VINCENT BEAUDOIN

Développement de nouvelles techniques de compression de données sans perte

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise Informatique
pour l'obtention du grade de maître ès sciences (M.Sc.)

Sciences et génie
UNIVERSITÉ LAVAL
QUÉBEC

2008

Résumé

L'objectif de ce mémoire est d'introduire le lecteur à la compression de données générale et sans perte et de présenter deux nouvelles techniques que nous avons développées et implantées afin de contribuer au domaine.

La première technique que nous avons développée est le recyclage de bits et elle a pour objectif de réduire la taille des fichiers compressés en profitant du fait que plusieurs techniques de compression de données ont la particularité de pouvoir produire plusieurs fichiers compressés différents à partir d'un même document original. La multiplicité des encodages possibles pour un même fichier compressé cause de la redondance. Nous allons démontrer qu'il est possible d'utiliser cette redondance pour diminuer la taille des fichiers compressés.

La deuxième technique que nous avons développée est en fait une méthode qui repose sur l'énumération des sous-chaînes d'un fichier à compresser. La méthode est inspirée de la famille des méthodes PPM (prediction by partial matching). Nous allons montrer comment la méthode fonctionne sur un fichier à compresser et nous allons analyser les résultats que nous avons obtenus empiriquement.

À mes parents, Christiane et Benoît,
pour leur soutien durant toutes ces années

Table des matières

Résumé **ii**

Table des matières **iv**

Liste des tableaux **vii**

Table des figures **viii**

1 Introduction **1**

2 Compression de données **3**
2.1 Définition 3
2.1.1 Compression de données sans perte 3
2.1.2 Compression de données avec perte 4
2.2 Concepts généraux 4
2.2.1 Précisions sur les mathématiques employés 4
2.2.2 Entropie 4
2.2.3 Redondance 5

3 Codage **6**
3.1 Définition 6
3.2 Concepts généraux 6
3.2.1 Code 6
3.2.2 Code préfixe 7
3.2.3 Code complet 7
3.3 Technique de codage et algorithme de compression 7

4 Techniques de codage **8**
4.1 Codage de Shannon-Fano 8
4.2 Codage de Huffman 10
4.2.1 Codage de Huffman adaptatif 12
4.3 Codage arithmétique 13
4.4 Code universel 16

4.4.1 Elias gamma 16
4.4.2 Levenshtein 17

5 Algorithmes de compression **19**
5.1 Run-length encoding (RLE) 19
5.2 Algorithmes à base de dictionnaires 19
5.2.1 LZ77 19
5.2.2 LZ78 21
5.3 Burrows-Wheeler transform (BWT) 21
5.3.1 Move-to-front (MTF) 23
5.4 Prediction by Partial Matching (PPM) 24

6 Recyclage de bits **26**
6.1 Introduction 26
6.2 Définitions 27
6.3 Possibilités d'encodages multiples 27
6.3.1 Encodages multiples et conséquences 27
6.3.2 Messages équivalents 28
6.3.3 Messages non-équivalents 28
6.3.4 Lien avec les encodages multiples de fichiers 29
6.4 Recyclage 29
6.4.1 Idée générale de message caché 29
6.4.2 Idée de transmission de parties de fichier compressé dans les messages cachés 30
6.4.3 Recyclage plat 31
6.4.4 Recyclage proportionnel 31
6.4.5 Capacité du canal auxiliaire 32
6.5 LZ77 et recyclage de bits 32
6.5.1 LZ77 conventionnel 32
6.5.2 LZ77 et recyclage de bits 34
6.6 Résultats expérimentaux 35
6.6.1 gzip 35
6.6.2 Expérience 1 37
6.6.3 Expérience 2 40
6.6.4 Expérience 3 41
6.6.5 Expérience 4 42
6.6.6 Expérience 5 43
6.6.7 Expérience 6 44
6.6.8 Expérience 7 45
6.6.9 Recyclage proportionnel optimal 47
6.6.10 Expérience 8 55

6.6.11 Expérience 9 56
6.7 Travaux futurs 62
6.8 Conclusion 62

7 Méthode des papillons **64**
7.1 Introduction 64
7.2 Fonctionnement 64
7.2.1 Étape 1 : Énumération des sous-chaînes 65
7.2.2 Étape 2 : Transmission de l'énumération au décompresseur . . . 68
7.2.3 Étape 3 : Identification de la chaîne de bits originale 72
7.3 Développement du prototype 72
7.4 Mesures empiriques 74
7.5 Analyse des résultats 74
7.6 Conclusion 74

A Corpus **76**

Bibliographie **77**

Liste des tableaux

4.1 Fréquences d'apparition des symboles de l'alphabet Σ 9
4.2 Code de Shannon-Fano pour l'alphabet Σ 9
4.3 Code de Huffman pour l'alphabet Σ 11
4.4 Fréquences d'apparition des symboles de l'alphabet Σ 13
4.5 Première division de l'intervalle pour l'alphabet Σ 15
4.6 Procédure de renormalisation pour l'alphabet Σ 15
4.7 Exemples de mots de code Elias gamma 16
4.8 Exemples de mots de code de Levenshtein 17

5.1 LZ78 sur la chaîne d'octets *ABBCBCABA* 21
5.2 Exemple de transformation BWT 22
5.3 Exemple de transformation MTF 24

6.1 Résultats des expériences 1 à 6 37
6.2 Résultats des expériences 7 à 9 45

7.1 Liste des n rotations triées 65
7.2 Énumération des sous-chaînes 66
7.3 Paramètres du papillon 69
7.4 Résultats du prototype 73

Table des figures

4.1 Algorithme de Shannon-Fano sur l'alphabet Σ 9
4.2 Arbre de Huffman pour l'alphabet Σ 11
4.3 Codage arithmétique pour le message A-C-EOF 13

5.1 Exemple LZ77 20

6.1 Exemple de messages équivalents 28
6.2 Algorithmes LZ77 conventionnel 33
6.3 Algorithmes LZ77 avec recyclage de bits 34
6.4 Exemples de fichiers avec les différentes traversées possibles 57
6.5 Algorithmes pour le recyclage de bits avec tous les messages 60

7.1 Arbre des sous-chaînes 66
7.2 Arbre compact des sous-chaînes 67
7.3 Arbre des sous-chaînes infiniment profond 68
7.4 Un papillon 69
7.5 Exemples de papillons 70

Chapitre 1

Introduction

L'informatique fut créée afin d'automatiser le traitement de l'information. D'ailleurs, cette discipline scientifique tire son nom du terme "information" lui-même.

Depuis les débuts de l'informatique, nous avons été confrontés à la limite des ordinateurs à mémoriser et à transmettre l'information. Tirant parti de l'observation que la presque totalité de l'information que nous manipulons n'est pas aléatoire, des méthodes de compression de données ont été élaborées. Ces méthodes visent à transformer des données en une forme plus compacte (compression), quitte à ce que celles-ci soient inintelligibles, et à les récupérer plus tard sous leur forme originale (décompression). La représentation plus compacte permet d'utiliser moins d'espace lors du stockage ou moins de bande passante lors de la transmission via un quelconque moyen de communication. Bien que les ordinateurs soient toujours plus rapides et possèdent toujours plus de capacité de stockage, l'encombrement dû à la quantité d'information à manipuler ne diminue pas, car nous ne cessons de manipuler des masses d'information de plus en plus considérables. Ainsi, l'utilité des méthodes de compression de données demeure aussi grande qu'à leurs débuts.

On retrouve toutes sortes de méthodes de compression de données. Certaines sont générales tandis que d'autres s'attendent à des données d'une certaine nature (par ex., des images). Certaines effectuent un encodage sans perte et d'autres, avec perte. Les premières ont pour mandat de reconstituer les données originales sans différence aucune après leur avoir fait subir les étapes de compression et de décompression. Les dernières peuvent altérer les données après compression et décompression en autant que les données reconstituées soient relativement semblables aux données originales. Par exemple, dans de nombreuses applications, les documents audio ou vidéo peuvent subir des altérations pourvu que celles-ci soient à l'intérieur de certaines tolérances. Le taux de

compression et la vitesse de traitement varient d'une méthode à l'autre. Tout dépendant de l'application, on préfère soit un traitement rapide, soit la qualité de la compression. Les performances des méthodes quant au taux de compression varient généralement en fonction de deux facteurs : la modélisation et l'encodage. Si ces facteurs influencent tous deux la vitesse de traitement, c'est la modélisation qui a l'effet le plus important sur le taux de compression. Il y a encore place à amélioration en compression de données dans pratiquement toutes ses facettes. De nouveaux perfectionnements continuent d'être présentés régulièrement.

Chapitre 2

Compression de données

2.1 Définition

En informatique, la compression de données est la technique dans laquelle on emploie une paire de fonctions $\mathcal{C}$ et $\mathcal{D}$ sur des chaînes. La fonction $\mathcal{C}$ a pour objectif de compresser les données et la fonction $\mathcal{D}$, de les décompresser. L'effet souhaité est d'avoir $|\mathcal{C}(x)| < |x|$. On ne l'observe pas nécessairement pour tous les x.

Le but de la compression de données est de représenter l'information sous une forme plus compacte. Les données compressées occupent moins d'espace que les données originales : le nombre de bits utilisés pour représenter les données est réduit. Les données peuvent être compressées avec ou sans perte. Dans ce projet de maîtrise, nous nous sommes intéressés exclusivement à la compression de données sans perte.

2.1.1 Compression de données sans perte

La compression est dite sans perte lorsqu'il y a interdiction d'altérer les données lors de la compression. La compression sans perte implique $x = \mathcal{D}(\mathcal{C}(x))$ pour tout x. En d'autres mots, lorsqu'un document original X est compressé en un fichier compressé Y et que Y est par la suite décompressé, nous retrouvons X sans aucune altération. La compression de données sans perte est généralement utilisée lorsque les données ne

doivent subir aucune altération, à cause de leur nature [1] ou à cause du contexte [2] dans lequel s'effectue la compression.

2.1.2 Compression de données avec perte

La compression est dite avec perte lorsqu'il y a permission d'altérer les données lors de la compression. Cette perte d'information permet, entre autres, de compresser davantage les données. La compression de données avec perte est plus appropriée aux données comme l'image et le son, car il est rarement nécessaire de conserver toute l'information disponible à la source. Par exemple, il est très avantageux de compresser le son avec une certaine perte, car l'oreille humaine n'est pas en mesure d'utiliser toute l'information disponible à la source.

2.2 Concepts généraux

2.2.1 Précisions sur les mathématiques employés

- Tous les logarithmes utilisés sont en base 2.
- $\{0,1\}^*$ est l'ensemble des séquences de bits de longueur finie (incluant la longueur 0).
- $\{0,1\}^\infty$ est l'ensemble des séquences de bits de longueur infinie.
- 2^S est l'ensemble puissance de S (l'ensemble de tous les sous-ensembles de S).

2.2.2 Entropie

En informatique, l'entropie désigne la *quantité* d'information que peut représenter ou contenir une source d'information. Ce concept a été introduit en 1948 par Claude Shannon dans l'article 'A Mathematical Theory of Communication' [17]. Ici, une source d'information est généralement un fichier informatique quelconque.

1. Des fichiers exécutables ou des documents à usage inconnu sont des données dont la nature ne permet pas les altérations.

2. Le domaine médical, à cause de règlements ou de législations, et l'astronomie, à cause du coût de l'acquisition de certaines observations, sont des contextes où les données ne doivent subir aucune altération.

Plus une source d'information est redondante, moins elle contient d'information au sens de Shannon. Une source d'information dont tous les symboles sont équiprobables a une entropie maximale.

L'entropie est aussi la plus petite quantité moyenne de bits nécessaire pour communiquer la vraie valeur d'une variable aléatoire. En d'autres mots, l'entropie est la plus petite quantité moyenne de bits capable de représenter l'information tirée de la variable aléatoire. Il s'agit d'une limite mathématique fondamentale pour la compression de données sans perte.

L'entropie d'une variable aléatoire discrète X, qui peut prendre les valeurs $\{x_1, x_2, ..., x_n\}$, est :

$$H(X) = \sum_{i=1}^{n} p(x_i) \log_2(1/p(x_i)) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

où $p(x_i)$ est la probabilité de x_i.

2.2.3 Redondance

La redondance est le rapport entre le nombre de bits utilisé pour stocker des données et le nombre de bits minimal nécessaire pour représenter l'information que ces données renferment. La compression de données sans perte a pour objectif de réduire le plus possible la redondance.

Chapitre 3

Codage

3.1 Définition

Le codage consiste à donner un mot de code à chaque symbole d'un alphabet donné. Une fonction de codage $C : \Sigma \rightarrow \{0,1\}^*$ permet de traduire un symbole de l'alphabet Σ en une séquence de bits, ou mot de code. De plus, grâce au théorème de Shannon sur le codage source [17], nous savons que la longueur optimale d'un mot de code m est $-log(p)$, p étant la fréquence d'apparition du symbole représenté par m.

3.2 Concepts généraux

3.2.1 Code

Un code est une fonction de codage C qui produit un ensemble de mots de code $\{m_1, m_2, \ldots, m_n\}$. Chaque mot de code m représente un symbole s appartenant à un alphabet Σ. Plus précisément, un code produit une représentation alternative d'un alphabet de symboles.

3.2.2 Code préfixe

Un code préfixe est une fonction de codage C capable de produire un ensemble de mots de code de façon à ce qu'aucun mot de code de l'ensemble soit un préfixe d'un autre. On dit aussi de cet ensemble qu'il est exempt de préfixe. Formellement, pour toute séquence de bits infinie $seq \in \{0,1\}^\infty$, il doit y avoir au plus un symbole $s \in \Sigma$ tel que $C(s)$ est un préfixe de seq. L'ensemble $\{0, 10, 11\}$ est le produit d'un code préfixe valide. L'ensemble $\{0, 1, 10, 11\}$ n'est pas le produit d'un code préfixe valide, car les mots de code 1 et 10 sont tous deux préfixes de la séquence de bits 1011001100..., par exemple.

3.2.3 Code complet

Un code est complet si pour toute séquence de bits infinie $seq \in \{0,1\}^\infty$, il existe au moins un symbole $s \in \Sigma$ tel que $C(s)$ est un préfixe de seq. Un code est à la fois préfixe et complet si pour toute séquence de bits infinie $seq \in \{0,1\}^\infty$, il existe un et un seul symbole $s \in \Sigma$ tel que $C(s)$ est un préfixe de seq.

3.3 Technique de codage et algorithme de compression

Dans la littérature, on a souvent tendance à confondre technique de codage et algorithme de compression. Malheureusement, la différence est subtile et parfois, ces deux concepts sont tellement liés qu'il est pratiquement impossible de les différencier. Cependant, une différence demeure. Une technique de codage, comme nous l'avons dit précédemment, sert à donner des mots de code aux symboles d'un alphabet donné. De son côté, un algorithme de compression sert à modéliser les données qui sont compressées et à choisir un codage particulier pour ces données. La modélisation sert généralement à rendre les données plus compressibles. Souvent, la modélisation des données se traduit par une transformation qui est appliquée aux données. Comme nous allons le voir, il existe plusieurs algorithmes de compression qui transforment les données afin de les rendre plus compressibles.

Chapitre 4

Techniques de codage

4.1 Codage de Shannon-Fano

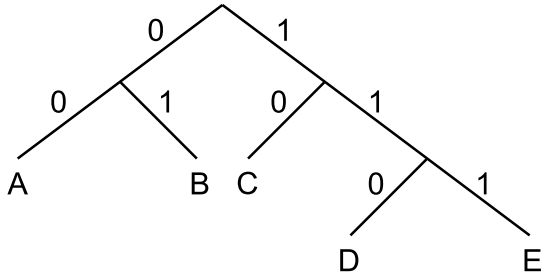
Le codage de Shannon-Fano a été proposé en 1948 par Claude Shannon dans l'article 'A Mathematical Theory of Communication' [17]. L'algorithme pour produire un code de Shannon-Fano pour un alphabet Σ est le suivant :

1. Trier les symboles de l'alphabet Σ en ordre décroissant de fréquence d'apparition (du plus fréquent au moins fréquent).
2. Diviser Σ en deux sous-ensembles Σ_1 et Σ_2 en respectant les deux contraintes suivantes :
   (a) Σ_1 doit contenir les n plus fréquents symboles de Σ et Σ_2 doit contenir les $|\Sigma| - n$ moins fréquents symboles de Σ.
   (b) La différence entre la somme des fréquences d'apparition des symboles de l'ensemble Σ_1 et la somme des fréquences d'apparition des symboles de l'ensemble Σ_2 doit être la plus petite possible. Formellement, nous cherchons les ensembles $\Sigma_1 = \{s_1, s_2, \ldots, s_{j-1}\}$ et $\Sigma_2 = \{s_j, s_{j+1}, \ldots, s_N\}$ tels que :
   $$arg\_min_{1 \leq j \leq N} \left| \sum_{i=1}^{j-1} p(s_i) - \sum_{i=j}^{N} p(s_i) \right|$$
   où $N = |\Sigma|$ et $p(s_i)$ est la fréquence d'apparition de s_i.
3. Attribuer aux symboles dans Σ_1 le bit 0 comme premier bit de leurs mots de code et attribuer aux symboles dans Σ_2 le bit 1.

TABLE 4.1 – Fréquences d'apparition des symboles de l'alphabet Σ

Symbole	A	B	C	D	E
Apparitions	15	7	6	6	5
Fréquence	15/39	7/39	6/39	6/39	5/39

FIGURE 4.1 – Algorithme de Shannon-Fano sur l'alphabet Σ



4. Répéter l'algorithme récursivement, à partir de l'étape 2, sur chacun des ensembles jusqu'à ce qu'il ne reste qu'un symbole dans chaque ensemble. À chaque fois qu'un ensemble est divisé, les symboles appartenant à cet ensemble se voient attribuer un bit supplémentaire à leurs mots de code.

Le tableau 4.1 contient les fréquences d'apparition des différents symboles d'un alphabet $\Sigma = \{A, B, C, D, E\}$. La figure 4.1 illustre un arbre représentant l'application de l'algorithme de Shannon-Fano sur l'alphabet Σ. Les symboles sont représentés par les feuilles de l'arbre. Un noeud interne représente la division d'un ensemble de symboles en deux sous-ensembles. Une branche représente un ensemble de symboles et porte l'étiquette 0 ou 1 servant à former les mots de code des symboles. Le tableau 4.2 présente le code de Shannon-Fano pour l'alphabet Σ.

TABLE 4.2 – Code de Shannon-Fano pour l'alphabet Σ

Symbole	Mot de code	-log(p)
A	00	1.38
B	01	2.48
C	10	2.70
D	110	2.70
E	111	2.96

4.2 Codage de Huffman

Introduit en 1952 par David A. Huffman dans l'article 'A Method for the Construction of Minimum-Redundancy Codes' [10], le codage de Huffman est utilisé intensivement en compression de données. Le codage de Huffman produit un code préfixe optimal [1]. La construction d'un code de Huffman est assez simple. Bien qu'il existe des techniques plus efficaces pour construire un code de Huffman, la technique présentée se veut simple et est souvent utilisée dans la littérature.

La technique utilise les arbres binaires et deux files d'attente. Dans les arbres binaires, les feuilles représentent les symboles d'un alphabet Σ et chacun des noeuds a une probabilité p qui lui est associée. Pour les feuilles, p est simplement la fréquence d'apparition du symbole représenté. Pour les noeuds internes, p est égal à la somme des probabilités des noeuds fils : $p = p_1 + p_2$, où p_1 est la probabilité du fils gauche et p_2, la probabilité du fils droit. Deux files d'attente sont utilisées pour ne pas avoir à retrier les noeuds à chaque fois. La première file, F_1, est utilisée pour les feuilles et la deuxième file, F_2, pour les noeuds internes. L'algorithme pour construire un code de Huffman pour un alphabet Σ est le suivant :

1. Créer une feuille pour chacun des symboles de l'alphabet Σ
2. Ajouter chacune des feuilles, en ordre croissant de probabilité p [2], à F_1
3. Tant que $|F_1| + |F_2| > 1$
   (a) Retirer de F_1 et/ou F_2 les deux noeuds ayant les plus petits p [2]
   (b) Créer un nouveau noeud ayant comme fils les deux noeuds qui viennent d'être retirés des files ; la probabilité du noeud est égale à la somme des probabilités de ses deux fils : $p = p_1 + p_2$, où p_1 est la probabilité du fils gauche et p_2, la probabilité du fils droit
   (c) Ajouter le nouveau noeud à la fin de F_2
4. Le noeud restant est le noeud racine de l'arbre (arbre de Huffman)

Une fois l'arbre construit, nous sommes en mesure d'obtenir le mot de code de chacun des symboles de l'alphabet Σ. Pour ce faire, il suffit simplement d'accumuler les bits dans l'ordre racine $\rightarrow$ feuille (chacune des feuilles représente un symbole). On peut donc retrouver le mot de code associé à un symbole en temps linéaire au nombre de bits qu'il contient.

1. Le codage de Huffman est optimal parmi les codes préfixes qui assignent des mots de code de longueurs entières aux symboles.

