View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CorpusUL

L’algorithme de Pacman pour la construction efficace des codes équilibrés

Mémoire

Mounir Mechqrane

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

Québec, Canada

© Mounir Mechqrane, 2017

L'algorithme de Pacman pour la construction efficace des codes équilibrés

Mémoire

Mounir Mechqrane

Sous la direction de:

Danny Dubé, directeur de recherche

Résumé

Un bloc de bits est équilibré s'il contient un nombre de bits à 0 égal à celui des bits à 1. Les codes équilibrés (*Balanced Codes*, BC) sont largement appliqués dans plusieurs domaines. Par exemple, ils sont utilisés pour réduire le bruit dans les systèmes VLSI (Tabor, 1990). Dans le domaine de la télécommunication, ils sont utilisés dans la synchronisation et la transmission des données par fibre optique (Bergmann *et al.*, 1986). Leur utilisation dans le domaine de l'identification par radiofréquence (RFID) permet d'augmenter les taux de transfert de données via les canaux RFID (Durgin, 2015). Étant donnée leur importance, plusieurs travaux de recherche ont été menés pour optimiser leur construction.

Knuth fut le premier à trouver une méthode simple et rapide pour l'élaboration des codes équilibrés (Knuth, 1986). Il a introduit un algorithme très simple pour générer les codes équilibrés sans l'utilisation des tables de correspondance. Cependant, cet algorithme ajoute presque le double de la redondance minimale nécessaire pour la création des codes équilibrés. Une partie de cette redondance est due à la multiplicité d'encodage (ME) de cet algorithme. Plusieurs chercheurs ont essayé de réduire la redondance de l'algorithme de Knuth (Immink et Weber, 2009a, 2010; Immink *et al.*, 2011; Al-Rababa'a *et al.*, 2013). Dans les derniers travaux de Al-Rababa'a *et al.* (2013), les auteurs ont réussi à éliminer la redondance créée par ME et pourtant un écart par rapport au seuil minimal subsiste.

Ce travail présente une alternative à l'algorithme de Knuth pour créer les codes équilibrés sans surplus de redondance. Nous proposons un algorithme nommé « Pacman » [1] basé sur les permutations et les nombres entiers à précision limitée. En effet, le processus de codage de cet algorithme peut être assimilé à un Pacman qui consomme et produit des blocs d'informations d'une façon cyclique. Au cours de ce travail, nous allons montrer analytiquement et expérimentalement que la redondance introduite par notre technique est particulièrement faible, que les résultats sont nettement meilleurs que ceux des travaux antérieurs et que les complexités temporelle et spatiale utilisées sont linéaires.

1. Inspiré de la marque de commerce PAC-MAN de l'entreprise BANDAI NAMCO.

Abstract

A block of m bits is said to be balanced if it contains an equal number of zeros and ones. Note that m has to be an even number. Balanced codes (BC) have applications in several domains. For example, they are used to reduce noise in VLSI systems (Tabor, 1990). In telecommunication, they are used in synchronization and data transmission by optical fibers (Bergmann *et al.*, 1986). Their use in the field of radio frequency identification (RFID) can help to boost data transfer rate through RFID channels (Durgin, 2015). Given their importance, several research works have been carried out to optimize their construction.

Knuth was the first to find a simple and fast method to create balanced codes (Knuth, 1986). He introduced a very simple algorithm to generate balanced codes without using lookup tables. However, Knuth's balanced codes incur redundancy that is almost twice the one attested by the lower bound. A part of this redundancy is due to the multiplicity of encoding (ME) of this algorithm. Improvements to the Knuth's algorithm are discussed in several research works (Immink et Weber, 2009a, 2010; Immink *et al.*, 2011; Al-Rababa'a *et al.*, 2013). In the last one (Al-Rababa'a *et al.*, 2013), redundancy created by ME was eliminated and yet there is still some gap that needs to be closed.

This work presents an alternative to Knuth's algorithm for creating balanced codes without unwanted redundancy overhead. We propose an algorithm called "Pacman" [2] that is based on permutations and limited-precision integers. Indeed, the coding process of this algorithm can be assimilated to a special Pacman that consumes *and produces* pills of information in a cyclical manner. In the presented work, we prove analytically and experimentally that our algorithm closes the mentioned redundancy gap while preserving a favorable compromise between calculation speed and memory consumption.

2. Inspired by the trademark PAC-MAN of BANDAI NAMCO.

Table des matières

Résumé iii

Abstract iv

Table des matières v

Liste des tableaux vii

Liste des figures viii

Abréviations x

Remerciements xii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Problématique 2
- 1.3 Méthodologie suivie 5
- 1.4 Organisation du mémoire 8

2 Revue de la littérature générale 9
- 2.1 Théorie de l'information 9
  - 2.1.1 Information propre 9
  - 2.1.2 Entropie 10
  - 2.1.3 Redondance 12
- 2.2 Modélisation et codage 13
  - 2.2.1 Modélisation des données 13
  - 2.2.2 Codage vs compression 15
  - 2.2.3 Types de codage 17
  - 2.2.4 Code et unicité de décodage 19
- 2.3 Permutations 22
- 2.4 Performance de codage 24

3 Revue de la littérature sur les codes équilibrés 27
- 3.1 Les codes équilibrés 27
- 3.2 Travaux de recherche antérieurs 29

4 Méthode 35
- 4.1 Notions et définitions 35

4.1.1 Notation conventionnelle des permutations 35
4.1.2 Notation par index des permutations 35
4.1.3 Permutations et bloc équilibrés 37
4.1.4 Reproduire une grande permutation à partir de petites permutations 39
4.1.5 Pacman 39
4.1.6 Limitation de la mémoire de Pacman 41
4.2 La technique de Pacman 43
4.2.1 Programmation de l'algorithme de Pacman 43
4.2.2 Viabilité d'une programmation 45
4.2.3 Cycle de codage 45
4.2.4 Cycle de décodage 46
4.2.5 Initialisation et terminaison 46
4.2.6 La conception de la programmation 47

5 Résultats **54**
5.1 Mode de redondance minimale 54
5.2 Mode pour les nombres entiers à précision limitée 59
5.3 Calcul théorique des bornes supérieures de la redondance 60

6 Conclusion **63**

Bibliographie **64**

Liste des tableaux

2.1 Codes issus des arbres binaires 21
2.2 Exemple de la notation grand O 26

4.1 Exemple de consommation d'index 41
4.2 Exemple de production d'index 42
4.3 Exemple de production d'index avec redondance 42

5.1 Calcul de la redondance minimale et celle des algorithmes : Knuth, I&W, Al-Rababa'a et al. et Pacman pour x =2, 3, 4 et 5 62

Liste des figures

1.1 Comparaison entre les résultats de l'algorithme de Knuth avec et sans recyclage de bits, les résultats de la technique de Immink et Weber, et la borne théorique I. Ce graphique est tiré des travaux de recherche de Al-Rababa'a *et al.* (2013) . 5
1.2 Cycle de transfert d'énergie ADP/ATP 6
1.3 Cycle de transfert d'information de Pacman 7

2.1 Bloc de transmission en série par UART 13
2.2 Compression sans perte 16
2.3 Compression avec perte 16
2.4 Arbres de codes binaires 21
2.5 Exemples de permutations cycliques 24

3.1 Bloc de 8 bits non équilibré 30
3.2 Préfixe du code 30
3.3 Bloc de 14 bits équilibré 30
3.4 La longueur moyenne du préfixe en fonction de $\log(m)$ des codes équilibrés issues de l'algorithme de Knuth à préfixe VL équilibré. Nous avons représenté la redondance minimale de la construction de Knuth $\log(m)$ et $\lceil \log(m) \rceil$. Le graphique est tiré des travaux de recherche d'Immink et Weber (2010). 32
3.5 Les principales étapes du recyclage de bits pour l'algorithme de Knuth. L'image est tirée des travaux de recherche d'Al-Rababa'a *et al.* (2013). 33
3.6 Comparaison entre les résultats de l'algorithme de Knuth sans et avec recyclage de bits ($HuBR_k$ et $ACBR_k$), les résultats de la technique de Immink et Weber, et la borne théorique 3.3. Le graphique est tiré des travaux de recherche d'Al-Rababa'a *et al.* (2013). 34

4.1 Exemple de la notation par index 36
4.2 L'analogie de la technique Pacman avec le cycle de transport d'énergie par l'ATP/ADP 40
4.3 Le principe de codage illustré en notation par index 44
4.4 Illustration des séquences de codage et de décodage 48

5.1 La variation de Ω_{min} et du rapport $\binom{m}{m/2}/2^q$ pour différentes valeurs de m . . . 55
5.2 L'interface principale de l'application 56
5.3 L'interface de codage de l'application 56
5.4 Fichier source "`fichierSource.txt`" 57
5.5 Démarrage de l'opération de codage 57
5.6 Fichier codé "`fichierCodé.txt`" 58

5.7 Fichier décodé "`fichierDécodé.txt`" 58
5.8 Le tracé de la redondance pour différentes valeur de m, Ω_{min} et σ_0 59
5.9 Tracé des bornes de la redondance pour différentes valeur de m et de Ω 62

Abréviations

GSM Système mondial de communications mobiles "*Global System for Mobile Communications*"
ASCII *American Standard Code for Information Interchange*
TCP Transmission Control Protocol (
ME Multiplicité d'encodage
FF Encodage fixe à fixe
FV Encodage fixe à variable
VF Encodage variable à fixe
VV Encodage variable à variable
LZ77 Encodage Lempel-Ziv 1977
LZ78 Encodage Lempel-Ziv 1978
LZW Encodage Lempel-Ziv-Welch
LZSS Encodage Lempel-Ziv de Storer et Szymanski
BC Code équilibré (*Balanced code*)
ACBR$_K$ *Arithmetic-Coding-based Bit Recycling* pour l'algorithme de Knuth
HuBR$_K$ *Huffman-based Bit Recycling* pour l'algorithme de Knuth

À mes chers parents pour leurs
soutien, amour et sacrifices
durant toutes ces années

Remerciements

J'adresse mes chaleureux remerciements à tous ceux qui ont contribué, de près ou de loin, à la réalisation de ce travail.

Je remercie mon superviseur le professeur Danny Dubé, de m'avoir accueilli dans son équipe, de m'avoir encadré durant mes études de recherche et de m'avoir donné des idées et des directives pertinentes.

Je tiens aussi à exprimer mes remerciements à Ahmad Al-Rababa'a et Fatma Haddad qui ont contribué énormément au bon déroulement de ma maitrise.

Je remercie aussi mes parents et ma famille en général, sans eux je ne serais tout simplement pas là.

Chapitre 1

Introduction

1.1 Motivation

Un bloc de bits est équilibré s'il contient un nombre de bits 0 égal à celui des bits 1. Les codes équilibrés sont largement appliqués dans plusieurs domaines (Knuth, 1986). Par exemple, dans les circuits et les mémoires des systèmes VLSI, les codes équilibrés sont utilisés pour la détection des erreurs unidirectionnelles (Saitoh et Imai, 1991; Piestrak, 1997). Leur utilisation dans les systèmes VLSI permet aussi d'obtenir une réduction du bruit dans les circuits intégrés (Tabor, 1990). Dans les supports de stockage à écriture unique (*Write Only Memory*), ils sont utilisés pour assurer l'intégrité des données, lesquelles peuvent subir une altération à cause d'un 0 qui peut être changé en un 1 (Leiss, 1984). Ils sont utilisés aussi dans les affectations d'état en conception de circuit séquentiel à tolérance de pannes et à sécurité intégrée (Tohma *et al.*, 1971). Dans le domaine de la télécommunication par fibre optique, les codes équilibrés sont utilisés dans la synchronisation et la transmission des données dans un système de communication à fibre optique et dans l'identification de codage dans les réseaux de fibres optiques (Takasaki *et al.*, 1976; Bergmann *et al.*, 1986; Ofek, 1990; Widmer et Franaszek, 1983). Ils sont aussi utilisés pour établir des communications insensibles au retard dans des systèmes asynchrones (Blaum et Bruck, 1993). Dans le domaine de l'enregistrement optique et magnétique, on trouve leur utilisation dans la conception de circuits à tolérance de pannes et dans la détection d'erreurs dues aux perturbations de basse fréquence dans les supports magnétiques (Immink, 1989; Karabed et Siegel, 1991; Roth *et al.*, 1994). Dans le domaine de l'identification par radiofréquence (RFID), l'utilisation de ces codes permet d'augmenter les taux de transfert de données via les canaux RFID d'au moins 50% sans sacrifier la consommation d'énergie, la simplicité ou les conceptions matérielles radiofréquence actuelles. Avec la troisième révolution du web et l'apparition de l'internet des objets (IoT), on observe l'émergence d'applications qui nécessitent de plus en plus de capteurs connectés et de communications M2M (*Machine to Machine*). Ainsi, ces gains de débit deviennent très importants pour satisfaire à l'augmentation du besoin dû à ce genre d'applications (Durgin, 2015). Ces exemples que nous avons

cités dénotent de l'importance de de la création de codes équilibrés efficaces.

Plusieurs recherches ont été menées dans l'optique de concevoir des algorithmes simples, rapides et efficaces pour la création des codes équilibrés. Knuth (1986) fut le premier à trouver une méthode simple et rapide pour l'élaboration des codes équilibrés. Il a introduit un algorithme très simple pour générer les codes équilibrés sans l'utilisation des tables de correspondance. Cependant, cet algorithme ajoute presque le double de la redondance minimale nécessaire pour la création des codes équilibrés. Une partie de cette redondance est due à la multiplicité d'encodage (ME) de cet algorithme. La multiplicité d'encodage est le fait d'avoir plusieurs encodages possibles pour le même bloc source. Plusieurs chercheurs ont essayé de réduire cette redondance en modifiant l'algorithme de Knuth (Immink et Weber, 2009a, 2010; Immink *et al.*, 2011; Al-Rababa'a *et al.*, 2013). Dans les derniers travaux de Al-Rababa'a *et al.* (2013), la redondance créée par ME a été éliminée et pourtant un écart par rapport au seuil minimal subsiste. Ainsi, la méthode de Knuth semble atteindre ses limites. L'objectif du présent travail est de concevoir une nouvelle technique, simple, rapide et efficace en redondance, en se basant sur les permutations et le personnage du jeu vidéo PC-MAN (Slepian, 1965; Gabrys et Milenkovic, 2016).

1.2 Problématique

En théorie de l'information, un code est un dictionnaire qui contient les mots utilisés pour coder les entrées d'une source d'information. Les mots de ce dictionnaire sont appelés des mots de code. Les entrées de la source peuvent être des symboles ou des chaînes de symboles d'un alphabet donné. Un codage est donc le fait d'associer un mot de code spécifique à chaque entrée de la source. Par exemple, un codage des chiffres décimaux 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9 en code binaire à 4 bits serait respectivement 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000 et 1001.

En binaire, un code équilibré est un code dont chacun des mots de code comporte un nombre de bits à 1 égal au nombres de bits à 0. Par exemple, l'ensemble des mots binaires $\{0011, 0101, 0110, 1001, 1010, 1100\}$ est un code équilibré de longueur 4. Ce code est équilibré puisque chaque mot binaire comporte deux bits à 1 et deux bits à 0. Les codes équilibrés sont des codes *non ordonnés* (Al-Bassam et Bose, 1990). C'est-à-dire que dans un code équilibré, aucun mot de code n'est contenu dans un autre. Ainsi, le code $\{0101, 10, 0011, 1100\}$ n'est pas un code non ordonné vu que le mot de code « 10 » est contenue dans « 1**10**0 ».

Par nature, la création des codes équilibrés nécessite l'ajout de redondance. Par exemple, pour équilibrer l'ensemble des mots binaires $\{00, 01, 10, 11\}$ de taille 2, nous devons ajouter au moins deux bits redondants à chaque mot. Un code équilibré pour cet ensemble serait donc $\{0011, 0110, 1001, 1100\}$. Nous pouvons aussi utiliser le code $\{001101, 011010, 100110, 110001\}$

de taille 6 pour coder l'ensemble des mots binaires de taille 2. Cela va donner une redondance de 4 bits. Pour des raisons d'efficacité, nous souhaitons créer des codes équilibrés ayant le moins de redondance possible.

Mathématiquement, la création de codes équilibrés efficaces est simple. D'après le lemme de Sperner (Sperner, 1928), pour un ensemble de mots d'entrée de taille n, la meilleure façon de créer des codes équilibrés consiste à construire la liste de tous les mots de codes de taille $m = n + p$, où p est le nombre de bits de parité minimale (le minimum de bits redondants à ajouter, le seuil théorique) et $p \approx \frac{1}{2} \log m$ (Knuth, 1986). Par la suite, la correspondance un à un entre les mots d'entrée et les mots de code est établie par ordre lexicographique. Pratiquement, cela est fait en utilisant des tables de consultation qui contiennent les mots d'entrée et les mots de code correspondants (Al-Bassam et Bose, 1994). La redondance dans ce cas est minimale. Cependant, pour encoder des mots de taille n en utilisant cette méthode, nous avons besoin d'au moins 2^n mots de code différents de taille m. Ainsi, nous avons besoin d'une table de correspondance qui utilise un espace de stockage de l'ordre de $\mathrm{O}(2^n)$ mots de taille m. Cet espace de stockage devient énorme quand la taille n des mots à encoder devient très grande. Par exemple, des blocs d'entrée qui sont seulement de taille $n = 512$ (1/16 d'un kilo octet) vont nécessiter une table de consultation qui a beaucoup plus d'entrées qu'il y a d'atomes dans l'univers [3] :

$$2^{\frac{1}{16} \times 1024 \times 8} = 2^{512} > 2^{326} = 2^{2+4 \times 81} = 4 \times 16^{81} > 4 \times 10^{81}. \quad (1.1)$$

L'autre méthode de codage utilisée est le codage énumératif (Cover, 1973). Dans cette méthode nous calculons la position du mot à coder par rapport à l'ensemble des mots d'entrée en ordre lexicographique. Nous utilisons cette position par la suite pour trouver le mot de code correspondant. Autrement dit, nous essayons de trouver le mot de code qui se trouve dans la même position si nous classifions l'ensemble des mots de code en ordre lexicographique. La consommation des ressources mémoires dans ce cas est raisonnable étant donné qu'il n'y a pas d'utilisation de tables de consultation. Par contre, les calculs réalisés pour trouver les mots de codes sont considérés trop coûteux. Ceci est dû à la manipulation de grands entiers. En effet, en codage énumératif, les entiers manipulés ont un encodage d'une longueur proportionnelle à la taille des données traitées. Ceci entraîne des calculs devant se faire en temps supra-linéaire, ce qui est considéré trop coûteux dans la plupart des applications.

Il est nécessaire donc de trouver une technique simple pour créer des codes équilibrés dans laquelle l'utilisation des tables de consultations est éliminée ou limitée et la consommation des ressources (temps, espace mémoire et taille des registres) est raisonnable.

L'idée la plus simple pour l'élimination de tables de consultation est la création des mots de code équilibrés w_r en ajoutant le complément w' du mot à coder w de façon à ce que

3. Source : https://www.cs.mcgill.ca/~rwest/link-suggestion/wpcd_2008-09_augmented/wp/o/Observable_universe.htm.

$w_r = ww'$ (Knuth, 1986). La taille des mots de code est ainsi égale à $2 \times n$, ce qui sûrement diminue l'efficacité du codage. Il est nécessaire donc de viser une méthodologie qui fait le compromis entre la consommation des ressources et l'efficacité de codage. Il faut aussi prendre en considération la simplicité de la méthode.

Knuth a proposé un algorithme simple et rapide qui utilise des tables de consultations qui ne consomment pas beaucoup de ressources. Sa méthode consiste à coder les mots w en blocs équilibrés uw_k. Le fragment w_k, représente la transformation de w après avoir complémenté ses k premiers bits, où k est choisi de telle façon que w_k ait un nombre de bits à 1 égal au nombre de bits à 0. Le préfixe u est un bloc équilibré qui contient le codage du nombre k. La taille de u est q, $p \leq q < n$ (Knuth, 1986). Par contre, en termes de redondance, l'algorithme de Knuth n'est pas efficace ; le nombre de bits de parité ajoutés est $q \approx \log m$. Ce qui est presque le double de la parité minimale $p \approx \frac{1}{2} \log m$ (Immink et Weber, 2009a, 2010; Immink *et al.*, 2011). Ceci est dû à la liberté de sélection créée par la multiplicité d'encodage (ME) de cet algorithme. La multiplicité d'encodage est le fait d'avoir plusieurs encodages possibles pour le même bloc source. Par exemple, pour coder le mot de bits « 110101 » en bloc équilibré, nous pouvons utiliser le mot de code « $u_1 \cdot 010101$ » en complémentant un bit, le mot de code « $u_3 \cdot 001101$ » en complémentant 3 bits ou le mot de code « $u_5 \cdot 001011$ » en complémentant 5 bits. Le préfixe u_k est un bloc équilibré que nous utilisons pour coder le nombre de bits complémentés k. Nous avons donc le choix entre plusieurs cas possibles. Autrement dit, nous avons une liberté de sélection due à ME. Pour préciser le codage choisi, nous avons besoin de transmettre plus d'information. Ainsi, la taille du préfixe u_k va être plus grande. Pour Knuth l'encodeur et le décodeur s'accordent pour utiliser le premier k trouvé. Dans la section 3.2, nous expliquons plus longuement ce problème de ME.

La simplicité et l'efficacité de cet algorithme ont attiré plusieurs chercheurs tels que Alon *et al.* (1988); Al-Bassam et Bose (1990); Tallini et Bose (1999); Youn et Bose (2003); Mascella et Tallini (2005, 2006); Medvedeva et Ryabko (2010) et Skachek et Immink (2014) qui ont essayé de réduire sa redondance par rapport au seuil théorique. Immink et Weber ont réussi à diminuer cette redondance en appliquant plusieurs approches (Immink et Weber, 2010). Toutefois, leurs modifications ont complexifié l'algorithme. Al-Rababa'a et al. ont appliqué le recyclage des bits sur l'algorithme de Knuth pour réduire sa redondance. Leurs modifications n'ont pas beaucoup influencé la simplicité et la rapidité de l'algorithme. Cependant, leurs résultats sont plus proches de l'optimum. En effet, ils ont réussis à éliminer la totalité de la redondance créée par ME. Le graphique à la figure 1.1 est une comparaison qui illustre bien les résultats de ces travaux de recherche. L'exploitation de ME a diminué la redondance de l'algorithme de Knuth. À ce point-là, les améliorations ne sont plus bénéfiques et pourtant un écart subsiste toujours par rapport au seuil théorique. Dans les travaux de recherche présentés dans ce mémoire, nous exploitons de nouvelles idées pour résoudre ce problème.

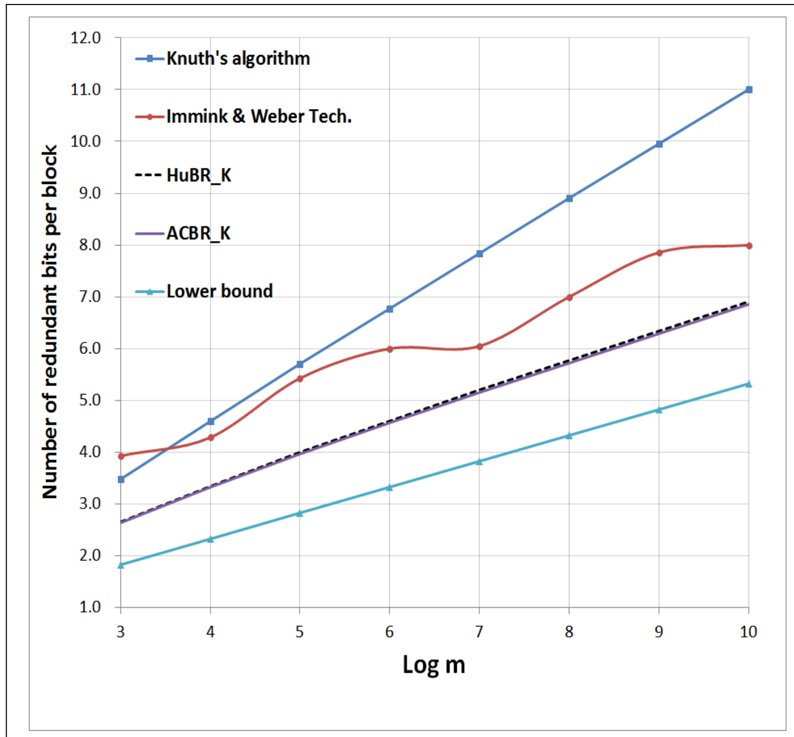

FIGURE 1.1 – Comparaison entre les résultats de l'algorithme de Knuth avec et sans recyclage de bits, les résultats de la technique de Immink et Weber, et la borne théorique I. Ce graphique est tiré des travaux de recherche de Al-Rababa'a *et al.* (2013)

1.3 Méthodologie suivie

La nature a été depuis toujours une source d'inspiration pour les hommes. De la construction des bâtiments et des ponts à la pratique de la médecine et de la manipulation des matériaux, la diversité et la complexité des conceptions de la nature ont toujours influencé la pensée humaine et permis d'aboutir à de telles réalisations. La chose commune entre l'énergie nucléaire, les avions, le téléphone et les antibiotiques est que ceux-ci sont tous des inventions inspirées de la nature.

Le métabolisme est l'une des caractéristiques qui diffèrent les êtres vivants des êtres non vivants. En effet, cette fonctionnalité est intrinsèque à la vie d'un être. C'est un ensemble de réactions chimiques qui permet aux corps vivants de s'entretenir, de croître et de se reproduire. Ces réactions chimiques sont regroupées en ce qu'on appelle des voies métaboliques. Chaque voie métabolique permet de réaliser une fonction biologique particulière. Par exemple, le cycle ATP/ADP est une voie métabolique qui permet de transférer l'énergie dans les cellules (voir figure 1.2). L'ATP (adénosine triphosphate) est une coenzyme qui catalyse les réactions chimiques dans les voies métaboliques. En d'autres termes, c'est une unité moléculaire qui transporte l'énergie nécessaire, à partir de la banque d'énergie de l'être vivant (ex. le foie

pour les mammifères), aux cellules pour réaliser les fonctionnalités biologiques vitales. Dans le cycle ATP/ADP, la phosphorylation oxydative permet de transformer les supports ADP (adénosine diphosphate), en ajoutant de l'énergie issue des autres voies métaboliques comme le catabolisme, en des unités de transport ATP. Par la suite, les unités ATP vont fournir l'énergie nécessaire aux réactions chimiques réalisées au niveau de la cellule. L'utilisation de l'énergie contenue dans l'ATP libère les supports ADP. Ces derniers peuvent être à nouveau utilisés pour créer d'autres unités de transport énergétique ATP et ainsi de suite.

