▫univangers.jpg

Université d'Angers

PROJET TUTEURÉ ALGORITHME D'HUFFMAN

CHERRUAU Anthony
POUPELIN Bastien
THEBAUDIN Corentin
TUTEUR :
M.LARDEUX

16 novembre 2016

Table des matières

1 Introduction

David Albert Huffman

Qui est Huffman ?

Huffman est un professeur né le 9 août 1925 et mort le 7 octobre 1999. Il fut un pionnier dans le domaine informatique : Durant toute sa vie, Huffman apporta des contributions importantes à l'étude des machines à états finis. Mais Huffman est principalement connu pour l'invention du codage de portant son nom, utilisé dans presque toutes les applications qui impliquent la compression et la transmission de données numériques comme les fax, les modems, les réseaux informatiques et la télévision à haute définition. Huffman contribua beaucoup au développement de différents domaines, notamment dans la théorie de l'information et du codage, où il a été un pionnier dont les découvertes sont à la base des systèmes de compression de fichiers informatiques dans toutes les machines de nos jours.

Principe du codage de huffman

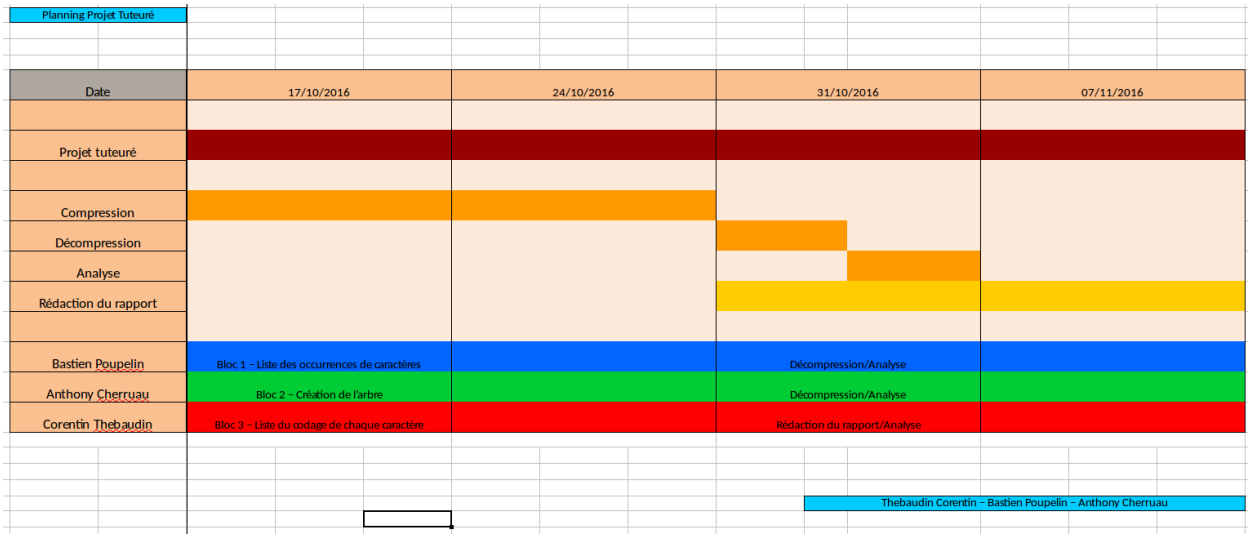
Le principe du codage de huffman repose sur la création d'un arbre composé de nœuds. Dans une phrase à coder, on compte d'abord le nombre d'occurrence de chaque lettre. Chaque caractère constitue donc une feuille de l'arbre auquel on y associe sa valeur d'occurrence. A la suite de cela, l'arbre contenant les caractères du fichier à compresser est créé. On associe les deux nœuds ou feuilles avec les occurrences les plus faibles pour créer un nouveau nœud dont son occurrence et la somme des deux occurrences des feuilles. On fait cela jusqu'à qu'il en reste un qui devient la racine. Ensuite, on associe le code 0 pour la branche gauche et le code 1 pour la branche droite. Ainsi, pour obtenir le code binaire de chaque caractère, on parcours l'arbre à partir de la racine jusqu'au feuilles en rajoutant un 0 ou un 1 en fonction de la branche suivie. Ainsi, chaque caractère du fichier à coder sera remplacé par son codage correspondant lu dans l'arbre. Le fichier sera ainsi compressé, et le système de décompression suit la même logique mais dans le sens inverse : le fichier compressé sera lu et dans l'en-tête du fichier sera inscrit la table de codage du fichier. Dès qu'une suite de codage binaire correspond à un caractère, il est remplacé. On retrouve ainsi le texte originel sans perte.

