Algav – Projet Huffman Dynamique

MI015 – Devoir de programmation

Année scolaire 2012/2013

Barbier Clément (3061254)

I. Modification de l'arbre :

Question 0 :

Propriété 1. Soit H un AHA avec n feuilles et $n-1$ nœuds internes ; dans la numérotation hiérarchique GDBH *GaucheDroiteBasHaut* $x_1, x_2, \ldots, x_{2n-1}$ des nœuds, on a :

(1) $W(x_1) \leq W(x_2) \leq \ldots \leq W(x_{2n-1})$, où $W(x_i)$ est le poids du nœud x_i.

(2) Pour $1 \leq i \leq n-1$, les nœuds x_{2i-1} et x_{2i} sont frères (i.e. ont le même père dans l'arbre).

Démonstration :

D'après l'énoncé, on sait que l'arbre H est un arbre de Huffman adaptatif. Selon la définition du cours un arbre de Huffman adaptatif (AHA) est un arbre de Huffman tel que le parcours hiérarchique GDBH (x_1, ..., x_{2n-1}) donne la suite des poids en ordre croissant $W(x_1) \leq ... \leq W(x_{2n-1})$. Nous avons donc bien l'arbre H avec $W(x_1) \leq ... \leq W(x_{2n-1})$.
L'arbre de Huffman adaptatif étant un arbre binaire complet, on sait qu'un nœud fils gauche a toujours son frère droit si son père est un nœud interne car le nœud père (obligatoirement interne) a obligatoirement deux fils. Selon la définition des arbres binaires on sait également que le poids du fils gauche est inférieur ou égal au poids du fils droit. Dans un parcourt GDBH de l'arbre on a à droite un fils d'indice paire et à gauche un fils d'indice impair ce qui explique que les nœuds x_{2i-1} et x_{2i} sont frères pour $1 \leq i \leq n-1$.

Propriété 2. Etant donné un AHA et une feuille f, de numéro x_{i_0}, dont le chemin à la racine est $\Gamma_f = x_{i_0}, x_{i_1}, \ldots, x_{i_k}$ $(i_k = 2n-1)$, alors l'arbre résultant de l'incrémentation de $W(f)$ sera encore un AHA ssi $W(x_{i_j}) < W(x_{i_j+1})$, pour $0 \leq j \leq k-1$. On dira dans ce cas que tous les nœuds du chemin Γ_f sont *incrémentables*.

Démonstration :

Selon la propriété 1, on a $W(x_1) \leq ... \leq W(x_{2n-1})$ où $W(x_i)$ est le poids du nœud x_i dans la numérotation GDBH des nœuds x_1, x_2, …, x_{2n-1}. Si l'on numérote les nœuds du chemin tel que x_0 est le numéro d'une feuille dont le chemin à la racine est x_{i0}, x_{i1}, …, x_{ik} avec ik = 2n-1 on aura $W(x_{ij}) <= W(x_{ij+1})$ pour $0 \leq j \leq k-1$.
Si l'on incrémente le poids d'une feuille, on sait que la différence entre $W(x_{ij})$ et $W(x_{ij+1})$ sera au moins de 1 ce qui induit que $W(x_{ij})$ est strictement inférieur à $W(x_{ij+1})$.

Question 1 :

Montrer que l'algorithme n'échange jamais un nœud avec un de ses ancêtres.

Dans arbre Huffman adaptatif chaque nœud possède 0 ou 2 fils, comme dans un arbre binaire complet (entier) peu importe le nombre n de symbole qu'il contient. Ceci peut être démontré par récurrence avec comme initialisation un arbre ne possédant qu'une seule feuille, la feuille spéciale notée « # ». Et étant donné que l'ajout d'un élément consiste à remplacer la feuille spéciale « # » par un sous-arbre complet, l'arbre Huffman résultant de l'ajout restera complet.

Soit f une feuille quelconque de l'arbre (différente de la racine). Etant donné que l'arbre Huffman est complet, on a alors W(père(f)) = W(f) si et seulement si W(frere(f)) = 0 car pour tout sommet x de l'arbre on a W(x) = W(FilsGauche(x))+W(FilsDroit(x)). Selon l'énoncé on sait que W(f) = 0 pour f = # (feuille spéciale), or selon ce qui précède on a W(père(f)) = W(f) si et seulement si f = frere(#).