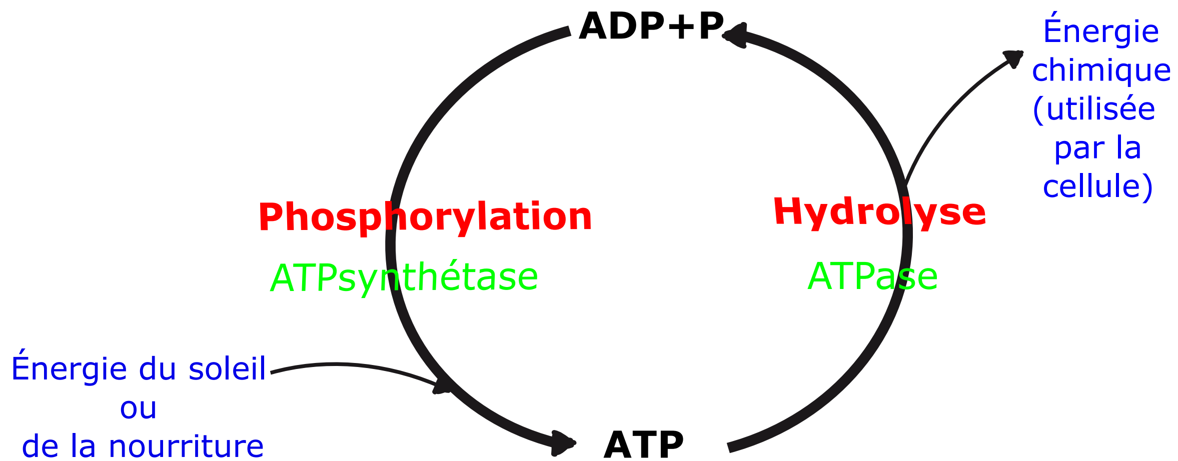

FIGURE 1.2 – Cycle de transfert d'énergie ADP/ATP

De la même manière, nous avons besoin dans nos recherches de transporter l'information source (mots binaires non contraints) sous une forme convenable (mots de code équilibrés efficaces) pour l'utiliser dans diverses applications. Nous choisissons comme coenzyme les permutations de taille $m \in 2\mathbb{N}$. Certes, le transport d'énergie est une fonctionnalité vitale. Elle permet aux êtres vivants de transformer les nutriments consommés en énergie indispensable dans toutes les activités. Cependant, cette fonctionnalité n'est qu'une partie d'un système plus complexe qui est l'être vivant. L'algorithme proposé peut être considéré comme un être dans lequel se font les transformations d'énergie. Pour bien illustrer cela, nous avons appelé notre algorithme *Pacman*. En effet, notre algorithme est comme un être qui consomme des nutriments (données sources) et, après digestion, libère de l'énergie (codes équilibrés) (voir la figure 1.3).

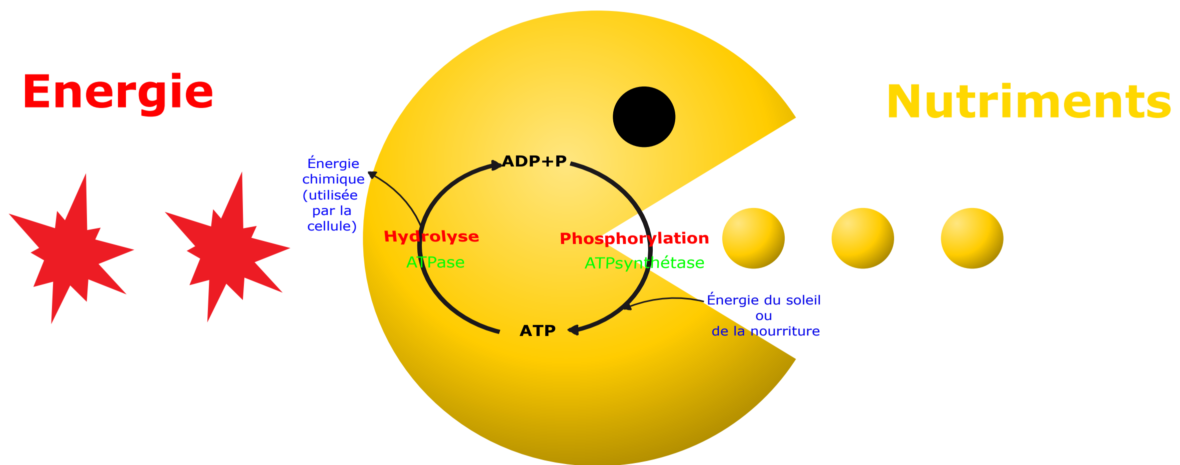

Figure 1.3 – Cycle de transfert d'information de Pacman

Dans ces travaux de recherche, nous montrons pratiquement et théoriquement que l'algorithme proposé réalise un compromis entre l'utilisation des ressources computationnelles et l'efficacité de codage. Pour évaluer la contribution apportée par l'algorithme de Pacman, une comparaison des résultats avec les travaux antérieurs est faite.

1.4 Organisation du mémoire

Le présent travail est organisé en six chapitres, incluant l'introduction et la conclusion. Le chapitre 2 contient une revue de littérature générale dans laquelle nous présentons les notions importantes pour la compréhension de la problématique et de l'algorithme proposé. Le chapitre 3 contient une revue de littérature spécifique sur les codes équilibrés et les différentes techniques de construction réalisées dans les travaux antérieurs. Le chapitre 4 traite en détails des différents aspects de la méthodologie proposée dans le présent travail. Les résultats pratiques et théoriques vont être présentés dans le chapitre 5. Ce dernier chapitre est conclu par une comparaison empirique avec les réalisations antérieures qui montre la contribution apportée.

Chapitre 2

Revue de la littérature générale

Ce chapitre est divisé en quatre sections[4]. Dans la section 2.1, nous présentons un aperçu général sur le domaine de la théorie de l'information. La section 2.2 est nécessaire pour une bonne compréhension de la problématique. Elle explique les notions importantes sur le codage et la modélisation d'une façon générale. La section 2.3 contient des notions importantes sur les permutations que nous utilisons dans ces travaux de recherche. La section 2.4 énonce les différents aspects qui permettent de mesurer les performances d'un algorithme.

2.1 Théorie de l'information

La théorie de l'information est la science qui s'intéresse à la quantification de l'information. « Ce ne sont que des faits, mais des faits quantitatifs et des bases de la science » proclamée par Cattell et Farrand (1896). Dans les sciences naturelles, tout ce qui est capté par les sens de l'homme est comptable, calculable et modélisable. Ceci est valable aussi pour l'information. L'information est tout message communiqué par des symboles spécifiques. Ces symboles peuvent être un alphabet de lettres, des bits, des idéogrammes ou des pictogrammes. La chose commune entre ces symboles c'est qu'ils sont tous comptables. C'est tout ce qui est nécessaire pour quantifier l'information. La théorie de l'information a été développée par Claude Elwood Shannon (1949a) dans les laboratoires de Bell.

2.1.1 Information propre

Un message est toujours produit par une source d'information et communiqué vers un récepteur. À titre d'exemple, nous pouvons considérer qu'Alice est notre source d'information qui veut informer Benoît sur la théorie de l'information. Alice va lui envoyer le message suivant : « Dans la théorie de l'information on quantifie l'information ». Ce message est constitué des mots suivants en ordre alphabétique : d' ; dans ; information ; l' ; la ; on ; quantifie ; théorie. Chaque mot est une chaîne de symboles qui a une fréquence d'occurrence : d' (une fois) ; dans

4. Inspiré du cours de compression de données IFT-7023 ULaval.

(une fois) ; information (deux fois) ; l' (une fois) ; la (une fois) ; on (une fois) ; quantifie (une fois) ; théorie (une fois). Ainsi, nous pouvons donc quantifier le message mathématiquement et évaluer les caractéristiques statistiques de chacune de ses chaînes de symboles. En d'autres termes, nous sommes capable d'extraire les informations mathématiques intrinsèques à chaque élément du message. Nous appelons ces informations « « information propre ».

Pour un symbole A de probabilité $P(A)$ nous définissons l'information propre par (Shannon (2001)) :

$$i(A) = \log_b \left(\frac{1}{P(A)} \right) = -\log_b \left(P(A) \right). \tag{2.1}$$

Si un symbole ou une chaîne de symboles d'entrée qui est peu probable apparait, nous allons recevoir beaucoup d'information et vice versa. Par exemple, dans la langue anglaise, l'apparition de l'article « the », qui est très fréquent, ne dit pas grand-chose étant donné que son information propre est faible. Par contre, le mot « quantification » qui est moins fréquent donne plus d'information sur le contexte du message. Si, en plus, nous avons le mot « quantification » qui vient après le mot « information », nous allons avoir une idée plus claire sur le contenu du message. Nous parlons ici d'information propre mutuelle. Si b est égal à 2, les unités de $i(A)$ sont des bits, si b est égal à e, les unités sont des nats, et si b est égal à 10, les unités sont des hartleys. Tous les logarithmes de ce mémoire sont à base 2.

2.1.2 Entropie

Une des nombreuses contributions de Shannon était de montrer que s'il y avait une source de données S qui émet des symboles A_i d'un ensemble A, alors l'entropie est le nombre moyen de symboles binaires nécessaires pour représenter la source. Considérons la source générale S définie sur l'alphabet $A = \{1, 2, ..., n\}$ qui génère la suite d'événements suivante X = {X1, X2, ..., Xn}, où $X_i \in A$ pour $1 \leq i \leq n$. Si chaque élément de X est indépendant et identiquement distribué, l'entropie de S est (Shannon, 1949b) :

$$H(S) = -\sum_{i=1}^{n} P(i) \log P(i). \tag{2.2}$$

Dans la réalité, les événements d'apparitions de symboles ne sont pas toujours indépendants et identiquement distribués. En effet, il y a plusieurs facteurs qui influencent l'apparition des symboles. Par exemple dans un texte français, l'apparition des lettres dépend de la structure de la langue française. En conséquence, la valeur de l'entropie de la source dépend aussi de ces facteurs. Ainsi, l'expression en haut, ne représente pas tout à fait la vraie entropie.

Par exemple, supposons que, dans la séquence d'entiers suivante, la fréquence d'occurrence de chaque entier est représentée avec précision par le nombre de fois de son apparition :

$$1\ 1\ 2\ 4\ 4\ 5\ 5\ 5\ 6\ 7\ 8\ 8\ 8\ 9.$$

Nous pouvons donc estimer la probabilité d'occurrence de chaque nombre comme suit :

$P(1) = P(4) = \frac{2}{14}$
$P(2) = P(6) = P(7) = P(9) = \frac{1}{14}$
$P(5) = P(8) = \frac{3}{14}$.

L'entropie est :

$$H(S) = -\sum_{i=1}^{n} P(i) \log P(i) = -2\frac{2}{14}\ \log\frac{2}{14} - 4\frac{1}{14}\log\frac{1}{14} - 2\frac{3}{14}\log\frac{3}{14} = 3.64 \text{ bits.} \tag{2.3}$$

Le meilleur schéma de compression sans pertes pour représenter cette séquence doit utiliser en moyenne 3.64 bits/échantillon (Shannon, 1949b).

Si nous prenons en considération la corrélation entre les échantillons consécutifs, nous pouvons coder la différence entre les symboles voisins et la source devient :

$$1\ 1\ 2\ 4\ 4\ 5\ 5\ 5\ 6\ 7\ 8\ 8\ 8\ 9 => 1\ 0\ 1\ 2\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1,$$

$$\text{où } P(0) = 6/14,\ P(1) = 7/14 \text{ et } P(2) = 1/14.$$

Donc

$$H(S) = -\sum_{i=1}^{n} P(i) \log P(i) = -\frac{6}{14}\log\frac{6}{14} - \frac{7}{14}\log\frac{7}{14} - \frac{1}{14}\log\frac{1}{14} = 1.29 \text{ bits.} \tag{2.4}$$

Nous avons pu donc améliorer la valeur moyenne du nombre de symboles binaires nécessaires pour représenter la source pour qu'elle soit plus proche de l'entropie réelle en utilisant la corrélation entre les symboles.

Nous pouvons donc déduire que :

- l'entropie dépend des probabilités et des informations propres des symboles de la source. Ainsi, il est donc nécessaire d'avoir des probabilités qui représentent les vraies caractéristiques statistiques des symboles.
- si les probabilités sont déduites à partir de la séquence d'évènements générés par la source, il est donc nécessaire d'avoir une séquence très grande pour approcher l'entropie réelle qui représente la vraie structure de la source. Pour avoir l'entropie réelle, la séquence doit être infinie.
- l'entropie est la limite théorique de compression sans perte qu'on peut atteindre (Shannon (1949b)).
- quand la valeur moyenne du nombre de symboles nécessaires pour représenter la source est supérieure à l'entropie, nous disons qu'il y a de la redondance dans l'information.

2.1.3 Redondance

En théorie de l'information, la redondance correspond au nombre de bits nécessaires pour transmettre un message auquel on soustrait le nombre de bits correspondant aux informations réellement contenues dans ce même message. Officieusement, la redondance correspond à l'« espace » utilisé mais non occupé pour transmettre certaines données . Mathématiquement, c'est la différence entre le taux absolu de la source et l'entropie moyenne par symbole de cette source (MacKay, 2003). L'entropie moyenne par symbole d'une source S, définie comme dans la sous-section précédente, est :

$$H_n = \lim_{n \to \infty} \frac{1}{n} H(X_1, X_2, ..., X_n). \tag{2.5}$$

Cette valeur converge vers l'entropie de la source $H(S)$ si les symboles de la source S sont indépendants et distribués d'une manière identique.

Le taux absolu d'une source (entropie de Hartley) est le taux d'information maximal, dans notre cas, le nombre de bits maximal, que nous pouvons transmettre en utilisant les symboles de cette source. Sa valeur est calculée comme suit :

$$H_a = \log |\mathbb{X}|, \quad \text{où } |\mathbb{X}| \text{ est la cardinalité de l'espace message.} \tag{2.6}$$

La valeur de la redondance est donc :

$$R = H_a - H_n. \tag{2.7}$$

En codage binaire, la redondance est le nombre de bits de parité ajoutés aux bits sources pour avoir le code voulu. Par exemple, dans notre cas, à partir de n bits sources nous voulons avoir un code équilibré de taille $m \geq n$. Nous devons donc ajouter p bits de parité tel que $m = n+p$, où $0 \leq p < n$ et $\log_2 \binom{m}{\lfloor \frac{m}{2} \rfloor} \geq 2^n$. $\binom{m}{\lfloor \frac{m}{2} \rfloor}$ est la cardinalité de l'ensemble des codes équilibrés. La redondance donc est (Immink et Weber, 2010) :

$$p = m - \log_2 \binom{m}{\lfloor \frac{m}{2} \rfloor}. \tag{2.8}$$

En effet, il y a deux types de redondance. La redondance absolue représentée ici par p qui est le nombre de bits redondants, et la redondance relative qui est le rapport de p par la taille m des mots de code.

En codage, l'ajout de bits redondants peut être utilisé pour la détection et la correction des erreurs. Par exemple, dans une transmission de données en série, un bit de parité est ajouté aux bits sources pour détecter une éventuelle altération des données en vérifiant la parité de la somme des bits (voir la figure 2.1).

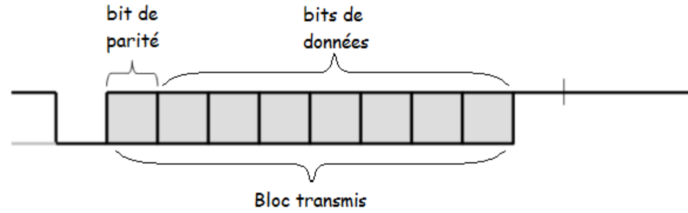

Figure 2.1 – Bloc de transmission en série par UART

2.2 Modélisation et codage

2.2.1 Modélisation des données

La modélisation des données est l'analyse et la conception de l'information. Il s'agit essentiellement d'identifier les entités logiques et les dépendances logiques entre ces entités [5]. En d'autres termes, elle permet d'avoir une idée plus claire sur les vraies caractéristiques statistiques des symboles.

Dans la sous-section 2.1.2 nous avons vu dans l'exemple cité qu'en prenant en considération la corrélation entre les échantillons adjacents, l'entropie s'est améliorée par 75 %.

Passons maintenant à un autre exemple. Considérons la séquence suivante :

$$1\ 5\ 1\ 5\ 6\ 6\ 6\ 6\ 1\ 5\ 6\ 6\ 6\ 6\ 1\ 5\ 6\ 6\ 1\ 5. \tag{2.9}$$

Supposons que la séquence représente bien les probabilités de chaque élément. Les probabilités sont égales à :

$$P(1) = P(5) = \frac{1}{4},\ P(6) = \frac{1}{2}. \tag{2.10}$$

L'entropie est donc égale à : 1,5 bit/symbole.

Considérons maintenant la modélisation structurelle suivante de la séquence. Au lieu d'utiliser les symboles (entiers) 1, 5 et 6 nous allons utiliser les deux chaînes de symboles 15 et 66.

Les probabilités des deux chaînes de symboles sont :

$$P(15) = P(66) = \frac{1}{2}. \tag{2.11}$$

L'entropie devient : 1 bit/paire de symboles.

5. Source : La page de Wikipedia "Modélisation des données"

En effet, la bonne modélisation des données peut mener à la réduction de l'estimation de l'entropie de la source. Dans la compression de données, cela peut aider à améliorer le taux de compression. En codage, nous sommes capable de concevoir des algorithmes plus efficaces avec un modèle de données pertinent. Ainsi, il est nécessaire de choisir le bon modèle pour chaque source d'information. Pour standardiser la modélisation, plusieurs approches ont été adoptées.

- Modèle physique : ce modèle est utilisé dans le traitement et la simulation numérique de sources d'information physiques comme la voix et les instruments de la musique. Il nécessite une connaissance préalable du processus de génération des données par ces sources d'informations. Parfois, la complexité des systèmes physiques des sources d'informations empêche d'avoir des modèles physiques concrets. Ainsi, l'obtention d'un modèle se fait en se basant sur les résultats des observations empiriques des statistiques de la source traitée.
- Modèle d'ignorance : utilisé quand nous n'avons aucune idée sur les dépendances et les probabilités des symboles. Nous considérons donc qu'ils sont indépendants entre eux et qu'ils ont les mêmes probabilités.
- Modèle de probabilité : utilisé quand nous avons une connaissance des probabilités des symboles, mais pas des dépendances. Nous considèrons donc qu'ils sont indépendants.
- Modèle de Markov : c'est un modèle mathématique aléatoire conçu par le mathématicien russe Andrey Markov (Norris, 1998). Son principe repose sur le fait que le futur est indépendant du passé, compte tenu du présent. Mathématiquement, la connaissance des dernières k-apparitions des symboles est équivalant à connaitre l'ensemble de l'historique de toutes les apparitions des symboles. C'est-à-dire, si nous connaissons les k-dernières apparitions, nous n'avons pas besoins de savoir le reste pour prévoir la prochaine apparition. Ainsi, si une séquence d'apparition de symbole $\{x_n\}$ vérifie ce principe, nous disons que celle-ci forme une chaîne de Markov discrète d'ordre k.

  La formule de probabilité dans ce cas est :

  $$P(x_n|x_{n-1}, ..., x_{n-k}) = P(x_n|x_{n-1}, ..., x_{n-k}, ..., x_1). \tag{2.12}$$

  Le modèle de Markov le plus utilisé est celui du premier ordre :

  $$P(x_n|x_{n-1}) = P(x_n|x_{n-1}, ..., x_1). \tag{2.13}$$

  Le modèle de Markov trouve plusieurs applications dans des domaines ou les données sont dynamiques (temporaires). À titre d'exemples :

  - dans la compression de texte : la probabilité de la lettre suivante dépend fortement des lettres qui la précèdent. Dans la langue anglaise, si nous avons déjà encodé par

exemple les deux lettres « Th », la probabilité d'avoir la lettre « e » comme lettre suivante est très grande par rapport aux autres lettres. Ainsi, le mot de code que nous allons associer à la lettre « e » va être petit par rapport aux mots de code des autres lettres.

- en traitement de langue dans la reconnaissance vocale et la traduction de la voix en texte (e.g. Dragon NaturallySpeaking).
- pour générer la musique.
- en météo, finance, etc.

2.2.2 Codage vs compression

Dans la littérature, le codage et la compression de données sont tellement liés qu'il est difficile à les différencier. Cependant, quelques différences demeurent. Le codage peut être utilisé pour différents objectifs. Par exemple, nous pouvons utiliser le codage pour compresser les données, pour crypter l'information ou pour remédier aux erreurs de transmission. La compression de données, quant à elle, est toujours utilisée pour compresser les données. Son objectif est de libérer les données de toute sorte de répétitions non nécessaires pour atteindre un taux de compression maximale. Aussi, pour compresser une source de données, nous avons besoin de faire une modélisation convenable de cette source avant de choisir la technique de codage à utiliser. Pour bien comprendre ces différences, nous présentons quelques définitions.

Code : c'est l'espace des messages $C = \{b_1, b_2, ..., b_m\}$ qui permettent de représenter d'une façon unique les symboles ou les chaînes de symboles de la source S définie sur l'alphabet $\Sigma = \{a_1, a_2, ..., a_n\}$.

Codage : c'est une technique qui associe une représentation alternative aux entrées (symboles ou chaînes de symboles) d'une source de données. Mathématiquement, c'est le fait d'associer un mot de code b_i, qui est souvent une séquence de bits, à partir de l'espace des messages C à un symbole a_i ou à une chaîne de symboles de l'alphabet Σ selon un modèle de données précis. Les techniques de codage peuvent être utilisées pour différentes raisons, par exemple pour intégrer une résistance aux erreurs de transmission, pour maximiser la transmission d'information dans des réseaux de communications complexes, pour minimiser la dépense d'énergie et aussi pour compresser les données.

Compression de données : c'est une technique informatique qui vise à compacter les données pour minimiser leur espace de stockage ou pour accélérer leur transmission dans les réseaux à bande passante réduite. Les données contiennent souvent une sorte de redondance d'information. L'idée derrière la compression est de minimiser le plus possible cette redondance. Il y a deux types de compression : la compression sans perte et la compression avec perte. Pour bien comprendre la différence entre ces deux types, nous supposons que nous avons

deux fonctions c et d de codage et de décodage. Respectivement, nous supposons aussi que nous avons une chaîne d'entrée w composée de sous-chaînes w_i et que le (codage/décodage) procède en faisant une transformation de la chaîne d'entrée de gauche à droite, sous-chaîne par sous-chaîne.

Compression sans perte : une compression est dite sans perte si les données après décompression sont identiques aux données originelles. En d'autres termes, dans la compression sans perte, la conception des deux fonctions c et d doit respecter la contrainte suivante : $\forall w_i\,,\ d(c(w_i)) = w_i$. Parmi les techniques de codage sans perte utilisées, nous trouvons le codage de Huffman (Huffman *et al.*, 1952), le codage arithmétique (Witten *et al.*, 1987) et le codage de Lempel-Ziv (Ziv et Lempel, 1977, 1978; Welch, 1984).

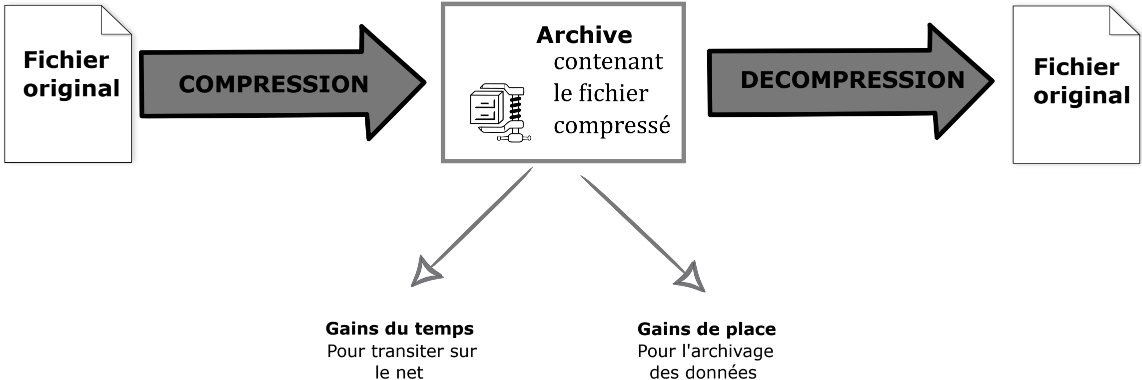

Figure 2.2 – Compression sans perte

Compression avec perte : une compression est dite avec perte si les données compressées subissent des pertes d'informations raisonnables de telle façon à ce que les données après décompression soient non identiques aux données originaires. Autrement dit, la conception des fonctions c et d ne respecte pas la contrainte $\forall w_i, d(c(w_i)) = w_i$. Notez que ceci permet d'avoir des taux de compression plus importants que ceux de la compression sans perte. Dans la compression de données avec perte, nous parlons notamment de techniques de sous-échantillonnage ou de quantification. Parmi les techniques de codage employées dans la compression avec perte, nous citons : JPEG (Wallace, 1992), MP3 (McCandless, 1999), MPEG (Le Gall, 1991), DIVX, etc.

