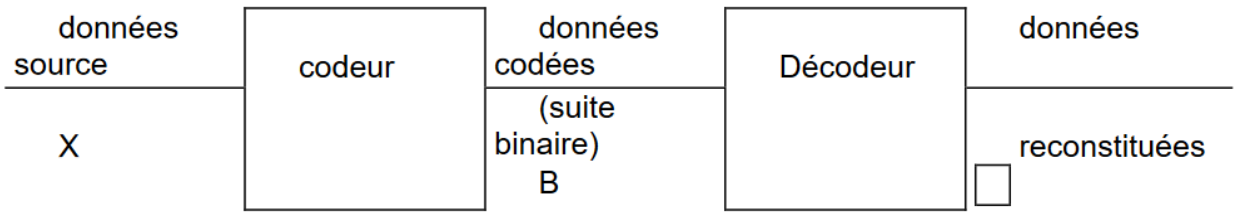
1.Introduction

Le codage de source ou compression des données sert à fournir une représentation efficace des données (un taux de compression important) tout en préservant l'information essentielle qu'elles portent. Il est employé pour le stockage ou la transmission de ces données (on appelle données le résultat de la numérisation de signaux comme ceux de parole ou d'images ou plus généralement les données disponibles sur un fichier d'ordinateur) .
Le codage de source est d'autre part connecté à d'autres applications techniques telles que la classification d'images, la reconnaissance vocale,...
Le bloc diagramme d'un système de compression est le suivant :



La théorie de l'information développée par Shannon dans les années 1940-1950 fournit des idées fondamentales pour la conception d'un grand nombre de techniques de codage de source. Une idée est que des messages numériques peuvent être comprimés en associant les mots de code les plus courts aux messages les plus probables et que le taux de compression (sans perte) maximum que l'on peut atteindre peut être déterminé à partir d'une description statistique des messages. Cette stratégie conduira à un codage à débit variable et à des techniques de codage sans perte (ou codage entropique) c'est-à-dire telles que les données reconstituées soient identiques aux données source.

Une autre idée est que les systèmes de codage sont plus performants lorsqu'ils opèrent sur des vecteurs (ensemble d'échantillons) plutôt que des échantillons individuels. Cette idée a été indirectement exploitée dans les années 70 lorsque l'on a effectué des pré-traitements tels que la prédiction ou une transformation avant de quantifier scalairement les échantillons. Elle a ensuite été pleinement utilisée dans les années 80 avec les techniques de quantification vectorielle.
Toutes ces techniques (qui opèrent une quantification des données) font partie des techniques de compression avec perte. Les données reconstruites ne coïncident pas avec les données source mais sont (idéalement) perceptuellement non distinctes de celles-ci.
La plupart du temps le système obtenu pour ces techniques est à débit fixe.

Exemple :
On cherche à comprimer l'image 4x4 suivante où chaque pixel est caractérisé par une couleur parmi 4 : R = « rouge », J = « jaune », B = « bleu », V = « vert » .

R	R	R	R
R	B	R	B
R	B	V	B
R	J	J	J

-Codage sans perte
La méthode triviale pour transmettre cette information sous forme binaire est d'associer un mot de code sur 2 bits à chaque couleur.
Ex : R = « 00 » B = « 01 » V = « 10 » J = « 11 »
l'image peut être codée sur 32 bits en parcourant les lignes de l'image de haut en bas et de gauche à droite :
00 00 00 00, 00 01 00 01, 00 01 10 01, 00 11 11 11
Une méthode plus efficace tiendra compte des probabilités de chaque couleur en opérant l'affectation de mots de taille différente à chaque couleur
R = « 1 » B = « 01 » J = « 001 » V = « 000 » ce qui conduit à coder l'image sur 28 bits.

11111 01 1 01 1 01 000 01 1 001 001 001

1.2 Le codage de Huffman

C'est un algorithme de compression de données sans perte. Le codage de Huffman utilise un code à longueur variable pour représenter un symbole de la source (par exemple un caractère dans un fichier). Le code est déterminé à partir d'une estimation des probabilités d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents.
Un code de Huffman est optimal au sens de la plus courte longueur pour un codage par symbole, et une distribution de probabilité connue. Il a été inventé par David Albert Huffman, et publié en 1952.

