BARBETTE Justin
RAMIRES Thibault

1A Apprenti

Compte Rendu

Algorithmique

Compression de fichiers par codage de Huffman

I. Table des matières

II. Introduction

On s'intéresse dans ce TP à la compression de fichiers textes selon le codage de Huffman.

Ce codage consiste à observer la fréquence ces caractères dans un fichier et à associer à chaque caractère un code à longueur variable contrairement au code ASCII dans lequel la longueur du code de chaque caractère est d'un octet.

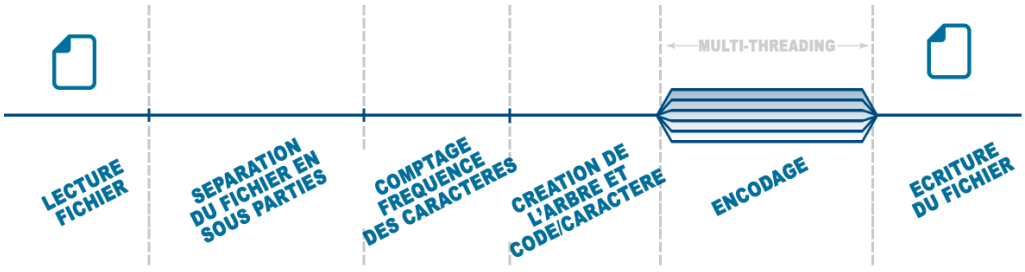
En associant ainsi chaque caractère fréquemment présent un code de longueur inférieure à un octet, on parvient à compresser un fichier.

Nous avons décidé pour ce projet de prendre comme critère principal la rapidité du programme de compression, et nous avons donc réalisé 2 choix techniques importants :

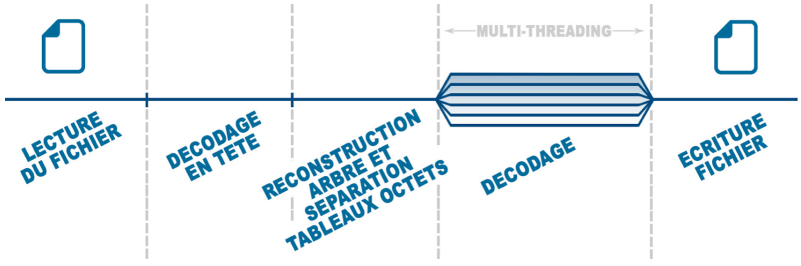
- Le chargement en RAM de chaque fichier à compresser / décompresser afin de limiter au maximum les accès disques qui sont très couteux.
- L'utilisation de plusieurs threads pour les phases « d'encodage » et de « décodage » de notre fichier.

Ces choix techniques ont pour conséquence d'engendrer une importante consommation de mémoire vive, ainsi que l'utilisation du processeur au maximum de ses capacités pendant le multithreading. Cependant cela nous permet d'avoir une vitesse de compression / décompression inégalée.

Étapes de la compression



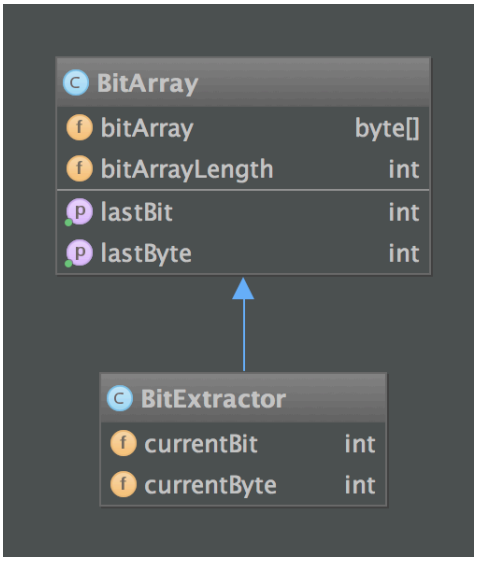
Étapes de la décompression



III. Structures de données

Comme indiqué dans l'énoncé, nous cherchons à obtenir la meilleur performances temporelle possible. Pour ce faire, nous somme descendus au plus « bas niveau » possible, en n'utilisant qu'une seule collection, la file de priorité pour la création de l'arbre de Huffman.

