Rapport Projet de Synthèse

Marius AMBAYRAC et Andrés Garcia

2018

Table des matières

1 Introduction

Que ce soit dans l'envoi d'un fac-similé, le stockage de données comme peut être une image digitale ou le téléchargement d'un fichier MP3, les outils numériques et plus généralement, la théorie de l'information est devenue de nos jours la base fondamentale des communications à distance. Avec la digitalisation des contenus, le gain de vitesse en termes de calcul informatique des dernières décennies et la démocratisation des technologies, notre société est capable aujourd'hui de transmettre de l'information d'un coin du monde à l'autre en quelques secondes à travers une multitude de moyens, permettant ainsi de travailler en réseau et d'établir des liens qui autrement seraient impossibles. Toutefois, à chaque seconde cette société génère d'énormes flux d'informations et les technologies de la communication ont du savoir se réinventer. Pour résoudre ce problème, le codage est apparu.

Le codage correspond simplement à la transition d'une représentation de données selon un système de règles vers un autre différent. Parmi tous les codages, on différencie notamment deux types : le codage de canal, qui vise à utiliser une représentation plus redondante résistante aux erreurs de transmission tels que le bruit ; et puis le codage de source, codage sur lequel nous nous intéresserons sur ce travail et qui permet de compresser ces données (avec ou sans perte d'information). Les deux sont donc complémentaires et Shannon démontra que, dans certains cas, cette distinction opérationnelle est asymptotiquement optimale. Cependant, avant de continuer, il faudrait se poser plusieurs questions : Qu'est-ce qu'on comprend par « se communiquer », quels sont ces technologies existantes dont on en parle et quels sont les problèmes et limitations que nous retrouvons sur les modèles ? Cette étude est divisée en deux parties : dans la première, on comprendra et analysera la fondation théorique qu'existe derrière les moyens de communication actuels pour, dans un deuxième temps, approfondir dans l'implémentation sur différentes sources d'une méthode concrète de codage de source : le Codage entropique de Huffman à longueur variable.

2 Partie Théorique

Dans la suite de cette partie, nous établirons la base théorique qui permet le postérieur développement pratique à travers les notions des bases qui expliquent la caractérisation d'une communication efficace et puis ses consé-

quences moins évidentes.

2.1 Définitions essentielles

La première notion fondamentale à prendre conscience est le Paradigme de Shannon. Etudié par Claude E. Shannon et Warren Weaver en 1949, ce modèle s'agit d'un scénario conçu d'une façon schématique pour représenter *la transmission d'un message* à travers un système de communication.

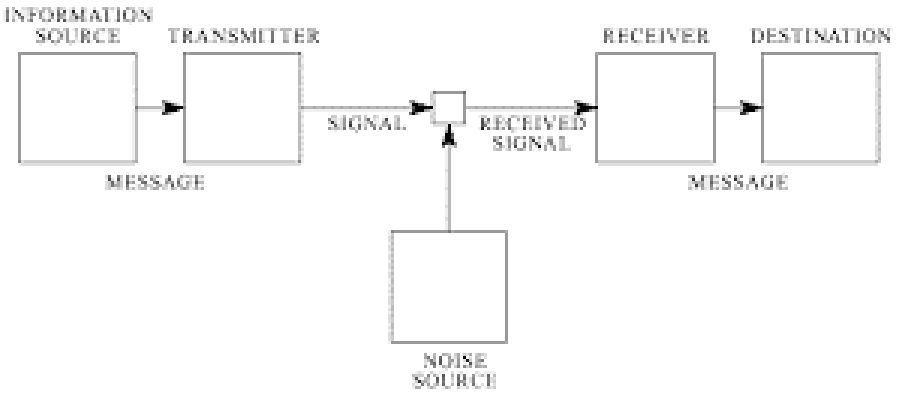


Fig. 1—Schematic diagram of a general communication system.

Sur celui-ci, la transmission du message démarre par la source, siège d'évènements aléatoires à l'origine du message émis et caractérisée par son *entropie* (qui consiste à une quantité *moyenne* d'information produite et est relié au concept d'*incertitude*. Ensuite se déroule le codage en source (élimine de la redondance et réduit la taille du message) et le codage canal sur le canal (qui transmet et dégrade le message) caractérisée par une *capacité*. Finalement les phases de décodages correspondantes (opérations réciproques pour restituer le message) avant d'arriver au destinataire.

Cette théorie est une théorie probabiliste et pour pouvoir continuer, il est nécessaire de citer quelques outils de base qui serviront à introduire la notion d'entropie.

La réalisation d'un évènement (comme l'est la réception d'un message déterminé) nous procure de l'information seulement si son contenu nous était inconnu à l'avance. En effet, et dans la même logique, on peut dire que plus

un évènement est rare (et donc plus sa probabilité de réalisation est faible), plus grande est la surprise ressentie lorsqu'il apparaît, i.e. l'information que nous apprenons une fois on le détecte. Pour pouvoir mesurer cette *incertitude* de l'évènement, Shannon proposa la formule suivante :

$$h(E) = log(\frac{1}{P(E)}) \tag{1}$$

De la même façon, nous pouvons aussi réfléchir à l'idée d'incertitude d'un évènement pour une probabilité conditionnelle sachant que B a été déjà réalisé :

