

# Projet informatique

## Compression de données – Codes de Huffman

IN 261 – ENSTA

<http://www.di.ens.fr/~pointche/enseignement/ensta2/>

Compte-rendu à transmettre au plus tard le **21 octobre 2005 à 12h00**.

### Abstract

Quand il s'agit de transmettre de l'information sur un canal non bruité, l'objectif prioritaire est de minimiser la taille de la représentation de l'information : c'est le problème de la compression de données (lors d'un canal bruité, on ajoute alors de la redondance – code correcteur d'erreurs – pour pallier les erreurs de transmission). Le code de Huffman (1952) est un code de substitution de longueur variable optimal: la longueur d'un texte codé est minimale. On observe ainsi des réductions de taille de l'ordre de 20 à 90%. Ce code est largement utilisé, souvent combiné avec d'autres méthodes de compression.

Il s'agit d'une méthode classique de construction de code optimal (table de conversion/substitution de chaque caractère en une chaîne binaire) qui utilise habituellement des arbres binaires. Nous allons l'implémenter avec seulement des tableaux.

## 1 Modalités de ce projet

Ce projet peut être effectué par binôme (2 personnes). Il sera évalué sur la base d'un compte-rendu envoyé par e-mail (à [David.Pointcheval@ens.fr](mailto:David.Pointcheval@ens.fr)) composé

- des sources (veiller à la clarté des sources, et ne pas hésiter à commenter chaque étape) – à titre d'information, hors fonction `main`, le programme fait moins de **150 lignes** ;
- d'un `Makefile` qui effectue la **compilation** de l'exécutable `huffman` —par l'appel de `make`—, ainsi que la série de tests présentée en fin de projet —par l'appel de `make test`— (voir la section 10.1 page 76 du polycopié, au sujet de l'utilitaire `make`, que vous avez intérêt à utiliser dès le début de vos travaux) ;
- d'un rapport, d'au plus 2 pages, au format électronique POSTSCRIPT (.ps) ou PDF Acrobat (.pdf) —tout autre format étant exclu— détaillant vos réflexions et les problèmes rencontrés, ainsi que ce que fait effectivement votre programme (fonctionnalités disponibles, parfaitement opérationnel, certaines restrictions, ...).

Ce compte-rendu doit être envoyé au plus tard le **21 octobre 2005 à 12h00**.

Consulter la page <http://www.di.ens.fr/~pointche/enseignement/ensta2/projet> pour toute information supplémentaire. Une liste de questions/réponses y sera également proposée.

## 2 Codes de Huffman

L'algorithme de Huffman utilise une table d'occurrences (ou fréquences = nombre d'apparitions) de chaque caractère pour construire un représentant de chaque caractère sous forme de chaîne binaire, ou liste de bits.

## 2.1 Idée de la compression

Considérons un fichier de 100 000 caractères que l'on souhaite stocker de façon compacte. On suppose que le nombre d'occurrences de chaque caractère est fourni sur la figure 1 : seuls six caractères apparaissent. Il y a alors plusieurs méthodes pour représenter ces caractères par des

Caractères	a	b	c	d	e	f
Nbre d'occurrences (en milliers)	45	13	12	16	9	5
Mot de code (binaire)						
Taille fixe	000	001	010	011	100	101
Taille variable	0	101	100	111	1101	1100

Figure 1: Table d'occurrences (fréquences)

chaînes binaires :

- code de taille fixe : tous les caractères sont codés sur le même nombre de bits. Pour représenter 6 caractères, il faut utiliser 3 bits. Notre fichier sera codé sur 300 000 bits.
- code de taille variable : les caractères sont représentés par des chaînes de longueur variable selon le nombre d'occurrences. Le 'a', très fréquent, est codé sur 1 bit, tandis que le 'e' et le 'f', moins fréquents, sont codés sur 4 bits. Notre fichier sera alors codé sur

$$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224000 \text{ bits.}$$

Soit un gain de 25%.

