University of Liège

Programmation avancée

INFO2050-1

Huffman

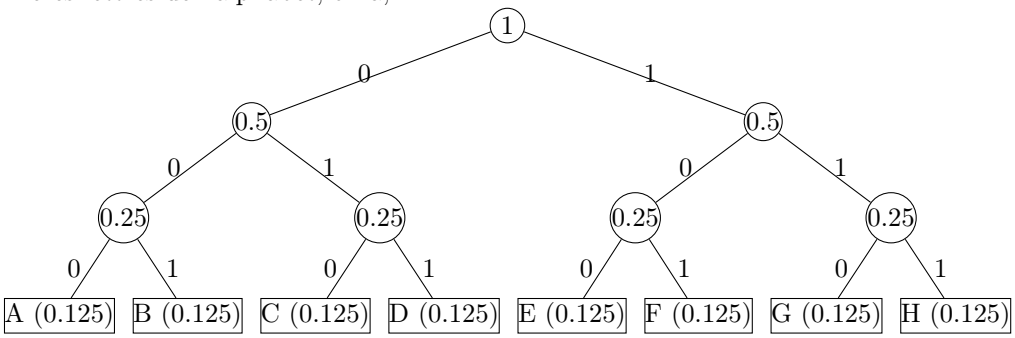
Julien Gustin, Mathias Carlisi

December 3, 2019

1 Structure des arbres

1.1 Equiprobable

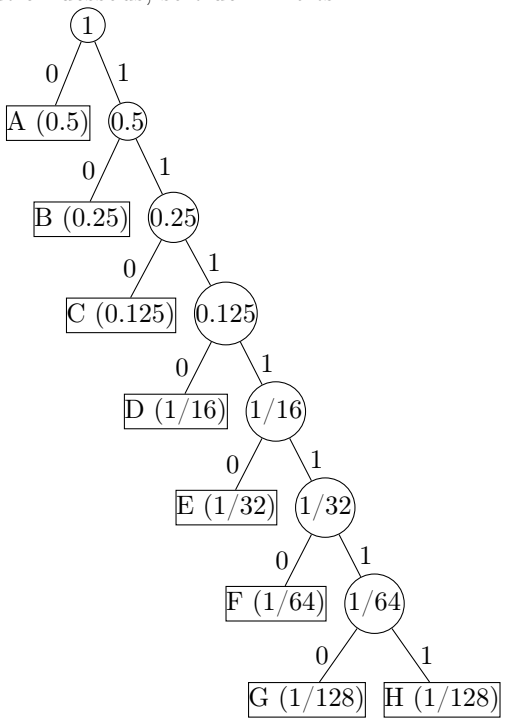
Pour une distribution des fréquences équiprobable, $f_i = \frac{1}{k}, i \in 1, ..., k$ entre chacun des caractères, l'arbre sera uniforme et donc les caracteres auront le meme nombres de bits à un bit près[1] Prenons un exemple avec k = 8, où k est le nombres de caractères différents. En prenant les huit premières lettres de l'alphabet, on a,



La séquence de bit lié à A sera : 000, B : 001 ... H : 111. Cela reste interessant tant que nous ne dépassons pas 256 caracteres différents, vu que le nombres de bits pour chacun des caracteres est de $\log_2 k$ alors qu'un caractère est stocké sur 8 bits au format usuelle.

1.2 Puissance de deux

Pour une distribution tel que $f_i = \frac{1}{2^i}, i \in 1, ..., k-1$ et $f_k = f_{k-1}$ l'arbre a à chaque niveau un caractère associé où le plus fréquent sera tout au dessus de l'arbre et donc de 1 bits et le dernier tout en dessous, soit de k-1 bits

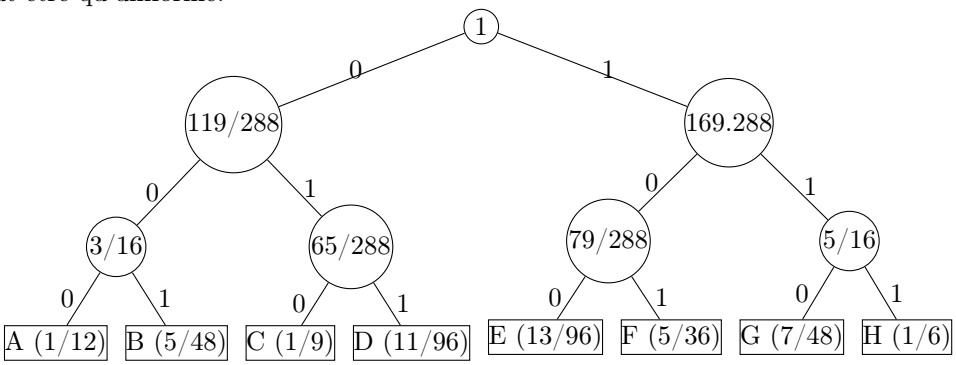


[1] Si k $\notin 2^i$, $i \in \mathbb{N}$

1.3 Epsilon

Pour des fréquences de caractères exprimé avec : $f_i = \frac{1}{k} + \epsilon_i$, $\epsilon_i = \begin{cases} \frac{-1}{3ki}, i \in \{1, ..., \frac{k}{2}\} \\ \frac{1}{3k(k-i+1)}, i \in \{\frac{k}{2}+1, ..., k\} \end{cases}$