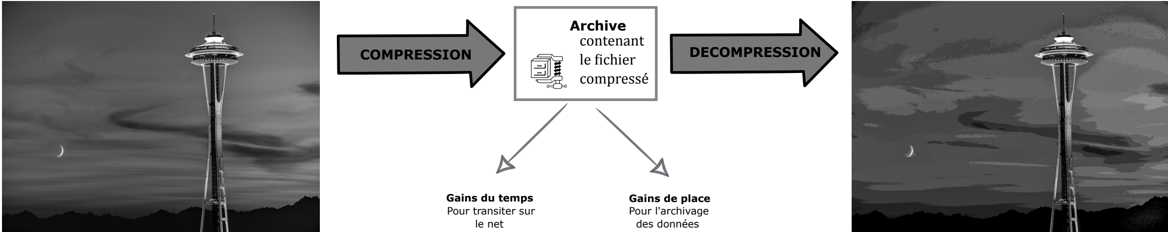

Figure 2.3 – Compression avec perte

Les techniques de codage peuvent être utilisées pour différentes raisons. Dans le cas de la compression de données par exemple, nous avons vu que le choix de la technique de codage dépend du contexte d'utilisation et de la nature de la source de données à compresser. Ainsi, pour compresser un fichier texte, dont la redondance peut être utilisée pour des raisons stylistiques comme l'utilisation de la rhétorique, nous devons utiliser une technique de codage sans perte. Par contre, pour compresser une image ou une vidéo, dont la redondance est utilisée pour des raisons de qualité comme une augmentation de niveaux de couleurs ou du nombre de pixels utilisés, nous pouvons utiliser une technique de compression avec perte pour gagner plus d'espace de stockage. Parfois, pour compresser un fichier texte, une technique de codage peut être meilleure qu'une autre. Dans tous les cas, pour une compression de données ou pour d'autres utilisations, il est nécessaire de faire une modélisation convenable des données que nous voulons traiter avant de choisir une technique de codage.

2.2.3 Types de codage

Nous avons vu dans la sous-section précédente que le codage est une technique qui permet de représenter l'information sous une forme différente que nous appellons code. Nous avons vu aussi que le codage est un outil que nous pouvons utiliser dans différents domaines pour différents objectifs. Nous pouvons regrouper les techniques de codage selon leurs utilisations en quatre types.

- **Codage source :** ce sont les techniques de codage utilisées dans la compression de données. Leur but est de réduire la redondance d'information. Nous citons par exemple :
  - **le codage de Shannon-Fano :** c'est une technique de codage développée par Claude E. Shannon (MIT) et Robert M. Fano dans les laboratoires de Bell (Shannon, 1949b). Son principe repose sur la création d'un arbre de code binaire. Chaque symbole de la source reçoit un mot de code de cet arbre d'une longueur qui varie selon sa probabilité. C'est un algorithme de compression de données sans perte. Comparé à d'autres méthodes, l'algorithme de Shannon-Fano est facile à mettre en œuvre. Cependant, il n'est pas toujours optimal.
  - **l'algorithme de Huffman** (Huffman *et al.*, 1952) **:** c'est un algorithme inventé par David Huffman en 1952. Ce dernier s'est inspiré du codage Shannon-Fano pour réaliser un algorithme qui crée des codes préfixes d'une façon simple et plus optimale. Cette technique est largement utilisée, car elle est très efficace ; la longueur moyenne d'un code crée par l'algorithme de Huffman peut atteindre l'entropie de la source. Habituellement, le taux de compression peut varier entre 20 % et 90 % selon la nature des données compressées.

- **le codage arithmétique** (Witten *et al.*, 1987) **:** c'est un codage entropique qui a le même principe d'association de code de l'algorithme de Huffman. Cependant, le codage arithmétique peut être meilleur que l'algorithme de Huffman étant donné que ce dernier utilise des mots de code de tailles entières et le premier peut utiliser des nombres flottants pour faire le codage. Par exemple, un symbole, dont la probabilité égale à 0.6, a une entropie égale à 0.44 bit/symbole. Pour coder ce symbole en utilisant l'algorithme de Huffman, nous avons besoin d'au moins 1 bit, par contre, en utilisant le codage arithmétique, nous avons besoin juste de 0.44 bit.

- **Codage de canal :** c'est un codage qui ajoute de la redondance aux données de la source pour compenser le bruit sur un canal de communication. Même si son nom parle seulement de canal, il a d'autres utilisations par rapport au stockage des données dans les disques durs, les CD-roms, les mémoires RAMS, etc. et aussi dans la transmission de données. En effet, le codage de canal ajoute de la redondance aux données pour pouvoir détecter et même corriger les erreurs que les données peuvent subir à cause des bruits de transmission ou des altérations dues aux dégradations des supports de stockage. Nous pouvons regrouper les codes issus de ce type de codage en deux catégories.
  - **Les codes détecteurs :** comme leur nom l'indique, ces codes permettent de détecter les erreurs. Par exemple, le protocole TCP (Postel, 1981), les algorithmes basés sur les sommes de contrôle comme le codage CRC (Contrôle de redondance cyclique) (Stigge *et al.*, 2006), les codes équilibrés (Knuth, 1986), etc.
  - **Les codes correcteurs :** ces codes permettent la détection et la correction des erreurs par reconstruction du message détérioré. Parmi les codes correcteurs, nous citons le code BCH (reprenant les initiales de ses inventeurs : Bose, Ray-Chaudhuri et Hocquenghem) (Bose et Ray-Chaudhuri, 1960) pour les ports série, le codage polynomial de Reed-Solomon (Reed et Solomon, 1960) pour les disques compacts, le code de Hamming (Moon, 2005), etc.

- **Le codage des caractères :** permet de représenter informatiquement l'ensemble des caractères. Le code ASCII (Bemer, 1960) est un exemple de ce type de codage.

- **Le codage numérique des données physiques en un format informatique :** par exemple le codage de la voix ou de la vidéo en format MP3, AVI, ... etc. ou par exemple le codage GSM qui utilise les codes cycliques (Van Lint, 2012) et les codes convolutionnels (Moon, 2005).

Nous avons vu dans la sous-section précédente que le codage permet d'associer une représentation alternative (mot de code) pour chacun des symboles a_i ou des chaînes de symboles d'un alphabet Σ donné. En effet, le codage se fait souvent pour des chaînes de symboles plutôt que des symboles. À partir d'un alphabet d'entrée Σ, nous créons un dictionnaire de mots (chaînes de symboles) nommé D. L'espace des messages C, qui contient les chaînes de sorties

que nous appelons les mots de code, est un autre dictionnaire sur un alphabet de sortie que nous appelons B. Habituellement, nous avons $|D| = |C|$. Le codage se fait en utilisant une fonction $c : \Sigma^* \to B^*$ injective définie sur tous les éléments de D. Selon les mots des deux dictionnaires d'entrée D et de sortie C, nous pouvons aussi faire une autre catégorisation pour les différentes techniques de codage.

- **Les codes variables à fixes (VF) :** les mots dans D ont des longueurs variables mais la longueur l des mots dans C est fixe. L'efficacité de c dépend de la longueur moyenne des mots d'entrée et de l. L'objectif ici est de maximiser la longueur moyenne des mots d'entrée. L'algorithme de Tunstall (Tunstall, 1967) et certains codages sources de Savari (1999); Visweswariah *et al.* (2001) sont des exemples de techniques qui utilisent les codes (VF).
- **Les codes variables à variables (VV) :** Les mots dans D et dans C ont des longueurs variables. L'efficacité de c dépend de la longueur moyenne des mots d'entrée et de la longueur moyenne des mots de code. L'objectif ici est de maximiser la longueur des mots d'entrée tout en minimisant celle des mots de code. Parmi les algorithmes de codage VV, nous citons LZ77 (Ziv et Lempel, 1977), LZ78 (Ziv et Lempel, 1978), LZW (Welch, 1984) et LZSS (Storer et Szymanski, 1982).
- **Les codes fixes à variables (FV) :** La longueur k des mots dans D est fixe mais celles des mots dans C sont variables. La longueur k peut être aussi basse que 1. L'efficacité de c dépend de k et de la longueur moyenne des mots de code. Nous essayons ici de minimiser la longueur moyenne des mots de code. L'algorithme de Huffman (Huffman *et al.*, 1952), l'algorithme de Shannon-Fano (Shannon, 1949b) et le codage arithmétique (Witten *et al.*, 1987) sont des techniques de codage FV.
- **Les codes fixes à fixes (FF) :** La longueur k des mots dans D et la longueur l des mots dans C sont fixes. L'efficacité de c dépend directement de k et l. Les codes équilibrés (Knuth, 1986) et certains codes sources sans perte de Arimura et Iwata (2010) sont des codes FF.

Nous avons vu dans cette sous-section qu'il y a entre autres quatre types de codage : codage source, codage de canal, codage de caractères et codage numérique. Les codes équilibrés font partie du deuxième type. Nous avons vu aussi la catégorisation des codes selon le type des mots de codes qu'ils fournissent. Les codes équilibrés sont des codes FF.

2.2.4 Code et unicité de décodage

La conception d'un code performant est très importante. Cependant, nous devons être capable de retrouver l'information source. En d'autres termes, nous ne devons pas oublier la décodabilité de l'encodage pendant la conception. Il est aussi nécessaire que la décodabilité

soit unique. Pour s'assurer qu'un code est uniquement décodable nous utilisons l'algorithme de test de décodabilité ci-dessous. Avant de présenter cet algorithme, nous définissons les suffixes pendants.

Suffixe pendant : le suffixe pendant d'une chaîne de bits est la partie qui reste après avoir enlevé un préfixe de cette chaîne. Par exemple, la chaîne de bits « 110 » est un préfixe de « 1100 ». En enlevant « 110 » de « 1100 », nous allons avoir « 0 » comme suffixe pendant.

Algorithme de test de décodabilité unique [6]

Cet algorithme est dérivé du théorème de Sardinas-Patterson.

Soit C l'ensemble des mots de code.

> $D := \text{diff}(C, C)$;
> **Faire**
> > $D := D \cup \text{diff}(C, D) \cup \text{diff}(D, C)$;
>
> **tant que** D s'accroît ;
> **Indiquer** que C est uniquement décodable **ssi** $C \cap D = \emptyset$

où diff produit les suffixes pendants entre un ensemble S, les préfixes en puissance, et T, les extensions en puissance :

$$\text{diff}(S, T) = \{w \mid u \in S \textit{ et } v \in T \textit{ et } w \neq \epsilon \textit{ et } u \cdot w = v\}.$$

Exemple : Considérons par exemple le code suivant $C = \{0, 01, 110, 111\}$

1. Initialement, nous avons $D := \text{diff}(C, C)$, donc $D = \{1\}$.
2. Première itération de la boucle : $D = \{1\} \cup \{\emptyset\} \cup \{10, 11\} = \{1, 10, 11\}$.
3. Deuxième itération de la boucle : $D = \{1, 10, 11\} \cup \{\emptyset\} \cup \{0\} = \{1, 10, 11, 0\}$.
4. Fin, D ne s'accroît plus.

Nous avons $C \cap D \neq \emptyset$, donc C n'est pas uniquement décodable. Par exemple, la chaîne de bits « 01110 », peut être interprété, en se basant sur ce code, de deux façons différentes : en « 01 » et « 110 » ou en « 0 », « 111 » et « 0 ».

Dans cet exemple, nous avons fait deux itérations. Ce nombre d'itérations peut augmenter avec l'augmentation du nombre de mots de code. L'utilisation d'un code préfixe permet d'éliminer toutes les itérations du test de décodabilité.

Code préfixe : c'est un dictionnaire de mots de code qui sont construits de façon à ce qu'aucun mot de code ne soit un préfixe d'un autre mot de code du même ensemble. Par exemple, le code $\{0, 10, 11\}$ est un code préfixe. Par contre, le code $\{0, 1, 10, 11\}$ n'est pas un code préfixe, car le mot de code « 1 » est préfixe des mots de code « 10 » et « 11 ». Cette

6. Source : tiré des notes du cours Ift-7023 de l'université Laval.

caractéristique permet aux codes préfixes d'être uniquement décodables. En effet, l'application de l'algorithme du test de décodabilité sur un code préfixe s'arrête à la deuxième étape sans faire aucune itération. Ceci est dû au fait qu'aucun suffixe pendant ne peut être trouvé étant donné qu'aucun mot de code n'est préfixe d'un autre.

Pour vérifier si un code est un code préfixe, il suffit de tracer son arbre binaire et de vérifier s'il est conforme aux deux conditions suivantes.

1. Chaque nœud a au maximum 2 branches.
2. Tous les mots de codes sont associés à des nœuds externes (feuilles).

Exemple : La figure 2.4 représente trois exemples d'arbres de codes dont le deuxième est le seul code préfixe. Le tableau 2.1 représente les codes issues des trois arbres (extrait du livre de Sayood (2012)).

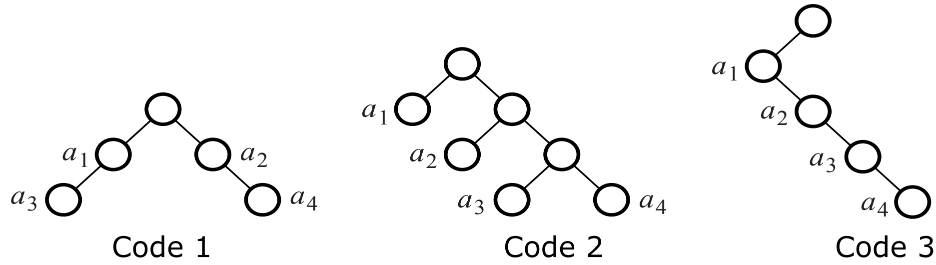

FIGURE 2.4 – Arbres de codes binaires

Tableau 2.1 – Codes issus des arbres binaires

Alphabet d'entrée	Code 1	Code 2	Code 3
a_1	0	0	0
a_2	1	10	01
a_3	00	110	011
a_4	11	111	0111

Code suffixe : de la même façon qu'un code préfixe, nous disons qu'un code est suffixe si ses mots de code sont construit de façon à ce qu'aucun mot de code ne soit un suffixe d'un autre mot de code du même ensemble. Un code suffixe est donc un code uniquement décodable si nous commençons le décodage de droite à gauche. Aussi, en inversant les mots de code d'un code préfixe nous obtenons un code suffixe. Par exemple, le code 3 dans le tableau 2.1 est un code suffixe.

Code non ordonné : un code binaire C est dite non ordonné quand aucun de ses mots de code n'est "contenu" dans un autre. C'est-à-dire que C est non ordonné si, et seulement si, les positions des bits à 1 dans un mot de code ne sont en aucun cas un sous-ensemble des positions des bits à 1 dans un mot de code différent. Mathématiquement, C est un code non

ordonné si, et seulement si, pour toute paire de mots de code (X,Y) différents dans C, nous avons $N(X,Y) \geq 1$, où N et le nombre de croisements $1 \rightarrow 0$ de X à Y. Par exemple, si $X =$ « 11100 » et $Y =$« 01111 », alors $N(X,Y) = 1$ et $N(Y,X) = 2$. (Knuth, 1986; Al-Bassam et Bose, 1990).

Nous avons vu dans cette sous-section comment vérifier l'unicité de décodage d'un code donné. Nous avons vu aussi que cette vérification peut être très rapide si le code vérifié est préfixe, suffixe ou non ordonné. Les codes équilibrés sont des codes binaires FF non ordonnés (Knuth, 1986). Ainsi, ces codes sont uniquement décodables.

2.3 Permutations

Dans notre vie quotidienne, nous avons souvent besoin d'énumérer des « événements » tels que l'arrangement des objets d'une certaine manière, la partition des choses sous une certaine condition, la distribution des objets selon une certaine spécification, etc. Parmi les problèmes fréquents que nous pouvons rencontrer, nous retrouvons le type de problèmes suivant :

De combien de façons pouvons-nous arranger 5 boules noires et 3 blanches en ayant deux boules blanches consécutives ?

Ce problème est une représentation simplifiée de ce qu'on appelle une permutation ou arrangement, selon les auteurs.

Nous entendons souvent un autre type de problème dans ce domaine que nous appellons combinaison. Pour bien comprendre la différence entre ces deux types de problèmes, voici un exemple simplifié de ce que nous appellons combinaison.

De combien de façons pouvons-nous diviser 10 boules noires, numérotées de 1 à 10, en trois groupes de 4, 3 et 2 boules consécutivement en rejetant une boule ?

Soit l'ensemble $A = \{a_1, a_2, ..., a_n\}$ une liste de n objets distincts. Pour $0 \leq r \leq n$, une r-permutation de A est une manière d'arranger r différents objets de A dans une rangée. Si $r = n$, nous appellons tout simplement la n-permutation de A une permutation de A.

Exemple : Soit $A = \{a, b, c, d\}$ l'ensemble de départ. Les 3-permutations de A sont :

$$\begin{gathered} abc,\ abd,\ acb,\ acd,\ adb,\ adc, \\ bac,\ bad,\ bca,\ bcd,\ bda,\ bdc, \\ cab,\ cad,\ cba,\ cbd,\ cda,\ cdb, \\ dab,\ dac,\ dba,\ dbc,\ dca,\ dcb. \end{gathered}$$

Nous avons donc 24 3-permutations.

Soit P_r^n le nombre totale des r-permutation de A. Dans l'exemple précédent nous avons $P_3^4 =$

24. En effet, pour construire une r-permutation nous procédons comme suit.

Pour choisir le premier élément nous avons n choix. Ensuite, pour choisir le deuxième élément il va nous rester $n-1$ choix. Pour choisir le kième élément, nous avons le choix entre $(n-k+1)$ éléments. Et ainsi de suite jusqu'au choix du r élément où nous avons $(n-r+1)$ choix.

Donc $P_r^n = n \times (n-1) \times ... \times (n-k+1)... \times (n-r+1)$ (Hall, 1983).

En utilisant la factorielle nous obtenons la formule suivante :

$$P_r^n = \frac{n!}{(n-r)!}. \tag{2.14}$$

Par convention, nous avons $0! = 1$, $P_0^n = 1$ et $P_n^n = n!$.

Retournons à l'exemple cité au début.

Nous voulons savoir combien de permutations que nous pouvons trouver en arrangeant 5 boules noires et 3 blanches. La condition imposée est :

- Nous devons avoir exactement deux boules blanches consécutives.

Nous voulons donc considérer les deux boules blanches comme étant un groupe à arranger tout seul.
Nous allons avoir donc $P_2^2 = 2$ permutations.
Nous allons par la suite considérer le groupe des 5 boules noires et une boule blanche comme étant un groupe à arranger. Donc $P_6^6 = 6!$.

Donc le nombre de permutation totale pour arranger 5 boules noires et 3 blanches en considérant que 2 boules blanches sont consécutives est de : $P = P_6^6 \times P_2^2 = 6! \times 2 = 720 \times 2 = 1440$.

Il y a quelques permutations particulières telles que l'identité et les permutations cycliques. La première est la permutation qui renvoie chaque élément de l'ensemble de départ à lui-même. Les deuxièmes sont des permutations qui nécessitent d'arranger leurs éléments dans une courbe circulaire fermée.

Exemples :

- Soit $A = \{a, b, c, d\}$ l'ensemble de départ. La permutation identité est $P_{id} = (a, b, c, d)$.
- Nous supposons que nous avons 3 points de cercle a, b et c fixes et équidistants.
  Les différents arrangements possibles de ces trois points représentés dans la figure 2.5 sont :

$$abc, \; cab, \; bca$$

Ces permutations sont appelées des permutations cycliques.

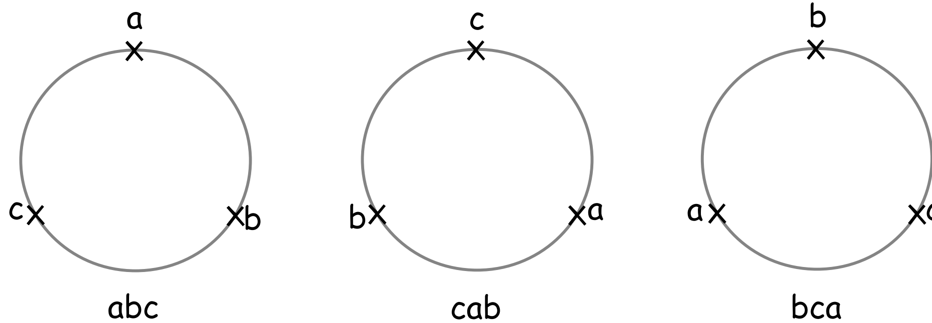

Figure 2.5 – Exemples de permutations cycliques

La formule utilisée pour calculer le nombre des permutations cycliques est (Hall, 1983) :

$$Q_r^n = \frac{P_r^n}{r}. \tag{2.15}$$

Les permutations trouvent plusieurs utilisations dans le domaine du codage (Slepian, 1965; Blake *et al.*, 1979; Bailey, 2009). Parmi ces utilisations, nous citons : l'utilisation des permutations dans la composante d'entrelacement des codes turbo (Berrou et Glavieux, 2003), dans la norme de télécommunication mobile 3GPP (Access) et dans le hachage à permutation unique (Dolev *et al.*, 2013). Nous utilisons aussi les permutations dans notre technique de codage.

2.4 Performance de codage

En informatique, nous avons souvent tendance à comparer les différents algorithmes pour déterminer ceux qui sont plus efficaces en termes de vitesse d'exécution et de consommation des ressources en espace de mémoire. Les ressources informatiques sont limitées. Ainsi, pour une efficacité maximale nous souhaitons minimiser l'utilisation de ces ressources le plus possible en prenant en considération toute optimisation possible même si elle n'apporte pas une amélioration remarquable. « Dans les disciplines d'ingénierie établies, une amélioration de 12%, facilement obtenue, n'est jamais considérée comme marginale et je crois que le même point de vue devrait prévaloir en génie logiciel »(Knuth, 1974).

Un algorithme est considéré efficace s'il peut fonctionner en un temps raisonnable dans les ressources disponibles. Il y a trois approches pour mesurer l'efficacité ou plutôt la complexité d'un algorithme.

- La première approche, c'est ce qu'on appelle l'analyse empirique (*Benchmarking*). Cette approche permet de savoir pratiquement en combien de temps l'algorithme peut être exécuté et combien de ressources il va consommer. Le problème avec cette approche; c'est que l'algorithme va être écrit dans un language de programmation particulier, les

calculs vont être faits sur un ordinateur spécifique ayant une architecture spécifique, avec un compilateur particulier et des données d'entrée particulières. Les résultats vont être ainsi intrinsèques aux conditions du test.

- La deuxième approche, c'est l'analyse théorique qui se fait indépendamment des spécifications du matériel et des données sources. Pour avoir un bon compromis entre la précision et la facilité d'analyse, nous utilisons souvent l'analyse asymptotique des complexités temporelle et spatiale. L'analyse asymptotique permet de savoir le nombre d'opérations fait par l'algorithme pour traiter une entrée de taille n en utilisant la notation « grand O ». Ceci est fait sans mettre en considération le temps exact (en secondes) nécessaire pour exécuter une opération ni la taille en octets de chaque valeur de l'entrée. Le tableau 2.2 représente quelques exemples de calcul de complexité et leur notation en « grand O » (Danziger, 2015). Parfois, pour mesurer l'efficacité spatiale des algorithmes de codage, nous calculons leur redondance. Ceci va nous permettre de comparer les performances de plusieurs algorithmes, surtout pour les algorithmes de codage de canal. Nous utilisons cette approche pour comparer l'algorithme Pacman et les réalisations antérieures à la théorie de la création des codes équilibrés.
- La dernière approche, c'est l'analyse des problèmes d'implémentation. Comme nous avons vu dans la première approche, en pratique, l'efficacité d'un algorithme peut être influencée par le choix du matériel, le choix du langage de programmation, le choix d'un compilateur spécifique pour le langage utilisé ou le choix du système d'exploitation utilisé. Ainsi, l'algorithme peut-être facile à implémenter mais très lent dans un environnement précis et difficile à implémenter mais très rapide dans un autre. Ainsi, un algorithme est meilleur s'il est performant et facile à implémenter dans plusieurs environnements de programmation.

Nous avons vu dans ce chapitre différentes notions de la théorie de l'information, du codage et de la modélisation. Ces notions sont très importantes pour une bonne compréhension de la problématique. Nous avons présenté aussi les différents types de codages existants. Ensuite, nous avons présenté quelques notions importantes sur les permutations. En fin, nous avons vu les différents aspects de mesure de performances d'un algorithme.

Tableau 2.2 – Exemple de la notation grand O

Notation	Nom	Exemple
O(1)	Constante	Tableaux
O(log(n))	Logarithmique	Recherche dichotomique
O(log(log(n)))	Double logarithmique	Recherche par interpolation
o(n)	Sous-linéaire	Problème de la distance du cantonnier (*Estimating Earth-Mover Distance*)
O(n)	Linéaire	Parcours de liste
O($n\log(n)$)	Loglinéaire, linearithmique, quasilinéaire ou supralinéaire	Tri fusion
O(n^2)	Quadratique	Parcours de tableaux 2D
O(n^3)	Cubique	Multiplication matricielle non-optimisée
O(n^c)	Polynômiale (classe différente pour chaque c>1)	Grammaire d'arbres adjoints (*Tree-adjoining grammar*)
O(c^n)	Exponentielle (classe différente pour chaque c>1)	Problème du voyageur de commerce (avec une approche dynamique)
O(n!)	Factorielle	Problème du voyageur de commerce (avec une approche naïve).

Chapitre 3

Revue de la littérature sur les codes équilibrés

Dans ce chapitre, nous présentons quelques notions et définitions sur les codes équilibrés, dans la première section. Nous présentons aussi dans cette section la construction mathématique de ces codes et nous montrons ses limites. Dans la deuxième section, nous parlons des principales approches de construction des codes équilibrés réalisées dans les travaux de recherche antérieurs.

3.1 Les codes équilibrés

Les codes équilibrés sont appliqués dans plusieurs domaines. Ils sont utilisés par exemple pour réduire le bruit dans les systèmes VLSI (Tabor, 1990). Dans le domaine de la télécommunication, ils sont utilisés dans la synchronisation et la transmission des données par fibre optique (Takasaki *et al.*, 1976). Leur utilisation dans le domaine de l'identification par radiofréquence (RFID) permet d'augmenter les taux de transfert de données via les canaux RFID (Durgin, 2015). Ces exemples d'applications dénotent de l'importance de l'optimisation de la création de codes équilibrés.

Comme nous l'avons vu dans la problématique (section 1.2), un code équilibré est un code dont chacun de ses mots de code a un nombre de bits à 1 égal à celui des bits à 0. Ainsi l'ensemble des mots binaires $\{0011, 0101, 0110, 1001, 1010, 1100\}$ est un code équilibré de taille 4.

