Compte-Rendu
Arbre de codage de Huffman

Kévin Vythelingum

19 mai 2013

1 Introduction

Pour coder une chaîne de caractères, on peut choisir de représenter chaque caractère par un nombre de bits fixé. Ainsi, pour coder N caractères, on aurait besoin de $log_2(N)$ bits.

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

Cependant, cela n'est pas optimal dans la mesure où certains caractères sont très rarement utilisés alors que d'autres sont très fréquents. On peut alors choisir de réduire la taille d'un texte total en allouant moins de bits aux caractères fréquents, et plus aux caractères rarement utilisés. C'est l'idée de l'arbre de codage de Huffman.

Le principe de l'arbre de Huffman est le suivant :

On attribue à chaque caractère un poids lié à sa fréquence d'apparition dans un texte. Par exemple, le caractère A, que l'on retrouve souvent, sera affecté d'un poids de 8 alors que le caractère H, plus rare, aura un poids de 1. Ensuite, on représente l'alphabet ainsi pondéré par un arbre en veillant à ce que les caractères les plus "lourds" soient les plus proches de la racine. Cet arbre permet alors de donner le code de chaque caractère en ajoutant 1 si on descend à gauche et 0 si on descend à droite. L'identification des caractères est possible grâce au fait qu'aucun code de complet n'est le préfixe d'un autre caractère.

A 1	C 0110	E 0100	G 0010
B 000	D 0111	F 0101	H 0011

L'objectif de ce TP sera de construire un arbre de codage à partir d'un alphabet pondéré, d'afficher ensuite le code associé à chaque lettre, et enfin de décoder une chaine de bits constituant un message.

2 Analyse du problème

2.1 Construire un arbre de codage

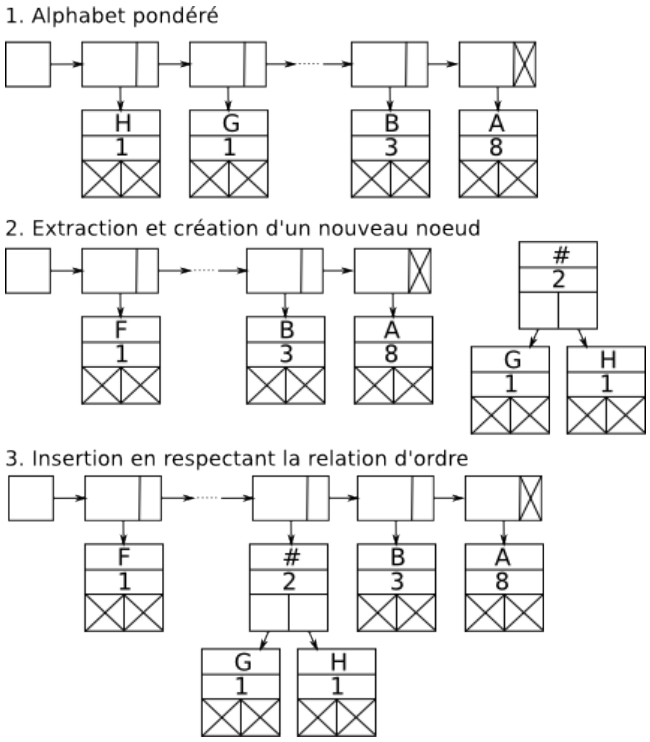
Pour construire un arbre de codage, il est d'abord nécessaire de déterminer la structure d'un noeud, un arbre n'étant qu'un pointeur sur un noeud : c'est

un champ *element* contenant le caractère représenté, un champ *poids* contenant le poids attribué au caractère représenté, ainsi que des champs *sag* et *sad* liant le noeud aux sous-arbres gauche et droit.

element	
poids	
sag	sad

Représentation d'un noeud dans l'arbre de codage

Ensuite, il est nécessaire de déterminer comment représenter l'alphabet pondéré qui va servir à constituer l'arbre de codage. On va se servir d'une liste chaînée où chaque maillon contiendra un arbre qui pointera sur un noeud correspondant à une lettre. Plus particulièrement, cette liste sera triée de telle sorte que les lettres seront classées du poids le plus faible au poids le plus fort. De cette manière, il suffira d'extraire les maillons de tête pour avoir les lettres de poids faible, par lesquelles on commence pour construire l'arbre de codage. À partir de cet alphabet pondéré, il est enfin possible de construire l'arbre de codage par un mécanisme de fusion : on extrait les deux premiers maillons de la liste triée que l'on assemble en tant que sous-arbres à un nouveau noeud de poids égal à la somme des poids des noeuds extraits, avant d'insérer ce nouveau noeud à la liste en conservant la relation d'ordre. En recommençant le processus jusqu'à ce qu'il n'y ait plus qu'un seul maillon à la liste, on obtient l'arbre de Huffman en récupérant l'arbre contenu dans cet unique maillon.



Principe de construction de l'arbre de codage

En somme, il est nécessaire d'avoir un module *liste* qui permettra d'accéder et de modifier les informations contenues par les maillons d'une liste, d'avoir des informations sur le nombre de maillons d'une liste, d'être capable de supprimer un maillon en tête de liste et d'insérer un maillon en conservant la relation d'ordre. Aussi, on pourra afficher une liste pour vérifier qu'elle est bien construite. De plus, il faudra écrire un module *arbre* qui servira à accéder et modifier les informations contenues par les noeuds d'un arbre, à construire et à afficher un arbre.

