Codage de Huffman

Durée : 2 heures

Le *codage* d'un ensemble de données consiste à représenter ces dernières par une suite de bits valant 0 ou 1. Lorsque chacune des données à la même fréquence d'apparition, on utilise un codage « uniforme » qui utilise le même nombre de bits pour chacune des données représentables. En revanche, lorsque la répartition des données à coder n'est pas uniforme, on peut utiliser des codages utilisant un nombre variable de bits en fonction de la donnée représentée (un code court étant associé aux données les plus fréquentes). Ce type de codage, qualifié de *stochastique*, est couramment utilisé pour compresser des données, c'est-à-dire réduire l'espace occupé par une information connue a priori.
Nous allons étudier une technique de construction d'un codage minimal due à Huffman. Nous appliquerons cette technique à la compression et la décompression d'une suite de caractères dont le contenu est connu a priori.

Partie I. Codage d'une suite de caractères

Dans tout le problème, on considère un alphabet Σ constitué de p caractères (avec $p \geqslant 1$), et on cherche à coder une suite de n caractères $s = s_1 s_2 \cdots s_n$ avec $s_i \in \Sigma$, $1 \leqslant i \leqslant n$. Nous appelons *nombre d'occurrences* d'un caractère $u \in \Sigma$ le nombre de fois que u apparaît dans la suite s. Cet entier est noté $\mathrm{Occ}(u)$ par la suite.

Une *clef de codage binaire* de Σ est une application injective qui associe à chaque caractère de Σ une suite non vide de valeurs binaires (0 ou 1). Par exemple, les trois applications ci-dessous sont des clefs de codage possibles pour $\Sigma = \{a, b, c, d\}$.

$$\begin{array}{lllll} c_1 : & a \mapsto \texttt{00}, & b \mapsto \texttt{010}, & c \mapsto \texttt{011}, & d \mapsto \texttt{1} \\ c_2 : & a \mapsto \texttt{00}, & b \mapsto \texttt{01}, & c \mapsto \texttt{10}, & d \mapsto \texttt{11} \\ c_3 : & a \mapsto \texttt{0}, & b \mapsto \texttt{01}, & c \mapsto \texttt{1}, & d \mapsto \texttt{10} \end{array}$$

La seconde clef est dite *uniforme* car elle utilise le même nombre de bits pour chaque caractère.

Une clef de codage permet de coder une suite de caractères par une suite de bits en remplaçant chaque caractère de la suite par son code. Par exemple, si nous considérons la suite *abcd*, sont code est **000100111** selon c_1, et **00011011** selon c_2, soient des codages de longueurs respectives 9 et 8. En revanche, si nous considérons la suite *dadbcd*, son code est **10010100111** selon la c_1 et **110011011011** selon c_2, soient respectivement 11 et 12 bits. La première clef de codage semble donc plus adaptée aux suites contenant un plus grand nombre de *d* que de *a*, *b* ou *c*.

Dans la suite du problème, un caractère de Σ est représenté par le type de base *char*. Une suite de caractères est représentée par le type *suite* équivalent à une liste de caractères. Un code binaire est représenté par le type *code* équivalent à une liste de booléens. Une clef de codage qui associe à un caractère son code binaire est représentée par le type *clef* équivalent à une liste de paires caractère/code binaire.

```
type suite == char list ;;
type code == bool list ;;
type clef == (char * code) list ;;
```

Question 1.

a) Écrire en Caml une fonction **coder** de type *suite –> clef –> code* telle que **coder s c** renvoie le code de la suite de caractères s selon la clef de codage c. Nous supposerons que la clef c contient tous les caractères de la suite s.

Indication. On pourra commencer par rédiger une fonction de type *char –> clef –> code* qui associe à tout caractère son codage selon c.

b) Le *coût* du codage de la suite s selon la clef de codage c est la quantité $\mathrm{C}(s) = \sum_{u \in \Sigma} \mathrm{Occ}(u) \times |c(u)|$, où $|c(u)|$ désigne le nombre de bits utilisés pour coder le caractère u.
Donner une estimation de la complexité temporelle dans le pire des cas de la fonction **coder** en fonction des entiers n, p et de la quantité $\mathrm{C}(s)$.

c) Écrire en Caml une fonction **cout** de type *suite –> clef –> int* telle que **cout s c** renvoie le coût du codage de s selon c.