2. Il faut assurer un accord entre le codeur et le décodeur quant à la façon de gérer les noeuds de

FIGURE 4.2 – Arbre de Huffman pour l'alphabet Σ

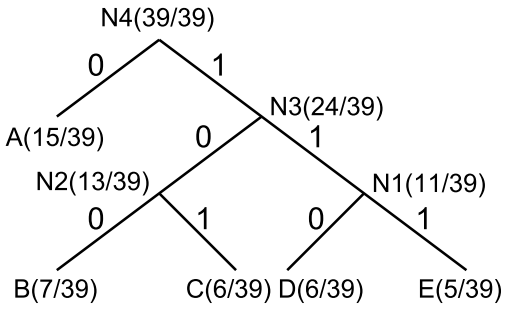


TABLE 4.3 – Code de Huffman pour l'alphabet Σ

Symbole	Mot de code	-log(p)
A	0	1.38
B	100	2.48
C	101	2.70
D	110	2.70
E	111	2.96

La figure 4.2 illustre l'arbre de Huffman pour l'alphabet $\Sigma = \{A, B, C, D, E\}$ (voir tableau 4.1). Les noeuds internes sont étiquetés $N1$, $N2$, $N3$ et $N4$. La probabilité associée à chacun des noeuds est entre parenthèses. Une branche porte l'étiquette 0 ou 1 servant à former les mots de code des symboles qui sont représentés par les feuilles de l'arbre. Le tableau 4.3 présente le code de Huffman pour l'alphabet Σ.

Maintenant, comparons le coût espéré d'un symbole avec le codage de Shannon-Fano (équation 4.1) et le coût espéré d'un symbole avec le codage de Huffman (équation 4.2) :

$$P(A) \times 2 + P(B) \times 2 + P(C) \times 2 + P(D) \times 3 + P(E) \times 3 = 2.28 \ bits \tag{4.1}$$

$$P(A) \times 1 + P(B) \times 3 + P(C) \times 3 + P(D) \times 3 + P(E) \times 3 = 2.23 \ bits \tag{4.2}$$

où $P(x)$ représente la probabilité du symbole x. Le coût espéré d'un symbole avec le codage de Huffman est 2.23 bits et avec le codage de Shannon-Fano, 2.28 bits. Le codage de Shannon-Fano n'est donc pas optimal.

Le codage de Huffman est, quant à lui, un code préfixe optimal ; optimal si les mots

même probabilité.

de code qui sont produits sont composés d'un nombre entier de bits et qu'on n'encode qu'un seul symbole à la fois. En combinant les symboles en courtes séquences, il y a moyens d'atténuer l'effet néfaste de l'intégralité des bits dans les mots de code.

4.2.1 Codage de Huffman adaptatif

Le codage de Huffman requiert de connaître à l'avance les fréquences d'apparition des symboles d'un alphabet Σ pour construire l'arbre qui est utilisé pour obtenir les mots de code. Malheureusement, cela implique que l'encodage doit être fait en deux passes : une passe pour accumuler les statistiques et une passe pour encoder les données. Cet aspect du codage de Huffman peut être problématique quand la rétention des données est difficile ou lorsque la technique est utilisée en temps réel.

Le codage de Huffman adaptatif est une variante du codage de Huffman qui ne requiert pas de connaître à l'avance les fréquences d'apparition des symboles : au fur et à mesure que les données sont lues, l'arbre de Huffman, qui est utilisé pour obtenir les mots de code, est mis à jour. Il n'est donc plus nécessaire de procéder en deux passes pour encoder les données : une seule passe suffit. De plus, un décodeur n'a plus à se faire informer de l'arbre de Huffman qui a été utilisé pour obtenir les mots de code : il peut désormais construire l'arbre de Huffman lui-même en s'y prenant de la même façon que l'encodeur. Évidemment, la technique est un peu plus complexe et malheureusement, nous ne la traiterons pas en profondeur. Néanmoins, l'idée de base est la suivante :

1. Initialiser un arbre de Huffman qui contient tous les symboles d'un alphabet Σ en assignant une probabilité par défaut à chaque symbole
2. Tant qu'il y a des données à encoder :
   (a) Lire le prochain symbole $s \in \Sigma$ qui doit être encodé
   (b) Encoder le symbole s
   (c) Mettre à jour l'arbre de Huffman en tenant compte de la nouvelle apparition du symbole s

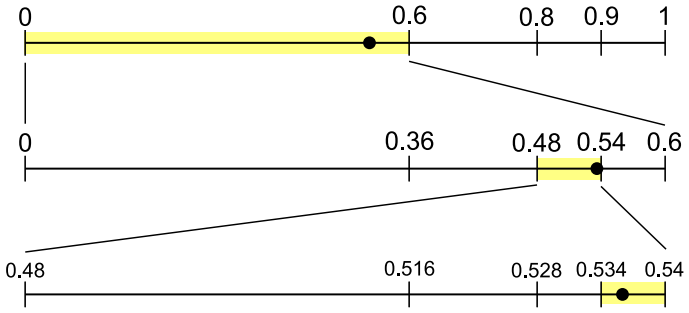
Une version naïve du codage de Huffman adaptatif pourrait rebâtir l'arbre de Huffman à chaque fois que l'arbre est mis à jour. Heureusement, il existe des techniques beaucoup plus efficaces pour mettre à jour l'arbre de Huffman et les mots de code qui sont produits.

Finalement, ce qu'il faut se rappeler, c'est qu'à chaque fois que l'arbre de Huffman est mis à jour, il se peut que les mots de code associés aux symboles soient modifiés.

TABLE 4.4 – Fréquences d'apparition des symboles de l'alphabet Σ

Symbole	A	B	C	EOF
Fréquence	6/10	2/10	1/10	1/10

FIGURE 4.3 – Codage arithmétique pour le message *A-C-EOF*



C'est pour cette raison que cette technique est qualifiée d'adaptative. Le codage s'adapte aux données au fur et à mesure qu'elles sont lues.

4.3 Codage arithmétique

Le codage de Huffman n'est optimal que si on utilise un nombre entier de bits pour représenter chacun des symboles d'un alphabet Σ. Malheureusement, cela a un impact important sur l'efficacité de cet encodage. Prenons, par exemple, un alphabet $\Sigma = \{A, B, C\}$ où la fréquence d'apparition du symbole A est de 1/3. L'entropie du symbole A est donc 1.6 bits. Cependant, étant donné que nous travaillons avec un nombre entier de bits, ne nous pouvons pas donner un mot de code de longueur 1.6 bits à A. Nous devons opter pour 1 bit ou 2 bits. On s'éloigne clairement de l'entropie calculée.

Le codage arithmétique [19, 14] n'a pas ce problème. Contrairement aux techniques présentées précédemment, le codage arithmétique n'associe pas des mots de code aux symboles d'un alphabet Σ. La technique remplace plutôt l'ensemble des symboles lus, les données, par un seul et unique nombre réel qui se situe entre 0 et 1.

Considérons l'alphabet $\Sigma = \{A, B, C, EOF\}$ [2] présenté au tableau 4.4. Maintenant, encodons le message *A-C-EOF* avec le codage arithmétique. Le codage arithmétique fonctionne en divisant un intervalle donné selon les fréquences d'apparition des symboles d'un alphabet Σ. Initialement, cet intervalle est $[0, 1[$. L'algorithme est le suivant :

1. Initialiser l'intervalle de départ à $[0, 1[$
2. Tant qu'il y a des symboles à encoder :
   (a) Diviser l'intervalle courant selon les fréquences d'apparition des symboles de l'alphabet Σ
   (b) Lire le prochain symbole $s \in \Sigma$ qui doit être encodé
   (c) L'intervalle courant devient le sous-intervalle représentant s (s est maintenant considéré encodé)
3. Lorsque tous les symboles sont encodés, transmettre au décodeur un nombre réel qui se situe à l'intérieur du dernier intervalle courant

La figure 4.3 illustre les différentes étapes du codage arithmétique pour encoder le message *A-C-EOF*. Après que le dernier symbole soit encodé, l'intervalle courant est $[0.534, 0.54[$. Nous devons donc transmettre au décodeur un nombre réel qui se situe à l'intérieur de cet intervalle, comme 0.538 par exemple. En connaissant ce nombre, le décodeur est en mesure de refaire toutes les étapes qui ont été réalisées par l'encodeur afin de reconstruire le message original. En effet, à la première itération, on se rend compte que 0.538 se situe dans l'intervalle $[0, 0.6[$ et donc nous avons un A. On procède de la même façon jusqu'à temps que le symbole de fin de transmission, EOF, soit lu. Évidemment, le symbole EOF nous permet de savoir si le décodage est terminé ou pas. Une méthode alternative à l'utilisation du symbole EOF serait de transmettre au décodeur la taille du message original avant de commencer la transmission.

Malheureusement, le codage arithmétique tel que nous venons de le voir est très difficile à implanter en pratique. L'usage de nombres de précision arbitraire est problématique. Par contre, il existe plusieurs implantations du codage arithmétique qui fonctionnent avec des nombres de précision finie.

L'idée est de remplacer l'intervalle continue $[0, 1[$ par l'intervalle discret $\{0, 255\}$ [3] que nous notons par $\{00000000_2, 11111111_2\}$ en binaire. Cet intervalle représente la

2. Le symbole *EOF* (*end of file*, fin de ficher) est un symbole artificiel fréquemment utilisé pour signaler la fin d'une transmission. Bien qu'il existe d'autres moyens de signaler la fin d'une transmission avec le codage arithmétique, l'utilisation du symbole *EOF* est la méthode la plus simple et la plus utilisée en pratique.

3. Nous avons choisi de travailler avec des nombres codés sur 8 bits.

TABLE 4.5 – Première division de l'intervalle pour l'alphabet Σ

Symbole	Fréquence	Intervalle
A	1/3	$\{00000000_2, 01010100_2\}$
B	1/3	$\{01010101_2, 10101010_2\}$
C	1/3	$\{10101011_2, 11111111_2\}$

TABLE 4.6 – Procédure de renormalisation pour l'alphabet Σ

Symbole	Fréquence	Intervalle	Renormalisation
A	1/3	$\{\mathbf{0}00000000_2, \mathbf{0}1010100_2\}$	$\{0000000\mathbf{0}_2, 1010100\mathbf{1}_2\}$
B	1/3	$\{01010101_2, 10101010_2\}$	$\{01010101_2, 10101010_2\}$
C	1/3	$\{\mathbf{1}0101011_2, \mathbf{1}1111111_2\}$	$\{0101011\mathbf{0}_2, 1111111\mathbf{1}_2\}$

suite des nombres entiers de 0 à 255. Considérons l'alphabet $\Sigma = \{A, B, C\}$, où chacun des symboles a une fréquence d'apparition de 1/3. Le tableau 4.5 illustre l'intervalle lorsqu'il est divisé pour une première fois.

Cependant, il est évident que tôt ou tard, nous allons nous retrouver avec un intervalle trop petit pour être divisé à nouveau, car nous semblons utiliser un intervalle de taille fixe. Heureusement, nous n'avons pas tout dit. En réalité, l'intervalle discret $\{00000000_2, 11111111_2\}$ correspond à l'intervalle réel $[0.00000000\ldots, 0.11111111\ldots[$ dont la taille est arbitraire (on peut généraliser en disant que l'intervalle discret $\{b_1 \ldots b_8, b_1 \ldots b_8\}$ correspondant à l'intervalle réel $[b_1 \ldots b_8 0 \ldots, b_1 \ldots b_8 1 \ldots [$). Concrètement, nous simulons la manipulation de nombres réels en ne considérant que les k bits les plus significatifs, à l'exclusion des bits les plus significatifs déjà déterminés (à cause du rétrécissement de l'intervalle).

Afin de simuler un tel intervalle, nous effectuons ce que nous appelons la renormalisation. La renormalisation se produit lorsque les valeurs à l'intérieur de l'intervalle partagent toutes $n > 0$ bits identiques à gauche [4]. On peut alors transmettre les n bits identiques au décodeur avant de les éliminer. Pour terminer la renormalisation, il suffit d'insérer n 0 à droite de la borne inférieure et n 1 à droite de la borne supérieure. La figure 4.6 illustre la procédure de renormalisation pour l'alphabet $\Sigma = \{A, B, C\}$.

4. La procédure de renormalisation est un peu différente lorsque nous avons un intervalle discret de taille plus petite que 1/4 qui est centré autour de 1/2, car il devient impossible d'observer des bits identiques à gauche (ex : $\{01111000_2, 10000101_2\}$).

TABLE 4.7 – Exemples de mots de code Elias gamma

Nombre entier	Mot de code Elias gamma
1	1
2	010
3	011
4	00100
5	00101

Encore une fois, le décodeur est en mesure de refaire toutes les étapes qui ont été réalisées par l'encodeur afin de reconstruire le message original. Évidemment, la mécanique est un peu différente, car le décodeur ne se fait pas communiquer un nombre d'un seul coup. Il va plutôt fonctionner de la même façon que l'encodeur, c'est-à-dire en faisant la renormalisation. La seule différence est que cette fois-ci, le décodeur va lire dans le fichier encodé les bits qui ont été éliminés pour renormaliser l'intervalle.

4.4 Code universel

Un code universel est un code préfixe servant à représenter des entiers positifs non-bornés. Un code universel a aussi la propriété suivante. Peu importe la distribution de probabilité sur les entiers, si $p(i) \geq p(i+1)\ \forall i$, alors les longueurs des mots de code assignés aux entiers ne diffèrent que d'un facteur constant des longueurs des mots de code qui seraient assignés par un code optimal.

4.4.1 Elias gamma

Un code Elias gamma est un code universel fréquemment utilisé pour encoder des nombres entiers positifs (excluant zéro) dont la borne supérieure est inconnue. Le mot de code pour un nombre entier donné est produit de la façon suivante :

1. Convertir le nombre entier en un nombre binaire b sans zéros superflus à gauche
2. Ajouter n 0 à gauche de b, tel que n est égal au nombre de bits utilisés pour encoder b moins 1

TABLE 4.8 – Exemples de mots de code de Levenshtein

Nombre entier	Mot de code de Levenshtein
0	0
1	10
2	110 0
3	110 1
4	1110 0 00
5	1110 0 01
6	1110 0 10
7	1110 0 11
8	1110 1 000
9	1110 1 001
10	1110 1 010

Le tableau 4.7 illustre quelques exemples. Pour obtenir le nombre entier correspondant à un mot de code, il suffit de :

1. Lire n 0 jusqu'à ce qu'un 1 soit lu
2. Convertir en nombre entier le nombre binaire composé du 1 qui vient d'être lu et des n prochains bits

4.4.2 Levenshtein

Un code de Levenshtein est un code universel utilisé pour encoder des nombres entiers non-négatifs dont la borne supérieure est inconnue. Le mot de code pour un nombre entier n est produit par la fonction L :

$$\underset{L : \mathbb{N} \to \{0,1\}^*}{L(n)} = \begin{cases} 0 & \text{si } n = 0 \\ 1L(k)N(n,k) & \text{sinon, où } k = \lfloor log_2 n \rfloor \end{cases}$$

$$\underset{N : \mathbb{N} \times \mathbb{N} \to \{0,1\}^*}{N(n,k)} = \begin{cases} \epsilon & \text{si } k = 0 \\ N(\lfloor n/2 \rfloor, k-1)b & \text{sinon, où } b = n \bmod 2 \end{cases}$$

Le tableau 4.8 illustre quelques exemples. Pour obtenir le nombre entier n correspondant à un mot de code, il suffit de :

1. Lire c 1 jusqu'à ce qu'un 0 soit lu (c est égal au nombre de bits 1 lus)
2. Si $c = 0$, alors le nombre entier est $n = 0$
3. Sinon, initialiser la variable $n = 1$ et répéter $c - 1$ fois :
   3.1 Lire n nouveaux bits, ajouter un 1 à gauche des bits lus et assigner le résultat à n

Chapitre 5

Algorithmes de compression

5.1 Run-length encoding (RLE)

Le RLE (*Run-length encoding*, codage par plages) est probablement l'algorithme de compression de données le plus simple à ce jour. Il ne fait que remplacer plusieurs apparitions d'un même symbole par un exemplaire du symbole et le nombre de fois qu'il apparaît consécutivement. Par exemple, si nous avons une série de A dans un document : $AAAAA$, le RLE remplace la série de A par un message du type $A5$. Évidemment, il faut un moyen pour différencier le message $A5$ de la séquence $A5$ dans les données originales. Une séquence d'échappement, c'est-à-dire un nouveau symbole que l'on introduit dans notre alphabet, peut être utilisée à cette fin. Il suffit alors de transmettre la séquence d'échappement avant de transmettre le nombre de fois qu'un symbole apparaît, le cas échéant.

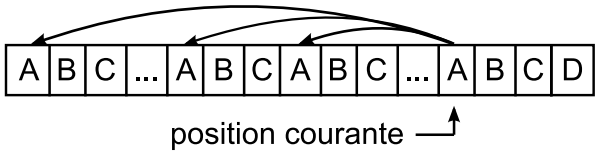
Le RLE est très efficace avec les fichiers dont les mêmes données se répètent souvent, comme une image en noir et blanc.

5.2 Algorithmes à base de dictionnaires

5.2.1 LZ77

LZ77 est un algorithme de compression de données sans perte introduit en 1977 par Jacob Ziv et Abraham Lempel dans l'article 'A Universal Algorithm for Sequential

FIGURE 5.1 – Exemple LZ77



Data Compression' [21]. L'idée derrière cet algorithme est la suivante : le compresseur envoie au décompresseur une série de messages qui vont permettre à ce dernier de reconstruire le fichier original. Les messages sont encodés et transmis au décompresseur sous la forme d'un fichier compressé. En voyant les données compressées comme étant une suite de messages entre le compresseur et le décompresseur, on retrouve deux sortes de messages : les littéraux $[c]$ et les copies $\langle l, d \rangle$. Le message $[c]$ indique au décompresseur le prochain octet du fichier original. Le message $\langle l, d \rangle$ indique au décompresseur que les l prochains octets sont une copie des l octets que l'on retrouve d octets plus tôt. La figure 5.1 illustre un exemple de fichier compressé avec LZ77. À la position courante, nous voyons qu'il existe 3 copies des 3 prochains octets. Nous pouvons donc transmettre une copie au décompresseur. Ensuite, nous voyons qu'il n'existe pas de copie pour le prochain octet. Nous transmettons donc un littéral [D] au décompresseur.

Deflate

Deflate [5] est l'implantation la plus courante de LZ77. Les copies sont transmises au décompresseur en deux étapes : on transmet premièrement leur longueur et ensuite leur distance. On utilise aussi deux codes de Huffman distincts pour la transmission des messages au décompresseur. Un premier code de Huffman est utilisé pour encoder à la fois les littéraux et les longueurs des copies. Un deuxième code de Huffman est utilisé pour encoder les distances des copies. En utilisant un code conjoint pour encoder les littéraux et les longueurs des copies, le décodeur sait automatiquement de quel type de message il s'agit (littéral ou copie) lors de la décompression. Pour compresser les données, Deflate fonctionne comme suit. Il essaie premièrement de trouver la plus longue copie possible (la longueur minimale d'une copie est de 3 octets et sa longueur maximale est de 258 octets). S'il n'existe pas de plus longue copie, un littéral est transmis au décompresseur. Lorsqu'il y a plusieurs plus longues copies, Deflate favorise toujours celle qui est la plus proche de la position courante. Évidemmment, cette description de Deflate est plutôt simplifiée. Entre autres, nous ne mentionnons pas comment les codes de Huffman sont construits. Nous ne mentionnons pas aussi comment la recherche de la plus longue copie s'effectue et la distance maximale entre une plus longue copie et la

TABLE 5.1 – LZ78 sur la chaîne d'octets *ABBCBCABA*

Itération	Index	Dictionnaire	Message
1	1	A	(0,A)
2	2	B	(0,B)
3	3	BC	(2,C)
4	4	BCA	(3,A)
5	5	BA	(2,A)

position courante (approx. 32KB).

5.2.2 LZ78

LZ78 est un algorithme de compression de données sans perte introduit en 1978 par Jacob Ziv et Abraham Lempel dans l'article 'Compression of Individual Sequences via Variable-Rate Coding' [22]. Contrairement à LZ77, LZ78 ne fait pas référence aux données vues précédemment dans le fichier original pour compressser les données (les copies). LZ78 utilise plutôt un dictionnaire contenant les chaînes d'octets déjà rencontrées. Au départ, le dictionnaire contient une entrée : l'entrée 0, qui représente la chaîne vide ; et au fur et à mesure que les données sont traitées, le dictionnaire est mis à jour. Évidemment, la taille du dictionnaire est limitée par la mémoire disponible.

L'algorithme est relativement simple. Le compresseur lit les prochains octets du fichier à compresser jusqu'à ce que la chaîne d'octets lue ne soit pas dans le dictionnaire. Il transmet alors deux choses au décompresseur. Premièrement, il transmet l'index, dans le dictionnaire, de la chaîne composée des $n-1$ octets qui ont été lus (ou l'index 0 si $n = 1$). Ensuite, il transmet l'octet n. Pour terminer, le compresseur met à jour le dictionnaire en y ajoutant la chaîne composée des n octets qui ont été lus. Le figure 5.1 illustre l'application dans l'algorithme LZ78 sur la chaîne d'octets *ABBCBCABA*.