Si on regarde l'algorithme de modification on peut lire :
Si Q est frere de # et pere(Q)=finBloc(H,Q),
Alors Q=pere(Q) Fin Si

Or cela éloigne notre condition qui dit qu'un nœud ne peut être échangé avec son père que si celui-ci est le frère de la feuille spéciale #.

L'algorithme ne peut donc jamais échanger un nœud avec un de ses ancêtres.

Question 2 :
Quel est le nombre maximal d'échanges réalisés lors d'une modification de l'arbre ?

Dans le pire des cas, si aucun nœud du chemin allant d'une feuille à la racine n'est incrémentable dans un AHA, on réalisera au maximum h-1 échanges avec h étant la hauteur de l'arbre. Or nous venons de démontrer que l'algorithme de modification n'échange jamais un nœud avec ses ancêtres.

Dans le pire cas, on pourrait avoir un arbre dont la racine a pour fils gauche la feuille à échanger et pour fils droit le sous-arbre regroupant toutes les autres feuilles, soit n-1 feuilles avec n le nombre de symboles dans l'arbre.

On a donc un nombre maximal d'échanges réalisés lors d'une modification dans l'arbre égal à n-1.

II. Compression et Décompression :

Question 1 :
Quelle est la complexité de l'algorithme de Compression en temps et en place ?

Notons :
- h : hauteur de l'arbre.
- t : taille du texte.
- n : nombre de symboles de l'alphabet.

Complexité en temps :

Lors de la compression on commence par rechercher le symbole s dans l'arbre qui s'effectue en O(h).
S'il existe on recherche et on transmet le codage du symbole en O(h), on incrémente le nombre d'occurrence du symbole en O(1).
S'il n'existe pas on recherche la feuille spéciale # en O(h) puis on doit l'ajouter à l'arbre le nouveau symbole s à l'arbre en O(1).
Dans les deux cas, si l'arbre ne respecte plus les propriétés des AHA, on effectue les modifications nécessaires sur l'arbre en O(n).
Tout ce qui précède est réalisé pour chacun des caractères du texte, on a donc une complexité finale en O(nt).

Complexité en espace :

Si l'on doit stocker l'arbre, la complexité en espace de l'algorithme sera en O(n).

Question 2 :

Abracadabra !!!

Etapes	Opérations	Valeur #	Transmissions
2	A(00000)		00000
3	B(00001)	0	000001
4	R(10001)	00	0010001
6	Incrémentation de A => 0	100	0
7	C(00010)	100	10000010
9	Incrémentation de A => 0	1100	0
10	D(00011)	1100	110000011
12	Incrémentation de A => 0	1000	0
13	Incrémentation de B => 110	1000	110
15	Incrémentation de R => 110	1000	110
16	Incrémentation de A => 0	1000	0

Question 3 :

Algorithme de décompression

```
Procédure Decompression(T : Texte)
        var H : Huffman, tmpH : Huffman, s : Symbole, b : bit
        var D : Texte          // Texte décompressé

        H = feuilleSpéciale#
        s = 5premiersBitsDe(T)                    // 8bits si on travaille en ASCII.
        D += s
        H = AjoutSymbole(H, s)
        tmpH = H

        Tantque T pas terminé
                Tantque !estFeuille(tmpH)
                        Si b == 0 alors
                                tmpH = filsGauche(tmpH)
                        Sinon
                                tmpH = filsDroit(tmpH)
                        FinSi
                        b = bitSuivant(T)
                FinTantque
                Si tmpH != # alors
                        D += symbole(tmpH)
                        H = Modification(H, symbole(tmpH))
                Sinon
                        s = 5BitsSuivantDe(T)          // 8bits si on travaille en ASCII.
                        D += s
                        H = AjoutSymbole(H, s)
                FinSi
                tmpH = H
        FinTantque
        retourner D
FinProcédure
```

