: N° d'ordre
: N° de série

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE ECHAHID HAMMA LAKHDAR - EL OUED
FACULTÉ DES SCIENCES EXACTES
Département D'Informatique

Mémoire de Fin D'étude
Présenté pour l'obtention du Diplôme de

MASTER ACADEMIQUE

Domaine : **Mathématique et Informatique**
Filière : **Informatique**
Spécialité : **Systèmes Distribués et Intelligence Artificielle**

Présenté par :

- BEN KHALIFA ABDELMADJID
- TERKI MOHAMMED

Thème

Compression de données binaires par une méthode à base de couplage des vecteurs

Soutenue le : 22-09- 2022 Devant le jury:

M.	BOURDJOUHE CHAFIK	MAA	Président
M.	GUIA SANA SAHAR	MAA	Rapporteur
M.	ZAIZ FAOUZI	MAA	Encadreur

Année Universitaire: 2021/2022

Dédicaces

Tout d'abord, je veux rendre grâce à Dieu, le Clément et le Très Miséricordieux pour son amour éternel. C'est ainsi que je dédie cette thèse à:

ma mère pour sa tendresse et mon père pour sa patience et encouragement
A mes très chers frères,mes très chères sœurs et leurs familles
A ma chère femme
A mon petit ange adoré Ahmed
A mes chers enfants Souhaib, Imane.
A mes oncles et mes tentes, mes cousins et mes cousines
A mon cher professeur Zaiz Faouzi
A mon binôme Terki Mohamed
A tout mes amis ,tous ceux que j'aime, je leur souhaite le bonheur et la réussite dans tous les domaines de la vie
A la promotion 2éme ANNEE MASTER INFORMATIQUE 2021/2022 Université Echahid Hamma Lakhdar d'El Oued

ABDELMADJID

Dédicaces

En signe d'amour de gratitude et de respect je dédie ce modeste travail:

ma mère pour sa tendresse et mon père pour sa patience et encouragement
A mes très chers frères,mes très chères sœurs et leurs familles
A ma chère femme
A mes enfants.
A mes oncles et mes tentes, mes cousins et mes cousines
A mon cher enseignant Zaiz Faouzi
A mon binôme Abdelmadjid Ben Khalifa
A tout mes amis ,tous ceux que j'aime, je leur souhaite le bonheur et la réussite dans tous les domaines de la vie
A la promotion 2éme ANNEE MASTER INFORMATIQUE 2021/2022 Université Echahid Hamma Lakhdar d'El Oued

TERKI MOHAMMED

Remerciements

En premier lieu nous remercions Dieu, le tout puissant, nous rendons grâce pour nous avoir donné santé, patience, volonté et surtout raison.

De plus, nous tenons à exprimer nos remerciement à :

***Mr ZAIZ Faouzi** d'avoir accepter de nous encadrer, pour la confiance qu'il nous a accordé tout au long de ce travail et pour l'intérêt qu'il a porté à celui-ci.*

A tout nos camarades de la promotion.

Grand remerciement destiné à l'ensemble de nos enseignants et enseignantes qui ont contribué à notre formation, depuis le cycle primaire jusqu'au cursus universitaire.

Nos remerciements sont adressés au président et aux membres de jury qui nous ont honorés de leur présence et d'avoir accepté de juger ce travail.

Et à tous ceux qui nous ont aider de près ou de loin.

Résumé

Alors que le monde dérive de plus en plus vers le réseau social, la taille et la quantité de données partagées sur Internet augmentent de jour en jour. Étant donné que la bande passante du réseau est toujours limité, nous avons besoin d'algorithmes de compression efficaces pour faciliter le partage rapide et efficace des données sur le réseau.

Dans ce travail, nous proposons une technique de pré-traitement des fichiers dans le but de les rendent plus compressible. nous comparons les modèles modernes couramment utilisés algorithmes de compression : Deflate,Deflate64, Bzip2, LZMA et PPMd en analysant leurs performances sur le corpus Silesia. Cette évaluation est réalisée dans deux cas sans et avec l'application de l'algorithme de pré-traitement.

Les résultats obtenus sont très encourageants et ouvre des perspectives dans le domaine de compression des données.

Mots Clés: Compression, lossless compression, Silesia, taux de compression, vitesse de compression.

Abstract

As the world drifts further and further social network, the size and amount of data shared on the internet is increasing day by day. Since the network bandwidth is still limited, we need efficient compression algorithms to facilitates fast and efficient sharing of data over the network.

In this work, we indicate a technique for pre-processing files in order to make them more compressible. we compare commonly used modern models compression algorithms: Deflate,Deflate64, Bzip2, LZMA and PPMd by analyzing their performance on the Silesia corpus. This evaluation is carried out in two cases without and with the application of the pre-processing algorithm.

The results obtained are very encouraging and open perspectives in the field of data compression.

Keywords: Compression, lossless compression, Silesia, compression ratio, compression speed.

الملخص

يتم استخدام ضغط البيانات على نطاق واسع من قبل المجتمع لأنه من خلال الضغط يمكننا توفير التخزين، ويمكن أن يؤدي ضغط البيانات أيضاً إلى تسريع نقل البيانات من شخص إلى آخر.

ففي هذا العمل، نشير الى تقنية تسمح بالمعالجة المسبقة للبيانات لجعلها أكثر قابلية للضغط ، ولقد قمنا بإستخدام قاعدة البيانات سيليزيا وطبقنا عليها خوارزميات الضغط الحديثة : Deflate ، Bzip2 ، LZMA ، PPMd و Deflate64 ومن ثم إجراء تحليل ومقارنة للملفات المضغوطة قبل المعالجة وبعد المعالجة.

النتائج التي تم الحصول عليها هي وجهات نظر مشجعة للغاية ومفتوحة في مجال ضغط البيانات.

كلمات مفتاحية: ضغط المعلومات Silesia، سرعة الضغط، الضغط بدون ضياع مقدار الضغط، .

Table des matières

Liste des figures

Liste des tableaux

Introduction

Depuis les débuts de l'informatique et avec le développement rapide de la technologie et avec le soutien de logiciels et de matériel qui facilitent de plus en plus la diffusion rapide de l'information sur Internet partout dans le monde. Les informations obtenues peuvent être transmises sur Internet comme moyen de communication aux experts en informatique. Cependant, toutes les informations ne peuvent pas être envoyées facilement. Il existe un volume important qui peut entraver le transfert rapide des données et économiser de l'espace de stockage sur l'ordinateur.

Pour pallier le problème de l'information ou des données qui seront transférées et sauvegardées, il était impératif de recourir à la méthode de compression qui permet de fournir de l'espace pour le stockage et le transfert des données. La compression des données est une exigence courante pour la plupart des applications informatiques. Il existe un certain nombre d'algorithmes de compression de données, qui sont dédiés à la compression de différents formats de données qui utilisent des méthodes différentes. Il existe deux types ou méthodes : la compression sans perte et la compression avec perte.

Dans ce travail, nous allons proposé une méthode de pré-traitement afin d'essayer de rendre les fichier plus compressible en faisant des permutations dans les segments ou parties du fichier.

Le reste du travail est organisé comme suit:

Le premier chapitre présente un état de l'art sur la compression des données. D'abord, nous allons présenter la situation et les problèmes liées compression. Ensuite, nous allons donner un aperçu sur les différente types de compression. Par la suite, nous allons décrire la modélisation et la manière de codage des données. Finalement, nous présentons la façon avec laquelle on peut évaluer un algorithme de compression.

Le deuxième chapitre présente un état de l'art sur les différentes méthodes de compression,

ainsi qu'une comparaison en ces techniques.

Le troisième chapitre va mettre l'accent sur les différents modes d'un système de compression (compression et décompression).

Le quatrième chapitre présente la validation des résultats. Dans lequel, nous allons voir comment fonctionne globalement notre système. Enfin, nous présentons les résultats obtenus ainsi que les discussions faites.

Nous terminons le travail par une conclusion sur les résultats obtenus par les méthode utilisées, et des perspectives de ce travail.

Chapitre 1

Introduction à la compression de données

Introduction

La compression des données est la science (et l'art) de représenter l'information dans un forme compacte. La compression de données est un domaine de recherche actif en informatique. Par "compresser des données", nous entendons en fait dériver des techniques ou, plus spécifiquement, concevoir des algorithmes efficaces pour [2]:

- représenter les données de manière moins redondante.
- supprimer la redondance dans les données.
- mettre en œuvre le codage, y compris le codage et le décodage.

Dans ce chapitre, nous allons voir un état de l'art sur la compression en général. En plus, nous allons présenter les deux types de compression avec et sans perte. Ainsi que comment modéliser et coder les données. Finalement, nous allons voir comment évaluer la performance d'un algorithme de compression.

1.1 Qu'est-ce que la compression de données

Une séquence sur un certain alphabet présente généralement certaines régularités, ce qui est nécessaire pour penser à la compression. Pour des textes anglais typiques, nous pouvons constater que les lettres les plus fréquentes sont e, t, a, et les lettres les moins fréquentes sont q, z. Nous pouvons également trouver des mots tels que the, of, to fréquemment. Souvent aussi, des fragments plus longs du texte se répètent, voire même des phrases entières. Nous pouvons utiliser ces propriétés d'une d'une manière ou d'une autre, et les sections suivantes développent ce sujet[9].

