Projet CAML :
Compression/Decompression de Huffman

Philippe LELEUX
Groupe C

12 décembre 2011

Résumé :
Ce projet est l'implantation de la compression de fichier par combinaison
-de la transformée de **Burrows-Wheeler**
-de l'algorithme **Move To Front**
-du Codage de **Huffman**

Table des matières

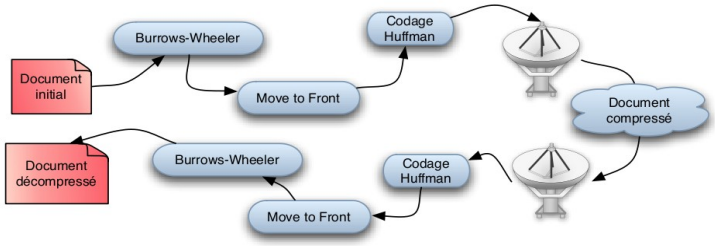
1 Présentation générale

1.1 Introduction

L'objectif est de créer un exécutable qui permet de compresser un fichier suivant la méthode du codage de Huffman. Cette transformation consiste à parcourir une liste et coder les unités par une suite de bits : les unités qui reviennent souvent seront codés par une plus petite suite de bits et ceci de manière à ce qu'aucune suite de bit ne puisse être préfixe d'une autre.

Ici ce ne sont pas directement les caractères que l'on va coder mais la distance entre deux occurrence d'un même caractère (par exemple dans « texet », la distance entre les deux 't' est de 2 car il y a un 'x' et un 'e' qui les sépare.) Pour cela on va utiliser l'algorithme Move-to-front.

Pour améliorer encore l'efficacité de la compression en diminuant la distance entre les même unités, on utilise la transformée de Burrows-Wheeler qui a l'avantage de former des groupes d'unités identiques.



1.2 Distribution fournie

Les modules concernant les fonctions par défaut sont fournies (.ml et .mli) ainsi qu 'un makefile qui permet à partir des 3 modules (burrows_wheeler, movetofront et huffman) de créer un fichier exécutable qui encode ou decode un fichier.

Les fichiers suivant étaient fournis pour effectuer ce projet :

-defaut.cmo, defaut.cmi, defaut.mli : module des fonctions de codage et décodage par défaut (sans compression ni réorganisation des données) pour chacune des trois phases. Ces fonctions pourront être utilisées pour compiler des fichiers sources non-complets.

-main.cmi et main.cmo : modules implantant l'application principale.

-huffman.mli : Module de spécification pour le type des arbre de huffman, pour la fonction de création de l'arbre de Huffman ainsi que des fonctions de compression et de décompression de huffman.

-burrows_wheeler.mli : Module de spécification pour les fonctions encode et decode selon la transformée de burrows_wheeler

-movetofront.mli : Module de spécification pour les fonctions encode et decode selon l'algorithme Move-to-front

Les 3 modules de spécification pour les fonctions et types des 3 modules à coder sont également fournis.

1.3 Spécifications

Tout d'abord, le projet doit être entièrement réalisé en fonctionnel (et codé sous CaML). Ensuite, les fonctions doivent être réalisées de manière à favoriser la lisibilité, en priorité par rapport à une quelconque optimisation. Les fonctions à implémenter doivent pouvoir manipuler des unités génériques.

Spécification des modules :

burrows_wheeler :

val encode : 'a list → int * 'a list
val decode : int * 'a list → 'a list

movetofront :

val encode : ('a -> int) → 'a list → int list
val decode : (int -> 'a) → int list → 'a list

huffman :

type 'a huffmantree
val build_tree : 'a list → 'a huffmantree
val encode : 'a list → 'a huffmantree * bool list
val decode : 'a huffmantree * bool list → 'a list

2 Conception et codage

2.1 Transformée de Burrows-wheeler

2.1.1 Encodage : val encode : 'a list → int * 'a list

L'encode selon la méthode de Burrows wheeler consiste tout d'abord, à partir d'une **'a liste** à créer la matrice carrée formée par les permutations circulaires de celle-ci. Ensuite, on range les lignes par ordre alphabétique. Enfin on renvoie le couple formé par la ligne correspondant à la liste originale et la dernière colonne de la matrice.

