Projet - Algorithmique avancée et programmation C
(Dé)compresseur de fichiers

Nicolas Delestre

L'objectif de ce projet est de développer un compresseur et un décompresseur (sans perte d'information) qui utilise l'algorithme de Huffman pour compresser/décompresser un fichier. Ce programme s'utilisera de la manière suivante :

— pour compresser : `huffman c nomFichier`
— pour décompresser : `huffman d nomFichier.huff`

1 Bibliographie : Présentation de la méthode

On se propose dans cette partie de présenter le principe de la compression de Huffman. Afin d'être plus clair, nous allons voir comment coder un texte (composé de caractères). Le même raisonnement peut être appliqué pour compresser des données binaires (composées d'octets).

1.1 Principe

Le principe de cette méthode est de remplacer l'utilisation d'un code [1] à longueur fixe (par exemple 8 bits) par un code à longueur variable : un caractère souvent présent dans un document source sera alors codé à l'aide d'un code plus court qu'un caractère n'apparaissant que quelques fois [2].

Pour réaliser cet algorithme nous avons besoin d'un arbre de Huffman dont les feuilles sont les caractères présents dans le document source. Le chemin permettant d'atteindre ces feuilles est le code correspondant (avec ici comme protocole 0 pour l'accès à un sous-arbre gauche et 1 pour l'accès à un sous-arbre droit).

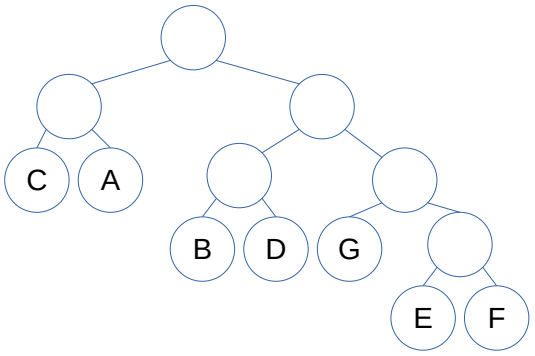
Par exemple si le texte source est "BACFGABDDACEACG", on obtient l'arbre binaire présenté par la figure 1a. Ce qui donne le code à longueur variable présenté par le tableau 1b.

1.2 Méthode pour construire l'arbre de Huffman

La méthode pour construire cet arbre est la suivante :

1. On commence par construire autant d'arbres qu'il y a de caractères dans le texte source en ajoutant la fréquence d'apparition. Dans notre exemple cela donne :

1. Un code est une suite de bits
2. C'est ce qui est appliqué dans le langage Morse



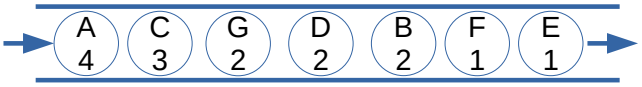
(a) Arbre de Huffman

Caractère	Code binaire
A	01_2
B	100_2
C	00_2
D	101_2
E	1110_2
F	1111_2
G	110_2

(b) Table de codage

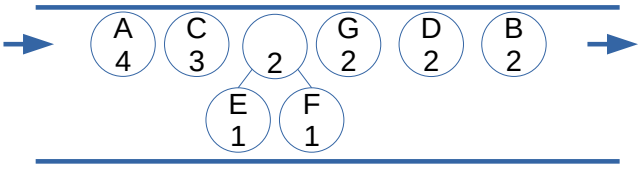
FIGURE 1 – Arbre de Huffman et table de codage correspondante

2. On construit alors une file de priorité d'arbres. On y insère ces arbres, dans l'ordre croissant de leur code ASCII (le 'A' sera inséré avant le 'B'), en considérant qu'un arbre a_1 est plus prioritaire qu'un autre arbre a_2, lorsque la pondération de la racine de a_1 est plus petite que celle de a_2. Dans notre exemple on obtient alors la file de priorité suivante :

