Huffmann

Définitions

— Si $\mathcal{A}$ est un ensemble fini (dit *alphabet*), on appelle ensemble des mots sur $\mathcal{A}$ et l'on note $\mathcal{A}^*$ l'ensemble des suites finies d'éléments de $\mathcal{A}$. Si $a_1, \dots, a_n \in \mathcal{A}$, l'élément $(a_1, \dots, a_n)$ de $\mathcal{A}^*$ sera simplement noté $a_1 a_2 \dots a_n$.
— On notera $|\mathcal{A}|$ le cardinal de $\mathcal{A}$, que l'on supposera systématiquement supérieur ou égal à 2.
— Si $u = a_1 \dots a_n$, où $a_1, \dots, a_n \in \mathcal{A}$, la longueur n de u sera notée $|u|$.
— Pour $a \in \mathcal{A}$ et $u \in \mathcal{A}^*$, on notera $|u|_a := \text{Card}\{i \in [1 \dots |u|] \mid u_i = a\}$. Autrement dit, $|u|_a$ désigne le nombre d'occurrences de la lettre a dans le mot u.
— Si $a_1, \dots, a_n, b_1, \dots, b_p \in \mathcal{A}$ et si l'on a $u = a_1 \dots a_n$ et $v = b_1 \dots b_p$, on notera $u \cdot v$ le mot $a_1 \dots a_n b_1 \dots b_p$, appelé *concaténation* de u et de v.
On notera souvent uv pour $u \cdot v$.
— On note ε l'unique élément de $\mathcal{A}^*$ de longueur 0 (mot vide). Pour tout u, on a $u \cdot \varepsilon = \varepsilon \cdot u = u$.
— Si u et v sont des éléments de $\mathcal{A}^*$, on dit que u est un *préfixe* de v si $v = u \cdot w$ où $w \in \mathcal{A}^*$. Si $w \neq \varepsilon$, on dit que u est un *préfixe strict* de v.
— On appelle *code binaire* sur $\mathcal{A}$ une application f injective de $\mathcal{A}$ dans $\{0,1\}^* \setminus \{\varepsilon\}$: à chaque lettre de $\mathcal{A}$, on associe une suite finie (non vide) de 0 et de 1.
Tous les codes considérés dans le sujet seront des codes binaires (et ce ne sera pas précisé à chaque fois).
— Si f est un code binaire sur $\mathcal{A}$, son *extension* $\overline{f}$ (que l'on notera souvent f pour alléger) est l'application :

$$\begin{array}{rccc} \overline{f} : & \mathcal{A}^* & \longrightarrow & \{0,1\}^* \\ & a_1 \dots a_n & \longmapsto & f(a_1) \cdot \dots \cdot f(a_n) \end{array}$$

Autrement dit, le codage d'un mot est obtenu en concaténant les codages des caractères qui le composent.
— Un code binaire est dit *uniquement déchiffrable* si son extension est injective, *ambigu* sinon.
— Un code binaire f est dit *préfixe* (on dit aussi *sans préfixe*, ce qui est quelque part plus logique) s'il n'existe pas de couple (a, b) d'éléments de $\mathcal{A}$ tels que $a \neq b$ et $f(a)$ soit un préfixe de $f(b)$.
— Un code binaire f est dit *à longueur fixe* si tous les $f(a)$ pour $a \in \mathcal{A}$ sont de même longueur, *à longueur variable* sinon.

1 Exemples et premières propriétés

1. On considère dans cette question l'alphabet $\mathcal{A} = \{a, b, c\}$ et le code f défini par $f(a) = 01$, $f(b) = 010$ et $f(c) = 1$. Calculer $\overline{f}(abc)$ et $\overline{f}(bca)$. Le code f est-il préfixe ? uniquement déchiffrable ?
2. Donner un exemple de code non préfixe uniquement déchiffrable. On justifiera brièvement le caractère uniquement déchiffrable du code. En représentant un élément de $\{0,1\}^\star$ par une liste de booléen, écrire une fonction

```
decode (m : bool list) : char list
```

permettant de décoder de tels messages.
3. Soit $\mathcal{A}$ un alphabet et $u, u', v, v' \in \mathcal{A}^*$, on suppose que $uu' = vv'$. Montrer que u est un préfixe de v ou v est un préfixe de u.
4. Montrer que tout code préfixe est uniquement déchiffrable.

