Roberto Medina
3360906

16 Décembre 2013

Algorithmique Avancée
Devoir de programmation :
Arbre de Huffman Adaptatif

Table des matières

Description du projet

Langage de programmation

Le projet a été développé en **C++** et compilé avec Clang. Ce langage me semble pertinent à cause des différentes structures de données qu'on peut créer et qu'on peut utilisé grâce à la librairie standard. C'est à dire que si j'avais besoin d'une structure de tableau dynamique je n'avais pas à recoder tout un module.

De même, je voulais bénéficier des pointeurs : la dynamique des pointeurs étant quelque chose d'assez simple, me permettait de faire des manipulations dans l'arbre assez facilement et avec un temps de traitement constant. Au lieu d'instancier des objets, j'alloue une structure de Nœud pour créer ou modifier mon arbre et je le fais que quand j'en ai besoin. C'est la même structure de Nœud qui est manipulée, je fais pas de recopie et destruction, ce qui serait très coûteux et difficile à gérer.

Finalement le fait de devoir manipuler des bits a été un autre facteur pour choisir ce langage de programmation. Cette manipulation a été faite par un module que j'ai créé. Vers la fin du projet j'ai trouvé une bibliothèque (BitMagic) qu'aurait pu être utilisé, mais le module était déjà presque terminé donc j'ai décidé de le conserver.

Le compilateur Clang me permettait d'avancer plus vite sur des erreurs grâce à l'affichage coloré. Normalement il y a pas de différences quand le projet est compilé avec GCC.

Modules

Le projet se décompose en trois modules:

Le **module Arbre** qui fait la gestion de l'Arbre de Huffman, on va trouver les algorithmes de Modification, de Traitement et d'échange pour les Noeuds. Le module possède les primitives sur les arbres binaires avec les traitements spécifiques pour Huffman. Les fichiers correspondants sont: `arbre.cpp` et `arbre.hpp`

Le **module Codage**, `codage.cpp` et `codage.hpp`, gère l'écriture avec des bits sur de octets.

Le **module Compression**, utilise les deux autres modules pour faire la compression et la décompression sur un fichier. Correspond aux fichiers `compression.cpp` et `compression.hpp`

Chaque module possède des accesseurs et des mutateurs pour leur structures correspondantes.

L'idée était de créer un autre module appellé Symbole qui aurait permis de faire un Arbre de Huffman générique contenant des Symboles sur les feuilles et pas de caractères.

Pour tester au fur et à mesure les fonctions des modules il y a des tests de regression qui ont été crées. On peut les compiler avec la directive du Makefile, **huffman-regression**.

Le programme principal peut être compilé avec la directive **huffman-dynamique** ou avec un simple make.

La syntaxe du programme est la suivante:

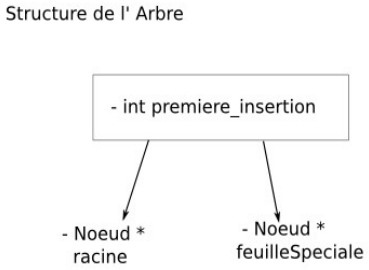
```
./huffman-dynamique -[cd] <fichier d'entrée> <fichier de sortie>
```

Structures de données

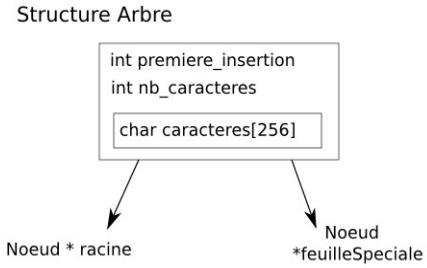
Comme expliqué au par avant, le choix du langage m'a permis de créer des structures de données plus ou moins complexes pour me permettre de gérer l'Arbre d'Huffman Adaptatif.

Entre la présentation du projet et l'édition de ce rapport il y a eu quelques changements dans les structures de données. Voici quelques schémas pour mieux comprendre comment fonctionnent ces structures :

La structure d'Arbre pour la présentation :



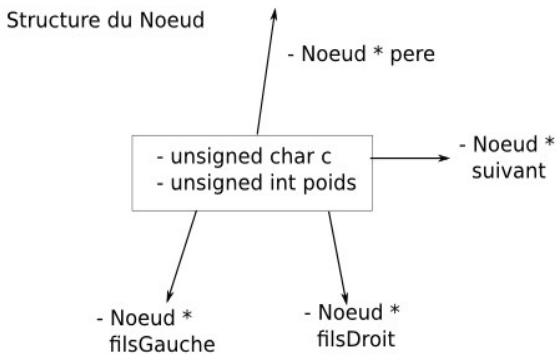
La structure d'Arbre après la présentation :