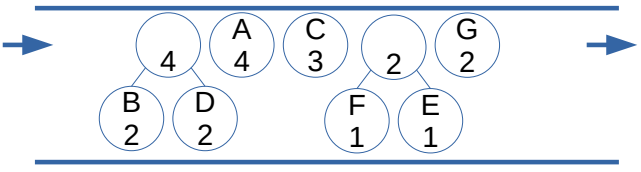


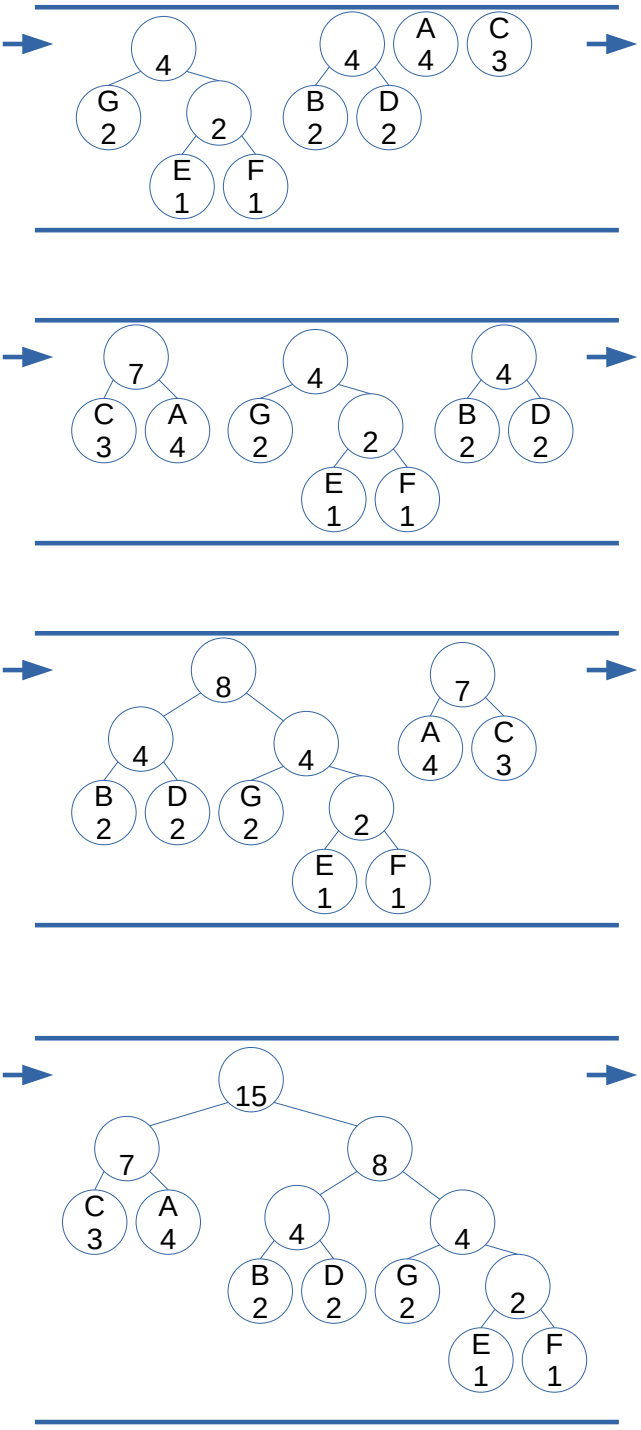
3. S'il y a au moins deux arbres dans la file de priorité :
   — on défile ces deux arbres
   — on en forme un nouveau dont :
      — la pondération de la racine sera la somme des pondérations des deux racines de ces fils,
      — le sous-arbre gauche est le premier arbre défilé,
      — le sous-arbre droit est le deuxième arbre défilé,
   — on insère ce nouvel arbre dans la file.

