Compression et Décompression de Huffman

Tchomgue Mieguem Ivan Brunel
Groupe D

12 décembre 2011

Résumé

Spécification et implémentation de la compression de Huffman en *CaML*

Table des matières

Chapitre 1

Présentation générale

1.1 Introduction

L'objectif de ce projet est la réalisation d'un algorithme de compression / décompression, fondée sur les arbres de Huffman. Pour améliorer l'efficacité de cette technique, deux autres méthodes lui seront adjointes.

1.2 Distribution fournie

Les differents modules fournis sont compilés avec la version 3.11.2 du langage *CaML*. Voici les différents fichiers *CaML* fournis dans la distribution :

huffman.mli : interface du module de compression/décompression de Huffman ;

burrows_wheeler.mli : interface du module de la transformation de Burrows-Wheeler ;

movetofront.mli : interface du module de la transformation "Move-to-front" ;

defaut.cmo, defaut.cmi, defaut.mli : module des fonctions de codage/décodage par défaut utilisées pour le déboggage.

main.cmo : module implantant l'application principale.

1.3 Spécifications

Chacune des trois étapes précitées sera implantée dans un style fonctionnel, c'est-à-dire sans références, sans tableaux et sans procédures. De plus, les solutions devront être motivées par une exigence de clarté, d'exhaustivité et de concision en priorité absolue par rapport à un soucis quelconque d'optimisation.

Chapitre 2

Conception et Codage

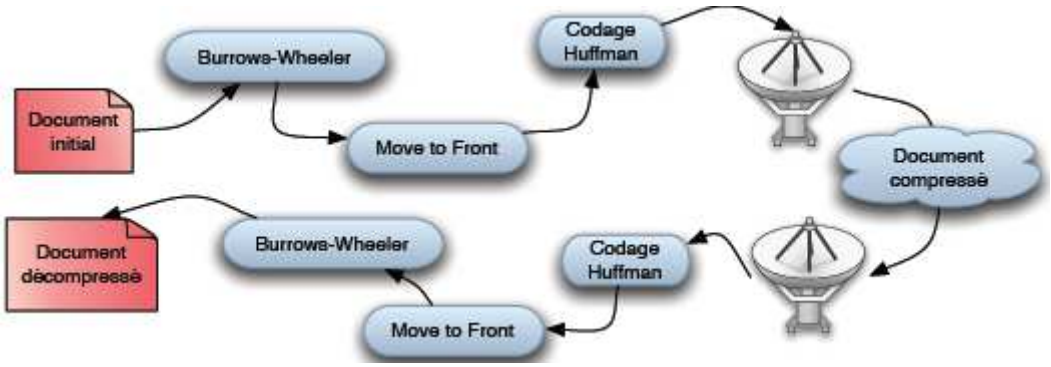


FIGURE 2.1 – Schéma général de la compression de Huffman

Le but est la compression puis la décompression d'un document. La chronologie des étapes est explicitée par le schéma ci-dessus.

2.1 Méthode de Burrows-Wheeler

2.1.1 Présentation

Comme indiqué sur la FIGURE 2.1, on appliquera en premier lieu une technique de codage, dite de Burrows-Wheeler. La transformée de Burrows-Wheeler ne compresse pas les données, elle se contente de les réorganiser de manière à obtenir un meilleur taux de compression. Elle consiste en fait à engendrer une permutation des unités de la donnée source. Cette permutation particulière, facilement inversible sans devoir la mémoriser en entier, permet souvent de regrouper les différentes occurrences d'une même unité.

2.1.2 Codage

Il faut engendrer l'ensemble des rotations de caractères de la séquence initiale, les ranger dans l'ordre de la table ASCII. Et retourner la position de la séquence initiale dans cet ensemble, ainsi que la liste formée de l'ensemble des derniers caractères de chacune des rotations préalablement rangées dans l'ordre croissant.

Exemple : Supposons que la séquence à coder soit **TEXTUEL**.Suite à un nombre de rotations déterminées on obtiendra différentes séquences que l'on triera par la suite.