1.2.1 Principe

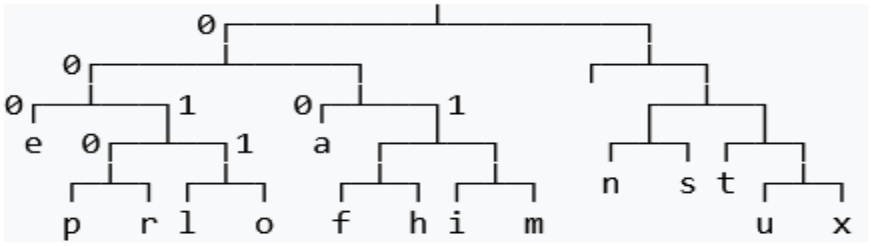
Le principe du codage de Huffman repose sur la création d'une structure d'arbre composée de nœuds.
Supposons que la phrase à coder est « *this is an example of a huffman tree* ». On recherche tout d'abord le nombre d'occurrences de chaque caractère. Dans l'exemple précédent, la phrase contient 2 fois le caractère *h* et 7 espaces. Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids égal à son nombre d'occurrences.
Occurrence de chaque caractère, trié du plus fréquent au moins fréquent :

Nœud		a	e	f	h	i	m	n	s	t	l	o	p	r	u	x
Occurrence	7	4	4	3	2	2	2	2	2	2	1	1	1	1	1	1

L'arbre est créé de la manière suivante, on associe chaque fois les deux nœuds de plus faibles poids, pour donner un nouveau nœud dont le poids équivaut à la somme des poids de ses fils. Premier nœud avec u et x (de poids 2 = 1 + 1), deuxième nœud avec p et r et troisième nœud avec l et o.

Nœud		a	e	f	h	i	m	n	s	t	⌐ u x	⌐ p r	⌐ l o
Occurrence	7	4	4	3	2	2	2	2	2	2	2	2	2

Quatrième nœud avec p-r et l-o de poids total 4 :



Et cette table de codage binaire :

Caractère	Code binaire
e	000
p	00100
r	00101
l	00110
o	00111
a	010
f	01100
h	01101
i	01110
m	01111
	10
n	1100
s	1101
t	1110
u	11110
x	11111

Pour obtenir le code binaire de chaque caractère, on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie. Il est nécessaire de partir de la racine pour obtenir les codes binaires car sinon lors de la décompression, partir des feuilles peut entraîner une confusion lors du décodage.

La phrase « *this is an example of a huffman tree* » se code alors sur 137 bits au lieu de 288 bits (si le codage initial des caractères tient sur 8 bits).

Il existe trois variantes de l'algorithme de Huffman, chacune d'elles définissant une méthode pour la création de l'arbre :

a. statique

Chaque octet a un code prédéfini par le logiciel. L'arbre n'a pas besoin d'être transmis (ex. : un texte en français, où les fréquences d'apparition du 'e' sont énormes ; celui-ci aura donc un code très court).

b. semi-adaptatif

Le fichier est d'abord lu, de manière à calculer les occurrences de chaque octet, puis l'arbre est construit à partir des poids de chaque octet. Cet arbre restera le même jusqu'à la fin de la compression. Cette compression occasionnera un gain de bits supérieur ou égal au codage de

Huffman statique mais il sera nécessaire, pour la décompression, de transmettre l'arbre, ce qui annulera généralement le gain obtenu.

c. adaptatif

C'est la méthode qui offre a priori les meilleurs taux de compression car il utilise un arbre connu (et ainsi non transmis) qui sera ensuite modifié de manière dynamique au fur et à mesure de la compression du flux selon les symboles précédemment rencontrés. Cette méthode représente cependant le gros désavantage de devoir modifier souvent l'arbre, ce qui implique un temps d'exécution plus long. Par contre, la compression est toujours optimale et il n'est pas nécessaire que le fichier soit connu *avant* de compresser. En particulier, l'algorithme est capable de travailler sur des flux de données, car il n'est pas nécessaire de connaître les symboles à venir.