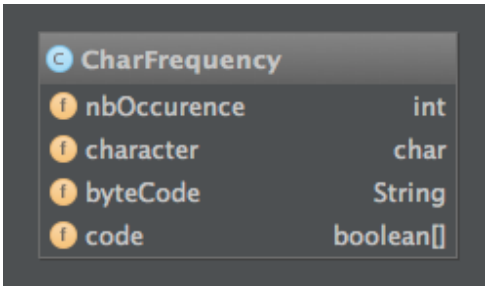
a. La gestion des codes à longueur variables



L'une des principales difficultés de ce programme est de gérer le caractère « variable » de la longueur des codes de Huffman des caractères. Pour traiter ce problème nous avons conçu 2 classes :

- BitArray : qui encapsule l'ajout des codes des caractères de Huffman dans un tableau d'octets au cours de la compression, et l'ajout de caractères lors de la décompression. Cette classe gère automatiquement l'extension de taille du tableau d'octet.
- BitExtractor : qui encapsule la lecture 1 à 1 lors de la décompression des caractères de Huffman au sein d'un tableau d'octets.

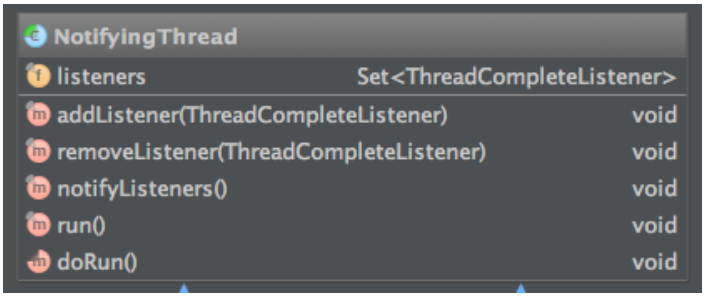
b. Gestion des caractères



Nous avons décidé d'encapsuler la gestion de nos caractères au sein d'une même classe.

Cette classe nous permet de les manager à la création de l'arbre (ou cette classe fait office de valeur portée par les feuilles de l'arbre), comme à l'encodage et au décodage, grâce à la présence à la fois du caractère, de son nombre d'occurrence et de son code de Huffman.

c. Gestion des threads



Afin de faciliter la gestion de nos threads, nous avons mis en place une classe parente à nos threads.

Cette classe permet à nos threads de notifier un ensemble de listeners lorsqu'ils ont fini leur tâche.

IV. Calculs de couts des algorithmes

a. Lecture et écriture dans les fichiers

Nous considérons que le coût de la lecture / l'écriture d'un fichier est linéaire en le nombre d'octets à lire / écrire : O(n) avec n le nombre d'octets.

b. Comptage de la fréquence des caractères

Nous parcourons le tableau d'octets (représentant le fichier à compresser) octet par octet, et réalisons qu'une seule opération : l'incrémentation d'un entier, le coût est linéaire en le nombre d'octets à traiter : O(n) avec n le nombre d'octets.

c. Création de l'arbre et du code de Huffman de chaque caractère

Nous utilisons une file de priorité qui a un coût d'ajout et de retrait logarithmique en le nombre d'éléments dans la file. Nous obtenons finalement un cout en O(n*log(n)) avec n le nombre de caractères (voir annexe).

Pour créer le code de Huffman de chaque caractère, nous faisons simplement un parcours complet de tout l'arbre, qui coute est linéaire en le nombre de nœuds dans l'arbre. Ici nous savons que nous avons n caractères, donc n feuilles, ce qui entraine $2^{n-1}-1$ nœuds dans l'arbre et donc un cout en : $O(2^n)$.

d. Encodage

Nous parcourons à nouveau le tableau d'octets (représentant le fichier à compresser) octet par octet, et réalisons l'ajout de chaque bit du code de Huffman associé à chaque caractère dans un tableau de byte. Nous obtenons finalement un cout en : O(n) avec n le nombre d'octets (voir annexe).

e. Décodage

Nous parcourons le tableau d'octets (représentant le fichier à décompresser) bit par bit et réalisons la conversion des codes de Huffman retrouvés en le caractère original. Nous obtenons finalement un cout en : O(n) avec n le nombre de bits (ou d'octets) (voir annexe).