Voici la matrice triée des rotations suivie de leur position :
E L T E X T U 1
E X T U E L T 2
L T E X T U E 3
T E X T U E L 4 < position de la séquence initiale
T U E L T E X 5
U E L T E X T 6
X T U E L T E 7

Le texte codé est alors constitué de la dernière colonne précédée de la position du texte original, soit : **(4, UTELXTE)**. La position de la séquence initiale sert au décodage.
La spécification de cette fonction sera la suivante :
`val encode: 'a list -> (int * 'a list) = <fun>`

Raffinage fonctionnel Les fonctions auxiliaires effectuant :
- La génération de la séquence des rotations
- Leur tri dans l'ordre croissant (on se réfère à l'ordre de la table ASCII)
- La recherche de la position de la séquence initiale dans la liste triée des rotations.
- La récupération du dernier caractère de chaque séquence de la liste préalablement triée.

nous seront indispensables pour la réalisation de notre algorithme.

Choix algorithmique

Génération de la séquence La rotation d'une liste s'obtient en concaténant sa queue à la liste formée uniquement de sa tête. Et le nombre de rotations nécessaire est exactement égale à la taille de ladite liste. On passe donc en parametre la séquence initiale et la taille de la liste. La spécification de la fonction est donc la suivante :
`val rotations_liste: 'a list -> int -> 'a list list = <fun>`

Tri de la séquence générée L'algorithme de tri utilisé ici sera le tri fusion. En effet plus le nombre d'éléments à trier augmente, plus efficace est le tri fusion par rapport aux autres algorithmes de tri (tri insertion, tri selection etc.). De plus sa complexité est de l'ordre de $\theta(nlog_2n)$. L'implantation du tri fusion se fait en 3 étapes :
- `val split: 'a list -> 'a list * 'a list = <fun>` qui coupe une liste en deux de taille égale à un élément près.
- `val merge: 'a list * 'a list -> 'a list = <fun>` qui crée une liste triée à partir de deux listes triées.
- `val tri_fusion: 'a list -> 'a list = <fun>` qui réalise le tri proprement dit.

Position de la séquence initiale On parcourt la liste, si l'élément cherché est en tête, on retourne 1 sinon on ajoute 1 à chaque appel récursif. Si on ne trouve pas l'élément on renvoie un "failwith". Concrètement ce cas n'arrivera que si l'on essaye de compresser un fichier vide. Ce qui est tout à fait logique puisqu'il n'y a aucun intérêt à compresser un fichier vide ! La spécification de la fonction est la suivante :
`val position_elt: 'a -> 'a list -> int = <fun>`

Récupération des derniers caractères Récupérer le dernier élément d'une liste revient à inverser ladite liste et à récupérer sa tête. On accumule alors les derniers éléments de chaque liste de rotation pour obtenir la liste formée des derniers éléments. Dès lors, la spécification de la fonction est :

```
val derniers_elt: 'a list list -> 'a list = <fun>
```

Encodage principal Il s'agit ici de renvoyer sous forme d'une paire la position de la séquence initiale obtenue par la fonction `position_elt` et la liste des derniers eléments de la matrice des rotations obtenue par la fonction `derniers_elt`.

2.1.3 Décodage

Le décodage consiste à reconstruire la matrice complète à partir de sa dernière colonne (texte codé « UTELXTE ») à partir de laquelle on reconstruit la colonne « suivante », c'est-à-dire, par rotation, la première, dont on sait qu'elle est dans l'ordre alphabétique (« EELTTUX »). On colle alors la dernière colonne juste avant cette première colonne, puis on classe dans l'ordre alphabétique les paires obtenues pour construire les deux premières colonnes. On répète ensuite cette opération jusqu'à constituer la matrice complète dans laquelle on retrouve le texte original par son numéro de ligne : Et on continue suivant