5.3 Burrows-Wheeler transform (BWT)

L'algorithme BWT (*Burrows-Wheeler transform*, transformation de Burrows-Wheeler) a été introduit en 1994 par M. Burrows et D.J. Wheeler dans l'article 'A Block-sorting Lossless Data Compression Algorithm' [3]. La motivation derrière le BWT

TABLE 5.2 – Exemple de transformation BWT

Ligne	Rotation non-triée	Rotation triée	L
0	abraca	aabrac	c
1	aabrac	abraca	a
2	caabra	acaabr	r
3	acaabr	bracaa	a
4	racaab	caabra	a
5	bracaa	racaab	b

est d'avoir un algorithme de compression presqu'aussi efficace que ceux basés sur les modèles statistiques (voir PPM à la section 5.4) tout en restant aussi rapide que les algorithmes à base de dictionnaires comme LZ77.

Contrairement aux autres algorithmes, le BWT fonctionne par bloc et non séquentiellement. L'idée générale est d'appliquer une transformation qui est réversible sur un bloc b afin de produire un bloc b' qui est plus facile à compresser. L'objectif de cette transformation est de regrouper les apparitions d'un même symbole. En d'autres mots, la probabilité de trouver un symbole $s \in \Sigma$ à proximité d'une autre apparition de s est augmentée. Le bloc b' résultant est habituellement transformé avec l'algorithme Move-to-front (voir MTF à la section 5.3.1) qui est très efficace quand plusieurs apparitions d'un même symbole se succèdent ou sont peu éloignées. Ensuite, le codage de Huffman ou le codage arithmétique peut être utilisé pour encoder ce qui est produit par le MTF.

Afin de respecter la terminologie utilisée par les auteurs du BWT, considérons les symboles d'un alphabet Σ comme étant des caractères et un bloc comme étant une chaîne de caractères S de longueur N. L'algorithme BWT est le suivant :

1. Les N rotations de la chaîne S sont placées dans une matrice M de taille $N \times N$ (il faut voir la chaîne S comme étant une chaîne circulaire)
2. La matrice M est triée en ordre lexicographique en échangeant les rangées
3. Le dernier caractère de chacune des N rotations dans M est conservé pour former une chaîne L (le *i-ième* caractère de L est le dernier caractère de la *i-ième* rotation de S dans M)
4. L'index I de la chaîne originale S dans M est aussi conservé

Le tableau 5.2 présente un exemple de transformation BWT avec $S = abraca$ et $N = 6$. La première colonne présente les lignes de la matrice des rotations, la deuxième

colonne présente la matrice des rotations non-triées, la troisième colonne présente la matrice des rotations triées et la dernière colonne présente la chaîne extraite de la dernière colonne de la matrice des rotations triées (L). Les résultats de la transformation sont la chaîne $L = caraab$ et l'index $I = 1$. La chaîne L est en fait le bloc transformé qui est maintenant plus facile à compresser. Comme nous pouvons le voir, nous avons maintenant deux apparitions du symbole a côte à côte. Finalement, avec la chaîne L et l'index I, le décompresseur est en mesure de reconstruire la matrice M et d'identifier la chaîne originale.

Maintenant, regardons pourquoi nous nous attendons à ce que les symboles identiques aient tendance à se regrouper. Considérons un texte écrit en langue anglaise contenant plusieurs fois le mot *the*. Il y a forcément plusieurs rotations commençant par *he*. De plus, les chances que ces rotations se terminent par t sont très élevées. Nous avons donc forcément plusieurs apparitions de t qui sont regroupées. L'algorithme MTF est en mesure de tirer avantage de cette propriété, ce qui fait du BWT un des algorithmes les plus performants.

5.3.1 Move-to-front (MTF)

L'algorithme MTF (*Move-to-front*, déplacer vers l'avant) effectue une transformation qui est réversible sur un bloc b afin de produire un bloc b' qui est plus facile à compresser. De plus, la transformation effectuée par le MTF est très efficace lorsqu'elle est appliquée à un bloc préalablement transformé par le BWT.

L'algorithme MTF transforme une chaîne L de longueur N en un vecteur R de longueur N également. La transformation se fait comme suit :

1. Les caractères d'un alphabet Σ sont placés dans une liste Y qui ne contient qu'une et une seule apparition de chacun des caractères
2. Pour chaque i, tel que $i = 0, \ldots, N-1$:
   (a) $R[i]$ est égal au nombre de caractères précédant le caractère $L[i]$ dans la liste Y
   (b) $L[i]$ est placé à la tête de la liste Y (les autres caractères sont décalés)

Considérons la chaîne $L = caraab$ utilisée précédemment et la liste $Y = [a, b, c, r]$. Le résultat de la transformation MTF, présentée à la figure 5.3, est le vecteur $R = (2, 1, 3, 1, 0, 3)$. Le vecteur R est en fait composé des index des N caractères de L dans

TABLE 5.3 – Exemple de transformation MTF

i	Y	R
0	$[a, b, c, r]$	(2)
1	$[c, a, b, r]$	(2,1)
2	$[a, c, b, r]$	(2,1,3)
3	$[r, a, c, b]$	(2,1,3,1)
4	$[a, r, c, b]$	(2,1,3,1,0)
5	$[a, r, c, b]$	(2,1,3,1,0,3)

Y. Ainsi, les caractères récemment utilisés ont toujours un index assez petit. Cette propriété est très intéressante, car les fréquences d'apparition des petits index risquent d'être plus grandes.

5.4 Prediction by Partial Matching (PPM)

Le PPM [4] (*Prediction by Partial Matching*, prédiction par reconnaissance partielle) est un algorithme de compression de données axé sur la modélisation de contextes et sur les prédictions. Le PPM est aussi un algorithme adaptatif qui est presque tout le temps utilisé avec le codage arithmétique. L'idée derrière le PPM est de prédire le prochain symbole à encoder en se basant sur les n symboles antérieurs. Les n symboles antérieurs forment ce que nous appelons un contexte. Un contexte a aussi un ordre. L'ordre d'un contexte est le nombre de symboles utilisés pour faire la prédiction du prochain symbole, c'est-à-dire les n symboles antérieurs. On utilise souvent la notation $PPM(n)$ pour parler d'un PPM au contexte d'ordre n et la notion PPM^* pour parler d'un PPM ayant un contexte non-borné.

Les statistiques concernant la prédiction dans un certain contexte viennent des observations précédentes dans le même contexte. Lorsqu'aucune prédiction ne peut être faite avec un contexte d'ordre n, on essaie avec un contexte d'ordre $n - 1$. Lorsqu'on arrive au contexte 0 (le contexte par défaut), c'est-à-dire lorsqu'aucun contexte non-trivial nous permet de prédire le prochain symbole s, nous faisons une prédiction qui ne dépend d'aucun contexte. Cela peut se produire si, par exemple, tous les symboles ont une probabilité égale.

La plus grande difficulté avec le PPM est de trouver une façon efficace de représenter un symbole qui apparaît pour la première fois. Un symbole peut apparaître pour la

première fois dans deux situations différentes. La première situation est lorsqu'aucun contexte ne permet de prédire le symbole. On doit alors utiliser le contexte par défaut (0). La deuxième situation est lorsque le contexte courant n'a jamais été rencontré juste avant le symbole. En d'autres mots, le symbole apparaît pour la première fois dans le contexte courant. Dans les deux cas, une des approches est d'utiliser un symbole spécial qui sert de séquence d'échappement pour introduire de nouveaux symboles. Cependant, décider de la probabilité que doit avoir ce pseudo-symbole est beaucoup plus compliqué. On peut, par exemple, initialiser à 1 le compteur d'apparitions de ce symbole. On peut aussi incrémenter le compteur d'apparitions du pseudo-symbole à chaque fois qu'un symbole apparaît pour la première fois.

Chapitre 6

Recyclage de bits

6.1 Introduction

Plusieurs techniques de compression de données ont la particularité de pouvoir produire plusieurs fichiers compressés différents à partir d'un même document original. C'est le cas, entre autres, de LZ77 et ses dérivés, car il est souvent possible de décrire, ou d'encoder, les copies de plusieurs façons différentes lorsqu'elles se retrouvent à plusieurs endroits dans la fenêtre [1] de compression. Évidemment, cette multiplicité des copies cause de la redondance. Notre technique de recyclage de bits [6, 8, 7] utilise cette redondance pour réduire la taille du fichier compressé, atténuant ainsi partiellement les effets néfastes de la redondance. La redondance n'est pas éliminée mais plutôt utilisée afin de transmettre gratuitement de l'information au décompresseur. Plus précisément, nous nous servons de la redondance comme canal auxiliaire et nous utilisons ce canal auxiliaire pour transmettre au décompresseur des parties du fichier compressé lui-même ; ce qui a pour conséquence de réduire sa taille et d'améliorer son taux de compression. Nous montrons comment notre technique s'applique au codage de Huffman qui est utilisé par Deflate, une implantation de LZ77.

1. On appelle fenêtre de compression la partie du fichier qui est examinée lors de la recherche d'une copie.

6.2 Définitions

La fonction produisant un code de Huffman est définie comme suit :

$$\mathit{Huffman} : (S \to \mathbb{R}^{>0}) \to (S \to \{0,1\}^*)$$

La fonction prend en entrée un ensemble de symboles et leurs fréquences d'apparition et retourne en sortie une fonction d'encodage optimale pour cet ensemble de symboles.

La fonction *flat* est définie comme suit :

$$\mathit{flat}(\{s_1, \ldots, s_k\}) = \mathit{Huffman}(\{(s_1, 1), \ldots, (s_k, 1)\})$$

Cette fonction donne à tous les symboles de l'ensemble $\{s_1, \ldots, s_k\}$ un mot de code de longueur semblable ; i.e. une différence d'au plus un bit entre les longueurs de deux mots de code. Évidemment, cette fonction pourrait être définie plus efficacement.

La fonction de recyclage R est définie comme suit :

$$R : (2^S \times (S \to \{0,1\}^*)) \to (S \to \{0,1\}^*)$$

La fonction prend en entrée un sous-ensemble d'un ensemble de symboles et une fonction d'encodage pour l'ensemble de symboles et retourne en sortie une fonction d'encodage, ou plus précisément de recyclage, pour cet ensemble de symboles. La fonction retournée est un code préfixe et sert à produire des mots de code que nous qualifions de recyclés.

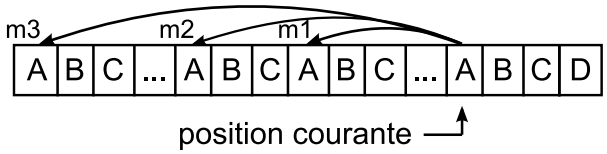
6.3 Possibilités d'encodages multiples

6.3.1 Encodages multiples et conséquences

Une technique de compression de données perd de son efficacité quand les mêmes données peuvent être encodées de plusieurs façons différentes. En effet, lorsqu'il existe plus qu'une façon d'encoder les mêmes données, les encodages ont tendance à être plus longs et la taille des fichiers compressés qui sont produits tend à augmenter.

Comme nous l'avons dit en introduction, LZ77 introduit de la redondance dans la façon d'encoder les copies. Cependant, comme nous le verrons, les implantations cherchent normalement à encoder les copies efficacement en choisissant systématiquement la plus longue copie ayant la plus courte distance. Néanmoins, la redondance due

FIGURE 6.1 – Exemple de messages équivalents



à la multiplicité demeure et la multiplicité des encodages tend à augmenter inutilement la taille des fichiers compressés.

6.3.2 Messages équivalents

Dans notre introduction à LZ77, nous avons introduit la notion de message. En résumé, il existe deux sortes de messages : les littéraux $[c]$ et les copies $\langle l, d \rangle$. Le message $[c]$ indique au décompresseur le prochain octet du fichier original. Le message $\langle l, d \rangle$ indique au décompresseur que les l prochains octets sont une copie des l octets que l'on retrouve d octets plus tôt. Les messages d'un ensemble $\{m_1, m_2, \ldots, m_n\}$ sont dits équivalents s'ils transportent tous la même information. L'ensemble des *plus longues copies* est justement un ensemble de messages équivalents. Chaque message de l'ensemble est de la forme $\langle l, \cdot \rangle$ et représente la même information. La seule et unique chose qui différencie un tel message d'un autre est sa position (distance) dans la fenêtre de compression. La figure 6.1 illustre un exemple où les messages de l'ensemble $\{m_1, m_2, m_3\}$ sont équivalents. En effet, chacun des trois messages peut être utilisé pour représenter les 3 prochains octets à la position courante.

Notez que pour la suite du texte, nous parlerons indifféremment de l'ensemble des messages équivalents, de l'ensemble des plus longues copies ou de l'ensemble des distances aux plus longues copies. En effet, l'ensemble des plus longues copies est constitué de copies qui représentent toutes la même suite d'octets et qui sont toutes de la même longueur. Seules leurs distances nous permettent de les différencier, d'où la notion d'ensemble de distances.

6.3.3 Messages non-équivalents

Deux messages sont non-équivalents s'ils ne transportent pas la même information au décompresseur. C'est le cas de deux littéraux différents, d'un littéral et d'une copie,

de deux copies qui ne sont pas de la même longueur ou de deux copies qui sont de la même longueur mais qui ne représentent pas la même suite d'octets.

6.3.4 Lien avec les encodages multiples de fichiers

Lorsque l'information à envoyer peut être envoyée à l'aide d'un message quelconque d'un ensemble de messages équivalents, le compresseur peut se permettre de choisir n'importe quel message dans cet ensemble. Il est donc possible d'encoder les mêmes données de plusieurs façons différentes. La figure 6.1 illustre un exemple où il est possible d'encoder les caractères *ABC*, situés à la position courante, de trois façons différentes. Évidemment, peu importe lequel de ces 3 messages est choisi par le compresseur, le décompresseur est en mesure de retrouver les caractères *ABC*.

Maintenant, ce qu'il faut constater, c'est que l'utilisation de l'un des messages de l'ensemble des messages équivalents plutôt que les autres produit un fichier compressé différent. En effet, comme chaque message a un encodage différent, le fichier compressé résultant est forcément différent lui aussi. L'existence des messages équivalents est donc une cause de l'existence des différents fichiers compressés que l'on peut avoir pour un même document original.

6.4 Recyclage

6.4.1 Idée générale de message caché

Lors de la compression, si à une position donnée nous avons un ensemble de messages équivalents $\{m_1, m_2, \ldots, m_n\}$, nous pouvons appliquer la technique de recyclage de bits. Le recyclage s'effectue premièrement par le choix d'un message parmi ceux de l'ensemble des messages équivalents. En effet, si nous avons un ensemble de messages équivalents $\{m_1, m_2, \ldots, m_n\}$ et que nous choisissons intentionnellement l'un d'entre eux lors de la compression, alors, lors de la décompression, le décompresseur est en mesure de constater notre choix, car il a lui aussi accès à l'ensemble des messages équivalents qui était à la disposition du compresseur. Maintenant, si nous assignons un message caché (une séquence de bits) à chacun des messages de l'ensemble $\{m_1, m_2, \ldots, m_n\}$, il devient alors possible de transmettre gratuitement de l'information au décompresseur. Les messages cachés sont transmis implicitement au décompresseur et on dit qu'ils le sont à travers de ce qui peut être vu comme un canal auxiliaire. Par exemple, si

nous avons un ensemble 4 messages équivalents $\{m_1, m_2, m_3, m_4\}$, nous pouvons leur assigner chacun une séquence de 2 bits. Si le compresseur choisit m_1, alors 00 est transmis gratuitement au décompresseur, si le compresseur choisit m_2, alors 01 est transmis gratuitement au décompresseur, et ainsi de suite. On a donc un moyen, par le simple fait de choisir un message parmi 4 messages équivalents, de transmettre 2 bits gratuitement.

6.4.2 Idée de transmission de parties de fichier compressé dans les messages cachés

L'utilisation d'un canal auxiliaire pour transmettre de l'information de façon implicite a été présentée dans le passé. Le canal auxiliaire a été utilisé à plusieurs fins. Le canal auxiliaire a été utilisé comme médium pour cacher un fichier secret à l'intérieur d'un fichier compressé (de taille relativement importante) [2]. Le canal auxiliaire a aussi été utilisé pour l'authentification de fichiers compressés [1] et pour l'inclusion de codes de correction d'erreurs [12, 13, 11, 20] dans les fichiers compressés pour les rendre moins vulnérables aux erreurs de transmission et de stockage. Il est à noter que toutes ces techniques utilisent le canal auxiliaire pour cacher de l'information sans toutefois changer le format du fichier compressé. Le décompresseur n'a pas besoin d'être modifié pour pouvoir récupérer le document original. En fait, il n'a nullement besoin de voir et d'interpréter l'information cachée dans le fichier compressé. Le décompresseur n'a même pas besoin d'être conscient qu'il y a de l'information cachée. Un décompresseur spécialisé est nécessaire afin de récupérer l'information cachée (et de l'exploiter, le cas échéant). La stéganographie (le terme plus exact pour la dissimulation d'information) peut aussi être accomplie par d'autres moyens que par la sélection d'encodages dans la production de fichiers compressés et peut être utilisée dans plusieurs autres applications. Nous invitons le lecteur à consulter la revue de la stéganographie de Petitcolas et autres [16].

Nous proposons ici d'utiliser le canal auxiliaire pour réduire la taille du fichier compressé. Nous enlevons une certaine quantité de bits du fichier compressé que nous transmettons gratuitement au décompresseur grâce au canal auxiliaire. Nous appelons notre technique le recyclage de bits. Évidemment, en procédant ainsi, nous modifions le format du fichier compressé et seul un décompresseur adapté peut décompresser le fichier compressé adéquatement. Nous avons effectué différentes expériences sur **gzip**, un outil de compression très populaire. Nous montrons que le recyclage de bits permet de réduire la taille du fichier compressé de façon significative.

6.4.3 Recyclage plat

Le recyclage plat est la première variante du recyclage que nous avons étudiée. Le fonctionnement du recyclage plat est fort simple : chaque message équivalent $m \in \{m_1, m_2, \ldots, m_n\}$ se voit attribuer un mot de code recyclé produit avec le codage de Huffman en utilisant des statistiques uniformes, c'est-à-dire avec la fonction *flat*. Par exemple, si nous avons le choix entre 4 messages équivalents, nous leurs assignons les mots de codes 00, 01, 10 et 11. La fonction de recyclage R^{flat} est utilisée pour assigner un mot de code recyclé à chaque message de l'ensemble $\{m_1, m_2, \ldots, m_n\}$. La fonction C est utilisée pour encoder les messages de l'ensemble $\{m_1, m_2, \ldots, m_n\}$ avant de les transmettre au décompresseur. Ici, la fonction C n'a aucun impact sur la fonction de recyclage R^{flat} :

$$R^{flat}(\{m_1, \ldots, m_n\}, C) = flat(\{m_1, \ldots, m_n\})$$

Évidemment, le mot de code recyclé qui est assigné à un message de l'ensemble $\{m_1, m_2, \ldots, m_n\}$ par la fonction de recyclage R^{flat} est en fait le message caché qui sera transmis au décompresseur par le canal auxiliaire, le cas échéant.

6.4.4 Recyclage proportionnel

Contrairement au recyclage plat, le recyclage de bits proportionnel prend en compte le coût d'encodage des messages équivalents $\{m_1, m_2, \ldots, m_n\}$ par la fonction C dans la construction du code de Huffman qui est utilisé pour le recyclage. Plus précisément, la longueur du mot de code recyclé assigné à un message équivalent $m \in \{m_1, m_2, \ldots, m_n\}$ tend à augmenter linéairement (suivant une fonction $x \mapsto x - a$, pour un certain a) avec le coût d'encodage du message qui est encodé par la fonction C. La fonction de recyclage $R^{\propto}$ est définie comme suit :

$$R^{\propto}(\{m_1, \ldots, m_n\}, C) = \mathit{Huffman}(\{(m_1, 2^{-|C(m_1)|}), \ldots, (m_n, 2^{-|C(m_n)|})\})$$

Ainsi, plus long est le mot de code utilisé pour encoder un message, plus long est le mot de code recyclé associé à ce même message. En d'autres mots, plus un message coûte cher à encoder, plus nombreux sont les bits recyclés si ce message est choisi.

Aussi, comme nous le verrons plus loin, les m_i sont choisis en fonction de leurs mots de code recyclés et les bits que l'on recycle sont de nature aléatoire. La probabilité qu'un message équivalent $m \in \{m_1, m_2, \ldots, m_n\}$ soit choisi dépend de la longueur de son mot de code recyclé. Un message m dont le mot de code recyclé est de longueur l bits a une probabilité de 2^{-l} d'être choisi. Ainsi, plus long est le mot de code recyclé associé à un message, moindres sont les chances que ce message soit choisi. Le recyclage proportionnel est donc très intéressant, car il permet de *pénaliser* les messages qui coûtent plus cher à encoder en leurs assignant des mots de code recyclés plus longs. Toutefois, lorsque ces messages sont choisis, plus de bits sont recyclés, ce qui permet de compenser pour le coût de leur encodage.