f. Cout total du programme

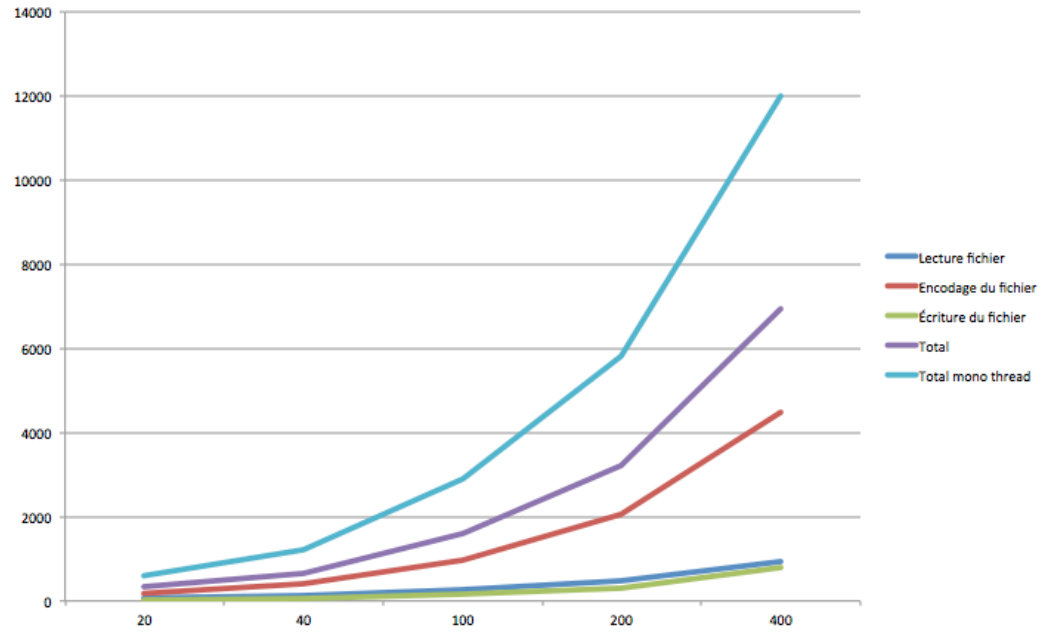
Dans le cas de la compression d'un fichier de texte, respectant les fréquences des caractères de la langue du fichier, le nombre d'octets est très grand devant le nombre de caractères, et on obtient un cout en : O(n) avec n le nombre de caractères.

V. Résultats pratiques

Pour étudier les performances de notre programme, nous avons utilisés des fichiers de taille variable, contenant des textes respectant les fréquences d'apparition des lettres.

Nous avons procédé à ces tests sur des fichiers en langue française et anglaise, et nous avons observé des résultats similaires en terme de taux de compression ainsi qu'en terme de vitesse d'exécution.

a. Compression et décompression



Graphique de performance du programme. Taille des fichiers en Mo en abscisse, temps d'exécution en ms en ordonnée

Nous observons que nous retrouvons bien un cout linéaire, pour la lecture/écriture, l'encodage, l'ensemble du programme (l'axe des abscisses étant en échelle pseudo logarithmique). (Les résultats de la décompression sont similaires).

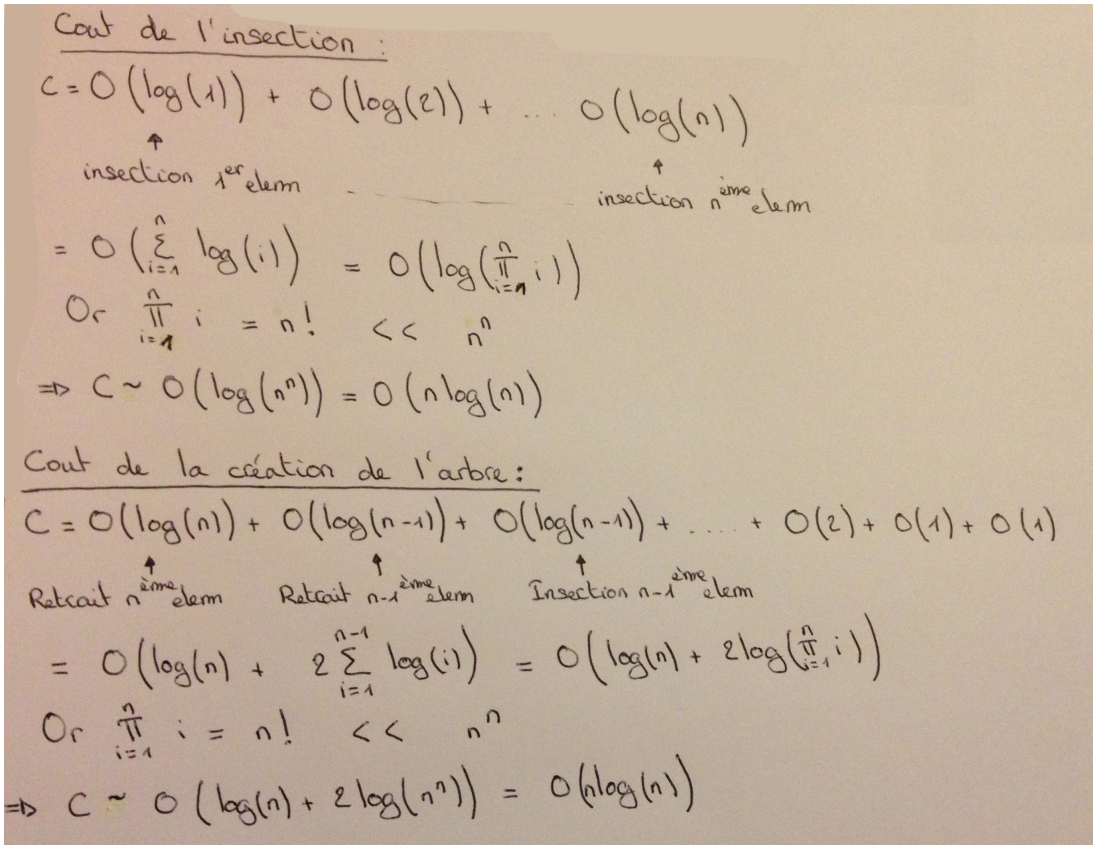
On observe également que le multithreading sur l'encodage est efficace est permet de diviser par 2 le temps d'exécution du programme.

