Grande Ecole de sciences, d'ingénierie, d'économie et de gestion de **CY Cergy Paris Université**

Programmation fonctionnelle

Projet : Méthodes de compression sans perte

Effectué du 12 Février au 31 Mars 2024

par

Patrice SOULIER, Quentin ROLLET, Marc-Antoine VERGNET, Valentin NOAILLES

Enseignant : *Romain DUJOL*

Contents

1 Introduction

Les algorithmes de compression, véritables énigmes mathématiques complexes, revêtent une importance cruciale dans le domaine de l'informatique. Alors que certains optent pour des méthodes de compression avec perte, particulièrement adaptées à des applications telles que la compression d'images, cette approche n'est pas toujours idéale, notamment lorsque la préservation intégrale des données est primordiale.

Dans ce rapport, nous explorerons minutieusement les mécanismes des algorithmes de compression sans perte. Ces derniers se distinguent par leur capacité à réduire la taille des données sans altérer leur contenu, une caractéristique essentielle dans de nombreux contextes. Nous aborderons les différentes phases du processus, de la conception à l'implémentation, en passant par une évaluation approfondie des méthodes existantes.

La première section de notre étude se penchera sur la conception des algorithmes, mettant en lumière la répartition des tâches et la mise en place rigoureuse des tests, deux aspects cruciaux dans le développement de solutions efficaces. Par la suite, nous plongerons dans l'implémentation, examinant en détail la méthode Run-Length Encoding (RLE), les approches statistiques, et les méthodes à dictionnaire.

La section suivante approfondira notre analyse en étudiant de près les différentes méthodes de compression, notamment le RLE, les méthodes statistiques avec une comparaison approfondie entre Huffman et Shannon-Fano, ainsi que les méthodes à dictionnaire, avec un examen détaillé des différences entre LZ78 et LZW. Nous conclurons cette section en identifiant les configurations les plus performantes au sein de chaque méthode.

Enfin, cette étude se clôturera par une conclusion éclairante, résumant les principales conclusions tirées de notre exploration approfondie des algorithmes de compression sans perte. À travers ce rapport, nous chercherons à offrir une vision complète et nuancée des différentes approches, permettant ainsi de guider les choix futurs dans le domaine de la compression des données.

2 Conception

2.1 Répartition des tâches

2.2 Mise en place des tests

La mise en place des tests permet d'assurer tout au long du projet que les fonctionnalités implémentées fonctionnent. Dans le cadre de ce projet ou l'on avait déjà la signature des différentes fonction à implémenter, il a été tout à fait naturel d'écrire les tests avant le développement de celles-ci. Même si ceux-ci peuvent être amenés à être changés car il n'est pas possible de tester les tests avant d'avoir les fonctions implémentés, cela permet de gagner un temps certain durant le projet car il permet de déceler imédiatements les différents bugs qui peuvent survenir.

Nous allons donc présenter les deux librairies de tests que nous avons utiliser dans le projet qui sont HUnit et QuickCheck.

2.2.1 Hunit

HUnit est une librairie de test pour Haskell qui est inspirée de JUnit pour Java. Cette librairie de test est utilisée pour faire des tests unitaires, c'est-à-dire des tests qui permettent de vérifier que des parties spécifiques du code fonctionne.

La plupart du temps, on teste des fonctions avec plusieurs valeurs d'entrées, de manière à faire en sorte que l'on passe par toutes les lignes de code de la fonction. Il faut alors varier les paramètres d'entrés afin d'essayer toutes les conditions présentent dans le code.

Il faut donc aussi préciser à HUnit quelle est la valeur de sortie attendue. HUnit se charge alors de nous prévenir si la valeur de sortie de la fonction n'est pas celle que l'on voulait.

2.2.2 QuickCheck

La philosophie de QuickCheck est différente de HUnit, ce qui apporte une complémentarité entre les deux librairies de tests.

En effet, QuickCheck est une librairie de test permettant de tester des propriétés sur des entrées aléatoires. Dans le cas de notre projet, on aimerait tester sur un grand nombre de messages différents la succession des l'algorithme de compression puis l'algorithme de décompression. Comme on est sur des algorithmes sans pertes de données, le message en entrée et le message en sortie doivent être identiques.

QuickCheck permet facilement de faire ceci. Il suffit de lui donner la structure de données adéquate et il est capable de générer aléatoirement cette structure de données. Il va ensuite appliquer les fonctions à appliquer puis de tester une propriété sur les données en sortie.

2.3 Conclusion

Nous pouvons donc voir que les librairies de tests permettent d'assurer tout au long du développement que les fonctions que nous programmons ne comportent pas d'erreur et que le fait d'utiliser plusieurs librairies de tests est complémentaire en permettant de tester tout le code grâce aux tests unitaires, et une utilisation du code normal grâce aux tests aléatoires.

