  probabilité : 0.216
Le mot du dictionnaire : cab  probabilité : 0.18
Le mot du dictionnaire : cac  probabilité : 0.06
Le mot du dictionnaire : cb  probabilité : 0.3
Le mot du dictionnaire : cba  probabilité : 0.18
Le mot du dictionnaire : cc  probabilité : 0.1
```

Bibliographie

[1] «Codes, codages et compression», "http://igm.univ-mlv.fr/~nicaud/poly/L1_3.pdf". [Consultée 25-Octobre-2016].

[2] «Data Growth, Business Opportunities, and the IT Imperatives», "http://belgium.emc.com/leadership/digital-universe/2014iview/executive-summary.htm". [Consultée 25-Septembre-2016].

[3] «La compression de données», "http://d.nouchi.free.fr/TER/c4.html". [Consultée 25-Juillet-2016].

[4] «État des lieux 2015 : Internet et les réseaux sociaux, en France et dans le monde», "http://www.blogdumoderateur.com/etat-des-lieux-2015-internet-reseaux-sociaux/". [Consultée 25-Juillet-2016].

[5] «Théorie et codage de l'information», "http://www.cedric-richard.fr/coursIT/transparent-c2.pdf". [Consultée 25-decembre-2016].

[6] Ahmad, A.-R. 2016, *Arithmetic Bit Recycling Data Compression*, thèse de doctorat, Université Laval.

[7] Arimura, M. et K.-i. Iwata. 2010, «The minimum achievable redundancy rate of fixed-to-fixed length source codes for general sources», dans *Information Theory and its Applications (ISITA), 2010 International Symposium on*, IEEE, p. 595–600.

[8] Baer, M. B. 2009, «Efficient implementation of the generalized Tunstall code generation algorithm», dans *2009 IEEE International Symposium on Information Theory*, IEEE, p. 199–203.

[9] Beaudoin, V. 2009, *Développement de nouvelles techniques de compression de données sans perte*, mémoire de maîtrise, Université Laval.

[10] Burrows, M. et D. J. Wheeler. 1994, «A block-sorting lossless data compression algorithm», *SRC Research Report*, vol. 124.

[11] Chan, S.-L. et M. J. Golin. 2000, «A dynamic programming algorithm for constructing optimal "1"-ended binary prefix-free codes», *IEEE Transactions on Information Theory*, vol. 46, nº 4, p. 1637–1644.

[12] Cleary, J. et I. Witten. 1984, «Data compression using adaptive coding and partial string matching», *IEEE transactions on Communications*, vol. 32, nº 4, p. 396–402.

[13] Drmota, M., Y. Reznik, S. A. Savari et W. Szpankowski. 2006, «Precise asymptotic analysis of the Tunstall code», dans *2006 IEEE International Symposium on Information Theory*, IEEE, p. 2334–2337.

[14] Dubé, D. , «Codage arithmétique», `"http://www2.ift.ulaval.ca/~dadub100/cours/H16/4003/04Acetates1.pdf"`. [Consultée 25-Septembre-2016].

[15] Dubé, D. , «Compression basée sur les contextes», `"http://www2.ift.ulaval.ca/~dadub100/cours/H16/4003/06Acetates1.pdf"`. [Consultée 29-decembre-2016].

[16] Dubé, D. 2016, «Notes de cours Compression basée sur les contextes», `"http://www2.ift.ulaval.ca/~dadub100/cours/H16/4003/06Acetates1.pdf"`. [Consultée 20-Aout-2016].

[17] Golin, M. J. et G. Rote. 1998, «A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs», *IEEE Transactions on Information Theory*, vol. 44, nº 5, p. 1770–1781.

[18] Huffman, D. A. et collab.. 1952, «A method for the construction of minimum-redundancy codes», *Proceedings of the IRE*, vol. 40, nº 9, p. 1098–1101.

[19] Klein, S. T. et D. Shapira. 2008, «Improved variable-to-fixed length codes», dans *International Symposium on String Processing and Information Retrieval*, Springer, p. 39–50.

[20] Sardinas, A. A. et G. W. Patterson. 1953, «A necessary and sufficient condition for unique decomposition of coded messages», dans *Proceedings Of The Institute Of Radio Engineers*, vol. 41, p. 425–425.

[21] Savari, S. A. 1999, «Variable-to-fixed length codes and plurally parsable dictionaries», dans *Data Compression Conference, 1999. Proceedings. DCC'99*, IEEE, p. 453–462.

[22] Sayood, K. 1996, *Introduction to data compression*, Morgan Kaufmann Publishers Inc.

[23] Shannon, C. E. 2001, «A mathematical theory of communication», *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, nº 1, p. 3–55.

[24] Storer, J. A. et T. G. Szymanski. 1978, «The macro model for data compression», dans *Proceedings of the tenth annual ACM symposium on Theory of computing*, ACM, p. 30–39.

[25] Storer, J. A. et T. G. Szymanski. 1982, «Data compression via textual substitution», *Journal of the ACM (JACM)*, vol. 29, nº 4, p. 928–951.

[26] Taieb, M. H. 2016, «Notes de cours Mathématiques pour la compression sans perte», "http://www2.ift.ulaval.ca/~dadub100/cours/H16/4003/01Acetates2.pdf". [Consultée 08-Aout-2016].

[27] Tunstall, B. P. 1967, *Synthesis of Noiseless Compression Codes*, Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA.

[28] Welch, T. A. 1984, «A technique for high-performance data compression», *Computer*, vol. 17, nº 6, p. 8–19.

[29] Yamamoto, H. et K.-i. Iwata. 2016, «A dynamic programming algorithm to construct optimal code trees of AIFV codes», *IEEE Transactions on Information Theory*.

[30] Yamamoto, H., M. Tsuchihashi et J. Honda. 2015, «Almost instantaneous fixed-to-variable length codes», *IEEE Transactions on Information Theory*, vol. 61, nº 12, p. 6432–6443.

[31] Yamamoto, H. et H. Yokoo. «Average-sense optimality and competitive optimality for almost instantaneous VF codes», *Les IEEE Transactions on Information Theory*, vol. 47.

[32] Yoshida, S. et T. Kida. 2010, «An efficient algorithm for almost instantaneous VF code using multiplexed parse tree», dans *2010 Data Compression Conference (DCC)*, IEEE, p. 219–228.

[33] Yoshida, S. et T. Kida. 2012, «Analysis of multiplexed parse trees for almost instantaneous VF codes», dans *Advanced Applied Informatics (IIAIAAI), 2012 IIAI International Conference on*, IEEE, p. 36–41.

[34] Ziv, J. et A. Lempel. 1977, «A universal algorithm for sequential data compression», *IEEE Transactions on information theory*, vol. 23, nº 3, p. 337–343.

[35] Ziv, J. et A. Lempel. 1978, «Compression of individual sequences via variable-rate coding», *IEEE transactions on Information Theory*, vol. 24, nº 5, p. 530–536.