Codage du Huffman

Mai 2010 – Joris Blatti

Théorie

Introduction

Le codage de Huffman est un algorithme de compression de données sans perte élaboré par David Albert Huffman, lors de sa thèse de doctorat au MIT. L'algorithme a été publié en 1952 dans l'article « A Method for the Construction of Minimum-Redundancy Codes », dans les « Proceedings of the Institute of Radio Engineers ». Le codage de Huffman utilise un code à longueur variable pour représenter un symbole de la source (par exemple un caractère dans un fichier). Le code est déterminé à partir d'une estimation des probabilités d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents. Les codes de Huffman sont des codes optimaux, au sens de la plus courte longueur.

Un code de Huffman est optimal pour un codage par symbole, et une distribution de probabilité connue. Il ne permet cependant pas d'obtenir les meilleurs ratios de compression. Des méthodes plus complexes réalisant une modélisation probabiliste de la source et tirant profit de cette redondance supplémentaire permettent d'améliorer les performances de compression de cet algorithme (voir Lempel-Ziv).

Principe du codage

Le principe du codage de Huffman repose sur la création d'un arbre composé de nœuds. Supposons que la phrase à coder est « Wikipédia ». On recherche tout d'abord le nombre d'occurrences de chaque caractère (ici les caractères 'a', 'd', 'é', 'k', 'p' et 'w' sont représentés chacun une fois et le caractère 'i' trois fois). Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids valant son nombre d'occurrences. Puis l'arbre est créé suivant un principe simple : on associe à chaque fois les deux nœuds de plus faibles poids pour donner un nœud dont le poids équivaut à la somme des poids de ses fils jusqu'à n'en avoir plus qu'un, la racine. On associe ensuite par exemple le code 0 à la branche de gauche et le code 1 à la branche de droite.

Pour obtenir le code binaire de chaque caractère, on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie. Il est en effet nécessaire de partir de la racine pour obtenir les codes binaires car lors de la décompression, partir des feuilles entraînerait une confusion lors du décodage. Ici, pour coder 'Wikipédia', nous obtenons donc en binaire : 101 11 011 11 100 010 001 11 000, soit 24 bits au lieu de 63 (9 caractères x 7 bits par caractère) en utilisant les codes ASCII (7 bits).

