Compression de Huffman

Juliusz Chroboczek

11 octobre 2014

1 Introduction

Un *compresseur sans pertes* ou *compresseur entropique* est un programme qui prend un fichier en entrée et génère un fichier plus court dont on peut extraire le fichier d'origine. Vous connaissez plusieurs logiciels qui contiennent un compresseur — *Zip*, *Gzip*, *7-Zip*, *Rar*, etc.

L'algorithme de Huffman est un algorithme classique de compression entropique. L'algorithme de Huffman est simple et flexible, et est souvent utilisé au sein d'un programme de compression plus complexe. Par exemple, l'algorithme *Deflate* utilisé par *Zip* et *Gzip* fait une première passe qui factorise les chaînes qui se répètent dans le fichier, puis applique l'algorithme de Huffman au résultat. De même, l'algorithme de compression d'images *JPEG* calcule une transformée de Fourier (modifiée) de l'image puis applique l'algorithme de Huffman pour compresser les coefficients.

Le but de ce projet est d'implémenter en Caml un compresseur et un décompresseur de fichiers basés sur l'algorithme de Huffman. Le format de fichier utilisé vous est décrit en détail et il est obligatoire de s'y conformer — je vous fournis le binaire d'un compresseur, et vous devez obligatoirement tester l'interopérabilité de votre programme avec le mien.

1.1 Codages préfixes et algorithme de Huffman

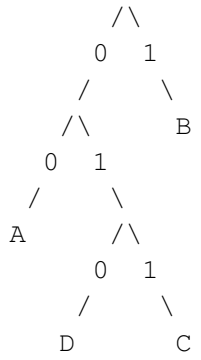
L'algorithme de Huffman réduit la taille des données en codant les symboles (les octets du fichier d'entrée) par un nombre variable de bits : les symboles les plus courants seront codés par un petit nombre de bits, tandis que les plus rares par un nombre de bits plus grand.

Le codage que construit l'algorithme de Huffman est un *codage préfixe* : aucun code n'est un préfixe d'un autre. Un codage préfixe peut être représenté par un arbre binaire, où les symboles à coder sont placés aux feuilles : le code associé à un symbole est obtenu en suivant le chemin menant à ce symbole depuis la racine de l'arbre, où une arête vers la gauche représente le bit 0, et une arête vers la droite le bit 1 (figure 1). L'algorithme de Huffman prend en entrée la fréquence f_s de chaque symbole s (le nombre de fois qu'il apparaît) et construit un codage préfixe optimal, au sens qu'il minimise la valeur

$$\sum_s f_s \cdot l_s$$

où f_s est la fréquence du symbole s, et l_s est la longueur du codage de s [1].

1. On peut faire mieux que Huffman en utilisant un nombre fractionnaire de bits par symbole ; c'est ce que fait le *codage arithmétique*, qui est cependant rarement utilisé car complexe et lent.



Symbole	Code
A	00
B	1
C	011
D	010

FIGURE 1 – Un arbre et le codage préfixe correspondant

1.2 Format de fichier

Votre compresseur-décompresseur devra être *interopérable* avec celui que j'ai écrit — un fichier généré par votre code devra pouvoir être interprété par le mien, et un fichier généré par le mien devra pouvoir être interprété par le vôtre. Une solution non-interopérable ne sera pas acceptée.

Un fichier compressé aura l'extension « `.hf` » et sera structuré comme suit :

— 4 octets « magiques » ayant la valeur 87_{16}, $4A_{16}$, $1F_{16}$, 48_{16} ;
— un octet de valeur *n* suivi de *n* octets ignorés par le décompresseur ;
— la représentation de l'arbre ;
— la suite de codes compressés ;
— de 0 à 7 bits valant 0, de façon à aligner la suite du fichier sur une frontière d'octet ;
— un octet *m* suivi de *m* octets ignorés.

Entête L'entête consiste de quatre octets, et permet d'identifier un fichier compressé.