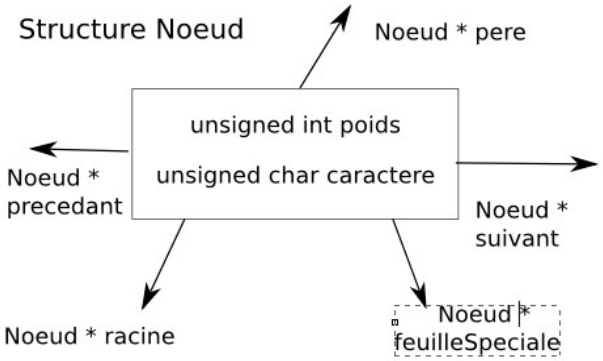
Cette nouvelle structure permet d'améliorer le test pour savoir si un caractère est présent dans l'arbre ou pas. La recherche se faisait avec deux appels récursifs, maintenant il faut juste parcourir le tableau de caractères pour savoir si le caractère est présent ou pas. Retrouver et renvoyer un Nœud ce fait toujours de manière récursive en regardant d'abord à gauche.

Les pointeurs vont sur la racine de l'arbre et sur la feuille spéciale. Le deuxième pointeur est nécessaire pour ne pas devoir faire une recherche récursive jusqu'à trouver la feuille spéciale.

La structure du Nœud pour la présentation :

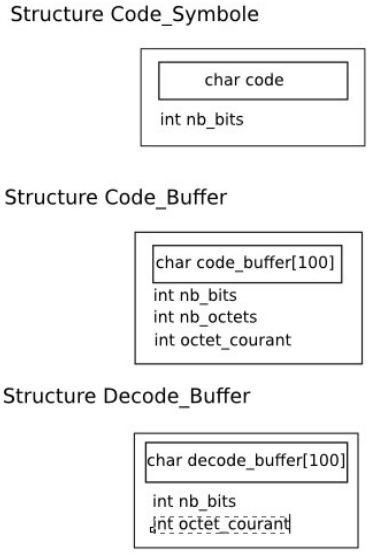


La structure du Nœud après la présentation :



Comparé à la version présentée on a un nouveau pointeur sur le prédécesseur du Nœud pour l'ordre GDBH. Ceci est utile quand on doit faire un échange avec beaucoup de décalage entre nœuds.

Pour le module qui fait le codage sur des bits d'autres structures de données ont été introduites :



La structure Code_Symbole possède un octet qui contient le code d'un symbole, et l'entier permet d'indiquer combien de bits sont réellement utilisés pour faire ce codage.

La structure Code_Buffer contient des octets où on trouve le codage du texte à l'étape *i*. À la fin de l'algorithme de compression on écrit les octets qui ont été réellement utilisés grâce à l'entier nb_octets. L'entier nb_bits est utile pour se repérer dans l'octet_courant.

Enfin, la structure Decode_buffer contient des octets avec le code compressé, on peut changer à la main le nombre d'octets pour les buffers. Cette chaîne est chargée au début de l'algorithme de décompression. On parcours la chaîne grâce aux deux entiers de la structure.

Des fonctions adaptées pour chaque structure sont dans le module.

Réponses aux questions

Modifications dans l'arbre

Question 1

L'algorithme de Traitement n'échange jamais un nœud avec un de ses ancêtres.

On la suite x_{i0}, x_{i1}, …, x_{ik} les feuilles allant jusqu'à la racine pour la feuille x_{i0}.

L'échange a lieu dès qu'on trouve une valeur dans la suite ayant la même valeur que son suivant, c'est à dire quand $W(x_{ij}) = W(x_{ij+1})$. L'échange se fait donc entre les sous arbres Q_m et Q_b où l'arbre Q_b est donné grâce à la fonction finBloc(H,m) et l'arbre Q_m est donne grâce au nœud x_{ij}, on a aussi $W(x_m) = W(x_b)$.

Or le même poids d'un nœud avec son père, dans l'ordre GDBH, peut seulement se donner que dans un cas : quand le frère de la feuille est la feuille spéciale. Donc le parcours de finBloc ne va jamais arriver sur le père du nœud qui est traité.

Ainsi l'algorithme n'échange jamais de nœud avec son père.