$$h(E/B) = log(\frac{1}{P(E/B)}) \tag{2}$$

Les études de Shannon prennent compte aussi sur la fondation de cette théorie à l'idée primordiale, celle d'information. Sur la base de l'incertitude, on dit que l'information que B apporte sur l'évènement E correspond à la différence des deux incertitudes précédentes :

$$I(B, E) = h(E) - h(E/B) = h(B) - h(B/E) \tag{3}$$

D'autre part, en se limitant aux sources discrètes (la plupart d'elles peuvent être considérées ainsi), on peut les décrire comme des dispositifs susceptibles de fournir de façon aléatoire des symboles issus d'un ensemble fini qu'on appellera *alphabet* (« A ») à m *éléments*. De cette façon, les données émises par une source U pourront être modélisées par une suite de variables aléatoires U_1, U_2, ... à valeurs dans A suivant une loi probabilistique déterminée et chaque symbole sortant de cette source à des instants donnés correspondront à une réalisation de la variable aléatoire U_k correspondante. De plus, la source U sera dite une source m-aire.

Ensuite, nous parlons de source stationnaire lorsque les propriétés probabilistes de la réalisation de ces évènements sont stables dans le temps, i.e. les différents U_k ont la même loi de probabilité. Parallèlement, on dit que la source a une mémoire d'ordre n lorsque la loi de probabilité du U_{k+1} dépendent des n variables U_k, U_{k-1}, ..., U_{k-n} antérieures et c'est pour le cas particulier de n = 1 que nous travaillons sur une source markovienne.

Néanmoins, comme on vient de dire, une source U peut être représentée par une variable aléatoire qui à son tour, se réalise sur un nombre fini de potentiels évènements et c'est ici où on réalise le parallélisme : tous les concepts antérieurs peuvent être étendus d'un évènement en particulier à toute une source et plus précisément, la moyenne pondérée des incertitudes des réalisations de la variable aléatoire U est appelée *entropie* H(X) et est calculée ainsi :

$$H(X) = \sum_{i=1}^{n} p_i * log(\frac{1}{p_i}) = -\sum_{i=1}^{n} p_i * log(p_i) \tag{4}$$

Et de même, l'entropie conditionnelle de Xk sachant Xk-1 (comme arrive à une source markovienne) :

$$H(X_k/X_{k-1}) = -\sum_{j=1}^{n} P(X_{k-1} = x_j) * H(X_k/X_{k-1} = x_j) \tag{5}$$

Pour terminer, si on élargit la définition d'information pour une variable aléatoire X sur Y :

$$I(X,Y) = h(X) - h(X/Y) = h(Y) - h(Y/X) \tag{6}$$

qui après développement correspond à :

$$I(X,Y) = E\left[log\frac{P(X,Y)}{P(X)P(Y)}\right] \geq 0 \tag{7}$$

avec l'égalité lorsque X et Y sont indépendants (le cas des sources sans mémoire). On peut donc conclure qu'une source à mémoire non nulle aura toujours moins d'entropie (c'est-à-dire, sera plus prévisible) qu'une sans mémoire. Ceci est facilement observable sur une source markovienne : le fait d'être sur un certain « état » permet de limiter les états suivants possibles.

2.2 Inégalité de Kraft-McMillan

Une fois que nous avons défini correctement ce qu'est la source, nous pouvons enfin décrire la démarche du codage source.

Du point de vue formel, un codage source C pour une variable aléatoire X modélisant une source correspond à un *data mapping* entre les évènements de l'univers X (support de X) à D, l'ensemble des vecteurs finis composés de symboles d'un alphabet k-aire A. Chacun de ces vecteurs sera appelé mot-code de xi selon C. De plus, on parle de codage *non-singulier* lorsque la transformation C est injective :

$$x_i \neq x_j \rightarrow C(x_i) = C(x_j) \tag{8}$$

et aussi sera de *longueur fixe* si tous les mots-code image de X ont la même taille d'éléments structurels fondamentaux (pour le codage binaire, on décrit ainsi les *bits*). Grâce à ce type d'applications, nous sommes capables de traduire n'importe quelle donnée d'un langage à un autre juste en précisant quels sont les liaisons entre les éléments des deux espaces. En revanche, ce fait succinct pas mal d'interrogatifs : est-ce que n'importe quel ensemble de liaisons est valable ? Et parmi ceux qui le sont, est-ce que tous sont aussi efficaces que les autres ou au contraire, nous pouvons converger sur une optimalité, notamment en termes de gain en longueur de la nouvelle représentation de la donnée ?

En effet, ils existeront plusieurs façons de coder les données d'une source selon le mapping que nous choisirons, très nombreux même en utilisant le même langage (disons désormais, du langage français au langage binaire). Une solution possible pourrait être celle d'utiliser un codage de longueur fixe, la longueur n de tous ces mots code serait $n = \lceil log_2 M \rceil$ où M correspondent au nombre d'états possibles de X. D'une manière certaine, on pourrait ainsi définir une application injective (tous les x auront leur mot code unique) mais concernant la longueur du message résultant, on peut trouver bien plus efficace (nous sommes en train d'utiliser des mots ayant la même longueur pour coder un « z » que les « a »). C'est ici où on trouve les codes de *longueur variable* (mots code de longueur variable) et intuitivement, on réfléchit à mettre ensemble les symboles les plus fréquents de la source aux variables de codage ayant le moins de bits. En revanche, tout système de codage source doit vérifier la textitcondition de déchiffrabilité : l'application C doit être réversible, i.e. que à partir de tout message codé doit avoir à travers l'appli-