6.4.5 Capacité du canal auxiliaire

Considérons un fichier compressé comme étant une série de messages $m_1, \ldots, m_n$. Considérons aussi les messages équivalents, c'est-à-dire différents messages qui sont en fait des copies des mêmes données. L'ensemble de messages équivalents pour m_i est $\{m_{i,1}, \ldots, m_{i,k_i}\}$, incluant m_i. Un fichier compressé peut alors être vu comme une série de messages $m_{1,j_1}, \ldots, m_{n,j_n}$ tel que pour tout $1 \leq i \leq n$ nous avons $1 \leq j_i \leq k_i$. Cela produit donc $\prod_{i=1}^{n} k_i$ fichiers compressés différents mais équivalents ; et en choisissant l'un d'entre eux, il est possible de transporter environ $\sum_{i=1}^{n} log_2(k_i)$ bits d'information. Cette estimation est valide si on suppose qu'on utilise le recyclage plat.

6.5 LZ77 et recyclage de bits

6.5.1 LZ77 conventionnel

Les algorithmes de la technique LZ77 conventionnelle sont présentés à la figure 6.2. Le compresseur transmet les messages au décompresseur en deux étapes. Premièrement, le compresseur informe le décompresseur de la nature du message en lui indiquant s'il s'agit d'un littéral $[c]$ ou d'une copie d'une certaine longueur ($\langle l, \cdot \rangle$). Deuxièmement, lorsqu'il s'agit d'une copie, il informe le décompresseur de la distance d de la copie.

À chacune des étapes, on utilise une fonction d'encodage pour communiquer avec le décompresseur (les fonctions C_1 et C_2, respectivement). Ces deux fonctions sont plus précisément un code de Huffman. Elles permettent à une implantation de LZ77 d'optimiser la compression. Finalement, on représente par σ la chaîne de bits qui sert

```
1. while not end of input do
2.     ⟨l, d⟩ := find best match ;
3.     if l < l_min then
4.         emit C_1([next input byte]) ;
5.     else
6.         emit C_1(⟨l, ·⟩) ;
7.         emit C_2(d) ;
where
8. procedure emit w :
9.     σ := σ · w ;
10.    return ;
11.
```

Compresseur

```
1. while end not signaled do
2.     msg := receive C_1 ;
3.     if msg = [c] then
4.         store c ;
5.     else let ⟨l, ·⟩ = msg
6.         d := receive C_2 ;
7.         store copy of ⟨l, d⟩ ;
where
8. procedure receive C :
9.     let msg, σ' s.t. C(msg) · σ' = σ ;
10.    σ := σ' ;
11.    return msg ;
```

Décompresseur

FIGURE 6.2 – Algorithmes LZ77 conventionnel

de canal de communication (principal) entre le compresseur et le décompresseur. C'est-à-dire que le compresseur accumule le fichier compressé dans σ et que le décompresseur extrait les bits du fichier compressé de σ.

Évidemment, ces algorithmes ne sont que schématiques. Ils n'incluent pas les mécanismes d'entrée/sortie, la méthode qui recherche une copie ainsi que les messages de contrôle qui sont envoyés au décompresseur pour signaler, entres autres, le début ou la fin de la transmission. Cependant, ces notions ne sont pas essentielles dans notre étude du recyclage de bits.

Il reste maintenant à définir ce qu'est une meilleure copie (*best match*). Puisque la compression est obtenue principalement par l'usage des copies, il semble judicieux de favoriser les copies qui sont les plus longues possibles. Cependant, cette définition est insuffisante, car la plus longue copie peut se retrouver à plusieurs endroits dans la fenêtre de compression. La plupart des implantations de LZ77 favorisent alors toujours celle qui est la plus proche de la position courante (celle dont la distance est la plus courte). Cette façon de faire permet de biaiser positivement le code de Huffman utilisé pour encoder les distances (C_2). En effet, en étant utilisées plus fréquemment, les courtes distances se voient attribuer des mots de code plus courts. Évidemment, cela a un impact important sur la taille du fichier compressé. Les courtes distances sont utilisées plus fréquemment donc elles ont des mots de code plus courts, ce qui peut diminuer considérablement la taille du fichier compressé. Cependant, pour le recyclage de bits, nous rejetons cette définition d'une meilleur copie.

```
1. while not end of input do
2.     ⟨l, {d_1, ..., d_k}⟩ := find best matches ;
3.     if l < l_min then
4.         emit C_1([next input byte]) ;
5.     else
6.         emit C_1(⟨l, ·⟩) ;
7.         ND-let i ∈ {1, ..., k} ;
8.         emit C_2(d_i) ;
9.         recycle R({d_1, ..., d_k}, C_2)(i) ;
where
10. procedure emit w :
11.     if w = ε or ρ = ε then
12.         σ := σ · w ;
13.     else if w = b · w' and ρ = b · ρ'
14.             /* where b ∈ {0, 1} */ then
15.         ρ := ρ' ;
16.         emit w' ;
17.     else
18.         error ;
19.     return ;
20. procedure recycle w :
21.     ρ := w · ρ ;
22.     return ;
```

Compresseur

```
1. while end not signaled do
2.     msg := receive C_1 ;
3.     if msg = [c] then
4.         store c ;
5.     else let ⟨l, ·⟩ = msg
6.         d := receive C_2 ;
7.         store copy of ⟨l, d⟩ ;
8.         ⟨l, {d_1, ..., d_k}⟩ :=
9.             find copies of ⟨l, d⟩ ;
10.        let i ∈ {1, ..., k} s.t. d_i = d ;
11.        recycle R({d_1, ..., d_k}, C_2)(i) ;
where
12. procedure receive C :
13.     let msg, σ' s.t. C(msg) · σ' = σ ;
14.     σ := σ' ;
15.     return msg ;
16.
17.
18.
19.
20. procedure recycle w :
21.     σ := w · σ ;
22.     return ;
```

Décompresseur

FIGURE 6.3 – Algorithmes LZ77 avec recyclage de bits

6.5.2 LZ77 et recyclage de bits

La figure 6.3 présente l'algorithme du compresseur et l'algorithme du décompresseur de LZ77 avec recyclage de bits. Il y a deux choses qui différencient ces algorithmes de ceux de LZ77 conventionnel. Premièrement, on considère maintenant toutes les plus longues copies (toutes décrites par des messages équivalents) et non seulement celle qui a la plus courte distance quand vient le temps de transmettre une copie au décompresseur. Nous rejetons ainsi la définition d'une meilleure copie de LZ77 conventionnel. Deuxièmement, nous avons besoin d'un peu plus de machinerie pour pouvoir effectuer le recyclage.

Du point de vue algorithmique, le recyclage de bits se comprend mieux en regardant d'abord comment le décompresseur se comporte. À la ligne 6, le décompresseur apprend quelle plus longue copie $\langle l, d\rangle$ a été envoyée par le compresseur. À la ligne 7, il identifie les octets que la plus longue copie représente. À la ligne 8, il est en mesure de retrouver l'ensemble des plus longues copies (l'ensemble des distances $\{d_1, d_2, \ldots, d_n\}$) que le compresseur avait en sa possession avant de choisir celle qui a été envoyée. À

la ligne 10, il peut finalement constater laquelle des plus longues copies le compresseur avait choisi d'envoyer ($d = d_i$). Pour terminer, à la ligne 11, le décompresseur effectue le recyclage en tant que tel. Le décompresseur détermine la séquence de bits associée au fait d'avoir choisi d_i parmi l'ensemble $\{d_1, d_2, \ldots, d_n\}$ et ajoute cette séquence devant σ. Le décompresseur peut maintenant passer à l'itération suivante et procéder. Évidemment, le prochain message peut être décodé à partir, entre autres, de bits provenant du recyclage.

Du côté compresseur, le recyclage de bits peut sembler plus obscur. À la ligne 7, le compresseur doit choisir la plus longue copie qu'il va envoyer. Le choix de cette plus longue copie détermine les bits qui seront recyclés par le décompresseur. Plus précisément, ces bits contribueront à encoder le ou les prochains messages. Cependant, à cette étape-ci, le compresseur n'est pas en mesure de savoir quels sont les prochains messages qui seront envoyés. Il est donc forcé de faire un choix non-déterministe. La procédure **emit** vérifiera à posteriori que le compresseur a fait le *bon* choix. Une chaîne de bits auxiliaire, ρ, est utilisée par le compresseur pour garder en mémoire ses prédictions. Conceptuellement, les bits mis dans ρ sont les bits qui seront transmis gratuitement au décompresseur par le canal auxiliaire. Ces bits ne seront pas transmis explicitement dans σ. En d'autres mots, nous pouvons dire que le choix non-déterministe qui est fait à la ligne 7 cause l'exécution de l'algorithme à se dupliquer en $|\{d_1, d_2, \ldots, d_n\}|$ exécutions concurrentes. Plus loin dans l'exécution de l'algorithme, toutes ces exécutions sauf une seront déclarées inconsistentes (en utilisant '**error**'), car les bits recyclés ne correspondront pas aux bits utilisés pour encoder le ou les prochains messages. L'unicité de la bonne option est une conséquence directe de l'utilisation d'un code préfixe pour le recyclage.

6.6 Résultats expérimentaux

6.6.1 gzip

Nous avons réalisé plusieurs expériences sur le recyclage de bits. Toutes ces expériences sont des modifications apportées à **gzip** [9], un outil de compression très populaire qui utilise l'algorithme Deflate, une implantation performante de LZ77.

L'outil **gzip** comprend un compresseur et un décompresseur. Le compresseur fonctionne par bloc, c'est-à-dire que le document original est divisé en plusieurs parties qui sont compressées individuellement. Chaque bloc est traversé linéairement et des mes-

sages sont émis au décompresseur pour que ce dernier puisse reconstruire le document original. Plus précisément, chaque bloc est traversé une première fois pour déterminer les messages qui seront envoyés au décompresseur. Après cette première passe, **gzip** est en mesure de construire les codes optimaux pour l'encodage des messages. Dans un deuxième temps, **gzip** envoie les messages encodés au décompresseur. Évidemment, les messages sont transmis au décompresseur par le biais du fichier compressé.

Les messages qui sont envoyés au décompresseur proviennent de l'application de l'algorithme Deflate sur chacun des blocs. Ces messages sont des littéraux et des plus longues copies. Pour encoder les messages, **gzip** utilise deux fonctions d'encodage : C_1 et C_2. La fonction C_1 sert à encoder les littéraux et les longueurs des plus longues copies. Il faut se rappeler que les copies sont de la forme $\langle l, d \rangle$, où l est la longueur et d la distance qui sépare la copie de la position courante. La fonction C_2 sert à encoder les distances des plus longues copies.

Évidemment, pour qu'un message puisse être envoyé, il lui faut un encodage. Cependant, **gzip**, par l'application de l'algorithme Deflate, ne produit pas un encodage pour tous les messages possibles. En effet, les littéraux et les plus longues copies qui n'ont pas été rencontrés dans la première passe ne reçoivent pas d'encodage. Pour le recyclage de bits, toutes les plus longues copies peuvent être choisies et non seulement celle dont la distance est la plus courte. Certains messages pourraient donc être émis alors qu'ils ne le seraient pas avec l'algorithme Deflate conventionnel.

D'ailleurs, nous remplacerons la fonction C_2 dans nos expériences, car comme nous venons de le mentionner, **gzip** ne produit pas un encodage pour toutes les plus longues copies possibles (leurs distances). En d'autres mots, bien que la longueur de la plus longue copie ne change pas, le recyclage de bits risque de causer l'emploi d'autres distances.

Ceci étant dit, nous utiliserons trois sortes d'encodage (ou fonctions d'encodage) pour encoder les distances lors de nos différentes expériences. Premièrement, nous utiliserons la fonction d'encodage C_2^{flat}. Avec C_2^{flat}, toutes les distances possibles (de 1 à 32768) reçoivent un mot de code de longueur semblable. Ensuite, nous utiliserons C_2^{gzip}. Il s'agit en fait de l'encodage de **gzip** pour encoder les distances. C_2^{gzip} a la propriété d'assigner des mots de code plus courts aux distances qui sont plus fréquentes et des mots de code plus longs aux distances qui se font plus rares. Aussi, ce ne sont pas toutes les distances qui reçoivent un mot de code : seulement celles qui sont réellement utilisées par l'algorithme Deflate lors de son exécution sur un bloc donné reçoivent un mot de code. Finalement, nous utiliserons C_2^{full}. C_2^{full} est en fait une version modifiée de C_2^{gzip}. La seule différence est que toutes les distances qui pourraient être utilisées

TABLE 6.1 – Résultats des expériences 1 à 6

Fichier	Original	Gzip	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 6
bib	111 261	34 900	34 305	34 445	34 049	33 859	33 943	33 863
book1	768 771	312 281	301 955	308 261	303 925	301 695	303 092	301 888
book2	610 856	206 158	202 430	203 369	201 592	200 547	200 456	200 165
geo	102 400	68 414	66 542	68 020	67 286	66 347	66 343	65 889
news	377 106	144 400	142 808	142 395	141 181	140 709	140 565	140 418
obj1	21 504	10 320	11 147	10 366	10 428	10 405	10 315	10 314
obj2	246 814	81 087	84 289	79 782	80 043	79 825	79 208	79 179
paper1	53 161	18 543	18 783	18 382	18 360	18 232	18 180	18 147
paper2	82 199	29 667	29 401	29 354	29 157	29 027	28 977	28 933
paper3	46 526	18 074	18 198	17 916	17 860	17 764	17 717	17 689
paper4	13 286	5 534	5 986	5 500	5 482	5 460	5 446	5 440
paper5	11 954	4 995	5 467	4 959	4 951	4 935	4 917	4 922
paper6	38 105	13 213	13 783	13 191	13 150	13 124	13 048	13 055
pic	513 216	52 381	54 257	52 023	52 303	52 040	51 742	51 621
progc	39 611	13 261	13 852	13 104	13 082	13 176	12 974	13 097
progl	71 646	16 164	16 653	15 942	15 923	15 848	15 782	15 771
progp	49 379	11 186	11 787	11 047	11 104	11 041	10 957	10 943
trans	93 695	18 862	19 340	18 658	18 641	18 518	18 490	18 454

lors du recyclage reçoivent un encodage. Pour pouvoir faire cela, l'algorithme recyclant prétend que toutes les distances des plus longues copies seront utilisées (au lieu des distances des plus proches plus longues copies seulement).

Pour terminer, nous disons qu'une distance est *éligible* pour le recyclage lorsqu'elle a un encodage et lorsqu'elle fait partie de l'ensemble des distances des plus longues copies. Une distance qui fait partie de l'ensemble des distances des plus longues copies mais qui n'a pas d'encodage n'est pas éligible pour le recyclage, car elle ne peut être transmise au décompresseur. Cette situation peut se produire avec l'encodage C_2^{gzip}.

6.6.2 Expérience 1

Idée

Pour notre première expérience, nous avons utilisé l'encodage plat (C_2^{flat}) pour encoder les distances. Avec l'encodage plat, toutes les distances sont éligibles pour le

recyclage. Pour le recyclage de bits, nous avons utilisé le recyclage plat (R^{flat}), c'est-à-dire que peu importe la distance qui est choisie, le nombre de bits recyclés est semblable. Nous avons procédé ainsi, car nous voulions que chaque distance ait la même chance d'être choisie pour le recyclage. Il faut se rappeler qu'un message m dont le mot de code recyclé qui lui est associé a une longueur de l bits a une probabilité de 2^{-l} d'être choisi. De plus, si chaque distance a la même chance d'être choisie, il semblait judicieux d'utiliser un encodage plat pour les encoder afin que le coût net d'encodage de chacun des messages soit semblable (le coût net d'encodage étant la différence entre le coût d'encodage et le nombre de bits recyclés).

Développement du prototype

Le recyclage de bits s'effectue en présence de messages équivalents. Nous avons donc besoin de l'ensemble des plus longues copies et non seulement de la plus longue copie quand nous sommes en présence de cette dernière. Pour ce faire, nous avons modifié la fonction qui recherche la plus longue copie afin qu'elle nous retourne l'ensemble des plus longues copies (ou plus précisément, l'ensemble des distances des plus longues copies).

Ensuite, nous avons modifié la façon dont le fichier compressé est produit. Originalement, lors de la deuxième passe, **gzip** envoyait les messages encodés au décompresseur. Pour faire le recyclage de bits, il faut bloquer cette émission hâtive des messages et remplacer la deuxième passe par deux sous-passes. En effet, en présence de messages équivalents, il faut maintenant choisir quelle distance envoyer au décompresseur. Cependant, le non-déterminisme de notre algorithme recyclant ne nous permet pas de choisir et d'envoyer du même coup cette distance, d'où la nécessité de procéder en deux sous-passses.

Lors de la première sous-passe, nous faisons le recyclage de bits. Malheureusement, l'algorithme recyclant tel que nous l'avons vu précédemment est très difficile à utiliser en pratique à cause de son non-déterminisme. Il nous faut donc un moyen pour contourner le non-déterminisme de notre algorithme recyclant et c'est d'ailleurs pour cette raison que nous procédons en deux sous-passes. L'idée générale est de traiter les messages dans l'ordre inverse. Plus précisément, nous commençons avec le dernier message (m_n) et nous terminons avec le premier (m_1). Ainsi, notre algorithme n'est plus non-déterministe. Lorsque nous devons faire le recyclage de bits, c'est-à-dire lorsque nous devons choisir une distance parmi l'ensemble des distances, nous connaissons maintenant le ou les prochains messages à transmettre au décompresseur.

Pour chacun des messages, il faut regarder ce que nous a retourné la fonction qui

recherche la plus longue copie. Il faut se rappeler que nous avons modifié la fonction qui recherche la plus longue copie afin qu'elle nous retourne l'ensemble des plus longues copies (ou encore, l'ensemble des messages équivalents) si un tel ensemble existe. Lorsque nous ne sommes pas en présence de messages équivalents, il n'y a aucun traitement spécifique à appliquer. Par contre, lorsque nous sommes en présence de messages équivalents, nous procédons comme suit.

Chacune des distances se voit attribuer un mot de code produit par la fonction R^{flat}. Ce sont les bits d'un de ces mots de code qui seront recyclés. Pour ce faire, les bits de chacun des mots de code sont comparés aux bits utilisés pour encoder le ou les prochains messages (ce ou ces prochains messages sont déjà connus, car nous procédons dans l'ordre inverse). Le mot de code dont les bits sont tous identiques aux bits utilisés pour encoder le ou les prochains messages est celui qui est utilisé pour le recyclage. Ce qu'il reste à faire, c'est d'enlever ces bits du ou des prochains messages et de choisir la distance associée au mot de code utilisé.

Lors de la deuxième sous-passe, il ne reste qu'à envoyer les messages encodés et modifiés par le recyclage au décompresseur.

Finalement, nous avons modifié la fonction servant à encoder les distances. Au lieu d'utiliser l'encodage de **gzip** (C_2^{gzip}), nous utilisons l'encodage plat (C_2^{flat}).

Mesures empiriques

Pour vérifier l'efficacité du recyclage de bits, nous avons compressé les 18 fichiers du corpus de Calgary (voir Annexe A) avec nos prototypes. De plus, bien que nous parlons exclusivement du compresseur dans la description de nos expériences, nous avons toujours développé un décompresseur afin de nous assurer que nos résultats expérimentaux proviennent d'implantations d'algorithmes de compression et de décompression valides.

Les résultats de l'expérience 1 sont présentés dans le tableau 6.1. La première colonne contient le nom des fichiers qui ont été compressés, la deuxième colonne contient la taille des fichiers avant compression (en octets), la troisième colonne contient la taille des fichiers compressés par **gzip** (en mode compression maximale) et la quatrième colonne contient la taille des fichiers compressés avec le prototype.

Analyse des résultats

Les résultats de l'expérience 1 montrent que pour certains fichiers, le recyclage de bits permet effectivement de réduire la taille des fichiers compressés. Cependant, la taille de plusieurs fichiers a augmentée. Ces augmentations sont attribuables au fait que nous utilisons un encodage plat pour encoder les distances. Étant fixé a priori, l'encodage plat ne tient pas compte du fait que les distances peuvent avoir des coûts d'encodage très différents. Ainsi, nous ne tenons pas compte des similarités que l'on peut retrouver localement dans les données à compresser ou encore des régularités spatiales.

6.6.3 Expérience 2

Idée

Pour l'expérience 2, nous avons utilisé l'encodage de **gzip** pour encoder les distances (C_2^{gzip}) et nous avons utilisé le recyclage plat (R^{flat}) pour le recyclage de bits. Évidemment, en utilisant C_2^{gzip} pour encoder les distances, ce ne sont pas toutes les distances qui sont éligibles pour le recyclage. De plus, nous avons ajouté un critère d'éligibilité supplémentaire pour les distances. Pour qu'une distance puisse être choisie, il faut que C_2^{gzip} lui ait assigné un mot de code de longueur minimale. Par exemple, si nous avons un ensemble de 5 distances $\{d_1, d_2, d_3, d_4, d_5\}$ et que $|C_2^{gzip}(d_1)| = 4$, $|C_2^{gzip}(d_2)| = 4$, $|C_2^{gzip}(d_3)| = 6$, $|C_2^{gzip}(d_4)| = 8$ et $|C_2^{gzip}(d_5)| = 8$, alors seules les distances d_1 et d_2 sont éligibles pour le recyclage. Cette expérience a pour but de pallier aux difficultés de l'expérience 1 causées par l'utilisation d'un encodage plat pour les distances. Évidemment, les possibilités de recyclage de l'expérience 2 sont moindres, car on effectue le recyclage avec seulement un (petit) sous-ensemble des messages équivalents disponibles. Cependant, en utilisant que les distances ayant un coût minimal, on devrait rarement produire un fichier compressé plus gros que celui produit par **gzip**.

