Algorithmique TP 19 : Codage de Huffman

On désire compresser un texte composé de caractères Ascii (code compris entre 0 et 255). Un caractère est classiquement représenté par une chaîne de 8 bits. L'idée de Huffman est de coder les caractères par des chaînes de bits de longueurs variables : les caractères fréquents seront représentés par des chaînes courtes, les caractères rares par des chaînes plus longues. Afin que le codage soit correct, la propriété suivante doit être vérifiée : le code d'un caractère ne doit pas être le préfixe d'un autre code (sinon la reconnaissance serait impossible).

Une façon simple de représenter un codage qui vérifie cette propriété est d'utiliser un arbre binaire tel que :

- les feuilles sont étiquetées par les caractères codés ;
- l'arête d'un fils gauche est étiquetée 0, l'arête d'un fils droit, 1 ;
- le code d'un caractère (une feuille) est le chemin depuis la racine de l'arbre.

Par exemple, pour le codage de `a` par `000`, `b` par `11`, `c` par `01`, `d` par `001` et `e` par `10`, on obtient l'arbre de la figure 1.

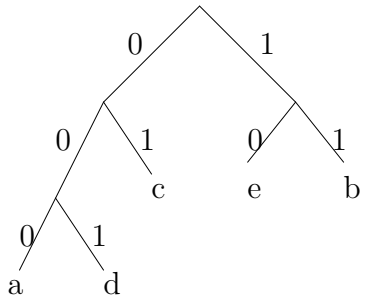

Fig. 1 – Arbre binaire de codage

On suppose que pour chaque caractère $c \in \mathcal{C}$, on connaît sa probabilité d'apparition $P(c)$. Si on note $l(c)$ la longueur (nombre de bits) du codage du caractère c, le problème est de trouver un codage tel que $\Sigma_{c\in\mathcal{C}} P(c)l(c)$ soit minimal. L'algorithme de Huffman est une technique pour trouver un tel codage optimal.

Algorithme de Huffman

L'algorithme travaille sur une forêt d'arbres, c'est-à-dire un ensemble d'arbres. Chaque arbre, dont les feuilles sont des caractères, possède un poids. Un arbre réduit à une feuille possède comme poids la probabilité du caractère c de cette feuille ; un tel arbre est noté $f[c, P(c)]$. Un arbre non réduit à une feuille a un poids égal à la somme des poids de ses deux fils ; il est noté $n[g, d, p]$ où g et d sont les fils gauche et droit du nœud et p son poids.

L'algorithme est le suivant :

$\mathcal{E} := \{f[c, P(c)]/c \in \mathcal{C}\}$;
TantQue $\mathcal{E}$ n'est pas un singleton
  n_1 := nœud de plus faible poids ;
  n_2 := nœud de deuxième plus faible poids ;
  $\mathcal{E} := \mathcal{E} \setminus \{n_1, n_2\} \cup \{n[n1, n_2, poids(n1) + poids(n2)]\}$;

On initialise la forêt d'arbres $\mathcal{E}$ avec l'ensemble des caractères. À chaque étape, les deux arbres (n_1 et n_2) de plus faibles poids sont supprimés de la forêt et le nœud ayant pour fils ces deux arbres est ajouté à la forêt. Ceci est répété jusqu'à obtenir une forêt réduite à un unique arbre.

Exemple : avec $\mathcal{C} = \{a, b, c, d, e\}$ et les probabilités suivantes,

caractère	a	b	c	d	e
probabilité	0.15	0.2	0.3	0.15	0.2

on obtient l'arbre de la figure 1 via les cinq étapes de la figure 2.

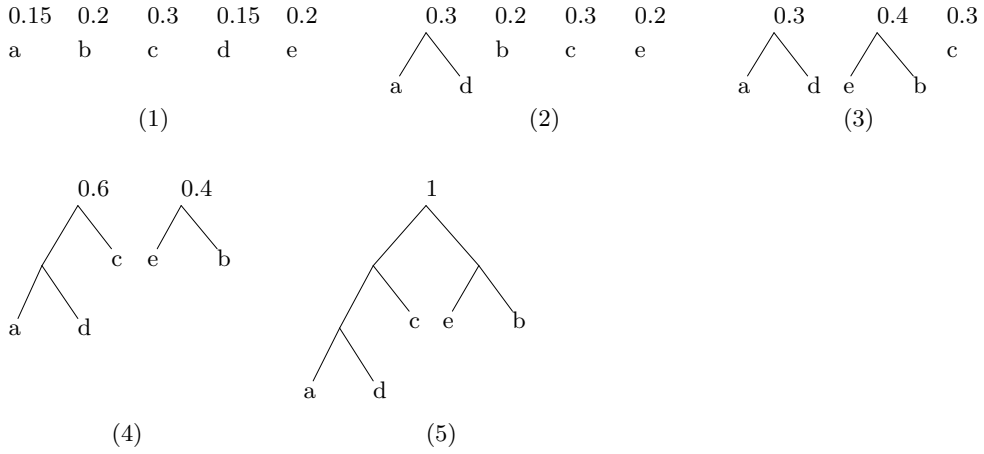

Fig. 2 – Étapes de la construction de l'arbre de Huffman

Mise en œuvre

Une solution pour implémenter l'algorithme de Huffman est d'utiliser une liste triée pour représenter la forêt $\mathcal{E}$.

1. Écrire une fonction d'insertion dans une liste triée. La fonction de comparaison de deux éléments est passée en argument.

```
#insertion (fun (x,_) (y,_) -> x < y) (2,'b') [(1,'u');(2,'d');(3,'t')];;
 - : (int * char) list = [(1, 'u'); (2, 'd'); (2, 'b'); (3, 't')]

#insertion (fun x y -> x > y) 'b' ['u';'t'; 'd'];;
- : char list = ['u'; 't'; 'd'; 'b']
```

2. Écrire le type `tree` pour représenter les arbres de Huffman.
3. Écrire la fonction `huffman` qui transforme une forêt en un arbre.

```
#huffman;;
 - : (char * float) list -> tree = <fun>
```

4. Écrire la fonction `codes` qui prend en argument un arbre de Huffman et qui calcule la liste des couples caractère-code.
5. Écrire la fonction `proba` prenant en argument une chaîne de caractères et retournant en résultat une liste de couple caractère-probabilité d'apparition. Indication : utiliser les fonction `Char.code` et `Char.chr` et un tableau de taille 256.
6. Compresser une chaîne de caractères de votre choix et calculer le gain de place (le codage standard d'un caractère nécessite 8 bits). Vous pourrez utiliser la fonction `List.assoc`.

```
#let m = "Abracadabra";;
 val m : string = "Abracadabra"

#let c = compresse m;;
val c : string = "01100101110101110011100010111"

#float (String.length c) /. (8. *. float (String.length m));;
- : float = 0.318181818181818177
```