L'arbre est lui aussi uniforme, vu que nous combinons les deux plus petites fréquences entre elles à chaque fois et l'addition de celles ci deviennent la plus grosse fréquences, ainsi l'arbre ne peut etre qu'uniforme.



2 Complexité

HUFFMAN(C)
1 $k = |C|$
2 Q = "create a min-prioity queue from C"
3 **for** $i = 1$ **to** $k - 1$
4 Allocate a new nodes z
5 $z.left = Extract - Min(Q)$
6 $z.right = Extract - Min(Q)$
7 $z.freq = z.left.freq + z.right.freq$
8 Insert(Q, z)
9 **return** Extract-Min(Q)

2.1 Liste

- ☐ Pour la ligne 2, nous avons $\mathcal{O}(k)$, provenant de la création de la "Priority queue" par pqCreate, qui appelle k fois pqInsert($\mathcal{O}(k)$.
- ☐ Pour la ligne 3 nous avons $k - 1 * \mathcal{O}(k)$ vu que retirer un élément de la "Priority Queue" a une complexité de $\mathcal{O}(1)$ et insérer un élément dans la "Priority Queue" a une complexité de $\mathcal{O}(k)$ dans le pire des cas.
- ☐ Et pour finir, comme dis precédement pour le Extract-Min, à la ligne 9 nous avons $\mathcal{O}(1)$.

Ainsi nous avons comme complexité totale $\mathcal{O}(k) + \mathcal{O}(k) + \mathcal{O}(1)$, que nous allons borner à $\mathcal{O}(k)$, où k correspond a la taille de l'alphabet.

2.2 Tas

- ☐ Pour la ligne 2, nous avons $\mathcal{O}(2k)$, provenant de la créations des racines de l'arbre pour chacun des caracteres et de la création de la "Priority queue" par pqCreate, qui appelle k/2 fois Min-heapify($\mathcal{O}(log(k))$. Nous avons calculer que $\mathcal{O}(\frac{k}{2} * log(k)) = \mathcal{O}(k)$, ainsi nous avons bien $\mathcal{O}(2k)$
- ☐ Pour la ligne 3 nous avons $k - 1 * \mathcal{O}(3log(k))$ vu que retirer et insérer un élément de la "Priority Queue" a une complexité de $\mathcal{O}(log(k))$. Nous allons la borner à $\mathcal{O}(3klog(k))$.

- Et pour finir, comme dis precédement pour le EXTRACT-MIN, à la ligne 9 nous avons $\mathcal{O}(log(k))$.

Ainsi nous avons comme complexité totale $\mathcal{O}(2k) + \mathcal{O}(3klog(k)) + \mathcal{O}(log(k))$, que nous allons borner à $\mathcal{O}(klog(k))$, où k correspond a la taille de l'alphabet. L'arbre associé au pire cas ressemble à celui du points **1.2**.

3 Algorithme de décodage

3.1 Peudo-code

DECODE(B, T)

```
1  Tmp.character = −1
2  Tmp.nextBit = 1
3  while Tmp.nextbit < GetNumberOfBits(B) //Evite de boucler à l'infinis
4        Tmp = ctDecode(T, B, Tmp.nextBit)
5
6        if Tmp.character == eof //Condition d'arret
7              return 1
8
9  return 0 //Si le fichier ne contient pas le caractere eof
```

3.2 Complexité

- **Meilleurs des cas** Le meilleur cas qu'on pourrait avoir avec ce type d'encodage est que notre fichier ne contiennes que le caractère avec la plus grandes fréquence et que celle si est assez frapantes comparé aux autres pour qu'elle ne soit stoqué que sur un bit, comme l'exemple du points **1.2**, si le texte ne contient que des 'A' avec cet arbre donné, nous aurons ainsi une complexité de $\mathcal{O}(n)$

- **Pire cas** Au contraire si le texte contient seulment le caractère avec la fréquence moindre nous aurons $\mathcal{O}(n*k-1)$, que nous bornerons à $\mathcal{O}(n*k)$, à chaque itération de décode et donc à chaque appel de ctDecode nous descendront récursivement jusqu'au plus bas de l'arbre, et nous savons que la profondeur maximum d'un arbre est *k-1*

3.3 Comparaison

La complexité de DECODE pour un encodage à largeur fixe est $\forall k \geq 2\ \mathcal{O}(n \log_2 k)$, ou k corresponds à la taille de l'alphabet et n à la taille original du texte. Le n provient du faite que nous devons traduire chacun des caractères un par un et le $\log_2 k$ lui est du à ctDecode qui comme le montre le points **1.1** vas devoir parcourir l'arbre jusqu'au dernier niveaux pour trouver une feuilles.