Développement du prototype

Le prototype de l'expérience 2 est basé sur celui de l'expérience 1. Cependant, nous sommes revenus à l'encodage de **gzip** pour encoder les distances (C_2^{gzip}). De plus, lorsque nous sommes en présence de messages équivalents, nous excluons tous les messages dont l'encodage de la distance n'est pas de longueur minimale dans les choix possibles pour le recyclage.

Mesures empiriques

Les résultats de l'expérience 2 sont présentés dans la colonne étiquetée *Exp. 2* du tableau 6.1.

Analyse des résultats

Les avis sur le succès de l'expérience 2 sont partagés. D'un côté, on peut remarquer que certains fichiers comme *book1*, *book2* et *geo* se compressent moins bien avec le deuxième prototype qu'avec le premier. Cela est une conséquence directe des possibilités de recyclage moindres. D'un autre côté, on peut remarquer que le nouveau prototype fait toujours mieux que **gzip** sauf pour un ficher : *obj1*. L'utilisation de l'encodage de **gzip** pour encoder les distances est donc appropriée.

6.6.4 Expérience 3

Idée

L'expérience 3 est une généralisation de l'expérience 2. Nous avons encore utilisé l'encodage de **gzip** pour encoder les distances (C_2^{gzip}) et nous avons encore utilisé le recyclage plat (R^{flat}) pour le recyclage de bits. Cependant, nous sommes plus permissifs quant aux distances qui sont éligibles. En effet, toutes les distances qui sont encodables, c'est-à-dire celles auxquelles l'encodage de **gzip** assigne un mot de code, sont éligibles. Contrairement aux deux premières expériences, on doit maintenant considérer le fait que certaines distances coûtent plus cher à encoder que d'autres. Le but de cette expérience est de se donner plus de possibilités de recyclage tout en utilisant l'encodage de **gzip**.

Développement du prototype

Le prototype de l'expérience 3 est basé sur celui de l'expérience 2. Cependant, nous avons enlevé la restriction concernant les distances qui sont éligibles. Maintenant, toutes les distances qui sont encodables par l'encodage de **gzip** (C_2^{gzip}) sont éligibles.

Mesures empiriques

Les résultats de l'expérience 3 sont présentés dans la colonne étiquetée *Exp. 3* du tableau 6.1.

Analyse des résultats

Les résultats de l'expérience 3 sont très intéressants. Presque tous les fichiers sont mieux compressés avec le nouveau prototype qu'avec le précédent, à l'exception de quelques-uns dont l'écart est négligeable. De plus, ce qui est important de constater, c'est que les bénéfices que l'on retire du recyclage de bits sont beaucoup plus importants lorsqu'on se donne plus de possibilités et ce, même si certaines distances coûtent plus cher à encoder.

6.6.5 Expérience 4

Idée

L'expérience 4 est une réaction directe à l'expérience 3. Nous avons constaté que le recyclage est plus performant lorsqu'on se donne plus de possibilités. Malheureusement, l'expérience 3 est limitée aux distances ayant un mot de code, ce qui est en fait une particularité de l'encodage de **gzip**. Le prototype de l'expérience 4 utilise donc l'encodage C_2^{full} pour encoder les distances afin que toutes les distances soient éligibles pour le recyclage. Nous avons encore utilisé le recyclage plat (R^{flat}) pour le recyclage de bits.

Développement du prototype

Le prototype de l'expérience 4 est basé sur celui de l'expérience 3. Cependant, nous avons remplacé l'encodage de **gzip** (C_2^{gzip}) par l'encodage C_2^{full} pour encoder les distances.

Mesures empiriques

Les résultats de l'expérience 4 sont présentés dans la colonne étiquetée *Exp. 4* du tableau 6.1.

Analyse des résultats

Les résultats de l'expérience 4 confirment ce que l'on a observé à l'expérience 3. Plus grandes sont les possibilités de recyclage, plus importants sont les bénéfices. Nous avons donc intérêt à utiliser le plus grand nombre possible de distances, même si certaines d'entre elles coûtent plus cher à encoder que d'autres.

6.6.6 Expérience 5

Idée

L'expérience 5 reprend les idées de l'expérience 3 en utilisant le recyclage proportionnel ($R^{\propto}$) au lieu du recyclage plat (R^{flat}). Le recyclage proportionnel a pour but de niveler le plus possible le coût net de la transmission lorsque nous sommes en présence de messages équivalents, peu importe la distance qui est choisie et le nombre de bits qu'il faut pour l'encoder. D'un point de vue pratique, plus une distance coûte cher à encoder, moindres sont les chances qu'elle soit choisie mais plus grand est le nombre de bits qui sont recyclés si elle est choisie. L'inverse est aussi vrai. Moins une distance coûte cher à encoder, meilleures sont les chances qu'elle soit choisie mais plus petit est le nombre de bits qui sont recyclés si elle est choisie. L'idée derrière le recyclage proportionnel est de ne pas se pénaliser en choisissant des distances qui coûtent plus cher à encoder. Finalement, comme dans l'expérience 3, les distances sont encodées avec l'encodage C_2^{gzip}.

Développement du prototype

Le prototype de l'expérience 5 est basé sur celui de l'expérience 3. Cependant, nous avons remplacé la fonction de recyclage R^{flat} par $R^{\propto}$. Évidemment, le coût d'encodage des distances a maintenant un impact sur la fonction de recyclage.

Mesures empiriques

Les résultats de l'expérience 5 sont présentés dans la colonne étiquetée *Exp. 5* du tableau 6.1.

Analyse des résultats

Les résultats de l'expérience 5 nous montrent que le recyclage proportionnel est très avantageux. En effet, tous les fichiers se compressent mieux avec le nouveau prototype qu'avec le troisième sur lequel il est basé. Décidément, le recyclage proportionnel nous permet de ne pas nous pénaliser en choisissant des distances qui coûtent plus cher à encoder.

6.6.7 Expérience 6

Idée

L'expérience 6 reprend les idées des expériences 4 et 5. Premièrement, les distances sont encodées avec l'encodage C_2^{full} afin que toutes les distances soient éligibles pour le recyclage. Deuxièmement, on utilise le recyclage proportionnel ($R^{\propto}$). Ainsi, on se donne un maximum de possibilités de recyclage tout en se protégeant des distances qui coûtent cher à encoder grâce au recyclage proportionnel.

Développement du prototype

Le prototype de l'expérience 6 est basé sur celui de l'expérience 4. Cependant, nous avons remplacé la fonction de recyclage R^{flat} par $R^{\propto}$. Évidemment, le coût d'encodage des distances a un impact sur la fonction de recyclage.

Mesures empiriques

Les résultats de l'expérience 6 sont présentés dans la colonne étiquetée *Exp. 6* du tableau 6.1.

TABLE 6.2 – Résultats des expériences 7 à 9

Fichier	Original	Gzip	Exp. 6	Exp. 7	Exp. 8	Exp. 9
bib	111 261	34 900	33 863	33 863	33 829	31 757
book1	768 771	312 281	301 888	301 888	301 538	279 435
book2	610 856	206 158	200 165	200 165	199 906	185 321
geo	102 400	68 414	65 889	65 889	66 133	63 341
news	377 106	144 400	140 418	140 418	140 142	132 679
obj1	21 504	10 320	10 314	10 314	10 304	10 043
obj2	246 814	81 087	79 179	79 179	79 068	75 360
paper1	53 161	18 543	18 147	18 147	18 129	16 938
paper2	82 199	29 667	28 933	28 933	28 892	26 720
paper3	46 526	18 074	17 689	17 689	17 675	16 422
paper4	13 286	5 534	5 440	5 440	5 440	5 156
paper5	11 954	4 995	4 922	4 922	4 916	4 688
paper6	38 105	13 213	13 055	13 055	13 031	12 225
pic	513 216	52 381	51 621	51 621	51 440	N/A
progc	39 611	13 261	13 097	13 097	13 069	12 212
progl	71 646	16 164	15 771	15 771	15 704	14 509
progp	49 379	11 186	10 943	10 943	10 911	10 186
trans	93 695	18 862	18 454	18 454	18 420	17 477

Analyse des résultats

Les résultats de l'expérience 6 confirment ce que nous avons découvert suite aux expériences 4 et 5. Premièrement, le recyclage de bits est beaucoup plus performant lorsqu'on utilise le plus grand nombre possible de distances. Deuxièmement, il est très avantageux d'utiliser le recyclage proportionnel pour faire le recyclage de bits. En effet, nous ne sommes plus pénalisés par les distances qui coûtent plus cher à encoder.

6.6.8 Expérience 7

Idée

Dans les expériences 1 à 6, le code pour encoder les distances est produit en fonction des distances qui auraient été utilisées par **gzip** sans le recyclage de bits. Plus précisément, lors de la première passe, **gzip** détermine, par l'application de l'algorithme

Deflate, l'ensemble des distances $\{d_1, d_2, \ldots, d_n\}$ servant à décrire les copies $\langle l, d\rangle$. À la fin de la première passe, **gzip** produit le code C_2 pour encoder les distances en utilisant les statistiques accumulées sur ces distances. Cependant, comme nous l'avons déjà mentionné, le recyclage de bits risque de causer l'emploi d'autres distances, car les distances sont choisies en fonction de leurs mots de code recyclés. Ainsi, suite au recyclage de bits, nous risquons de nous retrouver avec un nouvel ensemble de distances $\{d'_1, d'_2, \ldots, d'_{n'}\}$. Pour l'expérience 7, nous voulons reconstruire un nouveau code pour encoder les distances en utilisant les statistiques accumulées sur ce nouvel ensemble.

Il faut se rappeler que la deuxième passe de **gzip** a été divisée en deux sous-passes pour le recyclage de bits. Lors de la première sous-passe, nous effectuons le recyclage. À la fin de la première sous-passe, nous connaissons l'ensemble des distances $\{d'_1, d'_2, \ldots, d'_{n'}\}$ qui ont été choisies lors du recyclage. Normalement, après la première sous-passe, nous procédons directement avec la deuxième sous-passe. Cependant, cette fois-ci, nous ne procédons pas directement avec la deuxième sous-passe. Nous laissons plutôt tomber les messages encodés et nous ne conservons que le nouvel ensemble de distances $\{d'_1, d'_2, \ldots, d'_{n'}\}$ et les statistique accumulées sur ce nouvel ensemble. Nous utilisons ces nouvelles statistiques pour construire un nouveau code pour encoder les distances. Évidemment, il faut alors recommencer la première sous-passe pour faire le recyclage et encoder les messages à nouveau. Ensuite, nous sommes en mesure de procéder avec la deuxième sous-passe. L'idée derrière cette expérience est de recommencer la première sous-passe en utilisant un encodage différent pour encoder les distances. Cet encodage est produit à partir des distances choisies lors du recyclage (la première fois).

Développement du prototype

Le prototype de l'expérience 7 est basé sur celui de l'expérience 6. Nous avons évidemment introduit de la nouvelle machinerie afin de pouvoir accumuler de nouvelles statistiques basées sur les distances choisies lors du recyclage. Plus précisément, lors du recyclage et de l'encodage des messages, nous accumulons désormais des statistiques sur les distances qui sont choisies. Ensuite, au lieu d'envoyer les messages encodés au décompresseur, nous recommençons une seconde fois le recyclage et l'encodage des messages. Ces nouveaux messages encodés sont finalement transmis au décompresseur.

Mesures empiriques

Les résultats de l'expérience 7 sont présentés dans la colonne étiquetée *Exp. 7* du tableau 6.2.

Analyse des résultats

Les résultats de l'expérience 7 nous ont surpris à première vue. Les fichiers compressés avec le prototype 7 ont la même taille que ceux compressés avec le prototype 6. Nous avons donc regardé si les fichiers compressés avec les deux prototypes étaient identiques et ce fut le cas. L'explication qu'on peut donner est que les nouvelles statistiques sont semblables aux statistiques originales. Il faut se rappeler que les statistiques sont valides pour l'ensemble d'un bloc. Ainsi, le code pour encoder les distances n'a tout simplement pas changé. Nous n'avons donc pas avantage à accumuler de nouvelles statistiques basées sur les distances qui sont choisies lors du recyclage.

6.6.9 Recyclage proportionnel optimal

Comme le recyclage de bits proportionnel, le recyclage de bits proportionnel *optimal* prend en compte le coût d'encodage des messages équivalents $\{m_1, m_2, \ldots, m_n\}$ par la fonction C dans la construction du code qui est utilisé pour le recyclage. De plus, le recyclage de bits proportionnel optimal soustrait de l'ensemble des messages équivalents le sous-ensemble des messages qui font chuter les performances du recyclage, car ils coûtent trop cher à encoder pour les bénéfices qu'on en retire. Ainsi, nous pouvons nous assurer de l'optimalité du code qui est utilisé pour le recyclage. Notez que pour la suite de cette section, nous parlerons de l'ensemble des messages équivalents $\{m_1, m_2, \ldots, m_n\}$ comme étant l'ensemble des symboles $\{a_1, a_2, \ldots, a_n\}$.

La fonction de coût K est définie comme suit : $\forall a \in \{a_1, \ldots, a_n\}\ K(a) = |C(a)|$. Soit r un code de recyclage pour K (r n'est pas nécessairement optimal). r doit être défini pour au moins un symbole, i.e. $\emptyset \neq Dom(r) \subseteq Dom(K)$. Soit a_i un symbole dans $Dom(r)$. Le coût net de a_i est $K(a_i) - |r(a_i)|$. Le coût espéré du recyclage avec r est le coût net espéré des symboles dans $Dom(r)$, i.e. $\sum_{a_i \in Dom(r)}(K(a_i) - |r(a_i)|)/2^{|r(a_i)|}$. Nous disons que r est optimal si le coût espéré du recyclage avec r est minimal parmi tous les codes de recyclage valides pour K.

Pour le recyclage de bits proportionnel optimal, nous obtenons r en construisant un

arbre de recyclage. Un arbre de recyclage contient au moins un symbole et au plus tous les symboles dans $\{a_1, a_2, \ldots, a_n\}$. De plus, chaque symbole peut apparaître au plus une fois dans l'arbre de recyclage. L'opérateur $R(t_1, t_2)$ est utilisé pour représenter l'arbre composé des arbres t_1 et t_2. R peut aussi être utilisé comme fonction pour créer un arbre à partir de deux arbres. Le coût d'un arbre de recyclage est défini récursivement grâce à la fonction **coût**. Le coût d'un symbole a est coût$(a) = K(a)$, c'est-à-dire la longueur (en bits) de l'encodage du symbole a. Évidemment, il s'agit aussi du coût net de la transmission de a, car a tout seul nous permet de recycler aucun bit. Le coût d'un arbre composé des arbres t_1 et t_2 est coût$(R(t_1, t_2)) = \frac{\text{coût}(t_1)+\text{coût}(t_2)}{2} - 1$. Le coût net de la transmission de t_1 et de t_2 est donc le coût moyen de leur transmission moins un bit qui est recyclé, car nous pouvons choisir une option parmi 2 options et recycler un bit par le fait même.

Définissons maintenant l'arbre de recyclage optimal. Soit τ l'ensemble des arbres de recyclage possibles. τ est le plus petit ensemble qui inclut $\{a_1, \ldots, a_n\}$ et $\{R(t_1, t_2) | t_1, t_2 \in \tau$ et aucun a_i n'apparaît dans t_1 et $t_2\}$. L'arbre de recyclage optimal est $arg_min_{t \in \tau}$coût(t).

Nous construisons un arbre de recyclage optimal à l'aide de $R^{\propto opt}$:

$$R^{\propto opt}(\{a_1, \ldots, a_n\}, C) = RecOpt(\{(a_1, |C(a_1)|), \ldots, (a_n, |C(a_n)|)\})$$

La fonction *RecOpt* construit un arbre de recyclage optimal et retourne un code de recyclage optimal. Voici la définition de cette fonction :

RecOpt$(\{(a_1, c_1), \ldots, (a_n, c_n)\})$:

1. Sans perte de généralité, supposons que $c_i \leq c_{i+1}$ pour $1 \leq i \leq n-1$.
2. Si $n = 1$, retourner a_1.
3. Sinon,
   (a) Si $c_{n-1} \geq \frac{c_{n-1}+c_n}{2} - 1$, alors nous pouvons conserver a_n.
   Créer un nouveau symbole $\bar{a}$ tel que son coût $\bar{c} = \frac{c_{n-1}+c_n}{2} - 1$.
   Le symbole $\bar{a}$ sert de *représentant* de $R(a_{n-1}, a_n)$.
   Soit $t = RecOpt(\{(a_1, c_1), \ldots, (a_{n-2}, c_{n-2}), (\bar{a}, \bar{c})\})$.
   Si $\bar{a}$ apparaît dans t, alors remplacer $\bar{a}$ par $R(a_{n-1}, a_n)$.
   (b) Si $c_{n-1} \leq \frac{c_{n-1}+c_n}{2} - 1$, alors nous pouvons rejeter a_n.
   Retourner $RecOpt(\{(a_1, c_1), \ldots, (a_{n-1}, c_{n-1})\})$.

Une fois l'arbre construit, nous sommes en mesure d'obtenir le mot de code recyclé de chacun des symboles dans l'arbre. Ainsi, nous pouvons retourner un code de recyclage valide. Pour ce faire, il suffit simplement d'accumuler les bits dans l'ordre

racine → feuille (chacune des feuilles représente un symbole). On peut donc retrouver le mot de code recyclé associé à un symbole en temps linéaire au nombre de bits qu'il contient.

Avant de démontrer que l'algorithme produit toujours un arbre de recyclage optimal, nous introduisons différents lemmes sur les arbres de recyclage (optimaux ou pas, dépendamment du lemme). Ensuite, nous présentons la preuve d'optimalité de l'arbre produit par l'algorithme.

1. **Lemme sur les symboles qui apparaissent dans l'arbre de recyclage optimal**
   Soient $a_1, \ldots, a_n$ tels que $\text{coût}(a_i) \leq \text{coût}(a_{i+1})$ pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Soit a_i un symbole qui apparaît dans t.
   Alors pour tout $j < i$, a_j apparaît dans t.
   Preuve par l'absurde :
   Soit a_j un symbole qui n'apparaît pas dans t tel que $j < i$ ($\text{coût}(a_j) \leq \text{coût}(a_i)$).
   Remplaçons, dans t, a_i par $R(a_i, a_j)$ pour obtenir t'.
   Nous constatons alors que $\text{coût}(t') < \text{coût}(t)$.
   Contradiction, car t est optimal.
2. **Autre lemme sur les symboles qui apparaissent dans l'arbre de recyclage optimal**
   Soient $a_1, \ldots, a_n$ tels que $\text{coût}(a_i) \leq \text{coût}(a_{i+1})$ pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Soit a_i un symbole qui apparaît dans t.
   Soit a_j ($j \neq i$) un symbole qui n'apparaît pas dans t.
   Alors $\text{coût}(a_j) - \text{coût}(a_i) \geq 2$.
   Preuve par l'absurde :
   Supposons que $\text{coût}(a_j) - \text{coût}(a_i) < 2$.
   Remplaçons, dans t, a_i par $R(a_i, a_j)$ pour obtenir t'.
   Nous constatons alors que $\text{coût}(t') < \text{coût}(t)$.
   Contradiction, car t est optimal.
3. **Lemme sur le coût des symboles dans l'arbre de recyclage optimal**
   Soient $a_1, \ldots, a_n$ tels que $\text{coût}(a_i) \leq \text{coût}(a_{i+1})$ pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Supposons que t comporte au moins 2 symboles.
   Soient a_i et a_j deux symboles distincts qui apparaissent dans t.
   Si a_j est à un niveau inférieur à a_i dans l'arbre de recyclage optimal (les feuilles sont au niveau inférieur dans l'arbre),
   alors $\text{coût}(a_i) \leq \text{coût}(a_j)$.

Preuve par l'absurde :
Supposons que coût$(a_i) >$ coût(a_j).
Interchangeons de position, dans t, a_i et a_j pour obtenir t' tel que coût$(t') <$ coût(t).
Contradiction, car t est optimal.

4. **Lemme sur les symboles de même coût**
   Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage pour $a_1, \ldots, a_n$.
   Supposons que t comporte au moins 2 symboles.
   Soient a_i et a_j deux symboles distincts de même coût qui apparaissent dans t.
   Alors nous pouvons interchanger leurs positions sans changer le coût de t.
   Notez que nous ne démontrons pas ce lemme, car le résultat est évident.

5. **Lemme sur le changement de niveau**
   Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Supposons que t comporte au moins 2 symboles.
   Soient a_i et a_j deux symboles distincts qui apparaissent dans t.
   Supposons que $i < j$.
   Si a_i est à un niveau strictement inférieur à a_j dans t,
   alors nous pouvons interchanger de position a_i et a_j.
   Preuve :
   Grâce au lemme 3, nous savons que coût$(a_j) \leq$ coût(a_i).
   De plus, comme $i < j$, coût$(a_i) \leq$ coût(a_j).
   Alors les deux symboles ont forcément le même coût.
   Nous pouvons donc interchanger leurs positions grâce au lemme 4.