1.2.2 Performance de codage de Huffman

Prenons par exemple, un cas ou les probabilités suivantes sont associées à des éléments :

a1	a2	a3	a4	a5	a6	a7
0.38	0.24	0.1	0.1	0.1	0.05	0.03

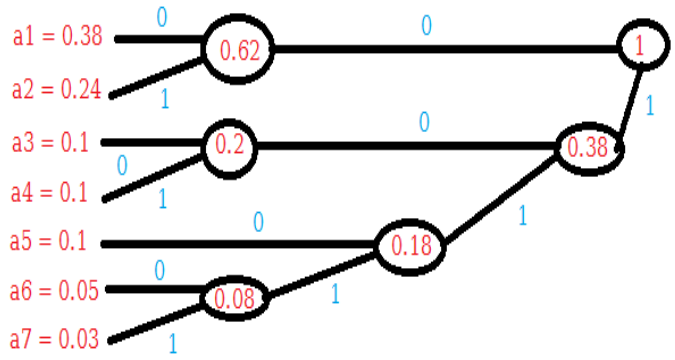
On commence par calculer H, la moyenne par symbole :

$H = -\Sigma\, pi*log2(pi)$:

$H = -[(0.38*log2(0.38)) + (0.24*log2(0.24)) + 3*(0.1*log2(0.1)) + (0.05*log2(0.05)) + (0.03*log2(0.03))]$

$H = 2.39\ sh/s$ (Shannon/ seconde)

Maintenant nous allons passer à la détermination du code de chaque symbole.



On doit additionner les probabilités pour arriver à 1. A chaque fois qu'un branche rejoint l'autre, on lui associe le nombre **1**, et quand une branche reste au même niveau, on lui associe le nombre **0**. Ensuite, pour la lecture du code, on part du résultat total de la somme (notre 1 final), et on découle jusqu'à arriver vers la probabilité de départ. Exemple : Ici le code du symbole de probabilité a1 sera égal à 00, car il y a deux zéros entre le 1 final et la probabilité a1. Pour a4, ça sera 101, pour a5, 110; etc …

Voici le tableau final trouvé :

Si	Pi	Code	Li
a1	0.38	00	2
a2	0.24	01	2
a3	0.1	100	3
a4	0.1	101	3
a5	0.1	110	3
a6	0.05	1110	4
a7	0.03	1111	4

- **Si** est le nom de la probabilité associée au symbole transmis.
- **Pi** la probabilité d'apparition du symbole associée.
- **Code** le code binaire qui représente le symbole.
- **Li** la longueur du code (le nombre de chiffres du code).

Maintenant on arrive au calcul de l'efficacité de compression du message, qui se traduit par :

*Ec = H / Σ(Pi*Li)*

Dans notre cas, Ec donne donc :

*Ec = 2.39 / (0.38*2 + 0.24*2 + 3*(0.1*3) + 0.05*4 + 0.03*4)* <=> *Ec = 2.39 / 2.46 = 97.6 %*

Notre efficacité sera donc de 97.6 %.

1.2.3 Utilisations

Le codage de Huffman ne se base que sur la fréquence relative des symboles d'entrée (suites de bits) sans distinction pour leur provenance (images, vidéos, sons, etc.). C'est pourquoi il est en général utilisé au second étage de compression, une fois la redondance propre au média mise en évidence par d'autres algorithmes. On pense en particulier à la compression JPEG pour les images, MPEG pour les vidéos et MP3 pour le son, qui peuvent retirer les éléments superflus imperceptibles pour les humains. On parle alors de compression non conservative (avec pertes).
D'autres algorithmes de compression, dits conservatifs (sans pertes), tels que ceux utilisés pour la compression de fichiers, utilisent également Huffman pour comprimer le dictionnaire résultant. Par exemple, LZH (Lha) et deflate (ZIP, gzip, PNG) combinent un algorithme de compression par dictionnaire (LZ77) et un codage entropique de Huffman.

