Structure de données et algorithmes

Projet 2 : Structures de donnees

LEWIN Sacha

MAKEDONSKY Aliocha

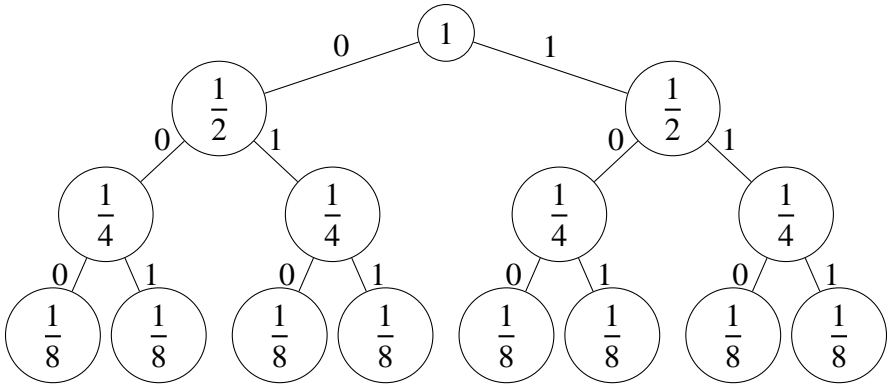
Ingénieur civil 2^{e} bachelier
Année académique 2019-2020

1 Structure des arbres sur base des distributions de fréquences

Dans ces arbres construits avec l'algorithme de *Huffman*, il y a toujours un nombre de feuilles équivalent au nombre d'éléments qu'on veut placer dans l'arbre (ce qui est logique puisque chaque feuille représente un élément et sa fréquence). Chaque parent contient une fréquence égale à la somme des fréquences contenues dans ses 2 fils. Par conséquent, la racine de l'arbre contient une fréquence de $1 = 100\%$. Parcourir l'arbre de la racine à une feuille nous donne le code binaire de l'élément contenu dans cette feuille (en regardant le bit inscrit sur chacune des branches), le code s'écrivant de gauche à droite en descendant dans l'arbre. Nous prendrons pour les 3 arbres que nous allons construire un alphabet contenant $k = 2^3 = 8$ caractères.