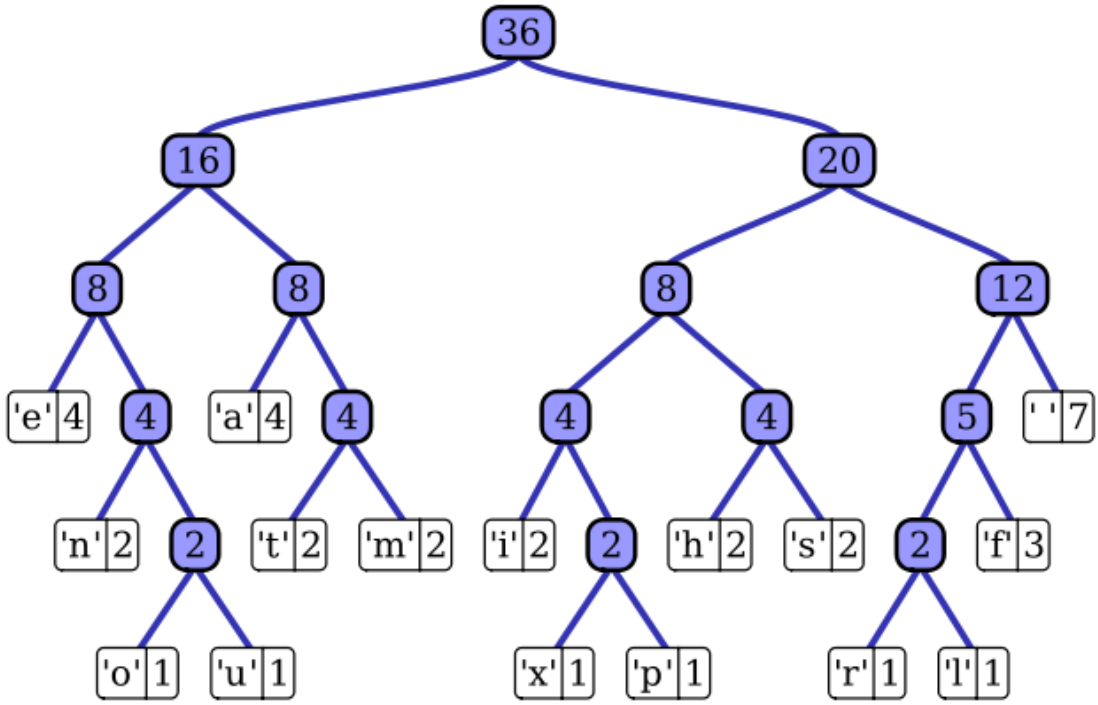


Fig1- Un arbre d'exemple avec la phrase "this is an example of a huffman tree"

Il existe trois variantes de l'algorithme de Huffman, chacune d'elle définissant une méthode pour la création de l'arbre :

1. **Statique :** chaque octet a un code prédéfini par le logiciel. L'arbre n'a pas besoin d'être transmis, mais la compression ne peut s'effectuer que sur un seul type de fichier (ex: un texte en français, où les fréquences d'apparition du 'e' sont énormes; celui-ci aura donc un code très court, rappelant l'alphabet morse).
2. **Semi-adaptatif :** le fichier est d'abord lu, de manière à calculer les occurrences de chaque octet, puis l'arbre est construit à partir des poids de chaque octet. Cet arbre restera le même jusqu'à la fin de la compression. Il sera nécessaire pour la décompression de transmettre l'arbre.
3. **Adaptatif :** c'est la méthode qui offre a priori les meilleurs taux de compression car l'arbre est construit de manière dynamique au fur et à mesure de la compression du flux. Cette méthode représente cependant le gros désavantage de devoir modifier souvent l'arbre, ce qui implique un temps d'exécution plus long. Par contre la compression est toujours optimale et le fichier ne doit pas être connu avant de compresser. Il ne faut donc pas transmettre ou stocker la table des fréquences des symboles. De plus, l'algorithme est capable de travailler sur des flux de données (streaming), car il n'est pas nécessaire de connaitre les symboles à venir.

Limitations du codage de Huffman

On peut montrer que pour une source X, d'entropie H(X) la longueur moyenne L d'un mot de code obtenu par codage de Huffman vérifie:

$$H(X) \leq L \leq H(X) + 1$$

Cette relation, qui montre que le codage de Huffman s'approche effectivement de l'entropie de la source et donc de l'optimum, peut s'avérer en fait assez peu intéressante dans le cas où l'entropie de la source est faible, et où un surcoût de 1 bit devient important. De plus le codage de Huffman impose d'utiliser un nombre entier de bit pour un symbole source, ce qui peut s'avérer peu efficace. Une solution à ce problème est de travailler sur des blocs de n symboles. On montre alors qu'on peut

s'approcher de façon plus fine de l'entropie:

$$H(X) \leq L \leq H(X) + \frac{1}{n}$$

mais le processus d'estimation des probabilités devient plus complexe et coûteux.

De plus, le codage de Huffman n'est pas adapté dans le cas d'une source dont les propriétés statistiques évoluent au cours du temps, puisque les probabilités des symboles sont alors erronées. La solution consistant à ré-estimer à chaque itération les probabilités symboles est impraticable du fait de sa complexité. La technique devient alors le codage Huffman adaptatif : à chaque nouveau symbole la table des fréquences est remise à jour et l'arbre de codage modifié si nécessaire. Le décompresseur faisant de même pour les mêmes causes, il reste synchronisé sur ce qu'avait fait le compresseur.

En pratique, lorsque l'on veut s'approcher de l'entropie, on préfèrera un codage arithmétique qui est optimal au niveau du bit.