Octets ignorés Il y a dans le fichier deux plages d'octets qui sont ignorés, après l'entête et à la fin du fichier. Ces plages peuvent vous servir à implémenter des extensions tout en restant interopérable (paragraphe 4). Si vous ne voulez pas vous en servir, vous pouvez simplement écrire un octet 0 (indiquant zéro octets ignorés) et les ignorer lors de la lecture.

Représentation de l'arbre Pour pouvoir décoder un codage préfixe, il faut connaître soit le codage lui-même, soit, de façon équivalente, l'arbre dont il est dérivé. Le fichier compressé contient donc une représentation d'un parcours préfixe de l'arbre, ce qui est suffisant pour reconstruire l'arbre.

Codes compressés La représentation de l'arbre est immédiatement suivie de la suite des codes correspondant aux symboles du fichier d'origine. Notez que cette plage ne commence pas forcément à une frontière d'octet.

1.3 Entrées-sorties bit-à-bit : la bibliothèque *bitio*

Le compresseur traite son fichier d'entrée comme un flot d'octets de 8 bits, mais sa sortie est un flot de bits. Inversément, le décompresseur lit un flot de bits et produit un flot d'octets.

Les fichiers Unix sont des flots d'octets, et il est facile de manipuler un fichier d'octets binaires en Caml (`open_in_bin`, `open_out_bin`, `input_byte`, `output_byte`, etc.). Par contre, Unix ne supporte pas les flots de bits — il faut donc les simuler. Pour cela, je vous fournis `bitio.ml`, une bibliothèque qui permet de manipuler les fichiers Unix comme si c'étaient des flots de bits. Toutes les fonctions exportées par cette bibliothèque sont décrites dans les commentaires du fichier `bitio.mli`.

À vous de décider si vous désirer vous servir de `bitio.ml`, de la modifier, ou si vous préférez écrire vos propres fonctions d'entrées-sorties bit-à-bit. Si vous décidez de la modifier, le fichier `bitio_test.ml` contient une suite de tests sommaire.

2 Opération du compresseur

Le compresseur prend un fichier (une suite d'octets) en entrée et construit sa représentation compressée (une suite de bits) dans un autre fichier.

2.1 Construction de l'histogramme

Votre compresseur devra d'abord parcourir le fichier d'entrée et compter le nombre d'occurrences de chaque symbole (sa *fréquence*). La structure de données contenant toutes les fréquences s'appelle l'*histogramme*.

2.2 Construction de l'arbre

L'algorithme de Huffman manipule une collection d'arbres binaires dont les feuilles sont étiquetées par les symboles (une *forêt*), dont chacun est décoré d'un entier qui s'appelle son *poids*. Étant donnés deux arbres Λ et M de poids respectifs $p(\Lambda)$ et $p(\mathrm{M})$, la *fusion* de Λ et M est l'arbre dont le fils gauche est Λ, le fils droit est M, et dont le poids est $p(\Lambda) + p(\mathrm{M})$ (voir figure 2).

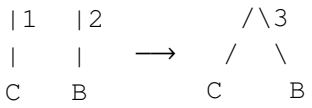


FIGURE 2 – Fusion

On commence avec la forêt triviale, qui a un arbre pour chaque symbole de fréquence non-nulle (les symboles de fréquence nulle sont inutiles), et dont le poids est la fréquence de ce symbole. À cette forêt, il faut ajouter un arbre représentant le *symbole de fin de fichier*, un symbole aditionnel de poids 1 qui servira à marquer la fin du fichier.

À chaque étape, on choisit deux arbres de poids minimal, on les enlève de la forêt, on les fusionne, et on insère le résultat dans la forêt. L'algorithme termine lorsque la forêt ne contient plus qu'un seul arbre (figure 3).

