Option Informatique
Arbre de Huffman (ESIM 97) - Corrigé

Sujet

3 avril 2007

Algorithme de construction de l'arbre

Question 1 *En utilisant l'arbre ci-dessus, décoder le message 111110010111111101.*

On parcourt l'arbre selon l'algorithme qui nous servira dans la question 10, à savoir :

```
DEBUT
texte_decode <- ""
TANT QUE texte_code <> ""
se placer à la racine de l'arbre de Huffman
  TANT QUE on n'est pas sur une feuille
    SI premier_chiffre(texte_code) = 0
    ALORS
      parcourir la branche gauche
    SINON
      parcourir la branche droite
    FIN SI
  FIN TANT QUE
    écrire le caractère contenu dans la
      feuille à la suite de texte_decode
FIN TANT QUE
afficher texte_decode
FIN
```

On obtient ainsi le mot HACHE.

Question 2 *Dans le cas où les* `nbcar` *caractères du texte à coder ont la même fréquence d'apparition, quelle est la profondeur de l'arbre de Huffman ?*

Dans cette question, on va être plus ambitieux que la preuve restreinte faite en TP, et montrer d'abord un résultat pour tout arbre de Huffman.

Si un caractère x a une fréquence $f(x)$ et un code de longueur $c(x)$ alors sa représentation dans le message encodé coûtera $f(x)c(x)$ bits.

On veut d'abord montrer que l'arbre de Huffman minimise le coût de la réprésentation du message, c'est à dire $\Sigma_x c(x)f(x)$.

Soit L la liste de construction de l'arbre de Huffman, et x et y deux arbres avec les poids $f(x)$ et $f(y)$ minimaux dans L.

Soit $L' = L \cup \{z\} \backslash \{x, y\}$, où $z \notin L$ est de poids $f(x)+f(y)$. Montrons que si T' arbre binaire strict, représente un arbre de codage *optimal pour la liste* L', alors l'arbre T, obtenu en remplaçant z par un noeud à deux fils x et y de poids $f(x)$, $f(y)$ dans T' est *optimal pour la liste* L.

Il suffira de considérer que l'arbre associé à la liste réduite à un seul élément est unique, et donc nécessairement optimal, pour conclure.

On note $C(T)$ la fonction de coût induite par l'arbre T codant les caractères de L.

$$C(T) = \Sigma_{x \in L} c(x)f(x)$$

On remarque

$$C(T) = C(T') + c(x)f(x) + c(y)f(y) - c(z)f(z)$$

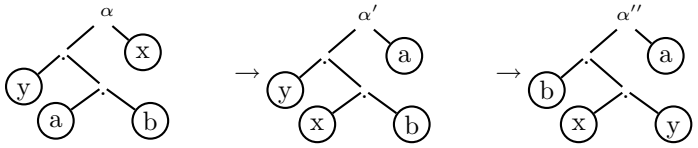
Or $c(z) = (1 - c(x)) = (1 - c(y))$ et $f(z) = (f(x) + f(y))$ par construction, donc :

$$\boxed{C(T) = C(T') + f(x) + f(y)}$$

Il reste à montrer que T est optimal pour L. Soit α un arbre binaire complet représentant un code préfixe optimal pour L. On veut montrer que $C(\alpha) = C(T)$.

x et y sont donc deux feuilles de α, de poids minimal, mais dont on ne connaît pas forcément la position. Ceci dit, on peut modifier l'arbre α de façon à ce que x et y soient deux frères de profondeur maximale dans α, sans changer son coût.

Pour cela, on considère a et b deux caractères de profondeur maximale dans α. Sans perte de généralité, on suppose $f(a) \leq f(b)$ et $f(x) \leq f(y)$. Puisque $f(x)$ et $f(y)$ sont les plus petites fréquences de L, et que $f(a), f(b)$ sont deux fréquences quelconques, on a $f(x) \leq f(a)$ et $f(y) \leq f(b)$. On échange alors les positions de a et x, d'une part, puis de b et y, comme montré dans la figure ci-dessous :



On obtient un nouvel arbre α' puis α''. La différence de coût induite est positive à chaque tranformation, donc conduit vers un arbre *plus optimal* :

$$\begin{aligned} C(\alpha) - C(\alpha') &= f(x)c(x) + f(a)c(a) - f(x)c'(x) - f(a)c'(a) \\ &= f(x)c(x) + f(a)c(a) - f(x)c(a) - f(a)c(x) \\ &= (f(a) - f(x))(c(a) - c(x)) \\ &\geq 0 \end{aligned}$$

On en tire que *quitte à effectuer quelques permutations de feuilles, on peut considérer que x et y sont deux frères de poids minimal dans α.*