3.4 Tailles de fichiers

Généralement la taille des fichiers pour l'encodage à largeur variable, si les fréquences correspondent bien au textes donné, sera plus petite que celle encodé à largeur fixe. En utilisant l'encodage à largeur variable d'huffman pour le texte donné en exemple (Alice) nous avons un fichier de 95,9 ko tendis que à largeur fixe 143,3 ko. Nous gagnons plus de 30% de places.
Cette méthode de compressions est ainsi très utile et nous permet de gagner beaucoups de places sur de gros fichiers.

3.5 Graphique

Par déduction nous somme dis que l'encodage à largeur variable allait etre plus rapide, et ceci ce concrétise sur les ordinateurs de montefiore. Cependant sur nos ordinateurs personnel le fixe est plus rapide, nous reviendront sur ce cas dans le point **3.6**.

Figure 1: Valeurs tester sur mon ordinateur personnel

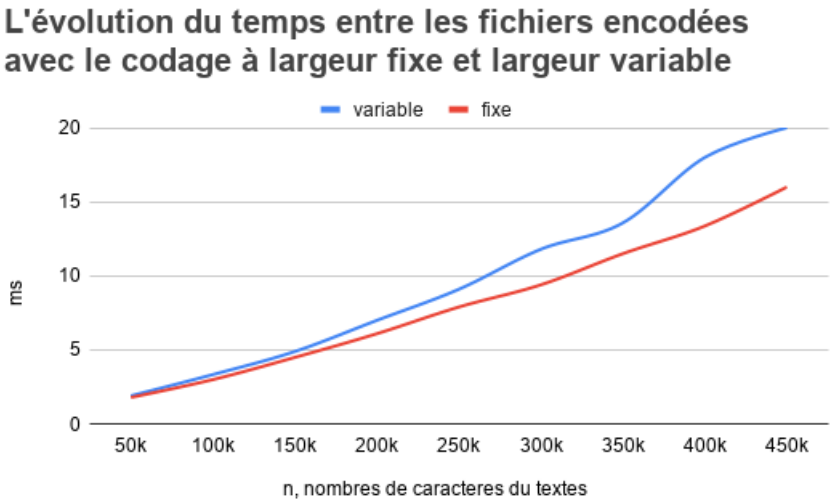
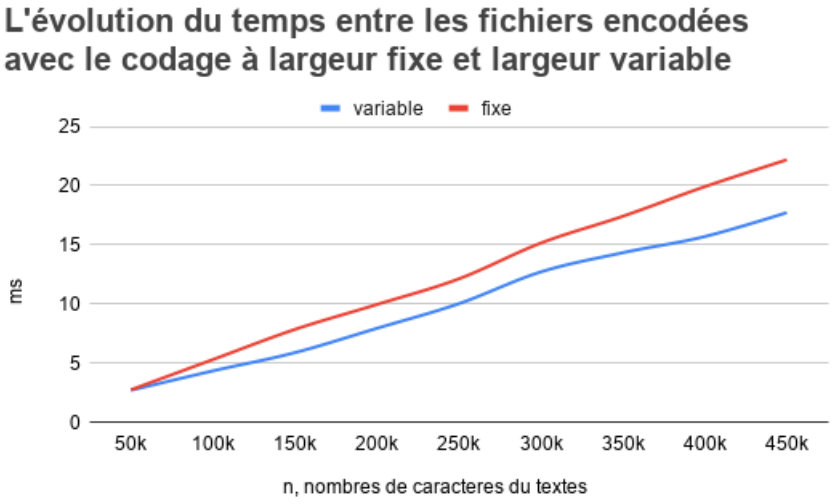


Figure 2: Valeurs tester sur les ordinateurs de montefiores



3.6 Analyse théorique vs résultats

Globalement les résultats obtenus sont ceux que nous aurions du avboir, la taille du fichier encodé à largeur variable est bien plus petit, notre seul petit problème est au niveau du temps d'execution.
Logiquement, nous nous sommes dis que l'encodage à largeur variable allait etre plus rapide, vu que notre fichier encodé est beaucoups plus petit. Cependant apres plusieurs test mis en oeuvre l'inverse s'est produit; comme nous pouvons le voir sur le graphique *figure 1*, les fichiers encodés à largeur variable sont décodé moins rapidement que ceux fixe. De là on ne peut faire que des déductions, est-ce un problème de cache ? Nous le savons pas cependant sur les ordinateurs de montefiore le résultat obtenus est le bon.