Utilisations

Le codage de Huffman ne se base que sur la fréquence relative des symboles d'entrée (suites de bits) sans distinction pour leur provenance (images, vidéos, sons, etc.). C'est pourquoi il est en général utilisé au second étage de compression, une fois la redondance propre au média mise en évidence par d'autres algorithmes. On pense en particulier à la compression JPEG pour les images, MPEG pour les vidéos et MP3 pour le son, qui peuvent retirer les éléments superflus imperceptibles pour les humains. On parle alors de compression avec perte.

D'autres algorithmes de compression, dits sans perte, tels que ceux utilisés pour la compression de fichiers, utilisent également Huffman pour comprimer le dictionnaire résultant. Par exemple, LZH (Lha) et deflate (ZIP, gzip) combinent un algorithme de compression par dictionnaire (dit de Lempel- Ziv) et un codage entropique de Huffman.

Anecdote

Les premiers Macintosh de la société Apple utilisaient un code inspiré de Huffman pour la représentation des textes : les 15 caractères les plus fréquents d'une langue étaient codés sur 4 bits, et la 16ème configuration servait de préfixe au codage des autres sur un octet (ce qui faisait donc tantôt 4 bits, tantôt 12 bits par caractère). Cette méthode simple se révélait économiser 30% d'espace sur un texte moyen, à une époque où la mémoire vive restait encore un composant coûteux.

Source : Wikipedia

Implémentation & Réalisation

Le but de cette partie est de vous donner un exemple complet du fonctionnement de l'algorithme

ainsi que le cheminement logique à mettre en œuvre pour appliquer le principe du codage de Huffman.

Exemple

Le but de cet exemple est de savoir « comment ça marche » nous prenons donc comme phrase à coder «THIS IS AN EXAMPLE OF A HUFFMAN TREE».

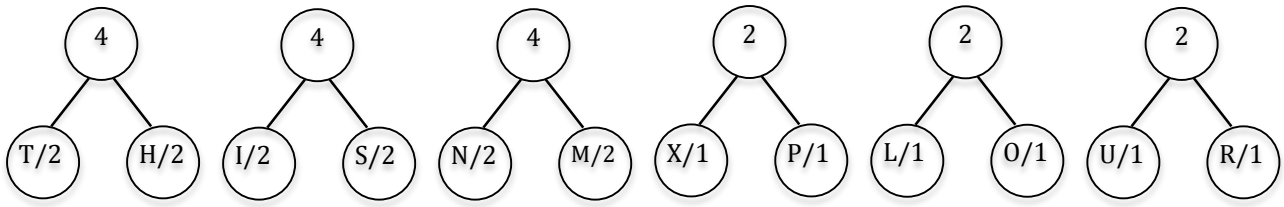
Etape 1 – Comptage du nombre d'occurrence

On va donc compter le nombre d'occurrence pour chaque caractère et les classer dans l'ordre décroissant comme dans le tableau ci-dessous.

‘ ‘	A	E	F	T	H	I	S	N	M	X	P	L	O	U	R
7	4	4	3	2	2	2	2	2	2	1	1	1	1	1	1

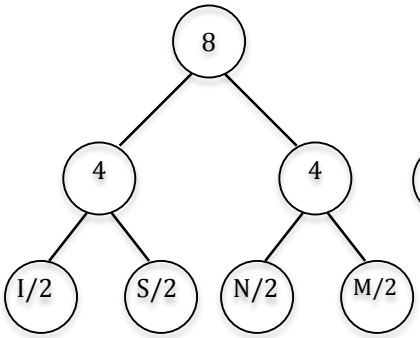
Etape 2 – Création de l'arbre de Huffman

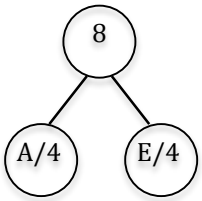
L'étape suivante et la mise en place de l'arbre binaire pour définir le codage de chaque caractère.