III. Implantation :

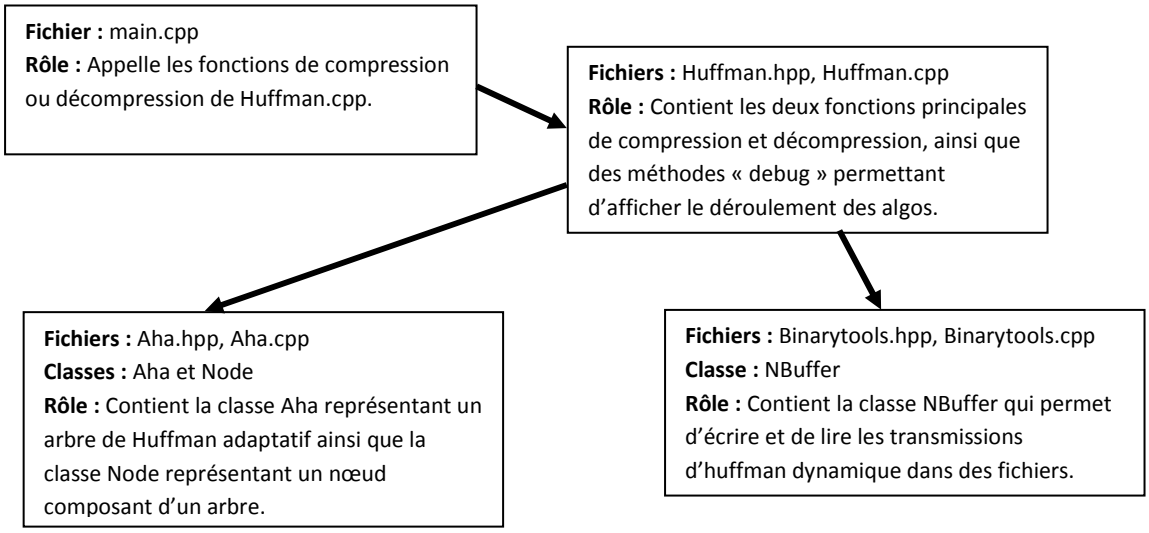
Question 1 :

Les choix d'implantation

1) Choix du langage :

Le choix du langage s'est fait pour le C++ pour plusieurs raisons :
En premier lieu pour la rapidité de calcul (C ou C++), en second lieu pour mon aisance dans ce langage m'évitant du temps à chercher comment faire certaines choses. Il n'est pas nécessaire de créer une interface élaborée pour le programme, la ligne de commande suffisant parfaitement. Même si le java apporte une plus grande (plus facile) portabilité, le choix d'un langage aussi gourmand ne me semble pas justifié pour la résolution d'algorithme. C'est également le critère de portabilité qui m'a fait évincé Python qui peut nécessiter l'installation de module complémentaire.
Enfin j'ai préféré le C++ au C, qui aurait été tout aussi performant, pour la programmation objet facilitant la compréhension globale du code, il est, à mes yeux, plus facile de se représenter, notamment en ce qui concerne la représentation de l'arbre et des nœuds qui le compose.

2) Structure globale :



3) Descriptions et optimisations :

Main :

Description : Point d'entrée du programme, prenant en paramètre une option (-c pour la compression, -d pour la décompression) ainsi que le nom d'un fichier d'entrée et le nom du fichier de sortie. Le main demande la compression ou décompression, affiche le temps d'exécution des fonctions, les tailles des fichiers avant et après l'opération et dans le cas d'une compression on affiche le taux de compression.

Huffman :

Description : Contient la méthode de compression et de décompression selon l'algorithme de huffman dynamique, prenant toutes les deux en entrée 2 fichiers. Ces deux fonctions font appel aux classes Aha, Node et Nbuffer, pour construire l'arbre au fur et à mesure de l'éxécution du code, lire et/ou écrire dans les fichiers les transmissions émises par ces méthodes.
Note : Dans une idée de programmation pleinement orientée objet, on aurait pu créer une interface CompressionDecompression possédant les deux méthodes utiles et implémentée par différentes stratégies (algorithme de compression et décompression différents) telles que HuffmanClassique, HuffmanSemiAdaptatif, HuffmanDynamique et pourquoi pas LZW par exemple aussi. Mais ce n'est pas le sujet du projet, c'est pourquoi je me suis contenté de deux méthodes.

Aha et Node :

Description : Aha et Node sont les deux classes nécessaires à la représentation d'un arbre huffman (binaire) en mémoire.