## 2.2 Codes sans préfixes

Un code est la table de correspondance (table de codage) entre chaque caractère et sa représentation binaire. Le codage d'un texte contenant plusieurs caractères consiste en la concaténation des représentants de chaque caractère.

**Exemple :** Le mot "abc" est représenté par "000.001.010" = "000001010" avec le code de longueur fixe et par "0.101.100" = "0101100" avec le code de longueur variable (voir figure 1).

En revanche, le décodage qui consiste à retrouver le texte initial à partir des représentants n'est pas toujours aisé. Comme on peut le remarquer, le codage à longueur variable satisfait la propriété dite "sans préfixe", c'est-à-dire qu'aucun mot de code n'est préfixe d'un autre mot. Cette propriété rend le décodage très simple : il suffit de repérer un mot de code en tête, le retirer et répéter l'opération :

$$\text{"001011101"} \rightarrow \text{"0.01011101"} \rightarrow \text{"0.0.1011101"} \rightarrow \text{"0.0.101.1101"} \rightarrow \text{"aabe"}$$

Pour un code quelconque (qui n'est pas "sans préfixes"), cette technique de décodage ne fonctionne pas :

Caractères	a	b	c	d	e	f
Mot de code (binaire)	0	01	011	0111	01111	011111

Ce code n'est pas sans préfixe puisque, par exemple, '0', code de 'a', est préfixe de tous les autres mots. Le décodage sera alors un peu plus complexe. Cependant,

**Proposition.** Une compression optimale obtenue par un code quelconque peut aussi être obtenue par un code préfixe.

Il n'y a donc aucune perte de généralité à se limiter aux codes sans préfixes. Ce que nous ferons dans la suite.

## 2.3 Arbres binaires

Le processus de décodage requiert une représentation adéquate du code préfixe, de telle manière que le mot de code de tête soit aisément isolé. Un arbre binaire, dont les feuilles sont les caractères, fournit une telle représentation (voir figure 2).

**Définition** (*arbre binaire*). Un *arbre binaire* est ou bien l'arbre vide ou bien un arbre dont chaque noeud contient exactement deux fils.  
Nous appellerons *feuille* un arbre binaire dont les deux fils sont l'arbre vide.

Par convention, on utilisera :

0 = bifurquer vers le haut = fils gauche

1 = bifurquer vers le bas = fils droit

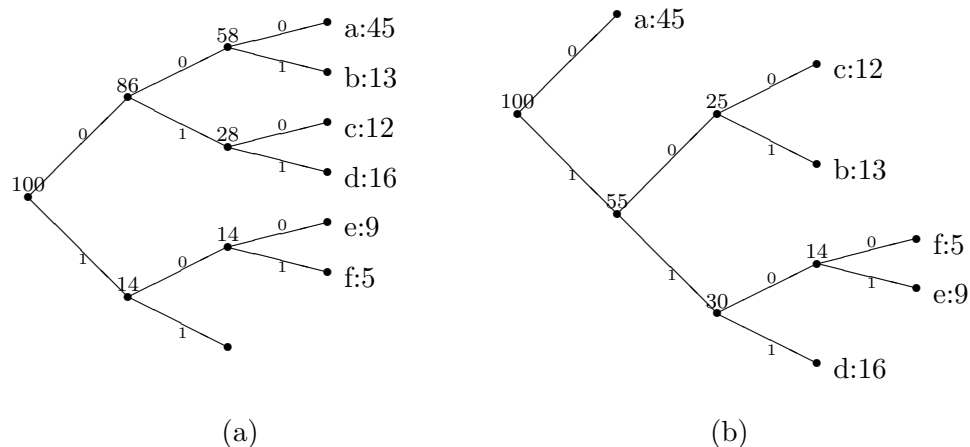


Figure 2: Arbres correspondant aux codages présentés figure 1

On interprète alors les codes comme des chemins dans l'arbre, de la racine au caractère à coder, où '0' signifie 'bifurquer vers le haut' (ou fils gauche) et '1' signifie 'bifurquer vers le bas' (ou fils droit) : le chemin à parcourir dans l'arbre (b) pour parvenir à la feuille contenant le caractère 'c' est 'droite' – 'gauche' – 'gauche'. Le mot de code représentant 'c' sera alors '100'.