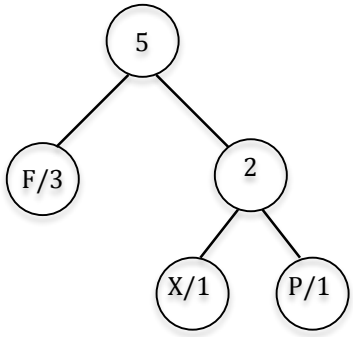


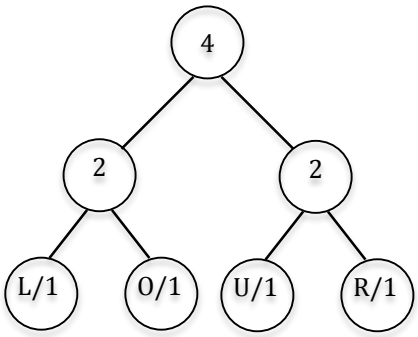
‘ ‘	TH	IS	NM	A	E	F	XP	LO	UR
7	4	4	4	4	4	3	2	2	2

Le principe est le suivant : On commence toujours par les occurrences les plus faibles que l'on assemble (1 & 1 -> 2). A chaque étape, c'est le même principe, on assemble les 2 probabilités les plus faibles, et ce, jusqu'à obtenir un arbre complet dit « équilibré » & « optimisé ». On pourrait imaginer de simplement remplir un arbre sans équilibrage en mettant en bas de l'arbre, les occurrences les plus faibles, mais bien que cette solution soit fonctionnelle elle n'est de loin pas optimale !