ArbreHuffmanAdaptatif :

La classe Aha contient donc un nœud racine qui correspond à la racine de l'arbre binaire. Pour réduire le temps de recherche de la feuille spéciale (dont on a besoin à chaque insertion), la classe Aha possède également un pointeur sur cette feuille spéciale mis à jour par les fonctions modifiant l'architecture de l'arbre, la complexité de la recherche de la feuille spéciale est en O(1). La classe Aha possède également la liste des symboles déjà stockés dans son arbre, ce qui évite de parcourir tout l'arbre à chaque lecture d'un symbole : pour que l'utilisation d'une telle liste soit justifiée il faut que « rechercher si symbole est déjà dans l'arbre » soit en O(1), ce qui implique un accès indexé à la liste sur le symbole. J'ai hésité entre l'utilisation de map de la STL et un simple tableau de booléens de 256 cases, l'index d'une case correspondant à un symbole et le booléen à 'vrai' marquant la présence (on n'a pas besoin de l'occurrence des caractères dans cette liste contrairement à l'algorithme de Huffman Classique, et le booléen étant une des plus petites variables à allouer (1octect)). Mon choix s'est porté pour le tableau de booléen, plus gourmand en mémoire qu'une map (puisque même avec un seul caractère dans le texte on doit stocker 256octects) mais plus rapide d'accès.
Pour finir la classe possède évidemment toutes les primitives des Arbres Huffman vues en cours, plus celle du sujet de projet *finbloc*.

Nœud :

La classe Nœud représente tous les nœuds et feuilles de l'arbre. Elle possède deux valeurs la première étant un symbole (dans le cas d'une feuille), la seconde étant la fréquence (occurrence) d'apparition de ce symbole dans le texte. La classe Node contient également un booléen indiquant si le nœud est la feuille spéciale ou non (cf III. Qst1. 4 feuille spéciale). Par définition un nœud peut posséder deux fils (fils gauche et fils droit). Afin de faciliter les parcours et les recherches, un nœud possède également un pointeur sur son père ainsi que sur son nœud suivant et précédent dans le parcourt de l'arbre en fonction des poids (fréquences).

Binarytools et NBuffer :

Description : Contient toutes les méthodes utiles pour travailler en bit à bit. La classe NBuffer sert écrire et lire des suites de bits dans des fichiers. Pour écrire et lire dans un fichier cela se fait obligatoirement par champ de 1 octect (8bits). Lors de l'ajout d'un nouveau symbole nous transmettons (n+8)bits : n bits correspondant au chemin de la feuille spéciale dans l'arbre suivit des 8 bits codant le symbole (un char étant codé sur 1octect). Lors d'une incrémentation on transmet n bits correspondants au chemin dans l'arbre jusqu'au nœud contenant le symbole répété. Quand on lit un fichier ainsi compressé avec à la suite tous les bits des différentes transmissions, on récupèrera tout le contenu du fichier que l'on parcourra ainsi bit par bit. On sait que l'on lit toujours un chemin (feuille spéciale ou feuille d'un symbole). Il nous suffit donc de lire le chemin bit par bit et l'on parcourt notre arbre actuel depuis la racine, quand on arrive sur une feuille on regarde si celle-ci est la feuille dans quel cas on devra lire d'un coup les 8bits suivant pour connaitre le nouveau symbole, si ce n'est pas la feuille spéciale, c'est que l'on doit incrémenter la fréquence du symbole déjà dans notre arbre.
Le buffer initialise chaque groupe de 8 bits (octect) à 00000000 et le modifie en fonction des transmissions, si l'octect en cours est plein, on créé un nouvel octect et ainsi de suite.
A la fin de la compression il suffira de recopier tous ces octects dans le fichier compressé. Aucun traitement n'est effectué sur le premier octect car pour le premier symbole lu, le chemin de la feuille spéciale est vide, ainsi il suffit de directement recopier le caractère lu dans le fichier compressé, ce qui est valable également pour la décompression.

4) Difficultés, problèmes rencontrés :

Stockage binaire :

Le principal problème sur lequel j'ai le plus réfléchis est le format de stockage des données dans les fichiers compressés (classe NBuffer). La description ci-dessus de Binarytools expose la manière dont sont effectuées les lectures et les écritures dans les fichiers binaires. Or un problème se pose lors de la décompression dans le cas où le nombre de bits transmis n'est pas un multiple de huit. Prenons un exemple simple pour illustrer :

Notre texte a compressé est « ab ». Si l'on suit l'algorithme de Huffman dynamique nous aurons donc deux transmissions :

- 8 bits = 8 bits du symbole « a » (le chemin de # étant vide). On transmet : 01100001

-9 bits = 1 bit pour le chemin de # : 0 puis 8bits du symbole « b ». On transmet : 001100010

Voila le contenu complet du fichier :

Octet n°1	Octet n°2	Octet n°3
01100001	00110001	00000000

NB : Dans le fichier les bits sont écris en sens inverse mais cela n'est pas important pour notre explication.