Décodage

Lé définition précédente est trop générale pour permettre une opération de décodage simple. Par exemple, le code **0110** peut être obtenu avec la clef c_3 à partir des mots *acca*, *acd*, *bca*, *bd*.
Notons $\mathcal{I}$ l'image de Σ selon la clef de codage ($\mathcal{I}$ contient donc l'ensemble des valeurs de codes possibles). Une clef de codage est dite *séparable* lorsqu'elle vérifie la propriété suivante :

$$\forall b = b_1 b_2 \cdots b_r \in \mathcal{I} \text{ (avec } b_i \in \{0,1\}), \quad \forall k \in [\![1, r-1]\!], \quad b_1 b_2 \cdots b_k \notin \mathcal{I}.$$

Autrement dit, le code d'un caractère n'est jamais le préfixe d'un code d'un autre caractère. Les codes c_1 et c_2 sont séparables, mais pas c_3.

Question 2.

a) Écrire en Caml une fonction **prefixe** de type *code –> code –> bool* telle que **prefixe b1 b2** renvoie **true** lorsque b_1 est préfixe de b_2 ou b_2 préfixe de b_1, et **false** dans le cas contraire.

b) En déduire une fonction **separable** de *clef –> bool* telle que **separable c** renvoie **true** lorsque c est un code séparable, et **false** sinon.

c) Donner en le justifiant la complexité temporelle de la fonction **separable** en fonction de l'entier p et de la taille maximale m d'un code présent dans c.

Une clef de codage est *optimale* lorsqu'elle vérifie la propriété suivante :

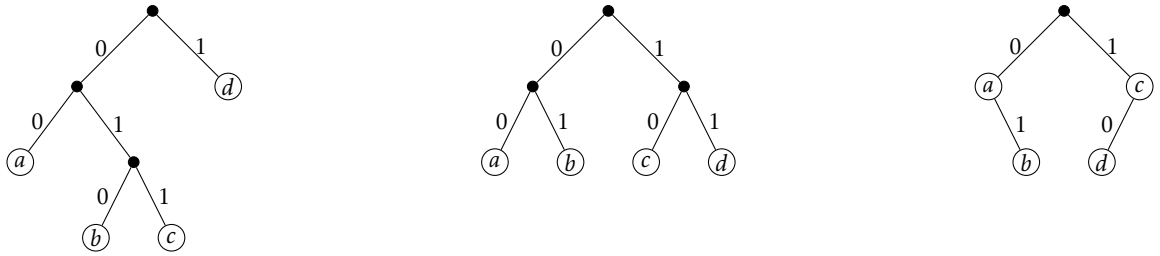
$$\forall b = b_1 b_2 \cdots b_r \in \mathcal{I} \text{ (avec } b_i \in \{0,1\}), \quad \forall k \in [\![1, r]\!], \quad \exists b' = b'_1 b'_2 \cdots b'_s \in \mathcal{I} \;\Big|\; b_1 = b'_1,\ b_2 = b'_2, \ldots, b_{k-1} = b'_{k-1} \text{ et } b_k \neq b'_k.$$

Cette propriété indique de le k^{e} bit permet de distinguer les codes $b_1 b_2 \cdots b_r$ et $b'_1 b'_2 \cdots b'_s$.

Un codage c est *minimal* pour une suite de caractères s donnée lorsqu'il permet de minimiser la quantité $\mathrm{C}(s)$.
L'algorithme que nous allons étudier dans la partie suivante construit une clef de codage séparable et optimale, assurant un codage minimal pour une suite de caractères donnée.

Partie II. Arbre de Huffman

La clef de codage d'un ensemble de caractères peut être représentée par un arbre binaire dont certains nœuds sont étiquetés par un caractère et les arêtes par la valeur d'un bit de code. Par exemple, les trois codages c_1, c_2 et c_3 sont respectivement représentés par les trois arbres suivants :



Nous ne considérerons que des arbres pour lesquels toutes les feuilles sont étiquetées par un caractère.

Question 3. On rappelle qu'un arbre binaire strict est un arbre dans lequel tout nœud interne possède exactement deux fils.

a) Montrer que la clef de codage associée à un arbre binaire strict est séparable si et seulement si seules les feuilles de cet arbres sont étiquetées par des caractères.

b) Montrer que la clef de codage associée à un arbre binaire strict est optimale.