1.3 Codage de Shannon-Fano

C'est un algorithme de compression de données sans perte élaboré par Robert Fano à partir d'une idée de Claude Shannon. Il s'agit d'un codage entropique produisant un code préfixe très similaire à un code de Huffman, bien que pas toujours optimal, contrairement à ce dernier.
La probabilité de chaque symbole à compresser doit être connue. Dans la plupart des cas, des fréquences ou occurrences fixes calculées à partir des données à compresser sont utilisées ; on parle alors de codage de Shannon-Fano semi-adaptatif. Il est également possible d'utiliser des probabilités fixes non dépendantes des données à compresser (codage statique) ou des probabilités variant au fur et à mesure de la compression (codage adaptatif).
Le processus est de trier les symboles à compresser par ordre croissant de leurs nombre d'occurrences. Puis l'ensemble trié des symboles est séparé en deux groupes de telle façon que le nombre d'occurrences des deux parties soient égaux ou presque (le nombre d'occurrences d'un groupe étant égale à la somme des occurrences des différents symboles de ce groupe). On réitère le processus jusqu'à obtenir un seul arbre. On associe ensuite par exemple le code 0 à chaque embranchement partant vers la gauche et le code 1 vers la droite.

1.3.1 Principe détaillé

L'algorithme se base sur le nombre d'occurrence de chaque caractère pour établir la table des codes binaires de taille variable.

Supposons que la phrase à coder est « *this is an example of a huffman tree* ». Il s'agit de la même phrase que pour le chapitre sur le codage de Huffman afin de pouvoir comparer les résultats.

On recherche tout d'abord le nombre d'occurrences de chaque caractère. Dans l'exemple précédent, la phrase contient 2 fois le caractère *h* et 7 espaces. Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids égal à son nombre d'occurrences.

Contrairement à l'algorithme d'Huffman, le tri n'est fait qu'une seule fois. De plus, le code binaire se construit au fur et à mesure, en commençant par diviser le tableau en deux parties auxquelles on associe le bit 0 et le bit 1 respectivement. Chaque division doit produire deux moitiés de poids égaux si possible. Chaque moitié est alors divisée à son tour, jusqu'à ce qu'il ne soit plus possible de diviser. À chaque étape, on ajoute une ligne au tableau indiquant les divisions effectuées, **le bit associé en gras**, et le poids en gris :

Nœud		a	e	f	h	i	m	n	s	t	l	o	p	r	u	x
Occurrence	7	4	4	3	2	2	2	2	2	2	1	1	1	1	1	1
Étape 1	**0** (18)				**1** (18)											
Étape 2	**0** (11)		**1** (7)		**0** (10)					**1** (8)						
Étape 3	**0** (7)	**1** (4)	**0** (4)	**1** (3)	**0** (6)			**1** (4)		**0** (4)			**1** (4)			
Étape 4					**0** (4)		**1** (2)	**0** (2)	**1** (2)	**0** (2)	**1** (2)		**0** (2)		**1** (2)	
Étape 5					**0** (2)	**1** (2)					**0** (1)	**1** (1)	**0** (1)	**1** (1)	**0** (1)	**1** (1)

On obtient donc cette table de codage binaire :

Caractère	Code binaire
	000
a	001
e	010
f	011
h	10000
i	10001
m	1001
n	1010
s	1011
t	1100
l	11010
o	11011
p	11100
r	11101
u	11110
x	11111

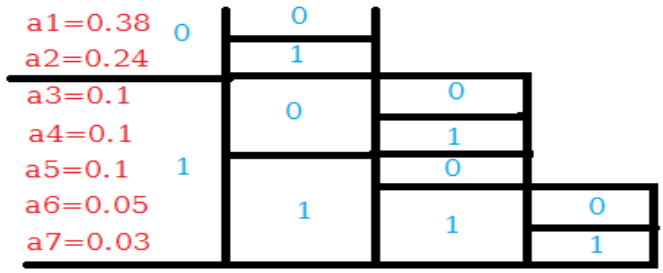
La phrase « *this is an example of a huffman tree* » se code alors sur 136 bits au lieu de 288 bits (si le codage initial des caractères tient sur 8 bits).