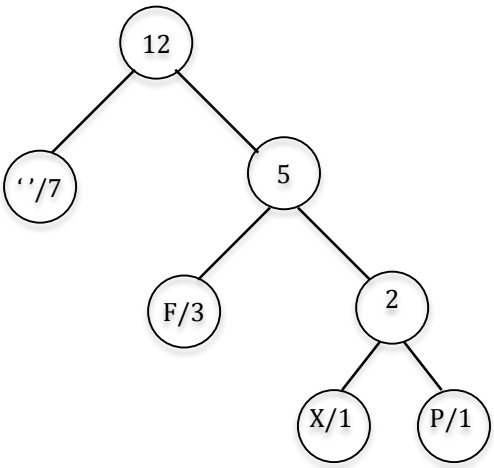


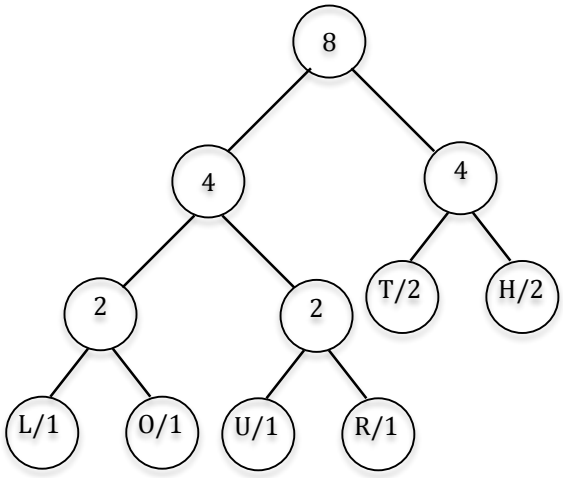




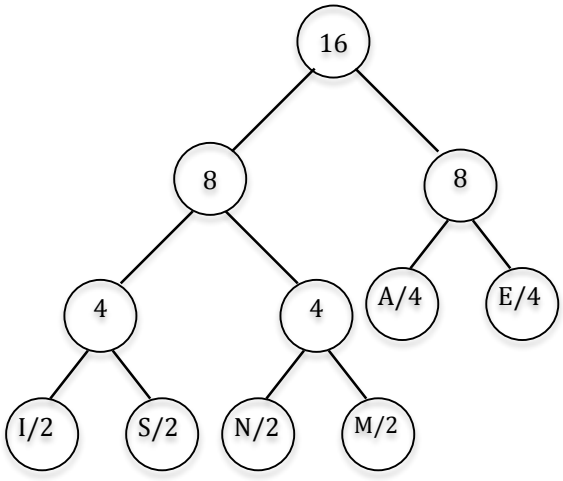


ISNM	AE	‘ ‘	FXP	LOUR	TH
8	8	7	5	4	4

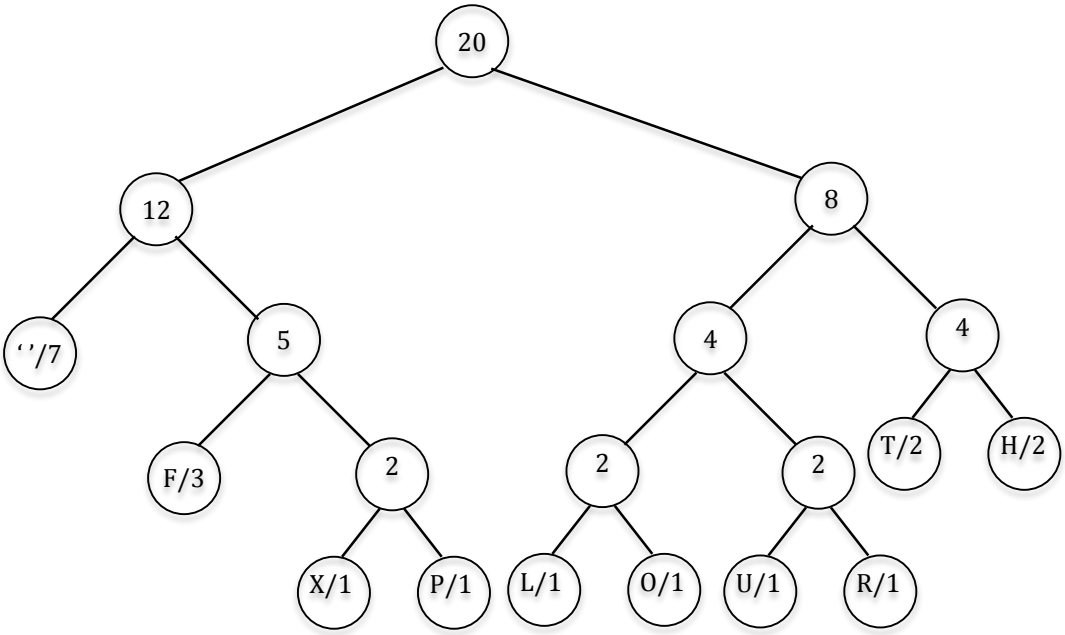




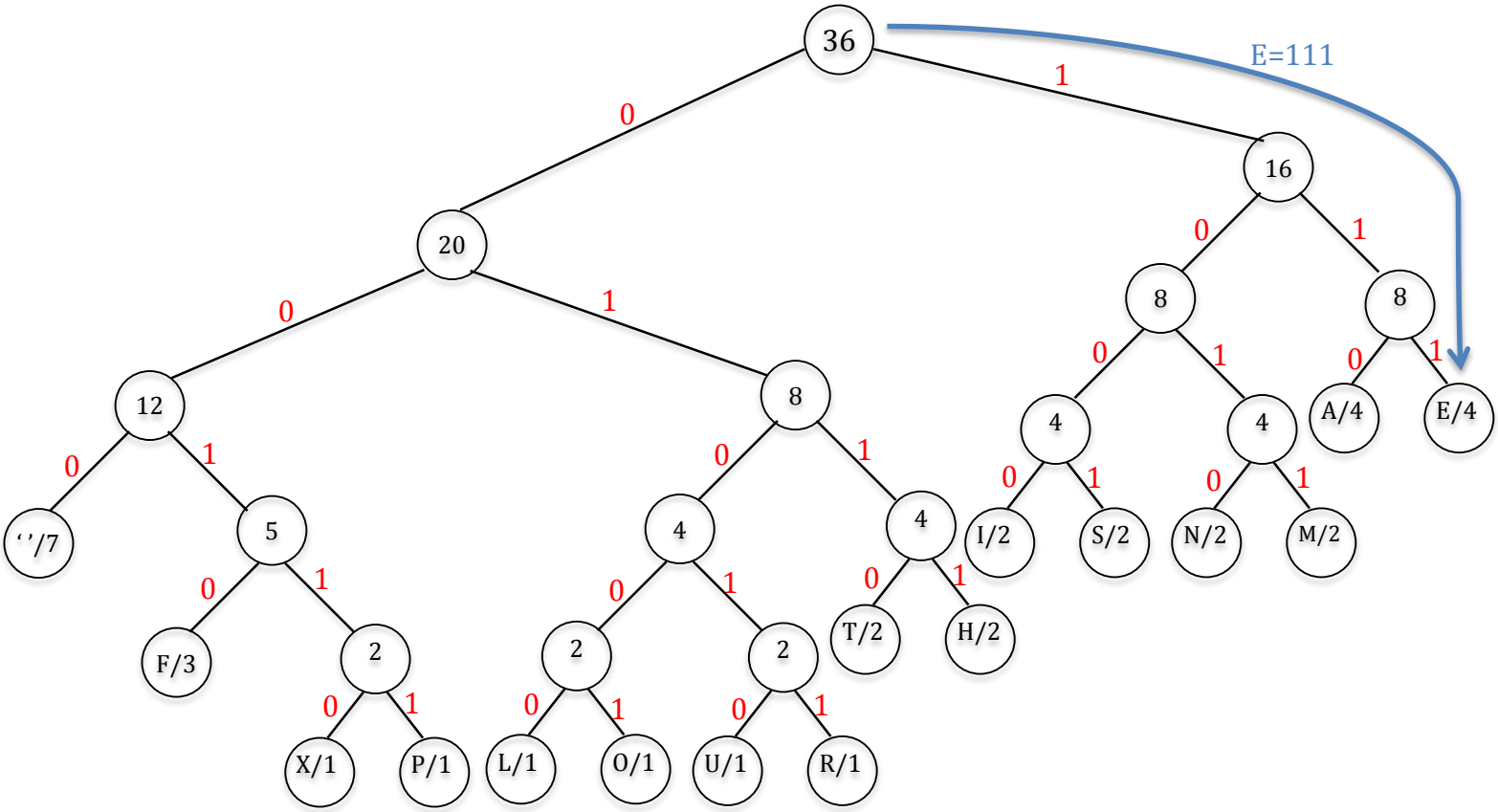
‘ ‘FXP	LOURTH	ISNM	AE
12	8	8	8



ISNMAE	‘ ‘FXP	LOURTH
16	12	8



‘ ‘FXPLOURTH	ISNMAE
20	16



Etape 3 – Interprétation de l'arbre & codage des caractères

La dernière étape consiste à interpréter l'arbre pour obtenir le codage de chaque caractère.

' '	000
F	0010
X	00110
P	00111
L	01000
O	01001
U	01010
R	01011
T	0110
H	0111
I	1000
S	1001
N	1010
M	1011
A	110
E	111

Finalement on obtient le code suivant pour la phrase «THIS IS AN EXAMPLE OF A HUFFMAN TREE» :

0110 0111 1000 1001 000 1000 1001 000 110 1010 000 111 00110 110 1011 00111 01000 111 000 01001 0010 000 110 000 0111 01010 0010 0010 1011 110 1010 000 0110 01011 111 111

qui utilise 135 bits au lieu de 36x7 = 252 bits.

Remarques pour l'implémentation

- L'exemple ci-dessus est fait avec des caractères alphanumériques, il est cependant, aisé de transposer ça avec des octets. On pourra ainsi traiter n'importe quel type de fichier.
- Prenez le temps de faire un pseudo-code pour l'implémentation de l'étape n° 2, la création de l'arbre optimisé et équilibré. C'est le point cruciale de l'algorithme de Huffman.