Julien Ducrey
Lionel Landwerlin

Le 29/01/04

Deuxième devoir de structures de données

Dans ce devoir de structures de données, il nous était demandé d'écrire un programme de compression de texte utilisant un arbre de Huffman et dont l'unité de compression est le mot. Dans ce rapport, nous allons d'abord décrire le fonctionnement d'un tel compresseur, puis nous verrons en détail la méthode, les structures de données et les algorithmes utilisés dans ce devoir, ensuite nous verrons le fonctionnement de notre compresseur et enfin nous commenterons les taux de compression entre notre programme et d'autre tel que gzip (utilisant l'algorithme Lempel-Ziv LZ77) ou encore bzip2 (utilisant l'algorithme Burrows-Wheeler ainsi qu'un codage de Huffman).

1.Méthodes de compressions à l'aide d'un arbre de Huffman :

On rappelle qu'ici, l'unité de base pour la compression est le mot. Ce sont donc des mots que l'on va « accrocher » aux feuilles de l'arbre de Huffman que nous allons construire. On rappelle aussi qu'un arbre de Huffman est un arbre binaire qui admet entre autres la relation suivante; le poids d'un noeud est égal à la somme des poids de son fils droit et de son fils gauche.

1.1.Principe de fonctionnement :

Le principe de compression consiste à remplacer chaque occurrence d'un mot par le code qui lui est attribué en fonction de sa place dans un arbre de Huffman. On part de la racine de l'arbre, on note 0 si l'on se déplace sur le sous noeud gauche ou bien 1 si l'on se déplace sur le sous noeud droit (il s'agit là d'une convention que l'on peut tout à fait reformuler). On obtient ainsi une suite de 0 et de 1, que l'on appellera ici un « *code de Huffman* », qui peut être écrite sous forme binaire.

Si les feuilles d'un arbre de Huffman sont pondérées, alors la longueur du code qui permet d'atteindre une feuille en partant de la racine de l'arbre est inversement proportionnelle au poids de la feuille. Ainsi, si l'on veut obtenir les codes les plus courts pour les mots les plus utilisés, il faudra pondérer chaque feuille de l'arbre avec le nombre d'occurrence du mot qui lui correspond.

1.2.Construction de l'arbre de Huffman :

Dans un premier temps on peut supposer que l'on connaît l'ensemble des mots d'un texte ainsi que pour chaque mot son nombre d'occurrence. On effectue alors une fois pour toute la création de l'arbre et il suffira donc de relire le texte que l'on souhaite compresser afin de remplacer chaque mot

par son code de Huffman. Nous appellerons cette méthode « *codage de Huffman statique* » ou « *codage de Huffman deux passes* » (puisqu'il est nécessaire d'effectuer deux lecture d'un texte pour le compresser). L'algorithme permettant de construire un tel arbre a été décrit par D. Huffman dans un article appelé « *A method for the construction of minimum redundancy codes.* » en 1952. C'est algorithme est optimal, c'est à dire que pour un ensemble de couple (mot, occurrence) obtenu à partir d'un texte, l'algorithme fournit un ensemble de codes pour lesquels si l'on remplace chaque mot du texte par son code de Huffman, on obtient un texte dont la longueur est minimal. C'est l'arbre obtenu à partir de l'algorithme de Huffman qui donne les codes selon le principe évoqué précédemment.

Il arrive parfois que l'on ne puisse pas relire un texte une deuxième fois à moins de garder une copie du texte en mémoire (le plus souvent cela est impossible car l'on ne dispose pas de suffisamment de mémoire). Dans ce cas il est donc impossible de remplacer les mots d'un texte par les codes de Huffman obtenu. Dans ce cas, particulièrement bien illustré sur les systèmes d'exploitation de type Unix où des processus peuvent communiquer par flux, l'algorithme de Huffman est donc caduc.