Ex : Pour « TEXTE » :

EXTET	ordre	ETEXT		
XTETE	alphabétique	EXTET		
TETEX	=>	TETEX	=>	(4, TTXEE)
ETEXT		TEXTE		
TEXTE		XTETE		

2.1.1.1 Création des permutations

val creer_permutations : 'a list → int → 'a list list

Cette fonction utilise une fonction auxiliaire récursive qui prend en paramètre la liste et un indice définissant le nombre de permutations (cet indice est en fait la taille de la liste à coder). Cet indice décrémente de 1 à chaque appel récursif, pendant lequel on crée une des permutations, jusqu'au cas terminal : n=0 qui stoppe la fonction.

2.1.1.2 Tri des permutations

val triFusion : 'a list → 'a list

Cette fonction tri la liste par la méthode de tri fusion : elle utilise

-une fonction auxiliaire de décomposition qui découpe une liste en deux sous-listes de même taille

val decomp : 'a list → 'a list * 'a list

-une fonction de recomposition qui à partir de deux listes triées crée une liste triée

val recomp : 'a list → 'a list → 'a list

A partir de ces deux fonctions appelées récursivement dans triFusion, on crée la liste finalement triée.

2.1.1.3 Emplacement de la liste

val emplacement_liste : 'a list → 'a list list → int

On trouve l'emplacement de la liste en appelant récursivement cette fonction en comparant à chaque appel la tête de la liste de permutations et la liste à rechercher (on renvoie la queue de la liste de permutation pour l'appel suivant).

Exception : si la liste n'est pas trouvée on renvoie « emplacement_liste : mot absent »

2.1.1.4 Dernière colonne

val derniere_colonne_perm : 'a list → 'a list

Pour trouver la dernière colonne, on appelle récursivement cette fonction sur chaque élément de la liste de permutation en faisant appel à une autre fonction : derniere_lettre (val derniere_lettre : 'a list → 'a) qui elle renvoie le dernier élément d'une liste.

2.1.1.5 Encode

val encode : 'a list → int * 'a list

Cette fonction fait simplement appel aux fonctions précédentes pour créer le couple (indice de la liste originale dans la liste des permutations et dernière colonne)

2.1.2 Decodage : val encode : int * 'a list → 'a list

Le décodage par cette méthode consiste à recréer la matrice des permutations en collant récursivement la liste obtenue au codage et en triant la liste de liste obtenue. Une fois la liste des permutations obtenue il suffit de prendre l'élément d'indice obtenu au codage.

Ex : Pour (4,TTXEE) :

	T		E		TE		ET		TET		ETE						
	T		E		TE		EX		TEX		EXT						
=>	X	=>	T	=>	XT	=>	TE	=>	XTE	=>	TET						
	E		T		ET		TE		ETE		TEX						
	E		X		EX		XT		EXT		XTE						

	TETE		ETEX		TETEX			ETEXT									
	TEXT		EXTE		TEXTE			EXTET									
=>	XTET	=>	TETE	=>	XTETE		=>	TETEX		=>	TEXTE						
	ETEX		TEXT		ETEXT			TEXTE←4									
	EXTE		XTET		EXTET			XTETE									

2.1.2.1 Recréer permutations

val recreer_permutations_triees: 'a list → int → 'a list list →'a list list

Pour recréer les permutations, on appelle récursivement la fonction auxiliaire coller_texte (val coller_texte : 'a list → 'a list list → 'a list list), qui colle élément par élément une liste dans une liste de liste,et on trie par ordre alphabétique cette liste obtenue à chaque appel.

2.1.2.2 Recherche du n-ième mot

val n_ieme_mot : 'a list list → int → 'a list

A chaque appel récursif, on renvoie la queue de la liste de permutation. Au bout de n appels, on

renvoie la tete de la liste : c'est le n'ième mot.

2.1.2.3 Decode

val decode : int * 'a list -> 'a list

Il suffit d'appeler les fonctions précédentes et de renvoyer le n'ième mot.

2.2 Algorithme Move to front

2.2.1 Encodage

Le codage par la méthode Move to front consiste à coder par une fonction arbitraire (qui peut être Char.code) une unité puis lorsqu'on l'a rencontrer, de lui donner le code 0 et d'ajouter 1 à chaque élément situé avant celui-ci. On effectue de même à chaque appel de la fonction encode en lui passant en paramètre la fonction de codage modifiée.