6. **Lemme sur le changement de niveau pour a_{n-1} et a_n**
   Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Si a_{n-1} et a_n apparaissent dans t,
   alors nous pouvons modifier la forme de t, si nécessaire, pour ramener a_{n-1} et a_n au niveau inférieur sans changer le coût de t.
   Preuve pour a_n :
   Si a_n n'est pas déjà au niveau inférieur, choisir un symbole a_i au niveau inférieur, alors $i < n$ et nous pouvons utiliser le lemme 5 pour interchanger de position a_n et a_i.
   Preuve pour a_{n-1} :
   Si a_{n-1} n'est pas déjà au niveau inférieur, choisir un symbole a_i autre que a_n au niveau inférieur.
   Un tel a_i existe, parce qu'il y a un nombre pair de symboles au niveau inférieur (t est un arbre *binaire*).

Alors $i < n-1$ et nous pouvons utiliser le lemme 5 pour interchanger de position a_{n-1} et a_i.

7. **Lemme sur le changement de parent**
   Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage pour $a_1, \ldots, a_n$.
   Soit a_i un symbole qui apparaît dans t.
   Soit a_j $(j \neq i)$ un symbole qui apparaît dans t au même niveau que a_i.
   Nous pouvons interchanger de position a_i et a_j sans changer le coût t.
   Les transformations suivantes ne changent pas le coût d'un arbre :
   $R(t1, t2) \Rightarrow R(t2, t1)$ et $R(R(t1, t2), R(t3, t4)) \Rightarrow R(R(t1, t3), R(t2, t4))$
   et elles permettent d'interchanger de position deux noeuds situés au même niveau.
   Notez que ce lemme ne sera pas démontré formellement.
8. **Lemme sur le changement de parent pour a_{n-1} et a_n**
   Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Supposons que a_{n-1} et a_n apparaissent dans t.
   Alors nous pouvons modifier la forme de t, si nécessaire, pour ramener a_{n-1} et a_n sous le même parent sans changer le coût de t.
   Preuve :
   Grâce au lemme 6, nous pouvons ramener a_{n-1} et a_n au niveau inférieur dans t.
   Si a_{n-1} et a_n ne sont pas sous le même parent,
   alors nous pouvons interchanger de position a_{n-1} et le frère de a_n grâce au lemme 7.
9. **Lemme sur l'optimalité de l'arbre de recyclage pour la liste courte**
   Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
   Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
   Si a_n n'apparaît pas dans t,
   alors t est aussi un arbre de recyclage optimal pour $a_1, \ldots, a_{n-1}$.
   Preuve :
   Soit τ l'ensemble des arbres de recyclage pour $a_1, \ldots, a_n$.
   Soit τ' l'ensemble des arbres de recyclage pour $a_1, \ldots, a_{n-1}$.
   Nous savons que $\tau' \subset \tau$.
   L'arbre de recyclage t est optimal pour $a_1, \ldots, a_{n-1}$ car $t \in \tau'$ et coût$(t) = \min_{t' \in \tau}$ coût$(t') \leq \min_{t' \in \tau'}$ coût$(t') \leq$ coût(t).
10. **Preuve d'optimalité de l'arbre produit par l'algorithme**
    Théorème :
    Soient $a_1, \ldots, a_n$ tels que coût$(a_i) \leq$ coût(a_{i+1}) pour $1 \leq i \leq n-1$ et $n \geq 1$.
    Soit t un arbre de recyclage optimal pour $a_1, \ldots, a_n$.
    Soit t^* un arbre construit par l'algorithme pour $a_1, \ldots, a_n$.
    Alors coût$(t^*) =$ coût(t).
    Preuve par induction :

Cas de base : $n = 1$ et $t = a_1 = t^*$.
Hypothèse d'induction :
Supposons que le théorème est vrai pour $n = 1 \dots m$.
Pas : $n = m + 1$. a_m et a_{m+1} sont extraits de la liste.

- **Cas 1 :** $\text{coût}(a_{m+1}) > \text{coût}(a_m) + 2$.
  L'algorithme rejette a_{m+1}.
  Montrons que a_{m+1} n'apparaît pas dans t.
  Preuve par l'absurde :
  Supposons que a_{m+1} apparaît dans t.
  Grâce au lemme 8, nous pouvons modifier la forme de t, si nécessaire, pour ramener a_m et a_{m+1} sous le même parent tout en conservant l'optimalité de t.
  Par définition, si $\text{coût}(a_{m+1}) > \text{coût}(a_m) + 2$, alors $\text{coût}(R(a_m, a_{m+1})) > \text{coût}(a_m)$.
  Remplaçons, dans t, $R(a_m, a_{m+1})$ par a_m pour obtenir t' tel que $\text{coût}(t') < \text{coût}(t)$.
  Contradiction, car t est optimal.
  Par hypothèse d'induction, nous savons que l'algorithme produit un arbre de recyclage optimal t^* pour $a_1, \dots, a_m$.
  Montrons que t^* est aussi un arbre de recyclage optimal pour $a_1, \dots, a_{m+1}$.
  Rappelons-nous que l'algorithme avait décidé de rejeter a_{m+1}.
  Montrons que a_{m+1} n'apparaît pas dans t^*.
  La preuve est similaire à la preuve précédente.
- **Cas 2 :** $\text{coût}(a_{m+1}) < \text{coût}(a_m) + 2$.
  L'algorithme conserve a_{m+1}.
  - **Cas 2.1 :** a_m et a_{m+1} apparaissent dans t.
    Grâce au lemme 8, nous pouvons modifier la forme de t, si nécessaire, pour ramener a_m et a_{m+1} sous le même parent tout en conservant l'optimalité de t.
    Remplaçons, dans t, $R(a_m, a_{m+1})$ par un nouveau symbole $a_{m'}$ de même coût (i.e. $\text{coût}(a_{m'}) = \text{coût}(R(a_m, a_{m+1}))$) pour obtenir t' tel que $\text{coût}(t') = \text{coût}(t)$.
    Soit $L = insert(a_{m'}; a_1, \dots, a_{m-1})$.
    Alors t' est un arbre de recyclage optimal pour L.
    Preuve par l'absurde (2 cas) :
    Soit t'' un meilleur arbre que t' pour L.
    - **Cas 2.1.1 :** $a_{m'}$ apparaît dans t''.
      Remplaçons, dans t'', $a_{m'}$ par $R(a_m, a_{m+1})$ pour obtenir t''' tel que $\text{coût}(t''') = \text{coût}(t'')$.
      Alors t''' est un arbre de recyclage pour $a_1, \dots, a_{m+1}$ tel que $\text{coût}(t''') = \text{coût}(t'') < \text{coût}(t') = \text{coût}(t)$.

Contradiction, car t est optimal.

- **Cas 2.1.2 :** $a_{m'}$ n'apparaît pas dans t''.
  t'' est déjà un arbre de recyclage pour $a_1, \ldots, a_m$ tel que coût$(t'') <$ coût(t') = coût(t).
  Contradiction, car t est optimal.

Par hypothèse d'induction, nous savons que l'algorithme produit un arbre de recyclage optimal t^* pour L.
Montrons que t^* est aussi un arbre de recyclage optimal pour $a_1, \ldots, a_{m+1}$.
Rappelons-nous que l'algorithme avait décidé d'agglomérer a_m et a_{m+1}.

- **Cas 2.1.A :** $a_{m'}$ apparaît dans t^*.
  Remplaçons, dans t^*, $a_{m'}$ par $R(a_m, a_{m+1})$ pour obtenir t^{**} tel que coût(t^{**}) = coût(t^*).
  Alors t^{**} est un arbre de recyclage optimal pour $a_1, \ldots, a_{m+1}$.
  Preuve par l'absurde :
  Soit t^{***} un meilleur arbre que t^{**} pour $a_1, \ldots, a_{m+1}$.
  Avec des arguments similaires à ceux du cas 2.1.1, nous pourrions démontrer que cela implique qu'il existe un meilleur arbre que t^* pour L.
  Contradiction, car t^* est optimal.
- **Cas 2.1.B :** $a_{m'}$ n'apparaît pas dans t^*.
  L'algorithme n'a rien à faire et t^* est forcément un arbre de recyclage optimal pour $a_1, \ldots, a_{m+1}$.
  Preuve par l'absurde :
  Soit t^{**} un meilleur arbre que t^* pour $a_1, \ldots, a_{m+1}$.
  Avec des arguments similaires à ceux du cas 2.1.2, nous pourrions démontrer que cela implique qu'il existe un meilleur arbre que t^* pour L.
  Contradiction, car t^* est optimal.

- **Cas 2.2 :** a_m apparaît dans t mais pas a_{m+1}.
  Contradiction du lemme 2 sur les symboles qui apparaissent dans l'arbre de recyclage optimal.
- **Cas 2.3 :** a_{m+1} apparaît dans t mais pas a_m.
  Contradiction du lemme 1 sur les symboles qui apparaissent dans l'arbre de recyclage optimal.
- **Cas 2.4 :** a_m et a_{m+1} n'apparaissent pas dans t.
  Soit $a_{m'}$ un nouveau symbole qui représente $R(a_m, a_{m+1})$ et tel que coût$(a_{m'})$ = coût$(R(a_m, a_{m+1}))$.
  Alors t est aussi un arbre de recyclage optimal pour
  $L = insert(a_{m'}; a_1, \ldots, a_{m-1})$.
  Le reste de la preuve est similaire à celle du cas 2.1.

- **Cas 3 :** coût(a_{m+1}) = coût$(a_m) + 2$.

- **Cas 3.1 :** l'algorithme conserve a_{m+1}.
  - **Cas 3.1.1 :** a_{m+1} apparaît dans t.
    La preuve est similaire à celle du cas 2.1.
  - **Cas 3.1.2 :** a_{m+1} n'apparaît pas dans t.
    - **Cas 3.1.2.1 :** a_m apparaît dans t.
      Remplaçons a_m par $R(a_m, a_{m+1})$ dans t.
      Ceci ne change par le coût de t (voir cas 3.1.1).
    - **Cas 3.1.2.2 :** a_m n'apparaît pas dans t.
      La preuve est similaire à celle du cas 2.4.
- **Cas 3.2 :** l'algorithme rejette a_{m+1}.
  - **Cas 3.2.1 :** a_{m+1} apparaît dans t.
    Grâce au lemme 8, nous pouvons modifier la forme de t, si nécessaire, pour ramener a_m et a_{m+1} sous le même parent tout en conservant l'optimalité de t.
    Comme coût(a_{m+1}) = coût$(a_m) + 2$, alors coût$(R(a_m, a_{m+1}))$ = coût(a_m).
    Remplaçons, dans t, $R(a_m, a_{m+1})$ par a_m pour obtenir t' tel que coût(t') = coût(t).
    t' est donc encore un arbre de recyclage optimal pour $a_1, \ldots, a_{m+1}$.
    Finalement, grâce au lemme 9, nous savons que t' est aussi un arbre de recyclage optimal pour $a_1, \ldots, a_m$.
    Par hypothèse d'induction, nous savons que l'algorithme produit un arbre de recyclage optimal t^* pour $a_1, \ldots, a_m$.
    Montrons que t^* est aussi un arbre de recyclage optimal pour $a_1, \ldots, a_{m+1}$.
    Rappelons-nous que l'algorithme avait décidé de rejeter a_{m+1}.
    Preuve par l'absurde :
    Soit t^{**} un meilleur arbre que t^* pour $a_1, \ldots, a_{m+1}$.
    Avec des arguments similaires à ceux de la preuve précédente, nous pourrions démontrer que cela implique qu'il existe un meilleur arbre que t^* pour $a_1, \ldots, a_m$.
    Contradiction, car t^* est optimal pour $a_1, \ldots, a_m$.
  - **Cas 3.2.2 :** a_{m+1} n'apparaît pas dans t.
    Grâce au lemme 9, nous savons que t est déjà un arbre de recyclage optimal pour $a_1, \ldots, a_m$.
    Par hypothèse d'induction, nous savons que l'algorithme produit un arbre de recyclage optimal t^* pour $a_1, \ldots, a_m$.
    Montrons que t^* est aussi un arbre de recyclage optimal pour $a_1, \ldots, a_{m+1}$.
    Rappelons-nous que l'algorithme avait décidé de rejeter a_{m+1}.
    Preuve par l'absurde :
    Soit t^{**} un meilleur arbre que t^* pour $a_1, \ldots, a_{m+1}$.
    Avec des arguments similaires à ceux de la preuve précédente, nous pour-

rions démontrer que cela implique qu'il existe un meilleur arbre que t^* pour $a_1, \ldots, a_m$.
Contradiction, car t^* est optimal $a_1, \ldots, a_m$.

6.6.10 Expérience 8

Idée

L'expérience 8 reprend l'expérience 6 en utilisant, cette fois-ci, le recyclage proportionnel optimal ($R^{\propto opt}$) au lieu du recyclage proportionnel ($R^{\propto}$). Le recyclage proportionnel optimal a pour but de soustraire de l'ensemble des distances éligibles le sous-ensemble des distances qui font chuter les performances du recyclage, car elles coûtent trop cher à encoder pour les bénéfices qu'on en retire. Comme dans l'expérience 6, les distances sont encodées avec l'encodage C_2^{full} afin que toutes les distances soient éligibles pour le recyclage. Évidemment, même si toutes les distances sont éligibles pour le recyclage, certaines distances risquent d'être éliminées par l'utilisation de $R^{\propto opt}$.

Développement du prototype

Le prototype de l'expérience 8 est basé sur celui de l'expérience 6. Cependant, nous avons remplacé la fonction de recyclage $R^{\propto}$ par $R^{\propto opt}$. De plus, il faut constater que pour la première fois, la fonction de recyclage a un impact sur les distances qui peuvent être choisies lors du recyclage. Certaines distances peuvent être rejetées lors du recyclage de bits.

Mesures empiriques

Les résultats de l'expérience 8 sont présentés dans la colonne étiquetée *Exp. 8* du tableau 6.2. Notez que le prototype de l'expérience 8 est de 2 à 10 fois plus lent que **gzip**. Cependant, plusieurs optimisations pourraient être apportées au prototype afin de le rendre plus rapide.

Analyse des résultats

Les résultats de l'expérience 8 sont très intéressants. Tous les fichiers, sauf un, sont mieux compressés avec le nouveau prototype qu'avec celui de l'expérience 6. Éliminer les distances qui coûtent trop cher à encoder et qui risquent de faire chuter les performances du recyclage est une amélioration sur les expériences précédentes. Maintenant, un lecteur attentif pourrait se demander pourquoi le fichier *geo* est mieux compressé lorsqu'on utilise le recyclage proportionnel ($R^{\propto}$) au lieu du recyclage proportionnel optimal ($R^{\propto opt}$). La réponse est en fait toute simple. Le recyclage proportionnel optimal est bel et bien optimal : le code qui est produit pour faire le recyclage est optimal. Cependant, en utilisant un autre code de recyclage (i.e. $R^{\propto}$), d'autres choix (parmi les messages équivalents) peuvent être faits lors du recyclage. Or, le recyclage proportionnel optimal se contente de construire un code de recyclage optimal, c'est-à-dire un code dont le coût net espéré est minimal. Par conséquent, les décisions prises avec le code de recyclage proportionnel optimal peuvent être parfois plus mauvaises que celles prises avec le code de recyclage proportionnel non-optimal.

6.6.11 Expérience 9

Idée

Quand nous avons réalisé l'expérience 9, notre objectif était d'augmenter considérablement les possibilités de recyclage afin de recycler plus de bits. Nous avons donc décidé de faire le recyclage avec tous les messages qu'il est possible d'envoyer au décompresseur. Dorénavant, nous ne sommes plus contraints de choisir une plus longue copie lorsque des copies existent. Nous ne parlons plus de l'ensemble des plus longues copies mais bien de l'ensemble des copies (toutes les copies, de longueur 3 à 258). À une position donnée dans le fichier à compresser, nous pouvons désormais envoyer au décompresseur un littéral $[c]$ ou une copie $\langle l, d\rangle$ parmi l'ensemble des copies si cet ensemble est non-vide. Évidemment, ces messages ne sont pas tous équivalents : ils ne décrivent pas tous la même suite d'octets. Dépendamment du message que nous envoyons au décompresseur, nous nous retrouvons à une position différente dans le fichier à compresser. En effet, si nous sommes à la position p et que nous envoyons un littéral $[c]$, alors nous nous retrouvons à la position $p + 1$. De même, si nous envoyons une copie $\langle l, d\rangle$, alors nous nous retrouvons à la position $p + l$. Il est donc maintenant possible de traverser le fichier à compresser de plusieurs façons différentes. La figure 6.4 illustre deux exemples de fichiers ainsi que les différentes *traversées*, ou séquences de messages, possibles. Avec le recyclage de bits basé sur les plus longues copies, seules les traversées 1a, 1b et 2a

Exemple 1 :	Traversée 1a :	[a] [b] [c] [1] $\langle 3,4\rangle$ [2] $\langle 3,4\rangle$
F_1^{ex} : abc1abc2abc	Traversée 1b :	[a] [b] [c] [1] $\langle 3,4\rangle$ [2] $\langle 3,8\rangle$
	Traversée 1c :	[a] [b] [c] [1] $\langle 3,4\rangle$ [2] [a] [b] [c]
	Traversée 1d :	[a] [b] [c] [1] [a] [b] [c] [2] $\langle 3,4\rangle$
	Traversée 1e :	[a] [b] [c] [1] [a] [b] [c] [2] $\langle 3,8\rangle$
	Traversée 1f :	[a] [b] [c] [1] [a] [b] [c] [2] [a] [b] [c]
Exemple 2 :	Traversée 2a :	[a] [b] [c] [1] [b] [c] [d] [2] $\langle 3,8\rangle$ [d]
F_2^{ex} : abc1bcd2abcd	Traversée 2b :	[a] [b] [c] [1] [b] [c] [d] [2] [a] $\langle 3,5\rangle$
	Traversée 2c :	[a] [b] [c] [1] [b] [c] [d] [2] [a] [b] [c] [d]

FIGURE 6.4 – Exemples de fichiers avec les différentes traversées possibles

sont valides.

Comme dans l'expérience 8, les distances sont encodées avec l'encodage C_2^{full} afin que toutes les distances soient éligibles pour le recyclage et nous utilisons le recyclage proportionnel optimal ($R^{\propto opt}$).

Recyclage de bits avec tous les messages

Le recyclage de bits avec tous les messages augmente énormément les possibilités de recyclage. Cependant, comme les messages ne sont pas tous équivalents, il est possible de traverser le fichier à compresser de plusieurs façons différentes. Supposons que pendant la compression du fichier F_2^{ex}, les décisions imposées par le recyclage de bits forcent le compresseur à choisir la traversée 2b. Pour les 8 premiers messages, le compresseur n'a pas d'autre choix que d'envoyer un littéral. À la position 9, le compresseur peut envoyer au décompresseur le message $\langle 3,8\rangle$ ou le message $[a]$. Ce choix semble être une opportunité pour le recyclage. Cependant, ces messages ne sont pas équivalents. Selon notre supposition, le compresseur envoie le message $[a]$ au décompresseur. Quand le décompresseur reçoit le message $[a]$, il prend connaissance de l'octet a mais il n'est pas capable de détecter que le compresseur avait deux options, car il ne connaît pas encore les octets aux positions 10 et 11. Heureusement, cela ne veut pas dire que l'opportunité pour le recyclage est perdue : elle est seulement remise à plus tard. Quand le décompresseur reçoit le prochain message, $\langle 3,5\rangle$, il prend connaissance des octets *bcd* et il est capable de détecter que ces 3 octets pouvaient être décrits de 2 façons différentes (avec la traversée 2b ou 2c). Si les deux descriptions sont considérées profitables, un bit peut être recyclé. De plus, le décompresseur est capable de détecter que les octets *abc* pouvaient être décrits de 2 façons différentes (avec la traversée 2a ou 2c) et que les octets *abcd* pouvaient être décrits de 3 façons différentes (avec les 3 traversées). Le délai observé dans la capacité du décompresseur à détecter les opportunités de recyclage est une conséquence directe des principes du recyclage de bits : pour faire le recyclage, ce

n'est pas suffisant pour le compresseur d'avoir des options ; le décompresseur doit aussi avoir la capacité de les détecter.

Définitions

La fonction L nous donne la longueur d'un message. Plus précisément, $L([c]) = 1$ et $L(\langle l, d\rangle) = l$.

Nous disons qu'un message M permet de sauter de la position p à la position q, que nous notons par $p \xrightarrow{M} q$, si M est une description valide des octets que l'on retrouve de la position p à la position $q-1$ inclusivement. Plus précisément, nous avons $p \xrightarrow{[c]} p+1$ si l'octet à la position p est c et nous avons $p \xrightarrow{\langle l,d\rangle} p+l$ si les octets de la position p à la position $q-1$ inclusivement sont identiques aux octets que l'on retrouve de la position $p-d$ à la position $p-d+l-1$ inclusivement.