Depuis la publication de l'article de D. Huffman d'autres chercheurs on tenté de mettre au point des algorithmes permettant de compresser sans avoir besoin d'effectuer une première lecture destinée à trouver la fréquence des mots. N. Faller en 1973, R. Gallager en 1978 puis D. Knuth en 1985 ont contribuer à la conception de l'algorithme FGK (initiales des trois personnes énoncées avant) utilisé dans la commande Unix *compact*. Enfin J. Vitter en 1987 a décrit l'algorithme V basé sur l'algorithme FGK mais comportant certaines améliorations permettant une meilleure compression, c'est cet algorithme que nous avons implémenté dans notre programme. Les algorithmes FGK et V sont des algorithmes « *adaptatif* » ou « *dynamique* ». Ils utilisent tous les deux un arbre de Huffman qu'ils modifient au cour de la lecture d'un texte. Le compresseur et le décompresseur effectuant chacun de leur côté les même modifications obtiennent du coup un même arbre de Huffman.

a) Algorithme de Huffman :

Pour commencer, on place tous les mots ainsi que leur nombre d'occurrences dans une liste ou un tableau trié selon le nombre d'occurrence. Prenons la liste suivante (on ne représente ici que le nombre d'occurrence) :

12	9	8	7	5	4	2	1	1

Chacune des cases du tableau ci dessus représente une feuille (noeud extérieur) de l'arbre.

L'algorithme procède de la manière suivante : il doit créer un nouveau noeud qui deviendra le père des deux noeuds de poids minimum. On supprime alors ces deux noeuds de la liste, et on insère le nouveau noeud que l'on vient de créer en gardant la liste triée. On répète l'opération jusqu`à ce qu'il ne reste plus qu'un seul noeud dans la liste qui est de fait la racine de l'arbre.

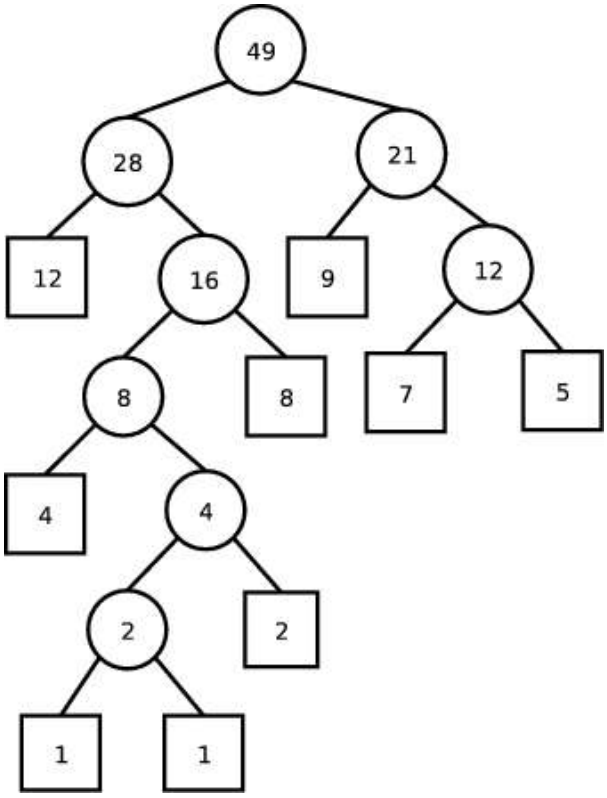
12	9	8	7	5	4	2	2	

On voit ci-dessus le résultat après une itération. On a remplacé les deux noeuds minimaux de poids 1 par un noeud de poids égal à la somme de ces deux noeuds qui deviennent alors ses fils. Ci-dessous les itérations restantes pour arriver à la racine.

12	9	8	7	5	4	4		
12	9	8	8	7	5			

12	12	9	8	8				
16	12	12	9					
21	16	12						
28	21							
49								

Ci dessous l'arbre obtenu :



On peut remarquer que lors de la création de l'arbre de Huffman, à chaque itération, il faut garder la liste triée. Un premier choix simple serait une insertion dans la liste déjà triée. Cette insertion est d'une complexité en temps de $O(n)$ avec n le nombre d'éléments dans la liste. On peut remarquer à l'itération 5 qu'il faut parcourir toute la liste pour venir insérer 16 (formé par les deux noeuds 8) en tête de liste. C'est le pire cas possible. Dans une liste de 100 éléments le nombre total de comparaison reste acceptable, mais dans une liste de 15000 éléments cela devient très coûteux. En effet il faut n itérations pour construire l'arbre avec une liste de n mots, cela reviendrait donc pour une liste de n mots au départ à un temps total de :