Une stratégie différente pour compresser la séquence de données d'image est nécessaire. Avec une photo de ciel nocturne on peut encore s'attendre à ce que la couleur la plus fréquente des pixels est noir ou gris foncé. Mais avec une photo générique, nous n'avons

généralement aucune information quelle couleur est la plus fréquente. En général, nous n'avons aucune connaissance a priori l'image, mais on peut y trouver des régularités. Par exemple, des couleurs successives pixels sont généralement similaires, certaines parties de l'image sont répétées.

1.1.1 Définition Compression de données

En informatique, la compression de données est la technique dans laquelle on emploie une paire de fonctions C et D sur des chaines. La fonction C a pour objectif de compresser les données et la fonction D, de les décompresser. L'effet souhaité est d'avoir $|C(x)| < |x|$.
On ne l'observe pas nécessairement pour tous les x [1].

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

1.1.2 Importance de la compression des données

Les techniques de compression de données sont principalement motivées par la nécessité d'améliorer l'efficacité du traitement de l'information. Cela comprend l'amélioration des éléments suivants principaux aspects dans le domaine numérique [2]:

- efficacité de stockage.
- utilisation efficace de la bande passante de transmission.
- mettre en œuvre le codage, y compris le codage et réduction du temps de transmission.

Bien que le coût du stockage et de la bande passante de transmission des données numériques ont chuté de façon spectaculaire, la demande d'augmentation de leur capacité dans de nombreux applications a connu une croissance rapide depuis. Il y a des cas où un stockage supplémentaire ou une bande passante supplémentaire est difficile à obtenir, voire impossible. La compression des données en tant que moyen peut rendre l'utilisation beaucoup plus efficace des ressources à moindre coût. La recherche active sur la compression des données peut conduire à de nouveaux produits innovants et aider à fournir de meilleurs services

1.1.3 Données source

le mot données comprend toute information numérique pouvant être traitée dans un ordinateur, ce qui inclut le texte, la voix, la vidéo, les images fixes, audio et films. Les données avant tout processus de compression (c'est-à-dire d'encodage) sont appelées les données source, ou la source en abrégé [2].

Les trois types courants de données source dans l'ordinateur sont le texte et (numérique) image et son.

- Les données textuelles sont généralement représentées par un code ASCII (ou EBCDIC).
- Les données d'image sont souvent représentées par un tableau bidimensionnel de pixels dans auquel chaque pixel est associé à son code couleur.
- Les données sonores sont représentées par une fonction d'onde (périodique).

Dans le monde des applications, les données source à compresser sont susceptibles d'être soi-disant multimédia et peut être un mélange de texte, d'image et de son.

1.2 Compression avec perte et sans perte

La compression de données est simplement un moyen de représentation numérique efficace d'une source de données telle que le texte, l'image et le son. source de données telle que le texte, l'image et le son. L'objectif de la compression compression de données est de représenter une source sous forme numérique avec aussi peu de bits que possible tout en répondant aux exigences minimales de reconstruction. Cet objectif est atteint en supprimant toute redondance présente dans la source
Il existe deux grandes familles de techniques de compression en termes de possibilité de reconstituer la source originale. Ils sont appelés compression sans perte et la compression avec perte [9].

1.2.1 Compression sans perte

Une approche de compression n'est sans perte que s'il est possible de reconstruire exactement les données d'origine de la version compressée. Il n'y a aucune perte d'informations lors de la compression Ceci, lorsqu'il est utilisé comme un terme général, en fait comprend à la fois la compression et décompression [9].

Les techniques de compression sans perte sont principalement appliquées aux données symboliques telles que texte en caractères, données numériques, code source informatique

et graphiques exécutables et icônes [2].

Les techniques de compression sans perte sont également utilisées lorsque les données d'origine d'un source sont si importants que nous ne pouvons pas nous permettre de perdre des détails. Par exemple, images médicales, textes et images conservés pour des raisons légales ; quelques fichiers exécutables par ordinateur, etc.

1.2.2 Compression avec perte

Une méthode de compression est une compression avec perte uniquement s'il n'est pas possible de reconstruire exactement l'original à partir de la version compressée. Il y a quelques détails insignifiants qui peuvent être perdus pendant le processus de compression.[9]

La reconstruction approximative peut être très bonne en termes de taux de compression, mais cela nécessite généralement un compromis entre le visuel la qualité et la complexité de calcul (c'est-à-dire la vitesse) [9].

Les données telles que les images multimédias, la vidéo et l'audio sont plus facilement compressées par des techniques de compression avec perte.

1.3 Modélisation et codage des données

Dans cette section, nous allons voir la notion de paradigme de compression, ainsi que la modélisation des données à compresser.

1.3.1 Paradigme moderne de la compression des données

Le paradigme moderne de la compression la divise en deux étapes : la modélisation et codage. Premièrement, nous reconnaissons la séquence, recherchons les régularités et les similitudes. Cela se fait dans la phase de modélisation. La méthode de modélisation est spécialisée pour le type de données que nous compressons. Il est évident que dans les données vidéo nous chercherons pour des similitudes différentes de celles des données textuelles. Les méthodes de modélisation sont souvent différent pour les méthodes sans perte et avec perte. Choisir la bonne méthode de modélisation est important car plus on trouve de régularités plus on peut réduire le longueur de la séquence. En particulier, nous ne pouvons pas du tout réduire la longueur de la séquence si nous ne savons pas ce qui est redondant dans la séquence. si nous ne savons pas ce qui est redondant dans la séquence. [9].

La deuxième étape, le codage, s'appuie sur les connaissances acquises lors de l'étape

de modélisation et supprime les données redondantes. Les méthodes de codage ne sont pas si diversifié car le processus de modélisation est l'étape où l'adaptation aux données sont faites. Par conséquent, nous n'avons qu'à encoder efficacement la séquence en supprimant la redondance connue.

Certaines méthodes de compression plus anciennes, telles que les algorithmes de Ziv-Lempel, ne peuvent être précisément classées comme représentatives de la modélisation-codage paradigme. Ils sont également toujours présents dans les solutions pratiques contemporaines, mais leur importance semble diminuer. Nous considérons qu'ils ont un meilleur vue de l'arrière-plan, mais nous prêtons notre attention aux algorithmes modernes.

1.3.2 Modélisation de données

L'étape de modélisation construit un modèle représentant la séquence à compresser, la séquence d'entrée et prédit les symboles futurs dans la séquence. Ici, nous estimons une distribution de probabilité d'occurrences de symboles [9].

La manière la plus simple de modéliser consiste à utiliser une table de probabilités précalculée d'occurrences de symboles. Meilleure est notre connaissance des occurrences de symboles dans la séquence actuelle, mieux nous pouvons prédire les futurs personnages. Nous pouvons utiliser des tables précalculées si nous savons exactement ce que nous compressons. Si nous savons que la séquence d'entrée est un texte anglais, nous pouvons utiliser des fréquences typiques d'occurrences de personnages. Si, toutefois, nous ne connaissons pas la langue du texte, et nous utilisons, par exemple, le tableau précalculé pour la langue anglaise à un polonais texte, nous pouvons obtenir des résultats bien pires, car la différence entre le les fréquences d'entrée et celles précalculées sont importantes. De plus, la langue polonaise utilise des lettres telles que ó,ś qui n'existent pas en anglais. Par conséquent, le tableau des fréquences contient une fréquence égale à zéro pour de tels symboles. C'est un gros problème et le compresseur peut ne pas fonctionner lorsque de tels symboles extraordinaires apparaître. De plus, la probabilité d'occurrences de symboles diffère pour les textes de divers auteurs [9].

Par conséquent, une meilleure façon est de ne pas trop présumer de la séquence d'entrée et construire le modèle à partir de la partie codée de la séquence. Pendant le processus de décompression, le décodeur peut construire son propre modèle de la même manière. Une telle approche de la compression est dite adaptative car le modèle est construit uniquement des symboles passés et s'adapte au contenu de la séquence. Nous ne pouvions pas utiliser les futurs symboles car ils sont inconnus du décompresseur. Autre, statique, les méthodes construisent le modèle à partir de la séquence entière avant la compression et puis utilisez-le. Le décompresseur n'a aucune connaissance de la séquence d'entrée, et

le modèle doit également être stocké dans la séquence de sortie. L'approche statique est allée presque hors d'usage car on peut prouver que la voie adaptative est équivalente.

1.3.3 Principales techniques de compression

La compression des données est souvent appelée codage en raison du fait que son objectif principal est de trouver un moyen court de représenter les données. Codage et décodage Utilisé pour signifier respectivement compresser et décompresser. Il existe de nombreux algorithmes de compression qui transforment les données afin de les rendre plus compactes Parmi les plus importants sont mentionnés ci-dessous [2]:

- Codage de longueur d'exécution (Run-length coding).
- Codage de Shannon-Fano.
- Codage de Huffman.
- Codage arithmétique.
- Code universel.
- Codage basé sur un dictionnaire.
- Burrows-Wheeler transform (BWT).
- Prediction by Partial Matching (PPM).