2 Arbre d'un code préfixe

2.1 Arbre binaire associé à un code préfixe

Dans cette partie (et uniquement dans cette partie), on considère des arbres binaires dont :
— chaque feuille est étiquetée par un caractère (type `char` en OCaml) ;
— chaque nœud interne a soit un, soit deux fils, et ne porte pas d'étiquette.
On utilise le type OCaml suivant :

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

1. Le type OCaml ci-dessus permet d'avoir des nœuds de la forme `Noeud (Vide, Vide)` (nœud interne n'ayant aucun fils) qui ne sont pas autorisés par la définition. Écrire une fonction `bien_forme : arbre -> bool`, qui renvoie `true` si et seulement si l'arbre reçu en argument ne comporte aucun nœud de ce type.

Quand on représentera graphiquement un tel arbre, on omettra les fils `Vide` :

```
let t1 =
  Noeud (
      Noeud (
          Feuille 'a',
          Noeud (Feuille 'b', Vide)),
      Noeud (
          Noeud (Vide, Feuille 'c'),
          Vide))
```

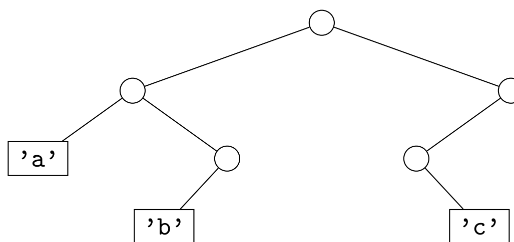


FIGURE 1 – Définition en OCaml et représentation graphique de l'arbre t_1.

Dans un arbre t, on définit l'*adresse* $add(x)$ d'un nœud x (interne on non) de la manière suivante :
— l'adresse de la racine est ε (le mot vide) ;
— si x est le fils gauche de y, alors $add(x) = add(y)0$;
— si x est le fils droit de y, alors $add(x) = add(y)1$.

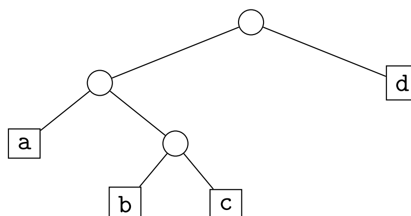
Soit $\mathcal{A}$ un alphabet de cardinal n. À un arbre binaire t ayant exactement n feuilles non vides étiquetées par les n lettres de $\mathcal{A}$, on associe un code préfixe f_t de la façon suivante :

pour toute lettre $a \in \mathcal{A}$, $f_t(a)$ est l'adresse de l'unique feuille de t étiquetée par a.

En reprenant l'arbre t_1 donné plus haut, on obtient alors :

Lettre	Code
a	00
b	010
c	101

1. Déterminer le code associé à l'arbre suivant :



Inversement, tout code préfixe sur $\mathcal{A}$ peut être représenté par un arbre dont les feuilles sont exactement les lettres de $\mathcal{A}$.

2. Dessiner l'arbre associé au code suivant :

Lettre	Code
a	010
b	011
c	001
d	10
e	11

2.2 Poids d'un code préfixe

Considérons un mot $s = s_1 \dots s_n$ sur un alphabet $\mathcal{A}$; *on supposera toujours que toutes les lettres de $\mathcal{A}$ apparaissent au moins une fois dans s* (ou, ce qui revient au même, que l'on restreint $\mathcal{A}$ pour ne garder que les lettres apparaissant dans s). Ce mot correspond en fait à la totalité du texte à traiter : pour nous, les espaces et les retours à la ligne sont

des caractères comme les autres. On cherche à compresser ce texte en trouvant un code préfixe f pour lequel l'image de s peut être stockée sur un petit nombre de bits. On définit donc le *poids* d'un code préfixe f, que l'on note $w_s(f)$, comme la longueur totale de l'image du mot s par le code :

$$\begin{aligned} w_s(f) &:= |f(s)| \\ &= \sum_{i=1}^{n} |f(s_i)| \\ &= \sum_{a \in \mathcal{A}} |s|_a \, |f(a)| \end{aligned}$$

Un code préfixe f est dit *optimal pour un mot* s si $w_s(f)$ est minimal parmi tous les codes préfixes. On notera $opt(s)$ le poids d'un code préfixe optimal pour s.

