Rapport

La compression

Lucas Labadens Isabelle Marino

Le 10 mai 2016

Introduction

La compression de données ou codage de source est un processus informatique permettant de transformer un document sous forme binaire en un autre document du même contenant un suite de bit plus courte que le précédent mais pouvant restituer les mêmes informations en utilisant un algorithme de décompression propre à algorithme de compression qui fut utilisé. En d'autre termes, la compression raccourcit la taille des données. La décompression est l'opération inverse de la compression.

Il y a notamment 2 grand type de compression :

— la compression sans perte de données

— la compression avec perte de données

Un algorithme de compression sans perte restitue après compression et décompression un document strictement identique à l'originale. Les algorithmes de compression sans perte sont utiles pour les documents, les archives, les fichiers exécutable ou les fichiers texte. Pour la compression de données sans pertes, on distingue principalement deux types de codage : le codage entropique et le codage algorithmique.

Avec un algorithme de compression avec perte, la suite de bits obtenue après les compression et décompression est différente de l'originale, mais l'information restituée est très proche. Les algorithmes de compression avec perte sont utiles pour les images, le son et la vidéo.

Les formats de données tels que Zip, RAR, gzip, MP3 et JPEG utilisent des algorithmes de compression de données.

La compression est un procédé très utiliser dans la vie courante pour par exemple envoyer certains document par e-mails ou pour le skockage de documents sur disque dur ou cloud center. Le but de notre projet est de compresser des fichiers sans perte de données. Nous allons donc vous présenter différents algorithmes de compression sans perte de données que nous avons coder puis tester. Dans la première partie on présentera entre autres le codage de Huffman statique et celui de Lempel-Ziv.

Pour cela nous avons choisit d'utiliser le langage Java pour nous permettre d'avoir des classes structurées et coder les différents algorithmes tout en gardant une structure et des fonctions communes.

Algorithme de compression

Huffman statique

Définition et Exemple

Défnition

L'algorithme d'huffman statique est un algorithme de compression qui repose sur la redondance des caractères.En effet dans un fichier certain caractère sont plus présent que d'autre par exemple dans un fichier texte en français on retrouve souvent beaucoup de 'e' et 'a' mais très peu de 'w'.Un caractère étant codé sur octet l'algorithme d'huffman code un caractère non plus sur un octet mais sur un nombre de bit,plus le caractère est récurent plus le nombre de bit utilisé pour l'encoder sera faible.Cette algorithme neccéssité deux lecutres du fichier pour compresser,une pour compter les caracères l'autre pour écrire la compression,mais il permet à la décompression d'avoir déjà accès au codage propre a chaque lettre.Pour cela on utilise un arbre binaire.On compte le nombre de fois qu'apparait un caractère dans un fichier et ce nombre sont poids.Une fois l'ensemble des caractères du fichier comptez.L'ensemble des caractères muni de leur poids constitueront les feuilles de l'arbre de compression.On relie ensuite les deux feuilles les plus faibles avec un node intermédiaire qui aura pour poids la somme des poids de ces deux feuille, puis l'on répète l'opération avec les feuilles restante en reliant toujours les deux poids les plus faibles.Puis l'on relie les nodes racines entre elle toujours en reliant d'abord les plus faibles jusqu'a ne plus avoir qu'un seul et unique arbre. Ainsi chaque feuille ayant un chemin unique celui servira de code pour le caractère.En partant de la racine si l'on va au fils gauche on ajoute le bit 0 et si l'on va vers le fils droit on ajoute le bit 1. Ainsi les caractère les plus redondants se trouvant en haut de l'arbre ,ils seront codé sur très peu de bit.

Compression

Pour cela on utilise un arbre binaire.On compte le nombre de fois qu'apparait un caractère dans un fichier et ce nombre sont poids.Une fois l'ensemble des caractères du fichier comptez.L'ensemble des caractères muni de leur poids constitueront les feuilles de l'arbre de compression.On relie ensuite les deux feuilles les plus faibles avec un node intermédiaire qui aura pour poids la somme des poids de ces deux feuille, puis l'on répète l'opération avec les feuilles restante