Comme on recherche à chaque étape les deux arbres de poids le plus faible, il est utile de représenter la forêt comme une liste chaînée ordonnée par poids croissant. Il faudra donc

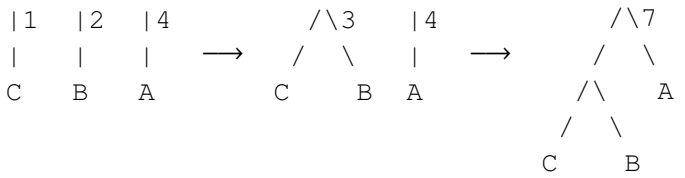


FIGURE 3 – Construction de l'arbre

s'assurer que la liste est initialement triée, et qu'à chaque étape on insère le nouvel arbre au bon endroit [2]. Je vous conseille d'afficher l'arbre dès que vous aurez implémenté cette partie.

2.3 Écriture de l'entête

Votre compresseur écrira tout d'abord un entête consistant des quatre octets donnés au paragraphe 1.2. Il écrira ensuite un octet de valeur 0, indiquant qu'il n'y a pas de données à ignorer.

2.4 Stockage de l'arbre

Pour pouvoir interpréter les données compressées, il faut avoir connaissance soit de l'arbre soit du codage utilisé. J'ai choisi de stocker une représentation de l'arbre dans le fichier.

Pour générer cette représentation, parcourez l'arbre en profondeur. À chaque fois que vous rencontrez un nœud interne, écrivez un bit 1 suivi de la représentation du fils gauche suivie immédiatement de la représentation du fils droit. Lorsque vous rencontrez une feuille portant le symbole s, écrivez un bit 0 suivi

— des 8 bits de s, en ordre gros-boutiste *(big-endian)* si $s \leq 254$;
— des 9 bits 1111 1111 0 si $s = 255$;
— des 9 bits 1111 1111 1 si s est le symbole de fin de fichier.

Par exemple, l'arbre donné en exemple à la figure 1 sera représenté par la suite de bits

$$110\ A\ 1\ 0\ D\ 0\ C\ 0\ B$$

où A, B, C et D sont les suites de huit ou neuf bits représentant les symboles respectifs.

2.5 Écriture des codes

L'arbre est immediatement suivi de la suite des codes correspondant aux symboles contenus dans le fichier d'entrée. Pour écrire ceux-ci, « rembobinez » le fichier d'entrée (à l'aide de la fonction `seek_in`), puis, pour chaque octet de celui-ci, écrivez le code correspondant. Ensuite, écrivez le code de fin de fichier.

Comme il est difficile de déterminer le bon code directement à partir de l'arbre, vous aurez probablement besoin de construire une autre structure de données.

2. Ce qui donne un algorithme en $O(n^2)$. On peut faire mieux en remarquant qu'il s'agit en fait d'une file de priorité, et qu'en l'implémentant par un tas *(heap)* on obtient un algorithme en $O(n \log n)$.

3 Opération du décompresseur

Le décompresseur fait le travail inverse du compresseur.

3.1 Vérification de l'entête et octets ignorés

Votre décompresseur lira quatre octets de son fichier d'entrée, et vérifiera qu'ils ont bien les valeurs données au paragraphe 1.2. Si ce n'est pas le cas, ce qui indique que le fichier en entrée n'est pas un fichier au bon format, il affichera un message d'erreur puis terminera.

Il lira ensuite un octet *n*, puis lira *n* octets qu'il ignorera.

3.2 Lecture de l'arbre

Votre décompresseur devra ensuite reconstruire l'arbre qui est stocké dans le fichier. Pour cela, lisez un bit ; si c'est un 0, lisez les 8 ou 9 bits qui suivent, et retournez une feuille. Si c'est un 1, récursez deux fois — lisez deux arbres, puis retournez la fusion des deux.

3.3 Écriture des symboles

Votre décompresseur devra ensuite lire la suite de codes et écrire la suite de symboles correspondante. Pour cela, lisez les bits du fichier d'entrée un par un tout en parcourant l'arbre — vers la gauche lorsque vous rencontrez un bit 0, vers la droite lorsque vous rencontrez un 1. À chaque fois que vous arrivez à une feuille, écrivez l'octet contenu dans celle-ci, puis reprenez l'arbre à la racine.