Une traversée, pour un fichier F, est une séquence de messages $M_0, M_1, \ldots, M_{n-1}$ et une séquence de positions $p_0, p_1, \ldots, p_n$ telles que $p_0 = 0$, $p_n = |\mathsf{F}|$ et $p_i \xrightarrow{M_i} p_{i+1}$ pour tout $0 \leq i \leq n-1$.

Nous notons par $\vec{\mu}(p)$ l'ensemble des messages qui décrivent les octets *à partir de* la position p et nous notons par $\overleftarrow{\mu}(q)$ l'ensemble des messages qui décrivent les octets *jusqu'à* la position q. Formellement, $\vec{\mu}(p) = \{M \mid \exists q.\ p \xrightarrow{M} q\}$ et $\overleftarrow{\mu}(q) = \{M \mid \exists p.\ p \xrightarrow{M} q\}$. Notez que $M \in \vec{\mu}(p)$ si et seulement si $M \in \overleftarrow{\mu}(p + L(M))$.

Algorithmes du compresseur et du décompresseur

Le recyclage de bits avec tous les messages est beaucoup plus complexe que le recyclage de bits basé sur les plus longues copies. Il n'est pas évident de voir comment et quand des bits peuvent être recyclés. De plus, une technique qui manipulerait l'ensemble de toutes les traversées possibles pour un fichier est condamnée à être trop lente, car le nombre de traversées possibles est gigantesque pour la plupart des fichiers. Ainsi, nous devons opter pour une approche où plusieurs traversées peuvent être regroupées ensemble. Cette approche est basée sur la *programmation dynamique* et elle est présentée en 5 parties.

Partie 1 Considérons l'ensemble des traversées possibles pour les *préfixes* de F. Nous notons par P_p le préfixe de F de longueur p. Évidemment, $\mathsf{P}_0 = \epsilon$ et $\mathsf{P}_{|\mathsf{F}|} = \mathsf{F}$. Si deux traversées de P_p, les traversées $M_0, M_1, \ldots, M_{k-1}, M_k$ et $M'_0, M'_1, \ldots, M'_{k-1}, M'_k$, ont comme *dernier* message le même message (i.e. $M_k = M'_k$), alors les traversées *raccourcies* $M_0, M_1, \ldots, M_{k-1}$ et $M'_0, M'_1, \ldots, M'_{k-1}$ sont toutes deux des traversées de P_q, où $q \xrightarrow{M_k} p$.

Parti 2 Les ensembles des traversées possibles pour P_0, P_1, ..., $\mathsf{P}_{|\mathsf{F}|}$ peuvent être obtenus par induction. Nous notons par S_p l'ensemble des traversées possibles pour P_p. Ainsi, nous avons $S_0 = \{P_0\}$, où P_0 est la séquence de messages vide, et nous avons :

$$S_p = \left\{ PM \mid q \xrightarrow{M} p \text{ et } P \in S_q \right\}$$

pour $p > 0$.

Partie 3 Comme nous l'avons déjà mentionné, les ensembles S_p sont trop grands pour être manipulés tels quels. Nous introduisons donc la notion de *sommaire* de S_p, que nous notons par $\mathcal{S}_p$. Un sommaire $\mathcal{S}_p$ est une entité mathématique qui agit comme un *représentant* de S_p. Cette entité a la capacité de produire une traversée pour P_p, en choisissant une parmi plusieurs possibilités, et d'associer un mot de code recyclé à la possibilité choisie.

Partie 4 Pour construire $\mathcal{S}_p$, on partitionne S_p en *groupes*. Un groupe contient les traversées qui ont comme *dernier* message un message particulier $M \in \overleftarrow{\mu}(p)$. Nous appelons M *l'étiquette* du groupe. Notez qu'il y a une correspondance une-à-un entre $\overleftarrow{\mu}(p)$ (i.e. les étiquettes) et les groupes de S_p. Notez aussi que n'importe quelle traversée dans le groupe étiqueté par M a la forme PM, où $P \in S_q$ pour la position q, telle que $q \xrightarrow{M} p$.

Partie 5 Un sommaire $\mathcal{S}_p$ est aussi utilisé comme code pour faire le recyclage de bits. Il donne à chacune des options (les groupes) un mot de code recyclé. Quand une traversée donnée est choisie, les bits associés à son groupe sont recyclés. En fait, une traversée peut être choisie en autant que son groupe (l'option de laquelle elle fait partie) est considéré profitable. La construction du code $\mathcal{S}_p$ consiste en les étapes suivantes : chaque groupe est considéré comme une option, chaque option se voit attribuer un coût (en utilisant une fonction de coût K), et alors $\mathcal{S}_p$ est $R^{\propto opt}(K)$, c'est-à-dire le code pour faire le recyclage de bits. Il reste à déterminer le coût d'une option. Prenons un groupe

```
1. p := 0 ;
2. while description incomplete do
3.     let M_s = ND-select in μ⃗(p) ;
4.     emit(C(M_s)) ;
5.     p := p + L(M_s) ;
6.     for all M ∈ μ⃖(p) do
7.         let K(M) = cost(M) ;
8.     if R^∝opt(K)(M_s) defined then
9.         recycle(R^∝opt(K)(M_s)) ;
10.    else abort ;
where
11. procedure cost(M) :
12.    return E[p − L(M)] + |C(M)| ;
13. procedure emit(w) :
14.    if w = ϵ or ρ = ϵ then
15.        σ := σ · w ;
16.    else if w = b · w' and ρ = b · ρ'
17.    /* where b ∈ {0,1} */ then
18.        ρ := ρ' ;
19.        emit(w') ;
20.    else abort ;
21. procedure recycle(w) :
22.    ρ := w · ρ ;
```

Compresseur

```
1. p := 0 ;
2. while description incomplete do
3.     let M_s = receive() ;
4.     interpret M_s ;
5.     p := p + L(M_s) ;
6.     for all M ∈ μ⃖(p) do
7.         let K(M) = cost(M) ;
8.
9.     recycle(R^∝opt(K)(M_s)) ;
10.
where
11. procedure cost(M) :
12.    return E[p − L(M)] + |C(M)| ;
13. procedure receive() :
14.    let M, σ' s.t. C(M) · σ' = σ ;
15.    σ := σ' ;
16.    return M ;
17.
18.
19.
20.
21. procedure recycle(w) :
22.    σ := w · σ ;
```

Décompresseur

FIGURE 6.5 – Algorithmes pour le recyclage de bits avec tous les messages

dans $\mathcal{S}_p$ étiqueté par $M \in \overleftarrow{\mu}(p)$, où $q \xrightarrow{M} p$. Le coût de l'option étiquetée par M est le coût d'une traversée de P_q plus $|C(M)|$. Cependant, la traversée de P_q est produite par $\mathcal{S}_q$, lequel est lui-même un code pour faire le recyclage de bits, et non une chaîne de bits bien déterminée. Cela veut dire que nous n'avons pas le coût définitif pour la traversée produite par $\mathcal{S}_q$. De plus, puisque nous ne savons pas à l'avance laquelle des options de $\mathcal{S}_p$ sera choisie, nous pouvons seulement obtenir un *coût espéré* pour $\mathcal{S}_p$. Le coût espéré pour chaque sommaire $\mathcal{S}_p$ est calculé inductivement et conservé dans un tableau $\mathcal{E}$. Pour chaque position p, $\mathcal{E}[p]$ est égal au coût espéré de $\mathcal{S}_p$. En fait, nous calculons seulement des *estimés* des coûts espérés avec une précision pouvant aller jusqu'au quart de bit afin de ne pas travailler avec des nombres de précision arbitraire. Ainsi, nous avons $\mathcal{E}[0] = 0$, car $\mathcal{S}_0$ représente une seule traversée vide, et nous avons :

$$\mathcal{E}[p] = \sum_{M \in \mathrm{Dom}(R^{\propto opt}(K))} \big(K(M) - |R^{\propto opt}(K)(M)|\big) \;/\; 2^{|R^{\propto opt}(K)(M)|}$$

pour $p > 0$, où la fonction de coût K est définie comme suit :

$$\forall M \in \overleftarrow{\mu}(p), K(M) = \mathcal{E}[p - \mathrm{L}(M)] + |C(M)|$$

Algorithmes La figure 6.5 présente les algorithmes du compresseur et du décompresseur pour le recyclage de bits avec tous les messages. À la ligne 5, la position courante, p, est maintenant mise à jour explicitement. À la ligne 7, nous utilisons la fonction **cost** pour calculer le coût des options. La fonction **cost**, définie à la ligne 12, utilise la fonction C pour encoder les messages. La fonction C est définie comme suit :

$$\begin{aligned} C([c]) &= C_1([c]) \\ C(\langle l, d\rangle) &= C_1(l)C_2(d) \end{aligned}$$

Comme dans l'expérience 8, les distances sont encodées avec C_2^{full} afin que toutes les distances soient éligibles pour le recyclage. À la ligne 9, la fonction de recyclage proportionnel optimal $R^{\propto opt}$ est utilisée pour faire le recyclage de bits.

Notez comment le compresseur choisit M_s à la position courante (celle *à partir de* laquelle M_s décrit les octets) mais fait le recyclage à la nouvelle position (celle *jusqu'à* laquelle M_s décrit les octets). Nous procédons de cette façon, car le décompresseur n'a accès à aucune information autre que celle contenue dans M_s et ses prédécesseurs. Finalement, le décompresseur doit calculer tous les éléments de $\mathcal{E}$ à la volée, aussitôt que des octets additionnels deviennent connus du décompresseur.

Développement du prototype

Le prototype de l'expérience 9 est très différent des prototypes précédents. Nous n'avons plus besoin des ensembles de plus longues copies. Nous avons plutôt besoin des sommaires $\mathcal{S}_p$ pour chacune des positions dans le fichier à compresser. Pour ce faire, nous avons modifié la première passe du compresseur afin d'y inclure la mécanique nécessaire pour construire les $\mathcal{S}_p$.

Ensuite, nous avons modifié la première sous-passe de la deuxième passe du compresseur afin d'y inclure la mécanique nécessaire pour construire le tableau $\mathcal{E}$ (les coûts espérés des $\mathcal{S}_p$) et de modifier la façon dont le recyclage est fait. Plus précisément, il faut maintenant tenir compte du coût espéré de chaque position p lors du recyclage. De plus, il faut mettre à jour la position p, car les messages que nous choisissons ne sont pas tous équivalents.

Mesures empiriques

Les résultats de l'expérience 9 sont présentés dans la colonne étiquetée *Exp. 9* du tableau 6.2. Notez que le prototype de l'expérience 9 est de 10 à 100 fois plus lent que

gzip. Cependant, plusieurs optimisations pourraient être apportées au prototype afin de le rendre plus rapide.

Analyse des résultats

Les résultats de l'expérience 9 sont excellents. Tous les fichiers, sans exception [2], sont mieux compressés avec le nouveau prototype qu'avec les précédents. Le nouveau prototype profite pleinement des nouvelles possibilités de recyclage. Il est sans aucun doute profitable de faire le recyclage de bits avec tous les messages.

6.7 Travaux futurs

Le recyclage de bits peut être appliqué à des techniques de codage autres que le codage de Huffman. Par exemple, nous pensons que d'appliquer le recyclage de bits au codage arithmétique pourrait être très intéressant, car cela permettrait peut-être d'améliorer considérablement les performances de cette technique déjà fort efficace.

Le recyclage de bits peut aussi être intégré à des algorithmes de compression autres que Deflate. Il suffit d'appliquer le recyclage de bits à la technique de codage utilisée pour encoder ce qui est produit par l'algorithme de compression ; en autant que la technique souffre de la redondance due aux encodages multiples. Par exemple, nous pourrions intégrer le recyclage de bits à l'algorithme de compression LZMA utilisé par l'outil de compression 7-Zip [15].

6.8 Conclusion

Le recyclage de bits permet de réduire considérablement la taille des fichiers compressés. Les 9 expériences que nous avons réalisées le démontrent clairement. Les 9 expériences nous ont aussi permis d'évaluer différents encodages pour encoder les distances et faire le recyclage. Pour encoder les distances, l'encodage C_2^{full} s'est avéré le meilleur choix, car il accroît les possibilités de recyclage en rendant toutes les distances éligibles. Pour faire le recyclage, $R^{\propto opt}$ est sans aucun doute le meilleur encodage. Le

2. Malheureusement, le compresseur de l'expérience 9 a besoin d'être optimisé pour pouvoir compresser le fichier *pic* dans un délai raisonnable.

recyclage proportionnel optimal a pour but de soustraire de l'ensemble des distances éligibles le sous-ensemble des distances qui font chuter les performances du recyclage, car elles coûtent trop cher à encoder pour les bénéfices qu'on en retire. Finalement, les résultats de l'expérience 9 indiquent clairement qu'il est profitable de faire le recyclage de bits avec tous les messages, car nous avons encore plus de possibilités de recyclage. Évidemment, plus nous avons de possibilités de recyclage, plus nous recyclons des bits et meilleur est le taux de compression ; et, ce, malgré le fait que les messages additionnels soient habituellement plus coûteux (les copies plus courtes).

Si on considère uniquement les résultats de l'expérience 9, le recyclage de bits permet une réduction de la taille des fichiers compressés avec **gzip** de 9.2%. Autrement dit, environ 1 bit sur 11 est recyclé. Les bénéfices que l'on retire du recyclage de bits sont donc considérables et ils nous permettent de croire que le recyclage de bits est une contribution importante au domaine de la compression de données sans perte.

Chapitre 7

Méthode des papillons

7.1 Introduction

La méthode des papillons est une nouvelle technique de compression de données basée sur l'énumération des sous-chaînes d'un fichier à compresser.

A priori, étant donné qu'un fichier de longueur n possède un nombre de sous-chaînes dans $O(n^2)$, lesquelles sont d'une longueur dans $O(n)$, il semble que l'énumération des sous-chaînes représente une somme de données dont la taille est dans $O(n^3)$. Toutefois, il n'en est rien lorsqu'on constate que la description des sous-chaînes de longueur l fournit une grande part de l'information nécessaire à la description des sous-chaînes de longueur $l + 1$. Comme la méthode est inspirée de la famille des méthodes PPM (Prediction by Partial Matching), nous croyons qu'elle a le potentiel d'être compétitive avec celles-ci.

7.2 Fonctionnement

L'objectif de la méthode des papillons est de compresser un fichier en procédant à l'énumération de ses sous-chaînes. La tâche principale du compresseur est de transmettre suffisamment d'information au décompresseur afin que ce dernier puisse récupérer l'énumération complète des sous-chaînes du fichier à compresser pour ainsi retrouver la chaîne originale (le contenu du fichier en tant que tel).

TABLE 7.1 – Liste des n rotations triées

00000101
00001010
00010100
00101000
01000001 (chaîne originale)
01010000
10000010
10100000

Sans perte de généralité, nous travaillons au niveau des bits plutôt que des octets. Pour procéder à l'énumération des sous-chaînes, on considère le fichier à compresser comme étant une chaîne de bits *circulaire* de longueur n. Nous obtenons ainsi n rotations de la chaîne de bits. Le tableau 7.1 présente les n rotations triées de la chaîne de bits 01000001 que nous utiliserons comme exemple tout au long de ce chapitre.

Les sous-chaînes sont énumérées en ordre croissant de longueur, c'est-à-dire de la longueur 1 à la longueur n. Une sous-chaîne peut commencer et finir à n'importe quelle position dans la chaîne de bits originale, celle-ci étant considérée comme circulaire. Le tableau 7.2 présente l'énumération complète des sous-chaînes de la chaîne de bits 01000001. Les entrées du tableau sont de la forme $m \times s$, ce qui signifie qu'une sous-chaîne s apparaît m fois dans la chaîne de bits originale (circulaire). Une fois l'énumération complétée et transmise au décompresseur, le compresseur n'a qu'à informer le décompresseur du numéro de la sous-chaîne de longueur n qui est identique à la chaîne de bits originale.

La compression du fichier, c'est-à-dire l'énumération des sous-chaînes, la transmission de l'énumération au décompresseur et l'identification de la chaîne de bits originale, se réalise donc en trois étapes.

7.2.1 Étape 1 : Énumération des sous-chaînes

La première étape consiste à bâtir un arbre contenant l'énumération complète des sous-chaînes du fichier à compresser. Chaque noeud de l'arbre peut avoir deux fils. Une branche représentant le bit 0 lie un noeud à son fils du haut et une branche représentant

TABLE 7.2 – Énumération des sous-chaînes

Longueur	Sous-chaînes			
1	6×0	2×1		
2	4×00	2×01	2×10	
3	3×000	1×001	2×010	1×100
	1×101			
4	2×0000	1×0001	1×0010	1×0100
	1×0101	1×1000	1×1010	
5	1×00000	1×00001	1×00010	1×00101
	1×01000	1×01010	1×10000	1×10100
6	1×000001	1×000010	1×000101	1×001010
	1×010000	1×010100	1×100000	1×101000
7	1×0000010	1×0000101	1×0001010	1×0010100
	1×0100000	1×0101000	1×1000001	1×1010000
8	1×00000101	1×00001010	1×00010100	1×00101000
	1×01000001	1×01010000	1×10000010	1×10100000

FIGURE 7.1 – Arbre des sous-chaînes

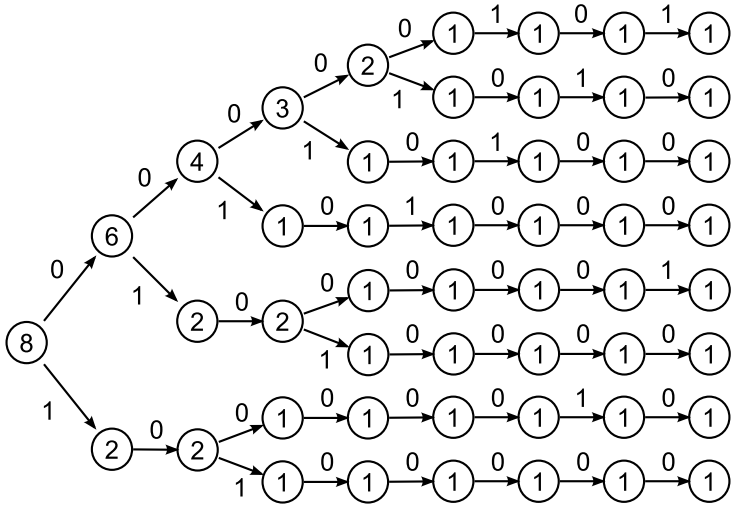
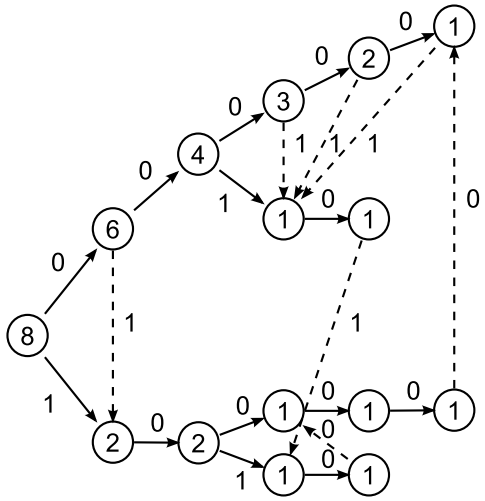


FIGURE 7.2 – Arbre compact des sous-chaînes



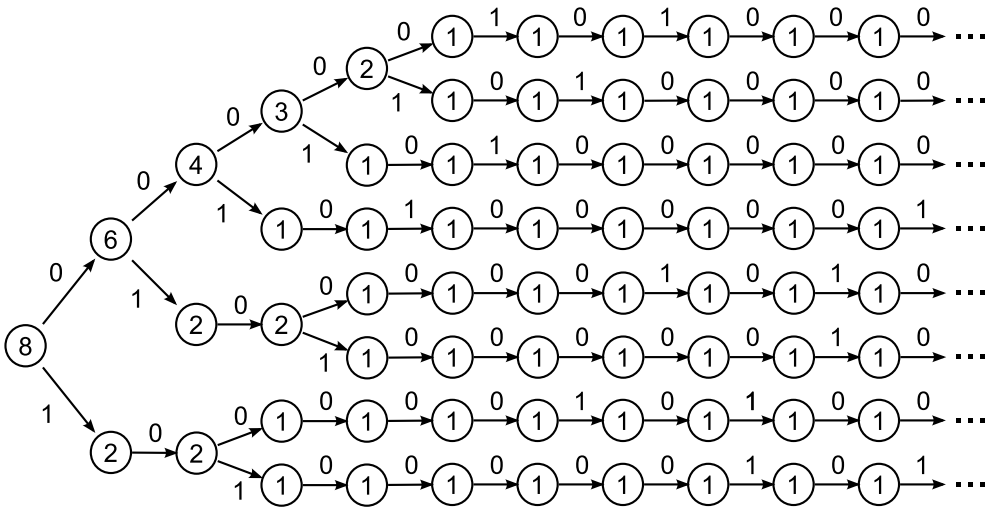
le bit 1 lie un noeud à son fils du bas [1]. Chaque noeud a aussi un compteur. Le compteur est égal au nombre d'apparitions de la sous-chaîne constituée des bits représentés par les branches qui lient la racine au noeud. La longueur de la sous-chaîne est égale à la profondeur du noeud dans l'arbre. La figure 7.1 montre à quoi ressemble l'arbre de l'énumération des sous-chaînes pour la chaîne de bits 01000001. Pour retrouver toutes les sous-chaînes de longueur l, par exemple, il suffit de partir de la racine de l'arbre et de visiter tous les noeuds de profondeur l dans l'arbre.

