TP MP2I : compression de Lempel-Ziv-Welch

Crédits

— Un travail de Vincent Simonnet,

— Wikipedia.

Présentation

L'algorithme de *Lempel-Ziv-Welch* (LZW [1]) est un algorithme de compression de données sans perte [2] Il a été utilisé dans des modems aujourd'hui obsolètes mais on le trouve encore dans la compression des images « GIFF » ou « TIFF » et les fichiers audio « MOD ». Il est à la base de la compression « ZIP ».

Facile à coder (c'est son principal avantage) il n'est souvent pas optimal car il effectue une analyse sommaire des données à compresser.

Préambule

Q.1 Ecrire une fonction `make_dict (alphabet: char array) : (string, int) Hashtbl.t` qui prend pour argument un alphabet (tableau de caractères) de taille n et crée un dictionnaire d (caractère (vu comme un string),position dans le tableau).

```
# let d = make_dict [|'a';'b';'c'|] in
Hashtbl.iter
  (fun k v -> Printf.printf "%s:%d," k v) d;;
    a:0,b:1,c:2,- : unit = ()
```

Compression

Pour les notions de *facteur* et *préfixe* d'un mot, on renvoie à ce cours.

On veut compresser un texte T selon l'algorithme LZW vu en cours.

1. du nom de ses inventeurs Abraham Lempel, Jakob Ziv qui l'ont proposé en 1977 et Terry Welch qui l'a finalisé en 1984

2. Après un cycle compression-décompression, on retrouve intégralement les données initiales.

Q.2 Appliquer l'algorithme à la main pour le mot

ababcbababaaaaaaa

dans l'alphabet $\Sigma = \{a, b, c\}$ avec un dictionnaire/tableau initialisé à

facteur	"a"	"b"	"c"	""	""	...
codage	0	1	2	?	?	?

Donner en particulier le texte codé et ce que devient le dictionnaire.

Q.3 Ecrire la fonction `compresse (s:string) (alphabet: char array) : int list` qui prend en paramètre une chaîne de caractères écrite dans l'alphabet ASCII et renvoie son codage LZW sous forme de listes d'entiers.

```
# let s="TOBEORNOTTOBEORTOBEORNOT" in compresse s ascii;;
- : int list =
[84; 79; 66; 69; 79; 82; 78; 79; 84; 256; 258; 260; 265; 259;
    261; 263]
```

Décompression

La procédure de décompression prend en entrée la liste produite par la fonction `compresse` et retrouve la chaîne de caractères initiale (la compression est sans perte). La difficulté réside dans le fait que l'on ne conserve pas le dictionnaire construit lors de la compression. Il faut donc le reconstruire lors de la décompression.

Principe de décompression

On note d le dictionnaire (code,facteur) qui est l'inverse de celui de la partie précédente (en fait, puisque l'ensemble des codes forme un intervalle de nombres, un simple tableau suffit).

Algorithme On initialise le dictionnaire avec l'alphabet (par exemple alphabet ASCII des caractères codés sur 8 bits).

```
fonction lzw_decompress(T' : texte compressé,
                        d : dictionnaire (code,facteur)):
    c ← Lire(T'); /*1er code lu*/
    /*le 1er code correspond toujours à une lettre*/
    Ecrire d[c]; /*ajouter le texte codé par c*/
    tant que il reste un codes de T' non lu faire
        n ← lire(T'); /*code courant*/
        si n est une clef de d /*code n déjà rencontré*/
            alors e ← d[n]; /*décompression*/
                  d[|d|] ← d[c] · e[0]
            sinon /*décompression, cas n = |d|*/
                  e ← d[c] · d[c][0];
                  d[|d|] ← e
        Ecrire e;
        c ← n
    fin_faire
```

Q.4 Décompresser à la main le codage obtenu à la question Q4.

Q.5 Écrire la fonction `decompresse : int list -> string` qui prend en paramètre un liste d'entiers (une liste de codage de compression d'un texte comme renvoyée par `compresse`), calcule le dictionnaire des codages et renvoie la chaîne de caractères initiale (celle d'avant le codage).

```
# decompresse [84; 79; 66; 69; 79; 82; 78; 79; 84; 256; 258;
    260; 265; 259; 261; 263];;
- : string = "TOBEORNOTTOBEORTOBEORNOT"
```

Évaluation du taux de compression

La représentation du résultat de la compression par une liste d'entiers OCAML n'est pas très réaliste : il faudrait *a priori* 30 [3] bits pour stocker chaque entier. Cependant, on remarque que la taille des entiers produits par l'algorithme de compression croît progressivement au fur et à mesure que l'on avance dans la liste (et que le dictionnaire se remplit). Dans la pratique, on peut donc utiliser la technique suivante pour coder la liste :

— Tant que tous les entiers sont strictement inférieurs à 255, coder ces entiers sur 8 bits.

— Lorsque l'on rencontre le premier entier supérieur ou égal à 255, émettre la séquence 11111111 (huit fois le bit 1) et continuer, tant que les entiers sont strictement inférieurs à 511, en codant les entiers sur 9 bits.

— Lorsque l'on rencontre le premier entier supérieur ou égal à 511, émettre la séquence 111111111 (neuf fois le bit 1) et continuer, tant que les entiers sont strictement inférieurs à 1023, en codant les entiers sur 10 bits.

De manière générale, tant que les entiers considérés sont strictement inférieurs à $n = 2^k - 1$, on peut les représenter sur k bits. Lorsque le premier entier supérieur ou égal à $2^k - 1$ est rencontré, on émet la séquence $1 \dots 1$ (k fois le bit 1) et on continue en codant les entiers sur $k + 1$ bits.

Donc si le code 10 se trouve au début de la liste des codages, il prend 8 bits d'espace mais après le premier nombre plus grand que 255, il prend 9 bits etc.

Dans la suite on s'intéresse à la compression d'un texte écrit en caractères **ASCII** et à la place mémoire utilisée.

Q.6 Écrire une fonction `val espace : int list -> int` qui étant donnée une liste d'entiers produite par l'algorithme de compression calcule *en octets* l'espace nécessaire pour stocker cette liste en utilisant le codage décrit ci-dessus.

```
# espace [10;255;10];;
- : int = 5
# espace [10;255;10;10;511];;
- : int = 8
```

3. 62 bits sur machines 64 bits

Cette fonction est utile pour calculer le *taux de compression* : le quotient de la taille mémoire du texte codé sur le nombre de caractères du texte initial (pour un texte écrit en ASCII).

Q.7 Écrire une fonction `bin_of_int : int -> int -> int list` qui prend pour argument deux entiers k et n et renvoie une liste de longueur k représentant l'entier n en binaire *little-endian* (bit de poids faible en tête). Cette liste sera tronquée (il manquera les bits de poids forts) si l'écriture binaire de n comporte plus de k chiffres.

```
# bin_of_int 4 15;;
- : int list = [1; 1; 1; 1]
# bin_of_int 4 16;;
- : int list = [0; 0; 0; 0]
# bin_of_int 5 16;;
- : int list = [0; 0; 0; 0; 1]
```

Q.8 En déduire une fonction `code : int list -> int list list` qui prend pour argument une liste d'entiers comme celle produite par la fonction `compresse` et retourne la liste binaire correspondante (en little-endian)

```
# code [10;255;10];;
- : int list list =
[[0; 1; 0; 1; 0; 0; 0; 0]; [1; 1; 1; 1; 1; 1; 1; 1];
 [1; 1; 1; 1; 1; 1; 1; 1; 0]; [0; 1; 0; 1; 0; 0; 0; 0; 0]]
```