University of Liège

Programmation avancée

INFO2050-1

Huffman

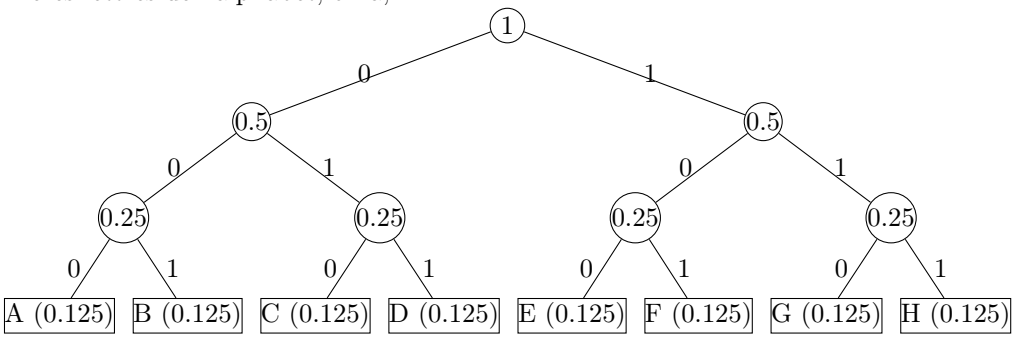
Julien Gustin, Mathias Carlisi

December 8, 2019

1 Structure des arbres

1.1 Equiprobable

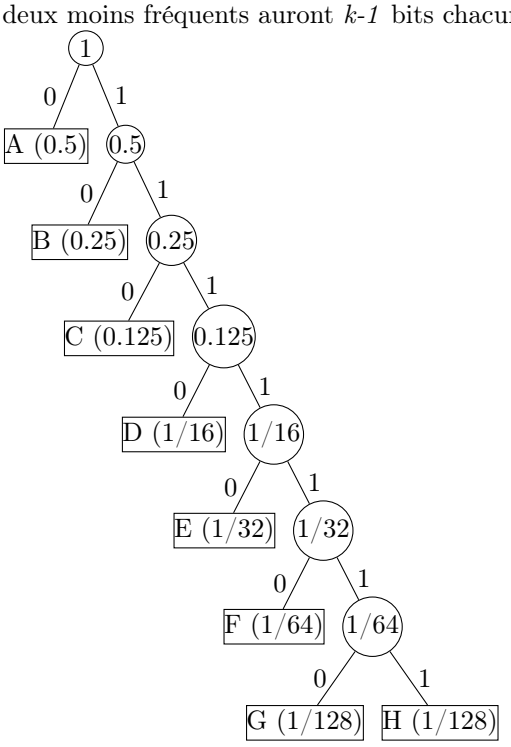
Pour une distribution des fréquences équiprobables, $f_i = \frac{1}{k}, i \in 1, ..., k$ entre chacun des caractères, l'arbre sera uniforme et donc les caracteres auront le même nombre de bits à un bit près[1] Prenons un exemple avec k = 8, où k est le nombre de caractères différents. En prenant les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



La séquence de bits liés à 'A' sera : 000, B : 001 ... H : 111. Cela reste intéressant tant que nous ne dépassons pas 256 caractères différents, vu le nombre de bits pour chacun des caractères 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 aura 1 seul bit, et les deux moins fréquents auront *k-1* bits chacun.

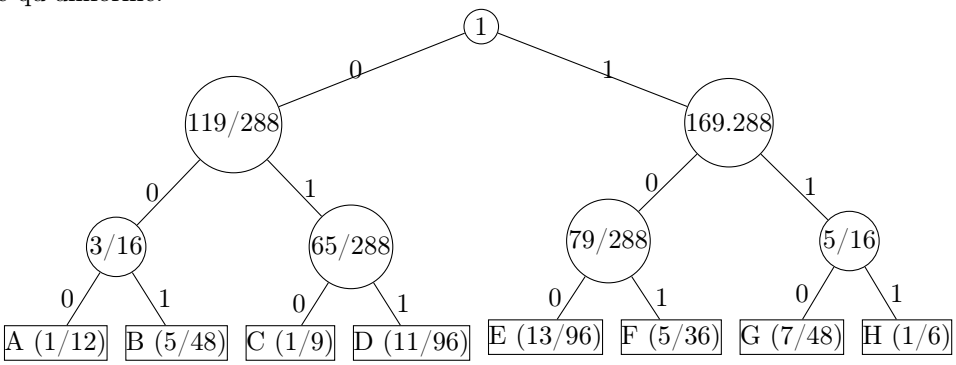


[1] Si k $\notin 2^i$, $i \in \mathbb{N}$

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 devient la plus grosse fréquence, ainsi l'arbre ne peut être 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, commençons par **le pire des cas**, si l'élément à placer a une priorité inférieure au dernier élément de la liste et une priorité supérieure à l'avant dernier élément, alors nous devrons parcourir tout le tableau sauf la dernière cellule. Ainsi nous aurons une compléxité de k * $\mathcal{O}(k)$, que nous pouvons borner par $\mathcal{O}(k^2)$.