   Dans notre exemple on obtient alors la file de priorité suivante :



4. On réitère l'opération jusqu'à ce qu'il n'y ait plus qu'un seul arbre




4
G 2
2
E 1
F 1
4
B 2
D 2
A 4
C 3
7
C 3
A 4
4
G 2
2
E 1
F 1
4
B 2
D 2
8
4
B 2
D 2
4
G 2
2
E 1
F 1
7
A 4
C 3
15
7
C 3
A 4
8
4
B 2
D 2
4
G 2
2
E 1
F 1

1.3 Compression

Ainsi le texte “BACFGABDDACEACG” qui nécessitait par défaut 15 octets pour être représenté [3] nécessite plus que 5 octets [4] (séparés par des points) :

$$\underbrace{100}_{B}\ \underbrace{01}_{A}\ \underbrace{00}_{C}\ \underbrace{1.111}_{F}\ \underbrace{110}_{G}\ \underbrace{01}_{A}\ .\ \underbrace{100}_{B}\ \underbrace{101}_{D}\ \underbrace{10.1}_{D}\ \underbrace{01}_{A}\ \underbrace{00}_{C}\ \underbrace{111.0}_{E}\ \underbrace{01}_{A}\ \underbrace{00}_{C}\ \underbrace{110}_{G}$$

La phase de compression peut être décomposée en différentes étapes :

1. calcul des statistiques à partir du flux source (considéré comme étant composé d’octets),
2. construction de l’arbre de Huffman à partir des statistiques,
3. calcul du code binaire de chaque octet,
4. production des octets compréssés dans un flux destination à partir des données du flux source et des codes de chaque octet. Ces données seront constituées :
   — du code d’identification qui permettra lors d’une demande de décompression de vérifier le type du fichier,
   — des statistiques du fichier source,
   — de la longueur du fichier source,
   — des données compressées.

   Ce qui se synthétise par la figure 3.

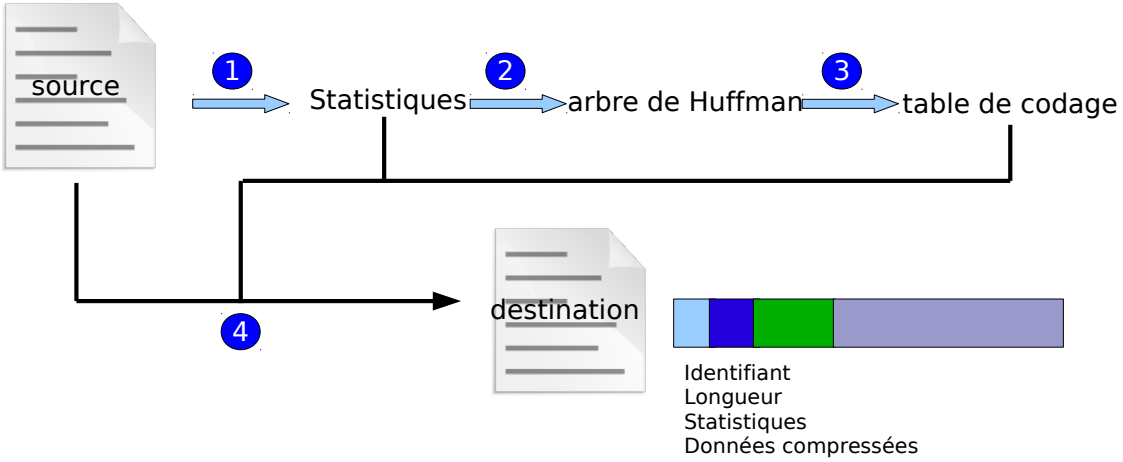


FIGURE 2 – Les phases de la compression

1.4 Décompression

Pour décompresser il suffit de se positionner au début du message compressé et au sommet de l’arbre de Huffman puis de lire le message bit par bit et se déplacer en conséquence dans l’arbre (si le bit est 0, aller dans le sous arbre gauche, ou dans le sous arbre droit sinon). Si le bit courant nous amène à une feuille de l’arbre, c’est que l’on vient de décoder le caractère correspondant. Il suffit alors de passer au bit suivant et de repartir depuis la racine de l’arbre.