Initiation	Tri	Collage	Tri	Collage	Tri
U	E	UE	EL	UEL	ELT
T	E	TE	EX	TEX	EXT
E	L	EL	LT	ELT	LTE
L	T	LT	TE	LTE	TEX
X	T	XT	TU	XTU	TUE
T	U	TU	UE	TUE	UEL
E	X	EX	XT	EXT	XTU

la même procédure jusqu'à obtenir la FIGURE 2.2 On retrouve bien le texte original à la ligne dont le numéro avait été transmis avec le texte codé.
La spécification de la fonction est la suivante :

```
val decode: int * 'a list -> 'a list = <fun>
```

Raffinage fonctionnel Les fonctions auxiliaires qui nous seront utiles sont :

- La fonction de collage qui concatènera à chaque fois la séquence initiale obtenue par le codage à une séquence triée, comme le montre la FIGURE 2.2.
- La fonction de tri qui vient juste d'être définie pour le codage.
- La fonction qui générera la matrice initiale.

Choix algorithmique

Collage de la séquence Il s'agit ici d'ajouter respectivement chaque élément de la séquence(obtenue par le codage) en tête de chaque élément d'une matrice de séquences triées jusqu'à ce que la longueur de chaque élément de la matrice soit égale à la taille de

Collage	Tri	Sélection
UELTEXT	ELTEXTU	1
TEXTUEL	EXTUELT	2
ELTEXTU	LTEXTUE	3
LTEXTUE	TEXTUEL	← 4
XTUELTE	TUELTEX	5
TUELTEX	UELTEXT	6
EXTUELT	XTUELTE	7

FIGURE 2.2 – processus de décodage de Burrows_wheeler

la séquence initiale.La spécification de cette fonction est :

```
val fusion: 'a list -> 'a list list -> 'a list list = <fun>
```

Génération de la matrice initiale On constate que pour regénérer la matrice initiale, il faut exactement répéter l'opération de "collage" (n-1) fois ; n étant la longueur de la séquence initiale. On passera donc en paramètre la séquence initiale, sa longueur, et aussi la matrice colonne qui n'est rien d'autre que le tri de ladite séquence.
Pour notre exemple les paramètres correspondent donc à :
([U ;T ;E ;L ;X ;T ;E], [[E] ;[E] ;[L] ;[T] ;[T] ;[U] ;[X]], 7).
Noter que : après chaque opération de collage, la matrice obtenue est triée. La spécification de cette fonction est la suivante :

```
val create_matrix: 'a list -> 'a list list -> int -> 'a list list = <fun>
```

Décodage principal Une fois, la matrice générée, il ne reste plus qu'à extraire la séquence située à la position souhaitée. On appliquera alors un `List.nth` à la matrice générée et à (pos - 1) où pos est la position souhaitée. Le moins un dans (pos - 1) vient du fait que le premier élément de la matrice est en réalité à la position 0.

2.2 Algorithme de transformation de Move To Front

2.2.1 Présentation

L'algorithme de Move To Front consiste à remplacer chaque caractère par un indice, donné par un tableau évoluant de manière dynamique. Cette technique est notamment utilisable en conjonction avec la transformée de Burrows-Wheeler.

2.2.2 Codage

On va transformer une liste de caractère en une liste de "distances" entre les divers caractères : lorsque l'on croise un caractère une première fois, on lui attribue un code arbitraire, et lors du prochain passage, on le codera en lui donnant le nombre d'éléments qui diffèrent entre sa première occurrence et lui. Pour les tests, nous utiliserons la fonction qui à un caractère associe son code *ASCII* :`Char.code` .

Indice	0	1	2	3	4	5	6	...	25
État initial	A	B	C	D	E	F	G	...	Z
Tableau modifié par le premier E	E	A	B	C	D	F	G	...	Z
Tableau conservé 4 fois par les 4 E suivants	...								
Tableau modifié par le A	A	E	B	C	D	F	G	...	Z

FIGURE 2.3 – Principe de codage de Move To Front

Exemple : D'après la FIGURE 2.3 ci-dessus la séquence EEEEEA serait transformée en la suite 400001 ; l'évolution du tableau est décrite par la figure.
La spécification de la fonction est la suivante :
`val encode: ('a -> int) -> 'a list -> int list = <fun>`

Raffinage fonctionnel Après émission d'un caractère, on remarque que :
- Tous les éléments qui précèdent le caractère émis voient leur indice augmenté de 1.
- Tous les éléments qui se situent après le caractère émis gardent le même indice.
- L'indice de l'élément émis devient 0.