Question 2

Le nombre maximal d'échanges dans l'arbre avec un alphabet de taille n :

Le pire cas se produit quand pour chaque nœud allant vers la racine, calculé après Modification, on doit faire un échange. C'est à dire $2n-1$ fois d'échanges, car le chemin vers la racine est égal à $2n-1$.

Ainsi la complexité de cet algorithme est donc $O(n)$.

Compression et Décompression

Question 1

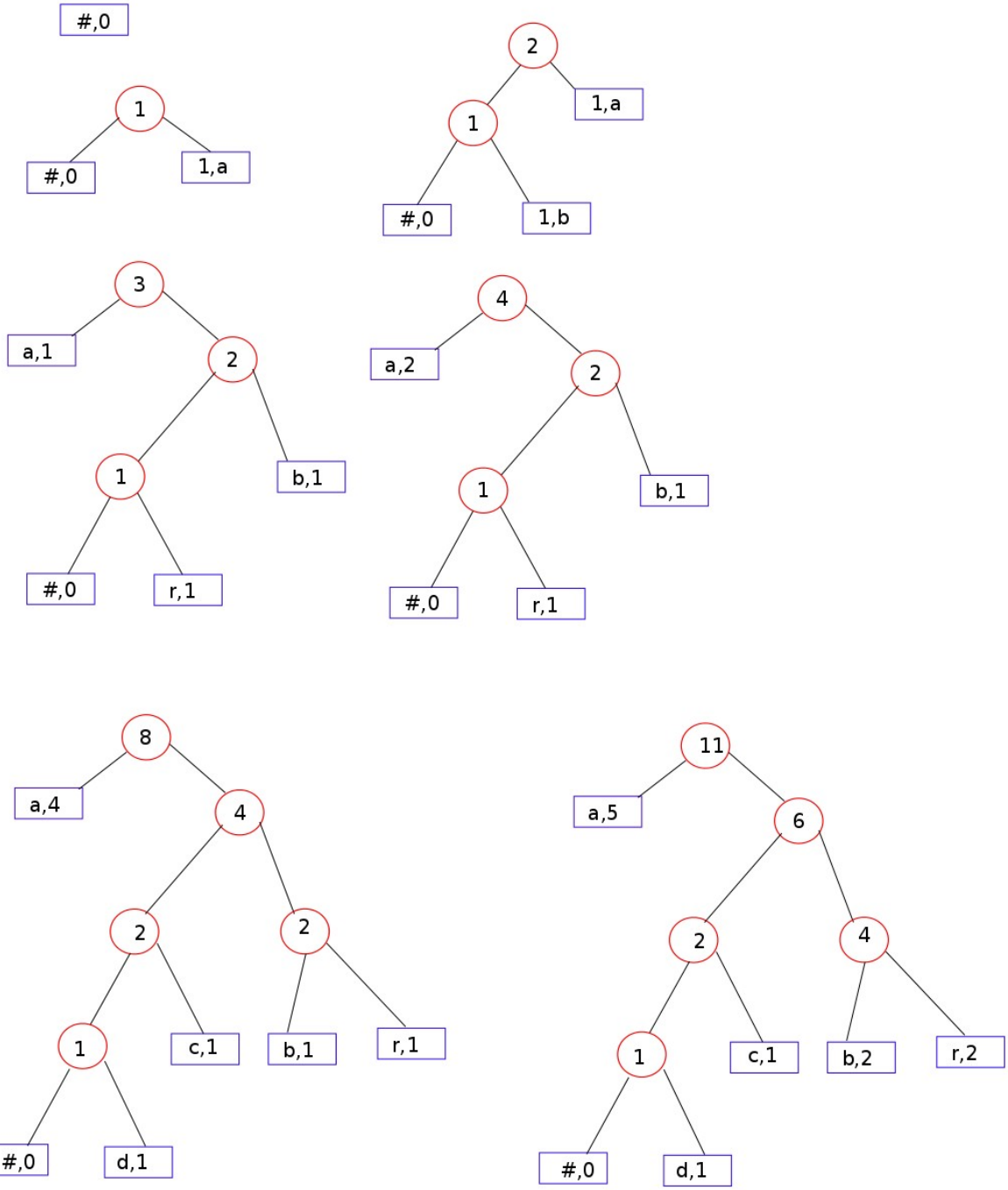
Soit N la taille du texte et n la taille de l'alphabet.

La complexité de l'algorithme de Compression est donnée par le nombre de fois qu'on passe par la boucle **Tant que**, c'est à dire N fois. Cette complexité dépend aussi de la transmission du code du symbole s dans H, comme on doit recherche une feuille, on a une complexité en $O(\log n)$ puisqu'il s'agit d'un arbre binaire digital.