1.4 Évaluation des performances

1.4.1 Performances de compression

Les performances d'un algorithme de compression peuvent être mesurées par différents Critères. Cela dépend de ce qui est notre préoccupation prioritaire. dans ce guide thématique, nous concernent principalement l'effet produit par une compression (c'est-à-dire différence de taille du fichier d'entrée avant la compression et la taille du fichier sortie après la compression) [2].

Il est difficile de mesurer les performances d'un algorithme de compression en général car son comportement de compression dépend beaucoup du fait que les données contient les bons modèles que l'algorithme recherche [2].

La façon la plus simple de mesurer l'effet d'une compression est d'utiliser le ratio de compression.

Le but est de mesurer l'effet d'une compression par le rétrécissement de la taille de la source par rapport à la taille de la version compressée.

Il existe plusieurs manières de mesurer l'effet de compression :

1. Ratio de compression [2].
   c'est simplement le rapport taille.après.compression à la taille.avant.compression ou
   $Rationdecompression = \frac{taille.après.compression}{taille.avant.compression}$
2. Facteur de compression
   c'est l'inverse du taux de compression
   $Facteurdecompression = \frac{taille.avant.compression}{taille.après.compression}$
3. Pourcentage d'économie
   il montre le rétrécissement en pourcentage
   $Pourcentaged'économie = \frac{taille.avant.compression - taille.après.compression}{taille.avant.compression}\%$
   Exemple : Un fichier image source (pixels 256 × 256) de 65 536 octets est compressé dans un fichier de 16 384 octets. Le taux de compression est de 1/4 et le facteur de compression est de 4. Le pourcentage d'économie est de : 75

Conclusion

Dans ce chapitre, nous avons abordé la définition de la compression des données et l'importance de la compression des données afin de l'exploiter au maximum dans les ressources de stockage et de transmission en utilisant le moins de bits possibles, avec une explication des deux principaux groupes de techniques de compression en termes de reconstruction de la source originale et de modélisation et d'encodage des données.

Chapitre 2

Méthodes de compression des données

Introduction

La compression des données est largement utilisée par tout le monde car grâce à la compression, nous pouvons économiser du stockage. La compression des données peut également accélérer le transfert de données d'une personne à une autre. Cette procédure nécessite une méthode de compression des données, les données qui peuvent être compressées ne sont pas seulement des données textuelles, il peut s'agir d'images et de vidéos. La technologie de compression des données est divisée en deux parties, la compression sans perte et la compression perte. Il existe de nombreuses méthodes de compression telles que Huffman, Shannon Fano, Tunstall, Lempel Ziv welch et le codage de longueur d'exécution. Chaque méthode a la capacité d'effectuer une compression différente. Nous expliquerons ces méthodes en détail pour voir le résultat de la création d'un fichier par la taille du fichier compressé, qui devient plus petit que le fichier d'origine.

2.1 Approches de compression des données

Toutes les méthodes de compression de données sont basées sur un principe logique simple. Si nous imaginons que les éléments plus fréquents sont codés avec des symboles plus courts, et les éléments moins fréquents avec des symboles plus longs, alors moins d'espace sera nécessaire pour stocker toutes les données que si tous les éléments étaient représentés par des codes de même longueur. La relation exacte entre les fréquences des éléments et les longueurs de code optimales est décrite dans la théorie dite du codage source, qui définit la compression sans perte maximale.

2.1.1 Codage de longueur d'exécution (Run Length Encoding)

L'algorithme de longueur d'exécution (Run Length Encoding) est l'un des algorithmes qui peuvent être utilisés pour compresser des données afin que la taille des données produites soit inférieure à la taille d'origine. C'est la forme la plus simple de technologie

de compression sans perte où l'idée de l'algorithme Run-length est de remplacer des symboles répétés successivement par une paire de codes constituée d'un symbole répétitif et de son nombre d'itérations, ou d'une séquence de symboles non répétitifs..[1, 6]

Exemple: La chaîne ABBBBBBBCC peut être représentée par Ar7Br2C ou A7B2C, où r7 et r2 signifient respectivement 7 et 2 occurrences.
L'algorithme pour produire un code de longueur d'exécution est le suivant :

1. L'algorithme recherche la répétition des symboles.
2. Les caractères répétitifs sont remplacés par un caractère d'échappement suivi du symbole et du nombre binaire.

2.1.2 Codage de Shannon-Fano

Le codage de Shannon-Fano a été proposé en 1948 par Claude Shannon dans l'article 'A Mathematical Theory of Communication'. L'algorithme pour produire un code de Shannon-Fano pour un alphabet $\sum$ est le suivant :[1]

1. Trier les symboles de l'alphabet $\sum$ en ordre décroissant de fréquence d'apparition (du plus fréquent au moins fréquent).
2. Diviser $\sum$ en deux sous-ensembles $\sum_1$ et $\sum_2$ en respectant les deux contraintes suivantes:

   (a) - $\sum_1$ doit contenir les n plus fréquents symboles de $\sum$ et $\sum_2$ doit contenir les $|\sum| - n$ moins fréquents symboles de $\sum$.
   (b) - La différence entre la somme des fréquences d'apparition des symboles de l'ensemble $\sum_1$ et la somme des fréquences d'apparition des symboles de l'ensemble $\sum_2$ doit être la plus petite possible. Formellement, nous cherchons les ensembles $\sum_1 = \{S_1, S_2, ..., S_{j-1}\}$ et $\sum_2 = \{S_j, S_{j+1}, ..., S_N\}$ tels que:

$$arg_{min1 \leq j \leq N} |\sum_{i=1}^{j-1} P(s_i) - \sum_{i=j}^{N} P(s_i)| \tag{2.1}$$

   où $N = |\sum|$ et$P(s_i)$ est la fréquence d'apparition de s_i.
3. Attribuer aux symboles dans $\sum_1$ le bit 0 comme premier bit de leurs mots de code et attribuer aux symboles dans $\sum_2$ le bit 1.

Tableau 2.1: Fréquences d'apparition des symboles de l'alphabet $\sum$.

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

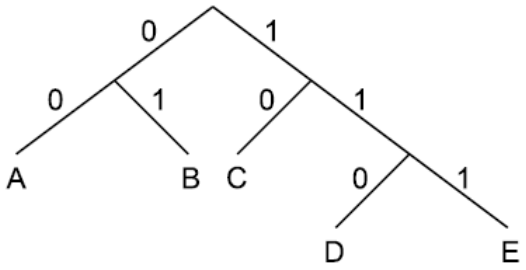

Figure 2.1: Algorithme de Shannon-Fano sur l'alphabet $\sum$ [8].

4. Répéter l'algorithme récursivement, à partir de l'étape 2, sur chacun des ensembles jusqu'à ce qu'il ne reste qu'un symbole dans chaque ensemble. A chaque fois qu'un ensemble est divisé, les symboles appartenant à cet ensemble se voient attribuer un bit supplémentaire à leurs mots de code.[1]

   Le tableau 2.1 contient les fréquences d'apparition des différents symboles d'un alphabet $\sum = \{A, B, C, D, E\}$. La figure 2.1 illustre un arbre représentant l'application de l'algorithme de Shannon-Fano sur l'alphabet $\sum$. Les symboles sont représentés par les feuilles de l'arbre. Un noeud interne représente la division d'un ensemble de symboles en deux sous-ensembles. Une branche représente un ensemble de symboles et porte l'étiquette 0 ou 1 servant à former les mots de code des symboles. Le tableau 2.2 présente le code de Shannon-Fano pour l'alphabet $\sum$.

Tableau 2.2: Code de Shannon-Fano pour l'alphabet $\sum$.

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

2.1.3 Codage de Huffman

Il est introduit en 1952 par David A. Huffman dans le livre 'A Method for the Construction of Minimum-Redundancy Codes', la notation de Huffman est largement utilisée dans le monde pour la compression de données. Bien qu'il existe des techniques plus efficaces et qu'il soit souvent utilisé dans les matières littéraires. La méthode de codage Huffman est la meilleure parmi les codes de préfixe qui attribuent des mots de code à Longueurs entières pour les symboles.[1]

La technique utilise les arbres binaires et deux files d'attente. Dans les arbres binaires, les

feuilles représentent les symboles d'un alphabet $\sum$ et chacun des noeuds a une probabilité p qui lui est associée. Pour les feuilles, p est simplement la fréquence d'apparition du symbole représenté. Pour les noeuds internes, p est égal à la somme des probabilités des noeuds fils : $p = p1 + p2$, où p1 est la probabilité du fils gauche et p2, la probabilité du fils droit. Deux files d'attente sont utilisées pour ne pas avoir à retrier les noeuds à chaque fois. La première le F1, est utilisée pour les feuilles et la deuxième file, F2, pour les noeuds internes. L'algorithme pour construire un code de Huffman pour un alphabet $\sum$ est le suivant :[1]

1. Créer une feuille pour chacun des symboles de l'alphabet $\sum$.
2. Ajouter chacune des feuilles, en ordre croissant de probabilité p,à F1.
3. Tant que $|F1| + |F2] > 1$.
   (a) - Retirer de F1 et/ou F2 les deux noeuds ayant les plus petits p.
   (b) - Créer un nouveau noeud ayant comme fils les deux noeuds qui viennent d'être retirés des files ; la probabilité du noeud est égale à la somme des probabilités de ses deux fils : $p = p1 + p2$, où p1 est la probabilité du fils gauche et p2, la probabilité du fils droit.
   (c) - Ajouter le nouveau noeud à la fin de F2
4. Le noeud restant est le noeud racine de l'arbre (arbre de Huffman).