Ex : lorsqu'on rencontre l'élément t : fonction2 carac = {fonction carac pour carac>t
{0 pour carac=t
{ fonction carac + 1 pour carac<t

2.2.2 Decodage

Le decode consiste de même à modifier récursivement la fonction passée en paramètre (qui est au départ arbitraire et peut être Char.chr)mais cette fois si on a 0 on envoie le caractère lu précédemment sinon pour des indices strictement inférieurs à l'indice lu avant on envoie l'élément d'indice décrémenté de 1.

Ex : lorsqu'on rencontre l'indice t : fonction 2 indice = {fonction indice pour indice>t
{t pour indice=t
{fonction (indice-1) pour indice<t

2.3 Codage de Huffman

2.3.1 Type de l'arbre de Huffman

On définit l'arbre de Huffman comme une Feuille : un couple composé unité, nombre de cette unité dans la liste

un nœud composé d'un arbre gauche, d'un entier (la somme des nombres de l'arbre gauche et l'arbre droit) et un arbre droit

On a le type :
```
type 'a huffmantree =
    | Feuille of 'a * int
    | Noeud of 'a huffmantree * int * 'a huffmantree;;
```

2.3.2 Construction de l'arbre

Pour construire l'arbre, on va commence par compter l'occurrence de chaque élément dans la liste à coder et créer ainsi une liste composée de couples (élément, nombre d'occurrence), ensuite on va transformer cette liste de couples en liste d'arbres triée.

Enfin on va récursivement fusionner les arbres de « poids » (nombre d'occurrence accumulée des éléments qui le constitue) inférieurs. On obtient ainsi un arbre unique.

Remarque : il existe d'autres manières de construire l'arbre qu'en fusionnant les deux plus petits mais il est montré que cette méthode permet d'obtenir un taux moyen de compression plus grand.

Ex : Pour « TEXTE » :

=> ((T,2),(E,2),(X,1)) =>

2.3.2.1 Créer liste (unité, nombre d'occurrence)

val compter : 'a list →'a list →('a * int) list

Pour cela on a besoin de plusieurs fonction :

-val set_of_list : 'a list → 'a list : transforme une liste en ensemble en utilisant la fonction occurrence (val occurence : 'a -> 'a list -> bool) qui elle renvoie vrai si un élément donné est présent dans la liste.

-val compter_occurence : 'a → 'a list : compte le nombre d'occurrence d'un élément donné dans une liste

Connaissant l'ensemble obtenu à partir de la liste, élément par élément on compte le nombre d'occurrence et on crée une liste constituée de l'élément et son poids

2.3.2.2 Liste d'arbre triée

val listree_of_list : 'a list →'a huffmantree list

Chaque couple (élément,poids) de la liste obtenue précédemment est converti en Feuille(élément,poids), il ne reste plus qu'à trier cette liste par la méthode de tri fusion munie de la relation d'ordre entre arbres (val ordre_alphabetique : 'a hoffmantree → 'a hoffmantree → bool).

2.3.2.3 Construction de l'arbre

val build_tree : 'a list →'a huffmantree

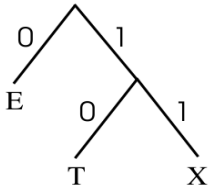
Cette fonction fait appel à une fonction auxiliaire récursive qui réduit la liste d'arbre triée en fusionnant les deux arbres de poids le plus petits avant de trier à nouveau la liste d'arbre. On a le cas terminal lorsque la liste n'est constituée que d'un arbre qu'il ne reste plus qu'à renvoyer.

2.3.3 Encodage

L'encodage par la méthode de Huffman consiste à coder chacun des éléments de la liste (qui sont maintenant chacun dans une feuille de l'arbre) par une suite de bits : en partant de la racine de l'arbre, pour coder chaque élément, on rajoute un false pour chaque branche de gauche à franchir et un true pour chaque branche de droite.

Une fois que l'on a le code de chaque élément, il ne reste plus qu'à traduire la liste en liste de bouléens.