3 Implémentation

3.1 Méthode RLE

3.2 Méthodes statistiques

3.3 Méthodes à dictionnaire

4 Étude des méthodes de compression

4.1 Méthode RLE

La méthode RLE (Run-Length Encoding) est probablement la manière la plus simple pour un algorithme de compression sans perte. En effet, le principe est simple. Il suffit de compter le nombre d'occurence d'un caractère consécutifs, et d'encoder ce nombre d'occurence avec le caractère. La chaine de caractère äaaabbcbbbd̈evient alors [(a,4), (b, 2), (c, 1), (b,3)].

Cependant, cette méthode possède un assez gros défaut car elle ne peut pas compresser tout ce que l'on souhaite. Par exemple, il est très rare dans un texte qu'il y ait un grand nombre de caractères consécutifs. Lors de la compression, la sortie sera plus longue que l'entrée, ce qui n'est pas vraiment souhaitable. Par contre, il peut être assez efficace dans certains cas où il y a un grand nombre de données consécutives. Étudions ceci dans l'exemple suivant.

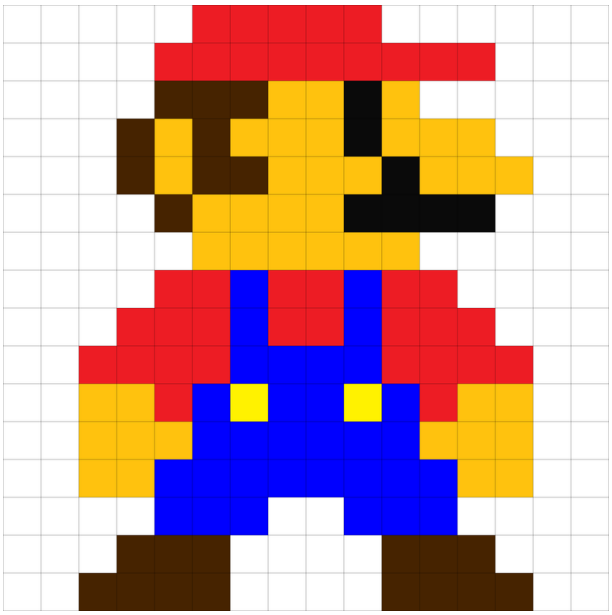

Figure 1: Mario 8-bits

Dans le cas du Mario en 8-bits, une représentation possible de l'encodage des bits pourrait être exprimée comme suit : [(5, blanc), (5, rouge), (10, blanc), ...]. Pour la première image, comprenant 256 pixels avec chaque pixel possédant 3 données (RGB), l'application de la compression RLE permet de réduire cela à seulement 75 changements de couleurs. Ainsi, la liste résultante a une longueur de 75, avec chaque élément comportant 4 données (le nombre de pixels identiques, RGB).

La première image a donc une taille de 768 bits, tandis que sa version compressée ne pèse plus que 300 bits. En conséquence, l'image compressée occupe environ 0.4 fois la

taille de son équivalent initial.

4.2 Méthodes statistiques

4.2.1 Comparaison Huffmann/Shannon-Fano

La différence entre Huffman et Shannon-Fano est l'approche du codage. L'approche de Shannon-Fano est descendante, c'est-à-dire que l'on commence par le symbole le plus fréquent et on descend jusqu'au symbole le moins fréquent. L'approche de Huffman est ascendante, c'est-à-dire que l'on commence par les symboles les moins fréquents et on remonte jusqu'au symbole le plus fréquent.

Le problème de l'approche descendante, c'est que l'on ne peut pas garantir que le code obtenu soit optimal. :En effet, lorsqu'il n'est pas possible de diviser les symboles en deux groupes de fréquences égales, on ne peut pas garantir que le code obtenu soit optimal. En revanche, l'approche ascendante garantit un code optimal.

En pratique, on décide plus souvent d'utiliser Huffman que Shannon-Fano car Huffman garantit un code optimal. Cependant, dans le cas où l'on a une liste de symboles triée par fréquence, Shannon-Fano est plus rapide à implémenter que Huffman. De plus, il est généralement plus facile de comprendre le fonctionnement de Shannon-Fano que celui de Huffman.

4.2.2 Meilleures configurations

Pour Huffman, les meilleurs configurations possibles sont celles qui permettent d'obtenir un arbre binaire parfait. C'est-à-dire que l'arbre est complet et que toutes les feuilles sont à la même profondeur. Cela permet d'obtenir un code optimal. Pour obtenir un arbre binaire parfait, il faut que le nombre de symboles soit une puissance de 2. Par exemple, la chaine de caractères "abracadabra" contient 5 symboles différents, il est donc impossible d'obtenir un arbre binaire parfait. Mais la chaine de caractères "perfect tree" contient 8 symboles différents, il est donc possible d'obtenir un arbre binaire parfait.

Dans le cas de Shannon-Fano, il faut que les symboles soient triées par fréquence pour obtenir un code optimal. Cependant, il est possible d'obtenir un code optimal sans que les symboles soient triées par fréquence. Cela dépend de la distribution des fréquences des symboles.

4.3 Méthodes à dictionnaire

4.3.1 Comparaion LZ78/LZW

4.3.2 Meilleures configurations

4.4 Conclusion

5 Conclusion