Déroulement du projet

Pour réaliser le projet, nous nous sommes réparti le travail en 4 parties :

- La fréquence des caractères du document source réalisée par Bastien
- La création de l'arbre binaire réalisée par Anthony
- Réalisation d'un dictionnaire listant le codage de chaque caractère réalisé par Corentin
- Mise en place de la compression et la décompression

Chaque partie réalisée individuellement dépendant des précedentes, elles ne peuvent pas être testées intégralement. Il a donc fallu effectuer un travail de regroupement des blocs.



Planning du projet

2 Compter la fréquence des caractères

Le principe de l'algorithme est de remplacer les lettres du document par des codes. Afin de gagner de la place, plus la lettre apparaît de fois et plus le code lui correspondant doit être court. Avant d'attribuer le code il est donc nécessaire de calculer la fréquence des caractères dans le document.

2.1 Structure de liste chainée

Pour compter les caractères j'ai utilisé une structure de liste chaînée contenant le caractère et l'occurrence de son apparition dans le document. Pour implémenter cette liste, on parcours les caractères du fichier un à un pour les ajouter dans la liste.

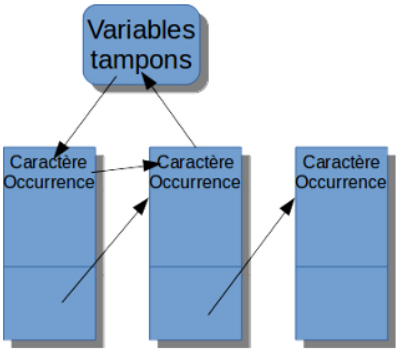
2.2 Problème des caractères de l'ASCII étendu

Malheureusement, un caractère UTF-8 ou ISO 8859-1 est composé de plusieurs octets pour pouvoir utilisé l'ASCII étendu (>127). Mais le langage C ne gère pas nativement ce codage car dans ce langage, 1 caractère = 1 octet. Ainsi seul les caractères de l'ASCII simples étaient correctement comptés donc les caractères accentués de la langue française, par exemple, ne l'étaient pas. Après une recherche de solutions improductives, nous avons en fin de projet, trouvé une solution. Avec la librairie WCHAR, on définit tous les caractères en type wchar_t et toute les fonctions telle que fprint sont transformé en wfprintf. Le problème des caractères de l'ASCII étendu sont donc désormais correctement dénombrés.

2.3 Insertion et tri dans la liste chainée

Pour revenir à la liste chaînée, elle est au début vide mais on ajoute à chaque fois le caractère lu à la première position s'il n'existe pas déjà dans la liste. Sinon, si il est déjà présent, on incrémente l'occurrence correspondant à ce caractère. Afin d'avoir un algorithme performant la liste doit être trié de l'élément le moins présent à celui le plus présent. Ainsi lors de l'incrémentation, il faut parfois trier la liste chaînée.
On a réalisé le tri non pas en échangeant les pointeurs de 2 maillons mais simplement en inversant les données (caractère, occurrence) entre maillons jusqu'à être à la bonne place. C'est à dire que l'occurrence contenue dans la

liste à droite de celle qu'on déplace doit être supérieur ou égal à l'occurrence de cette dernière.



Principe comptage occurrence

3 Arbre

3.1 Principe

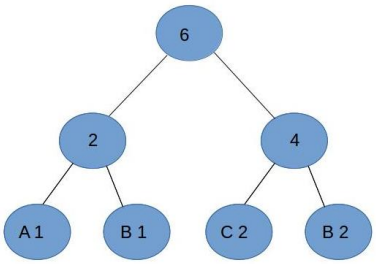
La création de l'arbre se fait en parcourant la liste triée. Elle prend les deux premières valeurs de la liste on va créer les feuilles puis on va créer un nœud et donc un arbre. Le nœud est créé en prenant les deux feuilles qui sont le fils gauche et le fils droit du nœud. Le nœud a pour valeur la somme des fréquences de ses fils. Ensuite on va réitérer cela a toute les valeurs de la liste.

3.2 Structure pour l'arbre

L'arbre contient la fréquence des lettres compté précédemment et les lettres qui vont nous permettre de créer les feuilles et les nœuds. Enfin on a une structure fils gauche et fils droit pour les noeuds ou le fils gauche et le fils droit vont pointer vers un nœud ou une feuille.

3.3 Création de l'arbre

La création de l'arbre se fait en parcourant la liste triée. Si l'élément suivant de la liste est différent de nulle alors on regarde si l'arbre est nulle ou non. Si c'est le cas alors on va créer le premier nœud avec les deux premières valeurs de la liste triée. Si ce n'est pas le cas alors on va regarder les fréquences dans la liste. Si la valeur de la fréquence est identique alors on va créer un nœud et on le met dans un autre arbre. Sinon on crée un nœud avec le premier nœud puis on assemble les deux arbres. A la fin il reste une valeur et donc on va créer la racine de l'arbre.



