# Compression et Décompression de Huffman

Tchomgue Mieguem Ivan Brunel
Groupe D

10 février 2012

# Chapitre 1

# Listings

```
(*****************************************************************************)
(*****************************************************************************)
(*****                    Transformee de Burrows_wheeler                  *****)
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*****                          Fonction d'encodage                        *****)
(*****************************************************************************)


(*****************************************************************************)
(* Sémantique: fonction matrix_rotation
 * creee la matrice des rotations différentes d'une liste.
 * Parametres:
 * liste : 'a list
 * l(length) : int, taille de la liste
 * Résultat : 'a list
 * Exceptions : Aucune
*)

(* Test :
    matrix_rotation['t';'e';'x';'t';'e'] 5 ;;
- : char list list =
[['t'; 'e'; 'x'; 't'; 'e'];
 ['e'; 'x'; 't'; 'e'; 't'];
 ['x'; 't'; 'e'; 't'; 'e'];
 ['t'; 'e'; 't'; 'e'; 'x'];
 ['e'; 't'; 'e'; 'x'; 't']]

 matrix_rotation [] 0;;
 - : 'a list list = [[]]
*)
(*****************************************************************************)

let rec matrix_rotation liste l =
        match liste,l with
        |[],_ -> [[]]
        |_,0  -> []
        |t::q,_ -> let list_rot = q@[t]
                        in liste::(matrix_rotation (list_rot) (l-1))
;;


(*****                          Debut tri fusion                           *****)

(*****************************************************************************)
(* Semantique: fonction split
 * coupe une liste en deux listes de taille egales +- 1 (on a choisi de renvoyer
 * la liste des positions paires et celle des positions impaires.
 * Parametres:
 * liste : 'a list
 * Résultat : 'a list
 * Exceptions : Aucune
*)

(* Test :
    split ['t';'e';'x';'t';'e'];;
- : char list * char list = (['t'; 'x'; 'e'], ['e'; 't'])

    split ['t'];;
```



```
- : char list * char list = (['t'], [])
*)
(*****************************************************************************)
let rec split l=
    match l with
    |[]->([],[])
    |[t]->([t],[])
    |t1::t2::q -> let (l1,l2)= split q
                        in (t1::l1, t2::l2)
;;


(*****************************************************************************)
(* Semantique: fonction merge
 * fusionne deux listes triées en une liste triée
 * Parametres:
 * l1 : 'a list
 * l2 : 'a list
 * Résultat : 'a list
 * Exceptions : Aucune
*)

(* Tests :
    merge [3;4] [];;
- : int list = [3; 4]
    merge [] [3;4];;
- : int list = [3; 4]
    merge [3;5] [1;2;3];;
- : int list = [1; 2; 3; 3; 5]
*)
(*****************************************************************************)
let rec merge l1 l2=
    match l1,l2 with
    |[],[]-> []
    |_,[] -> l1
    |[],_ -> l2
    |(t1::q1,t2::q2)->if  t1 < t2
                        then t1::merge q1 l2
                        else t2::merge l1 q2
;;


(*****************************************************************************)
(* Semantique: fonction tri_fusion
 * tri la liste dans un ordre croissant
 * Parametres:
 * l : 'a list
 * Résultat : 'a list
 * Exceptions : Aucune
*)

(* Tests :
    tri_fusion ['t';'e';'x';'t';'e'];;
- : char list = ['e'; 'e'; 't'; 't'; 'x']
*)
(*****************************************************************************)
let rec tri_fusion l=
    match l with
    |[]->[]
    |[t]->[t]
    |_-> let (l1,l2)=split l
         in merge (tri_fusion l1) (tri_fusion l2);;
```

```
(*****                          fin tri fusion                          *****)


(*****************************************************************************)
(* Semantique: fonction position_elt
 * Trouve la position d'une liste  dans une liste de listes
 * Parametres:
 * elt : 'a list
 * liste: 'a list list
 * Résultat : int