Par nature, les codes équilibrés contiennent de la redondance. Soit $\{0,1\}^{n\in\mathbb{N}}$ l'ensemble des blocs binaires de taille n. Cet ensemble contient 2^n blocs de taille n. On note $\mathcal{B}_m$ l'ensemble des blocs équilibrés de taille m. Le nombre des éléments de cet ensemble est $\binom{m}{m/2}$. Par exemple, pour $n = m = 2$, nous avons $\{0,1\}^2 = \{00, 01, 10, 11\}$ et $\mathcal{B}_2 = \{01, 10\}$. Pour coder tous les éléments de l'ensemble $\{0,1\}^{n\in\mathbb{N}}$ en blocs équilibrés, nous devons procurer une fonction Enc qui transforme les données binaires en blocs équilibrés. En d'autres termes, $Enc : 2^n \to 2^m$,

où $m > n$ et $m = n + p$. Le p ici représente le nombre de bits redondants ajoutés par Enc dans chaque bloc produit. Ces bits redondants sont appelés des bits de parité. Pour avoir un code non ordonné, p doit être égal pour tous les blocs binaires. Ainsi Enc est une fonction de codage fixe à fixe. Pour l'exemple précédent, si l'on choisit Enc comme étant la fonction qui ajoute à chaque bloc binaire son complément, nous obtenons le code équilibré suivant : $\{0011, 0110, 1001, 1100\}$. En codage, la redondance des codes équilibrés peut être utilisée pour ajouter des conditions de vérifications pour détecter les erreurs. Cependant, cette redondance doit être minimale pour avoir un codage efficace. On note $Dec : 2^m \rightarrow 2^n$ la fonction de décodage correspondante.

D'après le lemme de Sperner (Sperner, 1928), pour un dictionnaire de mots de taille n, la meilleure façon pour créer des codes équilibrés efficaces est la construction de la liste de tous les mots de codes de longueur $m = n + p$, où p est le nombre de bits de parité minimale (le minimum de bits redondants à ajouter) (Knuth, 1986). Dans ce cas, le nombre de blocs équilibrés de taille m est égal théoriquement à $\binom{m}{\frac{m}{2}}$. En utilisant l'approximation de Sterling, ce nombre est (Knuth, 1986) :

$$\binom{m}{\frac{m}{2}} \approx \frac{2}{\sqrt{2\pi m}} \ . \ 2^m, \ m \geq 1. \tag{3.1}$$

Notons que $\frac{2}{\sqrt{2\pi m}} < 1$.

Un mot de code équilibré peut donc coder $\log \binom{m}{\frac{m}{2}}$ bits source. Autrement dit, pour coder des blocs d'entrée de taille n, nous avons besoin de blocs équilibrés de taille m de façon à ce que $\binom{m}{m/2} \geq 2^n$.

Le nombre p de bits de parité minimale (seuil minimal) pour créer des blocs équilibrés de taille m est donc :

$$p \geq m - \log \binom{m}{\frac{m}{2}}. \tag{3.2}$$

Le calcul approximatif du seuil minimal réalisé par Knuth est (Knuth, 1986) :

$$p \approx \frac{1}{2} \log m + 0.326, m \gg 1. \tag{3.3}$$

Donc, théoriquement, la redondance minimale relative nécessaire pour la création de blocs équilibrés de taille m est :

$$R(m) = \frac{\frac{1}{2} \log m + 0.326}{m} \approx \frac{1}{2m} \log m. \tag{3.4}$$

Comme nous l'avons vu dans la section 1.2, l'utilisation des tables de consultation ou du codage énumératif pour la création des codes équilibrés n'est pas pratique. Il est nécessaire donc de trouver une technique simple pour créer des codes équilibrés dans laquelle l'utilisation des tables de consultations est éliminée ou limitée et la consommation des ressources (temps, espace mémoire et taille de registre) est raisonnable.

L'idée la plus simple pour l'élimination de tables de consultation est la création des mots de code équilibrés w_r en ajoutant le complément w' du mot à coder w de façon à ce que $w_r = ww'$ (Knuth, 1986). La taille des mots de code est ainsi égale à $2 \times n$, ce qui sûrement diminue l'efficacité du codage. Il est nécessaire donc de viser une méthodologie qui fait le compromis entre la consommation des ressources et l'efficacité de codage. Il faut aussi prendre en considération la simplicité de la méthode.

L'impraticabilité des méthodes précédentes a conduit de nombreux chercheurs à concevoir des techniques de construction pratiques, presqu'optimales pour des codes équilibrés. Le mot « pratique » signifie que ces techniques doivent être d'une complexité O(m) en temps et en l'espace pour le codage ou le décodage d'un bloc. Le mot « presqu'optimal » signifie qu'une nouvelle technique doit être comparable ou meilleure que d'autres techniques pratiques, en termes de redondance. Nous allons présenter les principaux travaux de recherche réalisés dans ce cadre dans la section suivante.

3.2 Travaux de recherche antérieurs

Knuth fut le premier à présenter les codes équilibrés (Knuth, 1986). Il a proposé un algorithme simple et rapide qui utilise des tables de consultations qui ne consomment pas beaucoup de ressources. Pour Knuth, un code de taille m est équilibré, si et seulement si, il contient $\lfloor \frac{m}{2} \rfloor$ bits à 1 et $\lceil \frac{m}{2} \rceil$ bits à 0. Le nombre de blocs balancés de taille m est $|B_m| = \binom{m}{\lfloor \frac{m}{2} \rfloor}$. Le nombre de bits de parité p qu'on doit avoir pour coder n bits d'informations dans les m bits du bloc équilibré doit vérifier l'inégalité $|B_{n+p}| \geq 2^n$. En partant de ces conditions, l'idée de l'algorithme de Knuth consiste à coder les mots binaires w en des codes équilibrés uw_k, où w_k représente la transformation de w après avoir complémenté ses k premiers bits. Le nombre de bits à 1 doit être égal au nombre de bits à 0 dans w_k. Le nombre de bits complémentés k est codé dans le préfixe équilibré u. La taille de u est q, $p \leq q < n$ (Knuth, 1986). Par contre, en termes de redondance, l'algorithme de Knuth n'est pas efficace ; le nombre de bits de parité ajoutés est $q \approx \log m$. Ce qui est presque le double de la parité minimale $p \approx \frac{1}{2} \log m$ (Immink et Weber, 2009a, 2010; Immink *et al.*, 2011). Ceci est dû à la liberté de sélection créée par la multiplicité d'encodage (ME) de cet algorithme. En effet, pour un certain bloc à encoder, il se peut qu'il y ait plusieurs valeurs possibles pour k. Toutefois, pour Knuth l'encodeur et le décodeur s'accordent pour utiliser le premier k trouvé. Weber et Immink ont également montré que le nombre c des valeurs possibles de k pour équilibrer un bloc binaire de taille n

est au plus $\frac{n}{2}$ (Weber et Immink, 2010) :

$$1 \leq c \leq \frac{n}{2}. \tag{3.5}$$

Par exemple, le bloc de bits suivant peut être équilibré en complémentant soit le premier bit, soit les 3 premiers bits, soit les 5 premiers bits, soit les 7 premiers bits : $k \in \{1, 3, 5, 7\}$.

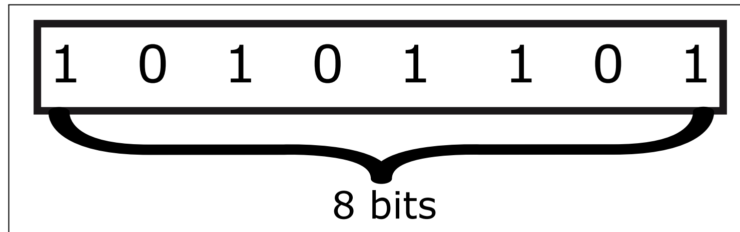

Figure 3.1 – Bloc de 8 bits non équilibré

Dans l'algorithme de Knuth, l'encodeur utilise toujours la première possibilité rencontrée. Toutefois, pour coder le nombre k de bits complémentés, Knuth utilise un préfixe équilibré. Dans l'exemple, le préfixe est de taille 6.

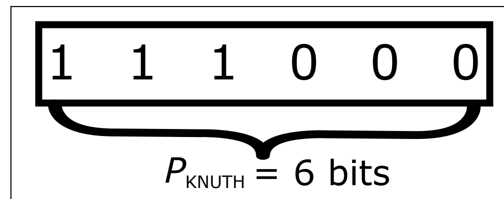

Figure 3.2 – Préfixe du code

Le résultat du codage en utilisant l'algorithme de Knuth est présenté dans la figure 3.3.

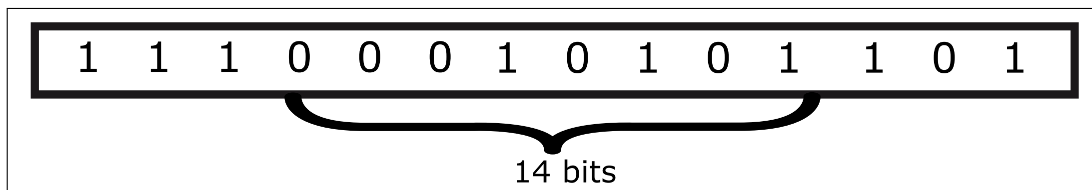

Figure 3.3 – Bloc de 14 bits équilibré

Donc, en équilibrant un mot de bits de taille $n = 8$ bits par l'algorithme de Knuth, sa taille totale devient $m = 14$ bits. En théorie, cela nécessite un préfixe de taille $p = m - \log\binom{m}{\frac{m}{2}} \simeq 14 - 11 = 3$ bits. Cette valeur est la moitié de ce que l'algorithme de Knuth utilise. En effet, nous sommes capable de coder 11 bits sources dans un code équilibré de taille égale à 14 bits

et non pas juste 8 bits sources. Cette redondance diminue l'efficacité de l'algorithme de Knuth surtout pour des blocs d'entrée de grande taille.

Knuth a pu démontrer dans ses résultats les plus favorables que la redondance pour le codage en un mot de code de taille m ($m \gg 1$) est presqu'égale à $\log(m) + \frac{1}{2}\log\log m$. Ce qui est à peu près le double des résultats théoriques (voir 3.3).

La simplicité et la rapidité de l'algorithme de Knuth ont attiré plusieurs chercheurs tels que Alon *et al.* (1988); Al-Bassam et Bose (1990) et Tallini et Bose (1999) qui ont essayé de réduire sa redondance par rapport à la théorie. Immink et Weber ont réussi à diminuer cette redondance en modifiant l'algorithme (Immink et Weber, 2010). Ils ont procédé de la même manière que l'algorithme de Knuth. C'est-à-dire, en équilibrant le mot d'entrée avec la première possibilité qui se présente et en ajoutant un préfixe équilibré qui permet au décodeur de savoir le nombre de bits complémentés. La différence cette fois-ci, c'est qu'ils ont décidé d'exploiter la multiplicité d'encodage de l'algorithme de Knuth pour diminuer la taille du préfixe. En effet, L'encodeur et le décodeur peuvent faire des prédictions pour éliminer des préfixes inutiles. Si, par exemple, nous avons obtenu, après avoir appliqué l'algorithme de Knuth sur un mot d'entrée de taille 6, le mot équilibré « 000111 ». Nous pouvons déduire intuitivement qu'il y a juste quatre cas possibles de l'information initiale (bloc de bits source) : « 100111 » $\rightarrow$ préfixe $= 1$, « 110111 » $\rightarrow$ préfixe $= 2$, « 111111 » $\rightarrow$ préfixe $= 3$, « 111011 » $\rightarrow$ préfixe $= 4$. Les autres cas seront éliminés. Par exemple : le cas « 111100 » est éliminé parce qu'on peut équilibrer le mot en complémentant juste le premier bit donc le préfixe=1 alors que ce cas-là est déjà présent. nous allons donc coder un préfixe à 4 éléments aux lieux d'un préfixe à 6 éléments. En procédant par cette manière Immink et Weber ont réussi à diminuer la redondance dans leur configuration surtout pour les grandes tailles de m (Weber et Immink, 2010). La figure 3.4 représente les résultats de leurs calculs.

Ils ont aussi montré que l'information moyenne, A_{SF}, qui peut être transmise par la liberté de sélection dans la technique de Knuth décrite ci-dessus est (Weber et Immink, 2010) :

$$A_{SF}(m) \approx \frac{1}{2}\log m - 0.916. \tag{3.6}$$

Al-Rababa'a *et al.* (2013) ont remarqué que cette liberté de sélection créée par ME est une bonne opportunité pour utiliser le recyclage des bits. En fait, le but de cette technique est de minimiser la redondance en exploitant ME. L'application sur l'algorithme de Knuth, illustrée dans la figure 3.5 (Al-Rababa'a *et al.*, 2013), est comme suit : à chaque fois qu'il y a ME dans l'encodage d'un bloc de bits, nous créons un arbre binaire de tous les cas possibles. Le choix du nombre de bits à complémenter se fait selon les premiers bits du mot de code suivant. Ainsi, ces bits vont être passés implicitement au décodeur. Cela va nous permettre de récupérer une partie des bits perdus à cause de la redondance. Pour simplifier l'implantation du recyclage des bits, les auteurs ont choisi d'enlever les bits recyclés à partir de la fin de la chaine encodée.

À la fin d'encodage, un mot de code spécial est envoyé au décodeur pour indiquer le nombre de bits d'intersection entre le codage et le recyclage de bits.

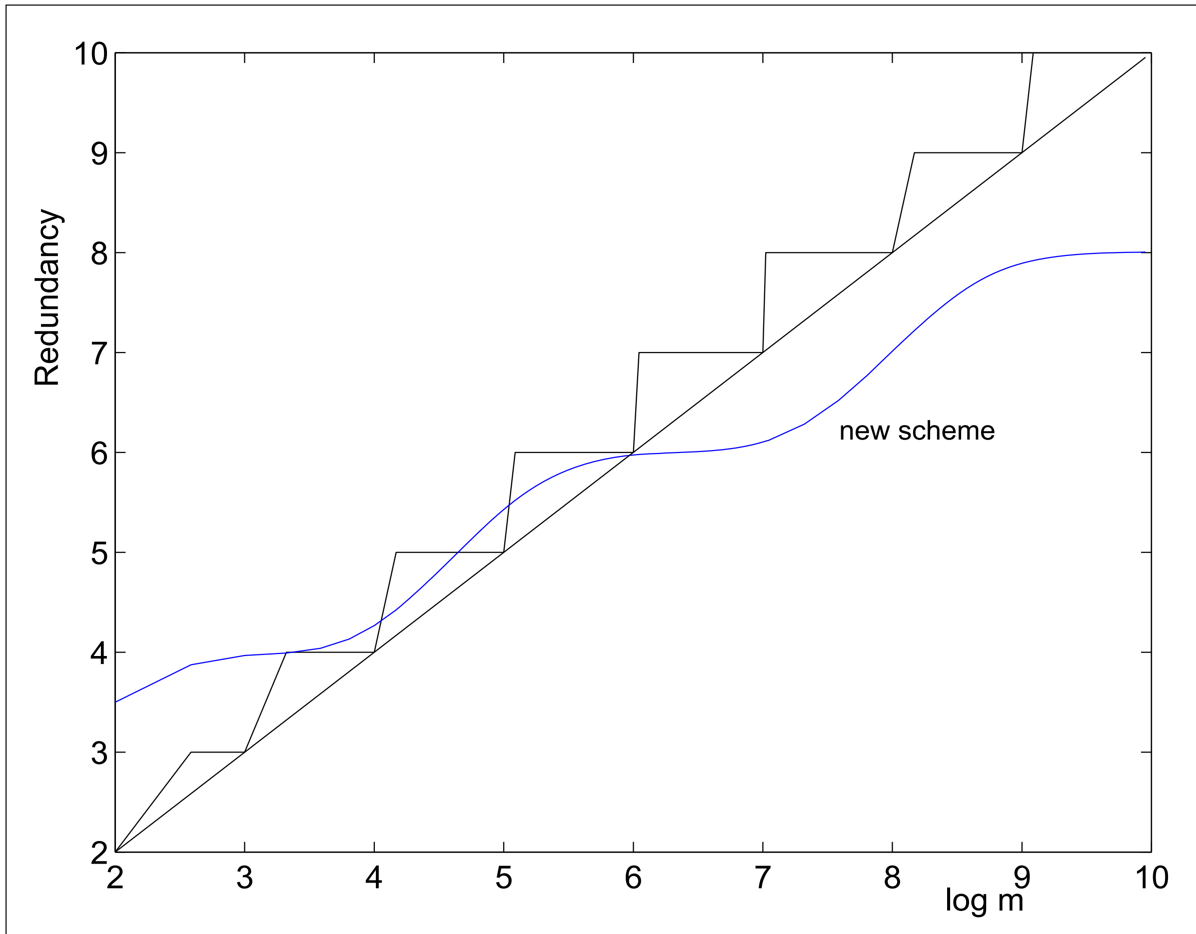

FIGURE 3.4 – La longueur moyenne du préfixe en fonction de $\log(m)$ des codes équilibrés issues de l'algorithme de Knuth à préfixe VL équilibré. Nous avons représenté la redondance minimale de la construction de Knuth $\log(m)$ et $\lceil \log(m) \rceil$. Le graphique est tiré des travaux de recherche d'Immink et Weber (2010).

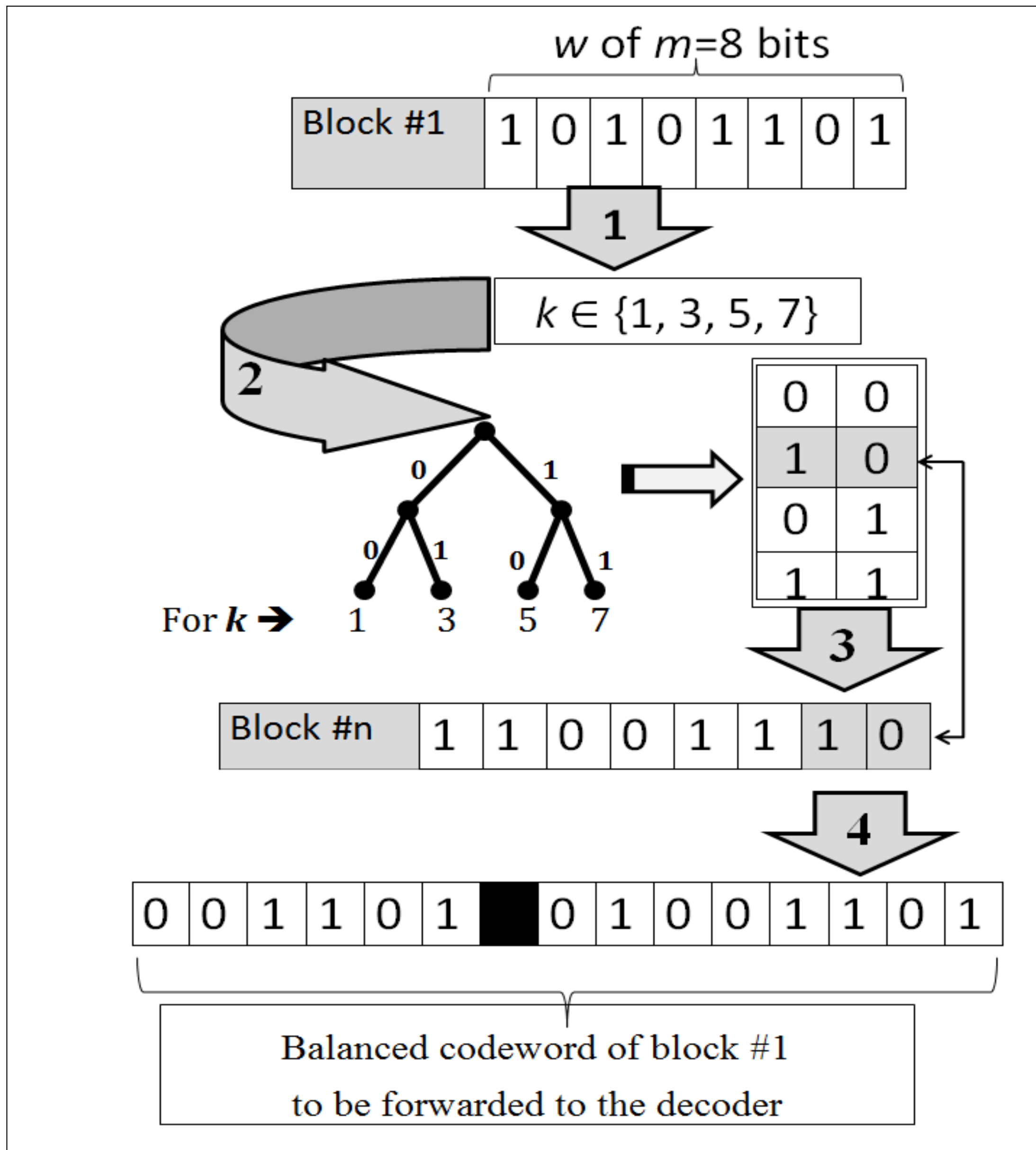

FIGURE 3.5 – Les principales étapes du recyclage de bits pour l'algorithme de Knuth. L'image est tirée des travaux de recherche d'Al-Rababa'a *et al.* (2013).

Pour prouver leur résultats, Al-Rababa'a et al. ont fait la comparaison illustrée dans la figure 3.6.

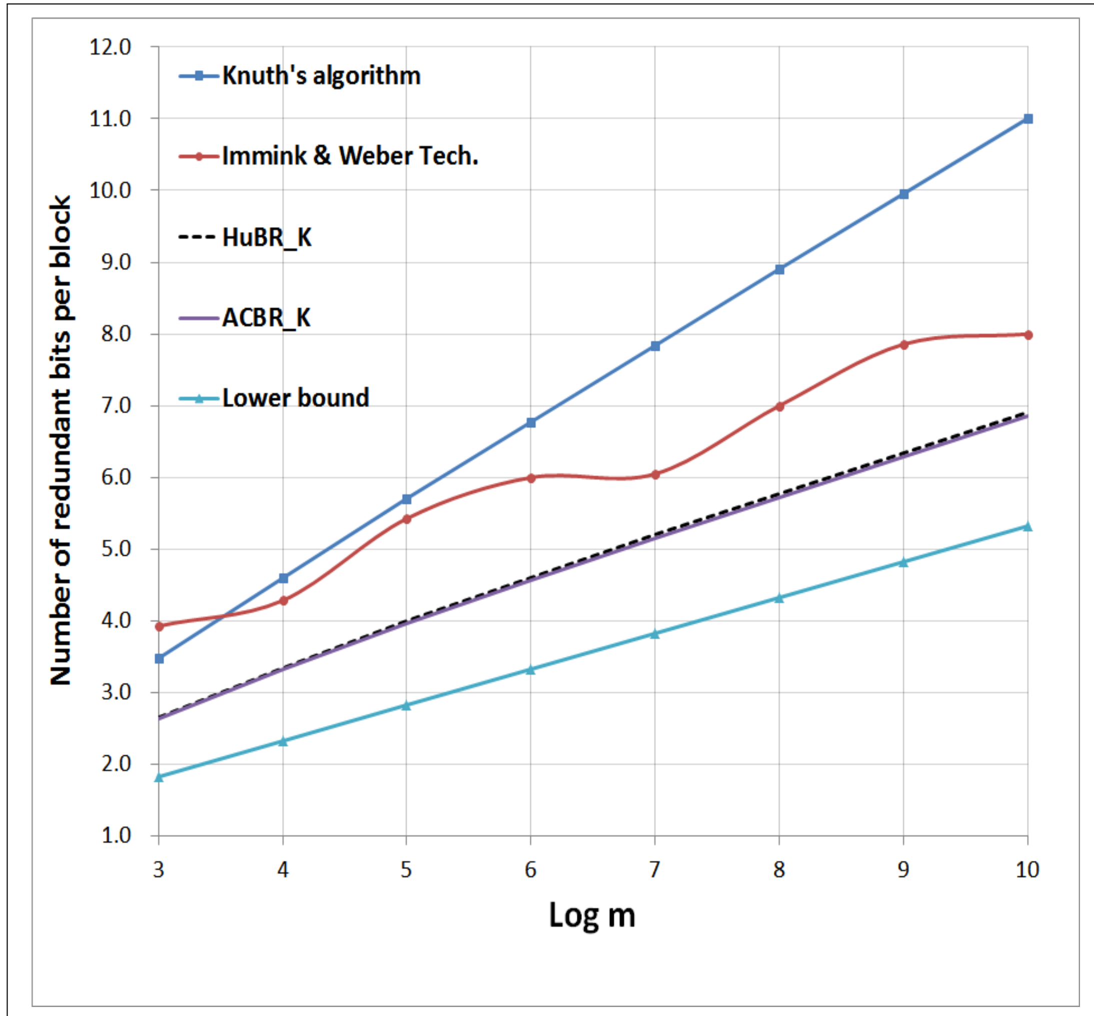

FIGURE 3.6 – Comparaison entre les résultats de l'algorithme de Knuth sans et avec recyclage de bits ($HuBR_k$ et $ACBR_k$), les résultats de la technique de Immink et Weber, et la borne théorique 3.3. Le graphique est tiré des travaux de recherche d'Al-Rababa'a *et al.* (2013).

Les résultats obtenus sont pertinents. Cependant, un écart subsiste encore par rapport au seuil théorique minimal.

Nous avons vu dans ce chapitre les codes équilibrés et les techniques pratiques, presqu'optimales, de leur construction réalisées dans les travaux de recherche antérieurs. Dans la même optique, nous essayons dans les travaux actuels de concevoir une technique de construction efficace, presqu'optimale. Nous visons la fermeture de l'écart de la redondance tout en gardant une complexité raisonnable en temps et en espace. Notre méthodologie est présentée dans le chapitre suivant.

Chapitre 4

Méthode

Notre technique utilise une variété d'outils habituels et nouveaux. Dans ce chapitre, nous introduisons pas à pas les notations, les définitions, les concepts et les algorithmes nécessaires. Dans la section 4.1, nous présentons les notations, les définitions et les concepts utilisés. Dans la section 4.2, nous présentons la programmation de la technique Pacman, les cycles de codage/décodage et leur manipulation, l'initialisation et la terminaison utilisées. Nous présentons au fur et à mesure le pseudocode utilisé.