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

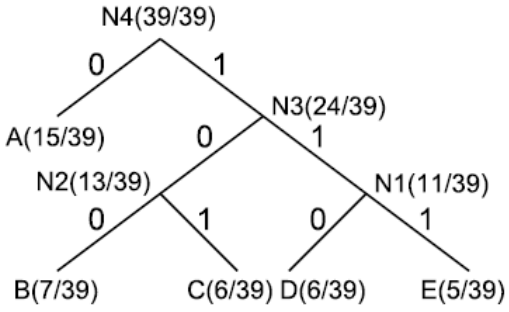

Figure 2.2: Arbre de Huffman pour l'alphabet $\sum$

La figure 2.2 illustre l'arbre de Huffman pour l'alphabet $\sum = \{A, B, C, D, E\}$ (voir tableau 2.3). Les noeuds internes sont étiquetés N1, N2, N3 et N4. La probabilité associée à chacun des noeuds est entre parenthèses.

Une branche porte l'étiquette 0 ou 1 servant à former les mots de code des symboles qui sont représentés par les feuilles de l'arbre. Le tableau 2.3 présente le code de Huffman pour l'alphabet $\sum$.

Tableau 2.3: Code de Huffman pour l'alphabet $\sum$.

Symbole	Mot de code	-log(p)
A	00	1.38
B	100	2.48
C	101	2.70
D	110	2.70
E	111	2.96

Maintenant, comparons le coût espéré d'un symbole avec le codage de Shannon-Fano (équation 2.2) et le coût espéréd'un symbole avec le codage de Huffman (équation 2.3) :

$$P(A) \times 2 + P(B) \times 2 + P(C) \times 2 + P(D) \times 3 + P(E) \times 3 + 2.28bits \tag{2.2}$$

$$P(A) \times 1 + P(B) \times 3 + P(C) \times 3 + P(D) \times 3 + P(E) \times 3 + 2.23bits \tag{2.3}$$

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

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

2.1.3.1 Codage de Huffman adaptatif

Le chiffrement de Huffman nécessite une connaissance préalable des fréquences d'occurrence Pour les symboles alphabétiques afin de construire l'arbre qui est utilisé pour obtenir les mots du code. Malheureusement, le cryptage se déroule en deux étapes La première fois pour compiler des statistiques et la deuxième fois pour chiffrer les données. Cela peut donc causer un problème de conservation des données ou lors de l'utilisation de la technologie de codage Huffman en temps réel.[1]

Le codage de Huffman adaptatif est une forme de codage qui ne nécessite pas la connaissance des fréquences des symboles pour apparaître à l'avance : comme lors de la lecture des données, l'arbre de Huffman est utilisé pour obtenir le fichier de mots de code et le mettre à jour. Il n'est donc plus nécessaire de passer par un ou deux chemins ou étapes de chiffrement des données : une seule fois suffit. De plus, une

unité n'est plus nécessaire.Évidemment, cette technologie est un peu plus compliquée et malheureusement, Nous allons donc montrer l'idée de base comme suit [6]:

1. Initialiser un arbre de Huffman qui contient tous les symboles d'un alphabet $\sum$ en assignant une probabilitée par défaut à chaque symbole

2. Tant qu'il y a des données à encoder :
   (a) - Lire le prochain symbole $s \in \sum$ qui doit être encodé.
   (b) - Encoder le symbole s.
   (c) - Mettre à jour l'arbre de Huffman en tenant compte de la nouvelle apparition du symbole s

Une instance de chiffrement de Huffman adaptatif peut reconstruire l'arbre de Huffman chaque fois que l'arbre est mis à jour. Heureusement, il existe des techniques plus efficace dans la mise à jour de l'arbre de Huffman et des mots de code qui sont produits.[1]

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

2.1.4 Codage arithmétique

Le codage arithmétique est l'un des moyens les plus efficaces pour compresser l'information, le codage arithmétique permet de crypter les messages avec une entropie inférieure à 1 bit par symbole. Parce que la plupart des algorithmes de codage arithmétique sont brevetés, ils représentent également chaque séquence possible de n messages avec un intervalle discret sur la droite numérique entre 0 et 1.[1]
Considérons l'alphabet $\sum$ = {A,B,C,EOF} présenté au tableau 2.4 suivant:

Tableau 2.4: Fréquences d'apparition des symboles de l'alphabet $\sum$.

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

Maintenant,encodons le message A-C-EOF avec le codage arithmétique. Le codage arithmétique fonctionne en divisant un intervalle donné selon les fréquences d'apparition des symboles d'un alphabet . Initialement, cet intervalle est $[0, 1]$. L'algorithme est le suivant :

1. Initialiser l'intervalle de départ à $[0, 1]$.

2. Tant qu'il y a des symboles à encoder : (a) - Diviser l'intervalle courant selon les fréquences d'apparition des symboles de l'alphabet $\sum$.
   (b) - Lire le prochain symbole $s \in \sum$ qui doit être encodé.
   (c) - L'intervalle courant devient le sous-intervalle représentant s (s est maintenant

considéré encodé).

3. Lorsque tous les symboles sont encodés, transmettre au décodeur un nombre réel qui se situe à l'intérieur du dernier intervalle courant.

La figure 2.3 illustre les différentes étapes du codage arithmétique pour encoder le message A-C-EOF. Après que le dernier symbole soit encodé, l'intervalle courant est $[0,534-0,54]$. Nous devons donc transmettre au décodeur un nombre réel qui se situe à l'intérieur de cet intervalle, comme 0,538 par exemple. En connaissant ce nombre, le décodeur est en mesure de refaire toutes les étapes qui ont été réalisées par l'encodeur afin de reconstruire le message original [7, 10].

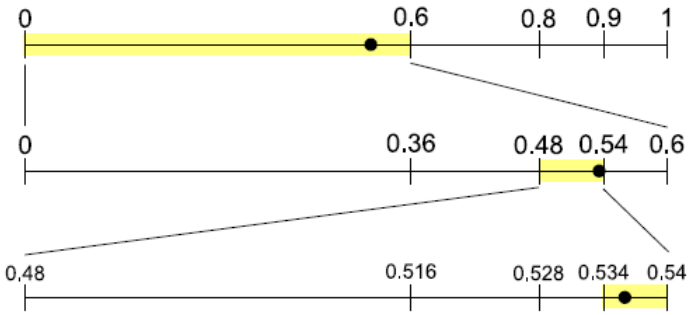

Figure 2.3: Codage arithmétique pour le message A-C-EOF

2.1.5 Codage universel

Un code universel est un code préfixe servant à représenter des entiers positifs non-bornés. Peu importe la distribution de probabilité sur les entiers, si $p(i) \geq p(i+1) \forall i$, alors les longueurs des mots de code assignés aux entiers ne diffèrent que d'un facteur constant des longueurs des mots de code qui seraient assignés par un code optimal[4].

2.1.5.1 Elias gamma

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

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

Le tableau 2.5 illustre quelques exemples. Pour obtenir le nombre entier correspondant à un mot de code, il suffit de :[1]

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

Tableau 2.5: Exemples de mots de code Elias gamma.

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

2.1.5.2 Levenshtein

Un code de Levenshtein est un code universel utilisé pour encoder des nombres entiers non-négatifs dont la borne supérieure est inconnue. Le mot de code pour un nombre entier n est produit par la fonction L :[4].

$$\begin{array}{rl} L(n) & \\ L : N \to \{0,1\}^* & = \begin{cases} 0 & \text{si n=0} \\ 1L(k)N(n,k) & sinon, o\grave{u} k = [log_2 n] \end{cases} \end{array} \tag{2.4}$$

$$\begin{array}{rl} N(n,k) & \\ N : NxN \to \{0,1\}^* & = \begin{cases} \in & \text{si k=0} \\ N([n/2, k-1])b & sinon, \text{ où n mod 2} \end{cases} \end{array} \tag{2.5}$$

Tableau 2.6: Exemples de mots de code de Levenshtein.