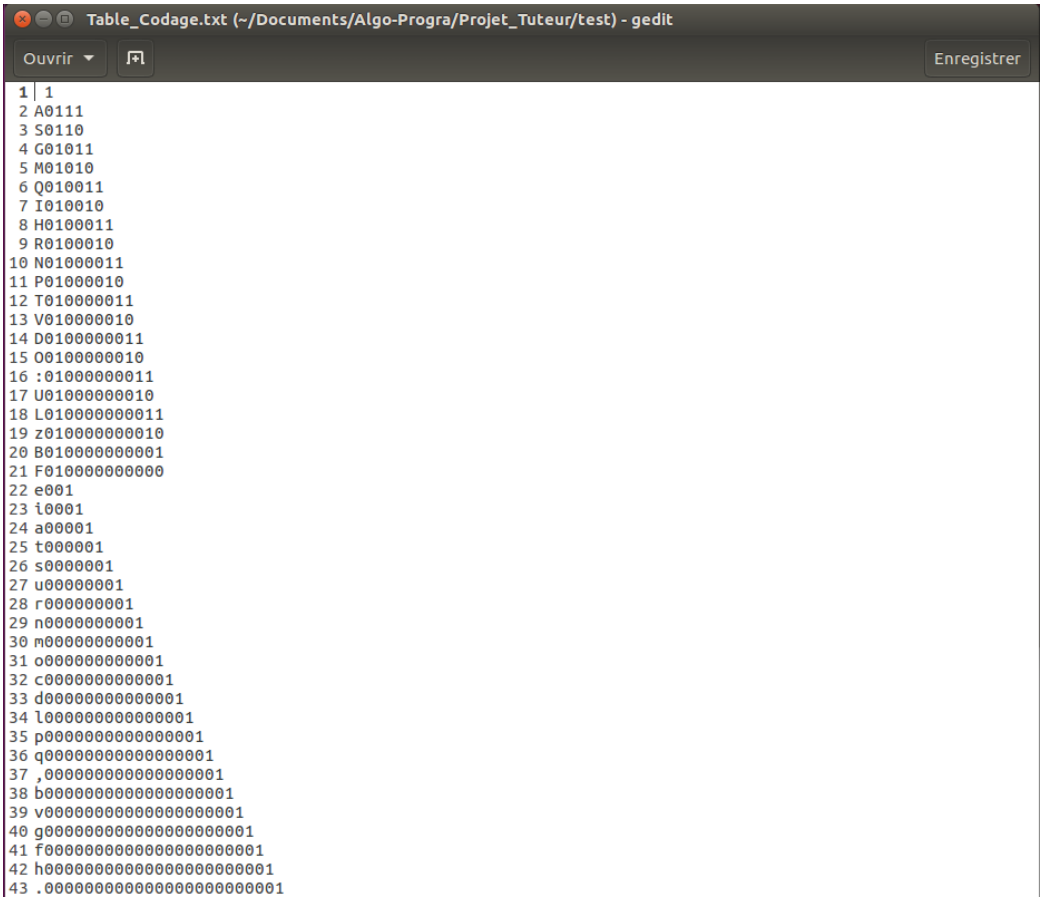
Exemple d'arbre obtenu

3.4 Réalisation d'un dictionnaire listant le codage de chaque caractère

Ce bloc permettra de coder chaque caractère du fichier à compresser en binaire. C'est cette opération que l'on appellera compression du document.

Pour réaliser ce bloc, il faudra utiliser l'arbre créé auparavant dans le bloc 2. Grâce à cet arbre, on pourra coder chaque caractère en binaire suivant le chemin emprunté dans l'arbre. Par exemple ici, on établira que pour chaque chemin vers la gauche emprunté, un "0" sera codé pour le caractère trouvé. Lorsque le chemin emprunté est celui du fils droit, le codage sera un "1".

C'est pourquoi la réalisation de ce bloc se basera sur une boucle de lecture de l'arbre. En premier temps la réalisation de cette lecture serait faite de manière préfixe, avec une variable binaire qui contiendrait le codage du caractère, s'incrémentant par un décalage de bit à chaque mouvement dans l'arbre jusqu'à attendre une feuille. Un "0" sera donc ajouté par un décalage à droite ("»"), et un "1" sera ajouté par concaténation avec la fonction strcat. Par soucis de temps et de solutions, un second choix a été réalisé : L'arbre sera lu de haut en bas jusqu'à atteindre une feuille, le codage et le caractère sera écrit dans le fichier puis la feuille sera supprimée. La lecture sera relancée, jusqu'à ce que l'arbre soit vide.