3. Sans compter les octets pour délimiter la chaîne.
4. Le fait que le dernier code binaire remplisse entièrement le dernier octet est pure coïncidence.

1.5 Synthèse

1.5.1 Compression

La phase de compression peut être décomposée en différentes étapes :

1. calcul des statistiques à partir du fichier source (considéré comme étant composé d'octets),
2. construction de l'arbre de Huffman à partir des statistiques,
3. calcul du code de chaque octet,
4. création du fichier destination à partir du fichier source et des codes de chaque octet, ce fichier destination contiendra :
   — un code d'identification qui permettra lors d'une demande de décompression de vérifier le type du fichier,
   — les statistiques du fichier source,
   — la longueur du fichier source,
   — les données compressées.

Ce qui se synthétise par la figure 3.

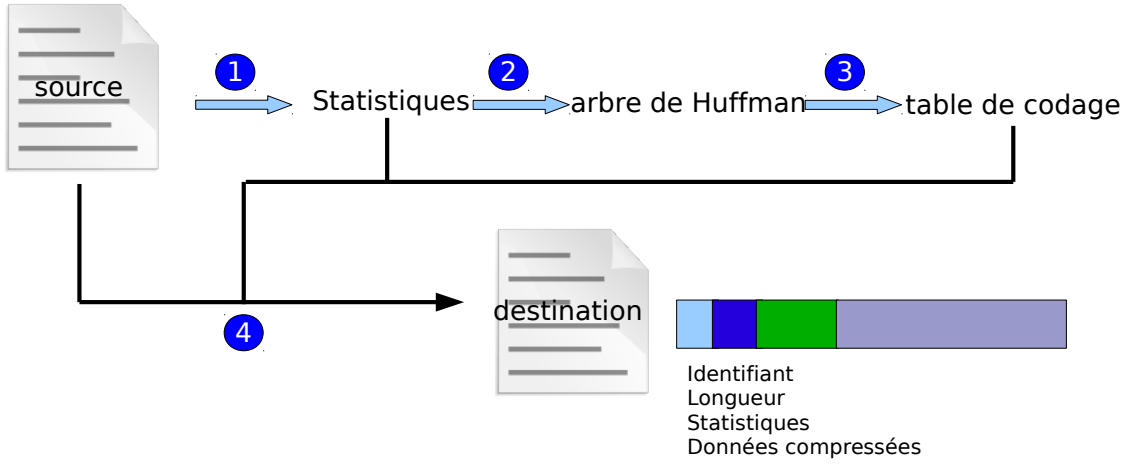


FIGURE 3 – Les phases de la compression

1.5.2 Décompression

De la même manière, la phase de décompression peut être décomposée de la manière suivante :
— vérification du type de fichier à décompresser,
— obtention de la longueur du fichier originel,

1. obtention des statistiques du fichier originel,
2. calcul de l'arbre de Huffman à partir des statistiques,
3. création du fichier originel.

Ce qui se synthétise par la figure 4.