Ce qui nous amène à définir une fonction auxiliaire qui effectuera cet algorithme.

Choix algorithmique

Codage d'un élément La fonction auxiliaire utilisée sera nommée mtf (pour Move To Front).
voici son type :`val mtf : ('a -> int) -> int -> 'a -> int = <fun>`
Voici son implémentation :

Listing 2.1 – Implémentation de la fonction de codage d'un élément

```
let mtf f last_code=
  fun x -> let code = f x
           in
           if code = last_code
               then 0
               else if code < last_code
                       then code + 1
           else code
 ;;
```

C'est en fait cette fonction qui rendra l'évolution du tableau dynamique. Puisqu'elle prend une fonction en entrée et retourne une autre fonction.
Regardons de près le fonctionnement de cette fonction. Elle prend en paramètre la fonction de codage `f` , le code du dernier élément codé `last_code` et l'élément courant `x`
Elle lui associe son code grâce à la fonction de codage :
- Si son code est le même que celui du dernier caractère, alors ils sont identiques : on retourne 0.

– Sinon, si son code est inférieur à celui de dernier caractère, alors on ajoute 1 à celui ci : il se situe avant lui dans la table de codage.
– Sinon, c'est la première fois qu'on voit ce caractère : il se situe après lui dans la table de codage.

Codage principal La fonction de codage a cette implémentation :

Listing 2.2 – Implémentation de l'encodage Move To Front

```
let encode fct char_liste =
  match char_list with
  |[] -> []
  |t::q -> let code = fct t
           in      if code = 0
                     then 0::(encode fct q)
                   else code::(encode (mtf fct code) q)
;;
```

Lors du premier appel, c'est la fonction choisie par l'utilisateur qui est utilisée pour coder l'élément (dans nos tests, ce sera la fonction Char.code).
– Si le code de l'élément est différent de 0, c'est-à-dire que l'élément est différent du dernier codé, alors on insère son code dans la table, et on fait un appel récursif à la fonction de codage définie plus haut.
On décalera ainsi le code de certains éléments suivants.
– Si le code est 0, alors l'élément est le même que le précédent. On insère donc 0 dans la table, et l'on fait appel à la même fonction pour coder le reste de la liste, puisqu'il n'y a pas besoin de décalage dans la table de codage (cas des occurences de E dans la table de la FIGURE 2.3)

2.2.3 Décodage

Décodage d'un caractère Le décodage suit la même logique que le codage. La fonction `mtf` du codage sera transformée en sa fonction réciproque que nous nommons `recip_mtf`. Elle a pour type :

```
val recip_mtf : (int -> 'a) -> int -> int -> 'a = <fun>
```

Voici son implémentation :

Listing 2.3 – Implémentation de la fonction de décodage d'un caractère

```
let recip_mtf recip_f char_code =
  fun x-> if x = 0
            then (recip_f char_code)
            else if x <= char_code
                  then recip_f (x-1)
          else recip_f x
;;
```

– Si le code de l'élément est 0, alors c'est que l'élément est identique à celui d'avant : on le décode avec la fonction de décodage passée en paramètre. Pour nos tests on utilisera la fonction qui à un code *ASCII* associe le caractère correspondant : `Char.chr`

– Sinon si son code est inférieur à celui ci, on décale son code de 1 dans la table de codage.
– sinon son code est émis.

Décodage principal La syntaxe est la même que pour le codage. La liste de caracteres devient une liste d'entiers.