$$O\left(n \sum_{i=1}^{n} i\right) = n\ (n-1)\frac{n}{2} \approx \frac{n^3}{2}$$

Une solution moins coûteuse en temps consiste à utiliser deux listes desquelles on extrait les noeuds dans un ordre croissant selon leur poids. La première (L1) est la même liste de départ que précédemment, la deuxième (L2) est vide. Au début (L2) étant vide, on commence donc par retirer deux noeuds de (L1), et on créé un nouveau noeud parent des deux noeuds que nous venons de retirer dont le poids et comme précédemment la somme de ses deux fils. Ensuite on procède de la manière suivante tant que (L1) et (L2) ne sont pas vide. On retire de le noeud de poids minimum à (L1) et (L2) ((L1) et (L2) étant triées il n'y a qu'un seul test à effectuer). On extrait de nouveau le noeud de poids minimum à (L1) et (L2). On a maintenant 2 noeuds, on en créé un nouveau qui sera

le père des deux noeuds provenant de (L1) et/ou (L2) (un seul nouveau test à effectuer). On accroche le noeud créé à la fin de la liste (L2). Si l'une des deux liste se retrouve vide, on se retrouve alors dans la situation de départ.

Voilà la représentation sous forme de tableau de cet algorithme (chaque ensemble de colonnes (L1) et (L2) représente une étape de l'algorithme l'ordre de lecture est du haut vers le base et de gauche à droite) :

(L1)	(L2)	(L1)	(L2)	(L1)	(L2)
12					
9					
8		12			
7		9		12	
5		8		9	
4		7		8	
2		5		7	
1		4		5	
1		2	2	4	4

(L1)	(L2)	(L1)	(L2)	(L1)	(L2)
12					
9					
8		12			
7		9	12	12	16
5	8	8	8	9	12

(L1)	(L2)	(L1)	(L2)	(L1)	(L2)
	21	28			
12	16	21			49

Les noeuds de poids minimaux sortent par le bas des listes/tableaux (L1) et (L2). A chaque fois que l'on souhaite retirer un noeud, il suffit de prendre le minimum de l'union de (L1) et (L2) (cela nécessite 1 ou 2 tests, le premier pour savoir si (L2) est vide et, dans le cas ou elle n'est pas vide, pour prendre le minimum des deux).

On effectue donc un nombre constant d'opération à chaque itération (ce nombre dépendant de l'implémentation), ainsi que l'on diminue le nombre d'éléments contenu dans les deux listes par un. La complexité totale s'écrit donc naturellement :

$$O(n\ )$$

Avant de commencer à convertir les mots en codes binaires, le compresseur comme le

décompresseur doivent tout deux avoir un arbre qui leur permet de faire la conversion mot <-> code. Le compresseur créé cet arbre à partir du texte qu'il lit. Pour le décompresseur par contre, il est nécessaire de fournir cet arbre dans le fichier compressé. La manière la plus simple de stocker ces données serait d'écrire chaque mot avec son nombre d'occurrence. Or le nombre d'occurrence des mots ne sert que pour reconstruire l'arbre. On pourrait stocker directement la structure de l'arbre en faisant un parcours préfixe, infixe ou suffixe et en écrivant un bit 1 si l'on se trouve sur un noeud, un bit 0 si l'on se trouve sur un feuille. Cela permet de diminuer la taille du fichier compressé.

b) Algorithme V :

Les algorithmes FGK et V décrivent tous les deux des méthodes dynamique de création d'un arbre de Huffman. L'arbre est créé et maintenu pendant la lecture du texte et permet donc de compresser un texte en le parcourant une seule fois. Par la suite nous ne traiterons que le cas l'algorithme V.

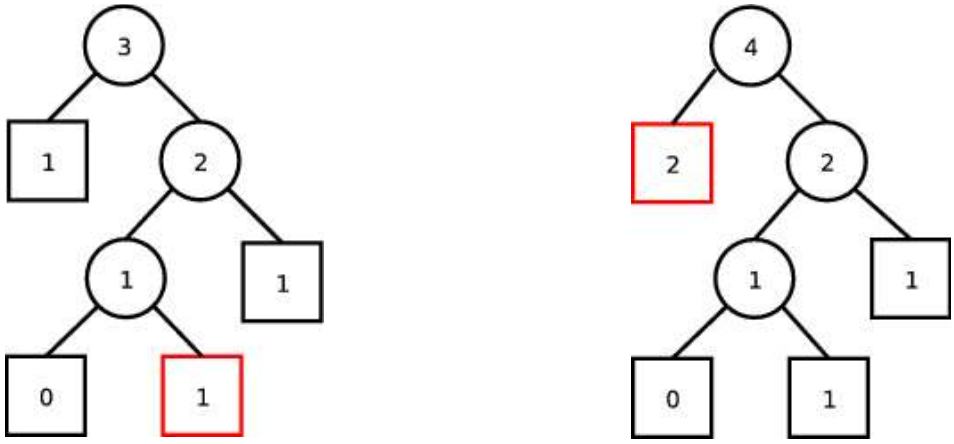
Il y a deux principales procédure de mise à jour; l'ajout d'une feuille et l'incrément du poids d'une feuille ou d'un noeud.

