**MAJDOUL Kaoutar** **Groupe B**
**LAAFOU Ayoub**

# SYSTÈMES D'EXPLOITATION

## Contrôle Continu

**Algorithme de Huffman**

**Décembre 2018**

# Introduction

Le codage de Huffman est un processus qui permet de compresser des données informatiques afin de libérer de la place dans la mémoire d'un ordinateur. Un fichier informatique est formé d'une suite de caractère codé par une suite de 0 et 1, que cela soit un fichier texte, une image ou encore un son.

L'idée du codage de Huffman est de repérer les caractères les plus fréquents et de leur attribuer des codes courts (c'est-à-dire nécessitant moins de 0 et de 1) alors que les caractères les moins fréquents auront des codes longs. Pour déterminer le code de chaque caractère on utilise un arbre binaire. Cet arbre est également utilisé pour le décodage.

Dans le cadre de l'unité d'enseignement HLIN303 de L2 informatique, nous avons été amené à développer un programme permettant de compresser des fichiers sans perte, c'est-à-dire de réduire la taille qu'ils occupent tout en conservant l'intégralité des données originales.
Ce programme utilise l'algorithme de compression dit de Huffman, basé sur le codage mis au point en 1952 par David Albert Huffman et toujours utilisé aujourd'hui. Ainsi il permet la compression de n'importe quel fichier et sa décompression, en affichant éventuellement des informations détaillées sur la compression telles que le gain en terme de place gagné suite à la compression.
Dans ce rapport, nous décrirons les différents choix techniques faits lors
la conception de ce programme et les spécifications nécessaire à son exécution.
De plus, en fin de celui-ci, sont présentes les réponses aux questions posées dans le sujet.

# Conception

Au sein du dossier *Projet Huffman* nous retrouvons les fichiers *huf.c, dehuf.c* ainsi que le *hufREP.py*. Ces fichiers contiennent les programmes nécessaires à la compression, décompression de fichiers choisi par l'utilisateur ainsi que la compression d'un répertoire entier.
Nous allons détailler le code présent dans chaque fichier ainsi que son utilité au sein du programme.

***huf.c :***

Ce fichier permet la compression d'un fichier passé en paramètre dans un fichier destination.

Nous avons choisi d'implémenter les structures et fonctions nécessaire avant le ***main()*** qui lui se charge d'initialiser les structures ainsi que de lire les caractères et leurs nombre d'occurrences dans le fichier. Ensuite à partir des nombres d'occurrences on calcule la fréquence et on construit l'arbre. Une fois l'arbre construit on génère les codes en partant de la racine puis on affiche l'arbre ainsi que les codes.

On affiche la longueur moyenne de codage puis on débute la compression du fichier. Tout d'abord nous sauvegardons le *Header* la partie contenant les codes et nous renvoyons le nombre d'octets écrits. A ce nombre d'octets écrits on ajoute le nombre d'octets renvoyé suite à l'encodage du fichier. Pour finir on affiche la taille originelle, la taille compressé et le gain. En cas d'erreur on précise que la compression n'a pas pu avoir lieu.

L'en-tête du fichier, placé avant les données binaires brutes calculées avec l'algorithme de Huffman, contient des informations sur la longueur originale du fichier et sur l'arbre binaire utilisé pour coder le fichier.
Le stockage de la longueur originale se fait par un entier en 64 bits. Elle permet d'ignorer les bits de remplissage ajoutés à la fin du fichier compressé pour que le nombre de bits soit multiple d'un octet. Autrement, des caractères supplémentaires seraient insérés à la fin du fichier par le processus de décompression
Les buffers utilisés dans le code permettent d'extraire des données d'un fichier bit par bit ou de les écrire également bit par bit. Ils sont notamment utilisés dans les portions du code écrivant ou lisant un arbre vers ou depuis un fichier, ou celles de compression et de décompression.

***dehuf.c*** *:*

Ce fichier permet la compression d'un fichier passé en paramètre dans un fichier destination.

Au sein du ***main()*** on initialise un tableau de code de 256 codes maximum.

Ensuite on lit le fichier passé en paramètre et on incrémente le nombre total de caractères. Suite à cela on lit le header et on le décompresse en stockant le nombre d'octets écrit dans une variable.

***hufREP.py*** **:**

Ce fichier permet la compression de tous les fichiers au sein d'un répertoire.
Il prend un répertoire passé en paramètre et le parcours afin d'utiliser le ***./huf*** sur chacun des fichiers trouvés.

.