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

Le tableau 2.6 illustre quelques exemples. Pour obtenir le nombre entier n correspondant à un mot de code, il suffit de :

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

2.1.6 Codage basé sur un dictionnaire

2.1.6.1 LZ77

LZ77 est un algorithme de compression de données sans perte qui a été introduit en 1977 par Jacob Ziv et Abraham Lempel dans l'article " A Universal Algorithm for Sequential Data Compression ". Le concept de fenêtre glissante est utilisé pour faire référence à des données précédemment lues. Si le mot ou la phrase en double, un pointeur de référence est défini sur l'itération précédente.[1]
L'algorithme LZ77 est le suivant :

1. Le fichier d'entrée est parcouru depuis le début, en lisant un seul caractère à la fois.
2. Un tampon, appelé fenêtre glissante, est maintenu, ce qui stocke l'occurrence des symboles précédents.
3. Lorsqu'un symbole répété est trouvé, il est stocké comme référence pointeur vers l'emplacement de l'occurrence précédente du symbole dans la fenêtre coulissante et le nombre de symboles qui sont assortis

2.1.6.2 Deflate

La contraction est l'algorithme évolutif développé par Phillip W. Katz. Il utilise à la fois les codecs LZ77 et Huffman pour compresser les données. Il est conçu l'origine pour le format de fichier .zip mais aujourd'hui, diverses versions du Deflate original sont utilisées pour différents formats de fichiers ou logiciels comme GZIP, 7-zip, etc.[1]

L'algorithme Deflate est le suivant :

1. L'ensemble du flux d'entrée est divisé en séries de blocs.
2. LZ77 est implémenté pour trouver les chaînes répétées dans chaque bloc et des pointeurs de référence sont insérés dans le bloc.
3. Les symboles sont ensuite remplacés par les mots-codes en fonction de leur occurrences dans le bloc à l'aide du processus de codage Huffman.

2.1.6.3 LZ78

LZ78 est un algorithme de compression de données sans perte introduit par Jacob Ziv et Abraham Lempel dans leur article de 1978 "Compression of Individual Sequences via Variable-Rate Coding". Ils sont à l'opposé du LZ77, ne se référant pas aux données précédemment vues dans le fichier de compression de données d'origine utilisent à la place un dictionnaire contenant les chaînes d'octets réellement rencontrées au début. Le dictionnaire a une entrée : l'entrée 0, qui représente une chaîne vide. Ainsi, avec le traitement des données, le dictionnaire est mis à jour [1, 5].
L'algorithme est relativement simple.

1. Le compresseur lit les prochains octets du fichier à compresser jusqu'à ce que la chaîne d'octets lue ne soit pas dans le dictionnaire.
2. Transmettre alors deux choses au décompresseur. Premièrement, il transmet l'index, dans le dictionnaire, de la chaîne composée des n-1 octets qui ont été lus (ou l'index 0 si n = 1). Ensuite, il transmet l'octet n.
3. le compresseur met à jour le dictionnaire en y ajoutant la chaîne composée des n octets qui ont été lus.

2.1.6.4 Lempel Ziv Markov Chain Algorithm (LZMA)

Cet algorithme a été introduit par Lempel Ziv Markov (LZMA) en 1998, C'est une sorte d'algorithme LZ77, utilisé pour la première fois dans 7zip. LZMA fournit un taux de compression élevé lors de la compression d'un flux d'octets inconnu. L'algorithme utilise une fenêtre glissante avec un filtre delta et un chiffrement de plage [9].
L'algorithme de LZMA est suivant:

1. La séquence d'entrée est transmise à un filtre Delta qui transmet la données sous forme de différence entre les symboles adjacents. Ainsi, le bit de sortie est la différence entre le bit actuel et le morceau précédent.
2. La sortie du filtre Delta est encodée par fenêtre glissante encodeur qui utilise un tampon de recherche pour faire des références comme dans le cas de LZ-77.
3. L'encodeur de plage encode les symboles avec des valeurs basées sur distribution de probabilité de chaque symbole. L'encodeur de gamme consistent en ces étapes :
   - Une grande plage intégrale est sélectionnée.
   - La distribution de probabilité de chaque symbole d'entrée est définie.
   - Diviser la plage entre les symboles proportionnellement à leur distribution de probabilité.
   - Chaque symbole est codé en divisant la plage actuelle jusqu'à la sous-gamme du symbole suivant.

2.1.7 Burrows-Wheeler transform (BWT)

L'algorithme BWT a été introduit par M. Burrows et D.J. Wheeler en 1994 dans l'article " A Blocksorting Lossless Data Compression Algorithm ". Le but de la méthode BWT est d'avoir un algorithme de compression presque aussi efficace que celui basé sur des modèles statistiques tout en étant aussi rapide que les algorithmes basés sur un dictionnaire, Contrairement aux autres algorithmes, le BWT fonctionne par bloc et non séquentiellement. [1].

L'idée générale est d'appliquer une transformation réversible à Bloc b pour produire un bloc b' facile à compresser. et pour regrouper les manifestations d'un même symbole. en d'autres termes, La probabilité de trouver le symbole $s \in \sum$ près d'une autre manifestation de s est Suite.
Afin de respecter la terminologie utilisée par les auteurs du BWT, considérons les symboles d'un alphabet $\sum$ comme étant des caractères et un bloc comme étant une chaîne de caractères S de longueur N. L'algorithme BWT est le suivant :

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

2.1.7.1 Move-to-front (MTF)

L'algorithme MTF (Move-to-front) effectue une conversion Il est réversible sur le bloc b pour produire un bloc b' facilement compressible. La transformation effectuée par l'algorithme MTF est également très efficace lorsqu'elle est appliquée sur un bloc préalablement transformé par l'algorithme BWT [1].
L'algorithme MTF transforme une chaîne L de longueur N en un vecteur R de longueur N également. La transformation se fait comme suit :

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

2.1.8 Prediction by Partial Matching (PPM)

PPM (prédiction de correspondance partielle ou prédiction de reconnaissance partielle) est un algorithme de compression de données apparu en 1984 qui se concentre sur la modélisation du contexte et sur les pronostics. L'idée principale de cet algorithme est de prédire le prochain code à encoder basé sur les n codes précédents. Puisque le contexte contient le nombre de symboles utilisés pour prédire le prochain symbole [1].
L'algorithme PPM est le suivant:

1. Un modèle de Markov d'ordre n est supposé.
2. Le fichier d'entrée (A) est parcouru depuis le début, en lisant un seul symbole à la fois.
3. Pour le symbole Ai , qui est à la position i du n précédent les symboles sont utilisés pour mettre à jour la probabilité de la séquence $B(Ai-n, Ai-n-1....Ai-1, Ai)$.
4. En supposant que B soit un symbole dans l'alphabet cible, mettez à jour tous les structures nécessaires.
5. Si l'un des symboles de B n'est pas présent, alors Ai est envoyé non codé ou B est raccourci puis ajouter à l'alphabet

2.2 Comparaison entre les techniques de compression

Dans cette section, nous allons voir la mesure de performances des algorithmes: RLE, Huffman, codage Huffman adaptatif et le codage arithmétique.

2.2.1 Mesure des performances de l'algorithme RLE

Étant donné que l'algorithme de codage de longueur d'exécution n'utilise aucune méthode statistique pour le processus de compression, Les temps de compression et de décompression, les tailles de fichier, le taux de compression et le pourcentage d'enregistrement sont calculés [1].
Plusieurs fichiers avec des tailles de fichiers et des modèles de texte différents sont utilisés pour le calcul.

2.2.2 Mesurer les performances des approches statiques de Huffman

Les approches Static Huffman Encoding et Shannon Fano sont implémentées et exécutées indépendamment. Pour ces deux approches, la taille des fichiers, les temps de compression et de décompression, l'entropie et l'efficacité du code sont calculé [1].

2.2.3 Mesure des performances du codage Huffman adaptatif

Le codage adaptatif Huffman est également implémenté afin de comparer avec d'autres algorithmes de compressions et algorithmes de décompression. Un mot de code dynamique est utilisé par cet algorithme. Tailles de fichiers, compression et les temps de décompression, l'entropie et l'efficacité du code sont calculés pour l'algorithme adaptatif de Huffman [1].

2.2.4 Mesure des performances de l'algorithme LZW

Comme cet algorithme n'est pas basé sur un modèle statistique, l'entropie et l'efficacité du code ne sont pas calculées [1].
Le processus de compression et de décompression, la taille des fichiers, le taux de compression et les pourcentages d'enregistrement sont calculés.

2.2.5 Mesure des performances de l'algorithme de codage arithmétique

Les temps de compression et de décompression et les tailles de fichier sont calculés pour cet algorithme. En raison de problème de débordement, le fichier d'origine ne peut pas être généré après le processus de décompression. Par conséquent, ces valeurs ne peuvent pas être considérées comme les valeurs réelles de l'algorithme de codage arithmétique. Donc, les résultats de ce algorithme ne sont pas utilisés pour le processus de comparaison [1].

Conclusion

Dans ce chapitre, nous avons vu les techniques de compression sans perte les plus connues et universellement utilisées, en mettant en évidence leurs algorithmes de compression, ainsi que la comparaison entre ces techniques en termes de vitesse de compression et décompression et de taux de compression.

Chapitre 3

Conception et Implémentation du système

Introduction

Pour faciliter le transfert de données à travers les réseaux ou en les partageant avec d'autres, et parce que ces données sont de grande tailles, deux étapes très importantes sont utilisées, à savoir la compression des données et la décompression des données par des algorithmes couramment utilisés.

Pour cela, nous aborderons dans ce chapitre une étude d'une méthode de compression et de décodage de données à travers d'un algorithme proposé.

Principalement, Le système fonctionne selon deux étapes:

1. Compression, et/ou
2. Décompression.

3.1 Compression des données

Le système commence par la lecture du fichier bloc par bloc (chaque bloc contient des valeurs dans l'intervalle [-128..127]). Ensuite, pour chaque bloc b calcule une matrice d'occurrence Occ_mat. Ce dernier, est utilisée afin de générer une liste de permutation L. Par la suite, cette lise est utilisée pour générer b' qui représente un bloque beaucoup plus compressible que b. Après cela, b' est compressé et le résultat de compression est écrit dans le fichier compresser cible. Le schéma 3.1 illustre le processus en détail.

