

CR apnée algo

Baptiste Cassou

Florian Sert-Vidal

November 2025

1 Introduction

Lors de cette apnée, nous avons recodé le codage de Huffman. l'apnée est divisée en 2 fichiers, `huff_encode.` et `huff_decode.c`. Le principe de l'algo est de construire un arbre dit de Huffman pour raccourcir le codage des caractères récurrents à partir d'une table d'occurrence des caractères.

2 `huff_encode.c`

- Dans ce programme, nous commençons tout d'abord par construire la table d'occurrence sans oublier d'initialiser tous les éléments à 0. La table peut reconnaître tous les caractères sur 8 bits, soit la table ASCII étendue. En cas de fichier vide, un fichier vide est donné en sortie
- Ensuite nous construisons la FAP composée d'arbres qui ne sont au final que des racines avec le nombre d'occurrences et sans enfants. Grâce à cette FAP nous construisons l'arbre en suivant l'algorithme (donc les éléments les moins récurrent en bas et les plus courants en haut).
- Pour finir, nous remplissons la table de Huffman avec le codage pour chaque caractère selon l'arbre précédemment construit. Pour cela nous parcourons en profondeur avec le codage de caractères suivant:
 - 0 Si on part à gauche
 - 1 Si on part à droite

Nous stockons également la profondeur qui sera la longueur du codage. Dans le cas où l'arbre est une racine, le codage est forcé à 0.

Il ne reste plus qu'à encoder en bit à bit (grâce au module `bfile`) le fichier encodé après avoir écrit l'arbre dans le fichier.

2.1 Arbre optimal ?

Cette version du codage de Huffman est optimale pour le texte sur lequel il est encodé car il se base sur la table d'occurrence du texte. On perd donc du temps pour compter les occurrences au début, mais cela assure l'optimalité du codage. De même, nous sommes obligés de stocker l'arbre dans le fichier car la fonction de décodage ne le connaît pas, cela nuit à l'optimalité de compression pour les fichiers courts.

3 huff_decode.c

Pour décoder un fichier encodé, nous lisons bit à bit le fichier après avoir lu l'arbre. En parcourant l'arbre en suivant le chemin indiqué par les bits (0 à gauche et 1 à droite) et dès qu'une feuille est atteinte, on affiche le caractère associé et on se replace à la racine. En cas de décodage d'un fichier vide, un fichier vide est donné en sortie.

Au cas où le fichier est composé d'un unique caractère, alors l'arbre sera uniquement une racine/feuille, or il y aura un 0 à lire dans le fichier. Donc pour pouvoir décoder, nous lisons l'étiquette du nœud sans descendre.

Nous sommes obligés de lire l'arbre car le décodeur n'a pas connaissance de l'arbre qui a été utilisé pour encoder le fichier.

4 TESTS

Pour effectuer les tests, nous avons créé un script bash pour tester le jeu de test et nous faire une synthèse des résultats.

4.1 jeu de test

Pour tester l'efficacité du programme nous avons testé plusieurs formats de fichiers textes et d'autres formats. Nous voulions tester des fichiers en plusieurs langues, des fichiers compressés (jpg) des images/vidéos et des textes avec des structures particulières (code C, autres fichiers binaire)

Notre protocole était de comparer la taille des fichiers compressés et décompressés et que le fichier décompressé soit égal au fichier de base. Pour tester la conservation des bits nous avons vérifié manuellement (vidéo qui se lit, exécutable qui s'exécute).

Voici la liste des types de fichiers que nous avons généré

- txt court
- texte long (candide et gargantua)
- des textes en plusieurs langues
- texte d'un seul caractère
- texte de plusieurs fois le même caractère
- texte composé d'une occurrence de chaque caractère
- un fichier en C
- un gloubi boulga de bits (généré aléatoirement)
- un fichier mp4
- une image
- une image compressé jpg
- un fichier video mp4
- un fichier vide

Table 1: Résultats de compression Huffman sur différents types de fichiers

Fichier	Taille originale (octets)	Taille compressée (octets)	Taux (%)	Gain (%)	Type
Fichiers très petits (≤ 100 octets)					
vide.txt	0	0	0,0	0,0	Vide
only1.txt	1	19	1900,0	-1800,0	1 caractère
Hallo.txt	10	190	1900,0	-1800,0	Texte court Allemand
hello.txt	11	190	1727,3	-1627,3	Texte court Anglais
2lettres.txt	11	44	400,0	-300,0	Texte court
bonjour.txt	13	240	1846,2	-1746,2	Texte court
only1plusieur.txt	47	24	51,1	48,9	Répétitif
Fichiers texte moyens (≤ 10 Ko)					
huff_decode.c	991	1918	193,5	-93,5	Code source
aleat	1086	1961	180,6	-80,6	Aléatoire
99luftballons.txt	1115	1926	172,7	-72,7	Texte
Nevergonnagiveuup.txt	1730	1741	100,6	-0,6	Texte
pete&repet.txt	18143	5235	28,9	71,2	Très répétitif
Fichiers texte longs (≤ 100 Ko)					
candide.txt	206142	119939	58,2	41,8	Roman français
candide_an.txt	213977	125257	58,5	41,5	Roman anglais
candide_al.txt	218420	127969	58,6	41,4	Roman allemand
gargantua.txt	273277	157459	57,6	42,4	Roman français
Fichiers binaires					
all.bytes	256	6395	2498,0	-2398,0	256 caractères
utBJivEMXIF0	5120	11241	219,6	-119,6	Binaire
huff_encode	33488	24460	73,0	27,0	Exécutable
rick.jpg	44237	50497	114,2	-14,2	Image JPEG
rick.png	221452	228400	103,1	-3,1	Image PNG
Colin_Walk_large.mp4	8983170	9024145	100,5	-0,5	Vidéo MP4

- Ce que l'on peut en dire est que les fichiers textes courts non répétitif ont une compression mauvaise à cause de l'arbre que l'on doit encoder dans le fichier.

- Par contre pour les textes plus longs et cohérents il y a une bonne compression. Cela est due à la répétition de caractère tel que le E en français, cela permet à l'algo d'exprimer de fonctionner de manière optimal.
- A contrario les binaires et autres fichiers sans structure répétitive ont une moins bonne compression car l'algo n'identifie pas d'octet répétitif ou une répartition uniforme des octets.
- Les chansons qui ont beaucoup de répétition de mots mais sans autres mots différents donc il y a une répartition plutôt uniforme donc cela entraîne une mauvaise compression.
- Également les fichiers compressés ont une moins bonne recompression avec Huffman car la précédente compression a déjà supprimé les redondances.

En conclusion on peut voir que le codage de Huffman est très efficace sur les textes en langage naturel (pas en C) par contre sur les autres formats sans répétition ou avec une répartition uniforme le codage obtient une compression négative.

Le codage préserve bien les bits, on peut par exemple réexécuter directement l'exécutable après compression/décompression.

Pour tout notre jeu de test, la compression et décompression prend moins de 1s sauf pour la vidéo de 9Mo qui prend 1.2 de compression et 1.6 de décompression. ce qui est logique car la complexité de compression est l'approximation de $O(n)$ et $N \times \log_2 N$ pour la décompression.