Le dernier facteur qu'on doit prendre en compte c'est la Modification de l'arbre. L'algorithme a un coût de $O(n)$ à cause du Traitement (Question 2, Partie 2) puis que le reste des opérations ce font en temps constant.

Ainsi, en combinant les termes on obtient une complexité en $O(N*n*\log n)$ pour la Compression.

Question 2


#,0
1
#,0
1,a
2
1
1,a
#,0
1,b
3
a,1
2
1
b,1
#,0
r,1
4
a,2
2
1
b,1
#,0
r,1
8
a,4
4
2
2
1
c,1
b,1
r,1
#,0
d,1
11
a,5
6
2
4
1
c,1
b,2
r,2
#,0
d,1

Question 3

Procédure Décompression (*Tc*: Texte compressé)

var: *H*: Arbre d'Huffman Adaptatif, *s*: Symbole, *T*: Texte décompressé, *b*: entier, *N*: noeud

H = feuille spéciale #

décalage n bits (*Tc, 1*)

s ← lire n bits (*Tc, 8*)

H ← Modification (*H, s*)

N ← racine (*H*)

Répeter

Tant que !estFeuille(*N*) faire

b = prochain bit de *Tc*

si *b* = 0 **alors**

N ← filsGauche (*N*)

sinon

N ← filsDroit (*N*)

fin si

fin tant que

si *N*→caractère = '#' **alors**

s ← lire n bits (*Tc, 8*)

sinon

s ← *N*→caractère

fin si

T ← *s*

H ← Modification (*H, s*)

N ← racine (*H*)

Jusqu'à *Tc* terminé

Retourner *T*

Fin procédure Décompression

En suivant cet algorithme on devrait tomber sur le texte décompressé. La seule différence est que sur le sujet on ne met pas le code de la feuille spéciale pour le premier symbole. Et dans le code du projet je lis 8 bits au lieu de 5.

En déroulant l'algorithme on va lire 0 d'abord puis 5 bits qui correspondent à la lettre 'a'. Puis je rajoute le symbole à mon arbre de Huffman.

Ensuite je vais lire un 0 et aller sur le fils gauche de mon arbre de Huffman. Il s'agit bien d'une feuille, la feuille spéciale, donc je vais lire 5 bits et mettre le symbole que j'ai trouvé sur mon arbre avec Modification. Pour le caractère 'r' ça va être pareil.

Quand je vais retrouver le caractère 'a' à nouveau je vais lire un 0 et aller vers le fils gauche de la racine mais cette fois la feuille spéciale sera dans le fils droit, donc je vais insérer le caractère de la feuille dans le texte décompressé.

On continue ainsi pour le reste du texte compressé.

Implantation

Question 2

Limites de l'implantation du projet

Malheureusement depuis la présentation du projet les erreurs pour la gestion de l'arbre et pour la décompression sont toujours présents.

Quand je rajoute un caractère plusieurs fois sur l'arbre et après je rajoute n'importe quel autre caractère, les pointeurs font qu'il y ait une boucle infinie quand j'essaye de remonter jusqu'à la racine pour l'algorithme de traitement. Ceci n'arrive pas tout le temps par contre, des fois quand le caractère est bien profond dans l'arbre

Pour la décompression il y a une erreur sur les derniers caractères que je n'ai pas réussi à résoudre non plus.

Je pense que ces erreurs auraient pu être résolues si j'aurais fait ce projet avec quelqu'un d'autre.

Tests sur quelques jeux d'essais bien choisis pour l'implantation

Texte	Texte Compressé	Taux de Compression
abracadabra – 11 octets	7 octets	36.4%
Carambar – 8 octets	7 octets	12.5%
Bonjour1000 – 4999 octets	1701 octets	66%

Conclusions sur le projet

D'après les résultats sur le jeux d'essais je peut conclure que même si l'implantation ne marche pas au 100 % on peut voir un gain assez significatif sur la compression d'un texte.

Je pense qu'avec un peu plus de temps et en travaillant avec un binôme ça aurait été plus simple de trouver les bugs qui sont présent maintenant dans le code. De même, le fait d'avoir codé une librairie qui gère les bits a été un peu pénalisant car ça m'a pris plus de temps que prévu puisqu'il fallait penser à aux structure de données utiliser et ensuite arriver à écrire dans le bon ordre les bits qui sont données.