cation réciproque un message unique dans le langage original et évidemment, à partir du moment où on choisit un codage de longueur variable, ce code ne doit pas être forcément décodable. Comme en français, on pourrait concevoir un symbole équivalent à l'espace pour pouvoir reconnaître la fin d'un mot codé et le début du suivant mais cela nuirait énormément à l'efficacité qu'on cherche (surtout du au fait que ce nouveau caractère serait probablement le plus fréquent de tous, comme arrive aussi au français). L'extension Ck d'un codage C est le *mapping* depuis les vecteurs finis de l'univers X aux vecteurs finis de D défini par la concaténation des codages des parties élémentaires du mot original :

$$C(x_1, x_2, \ldots, x_n) = c(x_1)c(x_2)\ldots c(x_k) \tag{9}$$

Un code sera dit uniquement déchiffrable si cette extension est aussi non-singulière, vérifiable par le test de Sardinas-Patterson. Ce problème polynomial consiste à chercher les mots codes qui seraient *préfixes* d'autres mots de D et de voir si la partie restante qui différencie ces deux derniers correspond à un autre mot de D. Si c'est le cas, il existeraient au moins deux façons d'interpréter une chaîne où ces deux mots seraient suivis et donc, l'extension C^k ne serait pas uniquement déchiffrable. En revanche, si aucun mot est préfix d'un autre, on parle donc d'un *code préfix* ou *code instantané* et on peut affirmer de lui qu'il sera uniquement déchiffrable puisqu'il n'existera pas l'ambigüité que vient d'être décrite précédemment mais en plus, il ne nous faudra pas regarder à la séquence complète pour pouvoir commencer à décoder puisque les mots commenceront et finiront sans équivoque.

Dans la même logique, et toujours sur le même exemple, on pourrait réfléchir au fait qu'en français, la probabilité de réalisation des différentes lettres ne sont pas indépendantes des réalisations précédentes (en effet, après un « q » on aura certainement un « u »). Il serait donc intéressant de tenir le fait de coder une source qui possède une mémoire et donc, de considérer cette particularité aussi sur le calcul des fréquences au moment d'attribuer le mot code à chaque chaîne de lettres.

Plus généralement, soit U une source à m symboles et d'entropie H (U. Alors son contenu est compressible sans perte d'information si $H(U) < log(m)$, c'est-à-dire, si son entropie n'est pas maximale. On dit alors que la source possède de la rédondance et l'opération de codage de source va consister à réduire celle-ci. Le fait qu'il existe de la redondance provient justement du déséquilibre de la loi de distribution par rapport à la loi uniforme ou aussi de l'existence d'une mémoire.

Cependant, est-ce qu'il existe une borne sur la longueur des mots parmi tous les codes instantanés possibles ? C'est ce qu'est indiqué par le théorème de Kraft : pour tout code instantanée C de taille finie M (nombre d'éléments en X) sur un alphabet de taille D, les longueurs L_i des mots codes correspondants satisfont l'inégalité :

$$\sum_{j=1}^{M} D^{L_j} \leq 1 \tag{10}$$

et grâce aux études de McMillan, cette inégalité fut étendu à tout codage uniquement déchiffrable, ce qui permet de créer un système de codage instantanée équivalent à tout codage vérifiant cette inégalité et réciproquement donc, que tout code C uniquement déchiffrable vérifie l'inégalité de Kraft.

2.3 Bornes sur la longueur moyenne attendue d'un code

La longueur attendue L d'un code instantané quelconque peut être défini comme :

$$L = E[L(x)] = \sum_{i=1}^{rang(X)} L(x_i)p(x_i) \tag{11}$$

où D est la taille de l'espace codé D. Cette valeur, qui peut être considérée comme la valeur moyenne des mots codes, est borné inférieurement par :

$$L \geq H_D(X) \tag{12}$$

où HD est l'entropie de la source U (formule 4) en prenant en compte sa mémoire (si elle l'a). L'égalité sera seulement si l'inégalité 12) est égalité et si :

$$L = -log_D(p_j) \tag{13}$$

ce qui donne une longueur optimale au mot code pour chaque valeur de X même si ce L ne doit pas être forcément entier positif, ce qui oblige à prendre celui directement supérieur et le système devenant pas optimale. C'est pour cela que toute distribution de probabilités de la forme D-n pour tous ces X sera appelée D-adic et sera optimale pour un code sur un alphabet de taille D (vérifiant l'égalité antérieure pour des Lj entiers).

Cette remarque conclut sur le théorème d'importance : le Théorème du codage de source. Celui dit que pour tout codage instantané (et donc uniquement déchiffrable) et D-aire, la longueur attendue minimale satisfait :

$$H_D(X) \geq L \geq H_D(X) + 1 \tag{14}$$

L'intervalle a donc une amplitude de 1 bit et comme on a dit, le minimum sera atteint pour une distribution D-adic. Il est intéressant de se dire que, si on divise l'inégalité par k (soit $l = L/k$ est le nombre moyen de symbole code correspondant à une lettre source), on obtient l'inégalité suivante :

$$\frac{H_D(X)}{k} \geq l \geq \frac{H_D(X)}{k} + \frac{1}{k} \tag{15}$$

On observe que si k tend vers l'infini, la paramètre l est effectivement bornée sur une seule valeur. La conclusion est que l'entropie d'une source U apparaît comme le nombre moyen de symboles D-aires nécessaires pour représenter une lettre source dans une limite que n'est pas forcément atteinte mais peut être approchée.