1. On définit $opt_{fixe}(s)$ comme le poids minimal d'un code préfixe *à longueur fixe* pour le mot s. Exprimer $opt_{fixe}(s)$ en fonction de $|s|$ et de $|\mathcal{A}|$.
2. Donner un exemple de mot s sur l'alphabet $\{a, b, c, d\}$ pour lequel on a $opt(s) < opt_{fixe}(s)$ (on justifiera cette inégalité).
3. Montrer que l'arbre associé à un code préfixe optimal ne contient aucun nœud n'ayant qu'un seul fils (non vide).

Comme on s'intéresse dans la suite à la construction d'un code optimal, on peut donc simplifier le type de nos arbres pour se limiter aux arbres binaires entiers (où chaque nœud interne a exactement deux fils). Le type obtenu, que nous utiliserons dans toute la suite du problème, est alors :

```
type arbre_code =
  | F of char
  | N of arbre_code * arbre_code
```

2.3 Fonctions de codage et décodage

On choisit les types suivants pour les différents objets :

— le texte que l'on souhaite compresser (le mot s) est représenté par une chaîne de caractères (type `string`) ;
— l'alphabet $\mathcal{A}$ est constitué des caractères ASCII (type `char`) apparaissant au moins une fois dans s ;
— le texte compressé (c'est-à-dire le résultat $f(s)$ de l'application du code au texte s de départ) est une suite de zéros et de uns ; il sera représenté comme une liste de booléens, où `false` correspond à 0 et `true` à 1 :

```
type bitstream = bool list
```

— le code f aura deux représentations :
  — une de type `arbre_code` qui sera utilisée pendant le décodage (et aussi la construction du code en fin de problème)
  — une autre utilisée pour l'encodage, détaillée dans la partie 2.3.2.

2.3.1 Décodage

1. Écrire une fonction `decode_caractere : arbre_code -> bitstream -> char * bitstream` prenant en entrée un code préfixe f sous forme d'arbre et le codage $u = f(s)$ d'un certain mot s, et renvoyant le couple (a, u') tel que $u = f(a) \cdot u'$. Autrement dit, cette fonction doit renvoyer le premier caractère du texte décodé et le reste du texte à décoder.
2. On donne la fonction suivante pour convertir une `char list` en `string` :

```
let string_of_char_list u = String.of_seq (List.to_seq u)
```

Écrire une fonction `decode_texte (f : arbre_code) (u : bitstream) : string` prenant un code préfixe f sous forme d'arbre et le codage $u = f(s)$ d'un certain mot s, et renvoyant s.

2.3.2 Encodage

Pour réaliser le codage, un arbre n'est pas très pratique : on préfère avoir un tableau `t` permettant d'obtenir directement le code associé à un caractère. On définit donc :

```
type table_code = bitstream array
```

Une `table_code` sera toujours de longueur 256, et contiendra dans sa case `i` le code du caractère `char_of_int i` ; pour les caractères n'apparaissant pas dans le texte (et n'ayant donc pas de code associé), la case contiendra la liste vide.

Par exemple, le code

Lettre	Code
a	010
b	011
c	00
d	1

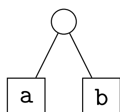
serait représenté par un `t : table_code` avec :
- `t.(0) = ... = t.(96) = t.(101) = ... = t.(255) = []` (car tous ces caractères n'ont pas de code) ;
- `t.(97) = [false; true; false]` (car `int_of_char 'a' = 97`) ;
- `t.(98) = [false; true; true]` (car `int_of_char 'b' = 98`) ;
- `t.(99) = [false; false]` et `t.(100) = [true]`, de même.