en reliant toujours les deux poids les plus faibles.Puis l'on crée un nouvel arbre en fusionnant les deux arbres ayant les racines de poids plus faible et à ce nouvel arbre on le fusionne avec l'arbre qui possède la racine de poids la plus faible et réitère l'opération jusqu'a qu'il ne reste aucun autre arbre.L'arbre obtenue suite à cela constituera notre arbre de compression. Ainsi chaque feuille ayant un chemin unique celui servira de code pour le caractère.En partant de la racine si l'on va au fils gauche on ajoute le bit 0 et si l'on va vers le fils droit on ajoute le bit 1. Ainsi les caractère les plus redondants se trouvant en haut de l'arbre ,ils seront codé sur très peu de bit.On écrit ensuite l'arbre de compression au début du fichier de compression.Puis on relis le fichier à comprimer en réécrivant chaque caractère avec leur nouveaux code dans le fichier de comprimé.

Decompression

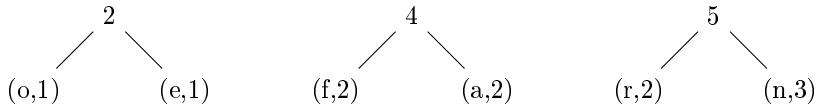
Pour décompresser un fichier comprimer avec l'algorithme d'Huffman statique,il suffit de récupérer l'arbre écrit en début de fichier puis de lire le fichier bit à bit.Si le bit lu est un 0 on se déplace à gauche dans l'arbre et à droite si c'est 1.Lorsqu'on arrive sur une feuille on écrit le caractère correspondant à cette feuille puis l'on retourne à la racine et on recommence l'opération jusqu'à avoir lu entièrement le fichier comprimer.Le chemin menant à une feuille étant unique on retrouve exactement le même fichier qu'au départ.

Exemple

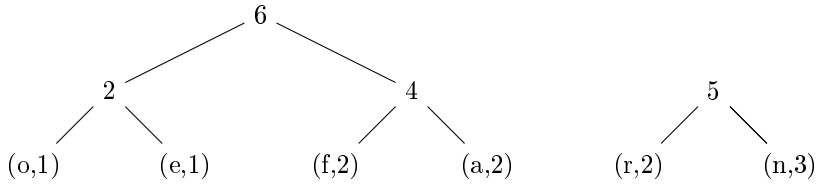
Prenons un fichier où il est écrit "fanfaronner".Les carctères présent sont 'f','a','r','o',et 'e'.Pour l'instant chaque caractère est codé sur un octet donc le mot pèse 11 octet soit 88 bit. On crée ensuite une feuille pour chaque caractère que l'on trie par ordre croissant de répétition :

(o,1) (e,1) (f,2) (a,2) (r,2) (n,3)

On relie ensuite les plus faibles entre eux :



On fusionne les deux arbres les plus faibles :



Lempel-Ziv

Définition

L'algorithme de compression de Lempel-Ziv est un codage algorithmique. Le codage algorithmique n'a pas besoin de transmettre des informations autres que le résultat du codage. Donc il n'y a pas besoin de transmettre dans le fichier compressé un code pour décompresser celui-ci. Il y a donc un autre avantage à ce style de compression : la lecture unique du fichier source. En effet, comme l'algorithme ne se base pas sur la récurrence de modèle dans le fichier source, nous pouvons lire et écrire le fichier compresser en même temps, ainsi cette algorithme nécessite une unique lecture du fichier source, ce qui est plutôt intéressant lors de la compression de gros fichier en termes de gain de temps. Cette algorithme s'applique uniquement aux fichiers binaires, c'est à dire à tout fichier fichier dont les symboles sont représentés sur des bits, et donc par conséquent à tout type de document.

Compression

Le codage de la compression s'effectue avec un arbre binaire dont les nœuds sont étiquetés par des entiers. Chaque nœud correspond à un entier i. On début avec une unique racine étiquetée à 0. A l'étape i, on part de la racine on lit un bit et on se déplace à gauche ou à droite si cela est possible. On se déplace vers la gauche si on lit un 0 et vers la droite si on lit un 1. Puis on continue de lire bit à bit jusqu'à ne plus pouvoir se déplacer. Lorsque l'on ne peut plus se déplacer on crée un nouveau nœud fils au nœud courant qu'on numérote i et on écrit le numéro du nœud courant sur $\lceil log_2(i) \rceil$ suivant du 0 ou 1 que l'on est en train de lire. On retourne la racine et on passe à l'étape i+1.