 * Exceptions : Si le motif n'est pas trouve
*)
(* Tests :
    position_elt ['p';'a'] [['j';'e'];['p';'a'];['m']];;
- : int = 2
*)
(*****************************************************************************)
let rec position_elt elt liste =
        match liste with
        |[]->failwith"position_elt:l'element est absent"
        |t::q ->if elt=t then 1
               else 1+position_elt elt q
;;

(*****************************************************************************)
(* Semantique: fonction derniers_elt
 * Recupere le dernier element de chaque liste d'une liste de listes
 * Parametres:
 * liste: 'a list list
 * Résultat : 'a list
 * Exceptions : Si la liste est de la forme [[]]
*)
(* Tests :
    derniers_elt [['j';'e'];['p';'a'];['m']];;
- : char list = ['e'; 'a'; 'm']
*)
(*****************************************************************************)
let rec derniers_elt liste  =
    match liste with
    |[] -> []
    |t::q -> let dernier_elt liste = List.hd (List.rev liste)
                in (dernier_elt t)::(derniers_elt q )
;;

(*****************************************************************************)
(* Semantique: fonction encode
 * Renvoie l'indice de position et la séquence associee
 * Parametres:
 * liste: 'a list
 * Résultat : int * 'a list
 * Exceptions : Aucune
*)

(* Tests :
    encode ['t';'e';'x';'t';'e'];;
- : int * char list = (4, ['t'; 't'; 'x'; 'e'; 'e'])
*)
(*****************************************************************************)
let encode l=
  match l with
  |[] -> (0,[])
```



```
  |_ -> let l1 = tri_fusion (matrix_rotation l (List.length l))
          in ((position_elt l l1), derniers_elt l1)
;;

(*****                fin encodage burrows_wheeler                       *****)




(*****************************************************************************)
(*****                        Fonction  de decodage                      *****)
(*****************************************************************************)


(*****************************************************************************)
(* Semantique: fonction fusion
 * fusionne les elements d'une liste respectivement en tete de chaque liste
 * d'une liste de listes
 * Parametres:
 * l1: 'a list
 * l2: 'a list list
 * Résultat : 'a list list
 * Exceptions : Aucune
*)

(* Tests :
    fusion ['t';'e';'x';'t';'e'] [['a'];['b'];['c'];['d'];['e']];;
- : char list list =
[['t'; 'a']; ['e'; 'b']; ['x'; 'c']; ['t'; 'd']; ['e'; 'e']]

*)
(*****************************************************************************)
let rec fusion l1 l2=
  match l1,l2 with
   |[],[] -> []
   |_,[] -> [l1]
   |[],_ -> l2
   |t1::q1,t2::q2-> ([t1]@t2)::(fusion q1 q2)
;;


(*****************************************************************************)
(* Semantique: fonction create_matrix
 * genere la matrice de rotations initiale et triee de la liste qu'on veut
 * coder à  partir de sa derniere colonne et de la position de la liste à  coder
 * dans cette matrice.
 * Parametres:
 * l1: 'a list
 * matrice: 'a list list , il s'agit de la matrice issue du tri de l1
 * length: int, (longueur de la liste - 1)
 * Résultat : 'a list list
 * Exceptions : Aucune
*)

(* Tests :
    create_matrix ['t';'t';'x';'e';'e'] [['e'];['e'];['t'];['t'];['x']] 4;;
- : char list list =
[['e'; 't'; 'e'; 'x'; 't'];
 ['e'; 'x'; 't'; 'e'; 't'];
 ['t'; 'e'; 't'; 'e'; 'x'];
 ['t'; 'e'; 'x'; 't'; 'e'];
```

```
 ['x'; 't'; 'e'; 't'; 'e']]
*)
(*****************************************************************************)
let f l=List.map (fun x->[x]) l;;

let rec create_matrix l1 matrice length  =
        if (length=0)
        then matrice
        else create_matrix l1 (tri_fusion (fusion l1 matrice)) (length-1)
;;

(*****************************************************************************)
(* Semantique: fonction decode
 * genere la sequence initiale à  partir des donnees issues du codage
 * Parametres:
 * (position,l1): int * 'a list
 * Résultat : 'a list
 * Exceptions : Aucune
*)

(* Tests :
   let l4 = encode ['t';'e';'x';'t';'e'];;
   decode l4;;
- : char list = ['t'; 'e'; 'x'; 't'; 'e']
*)
(*****************************************************************************)
let decode (position,l)=
     List.nth (create_matrix l (tri_fusion (f l)) ((List.length l)-1))
                (position-1)
;;


(****                  fin decodage burrows_wheeler                       ****)
```

```
(*****************************************************************************)
(*****************************************************************************)
(*******                        Encode Move To Front                    *******)
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*******                      Fonctions d'encodage                      *******)
(*****************************************************************************)

(*****************************************************************************)
(* Semantique:fonction mtf
 * On compare le code du nouveau caractere a celui de l'ancien.
 * Retourne 0 si les codes sont identiques,
 * sinon on appelera recursivement la fonction sur le reste de la liste.
 * Parametres:
 * f : ('a -> int) : fonction de codage d'un caractere
 * last_code : int : code du caractere precedent
 * Resultat : 'a -> int : Nouvelle fonction de codage
* Exceptions : Aucune
*)

(* Tests :
  mtf Char.code (Char.code 'A') 'A';;
- : int = 0
  mtf (mtf Char.code (Char.code 'A')) (Char.code 'B') 'A';;
- : int = 1
*)
(*****************************************************************************)
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
;;


(*****************************************************************************)
(* Semantique:fonction encode
 * code la liste d'elements en utilisant l'algorithme MTF
 * Parametres:
 * fct : ('a -> int) : fonction de codage
 * liste_char : liste d'elements Ã  coder
 * Resultat : int list :liste codee
 * Exceptions : Aucune
*)

(* Tests :
    encode Char.code ['a';'e';'e';'a';'b'];;
- : int list = [97; 101; 0; 1; 99]
 *)
(*****************************************************************************)
let rec encode fct liste_char =
    match liste_char with
    |[]->[]
    |t::q-> let code = fct t
            in
                if code = 0
```



```
                then 0::encode fct q
            else code::(encode (mtf fct code) q);;

(*******                    fin encodage move to front               **********)


(*****************************************************************************)
(*******                       Fonctions de decodage                  *******)
(*****************************************************************************)


(*****************************************************************************)
(* Semantique:fonction recip_mtf
 * si le code est 0, on decode le caractere,
 * sinon, on compare le code Ã  celui de l'element precedent
 * Parametres:
 * recip_f : (int -> 'a) : fonction de decodage
 * last_code : int : code du caractere precedent
 * Resultat : 'a -> int : Nouvelle fonction de codage
* Exceptions : Aucune
*)

(* Tests :
  recip_mtf Char.chr 97 0;;
- : char = 'a'
  recip_mtf (recip_mtf Char.chr 97) 98 1;;
- : char = 'a'
*)
(*****************************************************************************)

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
;;

(*****************************************************************************)
(* Semantique:fonction decode
 * reciproque de la fonction encode: apllique le decodage en
 * utilisant l'algorithme MTF
 * Parametres:
 * fct : (int -> 'a) : fonction de decodage
 * int_liste : int list: liste d'entiers obtenu par le codage
 * Resultat : 'a list :liste decodee
 * Exceptions : Aucune
*)

(* Tests :
    decode [97; 101; 0; 1; 99];;
 - : char list = ['a';'e';'e';'a';'b']
 *)