Ce que ce théorème suggère est tout simplement une procédure algorithmique pour définir un codage optimal : face à une distribution probabilistique de X, il nous faut trouver la distribution D-adic plus proche à celle de X, ce qui est possible grâce à l'algorithme de Huffman.

2.4 Propriétés des codes binaires optimaux

2.5 Optimalité du Codage de Huffmann

Le codage de Huffman a une propriété remarquable dans la classe des codage sans perte d'information. L'idée du codage de Huffman est d'attribuer aux motifs les plus fréquents les codes les plus courts. Il faut toutefois respecter des propriétés comme l'unique déchiffrabilité, pour avoir un codage utile. Voici le pseudo code de l'algorithme de Huffman, nous allons l'expliquer par la suite :

Algorithm 1 Huffman$(p_1, p_2, \ldots, p_m)$

Require: probability distribution p
Ensure: optimal binary code
1: **if** $m = 2$ **then**
2: $C(1) \leftarrow 0$ and $C(2) \leftarrow 1$
3: **else**
4: sort $p_1 \geq p_2 \geq \ldots \geq p_m$
5: $C' \leftarrow$ **Huffman**$(p_1, p_2, \ldots, p_{m-2}, p_{m-1} + p_m)$
6: **for** $i = 1, \ldots, m-2$ **do**
7: $C(i) \leftarrow C'(i)$
8: **end for**
9: $C(m-1) \leftarrow C'(m-1)0$
10: $C(m) \leftarrow C'(m-1)1$
11: **end if**
12: return C

L'idée est qu'on crée le code par récurrence. On part d'une distribution d'une source, que l'on ordonne par probabilité décroissante : $[p_1, p_2, ..., p_m]$.

- **1er cas :** Si m=2 On sait résoudre le problème : le codage optimal est donné en codant un motif avec 1 et un motif avec 0.
- **2 cas : m>2, supposons qu'on sache générer un codage optimal pour une source avec m-1 motifs grâce à l'algorithme de Huffman**.
  On met alors de côté le dernier motif : celui avec la plus faible probabilité de présence. On applique alors l'algorithme mais avec la distribution de probabilité de taille m-1 $[p_1, p_2, ..., p_{m-1} + p_m$. Par hypothèse, cet algorithme renvoie un codage optimal $[c'_1, ..., c'_{m-1}]$. On associe alors aux m-2 premiers motifs le même code : $\forall i \in \{1, ..., m-2\} c_i = c'_i$ et $c_{m-1} = c'_{m-1}0$ et $c_m = c'_{m-1}1$.
  La longueur attendue du codage vaut alors : $L(C_{m-1}) + p_{m-1} + p_m$. Comme on $p_{m-1}+p_m$ ne dépend pas de la longueur du codage et comme la longueur de $L(C_{m-1})$ est minimale par hypothèse de récurrence alors la longueur attendue de notre codage est minimale.

2.6 Limitations du Codage de Huffmann

Même si le codage de Huffman est méthodique et universel, il existe des limitations face à plusieurs situations :
- Les codages de Huffman une fois sont établis ne sont pas robustes face aux modifications dynamiques (par exemple, la rédaction d'un rapport) puisqu'ils ne sont pas facilement adaptables aux changements de distributions à la source (changement d'usager).
- Pour chaque source, l'arbre de codage de Huffman doit donc changer (raison de la première limitation)
- Le codage de Huffman n'est pas capable d'incorporer l'information dans un cadre plus général de séquences d'éléments autre que sur l'émission des éléments un à un par la source (exemple distribution lettres / mots).
- Connaissance au préalable des probabilités sur la source : la table de probabilités associés aux codes choisis devra être aussi transmise pour décoder et cela aura un effet négatif sur le but du codage source.

3 Partie pratique

3.1 Source binaire stationnaire sans mémoire

Considérons une source binaire stationnaire et sans mémoire. Prenons $p_0 = 0.9$ la probabilité que la source émette un 0. Nos objectifs sont alors les suivants :

- Construire un codage de Huffman binaire pour une 3-extension de la source
- Calculer l'entropie de la source en bits

On peut déjà calculer l'entropie de la source : $H_2(X) = 0.469$ bits
La première étape consiste ensuite à établir le tableau associant à chaque motif sa probabilité d'apparition. Calculons par exemple la probabilité pour le motif '010' :
Notons X la variable aléatoire correspondant à la source. Note Y la variable aléatoire correspondant à la 3-extension de la source

$$P(Y =' 010') = P(X =' 0') * P(X =' 1') * P(X =' 0') \tag{16}$$

Par indépendance qui découle du caractère sans mémoire de la source.
On obtient ainsi le tableau de probabilité suivant :

motif	probabilité
'000'	$0.9^3 = 0.729$
'001'	$0.9 * 0.9 * 0.1 = 0.081$
'010'	$0.9 * 0.1 * 0.9 = 0.081$
'011'	$0.9 * 0.1^2 = 0.009$
'100'	$0.1 * 0.9^2 = 0.081$
'101'	$0.1 * 0.9 * 0.1 = 0.009$
'110'	$0.1^2 * 0.9 = 0.009$
'111'	$0.1^3 = 0.001$