Compte tenu du résultat précédent, on choisit pout représenter des arbres permettant de construire une clef de codage optimale séparable le type Caml suivant :

```
type arbre = Vide | Feuille of char * int | Noeud of arbre * arbre ;;
```

Chaque feuille est étiquetée par un couple $(u, \mathrm{Occ}(u))$ constituée d'un caractère de Σ ainsi que du nombre d'occurrences de celui-ci dans la suite s. Nous supposerons par la suite qu'un même caractère ne figure pas dans plusieurs feuilles distinctes. Un tel arbre est appelé un *arbre de Huffman*.

Question 4. Le nombre global des occurrences des différents caractères qui figurent dans un arbre de Huffman a est la somme des nombres d'occurrences de toutes les feuilles de cet arbre ; cette quantité est nottée $\mathrm{Occ}(a)$.
Écrire en Caml une fonction **`nombre`** de type *arbre –> int* telle que **`arbre a`** renvoie le nombre d'occurrences $\mathrm{Occ}(a)$ des caractères figurant dans l'arbre a.

Codage de Huffman

L'algorithme de Huffman exploite la connaissance du contenu de la suite à coder pour générer une clef de codage minimale. Son principe repose sur la construction de l'arbre de Huffman minimal [1] par transformations successives d'un arbre de Huffman quelconque. L'algorithme découle de l'idée naturelle suivante : *pour minimiser le nombre de bits utilisés par le codage d'une suite de caractères, il faut associer le plus petit nombre de bits aux caractères les plus fréquents et le plus grand nombre de bits aux caractères les moins fréquents*.
La longueur d'un code correspond à la profondeur du caractère associé dans l'arbre. Pour optimiser les codes, il faut donc que les caractères les plus rares étiquettent les feuilles les plus profondes de l'arbre et que les caractères les plus nombreux étiquettent les feuilles les moins profondes. Réduire le coût d'un codage consiste donc à échanger une des feuilles les plus profondes de l'arbre avec un sous-arbre dont la profondeur et le nombre d'occurrences sont strictement plus petits.
Pour éviter les nombreux parcours d'arbres intermédiaires non minimaux pour rechercher les candidats à un échange, l'algorithme de Huffman construit directement l'arbre de Huffman minimal. Pour cela, il manipule un ensemble de sous-arbres de Huffman minimaux partiels. Cet ensemble contient initialement l'ensemble des feuilles étiquetées par les caractères de Σ. Chaque étape de l'algorithme remplace les deux sous-arbres g et d contenant les plus petites occurrences par un arbre de Huffman dont les fils sont g et d. l'algorithme s'arrête lorsque l'ensemble ne contient plus qu'un seul arbre qui est alors l'arbre de Huffman minimal de l'ensemble des caractères de la suite.

Question 5. Appliquer cet algorithme pour construire l'arbre de Huffman correspondant à l'exemple *dadbcd* en détaillant chaque ensemble intermédiaire.

Preuve de minimalité

Question 6. Considérons un arbre de Huffman H. L'arbre H′ est obtenu en permutant les feuilles étiquetées par les caractères u et v dans H.

a) Calculer la différence, notée Δ, entre le coût de H′ et celui de H en fonction des occurrences et profondeurs de u et v.

b) Montrer que dans un arbre de Huffman minimal, les feuilles de profondeur maximale contiennent les caractères d'occurrence minimale.

Question 7. Considérons un arbre de Huffman H possédant deux feuilles ayant même père n et étiquetées par deux caractères u et v d'occurrences minimales. L'arbre H′ est obtenu en enlevant les feuilles u et v dans H et en considérant que n est une feuille d'occurrence $\mathrm{Occ}(n) = \mathrm{Occ}(u) + \mathrm{Occ}(v)$.

a) Montrer que si H′ est un arbre de Huffman minimal, il en est de même de H.

b) En déduire que l'arbre de Huffman construit par l'algorithme de Huffman est minimal.

Partie III. Tri par insertion

L'algorithme de Huffman extrait de l'ensemble les sous-arbres contenant les plus petits nombres d'occurrences. Pour cela, nous exploiterons une liste de sous-arbres triée suivant le nombre d'occurrences. Cette liste sera triée par un algorithme de tri par insertion.

Question 8. Écrire de Caml une fonction **`comparer`** de type *arbre –> arbre –> bool* tel que **`comparer a1 a2`** renvoie la valeur **`true`** si $\mathrm{Occ}(a_1)$ est strictement plus petit que $\mathrm{Occ}(a_2)$ et la valeur **`false`** sinon.