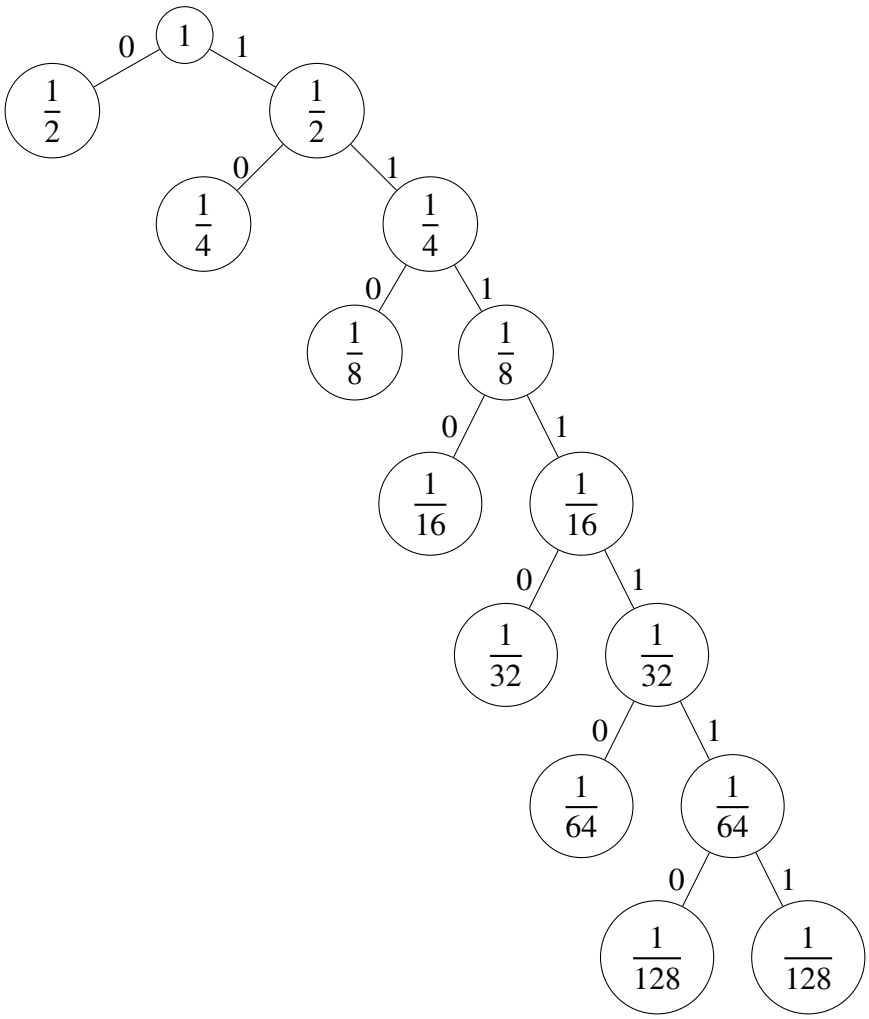
1.a Distribution équiprobable

La fréquence de chaque élément est donnée par $f_i = \frac{1}{k} \forall i \in \{1, ..., k\}$. Ici à chaque fois qu'on fusionne 2 feuilles, le parent aura forcément une fréquence plus grande que toutes celles des autres feuilles puisque toutes les feuilles ont exactement la même fréquence. Donc on fusionne à chaque fois les feuilles 2 par 2, puis une fois que c'est fait on fait pareil avec les parents qu'on fusionne 2 par 2 et ainsi de suite pour arriver au sommet.



1.b Distribution "puissance de deux"

La relation de cette distribution est : $f_i = \frac{1}{2^i} \forall i \in \{1, ..., k-1\}$ et $f_k = f_{k-1}$. La dernière partie de cette relation est là pour pouvoir obtenir une fréquence finale de 1, sinon on tendrait vers 1 mais on ne l'atteindrait jamais. A chaque fois qu'on fusionne les 2 feuilles de plus petite fréquence qui restent dans la liste, on obtient une fréquence égale à la plus petite fréquence restante dans la liste, on doit donc fusionner ces deux dernières, et c'est comme ça jusqu'à arriver à la fréquence finale de 1. Donc à chaque fois on fusionnera la plus petite des fréquences initiales avec la dernière fréquence résultante d'une fusion.

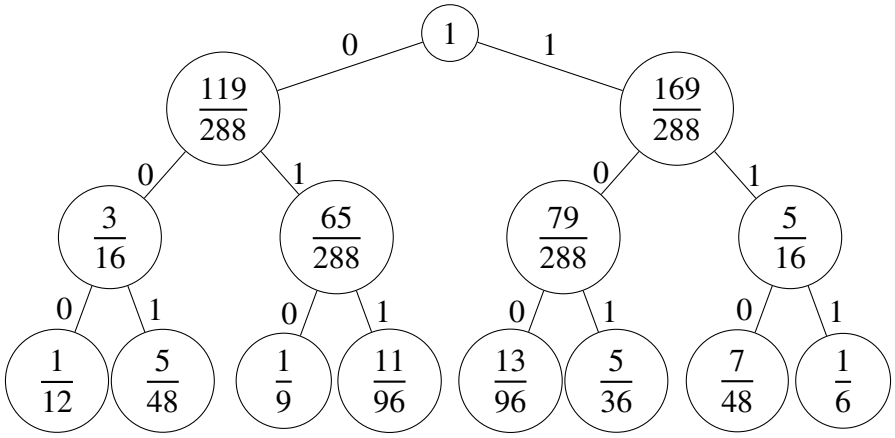


1.c Distribution "epsilon"

On a un arbre de la forme que nous pouvons observer en dessous puisque dés qu'on fusionne les deux feuilles dont les fréquences sont les plus petites, la fréquence résultante (somme des 2 fréquences) devient la plus grande des valeurs restant dans la file. Par conséquent on fusionne d'abord toutes les feuilles initiales 2 par 2, puis la plus petite fréquence sera celle résultant de la fusion des 2 feuilles initiales de plus petites fréquences, la suivante sera celle résultant de la fusion des 2 feuilles initiales suivantes, et ainsi de suite ... Pour avoir cette forme d'arbre il faut que les fréquences initiales soient assez proches les unes des autres. Par exemple ici, le rapport entre la plus grande fréquence initiale et la plus petite est donnée par :

$$\frac{\frac{1}{k}+\frac{1}{3k}}{\frac{1}{k}-\frac{1}{3k}} = \frac{\frac{4}{3k}}{\frac{2}{3k}} = 2$$

et ce peu importe la taille k de l'alphabet considéré.



2 Complexité de l'algorithme de Huffman

Soit un alphabet de taille k.

2.a Cas de la file à priorité avec liste

Dans l'algorithme de Huffman, on fait tourner une boucle k fois, et les différentes actions effectuées sont `pqExtractMin` qui est de complexité $O(1)$, et `pqInsert` qui est, dans le meilleur cas, de complexité $O(1)$ si on rajoute à chaque fois un élément au début de la liste, et $O(k)$ si on doit parcourir toute la liste pour insérer. Donc dans le pire des cas l'algorithme est $O(k^2)$ et dans le meilleur des cas $O(k)$. La complexité de l'algorithme est en fait la complexité habituelle de la liste liée multipliée par le nombre de caractères de l'alphabet.