**Proposition.** Un code optimal pour un fichier est toujours représenté par un arbre binaire *complet*, c'est-à-dire dont tous les noeuds ont soit deux fils différents de l'arbre vide, soit deux fils vides.

Ainsi, le code de longueur fixe, représenté figure 2 (a), n'est pas optimal puisque l'arbre n'est pas *complet* : il y a des mots commençant par '10...', mais pas par '11...'.

## 3 Construction du Code de Huffman

Huffman a proposé un algorithme pour construire un code préfixe optimal, appelé " Code de Huffman ". L'algorithme construit l'arbre de codage  $T$ , correspondant au code optimal, des feuilles vers la racine. Mais dans notre projet, dans la programmation, par soucis de simplicité, on se passera des arbres (qui seront vus en algorithmique), et on se contentera de tableaux : seule l'efficacité du décodage va en pâtir un peu, mais de façon peu sensible.

### 3.1 Description

La construction du code s'effectue de la façon suivante (voir la figure 3) :

1. On part d'une forêt d'arbres (forêt = plusieurs arbres), restreints chacun à une feuille, pour chaque caractère possible ;
2. On trie ces arbres dans l'ordre croissant des fréquences ;

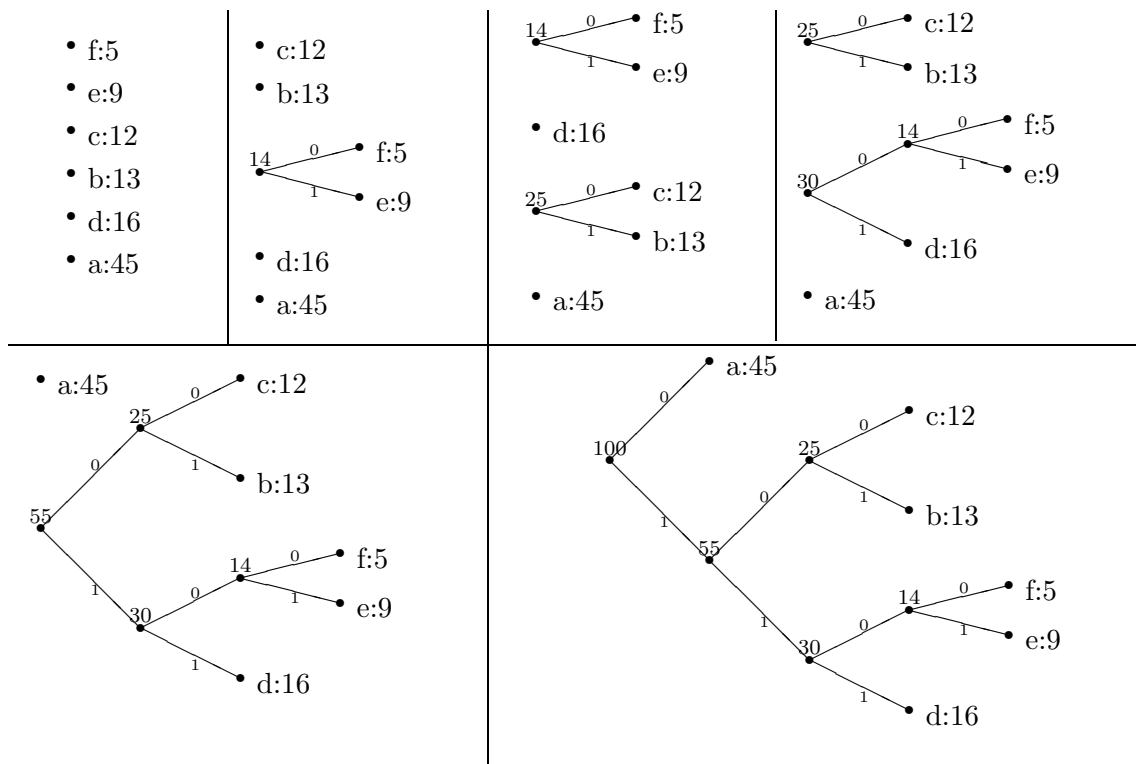


Figure 3: Fonction Huffman avec des arbres

3. On extrait les deux premiers arbres (correspondants aux caractères les moins fréquents) et on les regroupe, en additionnant les fréquences ;
4. Tant que la forêt contient plusieurs arbres, on retourne à l'étape 2.