(*****************************************************************************)

let rec decode fct int_liste =
match int_liste with
|[] -> []
|t::q -> let car = fct t
         in
         if t = 0
           then car::(decode fct q)
```

```
        else car::(decode (recip_mtf fct t) q)
;;
```

```
(*****************************************************************************)
(*****************************************************************************)
(******                          Code Huffman                           ******)
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(******                     Type arbre de Huffman                       ******)
(*****************************************************************************)
type 'a huffmantree =
    |Leaf of 'a
    |Node of 'a huffmantree * 'a huffmantree
;;

(*****************************************************************************)
(******              Fonction de creation de l'arbre                  ******)
(*****************************************************************************)


(*****************************************************************************)
(* Semantique:fonction insert
 * ajoute l'element e Ã  la liste triee en gardant l'ordre.
 * Parametres:
 * e :'a
 * liste : 'a list: une liste triee d'elements
 * Resultat : 'a list
* Exceptions : Aucune
*)

(* Tests :
   insert 'c' ['a';'b';'d';'e'];;
- : char list = ['a'; 'b'; 'c'; 'd'; 'e']
*)
(*****************************************************************************)
let rec insert e liste =
  match liste with
  |[] -> [e]
  |t::q -> if e <= t
           then e::liste
           else t::(insert e q)
;;
(*****************************************************************************)
(* Semantique:fonction build
 * construit l'arbre de huffman en partant des feuilles
 * Parametres:
 * tree : (int * 'a huffmantree)
 * Resultat : 'a huffmantree
* Exceptions : la liste passee en parametre est vide
*)

(* Tests :
 build [(1, Leaf 'x');(2, Leaf 'e');(3, Leaf 't')];;
- : char huffmantree = Node (Leaf 't', Node (Leaf 'x', Leaf 'e'))
*)
(*****************************************************************************)
let rec build tree_list =
        match tree_list with
        |[]-> failwith"on ne peut construire un arbre si on a aucun element"
        |[(p,tree)] -> tree
        |(w1,tree1)::(w2,tree2)::q -> build(insert (w1+w2, Node(tree1,tree2)) q)
;;
```



```
(*****************************************************************************)
(* Semantique:fonction add
 * incremente l'occurence de elt dans la liste
 * Parametres:
 * elt : 'a
 * liste: (int * 'a) list, liste d'elements (occurences,caractere)
 * Resultat : (int * 'a) list
* Exceptions : Aucune
*)

(* Tests :
   add  'e' [(1, 'x');(2, 'e');(3, 't')];;
- : (int * char) list = [(1, 'x'); (3, 'e'); (3, 't')]
*)
(*****************************************************************************)
let rec add elt liste =
        match liste with
        |[] -> [(1,elt)]
        |(occ,e)::q -> if e = elt
                       then (occ+1,elt)::q
                       else (occ,e)::add elt q
;;


(*****************************************************************************)
(* Semantique:fonction occurence_list
 * construit Ã  partir d'une liste d'elements la liste de paires (occurence,elt)
 * Parametres:
 * liste : 'a list :liste d'elements
 * accu: (int * 'a) list:accumulateur contenant la  liste de paires
 * (occurences,caractere)
 * Resultat : (int * 'a) list
* Exceptions : Aucune
*)

(* Tests :
  occurence_list ['t';'e';'x';'t';'e'] [];;
- : (int * char) list = [(2, 't'); (2, 'e'); (1, 'x')]
*)
(*****************************************************************************)

let rec occurence_list liste accu =
        match liste with
        |[] -> accu
        |t::q -> occurence_list q (add t accu)
;;


(*****                         Debut tri fusion                          *****)