L'ajout d'une feuille à un arbre se fait en remplaçant la feuille la plus en bas et le plus à gauche par un noeud dont les deux fils sont la feuille que l'on remplace et la feuille que l'on ajoute.

Ci dessus un exemple d'ajout d'une feuille à l'arbre de gauche, il en résulte l'arbre de droite (en jaune la feuille ajoutée et en vert le noeud nécessaire à cet ajout).

Remarque : Pour le décompresseur il est nécessaire de savoir à quel moment il doit ajouter un nouveau mot et quel est ce nouveau mot. Pour cela le compresseur et le décompresseur commencent tous les deux avec un arbre comprenant déjà deux feuille. L'une sert de feuille d'ajout, son emplacement nous indique à quel endroit on doit ajouter les nouvelles feuilles dans l'arbre. L'autre est une feuille qui est écrite juste avant l'apparition d'un nouveau mot. De cette manière le décompresseur peut savoir quand il doit ajouter une nouvelle feuille à l'arbre.

L'incrément d'un noeud se fait de la manière suivante; on échange d'abord ce noeud avec le noeud le plus haut et le plus à droite de même poids (avant incrément). Une fois l'échange effectué, on incrémente le poids du noeud en question et on passe au père pour effectuer la même opération jusqu'à ce que l'on arrive à la racine. Pour mieux illustrer le travail de l'algorithme, prenons l'exemple suivant :



A gauche l'arbre de départ, en rouge le noeud dont on va incrémenter le poids et à droite l'arbre d'arrivée.

Pour trouver le noeud avec lequel on doit échanger le noeud que l'on souhaite incrémenter, il est nécessaire de disposer d'une liste de tous les noeuds triés par ordre croissant de poids et de maintenir cette liste à jour en même temps que la structure de l'arbre. Il n'est pas toujours possible de faire un échange. Soit un noeud de poids p, l'échange avec un autre noeud est impossible si :

- ce noeud est le même (il n'existe qu'un seul noeud de poids p)
- ce noeud est le père de celui que l'on souhaite incrémenter.

Comme nous l'avons vu précédemment pour l'algorithme de Huffman, si il est nécessaire de parcourir un grande partie de la liste avant de trouver le noeud avec lequel nous devons faire un échange, cela devient coûteux en temps. La solution qui permet de trouver le noeud de même poids le plus éloigné dans la liste consiste à regrouper tous les noeuds de même poids. Pour cet ensemble on désigne alors un « *leader* » qui est le noeud avec lequel on doit faire un échange en cas d'incrément d'un noeud de même taille. Lorsque l'on incrémente un noeud de l'arbre, on fait d'abord l'échange avec son leader, puis on peut augmenter son poids. Le leader des noeuds de poids maintenant inférieurs au poids du noeud que l'on vient d'incrémenter devient le précédent de ce dernier.

c) Avantages et inconvénients des différents algorithmes :

L'algorithme de Huffman est celui qui fournit les meilleurs codes pour un ensemble de mot. Soit S le nombre de bit total écrit pour un texte dont on a remplacé les mots par les codes obtenu à partir de l'arbre construit avec l'algorithme de Huffman. J. Vitter a montré que l'algorithme FGK écrit entre $S - n + 1$ et $2S + t - 4n + 2$ avec t le nombre total de mots écrits et n le nombre de mots différents. Et pour l'algorithme V, on écrit entre $S - n + 1$ et $S + t - 2n + 1$. Le temps nécessaire à la mise à jour de l'arbre créé par l'algorithme FGK est $O(l\ /2)$ avec l la longueur du chemin entre la racine et la feuille que l'on doit mettre à jour. Pour l'algorithme V, ce temps est $O(l\ )$.