On peut ainsi appliquer l'algorithme de Huffman sur la source définie par le tableau de probabilité précédent. Notre code source est fourni en annexe 6.2. On pourra noter que le tri de la liste des probabilité à chaque appel récursif nous a posé problème. Ainsi pour réglér ce problème, on a donné comme motif : "new" à l'élément que l'on ajoute pour se ramener à un échantillon de taille (n-1) pour faire un appel récursif. On repositionne ensuite cet élément à la fin de la liste, pour qu'il ne crée pas de problème.
On obtient le tableau suivant :

motif	probabilité	code
'000'	0.729	'0'
'001'	0.081	'110'
'010'	0.081	'101'
'011'	0.009	'11110'
'100'	0.081	'100'
'101'	0.009	'11101'
'110'	0.009	'11100'
'111'	0.001	'11111'

On peut aussi calculer l'entropie par bit de la 3-extension de la source (qui, sans surprise vaut 3 fois l'entropie de la source). On obtient
$H_2(X) = -\sum_{i=0}^{7} -p_i log_2(pi) = 1.40699$ bits
Pour ce calcul, on a utilisé la fonction basique présentée en annexe 6.1.1.
Pour aller plus loin, nous avons aussi calculé la longueur moyenne du codage

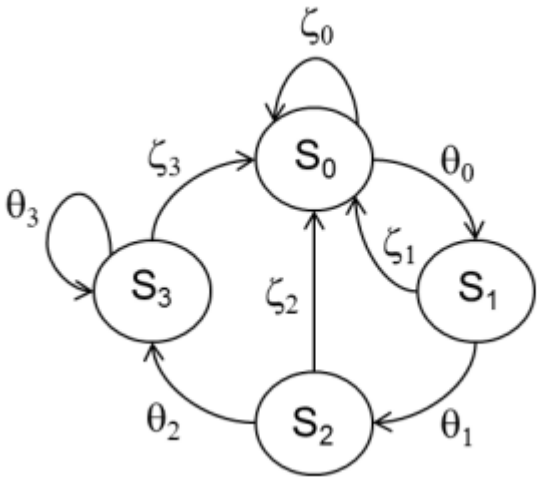
obtenu :

$$L = \sum_{i=1}^{8} L(x_i)p(x_i) \tag{17}$$

On utilise pour cela la fonction fourni en annexe 6.1.2. On obtient : 1.598 qui est bien sûr supérieur à l'entropie.

3.2 Source binaire Markovienne

Considérons maintenant une source binaire stationnaire apériodique : une chaine de Markov. Cette chaine possède n états. La source émets 0 pour toute transition vers l'état S_0 et émets 1 pour toute transition vers S_j où $j \neq 0$.
Voici un exemple pour le cas n=3 :



On cherche alors à construire un codage de Huffman pour cette source.
Pour ce faire, on va suivre le cheminement suivant :

1. Spécifier la matrice de transition π
2. Trouver l'unique distribution limite μ
3. Donner l'expression simplifiée de μ si $\theta_j = \theta, \forall j \in \{1, ..., n\}$
4. Exprimer la probabilité θ_0 comme fonction de μ_0
5. Evaluer la distribution de probabilité pour une 3-extension de la source
6. Construire le codage de Huffmann pour cette distribution de probabilité
7. Calculer l'entropie de la source, en bits

On va donc traiter le cas $n = 3$, $\theta_0 = 1/16$ *et* $\forall j \in \{1, ..., 3\}$ $\theta_j = 1/2$
Dans la matrice de transition en place (i,j) on a la probabilité de passer de l'état i vers l'état j. Ainsi :

$$\pi = \begin{bmatrix} \xi_0 & \theta_0 & 0 & 0 \\ \xi_1 & 0 & \theta_1 & 0 \\ \xi_2 & 0 & 0 & \theta_2 \\ \xi_3 & 0 & 0 & \theta_3 \end{bmatrix} \tag{18}$$

Cherchons ensuite le vecteur μ distribution limite.

$$\mu\pi = \mu \tag{19}$$

Posons $\mu = [\mu_0; \mu_1; \mu_2; \mu_3]$ En écrivant les équations et en y ajoutant l'équation :

$$\sum_{i=0}^{3} \mu_i = 1 \tag{20}$$

On obtient : :

$$\begin{cases} \mu_0 = \dfrac{1}{1 + \theta_0 + \theta_0 * \theta_1 + \frac{\theta_0 * \theta_1 * \theta_2}{1 - \theta_3}} \\ \mu_1 = \theta_0 * \mu_0 \\ \mu_2 = \theta_0 * \theta_1 * \mu_0 \\ \mu_3 = \dfrac{\theta_0 * \theta_1 * \theta_2 \mu_0}{1 - \theta_3} \end{cases} \tag{21}$$

Ensuite, si $\theta_j = \theta, \forall j \in \{1, ..., n\}$, comme dans notre exemple, on a :

$$\begin{cases} \mu_0 = \dfrac{1}{1 + \theta_0 + \theta_0 * \theta + \frac{\theta_0 * \theta^2}{1-\theta}} \\ \mu_1 = \theta_0 * \mu_0 \\ \mu_2 = \theta_0 * \theta * \mu_0 \\ \mu_3 = \dfrac{\theta_0 * \theta^2 \mu_0}{1-\theta} \end{cases} \tag{22}$$