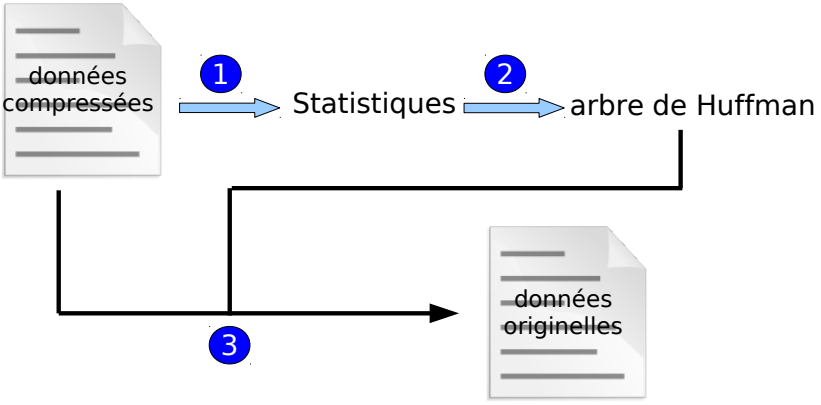


FIGURE 4 – Les phases de la compression

2 Préconisations

2.1 Analyse

On considère posséder les types et TAD suivants :
Type Bit **=** {bitA0,bitA1}
Type Mode **=** {lecture,ecriture}

Nom:	FichierBinaire	
Utilise:	**Chaine de caracteres**, Mode, Octet, **Caractere**	
Opérations:	fichierBinaire:	**Chaine de caracteres** $\rightarrow$ FichierBinaire
	ourvir:	FichierBinaire $\times$ Mode $\nrightarrow$ FichierBinaire
	fermer:	FichierBinaire $\nrightarrow$ FichierBinaire
	estOuvert:	FichierBinaire $\nrightarrow$ **Booleen**
	mode:	FichierBinaire $\nrightarrow$ Mode
	finFichier:	FichierBinaire $\nrightarrow$ **Booleen**
	ecrireOctet:	FichierBinaire $\times$ Octet $\nrightarrow$ FichierBinaire
	lireOctet:	FichierBinaire $\nrightarrow$ FichierBinaire $\times$ Octet
	ecrireNaturel:	FichierBinaire $\times$ Naturel $\nrightarrow$ FichierBinaire
	lireNaturel:	FichierBinaire $\nrightarrow$ FichierBinaire $\times$ Naturel
	ecrireCaractere:	FichierBinaire $\times$ **Caractere** $\nrightarrow$ FichierBinaire
	lireCaractere:	FichierBinaire $\nrightarrow$ FichierBinaire $\times$ **Caractere**
Sémantiques:	fichierBinaire:	creation d'un fichier binaire à partir d'un fichier identifié par son nom
	ouvrir:	ouvre un fichier binaire en lecteur ou ecriture. Si le mode est ecriture et que le fichier existe, alors ce dernier est écrasé
	fermer:	fermer un fichier binaire
	lireXX:	lit un XX à la position courante du fichier
	ecrireXX:	ecrit un XX à la position courante du fichier

Préconditions: ouvrir(f): non estOuvert(f)
fermer(f): estOuvert(f)
finFichier(f): mode(f)=lecture
lireXX(f): estOuvert(f) et mode(f)=lecture et non finFichier(f)
ecrireXX(f): estOuvert(f) et mode(f)=ecriture

L'analyse fonctionnelle précédente nous permet d'identifier les TAD suivants :

— `CodeBinaire`
— `Octet`
— `Statistiques`
— `ArbreDeHuffman`
— `FileDePriorité`
— `TableDeCodage`

1. Présentez ces TAD à l'aide du formalisme vu en cours.
2. Proposez une analyse descendante du problème de la compression et de la décompression d'un fichier.

2.2 Conception

2.2.1 Conception préliminaire

1. Donnez les signatures des fonctions et procédures permettant de manipuler les TAD précédents.
2. Donnez les signatures des fonctions et procédures issues de votre analyse descendante (fonctions et procédures métiers).

2.2.2 Conception détaillée

1. Proposez une implantation des TAD précédents ainsi que le corps de leurs fonctions et procédures les plus complexes.
2. Explicitez le corps des fonctions et procédures métiers.

2.3 Développement, tests unitaires et documentation

Développez le programme en C en testant au maximum vos fonctions à l'aide de l'API CUnit (la personne qui codera les tests unitaires ne doit pas être la personne qui implante les fonctions C). Documentez tout votre code à l'aide du logiciel *Doxygen*.