2.3 Codage de Huffman

2.3.1 Principe

Le principe du codage de Huffman repose sur la création d'un arbre composé de nœuds. On recherche tout d'abord le nombre d'occurrences de chaque caractère.Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids valant son nombre d'occurrences. Puis l'arbre est créé suivant un principe simple : on associe à chaque fois les deux nœuds de plus faibles poids pour donner un nœud dont le poids équivaut à la somme des poids de ses fils jusqu'à n'en avoir plus qu'un, la racine. On associe ensuite par exemple le code 0 (ou false) à la branche de gauche et le code 1 (ou true) à la branche de droite. Pour obtenir le code binaire de chaque caractère, on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie.

Exemples illustratif La figure de la PAGE 10 représente les étapes de la construction d'un code de Huffman pour l'alphabet source A, B, C, D, E, F, avec les probabilités P(A)=0.10, P(B)=0.10, P(C)=0.25, P(D)=0.15, P(E)=0.35 et P(F)=0.05.
Le code d'une lettre est alors déterminé en suivant le chemin depuis la racine de l'arbre jusqu'à la feuille associée à cette lettre en concaténant successivement un 0 ou un 1 selon que la branche suivie est à gauche ou à droite. Ainsi, sur la figure de la PAGE 10,
A=0111, B=010, C=10, D=00, E=11 et F=0110.

En appliquant ce raisonnement à la séquence TEXTE, On va tout d'abord créer la liste de paires (occurrences, éléments) suivante : [(1 ;X) ; (2 ; T) ; (2 ;E)]. On crée alors l'arbre comme suit :

[(1 ; Leaf X) ; (2 ; Leaf T) ; (2 ; Leaf E)]

$$\Downarrow$$

[(2 ; Leaf E) ; (3 ; Node(Leaf X , Leaf T))]

