Compte rendu Communication Numérique TP1

REGOUIN Roman ANDRIEUX Liam

10 Février 2020

Table des matières

1 Codage et Entropie

1.1 Entropie

L'entropie H d'une Source S peut être calculé grâce à la formule :

$$H(S) = -\sum p(a) \log_2(a)$$

Ici, nous avons $P_{e1} = 0.39, P_{e2} = 0.20, P_{e3} = 0.14, P_{e4} = 0.11, P_{e5} = 0.09, P_{e6} = 0.07$ Donc H(S) = 2,322788444 .

1.2 Longueur des mots

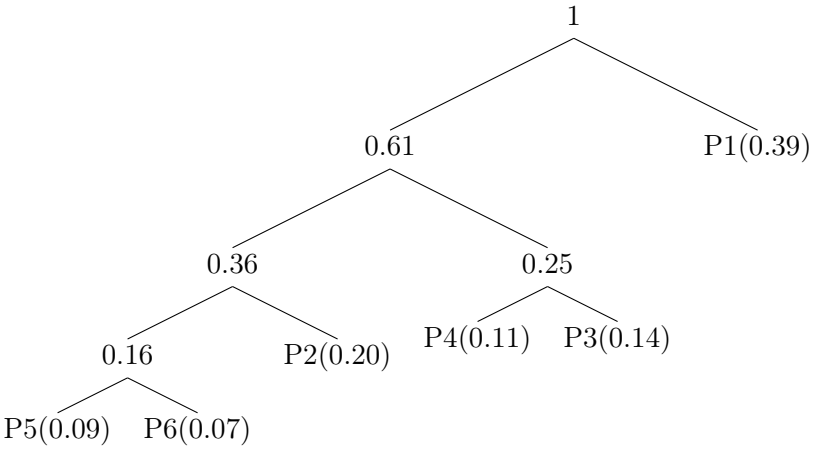
Nous avons 6 symboles distincts donc dans un code à taille fixe il nous faut des mots de longueur 3 :

$$2^2 < 6 < 2^3$$

On observe une distance d'environ 0.68 avec la borne inférieure ce qui est plutot élévé. On peut donc en déduire qu'il existe surement un meilleur codage pour se rapprocher de cette borne.

2 Code de Huffman

2.1 Arbre et code de Huffman



Pe1= 0.39,Pe2= 0.20,Pe3= 0.14,Pe4= 0.11,Pe5= 0.09,Pe6= 0.07 .
On obtient la table de code suivant :
P1 :0
P2 :110
P3 :100
P4 :101
P5 :1110
P6 :1111

2.2 Longueur moyenne et taux de compression

Pour la longueur moyenne, on a :

$$L(c) = \sum_{a \in A} p(a).|c(a)| = (1 * 0.39 + 3 * 0.20 + 3 * 0.14 + 3 * 0.11 + 4 * 0.09 + 4 * 0.07) = 2.38$$

La borne inférieur étant l'entropie qui est de 2,322788444, on est à une distance d'environ 0.06 de cette borne.
Lors de l'envoie du fichier codé, il faudra aussi envoyer le dictionnaire de correspondance dans le fichier ce qui implique un surcoût.
On a une somme de la taille du code qui est de 18bits.
Cependant il faut aussi coder les événements. Si on part du principe que les événements P1,P2,... sont codés sur 2 chars, on a un evenement qui fait une taille de 16bit (2octet/2char).Comme on dispose de 6 evenement on obtient un surcoût de 92bit soit un total de 110bit avec les bits de code.
Pour ce qu'il sagit du taux de compression, si on reprend l'éfficacité du "1 Codage et Entropie", on a alors :

$$Eff2 = Entropie \div L(c) = 2,322788444/2.38 = 0.97596153$$

$$Taux = (Eff2 - Eff1) \div Eff1 = 0.2609$$

On obtient donc un taux de compression de 26% avec l'algorithme de Huffman.

2.3 Rentabilité

Etant donné que la méthode avec l'algorithme de Huffman demande d'envoyer le dictionnaire des correspondance avec un surcoût, on va calculer à partir de quand cette méthode est rentable :

$$110 + 2.38X < 3X \Leftrightarrow 110 < 0.62X \Leftrightarrow X > 177.1428571$$

Donc à partir de 178 bits le codage de huffman est rentable.

3 Code de Huffman par bloc

On veut se rapprocher de la borne inférieure. L'idée est d'avoir la même démarche mais à partir dela liste des événements possibles pris deux à deux.
On obtient alors

$$n(X)^2$$

événements ou n(X) est le nombre d'événement de X soit

$$6^2 = 36$$

. On obtient la liste d'évenement suivant :
P1,1 ; P1,2 ; P1,3 ; P1,4 ; P1,5 ; P1,6 ; P2,1 ; P2,2 ; P2,3 ; P2,4 ; P2,5 ; P2,6 ; P3,1 ; P3,2 ; P3,3 ; P3,4 ; P3,5 ; P3,6 ; P4,1 ; P4,2 ; P4,3 ; P4,4 ; P4,5 ; P4,6 ; P5,1 ; P5,2 ; P5,3 ; P5,4 ; P5,5 ; P5,6 ; P6,1 ; P6,2 ; P6,3 ; P6,4 ; P6,5 ; P6,6