4.1 Notions et définitions

4.1.1 Notation conventionnelle des permutations

En notation conventionnelle, nous représentons une permutation de m éléments sous la forme $(a_1, ..., a_m)$, où $a_i \neq a_j$ lorsque $1 \leq i < j \leq m$. Soit S un ensemble de taille m. Nous disons que $(a_1, ..., a_m)$ est une permutation des éléments de S (ou, par abus de langage, une permutation de S) si $\{a_1, ..., a_m\} = S$. Nous définissons $\mathcal{P}_m$ comme l'ensemble des permutations de $\{1, ..., m\}$. Dans ces travaux de recherche nous utilisons les permutations de l'ensemble $\mathcal{P}_{m \in \mathbb{N}}$.

Exemple : $\pi = (4, 1, 3, 2)$ est une permutation de $\mathcal{P}_4 = \{(1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4), (1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2), (2, 1, 3, 4), (2, 1, 4, 3), (2, 3, 1, 4), (2, 3, 4, 1), (2, 4, 1, 3), (2, 4, 3, 1), (3, 1, 2, 4), (3, 1, 4, 2), (3, 2, 1, 4), (3, 2, 4, 1), (3, 4, 1, 2), (3, 4, 2, 1), (4, 1, 2, 3), (4, 1, 3, 2), (4, 2, 1, 3), (4, 2, 3, 1), (4, 3, 1, 2), (4, 3, 2, 1)\}$.

4.1.2 Notation par index des permutations

Dans ce mémoire, nous adoptons aussi une autre représentation pour les permutations qu'on appelle notation par index. Cette représentation indexée indique la position relative de chacun des nombres qui apparaissent dans une permutation $\pi \in \mathcal{P}_m$. Soit $\pi_0 \in \mathcal{P}_m$ la permutation identité de $\mathcal{P}_m$. Pour représenter π en notation par index, nous déterminons la position de chaque élément a de π dans π_0. Ce processus se fait en trois étapes. Premièrement, nous

déterminons tous les éléments qui précèdent a (à gauche de a) dans π. Deuxièmement, nous les supprimons de π_0 pour avoir une nouvelle permutation π_0'. Finalement, nous déterminons le rang de a par rapport aux éléments de π_0'. Celle-ci change chaque fois qu'on veut déterminer la position d'un élément. D'où l'utilisation du terme « relatif » pour décrire les positions dans cette notation. Par abus de langage, nous disons que η est une permutation indexée si η est une permutation décrite en utilisant la représentation indexée. Nous représentons une permutation indexée η sous la forme $\langle \iota_m, ..., \iota_1 \rangle$, où $1 \leq \iota_\rho \leq \rho$ pour $1 \leq \rho \leq m$. ρ est la course de l'index ι_ρ. Par abus de langage, nous disons que ρ est la taille de l'index ι_ρ. Nous définissons $\mathcal{H}_m$ comme l'ensemble des permutations indexées de taille m. L'exemple de la figure 4.1 illustre bien le principe de cette notation. Dans cet exemple, nous avons la permutation $\pi = (4, 1, 3, 2)$, en notation conventionnelle, de taille 4. π_0 de $\mathcal{P}_4$ est la permutation $(1, 2, 3, 4)$. Initialement, la permutation indexée est $\eta = \langle \rangle$.

Le rang de la première valeur 4 de π dans π_0 est 4. Nous enlevons cette valeur de π_0 et nous ajoutons son rang à notre permutation indexée η. Le résultat est $\pi_0 = (1, 2, 3)$ et $\eta = \langle 4 \rangle$.
Le rang de la deuxième valeur 1 de π dans la nouvelle permutation π_0 est 1. Le résultat est $\pi_0 = (2, 3)$ et $\eta = \langle 4, 1 \rangle$.
Le rang de la troisième valeur 3 de π dans la nouvelle permutation π_0 est 2. Le résultat est $\pi_0 = (2)$ et $\eta = \langle 4, 1, 2 \rangle$.
Finalement, le rang de la dernière valeur est 1. La permutation indexée est donc $\eta = \langle 4, 1, 2, 1 \rangle$.

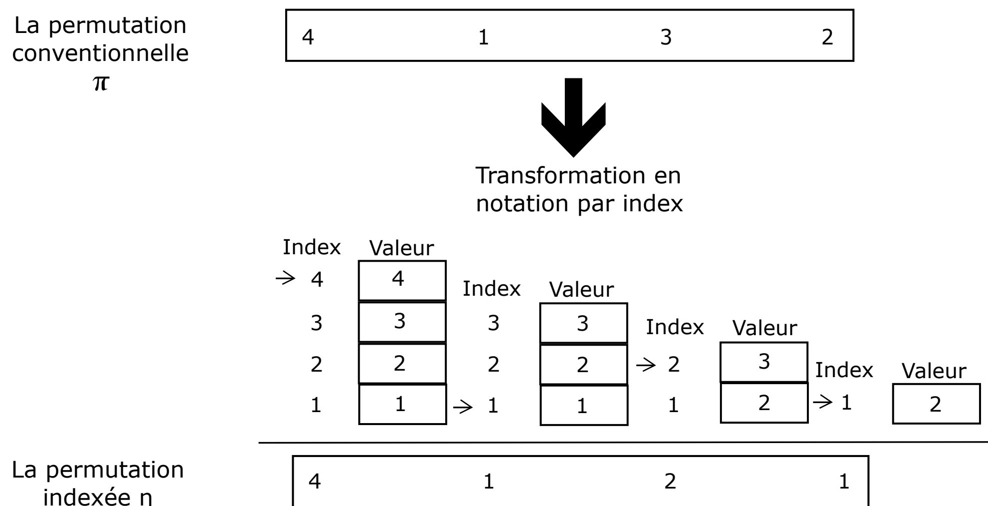

Figure 4.1 – Exemple de la notation par index

Les éléments d'une permutation indexée sont indépendants les uns des autres contrairement à une permutation conventionnelle : chez les permutations conventionnelles chaque élément doit être différent des autres éléments. Dans l'implémentation de notre technique, nous utilisons deux procédures pour faire les conversions d'une notation à l'autre (voir le pseudocode 1). La première procédure P fait la conversion en notation conventionnelle. La deuxième procédure

H fait la conversion en notation indexée.

Pseudo-code 1 Procédures de conversion de notation

```
1:  procedure P(η)
2:      m ← |η|
3:      π ← (), π_0 ← (1, 2, ..., m)
4:      for l ← (0 à m − 1) do
5:          a ← π_0(η(m − l) − 1)              ▷ (m − l) représente la taille ρ de l'index
6:          π ← π · (a)
7:          π_0 ← Remove(π_0, a)               ▷ Remove enlève a de π_0 et retourne le résultat
8:      end for
9:      return π
10: end procedure
11: procedure H(π)
12:     m ← |π|
13:     η ← ⟨⟩, π_0 ← (1, 2, ..., m)
14:     for l ← (0 à m − 1) do
15:         ι ← indexOf π(l) in π_0
16:         η(m − l) ← ι                       ▷ (m − l) représente la taille ρ de l'index ι
17:         π_0 ← Remove(π_0, π(l))            ▷ Remove enlève π(l) de π_0 et retourne le résultat
18:     end for
19:     return η
20: end procedure
```

Exemple : En appliquant respectivement H et P sur les deux permutations de la figure 4.1, nous avons : $\pi = (4, 1, 3, 2) = \mathrm{P}(\eta)$ *et* $\eta = \langle 4, 1, 2, 1 \rangle = \mathrm{H}(\pi)$.

4.1.3 Permutations et bloc équilibrés

Les permutations de $\mathcal{P}_{m \in 2\mathbb{N}}$ sont des permutations qui contiennent un nombre pair d'entiers. Aussi, ces permutations comportent $m/2$ nombres impairs et $m/2$ nombres pairs. Grâce à ces deux propriétés, nous sommes capables d'extraire les codes équilibrés à partir de ces permutations. En effet, l'extraction de la parité, sous la forme de 0 et de 1, des éléments d'une permutation $\Pi \in \mathcal{P}_{m \in 2\mathbb{N}}$ nous permet d'avoir un code équilibré B. Nous notons cette opération par $B = \Pi \ mod \ 2$. Par exemple, pour $\Pi = (4, 1, 3, 2)$, nous avons B = « 0 1 1 0 ». Ainsi, si nous parvenons à transformer entièrement les données d'entrée en permutations comme Π, nous aurons une procédure complète pour créer la fonction *Enc*. Notez que l'extraction de B, ne consomme pas la totalité de l'information contenue dans Π. En effet, Π est la description des positions des nombres impairs entre eux, des nombres pairs entre eux et des positions des nombres pairs par rapport aux nombres impairs. B contient juste la dernière information. Du point de vue du décodeur, la réception des deux autres parties d'information, qui restent, est nécessaire pour reconstruire Π. Soient $\pi, \pi' \in \mathcal{P}_{m/2}$ les permutations qui correspond respectivement à l'ordre relatif des nombre pairs et celui des nombres impairs. Dans l'exemple précédent, nous avons $\Pi = (4, 1, 3, 2)$ et $B = \Pi \ mod \ 2 =$ « 0 1 1 0 ». Les nombres pairs de Π

sont $(4, 2)$ qui, après une renumérotation monotone, équivaut à $\pi = (2, 1) \in P_2$. Les nombres impairs de Π sont $(1, 3)$ qui, après une renumérotation monotone, équivaut à $\pi' = (1, 2) \in P_2$. Le triplet (π, π', B) contient exactement la même information que Π. La relation entre Π et le triplet (π, π', B) est bijective. Cette transformation est essentielle dans la mise en œuvre de la fonction *Enc* de notre technique, qui se déroulerait en trois étapes. Premièrement, introduire des données d'entrée dans $\Pi \in P_m$. Deuxièmement, utiliser la transformée sur Π pour produire $B \in B_m$, qui va être émis sous forme de données codées, et $\pi, \pi' \in P_{m/2}$. Troisièmement, nous réutilisons les informations de π et π' pour construire une nouvelle permutation de P_m. Puisque la transformation peut facilement être inversée, elle peut aussi bien servir dans la mise en œuvre de *Dec*.

Nous avons vu que la transformation d'une permutation Π en un triplet (π, π', B) est bijective. Nous implémentons cette transformation en deux procédures. Nous appelons la première PB, pour dire la transformation d'une permutation à un code équilibré, et la deuxième BP pour signifier la procédure inverse.

Pseudo-code 2 Procédures de transformation entre Π et (π, π', B)

```
 1: procedure PB(Π)
 2:     π ← (), π′ ← (), B ← ϵ
 3:     for i in Π do                    ▷ for in boucle les elts de de gauche à droite
 4:         if i  mod 2 = 0 then
 5:             π  ← π · (i/2)
 6:             B  ← B · 0
 7:         else
 8:             π′  ← π′ · ((i + 1)/2)
 9:             B  ← B · 1
10:         end if
11:     end for
12:     return (π,π′,B)
13: end procedure
14: procedure BP(π, π′, B)
15:     Π ← ()
16:     for i in B do                    ▷ for in boucle les elts de de gauche à droite
17:         if i = 0 then
18:             Π  ← Π · (2 × π(0))
19:             π  ← Remove(π, π(0))
20:         else
21:             Π  ← Π · (2 × π′(0) − 1)
22:             π′  ← Remove(π′, π′(0))
23:         end if
24:     end for
25:     return Π
26: end procedure
```

La première procédure va nous permettre, pendant le codage, d'extraire le code équilibré pour

l'envoyer à la sortie. La deuxième va nous servir pendant le décodage pour reconstruire la grande permutation Π.

4.1.4 Reproduire une grande permutation à partir de petites permutations

Dans la notation conventionnelle, la réutilisation de l'information contenue dans les petites permutations $\pi, \pi' \in P_{m/2}$ pour reconstruire une nouvelle grande permutation $\Pi \in P_m$ n'est pas nécessairement évidente. Par contre, en notation indexée, la réutilisation de l'information contenue dans les permutations indexées $\eta, \eta' \in H_{m/2}$ pour produire une nouvelle grande permutation indexée $H \in H_m$ est plus évident. En effet, nous pouvons observer tout d'abord que η et η', ensemble, sont composés d'autant d'indices que H. En outre, les indices dans les permutations indexées sont indépendants les uns des autres. Nous proposons la procédure en trois étapes suivante pour reconstruire une grande permutation Π à partir des deux petites permutations π et π'. Tout d'abord, nous transformons π et π' en permutations indexées : $\eta = H_{m/2}(\pi)$ et $\eta' = H_{m/2}(\pi')$. Ensuite, nous produisons H à partir de η et η'. Enfin, nous transformons la permutation H en notation conventionnelle : $\Pi = P_m(H)$. Dans les sous-sections suivantes, nous expliquons comment effectuer la deuxième étape.

4.1.5 Pacman

Dans le présent travail, nous nous inspirons du célèbre personnage du jeu vidéo Pacman pour concevoir notre technique. Ce personnage consomme d'habitude des pilules pour gagner des points. En effet, notre algorithme est comme un Pacman spécial qui consomme et produit en même temps de l'information. À l'intérieur de notre Pacman, il y a plusieurs mécanismes. Nous avons vu le mécanisme d'extraction de codes équilibrés et celui de changement de notation. Nous allons voir dans cette sous-section le mécanisme qui permet l'incorporation des bits sources dans une permutation Π. Ce mécanisme peut être assimilé au cycle de transfert d'énergie ATP/ADP.

Comme nous l'avons vu dans la section 1.3, le cycle ATP/ADP est une voie métabolique universelle indispensable pour la continuation de la vie chez les êtres vivants. Cette voie métabolique permet de transférer l'énergie (p. ex. l'énergie solaire pour les plantes), sous une forme convenable (e.g. électrochimique), dans les cellules pour qu'elles réalisent les réactions chimiques nécessaires pour le maintien de la vie. Ce transfert d'énergie se fait par le moyen du coenzyme ATP. Au niveau cellulaire, ce coenzyme est considéré comme étant l'unité de transfert énergétique.

Le cycle d'ATP/ADP se divise en deux phases que nous appelons phosphorylation et hydrolyse. Pendant la phosphorylation, l'enzyme ATPsynthase permet de synthétiser le coenzyme ATP à partir de l'énergie issue des nutriments et de l'ADP. À l'aide du catalyseur ATPase,

l'hydrolyse de l'ATP permet de fournir l'énergie électrochimique à la cellule et de libérer l'ADP. Ce dernier peut être utilisé par la suite dans la construction d'une autre unité ATP.

De la même façon, pendant le cycle d'incorporation de données, Pacman consomme tous les index que η et η' contiennent et produit tous les index que contient Π, mais pas nécessairement dans l'ordre. Puisque deux petites permutations contiennent moins d'informations qu'une grande permutation, nous profitions de l'occasion pour que Pacman consomme, en plus des index des petites permutations, des bits sources. Globalement, pendant un cycle, Pacman doit consommer q bits d'entrée et tous les index de η et η' et doit produire tous les index de Π. La figure 4.2 illustre l'analogie entre le cycle de transfert d'énergie et le mécanisme d'incorporation des données source dans une grande permutation.

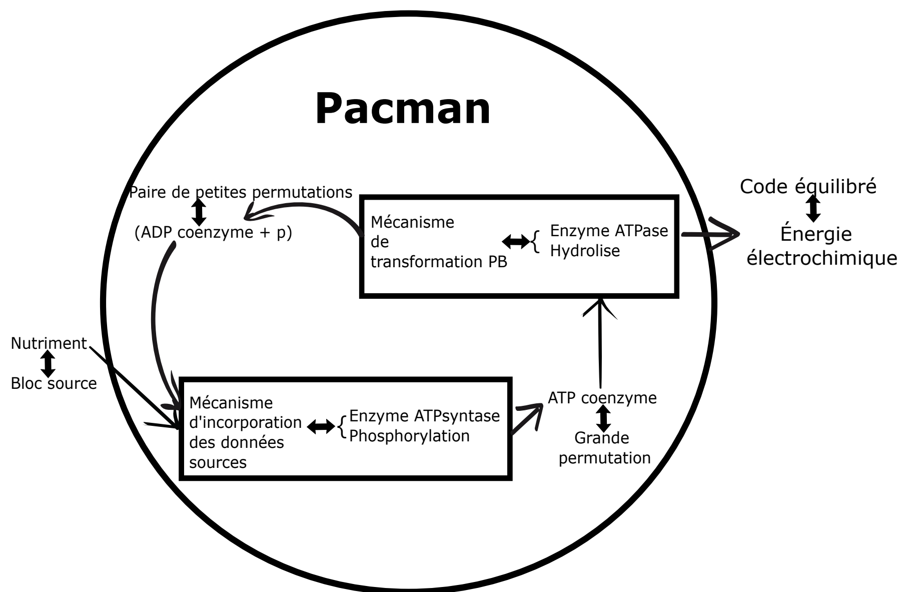

FIGURE 4.2 – L'analogie de la technique Pacman avec le cycle de transport d'énergie par l'ATP/ADP

Notez que l'objectif de notre technique consiste à transférer de l'information. Par conséquent, Pacman possède une mémoire. Lorsque ce dernier consomme un bit d'entrée ou un index, il gagne de l'information et nous disons que sa mémoire s'agrandit. D'autre part, quand il produit un index, il perd de l'information et nous disons que sa mémoire se rétrécit. Notez que, pour avoir un codage réversible (c'est-à-dire uniquement décodable), le transfert des données doit se faire sans perte d'information. Par conséquent, la consommation consécutive de plusieurs index élargit considérablement la mémoire de Pacman. Pour optimiser ce processus, Pacman doit donc alterner entre les consommations et les productions d'index d'une façon à ce que la taille de sa mémoire reste raisonnable.

4.1.6 Limitation de la mémoire de Pacman

Pour planifier et contrôler les opérations effectuées par Pacman, nous mesurons à chaque étape la taille de sa mémoire. Nous avons aussi adopté de nouveaux concepts. Nous considérons que la mémoire de Pacman est une petite mémoire. Nous disons qu'une mémoire est petite si elle ne peut contenir qu'une seule valeur à la fois. Nous mesurons la taille de la mémoire de Pacman d'une manière différente de ce qui est habituel. Nous considérons que la taille d'une mémoire qui peut contenir un nombre naturel dans la plage $1, ..., \sigma$, où $\sigma \geq 1$, est égale à σ. Habituellement, la taille d'une telle mémoire est mesurée comme étant $\log(\sigma)$ bits. Bien que notre définition soit directement liée à la définition habituelle ($\textit{taille conventionnelle} = \log(\textit{notre taille})$), notre définition nous convient mieux, comme nous le verrons plus loin.

Les opérations de Pacman ont l'effet suivant sur sa mémoire. Soient σ et σ' les tailles de la mémoire avant et après une opération donnée. Une opération de consommation d'un index de taille ρ agrandit la taille de la mémoire de Pacman jusqu'à la taille $\sigma' = \rho \times \sigma$. Une opération de production d'un index de taille ρ réduit la mémoire de Pacman jusqu'à la taille $\sigma' = \left\lceil \frac{\sigma}{\rho} \right\rceil$. L'arrondissement est nécessaire car σ n'est pas tout le temps divisible par ρ.

Les opérations de Pacman ont l'effet suivant sur la valeur de sa mémoire. Soient v et v' les valeurs de la mémoire avant et après une opération donnée. Lorsque Pacman consomme un index de valeur i et de taille ρ, nous obtenons :

$$v' = \rho \times (v - 1) + i.$$

Exemple : Les résultats de consommation d'index de taille $\rho = 3$ dans une mémoire de taille $\sigma = 4$ sont représentés dans le tableau 4.1.

Tableau 4.1 – Exemple de consommation d'index

v \ i	1	2	3
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

La paire (v, i) est associé injectivement à v'. En d'autres termes, on est capable de récupérer v et i à partir de v' si nous connaissons ρ. Ainsi, quand Pacman produit un index i de taille ρ, on obtient :

$$(i, v') = (((v-1) \mod \rho) + 1, \left\lceil \frac{v}{\rho} \right\rceil).$$

Exemple : Les résultats de production d'index de taille $\rho = 3$ à partir d'une mémoire de taille $\sigma = 12$ sont représentés dans le tableau 4.2.

Tableau 4.2 – Exemple de production d'index

v	1	2	3	4	5	6	7	8	9	10	11	12
v'	1	1	1	2	2	2	3	3	3	4	4	4
i	1	2	3	1	2	3	1	2	3	1	2	3

Les opérations ± 1 dans les calculs pour i ne sont que des ajustements pour la différence des plages d'index ou des valeurs de la mémoire $(1, ...)$ par rapport aux plages de la sortie de l'opérateur *mod* $(0, ...)$. Le lecteur aurait pu remarquer que nous n'avons pas mentionné explicitement l'effet de la consommation d'un bit d'entrée par Pacman. Nous choisissons de considérer la consommation d'un bit d'entrée comme étant la consommation d'un index de taille 2. Pour ce faire, nous ajustons la valeur du bit d'entrée en y ajoutant 1.

Nous remarquons que la consommation d'un index i qui associe (i, v) à v' n'introduit pas de redondance étant donné que la taille de la mémoire n'est pas arrondie ; $\sigma' = \rho \times \sigma$.

Par contre, la production d'un index i introduit la redondance étant donné que la taille de la mémoire est arrondie. En effet, la taille de la mémoire peut parfois être légèrement plus grande que nécessaire. C'est-à-dire que nous n'avons que $\sigma' \geq \frac{\sigma}{\rho}$ (voir l'exemple suivant). De plus, l'inégalité est stricte chaque fois que σ n'est pas un multiple de ρ, ce qui est le cas le plus fréquent. Ainsi, dans le cycle d'incorporation des données, les opérations de production d'index sont les seules qui peuvent introduire de la redondance.

Exemple : Supposons qu'on a une mémoire de taille $\sigma = 13$ et qu'on veut produire un index de taille $\rho = 3$. La taille actuelle de la mémoire n'est pas divisible par 3. La mémoire de Pacman doit donc être arrondie à $\sigma = 15$ pour pouvoir faire la production. En conséquence, les deux dernières valeurs de la mémoire gonflée vont être inutilisées. Les résultats sont dans le tableau 4.3.

Tableau 4.3 – Exemple de production d'index avec redondance

v	1	2	3	4	5	6	7	8	9	10	11	12	13	-	-
v'	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5
i	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3

En pire cas, la redondance ajoutée par les opérations de production d'index reste modeste étant donné que $\sigma' \leq \frac{\sigma+\rho-1}{\rho}$. Dans les situations les plus favorable, la redondance ajoutée converge vers zéro lorsque σ augmente ; $\lim_{\sigma\to\infty} \frac{\rho-1}{\sigma'} = 0$.

Cette observation est essentielle pour montrer l'efficacité de notre technique. Dans la sous-section 5.3, nous allons également calculer la valeur exacte de la redondance introduite pendant un cycle complet d'incorporation de données. Nous allons montrer aussi que celle-ci converge vers 0 quand σ augmente.

Nous implémentons les deux opérations en deux procédures. La première procédure nommée « CONS », pour consommation, permet de consommer un index i de taille ρ et de retourner la nouvelle taille σ' et la valeur v' de la mémoire. La deuxième procédure nommée « PROD », pour production, elle permet, à partir d'une valeur v' et d'une taille σ' de la mémoire, de produire un index i de taille ρ et de retourner une nouvelle valeur v et une nouvelle taille σ de la mémoire. Le pseudocode est comme suit.

Pseudo-code 3 Opérations de consommation et de production

1: **procedure** CONS(v, σ, i, ρ)
2: $\quad v' \leftarrow \rho \times (v-1) + i$
3: $\quad \sigma' \leftarrow \sigma \times \rho$
4: $\quad$ **return** (v', σ')
5: **end procedure**
6: **procedure** PROD(v, σ, ρ)
7: $\quad \sigma' \leftarrow \lceil \frac{\sigma}{\rho} \rceil$
8: $\quad v' \leftarrow \lceil \frac{v}{\rho} \rceil$
9: $\quad i \leftarrow ((v-1) \mod \rho) + 1$
10: $\quad$ **return** (v', σ', i)
11: **end procedure**

Nous avons vu dans cette section les différents mécanismes nécessaires à l'implémentation de l'algorithme de Pacman. Dans la section suivante, nous allons voir comment combiner ces différents outils pour avoir une programmation $\mathbb{P}$ valide.

4.2 La technique de Pacman

4.2.1 Programmation de l'algorithme de Pacman

Nous avons vu précédemment que Pacman produit une grande permutation indexée $H \in \mathcal{H}_m$ en utilisant l'information qui reste dans les deux petites permutations $\eta, \eta' \in \mathcal{H}_{m/2}$. En même temps, il incorpore les bits sources $b_1, ..., b_q$ dans cette nouvelle permutation. Ainsi, un cycle de transformation consiste à faire convertir les $\frac{m}{2}$ index $\eta_?$, les $\frac{m}{2}$ index $\eta'_?$, et les q bits $b_?$ frais en m index $H_?$. La figure 4.3 montre graphiquement la transformation de l'information dans un cycle de codage. Ceci se fait à l'aide d'une programmation $\mathbb{P}$ de Pacman qui, en

$(2 \times m + q)$ opérations, doit consommer $m + q$ index et produire m index. Une opération de consommation s'écrit $+\eta_i$ ou $+\eta'_i$ ou $+b_i$. Une opération de production s'écrit $-H_i$. $\mathbb{P}$ peut entremêler les consommations et les productions d'index sans restriction. Toutefois, chaque index à consommer doit être consommé une et une seule fois et chaque index à produire doit être produit une et une seule fois.