On peut alors écrire la probabilité θ_0 en fonction de μ_0 :

$$\mu_0 * [1 + \theta_0 * (1 + \theta + \frac{\theta^2}{1-\theta}] = 1$$
$$\theta_0 [\frac{1 - \theta + \theta * (1-\theta) + \theta^2}{1-\theta}] = \frac{1 - \mu_0}{\mu_0}$$

Et finalement on a

$$\theta_0 = \frac{(1-\mu_0)(1-\theta_0)}{\mu_0}$$

Dans le cas de notre exemple, on trouve donc les valeurs approchées suivantes :
$\mu = [0.901408; 0.056338; 0.028169; 0.028169]$
On évalue ensuite la distribution de probabilité pour une 3-extension de notre source :

motif	probabilité
'000'	$\frac{15}{16}^3 = 0.8240$
'001'	$\frac{15}{16}^2 * \frac{1}{16} = 0.0549$
'010'	$\frac{15}{16} * \frac{1}{16} * 0.5 = 0.0293$
'011'	$\frac{15}{16} * \frac{1}{16} * 0.5 = 0.0293$
'100'	$\frac{1}{16} * 0.5 * \frac{15}{16} = 0.0293$
'101'	$\frac{1}{16} * 0.5 * \frac{1}{16} = 0.0019$
'110'	$\frac{1}{16} * 0.5^2 = 0.0156$
'111'	$\frac{1}{16} * 0.5^2 = 0.0156$

Une fois qu'on a la distribution de probabilité, on peut appliquer notre algorithme 6.2 et en déduire directement le codage de Huffman correspondant !

motif	probabilité	code
'000'	0.8240	'0'
'001'	0.0549	'101'
'010'	0.0293	'1001'
'011'	0.0293	'1000'
'100'	0.0293	'111'
'101'	0.0019	'11001'
'110'	0.0156	'11000'
'111'	0.0156	'1101'

On peut ensuite en déduire grâce à 6.1.1 l'entropie : 1.1121 bits.
Et grâce à 6.1.2 on en déduire la longueur moyenne de notre codage : 1.4609 qui est bien sûr supérieur à l'entropie de la source.

3.3 Compression d'un texte

La fonction de compression d'un texte repose sur la détermination de la distribution de probabilité pour le texte. Pour celà on va définir la probabilité d'un caractère comme sa fréquence d'apparition dans le texte. On va ensuite appliquer l'algorithme de Huffman 6.2.

3.3.1 Fonction de compression

On va prendre en argument un texte, on va déterminer la fréquence d'apparition de chaque caractère. Puis on va appliquer l'algorithme de Huffman avec la distribution de probabilité ainsi obtenue. Ensuite, pour comprimer effectivement le texte, on va s'aider d'un dictionnaire (table de hachage) qui à chaque motif associera son code. On parcourt alors notre texte initial et on remplace chaque caractère par son code.
Au bilan, on obtient en sortie, le texte comprimé, et le codage de Huffman (correspondance motif et code) qui permettre de le décompresser. Cela donne le code fourni en 6.3.1.

3.3.2 Fonction de décompression

Pour la décompression, on va parcourir le texte progressivement et dès que l'on trouve une section qui correspond à un motif on la décode. Et on

avance comme celà progressivement. Cela n'est bien sur possible que parce le codage est uniquement déchiffrable. On a ajouté également un affichage qui permet de donner quelques étapes de la décompression, comme elle s'avère très longue. On obtient ainsi le code en 6.3.2. Il fonctionne très bien : le texte décompressé correspond exactement au texte compressé.

3.3.3 Extraire le texte d'un fichier texte

Pour aller plus loin, nous avons essayé de manipuler des fichiers. L'idée est de fournir en entrée un fichier texte, d'extraire le texte du fichier grâce à python, et de générer ensuite deux fichiers : le texte compressé, et un fichier qui code la correspondance de Huffmann entre chaque motif et son code (pour pouvoir décompresser). Pour enregistrer cette correspondance nous avons utilisé le module marshal de python qui permet ce stockage.
Ensuite nous avons également crée une fonction de décompression qui prend en entrée les deux fichiers (fichier compressé et dictionnaire de correspondance) et qui génère un fichier décompressé.
Malgrès beaucoup de problèmes initialement nous avons réussi à tout faire fonctionner correctement. La fonction de compression est visible 6.4.1 et la fonction de décompression est visible en 6.4.2. Nous avons par exemple essayé de compresser le tome1 des Misérables de Victor Hugo (700-800 pages). Et nous obtenu un rapport de compression (taille du texte compressé divisé par taille originale) de 56% !

3.4 Compression d'une image RGB

3.5 Fonction de compression

On récupère une image en RGB dont la couleur de chaque pixel est donné par un triplet de nombre. On met les tout bout à bout, et on a juste à appliquer notre fonction précédente en mettant entrée le texte composé des nombres. On stocke aussi la taille de l'image au début du texte. Pour gérer les images avec Python, on utilise le module PIL. L'implémentation est visible en 6.5.1.

3.6 Fonction de décompression

On doit d'abord récupérer le texte qui correspond à l'image crypté. On le décode puis on récupère la taille de l'image. On enlève les caractères qui ne nous serviront pas. Puis on reconstitue les triplets correctement. On peut ensuite générer une nouvelle image. Le code correspond est visible ici en 6.5.2. On arrive bien à reconstituer exactement la même image.