**Proposition.** La fonction HUFFMAN fournit un code préfixe optimal.

### 3.2 Quelques types utiles

Puisque nous allons nous passer d'arbres, il nous faut trouver une structure adaptée à la création et au stockage de la table de codage.

Notre alphabet sera l'ensemble des caractères, soit 256 valeurs possibles. La table de codage sera en fait un tableau de 256 codes : un code pour chaque caractère. Par construction, un code sera une chaîne de caractères de '0' et de '1', de longueur bornée par 256. Ainsi on peut définir le type

```
typedef char code[257];
```

La table de codage sera un simple tableau de 256 codes, initialisés à des chaînes vides :

```
code codage[256];
```

La forêt pondérée par les fréquences pourra être représentée par un tableau de fréquences :

```
typedef struct {
    long freq;
    char liste[256];
    int len;
} frequence;
```

freq	len	liste	car	code	freq	len	liste	car	code
5	1	f	a		12	1	c	a	
9	1	e	b		13	1	b	b	
12	1	c	c		14	2	f e	c	
13	1	b	d		16	1	d	d	
16	1	d	e		45	1	a	e	1
45	1	a	f					f	0
freq	len	liste	car	code	freq	len	liste	car	code
14	2	f e	a		25	2	c b	a	
16	1	d	b	1	30	3	f e d	b	1
25	2	c b	c	0	45	1	a	c	0
45	1	a	d					d	1
			e	1				e	01
			f	0				f	00
freq	len	liste	car	code	freq	len	liste	car	code
45	1	a	a		100	6	a c b f e d	a	0
55	5	c b f e d	b	01				b	101
			c	00				c	100
			d	11				d	111
			e	101				e	1101
			f	100				f	1100

Figure 4: Fonction Huffman avec des tableaux

Chaque objet de type `frequence` représentera un arbre :

- l'entier `freq` contient la fréquence totale des caractères contenus dans la liste ;
- la `liste` d'entiers stocke tous les caractères regroupés sous cet arbre. Ce sera en fait un tableau d'au plus 256 caractères ;
- l'entier `len` précise la taille effective du tableau `liste`.

Plus précisément, comme toujours, un tableau doit être associé à sa taille effective, donc la forêt sera représentée par :

```
typedef struct {
    frequence *val;
    int len;
} table;
```

Un objet de type `table` contient donc un tableau de `frequence` dans le champ `val`, ainsi que sa taille effective dans le champ `len`.

Avec ces objets, l'évolution de la figure 3 devient comme présenté sur la figure 4 : lors du regroupement de deux listes, on ajoute un '0' en tête du code de tous les éléments de la première liste, puis un '1' en tête du code de tous les éléments de la deuxième liste.

### 3.3 Entrées-sorties

Une des difficultés majeures sont les entrées-sorties. En effet, le fichier à compresser doit être lu, puis manipulé. Le résultat compressé doit être enregistré bit par bit, lors du codage de chaque caractère. Un certain nombre de fonctions sont donc mises à votre disposition pour gérer cela. Elles sont décrites dans les fichiers `my_io.c` et `my_io.h` :

- on commence par construire un nouveau type `string`, qui est en fait une chaîne de caractères, avec sa longueur. On pourra se demander pourquoi on redéfinit ce type `string`, qui semble être contenu dans le type “ chaîne de caractères ”, qui admet déjà une fonction de longueur (fonction `strlen`). Cependant, les objets manipulés dans ce projet seront des contenus de fichiers, c’est-à-dire des séries d’octets qui peuvent prendre toutes les valeurs, de 0 à 255. Or, dans une chaîne de caractères, l’octet de valeur 0 représente le caractère ‘\0’, et donc le caractère de fin de chaîne : le contenu du fichier serait alors tronqué à la première occurrence d’un octet nul. Par conséquent, il n’est pas question d’utiliser la fonction `strlen` sur le champ `content`.

```
typedef struct {
    char * content;
    long length;
} string;
```