Ex : Pour « TEXTE » :



=> TEXTE
E=0
T=10 => 10011100
X=11

2.3.3.1 Table de codage

val liste_encode :'a huffmantree → ('a*bool list) list

Pour trouver cette table de conversion arbre → liste de bouléens, on parcours l'arbre jusqu'à chaque feuille en ajoutant à chaque branche le booléens correspondant (gauche : false, droite : true) et en

concaténant les liste créées par chaque appel récursif.

2.3.3.2 Code binaire

val code_binaire : 'a liste → bool list

Pour chaque élément de la liste, on trouve son correspondant dans la table de conversion et on les place les uns à la suite des autres.

2.3.3.3 Encode

val encode : 'a list → 'a huffmantree * bool list

Pour cette fonction on appelle les fonctions précédentes pour renvoyer le couple (arbre de huffman, code binaire de la liste)

2.3.4 Decodage

Pour décoder la liste de booléen, on commence par construire une table de conversion liste de booléens → élément grâce à l'arbre.

Ensuite on compare un ensemble éléments de la liste de booléens avec les données de la table de conversion jusqu'à obtenir une correspondance que l'on prend alors avant de continuer sur la suite de la liste. Pour cela, il faut faire grandir l'ensemble d'élément que l'on compare à la table jusqu'à obtenir une correspondance.

2.3.4.1 Table de decodage

val liste_decode : 'a huffmantree → (bool list * 'a) list

C'est la même table que pour l'encode mais avec des couples inversés (simplement pour pouvoir utiliser la fonction List.assoc qui associe l'élément associé à la clé donnée dans une liste de paires)

2.3.4.2 Suite d'éléments

val find : bool list ->(bool list * 'a) list ->'a list

Pour retrouver les éléments de la liste de bouléens, on a besoin d'une fonction récursive auxiliaire qui prend en paramètre un bout de la liste de départ mais également le reste.

On teste ensuite récursivement si ce bout a une correspondance dans la table (pour cela on fait appel à une fonction qui précise si une liste est dans la table et qui renvoie l'élément correspondant et un booléen vrai si l'élément est trouvé : occurence_liste:bool list->(bool list*'a) list->int * bool).

Bout par bout jusqu'à ce que la liste soit épuisée, on trouve ainsi la liste décodée.

2.3.4.3 Decode

Pour décoder la liste de booléen avec l'arbre, il n'y a plus qu'à faire appel aux fonctions précédentes et renvoyer la liste décodée.

3 Complexité

3.1 Burrows-Wheeler

n=List.length liste

3.1.1 Encode

-creer_permutations : ordre n
-triFusion : ordre n*log n
-emplacement_liste : ordre n
-derniere_lettre : ordre n
-derniere colonne : ordre n^2
=> Encode : ordre n^2

3.1.2 Decode

-coller_texte : ordre n
-n_ieme_mot : ordre n
-recreer_permutations_triees : ordre n^2

=> Decode : ordre n^2

3.2 Move to front

3.2.1 Encode : ordre n

3.2.2 Decode : ordre n

3.3 Huffman

3.3.1Construction de l'arbre

n=List.length liste
m=nb d'éléments différents de la liste

-occurrence : ordre n
-set_of_list : ordre n^2
-compter_occurence : ordre n
-compter : ordre n^2*m
-listree_of_list : ordre m
-ordre_arbre : complexité nulle
-triFusion : ordre n*log n

=> build_tree : ordre n^2*m

3.3.2 Encode

-liste_encode : ordre n
-code_binaire : ordre n

=> encode : ordre n^2*m

3.3.3 Decode

-liste_decode : ordre n
-occurence_liste : ordre m
-find : ordre n*m

=> decode : ordre n*m

4 Tests

4.1 Méthode de test

Tout d'abord il faut faire des tests Unitaires, c'est-à-dire des tests pour s'assurer que chaque

fonction fait ce qu'on attend d'elle et seulement ça. Ces tests sont avec la spécification des fonctions, avant leur implémentation, dans les modules.

Ensuite, j'ai fait des tests d'intégration : des tests pour vérifier le bon fonctionnement de fonctions résultant d'assemblage d'autres fonctions. Les fonctions encode et decode de Burrows-Wheeler et Huffman sont des assemblages des fonctions précédentes par exemple.

Enfin les tests de validation permettent de vérifier le bon fonctionnement du code dans son ensemble. Pour faire fonctionner le programme, la procédure est simple, on met du texte dans un fichier et l'on entre ./huffman -e <<fichier>>.