Exemple de dictionnaire obtenu

4 Compression

4.1 Principe

Après avoir réalisé ces 3 blocs, nous avons à disposition tous les outils pour réaliser la compression et la décompression d'un fichier. La compression consiste à lire le fichier à compresser et remplacer chaque caractère par son codage listé dans le dictionnaire. On parcourt le fichier source et on recupère le premier caractère. On compare ensuite le caractère récupéré avec notre table codage. Une fois le caractère retrouvé dans notre table on va copier le code correspondant à ce caractère dans notre fichier codé.

4.2 Mise en place

Malheureusement la mise en place de la compression n'est pas implémenté. La gestion des bits est quelque chose que nous n'avons pas réussi a comprendre. Malgré nos différents essaie nous avons fait un programme qui compresse 5% du fichier ce qui ne correspond pas a l'algorithme d'huffman qui est plutôt de l'odre de 45%

5 Décompression

5.1 Principe

La décompression va lire le fichier compressé qui contient un en-tête et le fichier texte compressé. L'en-tête contient l'arbre crée précédemment. Pour décompresser le fichier on lit le fichier qui contient des 1 et des 0 puis on parcours l'arbres. Dès que l'on arrive a une feuille dans l'arbre alors on met la lettre trouvé dans le fichier décompressé et on continue ainsi jusqu'à la fin du fichier.

La décompression peut se faire de deux manières : l'en-tête peut contenir le dictionnaire et donc on lira le fichier compressé et on retrouvera le caractère correspondant au codage. Cette méthode est plus simple à réaliser pour l'en-tête, mais un peu plus complexe pour la décompression. La seconde méthode consiste à intégrer l'arbre dans l'en-tête, et on retrouvera le caractère en suivant le chemin suivi avec le codage du fichier compressé. Cette méthode est plus complexe pour intégrer l'arbre à l'en-tête, mais plus simple à la suite pour décompresser le fichier.

6 Comparaison de différentes manières de compression

Lorsque l'on compare différents outils de compression, Huffman est intéressant sur de gros fichiers en majorité. Mais lors de la compression d'un petit fichier, le fichier original est plus petit que celui compressé. Il n'est donc pas avantageux dans ce cas-ci. Voici d'aileurs un tableau récapitulatif de différentes compressions en comparaison de l'Algorithme d'Huffman.

Type de compression	taille du fichier	taille fichier compressé
Huffman	8,5ko	4,8ko
gzip	8,5ko	4ko
tar	8,5ko	10,2ko
bzip2	8,5ko	3,8ko

En comparant ces différents algorithmes, on remarque que l'Algorithme d'Huffman a un bon taux de compression avec un taux de 43%. Bzip2 reste celui avec le meilleur taux de compression de 55%.
On voit donc que l'algorithme d'Huffman seul est assez efficace comparé aux autres format de compression. De plus, il a l'atout d'assurer aucune perte de données.

7 Améliorations possibles

Notre code n'étant pas fini, de nombreux point peut être améliorés car des méthodes ont été mises en place pour palier au manque de temps.

Compression et Décompression Nous avons eu assez de mal a comprendre la mise en place des bits en c et c'est pour cela que la compression ne fonctionne pas. Nous avions choisi d'abandonner l'en-tête du fichier de compression et donc lire directement dans la table de codage. Il serait donc possible, moyennant le fonctionnement de notre méthode, ajouter l'en-tête du fichier de compression contenant le dictionnaire dans un premier temps, et l'arbe dans un second temps.

Amélioration de l'arbre dans des cas particuliers : Lorsqu'un fichier vide ou un fichier contenant deux caractères est traité, l'arbre ne peut être créé. Certe ce sont des cas assez particuliers et peu probables, mais cela reste un bug à traiter.

Encodage : Au lieu d'utiliser une liste qui contient le codage, on peut mettre cette table dans un fichier et pour faire la compression on lit ce fichier et le fichier source. Cette solution à été choisie par manque de temps, car la solution idéale n'a pu être mise en place à cause d'un problème d'écriture dans le fichier.

8 Apport personnel

Ce projet nous a permis de nous améliorer dans le langage C, en particulier de voir comment gérer les caractères accentués. Ils nous a aussi permis de revoir les listes , les arbres, la lecture et l'écriture de fichier. Ces outils n'étaient pas clairs pour nous avant le projet, mais après avoir travaillé dessus, cette idée de liste et d'arbre nous paraît beaucoup plus claire. On a enfin pu voir la mise en place des bitwize en C, malgré que cela ne soit pas implémenté dans notre code par manque de temps et de réussite.