Rouge : code du symbole a. Vert : chemin de #.

Orange : code du symbole b. Noir : bits de bourrage.

Si l'on exécute notre algorithme de décompression, il n'y a pas de soucis jusqu'à la fin de l'ajout de b, on a lu les 17bits encodés, mais comme il reste encore 7 bits et que l'on n'a pas de moyen de savoir qu'il faut s'arrêter après la lecture du 17eme bit. Notre décompresseur va comprendre que les sept 0 restants correspondent à des répétitions de symbole ou le chemin de la feuille spéciale (et cherchera donc à interpréter les 5 bits restant comme un nouveau caractère). C'est pourquoi il faut savoir combien de bits sont réellement significatifs…

Recherche de la solution :

Ma première idée était de stocker au début du fichier un entier (au minimum *unsigned short int* sur 2 octets de 0 à 65 535, au maximum *unsigned long int* sur 4 octets de 0 à 4 294 967 295) qui indiquerait le nombre de bits significatifs. L'avantage étant que si l'on a beaucoup de transmission on aura au maximum 32bits nous indiquant la taille. Mais le gros problème étant les très gros fichiers, si la transmission dépasse les quatre millions de bits notre solution n'est plus envisageable.

Ma seconde solution, elle est un peu plus couteuse en place (en bits) mais possède l'avantage de ne pas bloquer la taille maximum des transmissions. L'idée est d'insérer un bit à 1 avant chaque transmission (ajout ou insertion), comme les octets sont initialisés à 0, quand on lira un zéro lors d'une nouvelle lecture on saura que l'on a fini de décoder le fichier. Voici une illustration, on notera que normalement on n'applique pas cela au premier symbole lu mais cela n'est pas important pour notre démonstration. Avec notre nouvelle stratégie on émettra donc :

- 9bits = 1bit à 1 pour indiquer que c'est une transmission + 8 bits du symbole « a ». On transmet : 101100001

- 10 bits = 1bit à 1 de transmission + 1 bit pour le chemin de # puis 8bits de « b ». On transmet : 1001100010

Voici le nouveau contenu :

Octet n°1	Octet n°2	Octet n°3
10110000	11001100	01000000

Violet : Le bit indiquant que l'on doit continuer la lecture bit à bit.

On a donc ajouté deux bits (1 par transmission). A la fin de la lecture du b, quand on lira le 20ème bit égale à zéro on saura que l'on a finit de décompresser le fichier.

L'inconvénient de cette méthode est d'ajouter 1 bit à chaque transmission, si l'on a un code comprenant plus de 16 transmissions et dont la taille totale est de moins de 65 535 bits notre première solution serait plus avantageuse parce qu'elle n'ajoutera que 16bits mais comme nous ne pouvons faire du cas particulier la seconde solution est meilleure.

Après avoir utilisé la version de mon programme utilisant cette seconde solution je me sui rendu compte que finalement c'était beaucoup trop lourd d'ajouter un bit à chaque transmission, mon taux de compression moyen atteignait environ 38% sur l'ensemble des tests présentés après. En me repenchant sur le problème j'ai trouvé une troisième solution combinant les deux précédentes.

Si on reprend ma deuxième solution on comprend que ce qui nous intéresse réellement est de savoir quel est le dernier bit significatif dans le dernier octet. Si on reprend l'exemple « ab » compressé sur 3 octets et plus précisément 17bits, ce qui nous intéresse est le numéro du dernier bit significatif dans l'octet n°3, qui est 1 dans cet exemple. Mon idée était donc d'ajouter à la fin de l'encodage un dernier octet stockant ce numéro en caractère soit '1'. Ainsi lors de la décompression on extraira le dernier caractère du fichier soit '1' ici (afin qu'il ne soit pas décodé) et par un calcul simple on obtiendra alors

facilement le nombre total de bits significatifs en multipliant le nombre total de caractère dans le fichier moins 2 (il faut aussi enlever l'avant dernier octet car on est pas sur que tous ses bits soient significatifs) par 8 et en y ajoutant le nombre codé sur le dernier caractère : on aura ici nb_bit_sign = ((4-2)*8)+1 = 17bits.