Pour compresser, j'ai fais une fonction qui analyse les bits lu, elle permet de descendre à gauche si on lit et 0 et à droite si on lit un 1, comme expliquer précédemment. Ensuite si on ne peut pas descendre on lit l'entier du nœud sur lequel on se trouve. Une fonction le traduit en 0/1 sur le nombre de bit nécessaire. Ce nombre dépends du nombre de nœud présent dans l'arbre, ie $\lceil log_2(i) \rceil$ où i est le nombre de nœud dans l'arbre. On écrit alors cette traduction puis le dernier bit lu dans le fichier compressé. On répète cette opération jusqu'à ce qu'on ait plus de bit à lire dans le fichier source. Enfin, comme un fichier à un nombre de bit qui est un multiple de 8, car les fichiers sont lu comme des

suites d'octets et non seulement de bits (1 octet = 8 bits), alors nous devons complété le fichier pour pouvoir écrire e dernier octet. Nous avons donc choisit la convention de mettre 1 puis le nombre de 0 nécessaire.

Décompression

On crée au fur et à mesure de la lecture bit à bit un tableau à 2 dimensions Exemple : t[i]= [nœud][bit] où nœud et bit sont des entiers, nœuds et le nombre lu à la i ème étape et bit le bits lu juste après. On commence en initialisant la première case à bit=-1 et nœud =-1 , pour pouvoir donner un repère lors de l'écriture de la décompression, il représente la racine de l'arbre de compression. A l'étape i, on lit n=$\lceil log_2(i) \rceil$ et et le bit suivant. On stock alors le nombre écrit et et bit suivant.On remonte dans le tableau grâce à n, on lit et le bit stocker dans t[n] et on regarde t[n.nœud] jusqu'à arriver à t[0]. On écrit dans le fichier décompresser la suite de bit lu au fur et à mesure.

Pour décompresser, j'ai crée une fonction qui me permet de lire bit à bit mon fichier. Je lis alors mes bits en fonction de la puissance de 2 souhaité, puis je les met dans un tableau pour pouvoir les traduire en un entier i. Je trouve ma puissance de 2 en fonction du nombre de nœud déjà lu et stocké précédemment. Une fois l'entier i trouvé, je recherche dans mon tableau à 2 dimensions à partir de i et je regarde la case du père afin de récupérer tous les 0/1 de la suite. Je les retourne pour les copier dans le fichier décompressé ensuite je lis un dernier bit que je copie aussi dans le fichier décompresser. Je stocke à la fin de mon tableau ce nouveau nœud, composée de son père (l'entier i) et du dernier bit lu. Je refais cette procédure jusqu'à la fin du fichier.

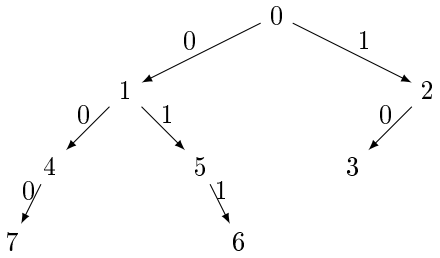
Exemple

Nous allons regarder le fichier composé de 2 caractère "ab".

Compression

En terme d'octet "ab" est représenter par : 01100001 01100010
On lit de la façon suivante pour avoir l'arbre ci dessous :

0 1 10 00 101 100 01 0



Il y a alors dans le fichier compressé on a (sans les "." et les "|") :
0|0.1|10.0|01.0|001.1|101.1| 100.0|010.0 On complète enfin le dernier octet avec 1 et le nombre de 0 nécessaire.

Décompression

Reprenons cette exemple. Nous avons donc dans notre fichier compressé :
00110001 00011101 11000010 01000000
Nous avons une liste composé de nœud avec 2 éléments un entier pour le père et un entier pour le bit représenté sur le flèche de l'arbre de compression. Nous initialisons le nœud 0 avec -1 en valeur pour le père et -1 en valeur pour le bit. Puis on lit, le nombre d'octet en fonction du nombre de nœud dans la liste (toujours selon les puissance de 2). Donc au commencement on lit 0 bit car il y a 2^0 nœud dans la liste, alors on ajoute le nœud 1 avec comme père 0 et comme bit le prochain bit lu, ici 0. On ajoute alors 0 dans le fichier décompressé.
On continue, on lit alors le bit 0 qui sera le père et 1. On ajoute le nœud 2 0,1 dans le liste.On ajoute alors 1 dans le fichier décompressé.
On lit les bit 10 qu'on traduit en 2 en nombre décimale on ajoute alors le nœud 3 2, 0. On parcours alors la liste on la voir le père 2 on lit son bit et on l'ajoute dans le fichier décompressé jusqu'à arrivé au père 0 . On ajoute 1 dans le fichier décompressé.