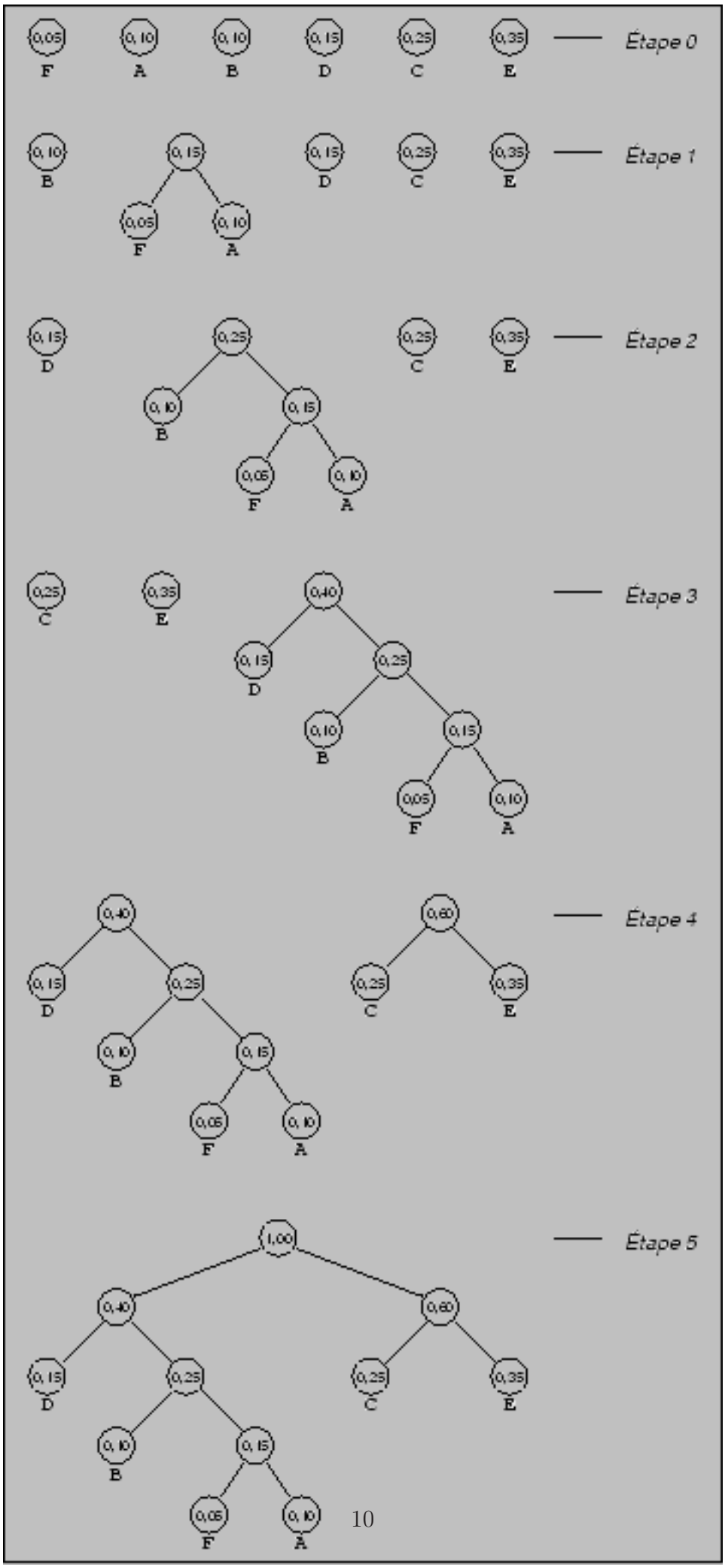
$$\Downarrow$$

[(5 ; Node (Leaf E , Node (Leaf X ,Leaf T)))]

Le code pour la séquence TEXTE est le suivant : 11 0 10 11 0, ce qui se transmet en `11010110 ou en [true;true;false;true;false;true;true;false]`

2.3.2 Représentattion d'un arbre de Huffman en *CaML*

Le type sera définit comme suit :


0,05
F
0,10
A
0,10
B
0,15
D
0,25
C
0,35
E
Étape 0
0,10
B
0,15
0,05
F
0,10
A
D
C
E
Étape 1
0,15
D
0,25
0,10
B
0,15
0,05
F
0,10
A
C
E
Étape 2
0,25
C
0,35
E
0,40
0,15
D
0,25
0,10
B
0,15
0,05
F
0,10
A
Étape 3
0,40
0,15
D
0,25
0,10
B
0,15
0,05
F
0,10
A
0,60
0,25
C
0,35
E
Étape 4
1,00
0,40
0,15
D
0,25
0,10
B
0,15
0,05
F
0,10
A
0,60
0,25
C
0,35
E
Étape 5

Listing 2.4 – définition du type d'un arbre d'huffman

```
type 'a huffmantree =
  |Leaf of 'a
  |Node of 'a huffmantree * 'a huffmantree
;;
```

2.3.3 Construction de l'arbre

La construction de l'arbre a déjà été décrite dans les pages précédentes. Nous aurons donc besoin de calculer les occurences de chacun des éléments dans la séquence à coder d'une part ; et de construire l'arbre en appliquant l'algorithme décrit ci-dessus d'autre part. Sa spécification est la suivante :

```
val build_tree : 'a list -> 'a huffmantree
```

Calcul des occurences Nous aurons besoin de la fonction `add` qui permet d'ajouter une paire `(1, elt)` à la liste si l'élément n'y figure pas ou alors d'incrémenter son occurence s'il y figure déjà. Voici le type de cette fonction :

```
val add: 'a -> (int * 'a) list -> (int * a) list = <fun>
```

La fonction de calcul d'occurence prend en paramètre une liste d'éléments, et retourne la liste des paires (occurrence,élément). la fonction de calcul des occurrences posséde la spécification suivante :

```
val occurrence_list : 'a list -> (int * 'a) list -> (int * 'a) list = <fun>
```

Création de l'arbre La liste de paires doit être triée par ordre croissant d'occurences. Afin d'inserer un élément le plus efficacement possible, on fera appel à la fonction `insert` qui insère un élément à sa place dans une liste prélablement triée par ordre croissant. Si l'element à insérer est inférieur ou égal à la tête de la liste, on le met en tête sinon on insère l'élément à la liste obtenue par appel réccursif sur la queue.
Cette fonction a le type suivant : `val insert: 'a -> 'a list -> 'a list = <fun>`