4.2 Tests

Afin de tester le programme :

Pour la compression :

-j'ai d'abord entré un fichier vide, ce qui a renvoyé une exception de liste vide contenue dans le module burrows_wheeler.
-ensuite j'ai entré différents textes :

Avec un fichier contenant un caractère :
Burrows-Wheeler transformation 5.00679016113e-06sec
Move to Front encoding 3.69548797607e-05sec
Huffman encoding 1.28746032715e-05sec
Compression rate achieved ..... -3550%

Prenons maintenant la phrase :
« **L'Assemblée Générale proclame la présente Déclaration universelle des droits de l'homme** comme l'idéal commun à atteindre par tous les peuples et toutes les nations afin que tous les individus et tous les organes de la société, ayant cette Déclaration constamment à l'esprit, s'efforcent, par l'enseignement et l'éducation, de développer le respect de ces droits et libertés et d'en assurer, par des mesures progressives d'ordre national et international, la reconnaissance et l'application universelles et effectives, tant parmi les populations des Etats Membres eux-mêmes que parmi celles des territoires placés sous leur juridiction. »

Burrows-Wheeler transformation 0.0973298549652sec
Move to Front encoding 0.000674962997437sec
Huffman encoding 0.0095431804657sec
Compression rate achieved ..... -19%

Alors qu'en la mettant deux fois :
Burrows-Wheeler transformation 0.43533205986sec
Move to Front encoding 0.00101804733276sec
Huffman encoding 0.0160908699036sec
Compression rate achieved 33%

Avec un extrait du livre de job (fichier texte de 8,2ko) :
Burrows-Wheeler transformation 23.2352020741sec
Move to Front encoding 0.00592803955078sec
Huffman encoding 0.141355037689sec
Compression rate achieved 68%

Le taux de compression pour des petits fichiers n'est pas du tout intéressant, et même négatif pour des petits fichiers. Pour que cette compression soit intéressante, il faut travailler avec des fichiers d'une

certaine taille.

Pour la décompression :

On obtient bien le même fichier en décompressant sauf pour une liste vide toujours le programme renvoie une Exception.

Critique :
Les compressions sont très longues, cela doit être du au fait que les fonction ont une mauvaise complexité.

5 Conclusion

J'ai trouvé ce sujet très intéressant, la compression est un procédé que l'on utilise quotidiennement sur les ordinateurs sans forcément savoir trop ce qu'il en retourne. Il est donc très enrichissant de, non seulement apprendre une méthode de compression, mais en plus s'efforcer à l'implémenter nous-même.

Cependant, peut-être que le langage fonctionnel pur n'aurait pas été mon choix pour les deux première partie de ce travail : Burrows-Wheeler et Move To front. En effet, on y manipule des matrices et on doit aller chercher des éléments à un endroits précis ce qui aurait été plus efficace avec des pointeurs et des langages comme le C par exemple. Mais, la troisième partie montre parfaitement la puissance du langage fonctionnel dans certaines application, en langage impératif je pense que cela aurait été une horreur à coder (si c'est possible).

Le projet que je rends est loin d'être parfait :
La complexité des fonctions pourrait être largement améliorée et également la manière de coder ces fonctions. Je ne sais pas si c'est le fait que je trouve plus simple de coder avec la structure de contrôle séquence ou si c'est juste normal mais le nombre de fonction auxiliaire nécessaire pour coder est rapidement devenu impressionnant.
De plus, la manière de rédiger un rapport de projet est encore un peu flou, je ne sais pas si j'ai btout y est, ni si certaines choses sont superflues.

Un projet tel que celui-ci permet d'approfondir les connaissance que l'on a acquise en fonctionnel au long de ce trimestre et de s'intéresser à certains fonctionnement et également aux problèmes de la complexité ! Une version de movetofront ne travaillant que sur la fonction en récursif m'a donné beaucoup de fil a retordre, avant qu'un professeur ne nous dise qu'elle avait une complexité exponentielle . Le code en était :

```
let rec encode fonction liste =
        match liste with
                |[] -> []
                |t::q -> let fonction2 carac =
                                if carac=t then 0
                                        else if (fonction carac)>(fonction t) then fonction carac
                                                                                else (fonction carac)+1
        in (fonction t)::(encode fonction2 q);;
```