2.b Cas de la file à priorité avec tas

Ici, on parcourt les k caractères (donc une boucle $O(k)$), et pour chacun, la complexité d'insertion dans un tas binaire minimal est de $O(\log k)$, ce qui résulte donc en une complexité de $O(k \cdot \log k)$

3 Algorithme de décodage

3.a Pseudo-Code de `DECODE`

```
1:  function DECODE(B, T)
2:      currentBit = 1
3:      destination = "New Char Vector"
4:      while currentBit ≤ B.nbBits do
5:          decoded = "Decode byte at currentBit with B, T"
6:          if decoded.character is endCharacter then
7:              break
8:          end if
9:          success = destination.add(decoded.character)
10:         if !success then
11:             return Erreur
12:         end if
13:     end while
14: end function
```

3.b Complexité de `DECODE` avec encodage de Huffman

Soit un alphabet $\mathcal{A}$ de taille k, et un texte de taille originale n.

Pour décoder un texte encodé à l'aide de l'encodage de *Huffman*, il nous faut parcourir l'arbre pour chacun des caractères à décoder. Nous devons donc d'abord parcourir chacun des caractères encodés du texte : $O(n)$. Pour chaque caractère, nous devons parcourir l'arbre et l'avantage de la méthode est de favoriser les meilleurs cas. Dans le meilleur cas, chaque caractère est le plus fréquent avec le plus petit encodage, le parcours de chaque caractère est donc $O(1)$ et on arrive à une complexité $O(n)$. Dans le pire cas c'est $O(n \log k)$ où le caractère est tout en bas, donc le parcours est $O(\log k)$. En moyenne, on sera bien plus bas que $O(\log k)$. En effet, grâce à l'encodage plus petit sur les caractères aux fréquences les plus élevées, on augmente considérablement le nombre de meilleurs cas.

3.c Complexité de `DECODE` avec encodage à largeur fixe

Nous avons dès lors un arbre de codage qui est un arbre binaire complet, tous les caractères sont sur des feuilles à la même hauteur, utilisant le même nombre de bits. Peu importe le caractère, le parcours de l'arbre est $\Theta(\log k)$. Devant parcourir tout le texte, nous avons chaque parcours imbriqué dans une boucle $\Theta(n)$.

La complexité est donc : $\Theta(n \log k)$

3.d Différence de taille des fichiers

Nous pouvons remarquer une diminution d'environ 40% entre la taille du fichier encodé à largeur fixe, et celui encodé avec l'encodage de Huffman.

En effet, pour différents fichiers testés, voici à la Figure 1 les données qu'on en tire et la diminution entre l'encodage à largeur fixe et l'encodage de Huffman pour chacun des livres testés :

Livre	Taille (largeur fixe, en kB)	Taille (Huffman, en kB)	Diminution (%)
Frankenstein	443	247	44.24
Pride and Prejudice	785	423	46.11
Alice in Wonderland	164	96	41.46
The Importance of Being Earnest	139	82	41
The Master of Man	1022	586	42.66

FIGURE 1 – Temps de décodage pour chacun des encodages

3.e Temps de décodage

Voici, reportés à la Figure 2, les temps de décodage pour chacune des deux méthodes. Une nette amélioration est visible, le temps de décodage pour un texte d'une taille de 1 MégaByte (donc 1000 KiloBytes) avec Huffman est similaire à celui d'un texte de 200 KiloBytes avec un encodage à largeur fixe.

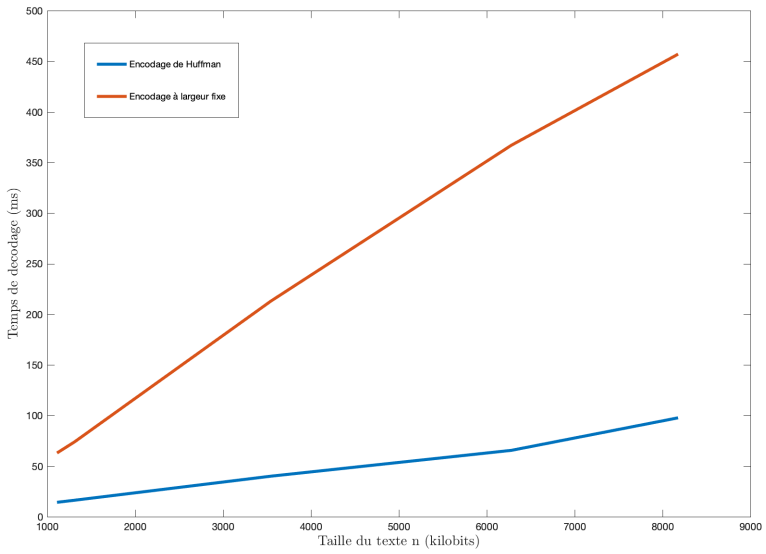


FIGURE 2 – Temps de décodage pour chacun des encodages

3.f Adéquation entre théorique et pratique

Comme le montrent les résultats empiriques, ils sont en adéquation avec la théorie. On encode sur moins de bits les caractères les plus fréquents, ce qui résulte en une nette diminution de la taille du fichier. De plus, grâce à cette propriété, le parcours de l'arbre lors du décodage est bien plus court lors d'un caractère fréquent, ce qui donne en moyenne un gain en temps, également visible dans les résultats empiriques.