Appelons maintenant β l'arbre obtenu à partir de α en enlevant x et y de l'arbre, et en le remplaçant par un noeud unique z de poids $f(x)+f(y)$. On obtient, à partir de α (arbre de codes pour L) un arbre β (de codes pour L'). Cést la transformation inverse de celle utilisée pour passer de T' à T ci-dessus, et le même calcul s'applique :

$$C(\alpha) = C(\beta) + f(x) + f(y)$$

Mais comme T' est optimal pour L' :

$$C(T') \leq C(\beta)$$

Donc

$$C(T) = C(T') + f(x) + f(y) \leq C(\beta) + f(x) + f(y) = C(\alpha)$$

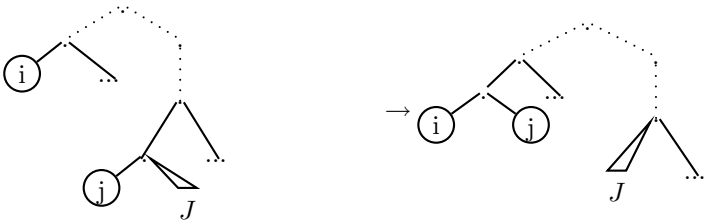
Donc $C(T) = C(\alpha)$, par optimalité de α. L'arbre T est optimal pour L.

On se place maintenant dans le cas où toutes les feuilles ont le même poids. On va montrer que l'arbre de Huffman est *quasi-complet*, c'est à dire que *les profondeurs de deux feuilles quelconques de l'arbre diffèrent au plus de 1.*

On a $\forall x, f(x) = f$.

Par l'absurde, soit T un arbre supposons qu'il existe i, j feuilles telles que $|c(i) - c(j)| \geq 2$. On suppose, sans perte de généralité, $c(i) \leq c(j) - 2$. On note J' le sous-arbre frère de j (arbre binaire strict).

On considère maintenant l'arbre T' dans lequel on a enlevé la feuilles j, remonté son frère J' à la place de leur père et où l'on a remplacé la feuille i par un noeud ayant pour fils i, j.



Alors $C(T) - C(T') = f(j)c(j) + f(i)c(i) - (f(j)c'(j) + f(i)c'(i) - \Sigma_{k \in J} f(k))$, et comme toutes les fréquences sont identiques :

$$C(T) - C(T') = f\,(c(j) - c'(j) + c(i) - c'(i) - |J|) > 0$$

Car $c'(i) = c(i) + 1$ mais $c'(j) = c'(i) = c(i) + 1 \leq c(j) - 2 + 1$ et $|J| \geq 1$. On a trouvé un arbre binaire strict dont le coût est strictement plus petit que celui de l'arbre de Huffman, et par optimalité, c'est absurde.

Donc l'arbre considéré est quasi-complet, donc si il a n caractères distincts sa hauteur est $\lceil \ln_2(n) \rceil$

Question 3 *Écrire la procédure sans paramètre* `initialisations` *qui doit initialiser les tableaux* `car` *et* `frequence` *et la variable* `nbcar`*. Pour ce faire, la procédure doit examiner tous les caractères du texte à compresser* `txt`*.*

```
let initialisations () =
  let texte =!txt in
  for k = 0 to (string_length texte - 1) do
    let tk = texte.[k] and met=ref false in
    for j = 0 to!nbcar - 1 do
      if tk = car.(j) then begin
        frequence.(j) < - frequence.(j) + 1 ;
        met := true ;
        end
    done ;
      if (not!met) then begin
        incr nbcar ;
        car.(!nbcar - 1) < - tk ;
        frequence.(!nbcar - 1) < - 1 ;
      end ;
  done ; ;
```

Question 4 *Écrire les deux versions, récursive et itérative, de la fonction booléenne* `appartient` *qui teste si un caractère* `c` *est présent ou non dans une chaîne* `ch`*.*

```
let rec appartient c = function
   "" → false
  |ch → if nth_char ch 0 = c then true
         else appartient c
         (sub_string ch 1 (string_length ch - 1)) ; ;

let appartient c ch =
  let k = ref 0 in
  while (nth_char ch!k)<>c && (!k <= string_length ch -
1) do
    k :=!k+1 ;
  done ;
  not (!k = string_length ch) ; ;
```

Question 5 *Écrire la fontion* `insere` *qui renvoie la liste obtenueaprès avoir inséré un arbre dans une liste d'arbres ordonnée suivant l'ordre croissant des champs* `poids` *des noeud racines. La liste résultat doit respecter cette condition.*

On ne se préoccupe pas du cas de l'arbre `nil` dans la liste traitée, puisque celui-ci ne se produira jamais lors de la construction de lárbre de Huffman.

```
let rec insere a liste =
  match liste with
      [] → [a]
    |t : :q → match (a,t) with
          (noeud (g,x,d),noeud (g',x',d')) →
            if x.poids < x'.poids then
              a : :liste
            else
              t : :(insere a q) ; ;
```

Question 6 *En utilisant la fonction précédente, écrire la fonction* `creer_liste` *qui renvoie comme résultat la liste des* `nbcar` *arbres réduits à un seul noeud. La liste doit être classée par ordre croissant des champs* `poids` *des noeuds. Cette fonction utilise les tableaux* `car` *et* `frequence` *qui sont des variables globales.*

```
let creer_liste () =
  for k = 0 to !nbcar - 1 do
    liste := insere
   (noeud (nil,poids = frequence.(k) ;
    chaine = char_for_read car.(k),nil))
   !liste
  done ;
  liste
    where liste = ref [] ; ;
```