Dans ce qui suit, nous allons pas expliquer les étapes de lecture et écriture parce-que ce sont des fonctions java pré-programmé.

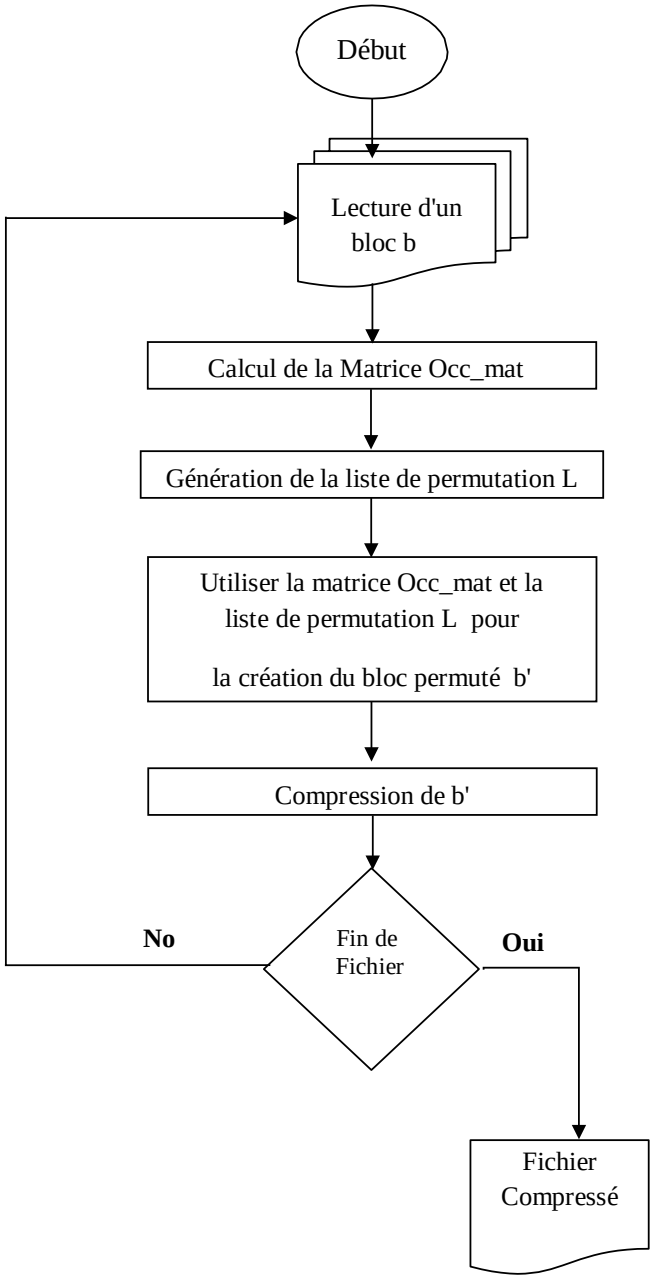

Figure 3.1: Etapes de compression.

3.1.1 Calcul de la matrice de permutation

Le Calcul de la matrice d'occurrence nécessite comme entrée le bloc b qui contient des valeurs dans l'intervalle [-128..127], et la taille des sous-blocs sbs=512. Les colonnes de la matrice représentent les chiffres de l'intervalle pré-décrit et les lignes représentent l'identificateur (l'ordre) du sous bloc dans le bloc b. Le processus consiste à diviser le bloc b en un ensemble des sous bloc de taille sbs. ensuite en calcule le nombre

d'occurrence de chaque chiffre et en l'affectant dans la colonne et la ligne courant. Le listing 3.1 illustre les étapes en détail.

```
static public int[][] createOccMat(byte[] buf, int len, int[][] mat, int sbs)
{
 int sbn=buf.length/sbs;

    if(((buf.length/(sbs+0.0))-sbn)!=0) sbn++;

 for(int i=0;i<mat.length;i++)
    {
  for(int j=0;j<mat[0].length;j++)
  {
  mat[i][j]=0;
  }

    }

 for(int i=0;i<sbn;i++)
 {
 int s=i*sbs;

 for(int j=s;(j<((i+1)*sbs))&&(j<len);j++)
 {
 mat[i][ Math.abs(buf[j])]++;
 }
 }

 return mat;
}
```

Listing 3.1: Fonction de calcule de matrice d'occurrence.

3.1.2 Génération de la liste de permutation

Jusque cette étape, nous avons la matrice d'occurrence. en utilisant ce dernier nous pouvons calculer la distance entre les sous-blocs afin de les réarranger et créer une autre séquence (liste de permutation) et évidemment un autre bloc b'. Alors la génération de la liste de permutation nécessite comme entrée la matrice d'occurrence et l'utilisation des deux fonctions: une qui calcule la distance entre deux lignes (voir le listing 3.2) et l'autre qui génère la liste de permutation (voir le listing 3.3).

```
  public static double doubleEuclideanDistance(int[][] mat, int p1, int p2)
  {
   long sum = 0;

   for (int i = 0; i < mat[p1].length; i++)
   {
   long d = mat[p1][i] - mat[p2][i];
   sum += d * d;

   }
```

```
    return  Math.sqrt(sum);
  }
```

Listing 3.2: Fonction de calcul de la distance.

```

public static int[] createMatchList(int[][] mat, int len)
{
double old_dist = Long.MAX_VALUE;
double new_dist = Long.MAX_VALUE;

int idx = -1;
int cl_idx = 0;

int[] list=new int[mat.length];


list[0]=0;
int ci=0;

while(ci!=-1)
{
idx = -1;
old_dist = Long.MAX_VALUE;
new_dist = Long.MAX_VALUE;


for (int i = 0; i < len; i++)
{
if((!contains(list, i))&&(ci!=i))
{
new_dist = doubleEuclideanDistance(mat, ci, i) ;
if(new_dist<old_dist) {old_dist=new_dist;idx =i;}
}
}


list[ci]=idx;
ci=idx;

}

return list;
}
```

Listing 3.3: Fonction de génération de la liste de permutation.

3.1.3 Génération du bloque permuté

En utilisant la liste de permutation générée précédemment et le bloc original b , nous pouvons créer un nouveau bloc b' comme l'illustre le listing 3.4 ci-dessous.

```

  public static byte[] createPrimeBuffer(byte[] src_buf,int len, int[] list, int sbs)
  {
   byte[] buf=new byte[src_buf.length];
   int i=0;
   int k=0;
```

```
7
8     int sbn=len/sbs;
9
10    if(((len/(sbs+0.0))-sbn)!=0) sbn++;
11    int n=0;
12
13    for(n=0; n<sbn;n++)
14    {
15    int s=i*sbs;
16
17    for(int j=s;(j<((i+1)*sbs))&&(j<len);j++)
18    System.out.print(" "+Math.abs(buf[j]));
19    System.out.println();*/
20
21    for(int j=s;(j<((i+1)*sbs))&&(j<len);j++)
22    {
23    buf[k]=src_buf[j];        //
24    k++;
25    }
26
27    i=list[i];
28
29    }
30
31    return buf;
32   }
```

Listing 3.4: Fonction de génération du bloc permuté.

3.1.4 Compression du bloc permuté

Dans cet étape, le système peut utiliser n'importe qu'elle méthode de compression (toutes les méthodes décrit précédemment peuvent être utilisées) afin de compresser le bloc permuté b'. Ce dernier est écrit/ajouté dans le fichier compressé.

3.2 Décompression des données

La décompression fait le processus inverse de la compression. Le système répète les deux étapes suivantes jusqu'à la fin du fichier compressé:

1. Lire la liste L,
2. Lire un bloc b',
3. générer le bloc original b en utilisant L et b'.

Le schéma 3.2 ci-après montre les différents étapes. En plus le listing 3.5 illustre en détail les étapes de régénération du bloc original b.