Exemple

Le codage de Shannon-Fano ressemble énormément à Huffman, la seule différence réside dans le fait qu'on choisit arbitrairement les codes associés aux probabilités des symboles. Pour être plus clair, on va à chaque fois découper en deux parties les plus égales, et attribuer à chaque partie un **0** et un **1** suivant les étapes suivantes :

1. Ceux-ci sont classés par ordre décroissant de la probabilité des symboles de source d'émission ;
2. Pour chaque étape du codage sont formés de deux ensembles pour lesquels la somme de la probabilité d'émission est la plus similaire possible ;
3. A chacune des deux parties vous sont affectées d'une « 1 » ou « 0 » arbitraire ;
4. Pour chaque sous-ensemble j'obtenir à partir de chaque étape, sont effectuées les étapes 2 et 3 jusqu'à ce que les sous-ensembles d'un seul élément ;
5. Le code pour chaque symbole sera obtenu par séquençage « 0 » et « 1 » à partir de la première étape jusqu'à la dernière.

Le schéma ci-dessous représente l'arborescence des codes de chaque probabilité.



Dans le tableau, on prend ici **a1** et **a2**, (dont la somme fait 0.62) et tout le reste, (dont la somme fait 0.38). On les sépare en deux, on met le zéro dans la partie du dessus, et les 1 dans la partie du bas. Puis on recommence pour chaque morceau de probabilité, on sépare le groupe du dessus en 2, et celui du dessous entre **a3** et **a4** pour le chiffre 0, et **a5,a6** et **a7** pour 1.

Lorsqu'on a séparé tous les groupes, il suffit de tracer des lignes entre chaque probabilité (de a1 à a7 ici), et de lire en partant de la droite, la ligne correspondante. Pour **a7**, on lit 1111.

On remarquera qu'ici on retrouve le même résultat qu'avec la méthode de Huffman, mais ceci, parce que nous avons déterminé à chaque fois les deux blocs étant les plus égaux l'un envers l'autre. Si l'évaluation est mauvaise, on trouvera forcément un résultat inférieur à la méthode de Huffman, qui est donc plus performante.

1.4 Codage Arithmétique

Le codage arithmétique est un codage statistique, c'est-à-dire que plus un caractère est représenté, moins il faudra de bits pour le coder.
Il s'agit d'un cousin du codage de Huffman qui cependant reste toujours plus efficace que ce dernier (sauf dans le cas particulier où tous les poids des feuilles/nœuds/racines de l'arbre de Huffman sont des puissances de 2). Il est aussi plus simple à implémenter.
L'avantage que possède le codage arithmétique sur le codage de Huffman est que ce dernier va coder un caractère sur un nombre entier de bits (il ne peut coder sur 1.5 bits) là où le codage arithmétique le peut. Par exemple, si un caractère est représenté à 90%, la taille optimale du code du caractère serait de 0.15 bit, alors que Huffman coderait sûrement ce symbole sur 1 bit, soit 6 fois trop.
Ce codage n'est que très peu utilisé en pratique mais elle reste présente, notamment dans le format JPEG2000.

1.4.1 Principe

Le codage arithmétique (au même titre que le Codage de Huffman) est un code à longueur variable, c'est-à-dire qu'un symbole de taille fixe (en bits) sera codé par un nombre variable de bits, de préférence inférieur ou égal à sa taille originale.
Ce qui différencie le codage arithmétique des autres codages sources est qu'il code le message par morceaux (théoriquement il peut coder un message entier de taille quelconque mais dans la pratique on ne peut coder que des morceaux d'une quinzaine de symboles en moyenne) et représente chacun de ces morceaux par un nombre n (flottant) là où Huffman code chaque symbole par un code précis. Le problème qui en résulte pour le codage Huffman est qu'un caractère ayant une probabilité très forte d'apparition sera codé sur au moins un bit. Par exemple, si on cherche à coder un caractère représenté à 90 %, la taille optimale du code du caractère sera de 0,15 bit alors que Huffman codera ce symbole sur au moins 1 bit, soit 6 fois trop.

1.4.2 Compression

La compression demande un tableau statistique (se rapprochant au maximum des statistiques observables sur le fichier f_n contenant n symboles à compresser) qui comprend :

- La totalité des symboles que l'on rencontre dans le message à compresser.
- Les probabilités p de rencontrer le symbole s dans le message.
- L'intervalle [0 ; 1[découpé en intervalles proportionnel à la probabilité p que le symbole apparaisse dans le message (ex: si s a 50 % de chance d'apparaître, son intervalle sera de longueur 0,5).