(*****************************************************************************)
(* Semantique: fonction split
 * coupe une liste en deux listes de taille egales +- 1 (on a choisi de renvoyer
 * la liste des positions paires et celle des positions impaires.
 * Parametres:
 * liste : 'a list
 * RÃ©sultat : 'a list
 * Exceptions : Aucune
*)
```

```
(* Test :
    split ['t';'e';'x';'t';'e'];;
- : char list * char list = (['t'; 'x'; 'e'], ['e'; 't'])

    split ['t'];;
- : char list * char list = (['t'], [])
*)
(*****************************************************************************)
let rec split l=
    match l with
    |[]->([],[])
    |[t]->([t],[])
    |t1::t2::q -> let (l1,l2)= split q
                      in (t1::l1, t2::l2)
;;


(*****************************************************************************)
(* Semantique: fonction merge
 * fusionne deux listes triÃ©es en une liste triÃ©e
 * Parametres:
 * l1 : 'a list
 * l2 : 'a list
 * RÃ©sultat : 'a list
 * Exceptions : Aucune
*)

(* Tests :
    merge [3;4] [];;
- : int list = [3; 4]
    merge [] [3;4];;
- : int list = [3; 4]
    merge [3;5] [1;2;3];;
- : int list = [1; 2; 3; 3; 5]
*)
(*****************************************************************************)
let rec merge l1 l2=
    match l1,l2 with
    |[],[]-> []
    |_,[] -> l1
    |[],_ -> l2
    |(t1::q1,t2::q2)->if  t1 < t2
                      then t1::merge q1 l2
                      else t2::merge l1 q2
;;


(*****************************************************************************)
(* Semantique: fonction tri_fusion
 * tri la liste dans un ordre croissant
 * Parametres:
 * l : 'a list
 * RÃ©sultat : 'a list
 * Exceptions : Aucune
*)

(* Tests :
    tri_fusion ['t';'e';'x';'t';'e'];;
- : char list = ['e'; 'e'; 't'; 't'; 'x']
*)
(*****************************************************************************)
```



```
let rec tri_fusion l=
    match l with
    |[]->[]
    |[t]->[t]
    |_-> let (l1,l2)=split l
         in merge (tri_fusion l1) (tri_fusion l2);;

(*****                          fin tri fusion                              *****)




(*****************************************************************************)
(* Semantique: fonction build_tree
 * construit l'arbre de huffman Ã  partir d'une liste d'elements
 * Parametres:
 * liste : 'a list, liste d'elements
 * RÃ©sultat : 'a huffmantree
 * Exceptions : la liste passÃ©e en parametre est vide
*)

(* Tests :
    build_tree ['t';'e';'x';'t';'e'];;
- : char huffmantree = Node (Leaf 't', Node (Leaf 'x', Leaf 'e'))
*)
(*****************************************************************************)

let f = fun (x,y) -> (x, Leaf (y));;

let build_tree liste =
  let list_occ = occurence_list liste []
      in let list_occ_triee = tri_fusion list_occ
          in build (List.map f list_occ_triee)
;;


(*****************************************************************************)
(******                          Fonction de codage                         ******)
(*****************************************************************************)

(*****************************************************************************)
(* Semantique: fonction dictionnaire
 * construit le dictionnaire de tous les caracteres, en les associant
 * leur chemin dans l'arbre passÃ© en parametre.
 * Parametres:
 * tree : 'a  huffmantree
 * RÃ©sultat : ('a * bool list) list
 * Exceptions : Aucune
*)

(* Tests :
    dictionnaire (Node (Leaf 't', Node (Leaf 'x', Leaf 'e')));;
- : (char * bool list) list =
[('t', [false]); ('x', [true; false]); ('e', [true; true])]
*)
(*****************************************************************************)
let fgauche= fun (e,g) -> e,(false::g);;
(*val fgauche : 'a * bool list -> 'a * bool list = <fun>*)

let fdroite= fun (e,d) -> e,(true::d);;
(*val fdroite : 'a * bool list -> 'a * bool list = <fun>*)
```