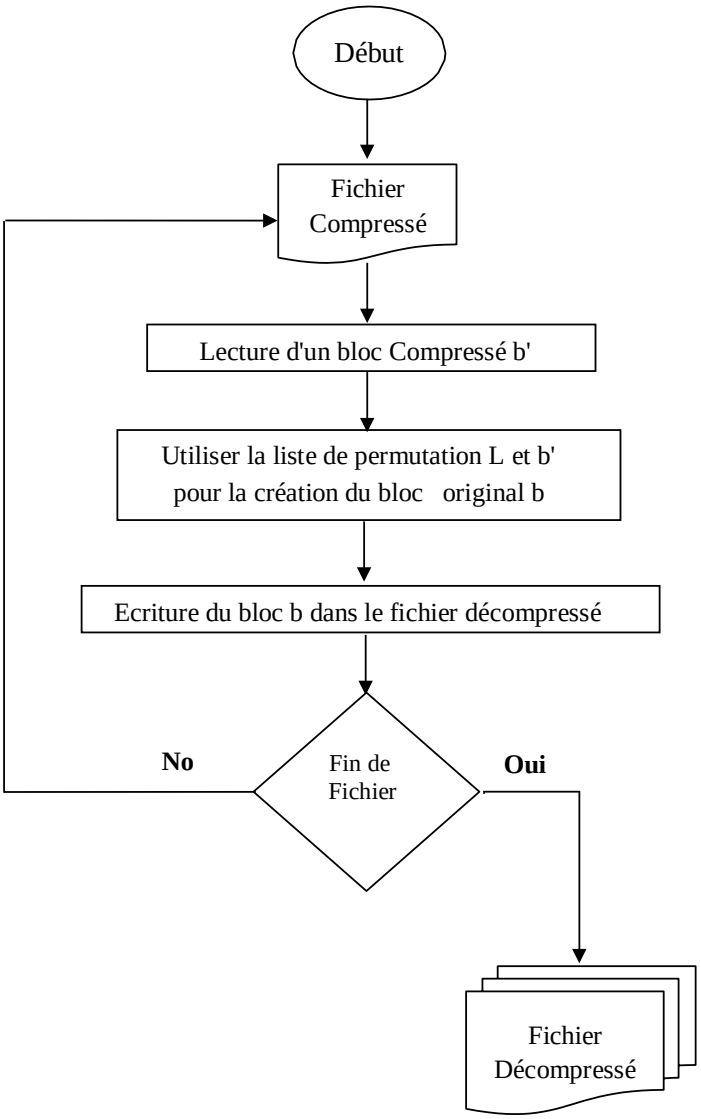

Figure 3.2: Etapes de décompression.

```
1   public static byte[] restoreOriginalBuffer(byte[] src_buf, int[] list, int sbs)
2   {
3    byte[] buf=new byte[src_buf.length];
4    int i=0;
5    int k=0;
6    int sbn=buf.length/sbs;
7
8    if(((buf.length/(sbs+0.0))-sbn)!=0) sbn++;
9
10   int n=0;
11
12   for(n=0; n<sbn;n++)
13   {
14   int s=i*sbs;
15
16   for(int j=s;(j<((i+1)*sbs))&&(j<src_buf.length);j++)
```

```
17      {
18      buf[j]=src_buf[k];        //
19
20      k++;
21      }
22
23      i=list[i];
24
25      }
26
27      return buf;
28    }
```

Listing 3.5: Fonction de génération du bloc permuté.

Conclusion

Dans ce chapitre, nous avons proposé la structure du système général. En plus nous avons vu en détail les deux principaux modes de traitement: compression et décompression. la méthode proposée est considérée comme une méthode de pré-traitement dans le but de rendre un fichier donné plus compressible.

Chapitre 4

Résultats de test

Introduction

Dans le chapitre précédent, nous avons vu l'architecture générale ainsi que les détails de conception du système proposé. Le dernier fonctionne en deux modes (compression et décompression) afin de réaliser sa tache.

Dans ce chapitre, nous allons voir le langage de programmation choisi, le jeu de données utilisé ainsi que les résultats de test obtenus.

4.1 Langage de programmation choisi

Java se démarque des autres langages de développement par ses fonctionnalités développées et son environnement sécurisé. Le langage est bien organisé et permet le développement d'applications performantes. Il possède les avantages suivantes:

- Une très bonne portabilité. La JVM (Java Virtual Machine) peut être utilisée dans tous les environnements (sous Windows, iOS, Linux...),
- Une programmation de haut niveau,
- Des concepts de programmation orientés objet,
- Un JDK riche avec de nombreuses bibliothèques,
- La création d'applications stables et fiables,
- Des IDE de qualité (Eclipse et Netbeans...),
- La possibilité d'implémentation et compilation,
- Une productivité importante,
- Un langage puissant.

En plus, le langage est très proche au langage de programmation C++ qui est considéré comme le plus rapide en exécution. Pour cela, la translation d'un code Java en code C++ ne pose pas des problèmes.

4.2 Jeu de données utilisées

Dans ce travail nous avons utilisé le jeu de données SILESIA pour le test. Il s'agit d'une collection de fichiers de différents types et tailles qui simule avec précision la grande variété de fichiers partagés sur Internet et elle est illustrée dans le tableau ci-dessous: tableau 4.1.

Tableau 4.1: Description des fichiers sur silesia corpus.

Nom du fichier	Type de données	Taille (Bytes)	Description
dickens	Texte en anglais	10,192,446	Travaux Collectés de Charles Dickens(du Projet Gutenberg)
mozilla	fichier exécutable	51,220,480	Exécutables tarrés de Mozilla 1.0 (edition Unix Tru64) (du projet Mozilla)
mr	Fichier image	9,970,564	Image médicale par résonance magnétique
nci	BDD	33,553,445	Base de données chimiques des structures
ooffice	Ficher exécutable	6,152,192	Un DLL du OpenOffice.org 1.01
osdb	BDD	10,085,684	Base de données chimiques des structures Exemple de base de données au format MySQL à partir du Benchmark Open Source Database
reymont	Fichier PDF	6,625,583	Texte du livre Chłopi by Władysław Reymont
samba	Fichier exécutable	21,606,400	Code source tarré de Samba 2-2.3 (du projet Samba)
sao	BDD Binaire	7,251,944	Le catalogue de Star SAO (des catalogues astronomiques et des formats de catalogue)
webster	Fichier HTML	41,458,703	Le dictionnaire intégral Webster de 1913 (du Projet Gutenberg)
xml	Fichier XML	5,345,280	fichier XML collecté
x-ray	Fichier Image	8,474,240	Image médicale aux rayons X

4.3 Résultats obtenus et discussion

Dans la phase de test nous avons utilisé 7-Zip V19.00 (x86). Nous avons appliqué les algorithmes dans les deux cas: sans et avec le processus de traitement proposé. Les tableaux libellés (A) représentent les résultats du premier cas, alors que les tableaux libellés (B) représentent les résultats pour le second cas.
Dans les résultats des tableaux nous pouvons voir clairement qu'il y a un gain de compression de 1.33 % et 1.66 % en utilisant la méthode de pré-traitement par rapport à sans pré-traitement les données dans le cas du fichier X-ray avec les méthodes Deflate et LZMA, respectivement.

Tableau 4.2: Performance of BZip2 on Silesia Corpus (A).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens	10192446	2799505	3.64	7.59	12.54
mozilla	51220480	17913191	2.86	10.24	30.17
mr	9970564	2441042	4.08	7.58	17.04
nci	33553445	1813142	18.51	4.37	19.70
ooffice	6152192	2862075	2.15	7.40	13.07
osdb	10085684	2802638	3.60	7.74	13.38
reymont	6627202	1246027	5.32	6.67	11.79
samba	21606400	4549793	4.75	8.10	26.28
sao	7251944	4940749	1.47	7.53	10.74
webster	41458703	8646000	4.80	7.54	18.61
x-ray	8474240	4051399	2.09	8.31	13.25
xml	5345280	440268	12.14	4.39	15.17

Tableau 4.3: Performance of Deflate on Silesia Corpus (A).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens	10192446	3685058	2.77	2.50	31.36
mozilla	51220480	18560507	2.76	4.09	58.92
mr	9970564	3485871	2.86	4.00	5.75
nci	33553445	3038975	11.04	3.37	86.48
ooffice	6152192	3010211	2.04	3.41	24.76
osdb	10085684	3628961	2.78	3.09	36.43
reymont	6627202	1705103	3.89	2.05	34.16
samba	21606400	5192981	4.16	3.72	56.30
sao	7251944	5244701	1.38	4.20	21.75
webster	41458703	11653560	3.56	2.35	52.51
x-ray	8474240	5763002	1.47	5.32	26.32
xml	5345280	667068	8.01	3.52	43.57

Tableau 4.4: Performance of Deflate64 on Silesia Corpus (A).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens	10192446	3511383	2.90	2.51	27.38
mozilla	51220480	18235900	2.81	3.83	63.60
mr	9970564	3436448	2.90	3.46	23.89
nci	33553445	2919417	11.49	3.51	81.01
ooffice	6152192	2957043	2.08	4.20	25.62
osdb	10085684	3462515	2.91	4.00	45.80
reymont	6627202	1640871	4.04	2.48	27.48
samba	21606400	4938127	4.38	4.54	68.68
sao	7251944	5195186	1.40	4.53	26.20
webster	41458703	11162230	3.71	3.20	64.29
x-ray	8474240	5712406	1.48	4.99	25.41
xml	5345280	639051	8.36	3.59	36.94

Conclusion

Dans ce travail, nous avons comparé les algorithme de compression sans perte en évaluant leurs performances sur le corpus Silesia. Cette évaluation a été faite selon deux cas: sans pré-traiter les données et avec pré-traitement des données.