Exemple : Pour $m = 4$ et $q = 2$, une programmation $\mathbb{P}$ de Pacman est :

$$\mathbb{P} = (+\eta_2, +\eta'_1, +\eta_1, +\eta'_2, +b_1, +b_2, -H_4, -H_2, -H_3, -H_1).$$

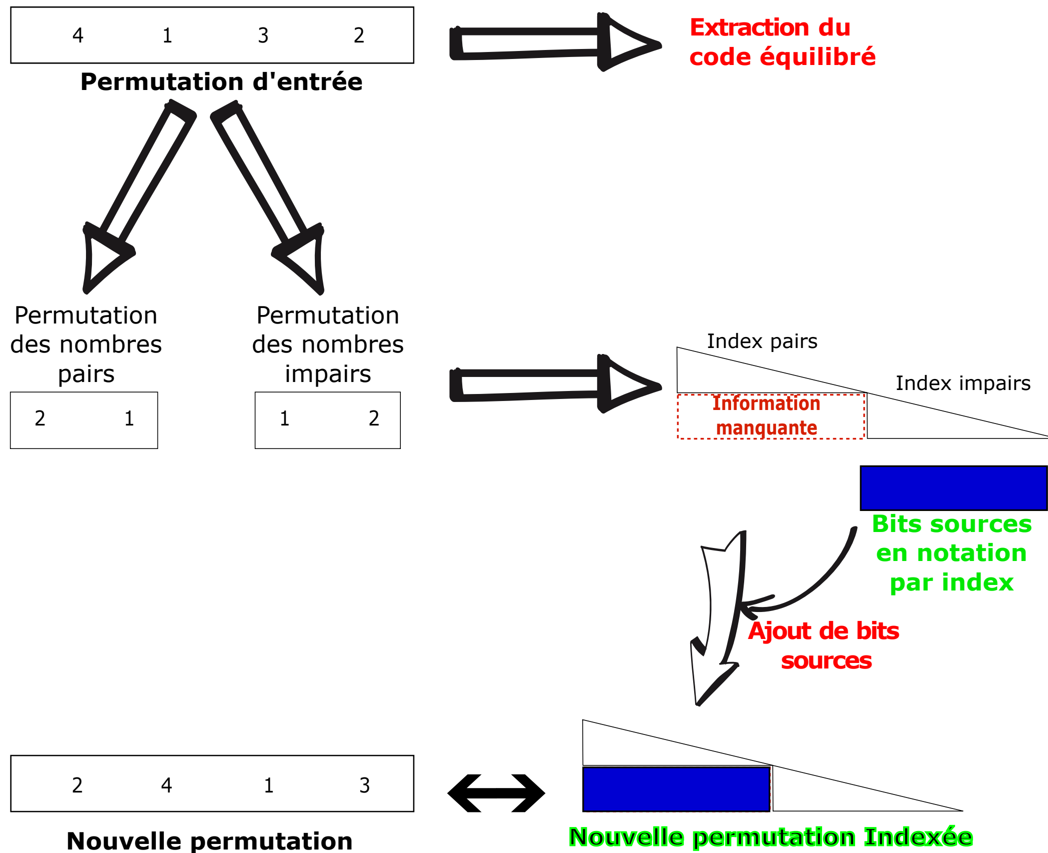

FIGURE 4.3 – Le principe de codage illustré en notation par index

Une information supplémentaire que nous devons attacher à $\mathbb{P}$ est σ_0. Celle-ci est la taille de la mémoire de Pacman au début de l'exécution de $\mathbb{P}$. À l'aide de σ_0, il est possible de déterminer la taille de la mémoire de Pacman à n'importe quelle étape de $\mathbb{P}$. Soit σ_i la taille de la mémoire après l'exécution des i premières instructions de $\mathbb{P}$. En particulier, la taille de mémoire à la fin de l'exécution de $\mathbb{P}$ est $\sigma_{2\times m+q}$.

4.2.2 Viabilité d'une programmation

Nous avons vu dans la sous-section 4.1.5 que la consommation consécutive de plusieurs index peut gonfler considérablement la mémoire de Pacman. L'alternance des opérations de consommation et de production va nous permettre de garder une taille raisonnable pour la mémoire de Pacman. Toutefois, il est nécessaire de déterminer le bon moment pour changer le type d'opération à réaliser. Pour cette raison, nous imposons une limite supérieure Ω à la taille de la mémoire. Chaque fois que nous voulons ajouter une nouvelle opération, on compare tout d'abord la taille actuelle σ_i à Ω. Donc, dans une programmation $\mathbb{P}$, la taille de la mémoire ne doit jamais dépasser Ω. En plus, pour pouvoir utiliser le même Pacman dans le codage de plusieurs séquences de bits, la taille de sa mémoire à la fin de chaque exécution doit être inférieure à la taille de la mémoire au début de la prochaine exécution. Pour résumer, une programmation $\mathbb{P}$ est valide si ses instructions sont arrangées de sorte que :

$$\forall\ 0 \leq i \leq 2 \times m + q.\ \sigma_i \leq \Omega \text{ et } \sigma_{2\times m+q} \leq \sigma_0$$

Par conséquent, si l'on choisit un q trop grand, alors on n'a aucun espoir d'établir une programmation valide. Plus précisement, il n'existe pas de $\mathbb{P}$ valide lorsque $q > \log \binom{m}{m/2}$ car cela signifierait que Pacman essaie d'intégrer plus de bits d'entrée que ce qu'un bloc équilibré peut contenir. Le choix d'un tel q entraînerait une augmentation de la taille de la mémoire de Pacman sans limites. Pour un $q \leq \log \binom{m}{m/2}$, il faut choisir un Ω suffisamment élevé pour pouvoir consommer la totalité des q bits.

Conjecture : Pour $m > 2$ et q adéquat, il existe un Ω suffisament grand pour qu'une programmation $\mathbb{P}$ valide existe.

Preuve : Expérimentalement, nous n'avons jamais rencontré de contradictions avec cette conjecture. Au mieux de nos connaissances, il devrait y avoir un une preuve de cette conjecture qui basée sur la théorie des nombres.

4.2.3 Cycle de codage

Nous définissons un cycle de codage comme étant les calculs qui provoquent la consommation d'un bloc de q bits d'entrée et la production d'un bloc équilibré. Puisqu'il peut y avoir plusieurs cycles en général, supposons que le cycle courant est le cycle $\#t$. En plus de la consommation d'un bloc d'entréee et la production d'un bloc équilibré en sortie, l'exécution d'un cycle de codage implique également la manipulation de données internes au codeur : la mémoire de Pacman et les deux petites permutations, π_{t-1} et π'_{t-1}. Ces données internes sont transmises d'un cycle à l'autre. Dans l'état où elle est laissée à la fin du cycle $\#t-1$, la mémoire de Pacman a une taille $\sigma_{2\times m+q}$ et contient une certaine valeur. Notez que l'indice dans $\sigma_{2\times m+q}$ n'est pas lié aux numéro de cycles : il indique seulement que l'exécution de $\mathbb{P}$ a été achevée dans le cycle $\#t-1$. La première étape du cycle $\#t$ consiste à changer la représentation des

petites permutations : $\eta_t = H_{m/2}(\pi_{t-1})$ et $\eta'_t = H_{m/2}(\pi'_{t-1})$. La deuxième étape du cycle $\#t$ consiste à redimensionner la mémoire de Pacman : la nouvelle taille est σ_0. Notez toutefois que la valeur dans la mémoire de Pacman reste inchangée. Cette étape est cohérente puisque $\sigma_{2\times m+q} \leq \sigma_0$. La troisième étape consiste à exécuter $\mathbb{P}$. Au cours de cette étape, Pacman consomme η_t et η'_t complètement, lit et consomme les q bits $b_1, ..., b_q$ d'entrée et produit H_t. La quatrième étape consiste à nouveau en un changement de représentation : $\Pi_t = P_m(H_t)$. La cinquième étape est une décomposition de Π_t en (π_t, π'_t, B_t). Dans la sixième étape, le bloc équilibré B_t est écrit en la sortie. Enfin, la mémoire de Pacman, π_t et π'_t sont transmises au cycle $\#t + 1$.

Note : Notez que l'objectif de notre algorithme n'est pas de compresser les données, mais de créer des codes équilibrés de taille minimale et en utilisant convenablement les ressources calculatoires. Notez aussi que nous considérons les données d'entrée uniformément denses en information. Ainsi, la programmation que nous appliquons dans chaque cycle de codage enchaine les mêmes opérations avec les mêmes conditions et la même consommation des ressources mémoires sur les données d'entrée.

4.2.4 Cycle de décodage

Le cycle de décodage $\#t$ inverse simplement les instructions du cycle de codage $\#t$. Nous supposons que la mémoire de Pacman, π_t et π'_t sont transmises par le cycle $\#t + 1$. (Notez que les cycles eux-mêmes doivent être inversés, pas seulement les instructions dans chaque cycle). La valeur dans la mémoire de Pacman, π_t et π'_t sont exactement les mêmes que celles qui existaient à la fin du cycle de codage $\#t$. Le bloc équilibré B_t est lu à partir de l'entrée. Le triplet (π_t, π'_t, B_t) est combiné pour reconstruire Π_t. Ensuite, il y a un changement de représentation : $H_t = H_M(\Pi_t)$. Puis, Pacman effectue une exécution inversée de $\mathbb{P}$. L'inversion signifie non seulement que les instructions sont exécutées de droite à gauche, mais aussi que les instructions qui ont déclenché la consommation dans le codeur déclenchent maintenant la production dans le décodeur et vice versa. L'exécution inversée de $\mathbb{P}$ produit η_t, η'_t et q bits. Les q bits sont écrits en sortie. Enfin, la mémoire de Pacman, $P_{m/2}(\eta_t)$ et $P_{m/2}(\eta'_t)$ sont passés au cycle $\#t - 1$.

4.2.5 Initialisation et terminaison

Maintenant que les cycles de codage et de décodage ont été décrits, nous avons presque une technique complète. Il reste à présenter le protocole d'initialisation et de terminaison pour le codeur. Pour le décodeur, il suffit simplement d'inverser les étapes du protocole du codeur.

L'initialisation doit préparer les données internes du codeur. Nous choisissons de l'initialiser avec des valeurs fixes. La mémoire de Pacman est initialisée à la valeur 1 et la taille $\sigma_{2\times m+q}$ (on suppose qu'il y a eu un cycle de codage antérieur #0 qui a donné ces résultats). Les permutations π_0 et π'_0 sont initialisées à la permutation identité, $(1, ..., m/2)$. On peut affirmer

qu'une telle initialisation à des valeurs fixes est plutôt un gaspillage. En effet, les premiers blocs équilibrés qui sont produits sont plus affectés par ces valeurs fixes que par les premiers blocs de bits d'entrée. C'est-à-dire que très peu d'information provenant des bits d'entrée est transférée dans les premiers blocs équilibrés. Néanmoins, cette procédure d'initialisation présente l'avantage d'être simple et, sur le codage d'une grande entrée, cette inefficacité finit par avoir un effet négligeable.

De la même façon pour la terminaison, on opte pour la simplicité et nous nous appuyons sur la longueur de l'entrée pour amortir l'inefficacité. Avant de commencer cette étape. Le codeur doit s'assurer d'avoir consommé tous les bits sources. En effet, parfois la taille de l'entrée n'est pas un multiple de q, la taille des blocs source codés. Par conséquent, le codeur doit ajouter de 0 à $q-1$ bits de remplissage au dernier bloc source. Le nombre de bits ajoutés est donc une information qui doit être transmise au décodeur pour assurer la réversibilité de l'encodage. Aussi le décodeur a besoin des informations internes de l'encodeur à la fin du cycle $\#t_{final}$ pour réaliser les opérations de $\mathbb{P}$ en ordre inverse. Le codeur doit pouvoir transmettre toutes ces informations sous forme de blocs équilibrés. Pour ce faire, nous utilisons l'algorithme de Knuth. Pour coder les préfixes, nous utilisons un codage énumératif appelé « PopCount » [7]. Ce codage permet de générer la liste de toutes les permutations d'une série de bits dans un ordre précis tout en choisissant le nombre de bits à 1 et de bits à 0. Ce calcul se fait rapidement : pendant un nombre fini d'itérations (l'utilisation de ce codage énumératif est juste pour coder les préfixes). Avec une longue chaîne d'entrée, l'inefficacité introduite par le codage de la terminaison devient négligeable. Dans la première étape de la terminaison, l'encodeur doit donc émettre un bloc équilibré, de taille m, pour décrire la longueur de remplissage. Dans la deuxième étape, il émet un bloc équilibré, de taille m, pour décrire la valeur finale de la mémoire de Pacman. Enfin, pour chaque élément des deux dernières petites permutations, il émet un bloc équilibré de taille m.

En sachant m et q, nous sommes capable de déduire la taille des blocs de terminaison et ainsi, extraire les données nécessaires pour faire le décodage.

Pour conclure, la figure 4.4 illustre la succession des cycles de codage, la transition et la succession des cycles de décodage dans l'algorithme de Pacman.

4.2.6 La conception de la programmation

Nous avons vu dans les sous-sections précédentes ce qu'est une programmation de Pacman et ce que sont les conditions de sa validité. Nous expliquons dans cette sous-section les configurations adoptées pour concevoir une programmation de Pacman.

Nous avons vu qu'une programmation qui consomme successivement plusieurs index nécessite la manipulation de grands nombres entiers étant donné que la mémoire de Pacman doit prendre

7. Source : http ://alexbowe.com/popcount-permutations/

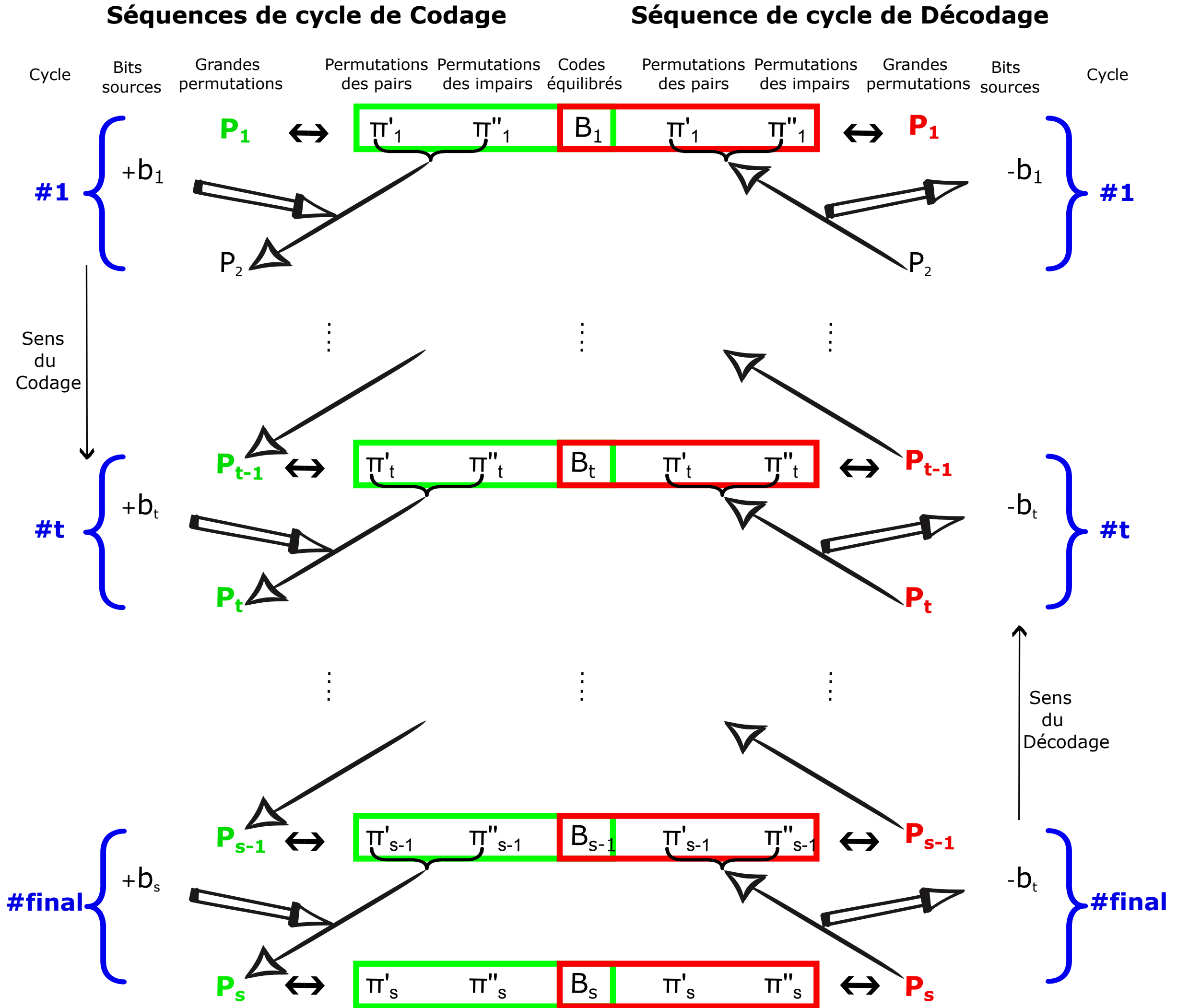

FIGURE 4.4 – Illustration des séquences de codage et de décodage

de l'expansion dans ce cas jusqu'à $(\frac{m}{2}!)^2 \times 2^q \times \sigma_0$. Pour éviter ce cas coûteux en ressources, nous imposons une borne Ω à la taille de la mémoire. Cette limite Ω dépend du nombre de bits source q encodés. Dans chaque bloc, la conception d'une programmation optimale nécessite donc d'avoir un contrôle sur ces paramètres. Cependant, nous n'avons pas formellement analysé la complexité de la conception d'une programmation, mais nous pensons qu'elle est NP-difficile. En effet, il ne semble pas facile de concevoir une programmation optimale, pour une définition raisonnable d'optimalité. En effet, on peut considèrer différentes définitions du mot « optimal ». Dans nos expériences, nous adoptons tour à tour deux définitions de l'optimalité. Dans chacune des définitions, nous considérons que m est imposé par l'application qui requiert des codes équilibrés. La première définition consiste à choisir $q \leq \log\binom{m}{m/2}$ a priori et à essayer de déterminer Ω_{min}, qui est le plus petit Ω pour lequel il existe une programmation valide $\mathbb{P}_\Omega$. La seconde consiste à choisir a priori Ω et $\sigma_0 \leq \Omega$ et essayer de déterminer q_{max}, qui est le

plus grand q pour lequel il existe une programmation valide $\mathbb{P}_{q_{max}}$.

Nous n'avons pas essayé de trouver les valeurs absolument optimales sous aucune des deux définitions. Au lieu de cela, nous utilisons deux heuristiques.

Dans la première heuristique, nous choisissons à priori m, q, σ_0 et Ω et nous tendons de créer une programmation vorace. À cette fin, nous plaçons les instructions de consommation et de production dans deux files d'attente séparées. Dans la file d'attente des consommations, nous favorisons la consommation des index pairs, ensuite les index impairs et enfin les bits sources. Les index dans les deux files sont mis dans un ordre décroissant : $+\eta_{m/2}, ..., +\eta_1, +\eta'_{m/2}, ..., +\eta'_1,$ $+ b_1, ..., +b_q$ et $-H_m, ..., -H_1$. Ensuite, un processus itératif sélectionne une nouvelle instruction à la fois à partir d'une des files d'attente, préférant les instructions de consommation à celles de production chaque fois que cela est possible et favorisant les premières instructions qui permettent à la taille de la mémoire de Pacman de rester inferieure à Ω.

Cette heuristique va nous permettre, pour un choix à priori de q et de σ_0, de déterminer Ω_{min} et $\mathbb{P}_{\Omega_{min}}$. En effet, nous réalisons des tests de construction de programmation $\mathbb{P}$ pour plusieurs valeurs de Ω. Notre objectif est d'avoir une programmation valide pour le plus petit Ω possible. Pour un choix raisonnable de q et σ_0, nous déterminons la plus petite valeur possible de Ω. Nous créons une programmation en fonction des paramètres choisis. La programmation est valide si elle vérifie les conditions de validité et elle réalise toutes les $(2 \times m + q)$ opérations demandées. Sinon, nous augmentons la valeur de Ω et nous tentons à nouveau à créer une programmation. L'itération de ce processus se fait jusqu'à ce que notre objectif soit atteint. Notez qu'une programmation valide existe toujours sous les conditions que nous avons décrites (voir la sous-section 4.2.2). Les opérations de la programmation résultante vont être utilisées (après optimisation) dans les fonctions *Enc* et *Dec*.

Nous avons créé une procédure nommée $MakeGreedyProg$ qui permet, pour Ω, σ_0, q et m données, de tenter de construire une programmation valide. Cette procédure utilise deux files d'attente. La première, nommée CT, contient les $m + q$ opérations de consommation. La deuxième, nommée PT, contient les m opérations de production. En cas de succès, la procédure retourne la programmation construite. Les opérations de cette programmation sont mises dans une liste appelée $\mathbb{P}$. Chaque élément *Op* de cette liste contient la taille de la mémoire à la fin de l'opération et la taille de l'index (produit ou consommé). Une deuxième procédure appelée $POmin$ utilise $MakeGreedyProg$ pour trouver Ω_{min} et $\mathbb{P}_{\Omega}$. Nous choisissons dans cette procédure des Ω égaux à m^x, où x est un entier dans l'intervalle $[2, 200]$. Ce choix est pratique vu que nous manipulons des blocs équilibrés de taille m. Cela va nous permettre aussi d'optimiser le temps de calcul de construction de la programmation.

La seconde heuristique repose sur un choix a priori de σ_0 et sur une programmation $\mathbb{P}$ déjà établie pour un q et un Ω fixés aussi a priori. L'heuristique envisage d'échanger deux instructions arbitraires dans $\mathbb{P}$ et vérifie si cela réduirait la taille de la mémoire de Pacman au

Pseudo-code 4 Procédures de la 1e heuristique

```
1: procedure MakeGreedyProg(Ω, σ_0, m, q)
2:     σ ← σ_0
3:     ℙ ← ()
4:     PT ← (−H_1, ..., −H_m) , CT ← (+η_1, ..., +η_{m/2}, +η'_1, ..., +η'_{m/2}, +b_1, ..., +b_q)
5:     OC ← 0
6:     while OC < (2 × m + q) do
7:         if ∃ +η_ρ ∈ CT such that (ρ × σ) ≤ Ω then
8:             v, σ ← Cons(1, σ, 1, ρ)
9:             CO ← CO · (["+η_ρ", σ, ρ])
10:            CT ← Remove(CT, "+η_ρ")
11:        else if ∃ +η'_ρ ∈ CT such that (ρ × σ) ≤ Ω then
12:            v, σ ← Cons(1, σ, 1, ρ)
13:            CO ← CO · (["+η'_ρ", σ, ρ])
14:            CT ← Remove(CT, "+η'_ρ")
15:        else if ∃ +b_j ∈ CT And (2 × σ) ≤ Ω then
16:            v, σ ← Cons(1, σ, 1, 2)
17:            CO ← CO · (["+b_j", σ, 2])
18:            CT ← Remove(CT, "+b_j")
19:        else if ∃ −H_ρ ∈ PT such that (⌈σ/ρ⌉ × ρ) ≤ Ω then
20:            v, σ, ι ← Prod(1, σ, ρ)
21:            CO ← CO · (["−H_ρ", σ, ρ])
22:            PT ← Remove(PT, Op)
23:        else
24:            return FAILURE()
25:        end if
26:        OC ← OC + 1
27:    end while
28:    if CT = ∅ And PT = ∅ then
29:        return SUCCESS(ℙ)
30:    else
31:        return FAILURE()
32:    end if
33: end procedure
34: procedure POmin(m, q, σ_0)
35:    for x ← (2 à 200) do
36:        Ω ← m^x
37:        if MakeGreedyProg(Ω, σ_0, m, q) = SUCCESS(ℙ_Ω) then
38:            return (Ω, ℙ_Ω)
39:        end if
40:    end for
41: end procedure
```

moins à certaines étapes, en particulier aux étapes où les tailles de la mémoire sont maximales. L'inversion des opérations se fait jusqu'à ce que la réduction devienne impossible.

Pour appliquer la deuxième heuristique, nous créons une programmation $\mathbb{P}_\Omega$ pour un σ_0 et un q fixés a priori à l'aide de *POmin*. Nous procédons par la suite à l'inversion d'opérations jusqu'à ce que les tailles maximales soient réduites. Nous vérifions par la suite si la programmation résultante est valide. Si la programmation n'est plus valide, nous nous arrêtons à ce niveau et nous envoyons la programmation précédente. Sinon, nous poursuivons le processus d'amélioration. Nous diminuons la taille de Ω à $max(\sigma_i)$, où $i \in \{0, 1, 2, ..., 2 \times m + q\}$, et nous recommençons l'inversion d'opérations. La répétition du processus se fait jusqu'à ce que la diminution de Ω_{min} devienne impossible. Nous utilisons la procédure *Optimize* pour effectuer d'optimisation.

Pseudo-code 5 Procédure de la 2e heuristique

1: **procedure** OPTIMIZE(m, q, Ω_{min}, σ_0, $\mathbb{P}_\Omega$)
2: $x \leftarrow \lceil \log_m(\Omega_{min}) \rceil$
3: $opt\mathbb{P}_\Omega \leftarrow \mathbb{P}_\Omega$
4: $tmp\mathbb{P}_\Omega \leftarrow \mathbb{P}_\Omega$
5: **for** $i \leftarrow (x$ à $2)$ **step** -1 **do**
6: $counter \leftarrow 0$
7: **while** $(\exists\, Op \in tmp\mathbb{P}_\Omega$ ***such as*** $\sigma_{Op} \geq m^i)$ **And** $(counter \leq 2m + q)$ **do**
8: **if** Op **has** $NextOp$ **then** ▷ Vérifier qu'il existe une opération $NextOp$ qui suit Op pour faire l'inversion
9: $tmp\mathbb{P}_\Omega \leftarrow$***Switch***($Op$, $NextOp$) ▷
10: $counter \leftarrow counter + 1$
11: **end if**
12: **end while**
13: **if** $\sigma_{(2\times m+q)_{tmp\mathbb{P}_\Omega}} > \sigma_0$ **then**
14: **break** ▷ Interrompre de la boucle for
15: **end if**
16: $opt\mathbb{P}_\Omega \leftarrow tmp\mathbb{P}_\Omega$
17: **end for**
18: **return** $opt\mathbb{P}_\Omega$
19: **end procedure**

La programmation résultante va être utilisée pour faire le codage. Pour le décodage, il suffit d'inverser les opérations du codage comme nous l'avons mentionné précédemment. Pour cela, nous créons les deux procédures *Enc* et *Dec* qui utilisent les opérations $opt\mathbb{P}_\Omega$ de la nouvelle programmation.