Feuille spéciale :

En faisant tourner mon programme sur les fichiers de test fournis, j'ai détecté une erreur lors de la présence d'un caractère « # » dans le texte. Comme le symbole « # » associé à la feuille spéciale n'est là que pour l'affichage mais n'est sensé correspondre à aucun caractère du texte, j'ai du ajouter un booléen dans la classe Node pour indiquer qu'une feuille est une feuille spéciale ou non. La feuille spéciale gardera la valeur '\0' donnée par défaut aux nœuds vides (qui ne sont pas des feuilles) et qui ne peut être lu dans le texte car il correspond à la fin de fichier mais le booléen indiquera lors de l'affichage que c'est le caractère « # » qu'il faut afficher. Si l'on a des caractères « # » dans le texte, on aura donc, à l'affichage, deux feuilles avec la valeur « # » mais la feuille spéciale sera détectable facilement car sa fréquence sera égale à 0.

Question 2 :
Exemples d'exécution

1) Un peu de magie ?

Afin de vérifier le bon déroulement des algorithmes, une méthode affiche chaque transmission lors de la compression et décompression, une autre permet d'afficher l'arbre complet dans ce format :
Pour chaque nœud :

(valeur, fréquence) filsgauche = (valeur, fréquence) filsdroit = (valeur, fréquence)

Exécutons maintenant notre compression puis décompression sur « abracadabra » :

Compression :

```
Compression de in.txt vers out.txt
Transmission(Ajout) : 01100001 avec #= et a(01100001)
Transmission(Ajout) : 001100010 avec #=0 et b(01100010)
Transmission(Ajout) : 0001110010 avec #=00 et r(01110010)
Transmission(Increment) : 0 (a)
Transmission(Ajout) : 10001100011 avec #=100 et c(01100011)
Transmission(Increment) : 0 (a)
Transmission(Ajout) : 110001100100 avec #=1100 et d(01100100)
Transmission(Increment) : 0 (a)
Transmission(Increment) : 111 (b)
Transmission(Increment) : 110 (r)
Transmission(Increment) : 0 (a)
(   , 11 ) fg=( a , 5 ) fd=(   , 6 )
( a , 5 ) fg=None fd=None
(   , 6 ) fg=(   , 2 ) fd=(   , 4 )
(   , 2 ) fg=(   , 1 ) fd=( c , 1 )
(   , 1 ) fg=( # , 0 ) fd=( d , 1 )
( # , 0 ) fg=None fd=None
( d , 1 ) fg=None fd=None
( c , 1 ) fg=None fd=None
(   , 4 ) fg=( r , 2 ) fd=( b , 2 )
( r , 2 ) fg=None fd=None
( b , 2 ) fg=None fd=None
Taille de in.txt = 11
Taille de out.txt = 9
Taux de compression : 18.1818 %
```

L'arbre obtenu et les transmissions correspondent bien à ce que l'on attend (cf II. Q2).

Décompression :

La compression retrouve bien le texte initial et voici le listing d'exécution :

```
Decompression de out.txt vers decompress.txt
Transmission(Ajout) : 01100001 avec #= et a(01100001)
Transmission(Ajout) : 001100010 avec #=0 et b(01100010)
Transmission(Ajout) : 0001110010 avec #=00 et r(01110010)
Transmission(Increment) : 0 (a)
Transmission(Ajout) : 10001100011 avec #=100 et c(01100011)
Transmission(Increment) : 0 (a)
Transmission(Ajout) : 110001100100 avec #=1100 et d(01100100)
Transmission(Increment) : 0 (a)
Transmission(Increment) : 111 (b)
Transmission(Increment) : 110 (r)
Transmission(Increment) : 0 (a)
(   , 11 ) fg=( a , 5 ) fd=(   , 6 )
( a , 5 ) fg=None fd=None
(   , 6 ) fg=(   , 2 ) fd=(   , 4 )
(   , 2 ) fg=(   , 1 ) fd=( c , 1 )
(   , 1 ) fg=( # , 0 ) fd=( d , 1 )
( # , 0 ) fg=None fd=None
( d , 1 ) fg=None fd=None
( c , 1 ) fg=None fd=None
(   , 4 ) fg=( r , 2 ) fd=( b , 2 )
( r , 2 ) fg=None fd=None
( b , 2 ) fg=None fd=None
Taille de out.txt = 9
Taille de decompress.txt = 11
```

2) Jeux de tests :