La fonction qui va fusionner les arbres d'une liste prend en paramètre une liste de couple (weight, tree), et parcourt chacun de ces éléments :

- Si la liste est vide on renvoie une exception, car on ne peut construire un arbre à partir "de rien".
- Sinon si la liste ne contient qu'un seul couple, alors l'arbre est complet : on le retourne.
- Sinon, elle contient au moins deux éléments (weight1 ; tree1) et (weight2 ; tree2). On appelle récursivement la fonction sur la nouvelle liste, obtenue par insertion (à l'aide de la fonction `insert`) du couple suivant : (weight1 + weight2 ; Node (tree1 , tree2)) dans le reste de la liste. Cette fonction est de type :
  `val build: (int * 'a huffmantree)list -> 'a huffmantree = <fun>`

Fonction principale A partir d'une liste donnée, on creara la liste des occurences à l'aide de la fonction `occurence_list`, puis on la triera avec le `tri_fusion`. On transformera cette nouvelle liste en une liste de paires (occurences , Feuilles) à l' aide de la fonction `f = fun (occ,y)->(occ, Leaf y)`. Il ne reste plus qu' à appliquer la fonction `build` à cette liste de paires pour obtenir notre arbre de Huffman.

2.3.4 Encodage

Il nous reste à renvoyer la suite de bits (ou la liste de booléens) . Pour cela, nous allons générer un dictionnaire : c'est une liste de paires qui à chaque caractère associe la séquence originale de son code dans l'arbre de Huffman. Ensuite, il s'agira de trouver chacun des caractères dans ce dictionnaire, et d'assembler les codes obtenus.

Création du dictionnaire Cette fonction prend en paramètre un arbre et renvoie la liste de paires (caractére, code). Elle utlisera une fonction qui agit sur l'arbre de gauche et une autre agissant sur l'arbre de droite. Voici l'implémentation de cette fonction :

Listing 2.5 – Implémentation de la fonction de génération du dictionnaire.

```
let fgauche= fun (e,g)->e,(false::g);;

let fdroite= fun (e,d)->e,(true::d);;

let rec dictionnaire tree =
    match tree with
    |Leaf car->[(car,[])]
    |Node (g,d)-> List.map fgauche (dictionnaire g) @
                                List.map fdroite (dictionnaire d)
;;

val dictionnaire : 'a huffmantree -> ('a * bool list)list = <fun>
```

Génération de la séquence de bits Pour définir cette fonction qu'on identifiera par `create_seq`, on aura besoin de la fonction `give_code` qui, pour un caractère donné retourne le code qui lui est associé dans le dictionnaire. On va appliquer cette fonction en concaténant le code pour le caractère (obtenu par appel à la fonction `give_code`), à la liste issue de l'appel récursif sur le reste de la liste. Les deux fonctions ont pour type :

```
val give_code : 'a -> ('a * 'b) list -> 'b = <fun>
val create_seq : 'a list -> ('a * 'b list) list -> 'b list = <fun>
```

Encodage principal Tous les éléments sont réunis. Il suffit de renvoyer la paire constituer de l'arbre et de la séquence de bits générés (la spécification impose ici qu'on retourne une liste de booléens).

2.3.5 Décodage

Il nous faut cette fois réaliser l'opération inverse. A partir d'une liste de bits, il faut recomposer par un parcours dans l'arbre le mot initial. La fonction a pour type :

```
val decode : 'a huffmantree * bool list -> 'a list = <fun>
```

Pour cela,nous aurons besoin d'une fonction intermédiaire qui parcourt l'arbre selon une liste de bits passée en paramètre et ressort le couple (caractère, liste restante).
Voici son implémentation :

Listing 2.6 – Parcours de l'arbre

```
let rec evince_letter tree liste =
    match tree,liste with
    |Leaf car,_ -> (car, liste)
```

```
    |Node _, [] -> failwith 'Erreur lors du codage'
    |Node (g,d), t::q ->if t
                        then evince_letter d q
                        else evince_letter g q
;;
```

Décodage principal Elle passe à la fonction `evince_letter` la liste de booléen et l'arbre de Huffman associé, et en récupère le caractère lu, ainsi que la nouvelle liste de booléen, qui sera son nouveau paramètre lors de son appel récursif.