Nous testons ici notre programme sur des fichiers de taille commençant à être importante, il faut spécifier à java d'allouer au moins 2048Mo de mémoire à notre programme via l'option –Xmx2048m. On pourrait diminuer cette quantité fortement en forçant l'exécution du GarbageCollector, notamment à la fin de la lecture du fichier, cependant, cela nuirait aux performances.

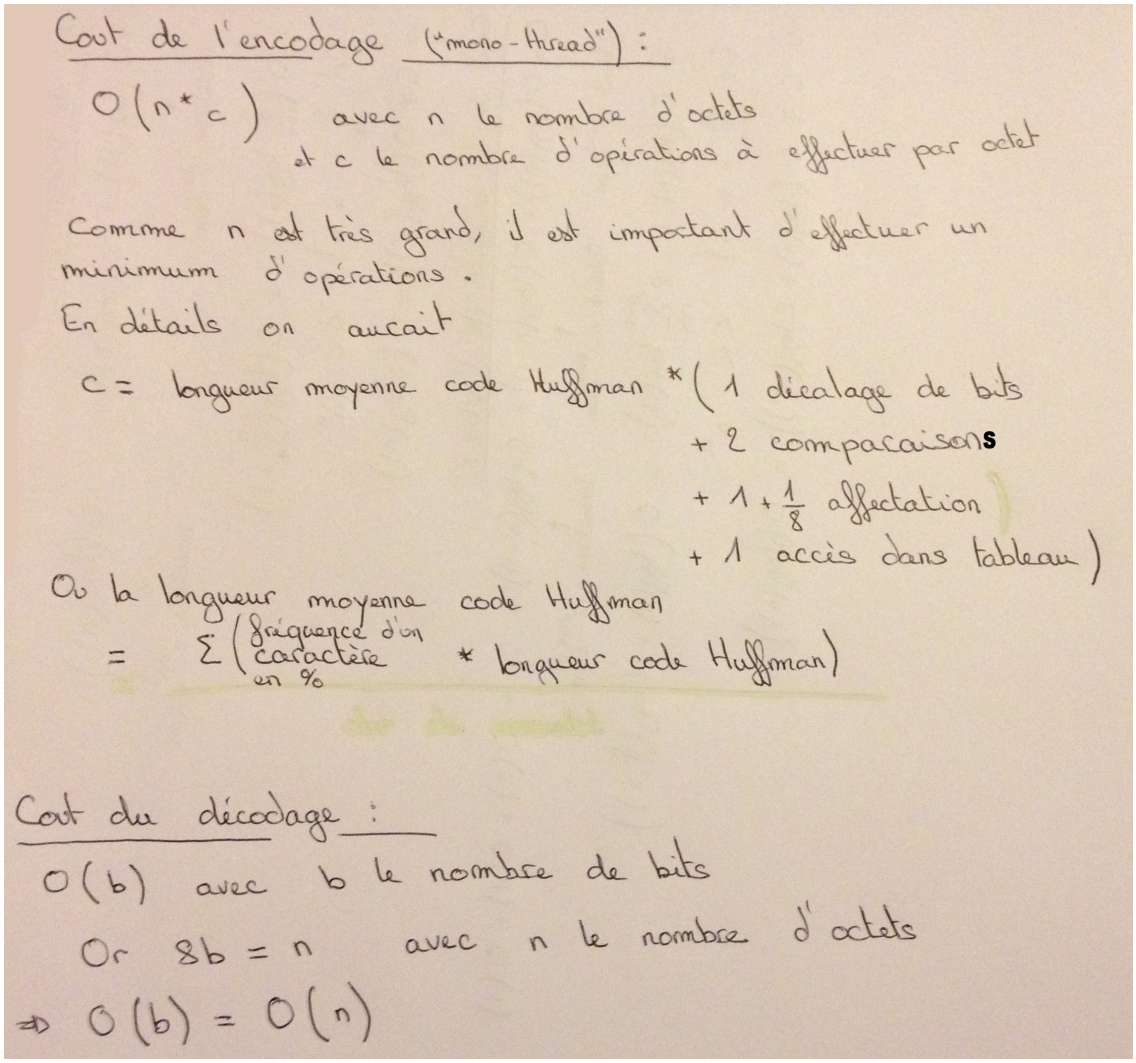
Plus de graphiques sont disponibles en annexes.