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

(*****************************************************************************)
(* Semantique: fonction give_code
 * ressort le chemin dans l'arbre pour obtenir le caractere.
 * Parametres:
 * car: 'a , caractere Ã  trouver
 * liste: ('a * 'b) list. Il s'agit en fait du dictionnaire
 * RÃ©sultat : 'b
 * Exceptions : Aucune
*)

(* Tests :
    give_code 'x' [('t', [false]); ('x', [true; false]); ('e', [true; true])];;
- : bool list = [true; false]
*)
(*****************************************************************************)

let rec give_code car liste =
    match liste with
    |[]->failwith "give_code: le caractere n'existe pas"
    |(t,e)::q -> if t = car
                 then e
                 else give_code car q
;;

(*****************************************************************************)
(* Semantique: fonction create_seq
 * construit le chemin Ã  parcourir dans l'arbre pour obtenir le mot entier.
 * Parametres:
 * char_liste: 'a list ,liste de caracteres (constituant le mot Ã  trouver)
 * dictionary: ('a * bool list) list. dictionnaire
 * RÃ©sultat : bool list
 * Exceptions : Aucune
*)

(* Tests :
    create_seq ['t';'e';'x'] [('t', [false]); ('x', [true; false]); ('e', [true;
 true])];;
- : bool list = [false; true; true; true; false]
*)
(*****************************************************************************)

let rec create_seq char_liste dictionary =
    match char_liste with
    |[] -> []
    |t::q -> (give_code t dictionary) @
                          (create_seq q dictionary)
;;


(*****************************************************************************)
(* Semantique: fonction encode
 * construit l'arbre de huffman et le code du mot passÃ© en parametre.
 * Parametres:
 * liste: 'a list ,liste de caracteres (constituant le mot Ã  coder)
```



```
 * RÃ©sultat : 'a huffmantree * bool list
 * Exceptions : la liste passee en parametre est vide.
*)

(* Tests :
   encode ['t';'e';'x';'t';'e']
    : char huffmantree * bool list =
(Node (Leaf 't', Node (Leaf 'x', Leaf 'e')),
 [false; true; true; true; false; false; true; true])
 *)
(*****************************************************************************)

let encode liste =
    match liste with
    |[]->failwith"encode:on ne peut coder une liste vide"
    |_-> (build_tree liste ,
                  create_seq liste (dictionnaire (build_tree liste )))
;;


(*****************************************************************************)
(******                          Fonction de decodage                   ******)
(*****************************************************************************)

(*****************************************************************************)
(* Semantique: fonction evince_letter
 * Ã  partir d'un chemin dans l'arbre,
 * elle ressort le caractere obtenu et le reste de la liste de booleen
 * Parametres:
 * tree: 'a huffmantree
 * liste: bool list: chemin dans l'arbre
 * RÃ©sultat : 'a * bool list
 * Exceptions : liste vide quand on est sur un noeud
*)

(* Tests :
    evince_letter (Node (Leaf 't', Node (Leaf 'x', Leaf 'e')))
    [false;true;true;true;false;false;true;true];;
 : char * bool list = ('t', [true; true; true; false; false; true; true])
 *)
(*****************************************************************************)

let rec evince_letter tree liste =
    match tree,liste with
    |Leaf car,_ -> (car, liste)
    |Node _, [] -> failwith "Erreur dans le codage"
    |Node (g,d), t::q ->if t
                        then evince_letter d q
                        else evince_letter g q
;;

(*****************************************************************************)
(* Semantique: fonction decode
 * decode le mot,
 * Parametres:
 * (tree, liste): 'a huffmantree * bool list
 * RÃ©sultat : 'a list
 * Exceptions : aucune
*)

(* Tests :
    decode (Node (Leaf 't', Node (Leaf 'x', Leaf 'e')),
```

```
    [false;true;true;true;false;false;true;true]);;
 - : char list = ['t'; 'e'; 'x'; 't'; 'e']

 *)
(*****************************************************************************)


let rec decode (tree, liste) =
    match liste with
    |[]->[]
    |_ -> let (a,b)= evince_letter tree liste
            in a::(decode (tree, b))
;;
```