2.2 Afficher le code associé à chaque lettre

Pour afficher le code associé à chaque lettre à partir de l'arbre de codage, on parcours l'arbre de la racine vers les feuilles en ajoutant 0 à droite du code si on descend vers la gauche et 1 si on descend vers la droite.

2.3 Décoder un message

Pour décoder un message, il faut le lire caractère en parcourant l'arbre à gauche ou à droite selon la valeur du caractère lu. Dès que l'on atteint une feuille, cela signifie qu'on a terminé de décoder une lettre donc on l'affiche et on recommence le parcours à partir de la racine.

Ces trois fonctionnalités seront implémentées dans un module *huffman*.

3 Fonctionnement des modules

3.1 Module Arbre Binaire

Ce module a pour but d'offrir des fonctions pour accéder et modifier les informations contenues par les noeuds d'un arbre, et pour construire et afficher un arbre. On distingue donc quatre catégories :

- les fonctions d'accès et de modification, très proches de la structure de donnée.
- les fonctions d'informations sur l'arbre, qui nous indiquent par exemple s'il est vide ou plein (on limite un arbre à 100 noeuds pour éviter des débordements de mémoire éventuels).
- la fonction de création d'un noeud à partir de deux sous-arbres, d'un caractère et d'un poids.
- les fonctions d'affichage.

En particulier, il est possible d'afficher un élément d'un noeud (ie le couple caractère-poids), ce qui est utile pour afficher une liste-alphabet ou un arbre de codage. La fonction d'affichage d'un arbre complet est basée sur un parcours infixé de l'arbre.

3.2 Module Liste Triée

Ce module a pour but de manipuler une liste triée et ses maillons. On distingue donc quatre catégories :

- les fonctions d'accès et de modification, très proches de la structure de donnée.

- les fonctions d'informations sur la liste, qui nous indiquent par exemple si elle est vide ou unitaire (il est utile de savoir s'il ne reste qu'un seul maillon dans une liste car c'est la condition d'arrêt de l'algorithme de construction de l'arbre de codage de Huffman).
- les fonctions d'insertion et de suppression de maillons de la liste (la fonction d'insertion qui respecte la relation d'ordre est récursive et s'appuie sur la fonction d'insertion en tête).
- la fonction d'affichage, récursive, qui s'appuie sur la fonction d'affichage d'un élément d'un arbre.

Puisque la fonction d'affichage de la liste s'appuie sur la fonction d'affichage d'un élément d'un arbre, ce module dépend du module précédent.

3.3 Module Huffman

Ce module implémente les trois fonctionnalités principales que sont la construction d'un arbre de codage, l'affichage de la table de codage et la fonction de décodage d'une expression.

Construction de l'arbre de codage : on applique l'algorithme précédemment expliqué. La fonction prend en entrée un pointeur sur une liste triée. Elle utilise trois arbres intermédiaires :

- un arbre Ad qui prend l'élément extrait du maillon de tête qui est supprimé.
- un arbre Ag qui prend l'élément extrait du maillon de tête suivant qui est supprimé.
- un arbre A qui est construit à partir de Ag en sous-arbre gauche et Ad en sous arbre droit. L'élément de cet arbre a pour caractère '#' et pour poids la somme des poids de Ag et Ad.

On insère ensuite A dans la liste triée et on recommence le processus jusqu'à ce que la liste soit unitaire. On retourne alors l'unique élément de cette liste qui est en fait l'arbre de codage voulu.

Affichage de la table de codage : cette fonction parcourt l'arbre en affichant en préfixe la lettre et son code si une feuille est atteinte. C'est une récursivité terminale, il faut donc initialiser la fonction avec une chaîne vide.

Décodage : c'est une fonction itérative qui prend en entrée l'arbre de codage et le code à décoder. Elle parcourt l'arbre jusqu'à tomber sur une feuille, auquel cas elle affiche la lettre trouvée et recommence à la racine de l'arbre tant qu'il reste des *bits* à décoder.

4 Tests

Bien qu'il y ait plusieurs modules, je n'ai fait qu'un seul programme de test final car il utilise toutes les fonctions des différents modules. De plus, les modules *arbre* et *liste* sont en très grande partie repris de TPs antérieurs et ont donc déjà été validés.

Au début du test, on insère les différentes lettres pondérées tour à tour dans la liste triée pour constituer notre alphabet pondéré. Par ailleurs, on affiche

chaque étape de la construction de la liste pour visualiser le processus, jusqu'à afficher la liste triée finale, résultat demandé dans l'énoncé du TP.

Ensuite, on créé un arbre de Huffman avec la première fonction à partir de la liste précédente et on l'affiche. On affiche alors la table de codage grâce à la deuxième fonction du module *huffman*.

Enfin, on affiche quelques exemples de décodage pour montrer le bon fonctionnement de la troisième fonction du module *huffman*, avant de laisser l'utilisateur décoder la séquence de *bits* de son choix.

5 Conclusion

En somme, l'ensemble des modules du programmme fonctionnent correctement, comme le montrent les résultats du programme de test. Pour aller plus loin, il aurait été possible de faire une fonction qui, à partir d'un texte, constituerait un alphabet pondéré. Pour cela, il faudrait compter les occurences de chaque lettre pour en déduire le poids adéquat et ensuite utiliser la fonction d'insertion en liste triée déjà implémentée.