Chapitre 3

Tests

Pour les tests, la méthode qui nous a été suggérée est la suivante :

* Engendrer l'application principale par la commande `make`, qui va automatiser la compilation séparée et produit l'exécutable nommé `huffman`.
* créer un fichier vide par la commande `touch «fichier»`. On l'édite en y mettant le texte à compresser.
* Produire le fichier codé par `./huffman -e «fichier»`
* Produire le fichier décodé par `./huffman -d «fichier».decoded`
* Comparer le fichier décodé au fichier initial via la commande `diff «fichier» «fichier».decoded`
* On a alors un compte rendu de l'ensemble de l'algorithme : temps d'exécution de chaque algorithme, temps d'exécution total et taux de compression.

Fichier vide La compression du fichier vide renvoie une exception comme prévu dans nos algorithmes. Il n'y a aucun intérêt à comprimer un fichier vide.

Fichier contenant un seul caractère Ici j'obtiens un taux de compression négatif. En d'autres termes le fichier est dilaté au lieu d'être compressé. Cela démeure compréhensible vu que le fichier codé contient aussi les informations nécessaires pour son décodage. Et la donnée d'un unique caractère est inférieure à ces informations.

Codage d'un unique caractère répété plusieurs fois sans espace Le taux de compression augmente avec le nombre de caractère identiques.

Codage de la touche retour chariot "entrée" «saut de ligne» Mon fichier est apparemment vide (pas de caractère visible), mais on obtient un très bon taux de compression si le nombre de retour chariot dans le fichier est important (pour environ 35 retours chariots on a 71% de taux de compression).

Fichiers textes On utilisera le générateur de faux texte Lipsum. `www.lipsum.com`.

1 paragraphe de 92 mots (les espaces ne sont pas comptabilisés) Statistiques de compression :

Burrows-wheeler transformation 0.057sec

Move to Front decoding 0.0058sec

Huffman encoding 0.0022sec

Compression rate achieved 15%

Statistiques de décompression :

Burrows-wheeler inverse transformation 0.853sec

Move to Front decoding 0.012sec

Huffman decoding 0.0000347sec

Les fichiers décodé et de départ sont identiques.

5 paragraphes de 362 mots,2437 bytes (les espaces ne sont pas comptabilisés) Statistiques de compression :

Burrows-wheeler transformation 0.809sec

Move to Front decoding 0.063sec

Huffman encoding 0.00533sec

Compression rate achieved 50%

Statistiques de décompression :

Burrows-wheeler inverse transformation 22.18sec

Move to Front decoding 0.07sec

Huffman decoding 0.00208sec

Les fichiers décodé et de départ sont identiques.

15 paragraphes de 1360 mots,9015 bytes) Statistiques de compression :

Burrows-wheeler transformation 14.58sec

Move to Front decoding 0.66sec

Huffman encoding 0.024sec

Compression rate achieved 67%

Les fichiers décodé et de départ sont identiques.

Conclusion sur les tests La plupart du temps nécessaire à la compression est consommé par la transformation de Burrows-wheeler. La décompression prend plus de temps que la compression avec mes algorithmes. Le taux de compression augmente avec le nombre de caractères pour les exemples que j'ai éffectué.

Chapitre 4

Conclusion

Le sujet permet une mise en œuvre pratique des connaissances reçues en cours. Le codage de Huffman reste optimal pour un codage par symbole, et une distribution de probabilité connue. Il ne permet pas cependant d'obtenir les meilleurs taux de compression. L'algorithme qui est vraiment important en ressources ici est la transformée de Burrows-Wheeler dont le temps d'execution est bien plus élevé que les deux autres algorithmes.
Concernant le choix du langage, je trouve que le choix de programmer de façon fonctionnelle, en utilisant la récursivité est bien adapté pour le codage de huffman. Bien que cela rende les algorithmes de Burrows-Wheeler très lourd.

Chapitre 5

Listings