L'avantage des codage dynamiques est surtout la possibilité de compresser un fichier en une seule passe. L'algorithme V est particulièrement intéressant car un fournit des codes proches des codes optimaux fournit par l'algorithme d'Huffman.

La résistance de l'algorithme de Huffman aux erreurs dans le fichier compressé est beaucoup plus grande que celle des algorithme dynamique. La raison est simple, l'arbre généré par l'algorithme de Huffman est toujours le même durant la compression d'un fichier. Une erreur sur un bit n'a donc aucune incidence sur la suite du codage du texte. Ce n'est pas le cas avec les codages dynamique. L'erreur sur un bit provoque une construction d'arbre différent et donc le décompresseur sera incapable de reconstruire la suite du texte de manière identique au fichier de départ.

2.Algorithmes et structures :

Nous allons maintenant détailler l'implémentation de notre programme. Comme nous l'avons dit plus haut, notre programme utilise l'algorithme V. Cette algorithme nécessite de maintenir deux types de structures pendant l'exécution de notre programme; une structure d'arbre ainsi qu'une structure de liste.

2.1.Structures de l'arbre :

Voici la structure de noeud utilisé dans notre programme :

```
typedef struct _HNode_t HNode_t;
struct _HNode_t
{
	HNode_t *parent;
	HNode_t *left;
	HNode_t *right;

	HNode_t *prev;
	HNode_t *next;
	HBlock_t *block;

	void *data;
};
```

Les pointeurs *parent*, *left* et *right* pointent respectivement sur le parent, le fils gauche et le fils droit du noeud courant. Les pointeurs *prev*, *next* maintiennent la structure de liste entre les noeuds. Le pointeur *block* désigne pointe sur une structure qui contient le leader du noeud ainsi que le poids de l'ensemble de ces noeuds. Enfin *data* pointe sur la donnée que l'on souhaite accrocher à ce noeud.

```
typedef struct _HBlock_t Hblock_t;
struct _HBlock_t {
	unsigned int count;

	unsigned int weight;

	HNode_t *leader;
};
```

Ci dessus la structure de block utilisée. Elle comporte un pointeur sur le *leader* des noeuds de poids *weight.* L'entier *count* permet de savoir combien de noeuds pointent vers cette structure. Cela est utile pour savoir quand on peut libérer cette structure. A chaque fois qu'un nouveau noeud pointe vers cette structure on incrémente *count*. Lorsque un noeud est incrémenté et qu'il va pointer sur le block de poids supérieur, on décrémente *count*. Lorsqu'aucun noeud ne pointe vers un block, la valeur de *count* est à zéro, on peut donc libérer cette structure. Cette technique s'appelle le compte de référence.

Enfin voilà notre structure d'arbre.

```
typedef struct _HTree_t Htree_t;
struct _HTree_t
{
	HNode_t *head;
```

```
    HNode_t *null;
    HNode_t *new;

    int width;
};
```

L'arbre contient trois pointeurs sur des noeuds dont il faut se souvenir. La racine de l'arbre *(head),* la feuille de poids le plus faible (*null)* et la feuille indiquant un nouveau mot (*new*). Enfin *width* représente le nombre de feuilles.

Les deux principales opérations que l'on peut effectuer sur un arbre sont l'ajout de feuilles et l'incrément du poids des feuilles. Les deux fonctions associées à ces opérations sont respectivement :

HNode_t *htree_add (HTree_t *tree, void *data);

qui prend en premier argument un pointeur sur un arbre, en second argument un pointeur sur la donnée à associer à la nouvelle feuille et qui retourne cette feuille.

void htree_increment (HTree_t *tree, HNode_t *node);

qui prend en argument un pointeur sur un arbre ainsi qu'un pointeur sur le noeud à incrémeter.

2.2. Méthode de compression :

On considère ici un mot comme une suite de caractères alphabétiques. Tout ce qui n'est pas une suite de caractères alphabétiques est donc considéré comme un séparateur. Un texte est donc constitué d'une alternance de mots et de séparateurs.

Le procédé de compression se déroule de la manière suivante; on commence par lire un mot, caractère par caractère. Lorsque un caractère non alphabétique est rencontré, on regarde si le mot lu est déjà apparu dans le texte. Cette recherche s'effectue dans un lexique. Le lexique est une structure de donnée qui nous permet de stocker un ensemble de mots et de vérifier l'appartenance d'un autre mot à cet ensemble rapidement grâce à une table de hachage. Si le nom est déjà apparu, on écrit son code donné à partir de l'arbre de Huffman et on incrémente le poids de la feuille de ce mot. Si ce mot n'a pas encore été rencontré, il faut écrire le code de la feuille indiquant un nouveau mot et ajouter une feuille à l'arbre. On doit ensuite écrire ce mot.