Pseudo-code 6 Procédures de codage et de décodage

```
1: procedure Enc(v_0, σ_0, π, π', BS, optℙ_Ω)                              ▷ BS : bloc source
2:     m ← 2 × Size(π)
3:     v, σ ← (v_0, σ_0)
4:     η ← H(π), η' ← H(π'), BS_I ← Transfer BS by 1, H ← ⟨⟩
5:     for Op in optℙ_Ω do                      ▷ for in boucle les elts de de gauche à droite
6:         if Op is " + η_ρ" then
7:             v, σ ← Cons(v, σ, ρ, η(ρ))
8:         else if Op is " + η'_ρ" then
9:             v, σ ← Cons(v, σ, ρ, η'(ρ))
10:        else if Op is " + b_j" then
11:            v, σ ← Cons(v, σ, 2, BS_I[j])
12:        else if Op is " − H_ρ" then
13:            v, σ, H(ρ) ← Prod(v, σ, ρ)
14:        else
15:            return FAILURE()
16:        end if
17:    end for
18:    Π ← P(H)
19:    π, π', BC ← PB(Π)
20:    return SUCCESS(v, π, π', BC)
21: end procedure
22: procedure Dec(v_0, σ_0, π, π', BC, optℙ_Ω)          ▷ BC : code équilibré (balanced code)
23:    m ← Size(BC)
24:    v, σ ← (v_0, σ_0)
25:    Π ← BP(π, π', BC)
26:    H ← H(Π)
27:    η ← ⟨⟩, η' ← ⟨⟩
28:    optℙ_Ω ← Reverse(optℙ_Ω) ▷ Inverser les opérations du codage pour faire le décodage.
29:    for Op ∈ OptDCO do                       ▷ for in boucle les elts de de gauche à droite
30:        if Op is " + η_ρ" then
31:            v, σ, η(ρ) ← Prod(v, σ, ρ)
32:        else if Op is " + η'_ρ" then
33:            v, σ, η'(ρ) ← Prod(v, σ, ρ)
34:        else if Op is " + b_j" then
35:            v, σ, I ← Prod(v, σ, 2)
36:            BS ← append(I − 1)
37:        else if Op is " − H_ρ" then
38:            v, σ ← Cons(v, σ, ρ, H(ρ))
39:        else
40:            return FAILURE()
41:        end if
42:    end for
43:    π ← P(η), π' ← P(η')
44:    return SUCCESS(v, π, π', BS)
45: end procedure
```

Dans ce chapitre, nous avons présenté les notations, les définitions, les concepts et les algorithmes utilisés dans notre technique. Nous avons présenté aussi les différentes étapes nécessaires pour concevoir une programmation de Pacman et les différents modèles que nous adoptons pour une conception optimale. Les résultats expérimentaux et théoriques de notre implémentation vont être présentés dans le chapitre suivant.

Chapitre 5

Résultats

Dans ce chapitre, nous présentons nos résultats expérimentaux et théoriques. Nous utilisons deux modes dans nos applications expérimentales. Nous les présentons successivement dans les sections 5.1 et 5.2. Dans le premier mode, nous choisissons d'avoir la redondance minimale et donc d'encoder le plus grand nombre q de bits sources possible par bloc. Dans le deuxième, nous choisissons de limiter la taille des entiers que notre technique peut utiliser dans les calculs ; c'est-à-dire que Ω est fixé arbitrairement. Dans la section 5.3, une analyse théorique de la redondance est effectuée pour le deuxième mode. Cette analyse consiste à calculer une borne supérieure sur la redondance possiblement introduite dans le cas d'une telle programmation de Pacman.

5.1 Mode de redondance minimale

Dans ce mode, nous choisissons q maximal ; c'est-à-dire $q = \left\lfloor \log \binom{m}{m/2} \right\rfloor$, pour différentes valeurs de m, et nous observons l'effet sur Ω_{min}. L'une des conditions de validité que nous avons imposées est $\sigma_{2\times m+q} \leq \sigma_0$. Ainsi, choisir un σ_0 suffisamment grand permet à la programmation de jouir de plus de liberté. Ce doit être ainsi pour n'importe quelle valeur de m. La taille σ_0 doit donc être une fonction de m tout en restant convenable pour les registres machines. Pour les valeurs de m inférieures à 64, nous avons remarqué que la programmation nécessite plus de liberté. Après plusieurs expérimentations, nous avons choisi de fixer σ_0 à $\max(64, 2^{\lceil \log m \rceil})$. La conception de la programmation se fait en utilisant nos deux heuristiques. La première courbe de la figure 5.1 montre les résultats. Nous observons que Ω_{min} a une tendance générale à croître en douceur à l'exception de quelques pics. Les pointes vers le haut sont celles qui sont moins souhaitables parce qu'elles nécessitent la manipulation de plus grands nombres entiers dans les calculs. La deuxième courbe dans la même figure nous montre l'origine de cet effet. Cette courbe représente le rapport du nombre total des blocs équilibrés de taille m par rapport au nombre total des blocs sources de taille q qui est $\binom{m}{m/2}/2^q$. Ce rapport évolue en dents de scie. Quand ce dernier se rapproche de 1, la marge de manœuvre devient de plus en

plus rétrécie ce qui rend la manipulation des grands entiers nécessaire. À part ces cas, le taux de croissance de Ω_{min} semble être polynomial, car on peut estimer approximativement que Ω_{min} est multiplié par 10 chaque fois que m double.

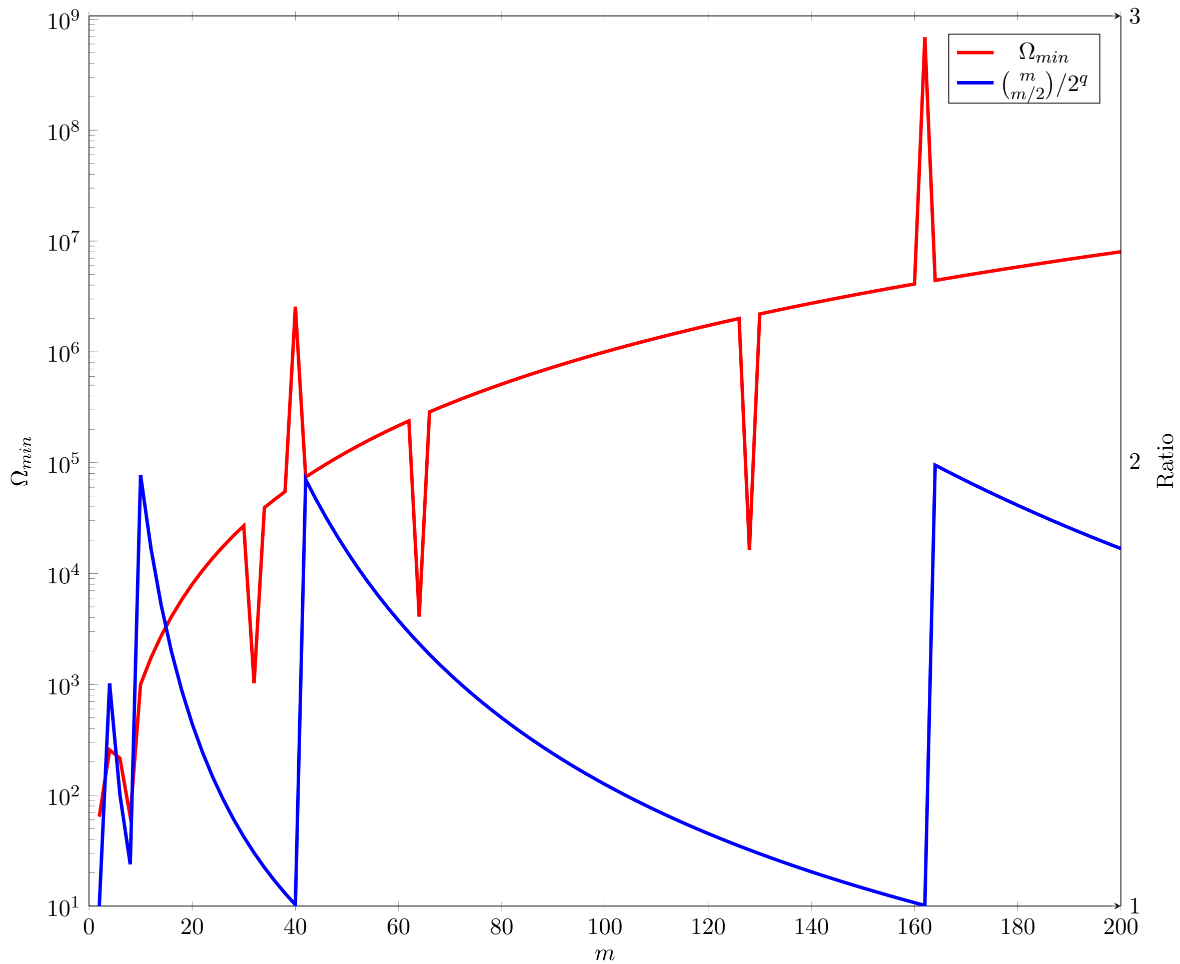

FIGURE 5.1 – La variation de Ω_{min} et du rapport $\binom{m}{m/2}/2^q$ pour différentes valeurs de m

Nous avons choisi d'implémenter ce mode dans une application informatique qui fait le codage et le décodage d'un fichier donné. Nous l'avons réalisé à l'aide du langage de programmation JAVA. Cette application permet à l'utilisateur de choisir la fonctionnalité qu'il veut utiliser au menu principal (codage ou décodage) (voir la figure 5.2). Ensuite, elle lui permet de choisir le fichier source, le fichier de destination et la taille des blocs équilibrés (voir la figure 5.3). Une fois lancée, l'application calcule σ_0 et Ω_{min} et crée les opérations de la programmation. Après la réalisation de la fonctionnalité demandée, elle affiche les valeurs de σ_0 et de Ω_{min} et la programmation établie. Un exemple de codage et de décodage d'un fichier est présenté dans les figures 5.4, 5.5, 5.6 et 5.7.

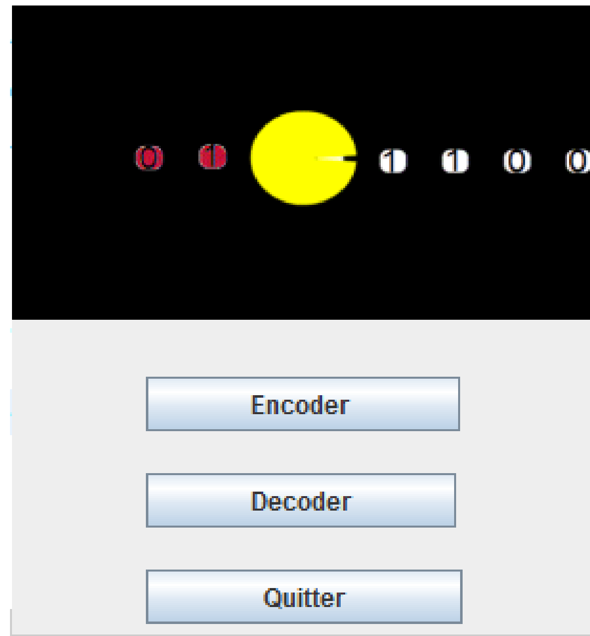

FIGURE 5.2 – L'interface principale de l'application

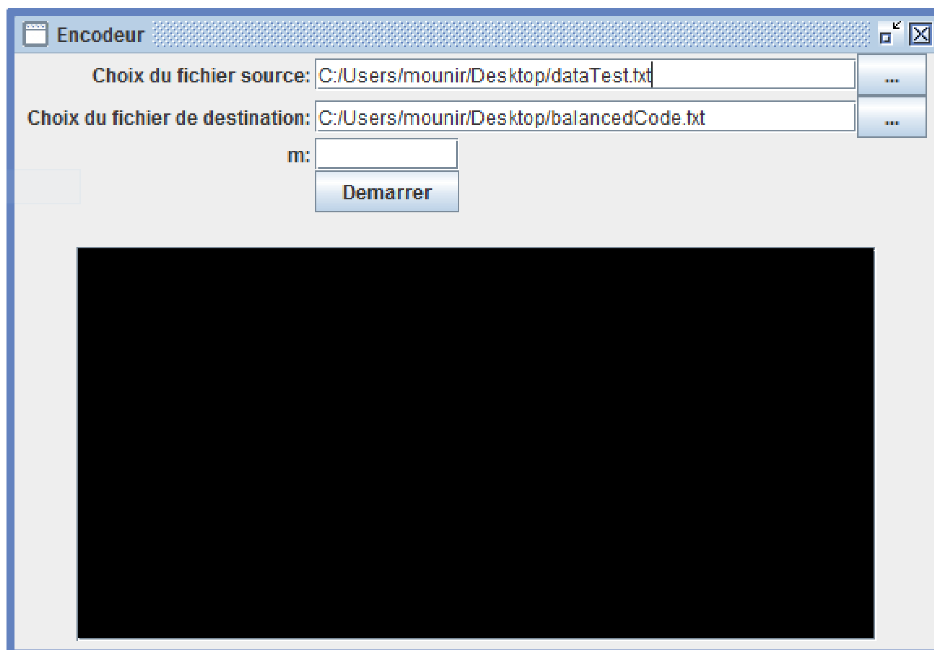

FIGURE 5.3 – L'interface de codage de l'application

Note : L'interface de décodage est similaire à celle du codage.

Voici un exemple de codage de fichier : « dataTest.txt »

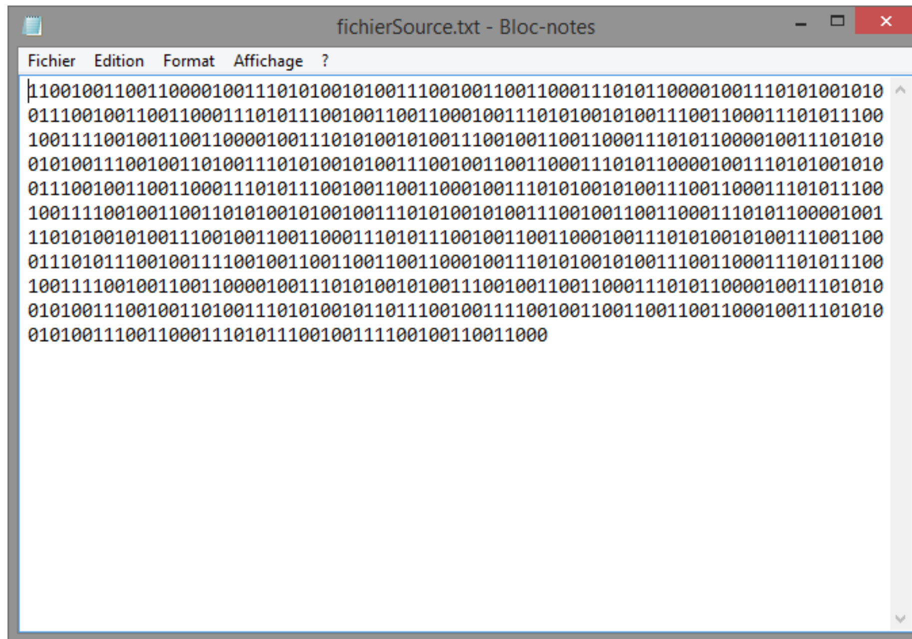

FIGURE 5.4 – Fichier source "`fichierSource.txt`"

Après avoir entré le chemin du fichier source, le fichier de destination du codage et du décodage et la taille m des blocs équilibrés, on démarre l'opération de codage ou de décodage.

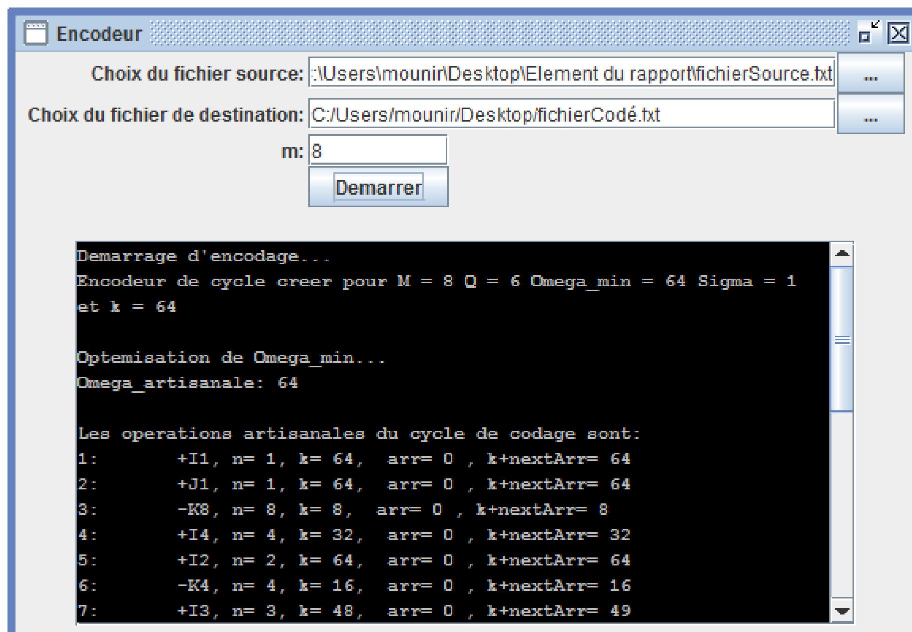

FIGURE 5.5 – Démarrage de l'opération de codage

Le résultat du codage donne le fichier suivant : « fichierCodé.txt »

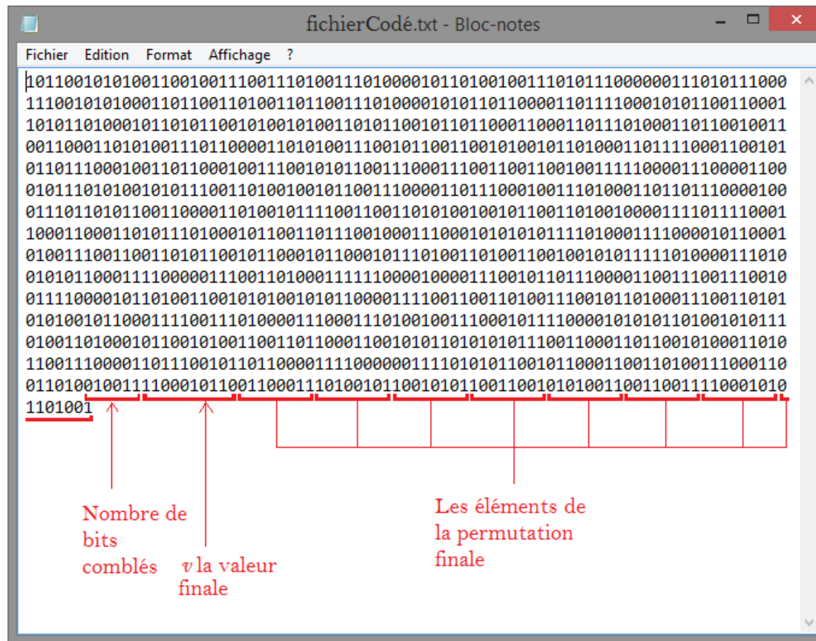

FIGURE 5.6 – Fichier codé "`fichierCodé.txt`"

Le résultat du décodage donne le fichier « fichierDécodé.txt » qui est identique au fichier « fichierSource.txt ».

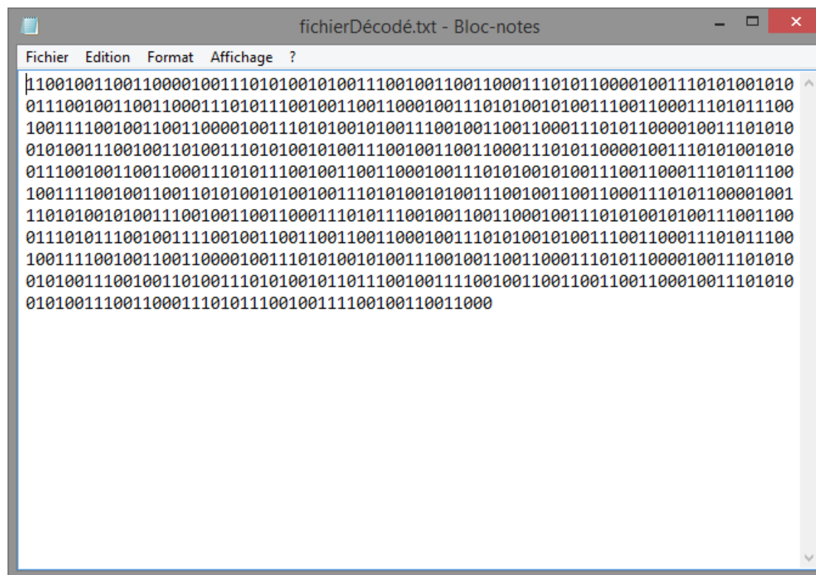

FIGURE 5.7 – Fichier décodé "`fichierDécodé.txt`"

5.2 Mode pour les nombres entiers à précision limitée

Dans ce mode, nous fixons arbitrairement Ω et m à différentes valeurs et nous observons l'effet sur la redondance, c'est-à-dire l'effet sur p_{min} ($p_{min} = m - q_{max}$). Nous définissons Ω comme une fonction de m. Principalement, nous utilisons dans notre implémentation des registres de taille $\log m^x$, où $x \in \mathbb{N}^* \setminus \{1\}$, c'est-à-dire $\Omega = m^x$. Nous traitons des blocs équilibrés de taille m, donc l'utilisation de registres de taille au moins égale à $\log(m)$ est obligatoire. Nous verrons par la suite dans l'analyse théorique de la redondance la raison pour laquelle x doit être supérieur à 1. Dans nos expériences, nous utilisons plusieurs valeurs de Ω en fonction de m. Par exemple, m^2, m^3, m^4 et m^5. Nous présentons les résultats uniquement pour les valeurs m^2, m^3 (voir la figure 5.8). Les autres valeurs de Ω n'améliorent pas p étant donné que le p optimal est virtuellement atteint pour $\Omega = m^3$. Ainsi, l'utilisation de plus grandes valeurs de Ω va demander plus de calcul. Nous traçons la parité p pour σ_0 égale à 1, 64 et $\max(64, 2^{\lceil \log m \rceil})$. Nous remarquons que la valeur de σ_0 influence aussi la parité surtout pour les petites valeurs de Ω comme m^2. Plus σ_0 est raisonnablement grand plus la parité est minimale.

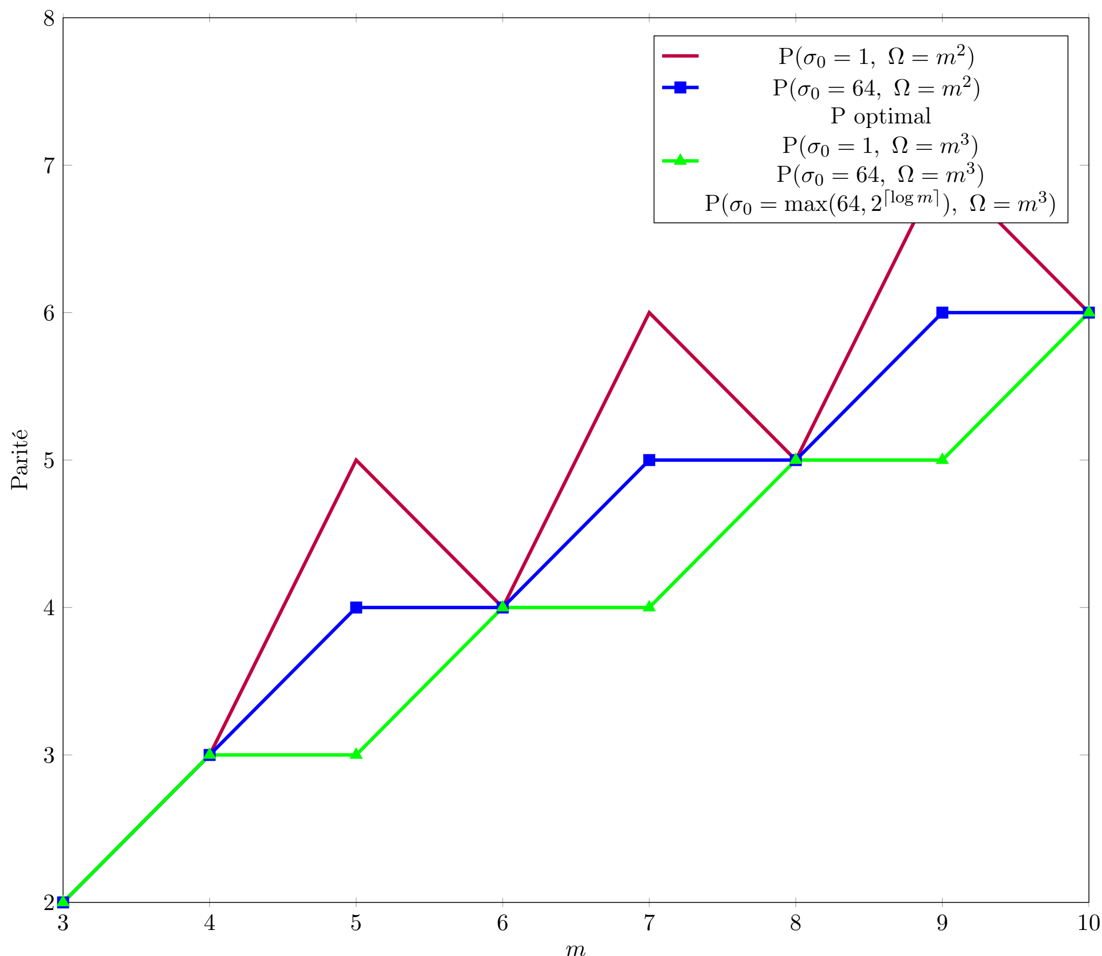

FIGURE 5.8 – Le tracé de la redondance pour différentes valeur de m, Ω_{min} et σ_0

Note : Nous avons tracé une seule courbe de redondance pour représenter la courbe de P optimal, la courbe de P($\sigma_0 = 1,\ \Omega = m^3$), la courbe de P($\sigma_0 = 64,\ \Omega = m^3$) et la courbe de P($\sigma_0 = \max(64, 2^{\lceil \log m \rceil}),\ \Omega = m^3$). Visuellement, c'est plus clair étant donné que ces courbes se superposent.