# Spécifications techniques

Le fichier *tester.sh* va permettre d'exécuter une suite de commande en shell les unes après les autres.

Tout d'abord, il va compiler le fichier huf.c :

```
gcc -o huf huf.c
```

Et par la suite, le fichier dehuf.c :

```
gcc -o dehuf dehuf.c
```

Enfin, le script va afficher à l'utilisateur la syntaxe d'utilisation de l'option de compression ou de décompression à exécuter :

```
echo "COMPRESSION :  ./huf fichiersource fichierdestination "


echo "DÉCOMPRESSION : ./dehuf fichiercompressé fichierdecompressé "
```

**1) Quel est le nombre maximum de caractères (char) différents ?**

Il y a 256 caractères différents possibles car il y a 256 caractères différents dans la table ASCII.

**2) Comment représenter l'arbre de Huffman ? Si l'arbre est implémenté avec des tableaux (fg, fd, parent), quels sont les indices des feuilles ? Quelle est la taille maximale de l'arbre (nombre de noeuds) ?**

L'arbre de Huffman est représenté selon le caractère son équivalent dans la table ascii dans un tableau.

| ***lettre*** | ***code*** |
|---|---|
| **A** | **0111** |
| **B** | **010** |
| **C** | **10** |
| **D** | **00** |
| **E** | **11** |
| **F** | **0110** |

Sachant que la racine de l'arbre est A[1], les indices des fg, fd et du parent sont :
FG[i] : 2i
FD[i] : 2i+1
Parent[i] : i/2

Si X est l'alphabet dont sont issus les caractères alors l'arbre représentant un codage préfixe optimal possède exactement X feuilles (une par lettre de l'alphabet) et X-1 nœuds.

**3) Comment les caractères présents sont-ils codés dans l'arbre ?**

Chaque feuille contient un caractère et sa fréquence d'apparition. Chaque nœud interne contient la somme des fréquences des feuilles de ses sous-arbres. Le code associé à un caractère = chemin de la racine à ce caractère avec 0 vers le fils à gauche et 1 vers le fils à droite.

**4) Le préfixe du fichier compressé doit-il nécessairement contenir l'arbre ou les codes des caractères ou bien les deux (critère d'efficacité) ?**

Un code préfixe est un ensemble de mots tel qu'aucun mot de l'ensemble n'est préfixe d'un autre mot de l'ensemble ce qui rend le décodage immédiat. Un code préfixe sur l'alphabet binaire peut être représenté par un trie qui est en fait un arbre binaire où tous les nœuds internes ont exactement deux successeurs. Les feuilles contiennent les caractères originaux, les branches par 0 ou 1 et les chemins depuis la racine jusqu'aux feuilles épellent les codes des caractères originaux donc l'utilisation d'un code préfixe assure que les codes sont bien représentés par les feuilles et facilite le décodage car il y a une absence d'ambiguïté. Il est donc nécessaire d'avoir l'arbre car aucun code n'est préfixe d'un autre code.

**5) Quelle est la taille minimale de ce préfixe (expliquer chaque champ et sa longueur) ?**

Un code de Huffman est un code préfixe à longueur variable. Il est optimal, au sens de la plus courte longueur, pour un codage par symbole. Dans la théorie des codes, le code à longueur variable est un code qui

associe les symboles de la source à un nombre variable de bits. Le code ascii étant codé sur 8 bit la taille minimale du préfixe doit être de 8 bits (2^8 pour les 256 caractères).

**6) Si le dernier caractère écrit ne finit pas sur une frontière d'octet, comment le compléter ? Comment ne pas prendre les bits de complétion pour des bits de données ?**

Il est essentiel de compléter le dernier bloc avec des bits complémentaires quand la taille de la donnée n'est pas un multiple de la taille d'un bloc. On appelle cela le bourrage. Cela consiste à compléter le dernier bloc par autant d'octets que nécessaire, chaque octet ayant pour valeur le nombre d'octets ajoutés. Cette façon de bourrer le dernier bloc permet de contrôler le bon déroulement du déchiffrement car le dernier bloc se termine par *n* octets identiques valant *n*.

**7) Le décompresseur doit-il reconstituer l'arbre ? Comment ?**
Codes + Texte codé → texte original

Il n'y a pas d'ambiguité dans un codage préfixe, puisqu'aucun code n'est le préfixe d'un autre. Le décompresseur contient l'arbre créé lors de la compression, à partir de là il analyse le texte codé en recherchant les codes (parcours de la racine vers les feuilles dans l'arbre de codage) afin de le décoder.