1. Écrire une fonction `cree_table : arbre_code -> table_code` permettant d'obtenir la représentation d'un code sous forme de table à partir de sa représentation sous forme d'arbre.
2. Écrire la fonction `encode (t : table_code) (s : string) : bitstream`, qui prend en entrées la table `t` représentant un code préfixe f et le texte `s` et renvoie $f(s)$ sous la forme d'une liste de booléens.

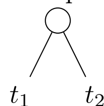
3 Algorithme de Huffman

3.1 Principe de l'algorithme

L'algorithme de Huffman permet de construire un code préfixe optimal pour un mot s donné.
- On calcule $|s|_a$ pour chaque $a \in \mathcal{A}$. On crée une feuille étiquetée a pour chaque a apparaissant dans s, et on crée une liste $q = \left[\left(\boxed{a_1}, |s|_{a_1}\right), \left(\boxed{a_2}, |s|_{a_2}\right), \dots, \left(\boxed{a_p}, |s|_{a_p}\right)\right]$.
- On détermine les deux feuilles a et b ayant les plus petits nombres d'occurrences (ie les deux couples ayant les plus petites deuxièmes composantes), on les sort de la liste et l'on met à la place le couple $(t, |s|_a + |s|_b)$, où t est l'arbre

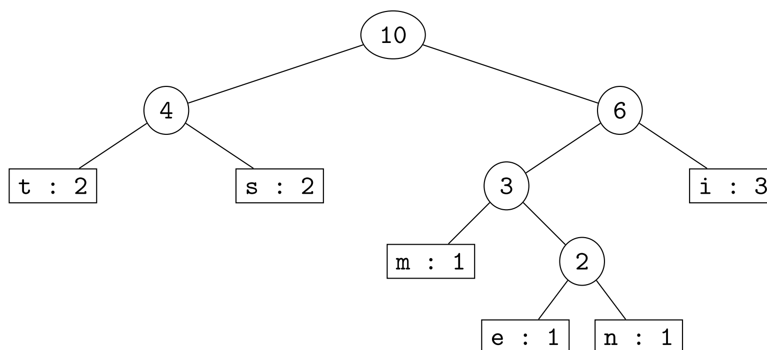


- On recommence l'étape précédente, en prenant les deux couples (t_1, n_1) et (t_2, n_2) ayant les plus petites deuxièmes composantes, et en les remplaçant par le couple $(t, n_1 + n_2)$ où t est l'arbre



  Ici, t_1 et t_2 ne sont pas nécessairement des feuilles.
- On continue jusqu'à ce qu'il n'y ait plus qu'un arbre dans la liste : cet arbre est le code de Huffman associé à s.

1. Vérifier que, appliqué au mot `"intimistes"`, l'algorithme de Huffman produit (ou plutôt peut produire, suivant comment l'on tranche en cas d'égalité) l'arbre :



   Les étiquettes entières des nœuds et des feuilles sont « virtuelles » : elles ont servi à la construction mais ne sont en fait pas stockées dans l'arbre.

2. Pour l'exemple ci-dessus, calculer :
   - le nombre de bits qu'occupe la chaîne de départ ;
   - le poids qu'aurait un code à longueur fixe (où l'on restreint l'alphabet aux caractères effectivement présents) ;
   - le poids du code de Huffman.

   Quelle caractéristique du texte initial le codage de Huffman exploite-t-il pour obtenir un poids inférieur à celui d'un code à longueur fixe ?

3. On considère un mot s sur un alphabet $\mathcal{A} = \{a_0, \dots, a_{n-1}\}$ vérifiant $|s|_{a_i} = 2^i$ pour $0 \leqslant i < n$. Donner (en justifiant) la forme d'un arbre de Huffman possible pour s et montrer que son poids vaut $2^{n+1} - n - 3$.
4. On définit le *facteur de compression* d'un code f pour le mot s comme le quotient $\frac{opt_{fixe}(s)}{w_s(f)}$. En reprenant le mot s de la question précédente, déterminer un équivalent simple de ce facteur de compression pour le code de Huffman quand n tend vers $+\infty$.