Le gain apporté par la seule compression des mots est insuffisant, il est donc nécessaire de compresser aussi les lettres des mots. On écrit ainsi les mots en remplaçant leurs lettres par leurs codes de Huffman obtenu à partir d'un arbre de Huffman dédié aux lettres. On applique la même méthode que celle que l'on utilise pour les mots, excepté que l'on doit écrire un nombre de variable de lettres, il est donc nécessaire au décompresseur de savoir combien de lettres il doit lire. Comme notre programme supporte la compression de fichiers binaires, il est impossible d'utiliser un caractère comme '\0' pour signaler la fin d'un mot. Une solution évoqué dans le sujet de ce devoir était d'écrire sur un nombre variable de bits le nombre de caractères avant d'écrire ces dernier. Nous utilisons ici une autre solution. On ajoute à l'arbre contenant les lettres une feuille supplémentaire qui représente un caractère de fin de mot, mais qui ne correspond à aucun des 256 caractères représentables sur une variable de type char. Le gain que nous avons remarqué sur un fichier texte de 1,5Mo est de l'ordre de 100Ko sur la taille totale du fichier compressé. Nous supposons qu'un mot est de longueur inférieure à 64 caractères. Si jamais nous avions à lire un mot de plus de 64 caractères, il sera alors séparé par des séparateurs de longueur nulle, il en va de même pour les séparateurs qui peuvent être découpé par des mots de longueur nulle.

3. Compilation et utilisation du programme hpress :

3.1.Compilation du programme :

Les sources de notre programme sont fournit dans un paquet contenant des script d'installation et de configuration. Voici les étapes à suivre pour compiler le programme:

- lancer le script de configuration des sources : ./configure
- puis lancer la compilation : make
- on peut ensuite installer le programme par : make install

Le répertoire d'installation par défaut est /usr/local/bin pour installer le programme dans /usr/bin par exemple, il faut le préciser au script de configuration comme ceci : ./configure –prefix=/usr

3.2.Utilisation du programme :

Le programme hpress lancé sans option provoque la compression des fichiers placé sur la ligne de commande. L'option -d provoque la décompression des fichiers placés sur la ligne de commande. L'option -c force le programme a utiliser la sortie standard au lieu d'écrire dans un fichier. Enfin l'option -h affiche l'aide du programme.

Notre programme tente de reproduire le comportement de base de la commande gzip.

La commande *hpress foo* a pour effet de compresser le fichier foo et place le résultat dans le fichier foo.H .

La commande *hpress foo.H* a pour effet de décompresser le fichier foo.H et place le résultat dans le fichier foo.H.D .

Il est possible de concaténer deux fichiers compressés : *hpress -c file1 > foo.H* puis *hpress -c file2 >> foo.H* . La décompression du fichier foo.H produira le même effet que la commande *cat file1 file2*.

Remarque : La commande *cat file1 file2 | hpress > foo.H* compresse mieux que la commande *hpress -c file1 file2 > foo.H* .

Il faut savoir que le programme hpress détermine le fait qu'un caractère est un alphabétique ou non à l'aide de la routine *isalpha*. Il est donc possible d'adapter la compression d'un texte au langage dans lequelle il se trouve et ainsi obtenir un meilleur taux de compression. Cela se fait en remplissant les valeurs des variables d'environnement LANG et/ou LC_COLLATE (man locale pour plus d'informations). Quelque soit la langage utilisé pour la compression d'un texte, le décompresseur sera capable de reproduire le fichier de départ à l'identique car il n'accorde par d'importance aux caractères qu'il gère, mais seulement aux codes de Huffman. Notre programme ne gère que les caractères encodés sur 8 bits, mais il peut être modifié facilement de façon à supporter les caractères multi-octets tels ceux utilisés dans les encodage comme Unicode.