Pour effectuer les tests de mon implémentation j'ai fait tourner l'algorithme de compression sur plusieurs fichiers fournit dans l'archive Data. On enregistre à chaque fois le nom du fichier, sa taille initiale, sa taille compressée, le taux de compression (Tcomp = 100 – (100 x (TailleCompressé / TailleInitiale))), la taille du fichier résultant de la décompression pour vérifier que l'on obtient bien le même fichier qu'à l'état initial, le nombre de caractère différents dans le texte initial, et le temps d'exécution (moyenne calculée sur 10 compressions du même fichier). On notera que le temps d'exécution sert uniquement de comparaison et non de référence car il est lié à la machine sur laquelle est exécuté le programme et que l'affichage des transmissions augmente plus ou moins ce temps d'exécution. Les tests ont été effectué sur un ordinateur portable Acer (I7-740QM 1.73Ghz 6Mb L3 cahe, 8Gb RAM, sous Ubuntu).

Fichier	Taille initiale	Taille compressé	Taux de compression	Taille décompressé	Caractères	Temps (en seconde)
"abracadabra"	11	9	18,18	11	5	0,001
"carambarbcm"	11	10	9,09	11	5	0,001
Data\Test1\test0	3 000	1 893	36,90	3 000	30	0,012
Data\Test1\test1	2 046	525	74,34	2 046	10	0,004
Data\Test1\test2	65 534	16 047	74,96	65 534	15	0,06
Data\Test1\test2rev	65 534	16 394	74,98	65 534	15	0,048
Data\Test1\test3	80 000	54 291	32,14	80 000	40	0,092
Data\Test1\test4	1 048 574	262 178	75,00	1 048 574	19	0,904
Data\Test1\upmc.eps	585 794	101 507	82,67	585 794	63	1,084
Data\Experimental Data\urns.m12.n4000.s100000	1 088 958	483 286	55,62	1 088 958	27	0,132
Data\Experimental Data\exp-rnd-33.txt	174 333	73 098	58,07	174 333	45	0,26
Data\Wikipedia\Wikipedia-20091130153946.xml	578 837	380 264	34,31	578 837	193	2,784
Data\Wikipedia\Wikipedia-20091130154119.xml	132 841	91 661	31,00	132 841	195	0,588
Data\Textual Data\phdthesis.ps	3 457 327	2 192 860	36,57	3 457 327	97	9,501
Data\Textual Data\eng-dict-1913.txt	8 025 699	4 613 720	42,51	8 025 699	87	21,149
Data\Textual Data\Bible\BDS\40026000	9 677	4 880	49,57	9 677	26	0,024
Data\Textual Data\Bible\ITA\40026000	8 318	4 176	49,80	8 318	22	0,016
Data\Textual Data\Bible\KJV\40026000	8 067	4 133	48,77	8 067	26	0,012
Data\Textual Data\Gutenberg\2505.txt	1 220 550	628 962	48,47	1 220 550	27	1,596
Data\Textual Data\Gutenberg-S3\2505-s3.txt	3 904 440	2 056 800	47,32	3 904 440	30	5,356

Si on calcul la moyenne des taux de compression on obtient 49.01 % de compression. On se rend compte que le taux de compression est vraiment mauvais pour les petits fichiers de quelques octets, mais on peut se dire que la compression d'un fichier de quelques octets est tout de même une opération très rare dans l'utilisation d'un système de compression sur un ordinateur. Donc juste à titre d'information j'ai recalculé cette moyenne en excluant les fichiers dont la taille était inférieure à 1ko (les deux premières lignes du tableau ci-dessus) et on trouve un taux de compression moyen de 52.94%.

Sur les exemples testés, tous ont retrouvé le fichier initial après compression puis décompression. Le temps d'exécution est presque parfaitement proportionnel à la taille du fichier à encoder. A taille de fichier égale le temps d'exécution diffère en fonction du nombre de caractère puis l'ordre d'apparition des caractères pour un nombre de caractère différents égal. Le taux de compression est évidemment lié au nombre de caractères différents à encoder à taille de fichier égale c'est le fichier contenant le moins de caractères différents qui aura le meilleur taux de compression, l'ajout d'un nouveau symbole étant beaucoup plus gourmand en place dans le fichier compressé (au minimum 8bits pour le tout premier caractère) qu'une répétition (au minimum 1 bit). Le taux de compression dépend également de l'ordre des caractères dans le texte, si un caractère est répété de nombreuses fois à la suite (exemple : aaaaa…aaaaabbbbb…bbbb…) le taux de compression est sensiblement meilleur qu'avec une répartition aléatoire des caractères dans le texte.