On fournit aussi des fonctions de création de tels objets, et de conversion. En effet, tout le programme manipulera le contenu du fichier “ décompressé ” (soit en entrée de la compression, ou en sortie de la décompression) sous forme de variable de type `string`. Voici donc les fonctions utiles :

- `string InitString(long len);`  
crée et alloue la mémoire pour un objet de type `string`, de longueur `len`.
- `string ReadString(char *file);`  
retourne le contenu du fichier de nom `file` sous forme de `string`, avec donc l’ensemble des caractères le constituant ainsi que sa taille.
- `void WriteString(char *file, string s);`  
sauvegarde la `string s` dans le fichier `file`.
- on fournit une fonction simple de conversion d’un `char` ou tout entier, en entier compris entre 0 et 255. Ceci sera utile pour accéder à une case d’un tableau, en utilisant un caractère comme index (alors que le langage C souhaite un `unsigned int`) :  
`unsigned short int byte(int c);`
- comme certaines erreurs pourront apparaître lors de la manipulation des fichiers, une fonction d’erreur est fournie, qui indique le type d’erreur, et stoppe l’exécution du programme. Elle pourra être utilisée dans votre projet pour interrompre l’exécution lors de la détection d’erreurs :  
`void error(int err);`
- une série de fonctions sont proposées pour écrire des bits dans un fichier, avec quelques options supplémentaires utiles :
  - `void WriteBitOpen(char *file);`  
crée le flux associé au fichier de nom `file`, et initialise la gestion de sauvegarde bit à bit.
  - `void WriteBit(int b);`  
sauvegarde le bit `b` (qui est un entier valant 0 ou 1) dans le flux créé ci-dessus.
  - `int WriteInt(long val);`  
sauvegarde un entier `val` dans le flux créé ci-dessus (on ignorera les 2 bits de poids fort, ce qui limite l’entier à  $2^{30}$ ). Attention, il y a des contraintes pour l’utilisation de cette fonction, puisqu’il faut avoir sauvegardé un multiple de 8 bits avant. Dans ce cas favorable, la fonction retourne 0, dans le cas défavorable, elle retourne 1. Mais *a*

*priori*, cette fonction ne sera utilisée qu'avant avoir sauvegardé le moindre bit (pour sauver le nombre total de caractères, ainsi que la table de fréquences).

- `void WriteBitClose();`  
clos le flux ci-dessus, en sauvegardant le buffer en cours. Contrairement au `fclose` qui se fait automatiquement à la fin d'un programme, l'omission de l'appel à cette fonction provoquera la perte des derniers bits en attente dans le buffer.
- une série de fonctions sont proposées pour lire des bits dans un fichier :
  - `void ReadBitOpen(char *file);`  
crée le flux associé au fichier de nom `file`, et initialise la gestion de lecture bit à bit.
  - `int ReadBit();`  
lit et retourne le bit en tête du flux créé ci-dessus. L'index sur le flux avance d'un cran.
  - `long ReadInt();`  
lit et retourne l'entier `val` marqué par l'index sur le flux créé ci-dessus. De même que pour la sauvegarde, il y a la contrainte d'avoir précédemment lu un multiple de 8 bits. Dans le cas contraire, l'entier  $2^{31}$  est retourné.
  - `void ReadBitClose();`  
clos le flux ci-dessus.