Arrêtez-vous lorsque vous trouvez le symbole de fin de fichier. Si le fichier d'entrée se termine avant de trouver le symbole de fin de fichier, affichez un message d'erreur, puis terminez.

Après le symbole de fin de fichier, vous devrez trouver 0 à 7 bits de 0 suivis d'un octet *m* et de *m* octets que vous ignorerez. Si ce n'est pas le cas, affichez un message d'erreur.

4 Extensions

Toutes les extensions seront les bienvenues, et leur présence sera prise en compte lors de l'évaluation.

Il se peut que vous ayez besoin d'inclure des données supplémentaires dans votre fichier pour implémenter l'extension dont vous rêvez. Les deux plages d'octets réservés sont prévues pour cela : vous pouvez y inclure les données supplémentaires dont vous avez besoin. Votre décompresseur devra obligatoirement fonctionner même si ces plages sont vides.

Afin d'éviter les collisions, je vous conseille d'utiliser un format facilement reconnaissable pour les plages d'extensions (par exemple en les commençant par un entête magique), et de prendre soin de les ignorer si elles ne correspondent pas à votre format.

5 Modalités d'évaluation

Il s'agit d'un projet, pas d'un TP — vous devez donc nous fournir un programme qui fait quelque chose. Il vaut mieux faire une partie du projet qui fonctionne, que d'essayer de tout faire et nous fournir un programme où rien ne fonctionne.

L'interopérabilité avec mon code est obligatoire, et vous devez absolument tester les fichiers produits par votre compresseur avec mon décompresseur. Un programme qui implémente l'algorithme de Huffman mais qui utilise un format de fichier différent ne sera pas accepté.

Le compresseur et le décompresseur consistent d'une partie algorithmique, qu'il est naturel d'écrire de manière fonctionnelle, et d'une partie qui fait des entrées-sorties, qu'il vaut peut-être mieux écrire de façon impérative. Nous apprécierons les programmes qui font preuve de bon goût dans les choix d'implémentation, et nous nous autoriserons à pénaliser votre projet juste parce qu'il est moche.

Nous nous réservons par ailleurs le droit de prendre tout autre critère en compte, y inclus la vitesse d'exécution, la consommation de mémoire, la gestion des erreurs, l'élégance de la ligne de commande, la présence de documentation, etc.

6 Modalités de soumission

Le projet est à faire par groupes de deux personnes : nous refuserons les soumissions faites par trois personnes ou plus. Nous accepterons les soumissions individuelles, mais nous vous les déconseillons : ce sujet est assez long et il est prévu pour un groupe de deux. Les soutenances se feront par groupe, les notes seront individuelles.

La date limite de soumission est le mercredi 31 décembre à minuit. Vous trouverez un lien vers le site de soumission sur ma page *web* [3].

Vous devrez nous soumettre une archive compressée `tar.gz` nommée *`nom1-nom2`*`.tar.gz`, où *nom1* et *nom2* sont vos noms, et qui s'extraira dans un répertoire *`nom1-nom2/`*. Par exemple, si vous vous appelez Georges Brassens et Jacques Brel, vous devrez nous soumettre une archive nommée `brassens-brel.tar.gz` et qui s'extraira dans un répertoire `brassens-brel/`.

Votre archive devra contenir :

- les sources de votre programme ;
- les éventuels fichiers nécessaires pour compiler votre programme ;
- un fichier nommé `README` qui contiendra vos noms complets et indique comment on se sert de votre programme ;
- un rapport au format *PDF* de une à quatre pages qui indiquera notamment quelle partie du projet vous aurez traitée, les *bugs* connus dans votre code, et les extensions que vous aurez implémentées ;
- tout-autre fichier que vous jugerez pertinent (par exemple une documentation au format « `nroff -man` »).

L'archive ne devra pas contenir de versions compilées de votre code.

3. `http://www.pps.univ-paris-diderot.fr/~jch/enseignement/pf/`