Cependant, en bâtissant l'arbre de façon naïve, il devient vite gigantesque lorsque n grandit ($O(n^2)$ en pire cas). La figure 7.2 montre qu'il est possible de diminuer énormément la taille de l'arbre. Empiriquement, nous avons déterminé que si la chaîne de bits originale (circulaire) n'est pas de période plus courte que n (ex. de chaînes étant de période plus courte que n : 00000000, 10101010 et 00110011), alors le nombre de noeuds dans l'arbre sera égal à $2n - 1$.

Contrairement au premier arbre, l'arbre *compact* ne contient aucune paire de noeuds isomorphes [2]. Pour bâtir l'arbre compact, il faut *imaginer* l'arbre des sous-chaînes infiniment profond (figure 7.3) contenant l'énumération des sous-chaînes de longueur 1 à longueur ∞. L'arbre compact doit être bâti niveau par niveau. Avant de créer un nouveau noeud, il suffit de regarder s'il n'existerait pas, à un niveau inférieur dans l'arbre, un noeud isomorphe au noeud que l'on s'apprête à créer. Plus précisément, pour savoir s'il existe un noeud isomorphe au noeud pour la sous-chaîne $b\alpha$ (b étant un bit), il

1. Nous parlons de "fils du haut" et de "fils du bas", car nous dessinons nos arbres sur le côté.
2. Cette notion est définie un peu plus loin.

FIGURE 7.3 – Arbre des sous-chaînes infiniment profond



suffit de regarder si le noeud pour la sous-chaîne α a le même compteur. Si les deux noeuds ont le même compteur, alors le noeud pour la sous-chaîne α est isomorphe au noeud pour la sous-chaîne $b\alpha$. Dans l'arbre compact (figure 7.2), une flèche en pointillé représente le lien entre un parent et un noeud isomorphe au noeud que l'on aurait créé dans l'arbre non-compact.

7.2.2 Étape 2 : Transmission de l'énumération au décompresseur

La deuxième étape consiste à transmettre suffisamment d'information au décompresseur afin que ce dernier puisse reconstruire l'arbre de l'énumération des sous-chaînes. Le décompresseur sera alors en mesure de récupérer l'énumération complète des sous-chaînes du fichier à compresser.

La forme d'une sous-chaîne w de longueur l, pour $l \geq 2$, peut se traduire par l'équation $w = b\alpha c$, où b et c sont des bits et α est une sous-chaîne de longueur $l - 2$. Par exemple, pour la sous-chaîne 1010, $b = 1$, $\alpha = 01$ et $c = 0$. De plus, lors de l'énumération de toutes les sous-chaînes de longueur l, nous disposons déjà de l'information sur l'énumération de toutes les sous-chaînes de longueur $l - 1$, car les sous-chaînes sont énumérées en ordre croissant de longueur. La forme d'une sous-chaîne de longueur l, pour $l \geq 1$, peut donc aussi se traduire par l'équation $w = bv = uc$, où b et c sont des bits et v et u sont des sous-chaînes de longueur $l - 1$. En d'autres mots, on

FIGURE 7.4 – Un papillon

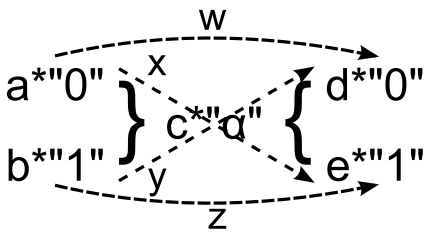


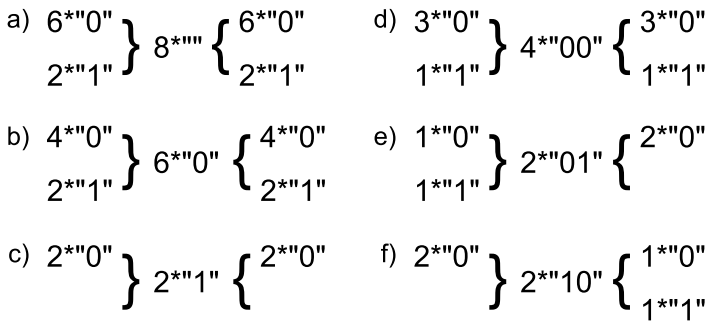
TABLE 7.3 – Paramètres du papillon

Paramètre	Description	Note
a	Nombre de $0\alpha\_$	$w + x = a$
b	Nombre de $1\alpha\_$	$y + z = b$
c	Nombre de $\_\alpha\_$	$a + b = d + e = c$
d	Nombre de $\_\alpha 0$	$w + y = d$
e	Nombre de $\_\alpha 1$	$x + z = e$
w	Nombre de $0\alpha 0$	
x	Nombre de $0\alpha 1$	
y	Nombre de $1\alpha 0$	
z	Nombre de $1\alpha 1$	

peut voir une sous-chaîne de longueur l comme étant une des sous-chaînes de longueur $l-1$ à laquelle on ajoute un bit au début ou comme étant une (autre) des sous-chaînes de longueur $l-1$ à laquelle on ajoute un bit à la fin.

Maintenant, si la sous-chaîne v (de longueur $l-1$) apparaît n_v fois dans la chaîne de bits originale (circulaire), alors le nombre de sous-chaînes de la forme $0v\_$ et $1v\_$ est lié et leur total fait n_v. Aussi, si la sous-chaîne u (de longueur $l-1$) apparaît n_u fois dans la chaîne de bits originale, alors le nombre de sous-chaînes de la forme $\_u0$ et $\_u1$ est lié et leur total fait n_u. Nous avons donc deux façons différentes de justifier l'existence de chaque sous-chaîne de longueur l. Pour profiter de ces deux sources d'information, on considère la sous-chaîne α (de longueur $l-2$) qui apparaît n fois dans la chaîne de bits originale et les 4 combinaisons $b\alpha c$ possibles : $0\alpha 0$, $0\alpha 1$, $1\alpha 0$ et $1\alpha 1$. Évidemment, ces 4 combinaisons sont liées en nombre. Le nombre de sous-chaînes de la forme $0\alpha\_$ et $1\alpha\_$ est lié et leur total fait n. De même, le nombre de sous-chaînes de la forme $\_\alpha 0$ et $\_\alpha 1$ est lié et leur total fait aussi n.

FIGURE 7.5 – Exemples de papillons



En tirant avantage de ces observations, nous utilisons un diagramme appelé papillon. La figure 7.4 illustre un papillon. Un papillon sert à décrire les 4 combinaisons de sous-chaînes (de longueur l) ayant toutes comme partie centrale une sous-chaine α donnée (de longueur $l-2$). Le tableau 7.3 présente les différents paramètres d'un papillon. Le paramètre c est égal au nombre total de sous-chaînes dont la partie centrale est α. Le paramètre c est aussi égal au nombre de fois que l'on retrouve la sous-chaîne α dans la chaîne de bits originale (circulaire). Les paramètres a et b représentent le nombre de sous-chaînes dont la partie centrale est α et qui ont comme préfixe le bit 0 (i.e. qui sont de la forme $0\alpha\_$) et le bit 1 (i.e. qui sont de la forme $1\alpha\_$), respectivement. Les paramètres d et e représentent le nombre de sous-chaînes dont la partie centrale est α et qui ont comme suffixe le bit 0 (i.e. qui sont de la forme $\_\alpha 0$) et le bit 1 (i.e. qui sont de la forme $\_\alpha 1$), respectivement.

La figure 7.5 illustre quelques exemples de papillons pour la chaîne de bits 01000001. Analysons le papillon b en détail. L'information qui est disponible grâce à ce papillon est la suivante :

- Il y a 6 sous-chaînes de longueur 3 dont la partie centrale est le bit 0
- Parmi ces 6 sous-chaînes :
  - 4 sous-chaînes ont comme *premier* bit 0
  - 2 sous-chaînes ont comme *premier* bit 1
  - 4 sous-chaînes ont comme *dernier* bit 0
  - 2 sous-chaînes ont comme *dernier* bit 1

Évidemment, on peut aussi retrouver cette information dans la chaîne de bits originale et dans l'arbre de l'énumération des sous-chaînes. Cependant, comme nous le verrons plus loin, cette information ne nous donne pas nécessairement la valeur des paramètres w, x, y et z. De plus, pour transmettre suffisamment d'information au

décompresseur afin que ce dernier puisse reconstruire l'arbre de l'énumération des sous-chaînes, il faut construire des papillons pour toutes les sous-chaînes de longueur 2 à n. Pour une longueur de sous-chaîne donnée, il y a autant de papillons que de parties centrales α possibles. Ceci étant dit, les papillons sont construits suite à une analyse de l'arbre de l'énumération des sous-chaînes.

Cependant, la construction des papillons n'est que l'étape préliminaire pour communiquer l'information au décompresseur. En fait, ce qu'il faut communiquer au décompresseur, c'est le nombre de sous-chaînes de la forme $0\alpha0$, $0\alpha1$, $1\alpha0$ et $1\alpha1$ pour un α donné. Les paramètres w, x, y et z présentés dans le tableau 7.3 sont utilisés à cette fin.

Il reste donc à voir comment on peut communiquer la valeur des paramètres w, x, y et z au décompresseur pour chacun des papillons construits. En fait, il suffit de lui communiquer uniquement la valeur de w et tous les autres paramètres pourront être déduits automatiquement. En effet, en sachant ce que vaut w, nous savons ce que vaut x, car $w + x = a$, et nous savons ce que vaut y, car $w + y = b$. Finalement, en sachant ce que vaut x, nous savons ce que vaut z, car $x + z = e$.

Pour communiquer la valeur du paramètre w, nous disposons de deux informations : la borne inférieure de la valeur du paramètre w est $max(0, a - e)$ et la borne supérieure est $min(a, d)$, parce que $w \geq 0$, $x \geq 0$, $y \geq 0$, $z \geq 0$ et :

$$x \geq 0 \Rightarrow a - w \geq 0 \Rightarrow w \leq a$$
$$y \geq 0 \Rightarrow d - w \geq 0 \Rightarrow w \leq d$$
$$z \geq 0 \Rightarrow e - x \geq 0 \Rightarrow e - (a - w) \geq 0 \Rightarrow w \geq a - e$$

Reprenons le papillon b de la figure 7.5. Le paramètre w peut prendre une valeur se situant entre $max(0, 4 - 2) = 2$ et $min(4, 4) = 4$. En analysant l'arbre de l'énumération des sous-chaînes, nous savons que $w = 3$ (voir tableau 7.2). Ainsi, grâce à ce papillon, nous pouvons communiquer au décompresseur que nous avons :

- 3 fois la sous-chaîne 000 ($w = 3$)
- 1 fois la sous-chaîne 001 ($x = 1$)
- 1 fois la sous-chaîne 100 ($y = 1$)
- 1 fois la sous-chaîne 101 ($z = 1$)

Finalement, il est très important de noter qu'il n'est pas toujours nécessaire de communiquer la valeur de w au décompresseur. En effet, pour plusieurs papillons, le paramètre w ne peut prendre qu'une seule valeur et donc w est communiqué implicite-

ment au décompresseur (i.e. $min(\ldots) = max(\ldots)$). C'est le cas des papillons c, e et f de la figure 7.5.

7.2.3 Étape 3 : Identification de la chaîne de bits originale

La troisième étape consiste à identifier laquelle des n sous-chaînes de longueur n représente la chaîne de bits originale dans l'arbre de l'énumération des sous-chaînes et à transmettre cette information au décompresseur.

7.3 Développement du prototype

Pour tester empiriquement la méthode des papillons, nous avons développé un compresseur et un décompresseur afin de nous assurer que nos résultats expérimentaux proviennent d'implantations d'algorithmes de compression et de décompression valides. Ceci étant dit, pour la suite du texte, nous parlerons uniquement du compresseur.

La première tâche du compresseur est de faire l'énumération complète des sous-chaînes du fichier à compresser [3]. Plus précisément, le compresseur doit bâtir l'arbre de l'énumération des sous-chaînes. Pour ce faire, nous utilisons d'abord une version modifiée de l'algorithme de tri *bucket sort* afin de trier toutes les sous-chaînes en ordre lexicographique. Pour une longueur de sous-chaîne donnée, l'algorithme ne conserve que les index du début des sous-chaînes. Grâce à l'information obtenue par l'algorithme de tri, nous pouvons ensuite bâtir l'arbre de l'énumération des sous-chaînes. Évidemment, il s'agit de la version compacte de l'arbre. L'arbre est bâti niveau par niveau. Pour chaque niveau, c'est-à-dire pour chaque longueur de sous-chaîne, nous utilisons l'information rendue disponible par l'algorithme de tri pour créer les nouveaux noeuds. Évidemment, avant de créer un nouveau noeud, nous regardons toujours s'il n'existerait pas, à un niveau inférieur, un noeud isomorphe au noeud que nous nous apprêtons à créer.

Une fois l'arbre bâti, le compresseur doit transmettre suffisamment d'information au décompresseur afin que ce dernier puisse récupérer l'énumération complète des sous-chaînes du fichier à compresser, c'est-à-dire l'arbre de l'énumération des sous-chaînes. Pour ce faire, nous transmettons premièrement au décompresseur la taille du fichier à compresser grâce au code Elias gamma (voir Section 4.4.1) et le nombre de fois qu'apparaît le bit 0 dans le fichier à compresser grâce au codage arithmétique (voir

3. Notez qu'à des fins pratiques, le fichier est compressé par blocs de 32K.

TABLE 7.4 – Résultats du prototype

Fichier	Original	Gzip	Prototype
bib	111 261	34 900	35 345
book1	768 771	312 281	301 897
book2	610 856	206 158	209 333
geo	102 400	68 414	77 237
news	377 106	144 400	157 077
obj1	21 504	10 320	13 718
obj2	246 814	81 087	93 359
paper1	53 161	18 543	18 537
paper2	82 199	29 667	28 490
paper3	46 526	18 074	17 172
paper4	13 286	5 534	5 270
paper5	11 954	4 995	4 929
paper6	38 105	13 213	13 093
pic	513 216	52 381	131 751
progc	39 611	13 261	13 649
progl	71 646	16 164	17 005
progp	49 379	11 186	12 258
trans	93 695	18 862	25 337

Section 4.3). Nous faisons ceci afin d'informer le décompresseur du nombre de sous-chaînes de longueur 1 (les sous-chaînes 0 et 1). Ensuite, pour chaque longueur de sous-chaîne (de longueur 2 à longueur n), nous construisons des papillons pour toutes les parties centrales α possibles. Chaque papillon est construit suite à une analyse de l'arbre de l'énumération des sous-chaînes. De plus, pour chaque papillon, il faut transmettre la valeur de w, si nécessaire. Les valeurs des w sont donc transmises les unes à la suite des autres grâce au codage arithmétique.

La dernière tâche du compresseur est d'identifier dans l'arbre de l'énumération des sous-chaînes laquelle des n sous-chaînes de longueur n est identique à la chaîne de bits originale et de transmettre cette information au décompresseur grâce au codage arithmétique.

7.4 Mesures empiriques

Pour vérifier l'efficacité du prototype de la méthode des papillons, nous avons compressé les 18 fichiers du corpus de Calgary (voir Annexe A).

Les résultats sont présentés dans le tableau 7.4. La première colonne contient le nom des fichiers qui ont été compressés, la deuxième colonne contient la taille des fichiers (en octets) avant compression, la troisième colonne contient la taille des fichiers (en octets) compressés par **gzip** (en mode compression maximale) et la quatrième colonne contient la taille des fichiers (en octets) compressés avec le prototype.

7.5 Analyse des résultats

Les résultats obtenus avec le prototype ont répondu à nos attentes. La technique étant toute jeune, nous ne pouvions pas espérer un taux de compression exceptionnel. Néanmoins, les résultats obtenus montrent clairement le potentiel de la technique. Le prototype a fait mieux que **gzip** pour 7 des 18 fichiers compressés, ce qui nous permet de croire qu'en travaillant davantage la technique, elle pourrait être très compétitive. Pour certains types de fichiers comme les fichiers texte (*book1*, *paper1* à *paper6*), la technique est très efficace. Malheureusement, la technique semble être moins efficace avec certains autres types de fichiers comme celui de *pic*. Cependant, il faut se rappeler que **gzip** est justement très performant avec les fichiers comme *pic*, car il peut remplacer une longue suite d'octets par un seul message (une copie). Finalement, le taux de compression moyen pour l'ensemble des fichiers est de 64% et le prototype est 5 fois plus lent que **gzip**. Cependant, plusieurs optimisations pourraient être apportées au prototype afin de le rendre plus rapide.

7.6 Conclusion

La méthode des papillons est une technique de compression de données encore toute jeune. Elle est inspirée de la famille des méthodes PPM et les résultats de notre expérience nous permettent de croire qu'elle a le potentiel d'être compétitive avec celles-ci. Évidemment, certaines modifications pourraient être apportées à la technique afin d'améliorer son taux de compression. Entre autres, nous pensons qu'il y a un moyen plus efficace de transmettre les valeurs des w au décompresseur. Présentement, les différentes

valeurs que peut prendre un w sont considérées équiprobables lors de l'encodage. Nous pourrions donc assigner plus fidèlement les probabilités aux différentes valeurs que peut prendre un w.

Annexe A

Corpus

Le corpus de Calgary [18] est un ensemble de fichiers fréquemment utilisé en compression de données pour comparer les techniques de compression entre elles.

Le corpus est composé de 9 fichiers différents. Cependant, certains fichiers sont du même type pour permettre de vérifier si les performances de certaines techniques sont consistentes. Plus précisément, les fichiers *book1*, *book2*, *paper1*, *paper2*, *paper3*, *paper4*, *paper5* et *paper6* sont des textes écrits en langue anglaise. Le fichier *bib* est une bibliographie en format *bibtex*. Le fichier *news* est composé d'articles provenant d'un groupe *usenet*. Les fichiers *progc*, *progl* et *progp* sont des fichiers sources de programmes. Le fichier *trans* est la transcription d'une session sur un terminal. Les fichiers *obj1* et *obj2* sont du code exécutable. Le fichier *geo* est composé de données géophysiques. Finalement, le fichier *pic* est une image en noir et blanc. Notez que le fichier *geo* est particulièrement difficile à compresser, car il contient une grande étendue de valeurs. Notez aussi que le fichier *pic* est très compressible, car l'image contient plusieurs espaces blancs qui sont représentés par des séries de zéros dans le fichier.

Bibliographie

[1] M. J. Atallah and S. Lonardi. Authentication of LZ-77 compressed data. In *Proceedings of the 18th ACM Symposium on Applied Computing*, pages 282–287, Melbourne, Florida, USA, mar 2003.

[2] A. Brown. `gzip-steg`, 1994.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[4] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4) :396–402, April 1984.

[5] L. Deutsch. Deflate compressed data format specification. `ftp ://ftp.uu.net/pub/archiving/zip/doc/`.

[6] D. Dubé and V. Beaudoin. Recycling bits in LZ77-based compression. In *Proceedings of the Conférence des Sciences Électroniques, Technologies de l'Information et des Télécommunications (SETIT 2005)*, Sousse, Tunisia, mar 2005.

[7] D. Dubé and V. Beaudoin. Improving LZ77 data compression using bit recycling. In *Proceedings of the International Symposium on Information Theory and Applications (ISITA)*, Seoul, South Korea, oct 2006.

[8] D. Dubé and V. Beaudoin. Bit recycling with prefix codes. In *Proc. of DCC*, page 379, Snowbird, Utah, USA, mar 2007. Poster.

[9] J. L. Gailly and M. Adler. The GZIP compressor. `http ://www.gzip.org`.

[10] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, sep 1952.

[11] S. Lonardi and W. Szpankowski. Joint source-channel LZ'77 coding. In *Proceedings of the IEEE Data Compression Conference*, pages 273–282, Snowbird, Utah, USA, mar 2003.

[12] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 scheme and its analysis. In *Proceedings of IEEE International Symposium on Information Theory (ISIT'04)*, page 56, Chicago, Illinois, USA, 2004.

[13] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 data compression : Algorithms, analysis, and experiments. *IEEE Transactions on Information Theory*, 53(5) :1799–1813, may 2007.

[14] A. Moffat, R.M. Neal, and I.H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3) :256–294, jul 1998.

[15] Igor Pavlov. 7-zip file archiver. `http ://www.7-zip.org`.

[16] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding—a survey. *Proceedings of the IEEE*, 87(7) :1062–1078, jul 1999.

[17] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, July, October 1948.

[18] I. Witten, T. Bell, and J. Cleary. The Calgary corpus, 1987. `http ://links.uwaterloo.ca/calgary.corpus.html`.

[19] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6) :520–540, jun 1987.

[20] Y. Wu, S. Lonardi, and W. Szpankowski. Error-resilient LZW data compression. In *Proceedings of the IEEE Data Compression Conference*, pages 193–202, Snowbird, Utah, USA, mar 2006.

[21] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3) :337–342, 1977.

[22] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5) :530–536, 1978.