4 Conclusion

Nous avons réussi à comprendre les mécanismes et les outils théoriques derrière le codage de Huffmann. Ensuite, nous avons pu implémenter ce dernier puis nous l'avons appliqué à un texte, contenu dans un fichier, et même à une image. Nous avons pu remarquer l'efficacité de ce codage et son respect des propriétés théoriques attendues (longueur moyenne plus grande que l'entropie de la source par exemple). Toutefois, nous avons noté le temps d'exécution très long, dès lors qu'on cherche à décompresser une image par exemple (la compression étant beaucoup moins longue). Ce qui a nécessité par exemple dans notre code python, d'ajouter des jalons au fur et à mesure de l'avancement d'une décompression, pour avoir une idée du temps qu'il restait. On a ainsi compris pourquoi le codage de Huffmann est en pratique couplé à d'autres algorithmes de compression (pour le format JPEG par exemple). Il s'agit d'optimiser le temps de compression/decompression et surtout on peut aussi inclure des compressions avec perte (tant qu'elles ne détériorent pas trop l'image ou le son, à l'échelle de l'oeil ou l'oreille humain(e)). Cela permet aussi d'améliorer le rapport de compressions (ce qui est normal puisqu'on introduit de la perte) et aussi le temps de compression ou de décompression.
En définitive, nous avons tous deux apprécié ce projet, car il était stimulant et assez ouvert pour que l'on puisse s'orienter dans la direction qui nous plaisait le plus. Vous pouvez trouver nos fichiers de code python en annexe ou sur le dépôt github suivant : https ://github.com/dedicad/codageEntropique.

5 Bibliographie

6 Annexe

6.1 Petites Fonctions

6.1.1 Calcul de l'entropie

```
def entropie(tab_proba):
    '''Cette fonction calcule l'entropie du tableau de probabilitÃ© fourni en
      entree sous la forme de tuple (frequence, motif)'''
    somme = 0
    for (pi,elm) in tab_proba:
        somme += pi*log(pi,2)
    return -somme
```

6.1.2 Longueur moyenne d'un codage

```
    def longueur_moyenne(source,codage):
    '''Cette fonction calcule la longueur moyenne d'un codage. Elle prend en
      entree une liste de tuple (frequence,motif) et une autre liste de tuple (
      code, motif).'''
    long = 0
    for (freq,motif) in source:
        for (code,motifBis) in codage :
            if motif==motifBis :
                long += len(code)*freq
    return long
```

6.2 Codage de Huffman

```
def Huffman(tab_proba):
    """Cette fonction prend en entree une liste de tuple (frequence,motif)
    et retourne une liste de tuple (code, motif)"""
    m = len(tab_proba)
    C = [("0","Initial")]*m
                                                        #C est le tableau de
       tuple que l'on renverra, il est compose de (code, motifACoder)
    if m == 2 :
                                                                #Cas de
      base de notre algorithme recursif
        C[0],C[1] = ("0",tab_proba[0][1]),("1",tab_proba[1][1])
    else :
        tab_proba.sort(reverse = True)
                                                    #On trie la liste des proba
           dans le sens decroissant
        temp = Huffman(tab_proba[0:-2] + [(tab_proba[-2][0]+tab_proba[-1][0],"new
          ")]) #On applique le procede sur une liste de taille m-1
        indice = recup(temp,"new")
        #On regarde ou l'element "special" s'est positionne suite au tri
        temp.append(temp[indice])
        temp.pop(indice)
                                                                #On l'enleve
          pour le repositionner a la fin de la liste
        for i in range(m-2):
                                                            #On recupere les
          codes pour les m-2 premiers motifs
            C[i] = temp[i]
        C[m-2] = (temp[-1][0]+"0",tab_proba[-2][1])
                                            #On cree les codes pour les 2 derniers
          motifs
        C[m-1] = (temp[-1][0] + "1",tab_proba[-1][1])
    return C
```

6.3 Compression d'un texte

6.3.1 Fonction de compression

```
def code(texte):
    '''Cette fonction prend en entree un texte sous forme de string, calcule la
      frequence d'apparition de chaque symbole,
    et retourne en sortie le code de Huffman obtenu ainsi que le texte code'''
    n = len(texte)
    if n<2 : return ('Le texte est court')
    else :
        dico = {}
        #On calcule la distribution de probabilite correspondant a notre texte
        for lettre in texte:
            if lettre not in dico :
                dico[lettre] = 1
            else : dico[lettre] += 1
        tab = []
        for cle in dico.keys():
            tab.append((dico[cle]/n,cle))
        sol = Huffman(tab)
        #On recree un dictionnaire parce que c'est plus facile a utiliser
        dico_code = {}
        for (code,motif) in sol:
            dico[motif] = code
        res = ""
        #On code le texte grace a notre dictionnaire
        for lettre in texte:
            res += dico[lettre]
        return (res,sol)
```

6.3.2 Fonction de décompression

```
def decode(texte,codage):
    '''Cette fonction prend en entree un texte crypte par Huffman et le codage de
       Huffman associe.
    Elle retourne le texte decode'''
    #On cree un dico dont la cle est le code et la valeur le motif associe, parce
       que c'est plus facile a utilisesr
    dico = {}
    for (code,motif) in codage:
        dico[code] = motif
```

```
    res = ""
    tps = time()
    n = len(texte)
    while len(texte)>0:
        j = 1
        section = texte[0:j]
        #On determine la prochaine section a decoder
        while section not in dico :
            j += 1
            section = texte[0:j]
        #On a trouve la section, on utilise alors juste la correspondance code
          <--> mottif, fournie par le dictionnaire "codage"
        res += dico[section]
        texte = texte[j:]
        #Les lignes qui suivent permettent de suivre l'avancement de la fonctions
        if time()-tps > 10:
            print ("On␣avance")
            tps = time()
        if len(texte)%((n//100+1)*2) ==0 :
            print ("Il␣reste␣",len(texte)//(n//100+1),"%␣du␣texte␣a␣decoder.")
    return res
```