3.2 Construction de l'arbre

1. Écrire une fonction `occurrences (s : string) : int array`. Cette fonction prend en entrée une chaîne s et renvoie un tableau t de taille 256 tel que $t.(i)$ contienne le nombre d'occurrences du caractère dont le numéro ASCII est i (c'est-à-dire de `char_of_int i`) dans la chaîne s. On demande une complexité en $O(|s|)$.

```
# let t = occurrences
"So we beat on, boats against the current, borne back ceaselessly into the
   past.";;
(* OCaml affiche le tableau de taille 256, omis ici... *)
# t.(97);;
- : int = 7
(* int_of_char 'a' vaut 97, et il y a sept 'a' dans la chaine donnée en
   argument. *)
```

2. Écrire une fonction `foret (s : string) : (arbre_code * int) list` qui prend une chaîne de caractères et renvoie la liste des `(Feuille c, f)`, où le caractère `c` apparaît f fois dans s. Les caractères n'ayant aucune occurrence dans s seront omis. L'ordre des éléments de la liste n'a pas d'importance.

```
utop[13]> foret "inimity";;
- : (arbre_code * int) list =
[(F 'i', 3); (F 'm', 1); (F 'n', 1); (F 't', 1); (F 'y', 1)]
```

3. Écrire une fonction `huffman (s : string) : arbre_code` qui renvoie un arbre de Huffman associé à la chaîne s.

   On pourra utiliser une structure de données (que vous devriez bien connaître, et pour laquelle vous devriez pouvoir copier-coller du code) adaptée à cette construction.

```
utop[11]> huffman "des dodos font dodo";;
- : arbre_code =
N (N (N (F 's', N (F 'n', F 'e')), N (N (F 'f', F 't'), F ' ')),
 N (F 'd', F 'o'))
```

4. Écrire une fonction `compresse : string -> (arbre_code * bitstream)` qui prend en entrée une chaîne et renvoie le code de Huffman correspondant, sous forme d'arbre, et le texte compressé sous forme de flux binaire.

3.3 Optimalité

Pour démontrer le caractère optimal du code de Huffman, nous allons modifier légèrement nos notations. On remarque que le code de Huffman associé à un mot s ne dépend pas de l'ordre des lettres dans s, et qu'il en est de même pour $opt(s)$: pour un code f, on aura toujours $w_f(edredon) = w_f(ddeeorn)$.

On laisse donc de côté la notion de mot pour se concentrer sur celle d'alphabet, que l'on étend pour inclure les fréquences d'apparition des différentes lettres :

- dans la suite, on appellera *alphabet* un ensemble fini de couples $\mathcal{A} = \{(a_1, n_1), \dots, (a_p, n_p)\}$ où les a_i sont des lettres (deux à deux distinctes) et les n_i des entiers vérifiant $1 \leqslant n_1 \leqslant n_2 \leqslant \dots \leqslant n_p$ (n_i représente le nombre d'occurrences de a_i dans le mot sous-jacent) ;
- on définit $h(\mathcal{A})$ comme l'arbre de Huffman associé à un mot constitué de n_1 lettres a_1, …, n_p lettres a_p et $w_h(\mathcal{A})$ son poids ;
- on définit également $opt(\mathcal{A})$ comme le poids minimal d'un code pour ce même mot ;
- l'objectif est donc de montrer que $opt(\mathcal{A}) = w_h(\mathcal{A})$.

On rappelle qu'on suppose systématiquement $|\mathcal{A}| \geqslant 2$.

1. Montrer qu'on peut toujours trouver un code optimal pour $\mathcal{A}$ dans lequel la feuille étiquetée a_1 a pour sœur la feuille étiquetée a_2.
   On rappelle que l'on a numéroté les lettres de manière à avoir $n_1 \leqslant n_2 \leqslant \dots \leqslant n_p$.
2. Pour un alphabet $\mathcal{A}$ vérifiant $|\mathcal{A}| \geqslant 3$, on définit $\mathcal{A}' = \{(b, n_1 + n_2), (a_3, n_3), \dots, (a_{|\mathcal{A}|}, n_{|\mathcal{A}|})\}$, où b est une nouvelle lettre, distincte de toutes les autres.
   Montrer que $opt(\mathcal{A}) \geqslant opt(\mathcal{A}') + n_1 + n_2$.

3. Montrer que le code construit par l'algorithme de Huffman est optimal.