Question 9. Écrire en Caml une fonction **`inserer`** de type *arbre –> arbre list –> arbre list* telle que **`inserer a l`** renvoie la liste d'arbres obtenue en insérant l'arbre a dans la liste l en respectant l'ordre croissant du nombre d'occurrences. Nous supposerons que la liste l respecte l'ordre croissant du nombre d'occurrences.

Question 10. Écrire en Caml une fonction **`trier`** de type *arbre list –> arbre list* tel que **`trier l`** renvoie une liste d'arbres triée selon l'ordre croissant des nombres d'occurrences et composée des mêmes arbres que l.

1. C'est-à-dire associé à une clef de codage minimale.

Partie IV. Construction des codes de Huffman

Pour calculer le code minimal pour chaque caractère de la suite, il faut déterminer l'ensemble des caractères distincts et le nombre d'occurrences de chaque caractère, puis construire l'arbre par fusion successive des éléments de cet ensemble.

Construction de la liste d'occurrences

Question 11. Écrire en Caml une fonction **`ajouter`** de type *char –> arbre list –> arbre list* telle que **`ajouter u l`** renvoie une liste de feuilles étiquetées par les mêmes caractères que l telle que si l contenait u alors son occurrence est augmentée de 1 et si l ne contenait pas u le résultat contient u avec l'occurrence 1. Nous supposerons que l est une liste de feuilles dans laquelle u figure au plus une fois.

Question 12. Écrire de Caml une fonction **`compter`** de type *suite –> arbre list* telle que **`compter s`** renvoie une liste de feuilles étiquetées par les mêmes caractères que s et par les occurrences de ces caractères dans s.

Construction de l'arbre

L'algorithme de Huffman de construction de l'arbre minimal consiste à remplacer les deux sous-arbres ayant les plus petits nombre d'occurrences par le nœud ayant pour fils ces deux sous-arbres tout en préservant l'ordre de la liste triée.

Question 13. Écrire en Caml une fonction **`fusionner`** de type *arbre list –> arbre* telle que **`fusionner l`** renvoie un arbre de Huffman contenant les mêmes caractères que la liste l et correspondant au codage minimal de la suite. On supposera que l est une liste non vide de feuilles triées selon le nombre d'occurrences des caractères.

Question 14. En déduire une fonction **`huffman`** de type *suite –> arbre* telle que **`huffman s`** renvoie un arbre de Huffman contenant les mêmes caractères que la suite s et correspondant au codage minimal de la suite.

Partie V. Codage et décodage d'une suite

L'étape précédente a permis de construire un arbre dont les chemins qui conduisent à chaque caractère sont étiquetées implicitement (0 pour le fils gauche et 1 pour le fils droit) par le code associé à ce caractère. Pour éviter les multiples parcours de l'arbre lors du codage d'une suite de caractères, il faut construire une clef de codage à partir de l'arbre qui associe à chaque caractère son code puis parcourir la suite pour remplacer chaque caractère par son code en utilisant la fonction **`coder`** proposée dans la question 1.

Question 15. Écrire en Caml une fonction **`construire`** de type *arbre –> clef* telle que **`construire a`** renvoie une clef de codage associant à chaque caractère contenu dans l'arbre de Huffman a son code, c'est-à-dire une suite de valeurs binaires correspondant au chemin suivi dans l'arbre pour atteindre ce caractère.

Le principal défaut du codage de Huffman est la nécessité de transmettre la clef de codage avec chaque suite codée. En effet, il faut reconstruire l'arbre de Huffman propre à chaque suite codée pour décoder celle-ci par un parcours de l'arbre en suivant chaque code de la suite codée.

Question 16. [difficile]

Écrire en Caml une fonction **`reconstruire`** de type *clef –> arbre* telle que **`reconstruire c`** renvoie un arbre de Huffman obtenu à partir de la clef de codage c, c'est-à-dire contenant les mêmes caractères que c placés dans l'arbre selon le code associé au caractère c. Puisqu'il n'est utile qu'au moment de la construction initiale de l'arbre, l'occurence de chaque caractère sera arbitrairement fixée à 0.

Question 17. Écrire en Caml une fonction **`decoder`** de type *code –> arbre –> suite* telle que **`decoder b a`** renvoie une suite de caractères correspondant au décodage de la suite de valeurs binaires contenue dans b. Ce décodage est effectué par un parcours de l'arbre de Huffman a en suivant les chemins correspondants aux valeurs binaires contenues dans b.

♦ ♦
♦