6.4 Compression à partir d'un fichier

6.4.1 Fonction de compression

```
def code_fichier(fichier_clair,fichier_crypte,fichier_codage):
    '''Cette fonction prend en entree le nom d'un fichier texte a crypte et les
      noms des fichiers ou on stockera les sorties et retourne en sortie le code
      de Huffman obtenu
     ainsi que le texte code dans le fichier fichier_crypte, qu'elle cree s'il n'
       existe pas'''
    #On extrait le texte du fichier
    fichier = open(chemin+fichier_clair,'r')
    contenu = fichier.read()
    fichier.close()
    #On code le texte grace a notre fonction qui code un texte (qui s'appuie elle
       meme sur le codage de Huffmann)
    temp = code(contenu)
```

```
    #On ecrit dans un fichier le texte compresse obtenu
    fichier = open(chemin+fichier_crypte,'w')
    fichier.write(temp[0])
    fichier.close()
    #On stocke la correspondance de Huffmann code <--> motif dans un fichier
      texte grace au module marshal
    marshal.dump(temp[1], open(chemin+fichier_codage,"wb"))
    print ("Le codage a bien ete effectue")
```

6.4.2 Fonction de décompression

```
def decode_fichier(fichier_codage,fichier_crypte,fichier_cible):
    '''Cette fonction prend en entree un fichier contenant le texte crypte, un
      autre fichier contenant le codage de Huffman associe, et le fichier ou on
      ecrira le texte en clair'''
    #On recupere le texte compresse
    fichier = open(chemin+fichier_crypte,'r')
    contenu = fichier.read()
    fichier.close()
    #On recupere la correspondance motif <--> code et on applique notre fonction
      de decodage
    res = decode (contenu,marshal.load(open(chemin+fichier_codage,"rb")))
    #On ecrit le texte decode dans un fichier
    fichier = open(chemin+fichier_cible,'w')
    fichier.write(res)
    fichier.close()
    print ("Le fichier a ete decode")
```

6.5 Compression d'une image en RGB

6.5.1 Fonction de compression

```
def code_image(image,fichier_codage,fichier_crypte):
    '''Cette fonction prend en parametre une image a coder et un fichier ou
      enregistrer le codage de Huffman associe et le fichier ou sera stocke l'
      image crypte sous forme binaire'''
    im = Image.open(chemin+image)
```

```
    liste = (list(im.getdata()))
    #On stocke la taille de l'image
    contenu = str(im.size[0])+";"+str(im.size[1])+";"
    #Ici on va aplatir la liste et la convertir en une chaine de caractere
    for pixel in liste :
        contenu += str(pixel)+";"
    #Le ; sert de separateur, comme dans les fichiers csv
    #On code le contenu recupere comme si c'etait un texte : on code les nombres
      chiffre par chiffre
    temp = code(contenu)
    #On ecrit le resultat dans un fichier
    fichier = open(chemin+fichier_crypte,'w')
    fichier.write(temp[0])
    fichier.close()
    #On stocke la correspondance code <--> motif dans un autre fichier
    marshal.dump(temp[1], open(chemin+fichier_codage,"wb"))
    print ("Le codage a bien ete effectue")
```

6.5.2 Fonction de décompression

```
def decode_image(image_crypte,fichier_codage,fichier_cible):
    #On recupere le texte code
    fichier = open(chemin+image_crypte,'r')
    temp = fichier.read()
    fichier.close()
    #On decode le texte apres avoir recupere le dictionnaire de correspondance
    res = decode (temp,marshal.load(open(chemin+fichier_codage,"rb")))
    #1e etape : on determine les donnees de taille de l'image
    long_temp = []
    i = 0
    for m in range(2):
        j = i+1
        section = res[i:j]
        while res[j] != ';':
            j += 1
        section = res[i:j]
        long_temp.append(section)
        i= j+1
    long = (int(long_temp[0]),int(long_temp[1]))
```

```
    contenu = []
    #On enleve les parentheses parce qu'elles nous gÃªnent
    res = res.replace('(','')
    res = res.replace(')','')
    n = len(res)
    #Maintenant on determine les valeurs de chaque pixel
    while i<n:
        j = i+1
        section = res[i:j]
        while res[j] != ';':
            j += 1
        section = res[i:j]
        #On a maintenant isole une section du type '107,103,22' que l'on doit
          transformer en tuple
        pix = []
        for l in range(2):
            m = 0
            p = 1
            while section[p] != ',':
                p += 1
            pix.append(int(section[m:p]))
            section = section[p+1:]
        pix.append(int(section))
        pix = (pix[0],pix[1],pix[2])
        contenu.append(pix)
        i= j+1
    #On cree l'image de la bonne taille
    image_finale = Image.new('RGB',long)
    #On y insere dedans les valeurs des pixels
    image_finale.putdata(contenu)
    #On enregistre l'image sur notre disque dur
    image_finale.save(chemin+fichier_cible)
    print ("Le␣fichier␣a␣ete␣decode")
```