3.3.Répartition des sources dans les différents fichiers :

Le répertoire *src* du paquet contient toutes les sources du programme hpress. Voici le détail de fichiers sources :

- alpha.[ch] : gestion du structure de donnée abstraite d'alphabet permettant la compression des lettres.
- bfile.[ch] : ensemble de fonction, englobant les appels *fread* et *fwrite,* reproduisant une partie de fonctions de la libc tel que *fopen*, *fclose*, *fputs*, *fgets*, etc...
- comp.[ch] : compresseur et décompresseur
- data.h : définissions de types génériques (utilisés par les listes et les tables de hachage)
- hash_close.[ch] : traitement des collisions en mode fermé (linéaire, quadratique et double)
- hash_func.[ch] : fonctions de hachage
- hash_open.[ch] : traitement des collisions en mode ouvert

- hash_table.[ch] : allocation/libération/initialisation des tables de hachage ainsi que des fonctions génériques d'ajout et de recherche
- lex.[ch] : gestion d'une structure de donnée abstraite (le lexique) permettant la compression des mots.
- huff.[ch] : gestion dynamique des arbres de Huffman
- main.c : initialisation du programme et gestions des arguments à la ligne de commande
- main.h : définition des informations configurant la compilation du programme
- palloc.[ch] : allocateur dynamique de mémoire pour des structures redondantes
- word.[ch] : gestion des structures de mot

4.Performances du programme hpress :

4.1.Taux de compression :

Voici les résultats obtenus à partir de différents fichiers :

fichier	taille originale	taille compressée avec gzip	taille compressée avec hpress	taille compressée avec bzip2
Confession.txt	1 497 993 octets	548 055 octets (36,5 %)	449 195 octets (29,9 %)	389 256 octets (25,9 %)
ColonelChabert	141 716 octets	54 173 octets (38,2 %)	51 684 octets (36,4 %)	44 266 octets (31,2 %)
comp.c	5 238 octets	1 598 octets (30,5 %)	1 869 octets (35,6 %)	1 671 octets (31,9 %)
hpress	35 810 octets	14 116 octets (39,4 %)	20 075 octets (56,0 %)	14 114 octets (39,4 %)

Les résultats que nous obtenons ne sont pas surprenants, ils montrent bien le bon fonctionnement des algorithmes basés sur les arbres de Huffman (hpress et bzip2) sur des données de type texte (Confession.txt et ColonelChabert). Mais on voit déjà le point faible de notre méthode basée sur les mots. En effet le fichier comp.c contient peu de mots, il s'agit pourtant d'un fichier texte. Alors que hpress permet de compresser mieux que gzip sur le Confession.txt et ColonelChabert, il fonctionne moins bien sur de fichiers texte de type langage de programmation. Enfin comme notre programme supporte la compression de fichiers binaires, nous avons tenté de compresser notre propre programme. Ce dernier test montre bien que la méthode de notre programme est particulièrement adaptée aux textes et non à d'autres types de données.

4.2.Temps d'éxecution :

Notre test s'effectue ici sur un seul fichier de 12Mo construit à partir de la commande suivante :

find /usr/share/man/fr -name '*.gz' -exec gunzip -c {} >> /tmp/tmp \;*

gzip	hpress	bzip2
1.045s	7.951s	9.917s

(Temps d'exécution obtenus sur un processeurs AMD Athlon 2000MHz)

Notre programme s'est révélé assez lent en comparaison d'autres programmes tels que ceux

utilisant un algorithme de Huffman. Nous savions que les algorithmes de construction d'un arbre dynamique entraînaient des temps de gestions de l'arbre importants. Pour nous assurez de cela et pour savoir dans quelles proportions cela affectait notre programme, nous avons compilé notre programme avec l'option -pg et utilisé l'utilitaire gprof. Les résultats sont les suivants; trois fonctions monopolisent à elles seule près de 60% du tenps total nécessaire à l'exécution du programme. Ces trois fonctions servent à maintenir l'arbre de Huffman à jour lors de l'incrément du poids d'une feuille.

5.Références :

Durant la réalisation de ce devoir il nous a été utile de lire un certain nombre d'articles et autres documentation afin de mieux comprendre les méthodes et les algorithmes de compression basés sur la construction d'arbres de Huffman. Voici la liste des documents que nous avons consulté :

- D.A Huffman, « *A method for the construction of minimum redundancy codes* », (1951)
- D.E. Knuth, « *Dynamic Huffman Coding* », (1985)
- J.S. Vitter, « *Design and Analysis of Dynamic Huffman Codes* », (Octobre 1987)
- J.S. Vitter, « *Dynamic Huffman Coding* », (1989)