### 3.4 Fonctions à programmer

La compression/décompression repose sur l'utilisation de la table de codage, qu'il convient de construire en premier, ensuite la compression et la décompression sont de simples substitutions :

- La fonction `huffman` consiste à construire la table de codage à partir de la table de fréquences. Pour cela, il faut donc commencer par
    - lire le fichier à compresser sous forme de `string` ;
    - construire et remplir la table de fréquences de type `table` ci-dessus ;
    - construire la table de codage `codage` :
      1. l'initialiser à des codes vides ;
      2. trier la table de fréquences pour faire apparaître en tête les moins fréquents ;
      3. mettre à jour le codage des éléments des listes des deux éléments de tête de cette table de fréquences (en ajoutant un '0' ou un '1' en tête des bonnes cases de `codage`) ;
      4. concaténer ces deux listes (mettre le contenu du deuxième tableau à la suite du contenu du premier), en mettant à jour la valeur de la fréquence ;
      5. supprimer le deuxième élément (une solution simple est d'imposer la fréquence de cet élément à `MAXLONG` —en pensant à inclure `values.h` qui définit cette constante—, ainsi lors du prochain tri, il se trouvera en toute fin, au delà de la taille effective `len` de ce tableau `val`, qui devra aussi être mise à jour) ;
      6. tant que ce tableau a une taille effective strictement supérieure à 1, on retourne à l'étape 2.
- la variable `codage` contient alors une table de codage optimale pour le fichier à compresser.

- La compression est alors simple : il suffit de lire les caractères un à un, et de les remplacer par leur codage, en faisant la sauvegarde bit par bit dans le fichier destination. Pour la décompression, un certain nombre d'informations seront nécessaires, notamment pour reconstruire la table de codage : le nombre de caractères au total, ainsi que la table de

fréquences. On commencera donc par enregistrer ces données dans le fichier destination, avant de procéder à la conversion bit à bit.

- La décompression est un peu plus délicate, en raison de l'absence d'arbre. Tout d'abord, on lit dans le fichier compressé le nombre de caractères du fichier résultat (afin de créer la variable de type `string` qui recevra le fichier décompressé), ainsi que la table de fréquences. On peut alors reconstruire la table de codage. Pour le décodage :
  - on lit un bit, puis on cherche s'il code un caractère dans la table de codage. Si oui, on a le caractère décompressé.
  - sinon, on lit un deuxième bit, puis on cherche si les deux bits codent un caractère, etc.

En raison du code sans préfixe, cette méthode décode parfaitement (de façon laborieuse, faute de structure d'arbre, mais c'est la machine qui fait cette recherche...).

## 4 Tests à effectuer

Le format d'appel de ce programme devra être le suivant :

```
huffman <c|d> <file-in> <file-out>
```

où

- le premier argument précise s'il s'agit de la compression ou de la décompression ;
- le second argument indique le fichier en entrée (à compresser ou à décompresser) ;
- le troisième argument indique le fichier de sortie (où sera sauvegardé le résultat).

Puis le test effectué par la commande `make test` sera sur le fichier nommé `example` : il fera tout d'abord la compression, puis la décompression, affichera les tailles respectives, et comparera le fichier décompressé avec le fichier initial :

```
test : huffman
      ./huffman c example example1
      ./huffman d example1 example2
      ls -la example*
      diff example example2
```

## 5 Améliorations

Un des inconvénients majeurs de la version programmée ci-dessus est la nécessité d'une en-tête assez importante : sauvegarde de la table des fréquences. Ainsi, une en-tête d'un kilo-octet est à ajouter au fichier converti.

Une possibilité pour éviter cette en-tête est de faire une table de codage dynamique, créée en fonction d'une table de fréquences partielle : pour convertir le  $i$ -ème caractère, on fabrique la table de fréquences sur les  $i - 1$  premiers caractères, ainsi que la table de codage associée. On en déduit le code du  $i$ -ième caractère. Cette méthode évite donc la sauvegarde d'une quelconque table de fréquence, mais n'est plus optimale en place (pour ce qui est de la partie exclusivement consacrée au codage en chaînes binaire, mais on économise un kilo-octet au départ), et est beaucoup plus coûteuse en temps. En contre-partie, elle permet une compression à la volée, en une seule passe, et donc permet de compresser un flux de données sans fin : on n'a pas besoin de lire tout le fichier avant de commencer la compression.