Question 7 *Écrire la fonction* `fusion` *qui prend pour argument deux variables* `a1` *et* `a2` *et renvoie comme résultat un nouvel arbre* `a` *tel que :*

- *le champ* `poids` *de* `a` *est la somem des champs* `poids` *des noeuds racines des arbres* `a1` *et* `a2`*.*
- *Le champ* `chaine` *de* `a` *est la concaténation des champs* `chaine` *des noeuds racines des arbres* `a1` *et* `a2`*.*
- *le champ* `fg` *de* `a` *est l'arbre* `a1`*.*
- *le champ* `fd` *de* `a` *est l'arbre* `a2`*.*

```
let fusion a1 a2 =
  match (a1,a2) with
    (noeud (_,x1,_),noeud (_,x2,_)) →
       noeud (a1,poids = x1.poids + x2.poids ;
         chaine = x1.chaine^x2.chaine,a2) ; ;
```

Question 8 *Écrire la fonction* `creer_arbre` *qui prend pour argument une liste ordonnée d'arbres et renvoie comme résultat l'arbre de Huffman. Il s'agit ici de programmer l'algorithme de construciton de l'arbre de Huffman.*

```
let creer_arbre liste =
  let rec f l = match l with
    [a] → a
   |a1 : :a2 : :q → f (insere (fusion a1 a2) q)
  in f !liste ; ;
```

Question 9 *Écrire la fonction* `codage` *qui prend pour argument une chaîne* `ch` *et qui renvoie comme résultat la chaîne codée, c'est à dire que chaque caractère de la chaîne est remplacé par son code de Huffman (suite de 0 et de 1). La variable globale* `huff` *désigne l'arbre de Huffman.*

La fonction trace détermine dans quel sous-arbre se trouve le caractère à coder, et appelle récursivement sur ce sous-arbre, après avoir enregistré en tête du résultat la branche choisie.

```
let codage ch =
  let rec trace c a =
    match a with
       noeud (nil,_,_) → ""
      |noeud (a1,_,a2) →
        match a1 with
          noeud (_,paire,_)
          when appartient c paire.chaine →
               (string_of_int 0)^(trace c a1)
         |_ → (string_of_int 1)^(trace c a2)
  in
  let ch_codé = ref ""
  in
  for k = 0 to string_length ch - 1 do
    ch_codé := !ch_codé^(trace ch.[k] !huff)
  done ;
 !ch_codé ; ;
```

Question 10 *Écrire la fonction* `decodage` *qui prend en argument une chaine correspondant à un texte codé et qui renvoie comme résultat la chaîne décodée.*

C'est l'algorithme appliqué en début de sujet :

```
let decodage ch =
 let rec piste ch' a =
   match a with
      noeud (nil,paire,_) → (paire.chaine,ch')
     |noeud (a1,_,a2) →
       if ch'.[0] = '0' then
         piste (sub_string ch' 1
           (string_length ch' - 1)) a1
       else
         piste (sub_string ch' 1
           (string_length ch' - 1)) a2
 in
 let ch_codé = ref ch and ch_décodé = ref ""
 in
 while !ch_codé <> "" do
   let (s,s') = piste !ch_codé !huff in begin
     ch_décodé := !ch_décodé^s ;
     ch_codé := s'
   end
 done ;
!ch_décodé ; ;
```

Question 11 *Écrire la fonction* `recherche` *qui renvoie comme résultat un nombre entier, qui indique le range où est placé le caractère* `c` *dans le tableau classé* `t`*. On demande, dans cette question, de programmer l'algorithme de recherche dichotomique.*

Question de cours.

```
let recherche c t =
  let g = ref 0 and d = ref (vect_length t - 1)
  and en_attente = ref (-1) in
    while !g <= !d && (!en_attente = -1) do
      let milieu = (!g + !d) / 2 in
      if t.(milieu) = c then
        en_attente := milieu ;
      if t.(milieu) < c then
        g := milieu + 1 else d := milieu - 1 ;
    done ;
 !en_attente ; ;
```

Question 12 *Quelle est la complexité de l'algorithme précédent ?*

$O(ln_2(n))$, où n est la taille du tableau (cours).

Question 13 *Écrire la procédure* `parcours` *qui parcourt l'arbre de Huffman une et une seule fois, de manière à ce que lorsqu'une feuille est atteinte, la variable globale* `code` *contienne le code de l'unique caractère «étiquetant» cette feuille. La valeur de la variable* `code` *est alors rangée dans le tableau* `huff_code` *à l'endroit voulu.*

```
let rec parcours = function
   noeud (_,p,_) when string_length p.chaine = 1 →
     huff_code.(recherche (nth_char p.chaine 0) car) < - !code
  |noeud (g,_,d) →
     code := !code^"0" ;
     parcours g ;
     code := sub_string !code 0 (string_length !code - 1) ;
     code := !code^"1" ;
     parcours d ;
     code := sub_string !code 0 (string_length !code - 1) ; ;
```