Avec le code de Huffman par double bloc on obtient une longueur moyenne de 4,6793. Avec le calcul :

$$H(S) \leq L(c2, S2)/2 < H(S) + 1/2$$

et

$$L(c2, S2)/2 = 4,6793/2 = 2,33965$$

On se rapproche donc bien de la borne inférieur.
Avec Huffman par code avec une multiplicité m choisie, on observe que plus le m est grand, plus on se rapproche de la borne inférieure. Cependant on ne peut pas dépassé m=10 sur nos machine car le nombre d'événement a traiter est trop grand, deplus avec notre programme on

ne peut pas non plus dépassé m=7 car avec nos différentes structure utilisé, la taille en mémoire est trop grande et la commande "valgrind nous signale un "stack overflow" ce qui nous amène à une érreur de segmentation.
Pour palier à ce problème, nous avons changé la structure de notre "Codehuffman" qui était :

```
struct code {
    int longueur;
    int code[30]; //ce tableau contenait que des 0/1
};
```

en la nouvelle structure :

```
struct code {
    int longueur;
    int code1;  // on va ici regarder les bits du int pour avoir nos 0/1
    int code2;
};
```

De cette façons nous ne stockons plus les 0 ou 1 dans un tableau de l'ancienne strucutre utilisant un tableau fixé a 30 mais nous stockons directement les informations dans les bits des int code1/code2 ce qui nous donne 64 emplacement.
De cette manière, nous avons pu économiser des ressources mémoires mais cela nous permet seulement d'arriver à m=7 et nous obtenons toujours la même erreur pour m=8.
Dans notre programme, les fonctions avec les plus grandes complexité sont la fonction "tab-mul" qui calcule la table des probabilités selon la multiplicité avec une complexité de n^{m+1}, la fonction "Construire Arbre" avec une complexité au pire de n^{m*2}(lorsque l'on doit ajouter tout le temps a la fin de la file à priorité, on a $(n^m * (n^m + 1))/2)$ et la fonction "ConstruireCode" qui fait appelle à la fonction "ParcoursArbre" qui doit parcourir tout l'arbre avec pour feuilles les n^m éléments, ce qui donne une complexité au pire de $2^{\lfloor log(n^m) \rfloor + 2}$
Quant à la complexité mémoire, on a une complexité de n^m (a multiplier par la taille d'un flotant) pour le tableau des probalités des événement ainsi que n^m (a multiplier par la taille de la structure "struct code") pour le tableau de codage des événements.

4 Codage arithmétique

4.1 Explication des algorithmes

Le premier algorithme est un algorithme de compréssion arithmétique. Le principe de cette compréssion est de représenter un message à l'aide d'un flottant compris entre 0 et 1. Pour cela, il faut créer une table qui contient la probabilité de chaque symbole et associé à chaque symbole un intervalle flotant qui le défini. Ensuite, pour chaque symbole lu, l'intervalle qui défini le message codé se modifie d'un montant défini par la probabilité d'apparition du symbole lu. Une fois tout les symboles lu, nous prenons une valeur arbitraire de notre intervalle codant le message comme par exemple le milieu.

Le deuxième algorithme est l'agorithme de décompréssion arithmétique. En possédant la table des symboles, il peut appliquer le processus inverse, c'est-à-dire prendre le nombre flottant codant le message et extraire la lettre associé et ensuite modifier l'intervalle en fonction de la probabilité de la lettre décodée.

4.2 Encodage

$\mathrm{V}_{wiki} = 0.171875$ $V_{kiwi} = 0.171875$ $V_{kikiwiwi} = 0.084717$

4.3 Décodage

En utilisant la table de codage du mot wiki :

$\mathrm{V}_{0.008}$: $wwwi$
$V_{0.517}$: $iiii$
$V_{0.164}$: $wikw$
$V_{0.312}$: $iwii$

En utilisant la table de codage du mot kiwi :
$V_{0.008}$: $kkki$
$V_{0.517}$: $iiii$
$V_{0.164}$: $kiwk$
$V_{0.312}$: $kkkk$

En utilisant la table de codage du mot kikiwiwi :
$V_{0.008}$: $kkkiiiii$
$V_{0.517}$: $iiiiwkik$
$V_{0.164}$: $kiwkwwwi$
$V_{0.312}$: $ikiiiiii$

4.4 Variation de n

Si lors de la décompression, le n utilisé est plus petit que la taille du message, alors nous aurons que les n premières lettres qui seront décodées.

Si lors de la décompression, le n utilisé est plus grand que la taille du message, alors nous aurons le message décodé corretement avec en plus des lettres qui varie en fonction de la table des symbole. En effet, une fois la taille du message dépassé, le calcul de Vmsg ce feras avec les dernière lettres du message jusqu'a ce que celui-ci passe dans l'intervalle d'une autre lettre et ainsi de suite.

Si nous on ne connait pas la taille du message initiale à l'avance, on peut stocker le Vmessage donné en argument et effectuer le décodage lettres par lettre tout en rencodant la lettre décoder jusqu'a retomber sur le Vmessage donné en argument.