On obtient de cette fonction la liste suivante :

noeud	0	1	2	3	4	5	6	7	8
père	-1	0	0	2	1	1	5	4	2
bit	-1	0	1	0	0	1	1	0	0

Pour écrire à l'étape 7, on regarde le père 4 puis le père 1 puis le père 0. On récupère leur bit respectif et on les écrit en commençant par le nœud 1 à jusqu'à celui du bit du nœud 7. On obtient alors la suite : 000. Pour le nœud 8 on obtient la suite : 10.

Première partie

Analyse des performances

Nous avons choisit un certain nombre de tests à effectuer identiques sur chaque algorithme. Nous commençons par des tests avec des suites théoriques puis nous ferons des tests avec des fichiers plus réaliste des fichiers textes(roman) son et image.

Tout d'abord un test de compression théorique simple avec un fichier et un seul caractère à l'intérieur, nous avons pris le caractère "0". Tout d'abord dans le fichier il a y un unique octet de 0, puis nous dupliquons ce a 10 fois, 20 fois ... Et ainsi de suite pour avoir 10 fichiers de 1 octet , 10, 20 ,30 ,... octet, jusqu'à 10 Mo. Puis nous continuons avec des suites plus complexe comme une suite périodique composé que de suite de 01, une suite générer aléatoirement et enfin une suite de Champarnown.

Enfin, une dernière comparaison entre les différents algorithmes avec comme fichier la bible, pour pouvoir comparer les performances entre les algorithmes.

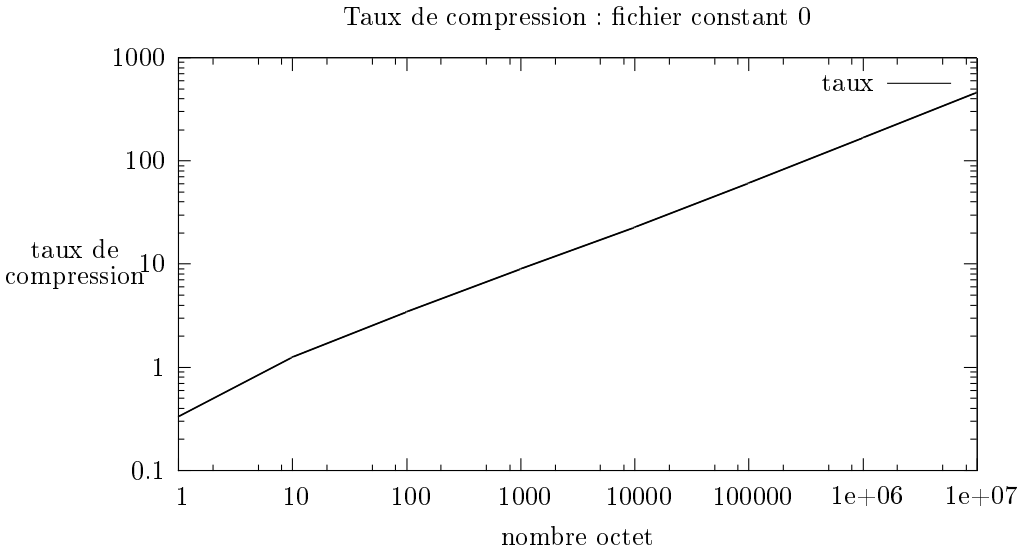
Nous utiliseront ces différents tests pour analyser les performance en terme de taux de compression, ainsi qu'en temps d'exécution pour la compression et la décompression.