Tableau 4.5: Performance of LZMA on Silesia Corpus (A).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens	10192446	2830582	3.60	1.21	18.00
mozilla	51220480	13506134	3.79	2.60	34.50
mr	9970564	2749292	3.63	2.01	27.72
nci	33553445	1980947	16.94	2.99	67.22
ooffice	6152192	2427630	2.53	2.46	18.39
osdb	10085684	2871232	3.51	2.25	29.32
reymont	6627202	1340515	4.94	1.43	31.92
samba	21606400	3849864	5.61	2.95	31.75
sao	7251944	4416870	1.64	2.78	12.48
webster	41458703	8798115	4.71	1.35	38.46
x-ray	8474240	4478468	1.89	2.62	14.48
xml	5345280	484814	11.03	3.70	22.26

Tableau 4.6: Performance of PPMd on Silesia Corpus (A).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens	10192446	2341442	4.35	4.30	3.69
mozilla	51220480	16455260	3.11	3.43	3.07
mr	9970564	2354047	4.24	4.93	4.34
nci	33553445	2300655	14.58	21.12	17.52
ooffice	6152192	2576909	2.39	2.30	2.14
osdb	10085684	2451301	4.11	4.04	3.42
reymont	6627202	1141461	5.81	6.86	6.97
samba	21606400	4196202	5.15	5.78	4.83
sao	7251944	4778574	1.52	1.39	1.27
webster	41458703	7068044	5.87	5.76	5.29
x-ray	8474240	4039189	2.10	2.05	1.70
xml	5345280	496463	10.77	7.36	9.39

Tableau 4.7: Performance of BZip2 on Silesia Corpus (B).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens_preprocessed	10243748	2898766	3.53	7.61	15.71
mozilla_preprocessed	51350038	18207268	2.82	10.25	29.52
mr_preprocessed	10112418	2516575	4.02	11.15	16.07
nci_preprocessed	33620488	2164591	15.53	3.59	20.71
ooffice_preprocessed	6172518	2905359	2.12	7.97	15.91
osdb_preprocessed	10112418	2897519	3.49	9.19	13.16
reymont_preprocessed	6697838	1325221	5.05	6.49	12.19
samba_preprocessed	21669458	4764972	4.55	7.99	23.84
sao_preprocessed	7354488	5019399	1.47	5.85	13.01
webster_preprocessed	41631618	9112630	4.57	7.37	15.58
x-ray_preprocessed	8536458	4124621	2.07	9.83	12.52
xml_preprocessed	5384538	506787	10.62	4.83	14.22

Tableau 4.8: Performance of Deflate on Silesia Corpus (B).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens_preprocessed	10243748	3824340	2.68	3.17	33.46
mozilla_preprocessed	51350038	19096871	2.69	4.74	69.07
mr_preprocessed	10112418	3615851	2.80	4.70	45.28
nci_preprocessed	33620488	3378283	9.95	4.05	71.73
ooffice_preprocessed	6172518	3123851	1.98	4.86	31.48
osdb_preprocessed	10112418	3715870	2.72	4.79	47.74
reymont_preprocessed	6697838	1826286	3.67	2.86	42.02
samba_preprocessed	21669458	5513199	3.93	4.51	73.02
sao_preprocessed	7354488	5341216	1.38	5.64	32.77
webster_preprocessed	41631618	12367290	3.37	3.54	70.40
x-ray_preprocessed	8536458	5667274	1.51	6.63	31.80
xml_preprocessed	5384538	739487	7.28	4.28	36.16

Tableau 4.9: Performance of Deflate64 on Silesia Corpus (B).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens_preprocessed	10243748	3628888	2.82	2.76	46.52
mozilla_preprocessed	51350038	18724400	2.74	4.39	71.39
mr_preprocessed	10112418	3546122	2.85	4.20	47.04
nci_preprocessed	33620488	3286149	10.23	3.88	79.76
ooffice_preprocessed	6172518	3040212	2.03	4.32	31.82
osdb_preprocessed	10112418	3527111	2.87	4.06	49.46
reymont_preprocessed	6697838	1719885	3.89	2.55	43.45
samba_preprocessed	21669458	5273010	4.11	4.17	75.98
sao_preprocessed	7354488	5288427	1.39	4.87	33.72
webster_preprocessed	41631618	11704875	3.56	3.15	75.34
x-ray_preprocessed	8536458	5633825	1.52	5.80	32.83
xml_preprocessed	5384538	698566	7.71	4.03	45.04

Tableau 4.10: Performance of LZMA on Silesia Corpus (B).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens_preprocessed	10243748	2898580	3.53	1.35	15.86
mozilla_preprocessed	51350038	14146013	3.63	2.87	29.66
mr_preprocessed	10112418	2798222	3.61	2.26	30.42
nci_preprocessed	33620488	2377053	14.14	2.98	46.20
ooffice_preprocessed	6172518	2498257	2.47	2.59	16.22
osdb_preprocessed	10112418	2970377	3.40	2.12	30.04
reymont_preprocessed	6697838	1381196	4.85	1.56	14.89
samba_preprocessed	21669458	4069047	5.33	3.25	48.51
sao_preprocessed	7354488	4547354	1.62	3.01	18.75
webster_preprocessed	41631618	9095608	4.58	1.49	39.39
x-ray_preprocessed	8536458	4418878	1.93	3.03	18.76
xml_preprocessed	5384538	538420	10.00	4.19	34.70

Tableau 4.11: Performance of PPMd on Silesia Corpus (B).

File Name	File Size (bytes)	Final Size	Compression Ratio	Compression Speed	Decompression Speed
dickens_preprocessed	10243748	2418663	4.24	4.13	4.02
mozilla_preprocessed	51350038	16713035	3.07	3.24	2.89
mr_preprocessed	10112418	2421772	4.18	4.27	4.40
nci_preprocessed	33620488	2401193	14.00	18.31	16.30
ooffice_preprocessed	6172518	2617025	2.36	2.18	2.06
osdb_preprocessed	10112418	2535455	3.99	3.85	3.18
reymont_preprocessed	6697838	1191928	5.62	7.22	7.07
samba_preprocessed	21669458	4380549	4.95	5.04	4.79
sao_preprocessed	7354488	4809357	1.53	1.44	1.30
webster_preprocessed	41631618	7454199	5.58	5.32	4.94
x-ray_preprocessed	8536458	4098127	2.08	1.95	1.85
xml_preprocessed	5384538	540205	9.97	10.21	9.07

Conclusion et perspectives

La compression de données, également appelée compactage, est le processus de réduction de la quantité de données nécessaires au stockage ou à la transmission d'une information donnée, généralement par l'utilisation de techniques de codage. Elle est très demandée vu l'augmentation exponentielle de la taille des données.

L'objectif de notre travail est de proposer une méthode de pré-traitement des fichiers dans le but de les rendre plus compressible. Le processus commence par la décomposition du fichier cible en un ensemble des bloque. Ensuite chaque bloque est divisé en un ensemble des segments de taille similaire (sbs=512). Par la suite, calcule une matrice d'occurrence qui contient le nombre d'occurrence de chaque nombre de l'intervalle [-128..127] dans les segments. Finalement, à l'aide de la matrice d'occurrence calcule une liste de permutation. Ce dernier est utilisé afin de changer la séquence des segments de données.

La méthode proposée est testée sur le benchmark Silesia, et l'aide des différents algorithmes de compression sans perte connus tels que : BZip2, Deflate, Deflate64, LZMA et PPMd. Les résultats obtenus sont très encourageants avec un gain de compression de 1.33 % à 1.66 %. Mais, il possible de faire d'autres améliorations au travail telles que :

- L'augmentation de la taille de la matrice d'occurrence (mais nécessite d'avoir un matérielles d'accélération,
- L'utilisation de la déviation standard au lieu du calcul de la différence entre les occurrences des valeurs.

Bibliographie

[1] G. A. Abandah, F. T. Jamour et E. A. Qaralleh – "Recognizing handwritten arabic words using grapheme segmentation and recurrent neural networks", *International Journal on Document Analysis and Recognition (IJDAR)* **17** (2014), no. 3, p. 275–291.

[2] S. Abe – *Support vector machines for pattern classification*, 2nd éd., Springer Verlag, 2010.

[3] A. A. Aburas et S. M. Rehiel – "Off-line omni-style handwriting arabic character recognition system based on wavelet compression", *Arab Research Institute in Sciences & Engineering* **3** (2007), p. 123–135.

[4] S. Alma'adeed, C. Higgens et D. Elliman – "Recognition of off-line handwritten arabic words using hidden markov model approach", *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3, IEEE, 2002, p. 481–484.

[5] L. Gavalakis et I. Kontoyiannis – "Fundamental limits of lossless data compression with side information", *IEEE Transactions on Information Theory* **67** (2021), no. 5, p. 2680–2692.

[6] A. Gopinath et M. Ravisankar – "Comparison of lossless data compression techniques", *2020 International Conference on Inventive Computation Technologies (ICICT)*, IEEE, 2020, p. 628–633.

[7] M. Goyal, K. Tatwawadi, S. Chandak et I. Ochoa – "Deepzip: Lossless data compression using recurrent neural networks", *arXiv preprint arXiv:1811.08162* (2018).

[8] K. Sayood – *Introduction to data compression*, Morgan Kaufmann, 2017.

[9] L. Wang – *Support vector machines: theory and applications*, vol. 177, Springer Science & Business Media, 2005.

[10] S. Yamagiwa, E. Hayakawa et K. Marumo – "Stream-based lossless data compression applying adaptive entropy coding for hardware-based implementation", *Algorithms* **13** (2020), no. 7, p. 159.