5.3 Calcul théorique des bornes supérieures de la redondance

Dans cette section, nous calculons théoriquement la redondance pour $\Omega = m^x$, où $x \geq 2$, dans la situation la plus défavorable que nous pouvons rencontrer. Nous supposons le pire des cas dans chaque opération de production d'index (une opération de consommation d'index n'introduit aucune redondance). Ainsi, nous supposons que la mémoire de Pacman est à sa plus petite taille avant d'exécuter une instruction de production. Si nous utilisons l'heuristique vorace pour concevoir une programmation $\mathbb{P}$, nous pouvons supposer que la taille de la mémoire de Pacman ne peut être inférieure à Ω que par un facteur m (autrement, une opération de consommation pourrait être effectuée au lieu d'une opération de production). Nous considérons donc que la taille de la mémoire de Pacman avant une production d'index ne peut être inférieure à Ω/m. Pour montrer que la redondance introduite par Pacman converge vers le seuil théorique ($p \approx \frac{1}{2}\log m + 0.326, m \gg 1$), nous calculons aussi les bornes supérieures de la redondance de $\mathbb{P}$, sous les mêmes conditions. Nous supposons en plus que l'arrondissement de la taille de la mémoire entraine toujours un ajustement maximal. C'est-à-dire la production d'un index de taille ρ va toujours causer l'ajout de $\rho - 1$ à la taille de la mémoire avant de faire la division par ρ.

Soient $k_1, k_2, ..., k_m$ les positions des instructions de production dans $\mathbb{P}$, où $k_1 < k_2 < ... < k_m$ et $\rho_1, \rho_2, ..., \rho_m$ les tailles des index consommés dans les positions $k_1, k_2, ..., k_m$.

La production d'un index de taille ρ_i implique que $\sigma_{k_i} = \lceil \frac{\sigma_{k_i - 1}}{\rho_i} \rceil$, où $\sigma_{k_i - 1} \geq \Omega/m = m^{x-1}$.

La redondance introduite par la production d'un index en bits est : $\log(\frac{\sigma_{k_i}}{(\sigma_{k_i-1}/\rho_i)})$.

Le cumul de la redondance R introduite par la production d'index pendant un cycle de codage est égal à $\log(\prod_{i=1}^{m} \frac{\sigma_{k_i}}{(\sigma_{k_i-1}/\rho_i)})$.

Donc $R = \sum_{i=1}^{m} \log(\frac{\sigma_{k_i}}{(\sigma_{k_i-1}/\rho_i)})$.

Supposons qu'il existe $l_i \in [0, \rho_i - 1]$ qui dépend de σ_{k_i-1} et ρ_i tel que $\sigma_{k_i} = \frac{\sigma_{k_i-1} + l_i}{\rho_i}$.

Donc la valeur de la redondance introduite par les arrondissements pendant un cycle de codage est :

$$R = \sum_{i=1}^{m} \log(1 + \frac{l_i}{\sigma_{k_i-1}}),\ 0 \leq l_i \leq \rho_i - 1. \tag{5.1}$$

Nous avons $\sigma_{k_i-1} \geq m^{x-1}$, donc $0 < 1 + \frac{l_i}{\sigma_{k_i-1}} \leq 1 + \frac{l_i}{m^{x-1}}$.

D'après la relation précédente, nous pouvons déduire que $R \leq \sum_{i=1}^{m} \log(1 + \frac{\rho_i - 1}{m^{x-1}})$.

En supposant que l'arrondissement de la taille de la mémoire entraine toujours un ajustement maximal, la borne de la redondance R est :

$$B = \sum_{i=1}^{m} \log(1 + \frac{\rho_i - 1}{m^{x-1}}). \tag{5.2}$$

Pour $i \in \{1, 2, ..., m\}$, nous avons $\rho_i \in \{1, 2, ..., m\}$.

Nous n'avons pas nécessairement $\rho_i = i$.

Nous définissons un nouvel index j en fonction de i (mappage) tel que $\rho_j = j$.

Nous utilisons cet index dans la relation de B. Le résultat est $B = \sum_{j=1}^{m} \log(1 + \frac{j-1}{m^{x-1}})$.

On pose $j' = j - 1$ donc $B = \sum_{j'=0}^{m-1} \log(1 + \frac{j'}{m^{x-1}})$

Pour $x > -1$, nous avons $\log(1 + x) \leq \frac{x}{\ln(2)}$, donc $\log(1 + \frac{j'}{m^{x-1}}) \leq \frac{1}{\ln(2)} \cdot \frac{j'}{m^{x-1}}$.

Donc $B = \sum_{j'=0}^{m-1} \log(1 + \frac{j'}{m^{x-1}}) \leq \frac{1}{\ln(2)} \cdot \sum_{j'=0}^{m-1} \frac{j'}{m^{x-1}} = \frac{1}{\ln(2)} \cdot \frac{1}{m^{x-1}} \cdot \frac{m(m-1)}{2} = \frac{1}{\ln(2)} \cdot \frac{m-1}{2m^{x-2}}$

La rapport $\frac{1}{\ln(2)} \cdot \frac{m-1}{2m^{x-2}}$ converge vers 0 quand x tend vers l'infini. Pratiquement parlant, en choisissant des registres de plus en plus de grandes tailles, mais toujours de taille polynomiale par rapport à $\log(m)$, la redondance R introduite par $\mathbb{P}$ devient négligeable.

En ajoutant la redondance de la programmation $\mathbb{P}$ au seuil théorique 3.3 (redondance due à la création des blocs équilibrés), nous obtenons la redondance totale nécessaire pour créer des blocs équilibrés en utilisant l'algorithme de Pacman.

$$R_{totale} = \text{Seuil théorique} + R \tag{5.3}$$

Seuil theorique : c'est le nombre minimal de bits de parité p nécessaire pour créer les codes équilibrés (voir 3.3).

Les bornes calculées montrent que la redondance converge très rapidement vers le seuil théorique lorsque x croît. La bonne nouvelle est que nous pouvons obtenir une redondance presqu'optimale tout en utilisant des tailles de registre très raisonnables. La figure 5.9 présente trois courbes qui montrent la redondance totale pour Ω égale à m^2, m^3 et m^4. Les autres courbes montrent le seuil minimal de la redondance introduite par bloc, la redondance introduite par l'algorithme de Knuth (Knuth, 1986), celle introduite par la technique d'Immink et Weber (Immink et Weber, 2010) et celle introduite par la technique de Al-Rababa'a et al. (Al-Rababa'a *et al.*, 2013). Les valeurs calculées sont inscrites dans le tableau 5.1. Pour $\Omega = m^4$ la courbe de la redondance coïncide presqu'avec le seuil théorique. Si nous revenons

à l'exemple de l'introduction, pour $q = 512$, la taille des blocs équilibrés va être $m = 518$ car 6 bits de parité sont nécessaires. Le réglage de Ω en m^4 nécessite l'utilisation d'une taille de registre minimum de $\lceil \log(m^4) \rceil = 37$, ce qui est disponible nativement aujourd'hui sur les processeurs 64 bits.

Tableau 5.1 – Calcul de la redondance minimale et celle des algorithmes : Knuth, I&W, Al-Rababa'a et al. et Pacman pour x =2, 3, 4 et 5

m	log(m)	seuil théorique	KR	Immink et Weber	Al-Rababa'a et al.	m^2	m^3	m^4
8	3	1,83	3,48	3,95	2,63	3,714171	2,070439	1,869279
16	4	2,33	4,6	4,32	3,32	7,335028	2,685159	2,355685
32	5	2,83	5,7	5,5	3,96	13,59175	3,160027	2,840256
64	6	3,33	6,78	6	4,56	26,05906	3,739764	3,336263
128	7	3,83	7,85	6,2	5,15	50,26899	4,252198	3,833252
256	8	4,33	8,9	7	5,73	98,65113	4,730952	4,331644
512	9	4,83	9,95	7,9	6,3	194,2037	5,212954	4,83077
1024	10	5,33	11	8	6,85	385,0061	5,703668	5,33038

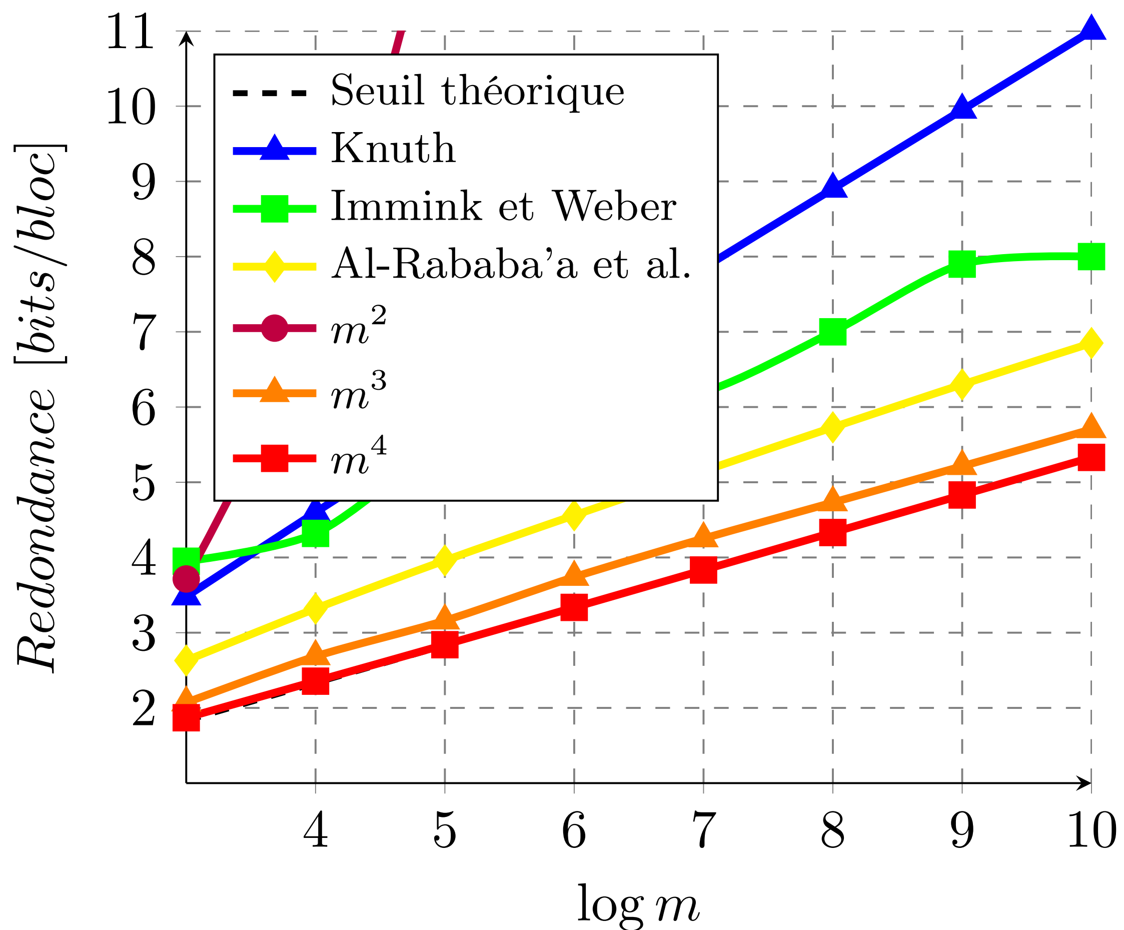

FIGURE 5.9 – Tracé des bornes de la redondance pour différentes valeur de m et de Ω

Chapitre 6

Conclusion

Nous avons présenté une technique basée sur les permutations, le personnage de jeu vidéo Pacman et les entiers à précision limitée pour encoder des données en mots de code équilibrés. La redondance introduite par le codage est particulièrement faible. Les résultats sont nettement meilleurs que ceux des travaux antérieurs. Néanmoins, les ressources requises par notre technique restent modestes : les calculs dans notre technique ne sont pas coûteux (pas de manipulation de grands nombres entiers) et la complexité temporelle et spatiale pour le codage ou le décodage d'un bloc est linéaire. Une des faiblesses de notre technique est que le codage et le décodage ne se font pas à la volée : avant de commencer le décodage, le décodeur doit recevoir la séquence complète des blocs équilibrés.

En rétrospective de ce travail, nous visons la conception de codages avec encore moins de redondance. En effet, en théorie, un mot de code équilibré de longueur m est capable d'incorporer $\log \binom{m}{m/2}$ bits source et non pas seulement $\left\lfloor \log \binom{m}{m/2} \right\rfloor$ bits. Nous pouvons aller au-delà des limites des nombres entiers et résoudre le problème avec des nombres rationnels. En d'autres termes, si $\log \binom{m}{m/2} \geq \frac{u}{v}$, alors il devrait être possible de concevoir un encodage de telle façon à ce que les blocs de u bits d'entrée soient transformés en v blocs équilibrés.

Un autre objectif, plus modeste, consiste à améliorer l'initialisation et la terminaison utilisées dans la programmation Pacman. En principe, ces étapes ajoutent seulement un nombre constant de blocs équilibrés à la séquence et cette addition a un effet négligeable sur une séquence arbitrairement longue. Cependant, en pratique, il est préférable de ne pas s'appuyer sur le comportement asymptotique et de rendre ces étapes aussi efficaces que possible. Ceci est particulièrement important lorsqu'une séquence relativement courte de blocs d'entrée, voire un seul, est transformée.

Bibliographie

Evolved Universal Terrestrial Radio ACCESS : Multiplexing and channel coding, 3GPP Std. TS 36.212. 2.3

Tinku ACHARYA : Efficient table-lookup based visually-lossless image compression scheme, décembre 28 1999. US Patent 6,009,201.

Sulaiman AL-BASSAM et Bella BOSE : Design of efficient balanced codes. *IEEE Transactions on Computers*, 43(3):362–365, 1994. 1.2

Sulaiman AL-BASSAM et Rella BOSE : On balanced codes. *IEEE Transactions on Information Theory*, 36(2):406–408, 1990. 1.2, 1, 2.2.4, 3.2

Sulaiman A. AL-BASSAM : *Balanced Codes*. Ph. D. Thesis, Oregon State University, Dept. of Computer Science, 1990.

Ahmad AL-RABABA'A, Danny DUBÉ et Jean-Yves CHOUINARD : Using bit recycling to reduce Knuth's balanced codes redundancy. *In 2013 13th Canadian Workshop on Information Theory (CWIT)*. IEEE, 2013. (document), 1.1, 1.1, 3.2, 3.5, 3.6, 5.3

Noga ALON, E.E. BERGMANN, Don COPPERSMITH et Andrew M. ODLYZKO : Balancing sets of vectors. *IEEE Transactions on Information Theory*, 34(1):128–130, 1988. 1, 3.2

Mitsuharu ARIMURA et Ken-ichi IWATA : The minimum achievable redundancy rate of fixed-to-fixed length source codes for general sources. *In 2010 International Symposium onInformation Theory and its Applications (ISITA)*, pages 595–600. IEEE, 2010. 2.2.3

Robert F BAILEY : Error-correcting codes from permutation groups. *Discrete Mathematics*, 309(13):4253–4265, 2009. 2.3

Robert W BEMER : A proposal for character code compatibility. *Communications of the ACM*, 3(2):71–72, 1960. 2.2.3

Ernest E BERGMANN, Andrew M ODLYZKO et Surash H SANGANI : Half weight block codes for optical communications. *Bell Labs Technical Journal*, 65(3):85–93, 1986. (document), 1.1

Claude Berrou et Alain Glavieux : Turbo codes. *Encyclopedia of Telecommunications*, 2003. 2.3

Ian F Blake, Gérard Cohen et Mikhail Deza : Coding with permutations. *Information and Control*, 43(1):1–19, 1979. 2.3

Mario Blaum et Jehoshua Bruck : Coding for skew-tolerant parallel asynchronous communications. *IEEE Transactions on Information Theory*, 39(2):379–388, 1993. 1.1

Raj Chandra Bose et Dwijendra K Ray-Chaudhuri : On a class of error correcting binary group codes. *Information and control*, 3(1):68–79, 1960. 2.2.3

Ebru Celikel Cankaya, Suku Nair et Hakki C Cankaya : Applying error correction codes to achieve security and dependability. *Computer Standards & Interfaces*, 35(1):78–86, 2013.

J McKeen Cattell et Livingstone Farrand : Physical and mental measurements of the students of Columbia University. *Psychological Review*, 3(6):618, 1896. 2.1

Ke-Chiang Chu, James O Normile, Chia L Yeh et Daniel W Wright : Variable length decoding using lookup tables, octobre 12 1993. US Patent 5,253,053.

T Cover : Enumerative source encoding. *IEEE Transactions on Information Theory*, 19 (1):73–77, 1973. 1

P Danziger : Big O Notation. 2015. 2.4

Suparna Datta et Steven W McLaughlin : An enumerative method for runlength-limited codes : permutation codes. *IEEE Transactions on Information Theory*, 45(6):2199–2204, 1999.

Shlomi Dolev, Limor Lahiani et Yinnon Haviv : Unique permutation hashing. *Theoretical Computer Science*, 475:59–65, 2013. 2.3

Gregory D Durgin : Balanced codes for more throughput in RFID and backscatter links. *In IEEE International Conference on RFID Technology and Applications (RFID-TA)*, pages 65–70. IEEE, 2015. (document), 1.1, 3.1

Ivan J Fair et Craig Jamieson : Tabular construction of balanced codes. *Electronics Letters*, 49(16):997–999, 2013.

Ryan Gabrys et Olgica Milenkovic : Balanced permutation codes. *arXiv preprint arXiv :1601.06887*, 2016. 1.1

Marshall Hall : Permutations and combinations. *Combinatorial Theory, Second Edition*, pages 1–7, 1983. 2.3, 2.3

David A Huffman *et al.* : A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 2.2.2, 2.2.3

Kees A Schouhamer Immink : Coding techniques for the noisy magnetic recording channel : A state-of-the-art report. *IEEE Transactions on Communications*, 37(5):413–419, 1989. 1.1

Kees A Schouhamer Immink et Jos H Weber : Simple balanced codes that approach capacity. *In 2009 IEEE International Symposium on Information Theory*, pages 1554–1558. IEEE, 2009a. (document), 1.1, 1, 3.2

Kees A Schouhamer Immink et Jos H Weber : Simple balanced codes that approach capacity. *In IEEE International Symposium on Information Theory, 2009. ISIT 2009.*, pages 1554–1558. IEEE, 2009b.

Kees A Schouhamer Immink et Jos H Weber : Very efficient balanced codes. *IEEE Journal on Selected Areas in Communications*, 28(2):188–192, 2010. (document), 1.1, 1, 2.1.3, 3.2, 3.2, 3.4, 5.3

Kees A Schouhamer Immink, Jos H Weber et Hendrik C Ferreira : Balanced runlength limited codes using Knuth's algorithm. 2011. (document), 1.1, 1, 3.2

Razmik Karabed et Paul H Siegel : Matched spectral-null codes for partial-response channels. *IEEE Transactions on Information Theory*, 37(3):818–855, 1991. 1.1

Donald E Knuth : Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974. 2.4

Donald E Knuth : Efficient balanced codes. *IEEE Transactions on Information Theory*, 32 (1):51–53, 1986. (document), 1.1, 1.2, 1, 2.2.3, 2.2.4, 3.1, 3.1, 3.1, 3.2, 5.3

C-S Laih et C-N Yang : Design of efficient balanced codes with minimum distance 4. *IEE Proceedings-Communications*, 143(4):177–181, 1996.

Didier Le Gall : MPEG : A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, 1991. 2.2.2

Ernst L Leiss : Data integrety in digital optical disks. *IEEE Transactions on Computers*, 100 (9):818–827, 1984. 1.1

David JC MacKay : *Information theory, inference and learning algorithms.* Cambridge university press, 2003. 2.1.3

Raffaele Mascella et Luca G Tallini : On symbol permutation invariant balanced codes. *In ISIT 2005. Proceedings. International Symposium on Information Theory*, pages 2100–2104. IEEE, 2005. 1

Raffaele Mascella et Luca G Tallini : Efficient m-ary balanced codes which are invariant under symbol permutation. *IEEE Transactions on Computers*, 55(8):929–946, 2006. 1

Michael McCandless : The MP3 revolution. *IEEE Intelligent Systems and their Applications*, 14(3):8–9, 1999. 2.2.2

Yu S Medvedeva et B Ya Ryabko : Fast enumeration algorithm for words with given constraints on run lengths of ones. *Problems of Information Transmission*, 46(4):390–399, 2010. 1

Todd K Moon : Error correction coding. *Mathematical Methods and Algorithms. John Wiley and Son*, 2005. 2.2.3

James R Norris : *Markov chains.* Numéro 2. Cambridge University Press, 1998. 3

Yoram Ofek : The conservative code for bit synchronization. *IEEE Transactions on Communications*, 38(7):1107–1113, 1990. 1.1

Marios S Pattichis, Alan C Bovik, John W Havlicek et Nicholas D Sidiropoulos : On the representation of wideband images using permutations for lossless coding. *In 4th IEEE Southwest Symposium on Image Analysis and Interpretation*, page 237, Austin, TX, USA, 2000. IEEE.

Stanislaw J Piestrak : Design of encoders and self-testing checkers for some systematic unidirectional error detecting codes. *In International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 119–127, Paris, France, France, 1997. IEEE. 1.1

J Postel : Protocole de contrôle de transmission-spécification du protocole du programme DARPA Internet. Rapport technique, STD 7, RFC 793, septembre, 1981. 2.2.3

Irving S Reed et Gustave Solomon : Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960. 2.2.3

Ron M Roth, Paul H Siegel et Alexander Vardy : High-order spectral-null codes-constructions and bounds. *IEEE transactions on information theory*, 40(6):1826–1840, 1994. 1.1

Yuichi Saitoh et Hideki Imai : Multiple undirectional byte error-correcting codes. *IEEE Transactions on Information Theory*, 37(3):903–908, 1991. 1.1

Serap A Savari : Variable-to-fixed length codes and plurally parsable dictionaries. *In Data Compression Conference, 1999. Proceedings. DCC'99*, pages 453–462. IEEE, 1999. 2.2.3

Khalid Sayood : *Introduction to Data Compression.* Newnes, 2012. 2.2.4

Claude E SHANNON : Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949a. 2.1

Claude E SHANNON : A mathematical theory of communication. *Bell system technical journal*, 27:379–423, 1949b. 2.1.2, 2.1.2, 2.1.2, 2.2.3

Claude E SHANNON : A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001. 2.1.1

Vitaly SKACHEK et Kees A Schouhamer IMMINK : Constant weight codes : An approach based on Knuth's balancing method. *IEEE Journal on Selected Areas in Communications*, 32(5):909–918, 2014. 1

David SLEPIAN : Permutation modulations. *Proceedings of the IEEE*, 53(3):228–236, 1965. 1.1, 2.3

Emanuel SPERNER : Ein SATZ ÜBER UNTERMENGEN EINER ENDLICHEN MENGE. *Mathematische Zeitschrift*, 27(1):544–548, 1928. 1.2, 3.1

Martin STIGGE, Henryk PLÖTZ, Wolf MÜLLER et Jens-Peter REDLICH : Reversing CRC–Theory and Practice. 2006. 2.2.3

James A STORER et Thomas G SZYMANSKI : Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982. 2.2.3

Jeff F TABOR : Noise reduction using low weight and constant weight coding techniques. 1990. (document), 1.1, 3.1

Yoshitaka TAKASAKI, Mitsuo TANAKA, Narimichi MAEDA, Kiichi YAMASHITA et Katsuyuki NAGANO : Optical pulse formats for fiber optic digital communications. *IEEE Transactions on Communications*, 24(4):404–413, 1976. 1.1, 3.1

Luca G TALLINI et Bella BOSE : Balanced codes with parallel encoding and decoding. *IEEE Transactions on Computers*, 48(8):794–814, 1999. 1, 3.2

Luca G TALLINI, Renato M CAPOCELLI et Bella BOSE : Design of some new efficient balanced codes. *IEEE Transactions on Information Theory*, 42(3):790–802, 1996.

Yoshihiro TOHMA, Yasuyoshi OHYAMA et Ryozo SAKAI : Realization of fail-safe sequential machines by using a k-out-of-n code. *IEEE Transactions on Computers*, 100(11):1270–1275, 1971. 1.1

Brian Parker TUNSTALL : Synthesis of noiseless compression codes. 1967. 2.2.3

Jacobus Hendricus VAN LINT : *Introduction to Coding Theory*, volume 86. Springer Science & Business Media, 2012. 2.2.3

Henk Van Tilborg et Mario Blaum : On error-correcting balanced codes. *IEEE Transactions on Information Theory*, 35(5):1091–1095, 1989.

Karthik Visweswariah, Sanjeev R Kulkarni et Sergio Verdú : Universal variable-to-fixed length source codes. *IEEE Transactions on Information Theory*, 47(4):1461–1472, 2001. 2.2.3

Gregory K Wallace : The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992. 2.2.2

Jos H Weber et Kees A Schouhamer Immink : Knuth's balanced codes revisited. *IEEE Transactions on Information Theory*, 56(4):1673–1679, 2010. 3.2, 3.2

Terry A. Welch : A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984. 2.2.2, 2.2.3

Albert X Widmer : Partitioned DC-balanced (0, 6) 16b/18b transmission code with error correction, mars 6 2001. US Patent 6,198,413.

Albert X. Widmer et Peter A. Franaszek : A DC-balanced, partitioned-block, 8B/10B transmission code. *IBM Journal of research and development*, 27(5):440–451, 1983. 1.1

Ian H Witten, Radford M Neal et John G Cleary : Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. 2.2.2, 2.2.3

Jong-Hoon Youn et Bella Bose : Efficient encoding and decoding schemes for balanced codes. *IEEE Transactions on Computers*, (9):1229–1232, 2003. 1

Jacob Ziv et Abraham Lempel : A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977. 2.2.2, 2.2.3

Jacob Ziv et Abraham Lempel : Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978. 2.2.2, 2.2.3