Analyse du taux de compression

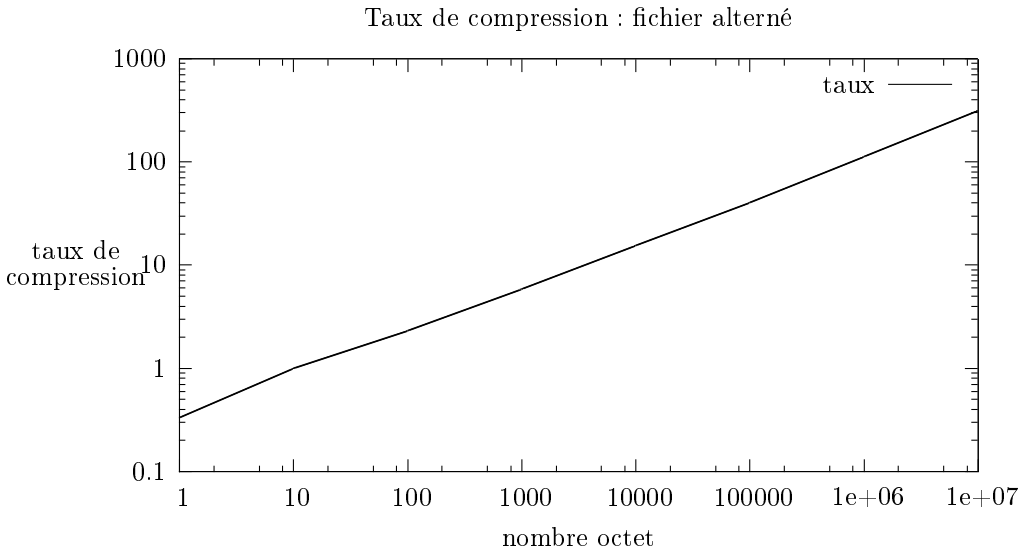
Analyse de l'agorithme de Huffman

Analyse de l'agorithme de Lemple Ziv

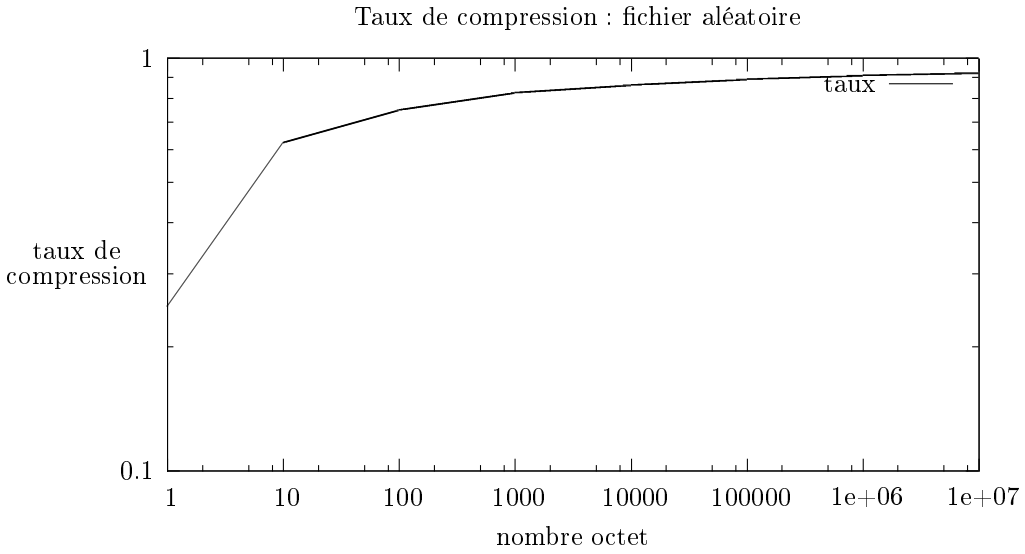
Tout d'abord regardons nos fichiers composés que de 0, nous obtenons le graphique ci-dessus.



On constate que la courbe est significative plus le nombre d'octet est important plus le taux de compression augmente, ainsi le fichier compressé devient réellement de plus en plus petit. On remarque néanmoins que lorsque l'on veux compressé un seul octet le taux de compression est en dessous de 1, c'est-à -dire que le fichier compressé est plus gros que le fichier source, ceci s'explique notamment par le stockage du premier octet supplémentaire pour connaître le mode de compression, ici Lempel-Ziv. Il s'explique aussi car le début de compression pour Lempel-Ziv remplace un bit par 2 et ainsi de suite. Nous devons donc attendre les 10 octets dans le fichier source pour que la compression ai vraiment lieu.



POur la suite alterner....



Analyse du temps d'exécution

Temps d'exécution pour la compression

Temps d'exécution pour la décompression

Différences entre les algorithmes