VI. Annexes

a. Détails des calculs de couts de l'insertion de la création de l'arbre



b. Détails des calculs de couts compression et décompression



Cout de l'encodage ("mono-thread") :

$O(n * c)$ avec n le nombre d'octets
et c le nombre d'opérations à effectuer par octet

Comme n est très grand, il est important d'effectuer un minimum d'opérations.
En détails on aurait

c = longueur moyenne code Huffman * (1 décalage de bits
+ 2 comparaisons
+ 1 + 1/8 affectation
+ 1 accès dans tableau)

Or la longueur moyenne code Huffman
= Σ (fréquence d'un caractère en % * longueur code Huffman)

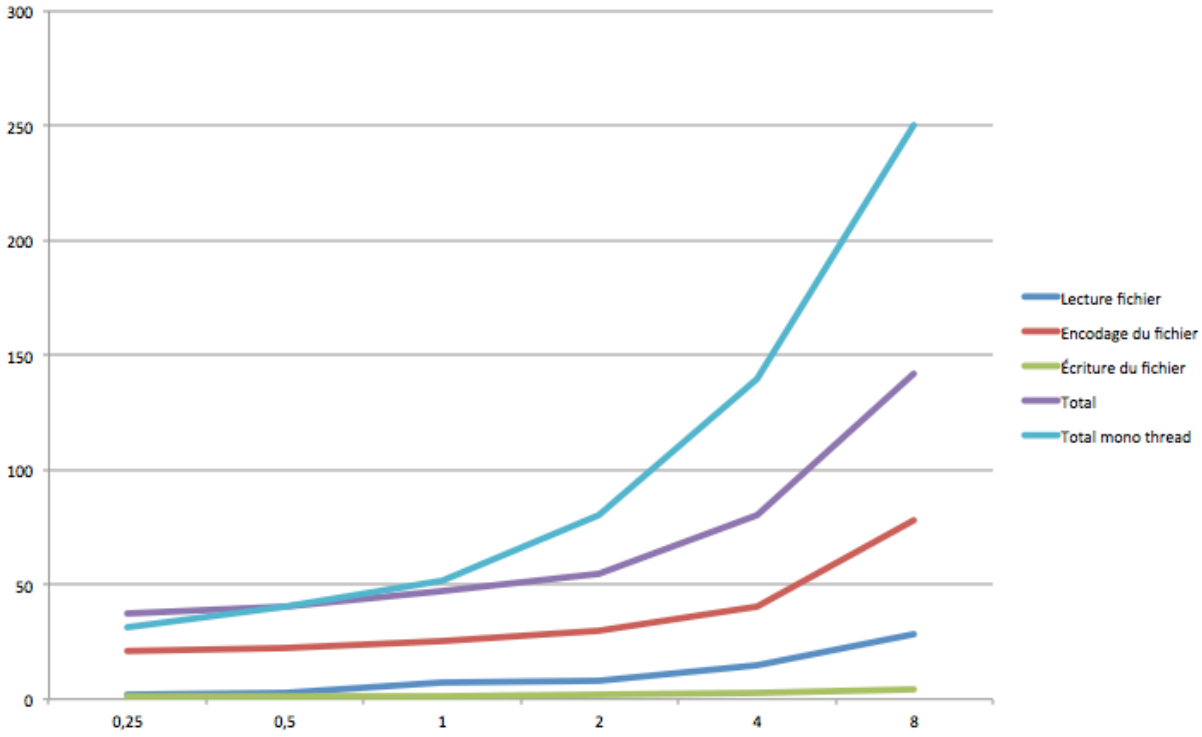
Cout du décodage :

$O(b)$ avec b le nombre de bits

Or $8b = n$ avec n le nombre d'octets

⇒ $O(b) = O(n)$

c. Temps de compression sur petits fichiers



Graphique de performance du programme. Taille des fichiers en Mo en abscisse, temps d'exécution en ms en ordonnée