Le meilleur des cas se présente si l'élément à placer a une priorité inférieure à la première cellule ou une priorité supérieure à la dernière cellule. Dans ce cas, INSERT aura une complexité de $\mathcal{O}(1)$. Ainsi nous aurons une compléxité de k * $\mathcal{O}(1)$, que nous pouvons borner par $\mathcal{O}(k)$.

☐ Pour la ligne 3, dans **le pire des cas** nous aurons $k - 1 * (\mathcal{O}(2) + \mathcal{O}(k))$, vu que retirer un élément de la "Priority Queue" a une complexité de $\mathcal{O}(1)$, ainsi nous pouvons borner la complexité par $\mathcal{O}(k^2)$.

Dans **le meilleur des cas** nous aurons $k - 1 * (\mathcal{O}(2) + \mathcal{O}(1))$, vu que retirer un élément de la "Priority Queue" a une complexité de $\mathcal{O}(1)$, ainsi nous pouvons borner la complexité par $\mathcal{O}(k)$.

☐ Et pour finir, quel que soit le cas comme dit precédement pour le EXTRACT-MIN, à la ligne 9 nous aurons $\mathcal{O}(1)$.

Donc nous avons comme complexité totale de $\mathcal{O}(k^2)$ pour **le pire des cas** et de $\mathcal{O}(k)$ pour **le meilleur**.

2.2 Tas

- ☐ Pour la ligne 2, nous avons $\mathcal{O}(k) + \mathcal{O}(k \log_2 k)$, provenant respectivement de la création des racines de l'arbre pour chacun des caractères et de la création de la "Priority queue" par pqCreate, qui appelle k fois pqInsert($\mathcal{O}(log(k))$.
- ☐ 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 precédemment pour le Extract-Min, à la ligne 9 nous avons $\mathcal{O}(log(k))$.

Ainsi nous avons comme complexité totale $\mathcal{O}(k) + \mathcal{O}(4klog(k)) + \mathcal{O}(log(k))$, 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 point **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'infini
4        Tmp = ctDecode(T, B, Tmp.nextBit)
5
6        if Tmp.character == eof //Condition d'arrêt
7              return 1
8
9  return 0 //Si le fichier ne contient pas le caractère eof
```

```
ctDecode(T, B, start)
1  if T.left == NIL and T.right == NIL
2        d.character = T.caractere
3        d.nextBit = start
4        return d
5  if biseGetBit(B, start) == ZERO
6        return (ctDecode(T.left, B, start+1)
7  return ctDecode(T.right, B, start+1)
```

3.2 Complexité

- ☐ **Meilleurs des cas** Le meilleur cas qu'on pourrait avoir avec ce type d'encodage serait que notre fichier contienne uniquement le caractère avec la plus grande fréquence. Cette dernière est assez élévée comparé aux autres pour qu'elle ne soit stoquée que sur un bit, comme l'exemple du point,**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 seulement le caractère avec la fréquence la plus faible 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. 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 correspond à la taille de l'alphabet et n à la taille original du texte. Le n provient du fait 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 point **1.1** va devoir parcourir l'arbre jusqu'au dernier niveau pour trouver une feuille.

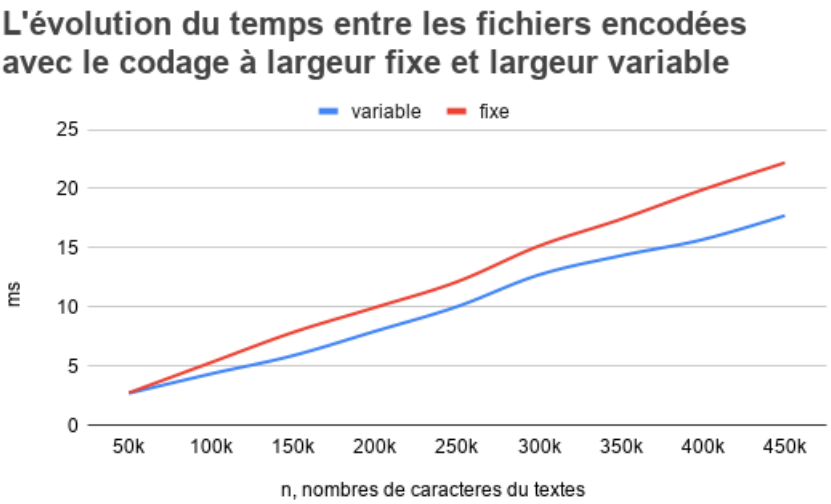
3.4 Tailles de fichiers

Généralement la taille des fichiers pour l'encodage à largeur variable (si les fréquences correspondent bien aux textes donnés), sera plus petite que celle encodée 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 tandis qu'à largeur fixe 143,3 ko. Nous gagnons plus de 30% de mémoire.
Cette méthode de compression est ainsi très utile et nous permet de gagner beaucoup de mémoire sur de gros fichiers.

3.5 Graphique

L'encodage à largeur variable est plus rapide que celui fixe, étant donné que le fichier est 30% moins volumineux pour le variable, il sera aussi 30% plus rapide car CTDECODE devra parcourir beaucoup moins de caractères.

Figure 1: Valeurs testées sur les ordinateurs de montefiores



3.6 L'adéquation entre l'analyse théorique et les résultats empiriques

Les résultats obtenus sont ceux escomptés. Nous avons bien un fichier compressé de taille plus petite que l' original.