Le but va maintenant être d'appliquer une suite d'opérations à un intervalle donné (couramment c'est l'intervalle [0 ; 1[qui est choisi) afin de modifier ses bornes à chaque ajout d'un symbole s et de restreindre au maximum le nombre de possibilités du nombre de sortie.
Voici les opérations à effectuer lors de l'ajout d'un symbole s :

- On enregistre la différence entre la borne supérieure (BS) et la borne inférieure (BI). On notera cette valeur BB.
- La BS prend la valeur suivante : BI + BB * (BS_du_symbole_s)
- La BI prend la valeur suivante : BI + BB * (BI_du_symbole_s)

Après l'ajout du premier symbole, l'intervalle aura les mêmes limites que l'intervalle du symbole ajouté, puis chaque ajout fera converger ses limites. Lorsqu'un nombre suffisant de symboles

auront été stockés, l'algorithme renverra un nombre flottant se trouvant dans cet intervalle. Ce nombre représente la totalité du fichier d'entrée.
On remarque qu'une suite de symboles ayant une forte probabilité d'apparaitre fera converger l'intervalle bien plus lentement que si on a affaire à une suite de symbole apparaissant peu. Un nombre dans un intervalle ayant convergé lentement sera donc codé avec moins de chiffres significatifs qu'un nombre dans un intervalle extrêmement précis, d'où le gain.

1.4.3 Décompression

Pour décompresser un fichier (représenté par un nombre *n*), il faut utiliser la même table qui a été utilisée pour la compression puis effectuer les étapes suivantes jusqu'à la fin du fichier :

- Observer dans l'intervalle de quel symbole *s* se trouve le nombre *s*, ajouter au fichier décompressé et garder en mémoire la probabilité *p* de *s* ainsi que sa BI.
- n prend la valeur $n = (n - BI)/p$.

Une fois le marqueur de fin atteint, l'algorithme s'arrête et le fichier est considéré comme décompressé.

D'où le nombre de bits du code associé à la séquence S : $NB = \lceil \log_2(1/p(S)) \rceil$

Exemple 1

-Pour présenter la compression, nous allons utiliser un exemple et nous décrirons chaque étape de compression. Codons le mot "ESIPE" à l'aide du codage arithmétique.
La première étape consiste à décompter chaque lettre du mot. Nous avons donc 2 'E', 1 'S', 1 'I' et 1 'P'. Nous en générons alors une probabilité de présence dans le mot soit 40% de chance de trouver un E et 20% de chance pour les autres lettres. Dernière actions à effectuer pour cette première partie, nous affectons à chaque lettre un intervalle entre 0 et 1 de la manière suivante :

- La lettre 'E' à une probabilité de 40% (soit 0.4). Son intervalle est donc [0,0.4[
- La lettre 'P' a une probabilité de 20% (soit 0.2). Son intervalle est donc [0.4,0.6[
- Etc...

On obtient dès lors le tableau suivant :

Lettre	Probabilité	Intervalle
E	4/10	[0,0.4[
S	2/10	[0.4,0.6[
I	2/10	[0.6,0.8[
P	2/10	[0.8,1.0[

Le codage va maintenant consister à remplacer le mot ESIPE par un nombre flottant lui correspondant. Pour cela, le mot va se voir affecter un intervalle compris entre 0 et 1 où chaque nombre compris entre les deux intervalles permettra de retrouver le mot ESIPE.
L'algorithme appliqué est le suivant : le mot commence avec un intervalle de [0,1[. Puis pour chaque lettre croisée, nous appliquons la formule suivante :

- La borne inférieure (BI) du mot est modifiée avec le résultat du calcul "BI + (BS – BI) * Borne_Inférieure_Lettre"
- La borne supérieure (BS) du mot est modifiée avec le résultat du calcul "BI + (BS – BI) * Borne_Supérieure_Lettre"

Le tableau suivant montre les étapes du calcul:

Lettre	Borne Inférieure	Borne Supérieure
	0.0	1.0
E	0.0	0.4
S	0.16	0.24
I	0.208	0.224
P	0.2208	0.224
E	0.2208	0.22208

Dès lors, tous nombre flottant entre 0.2208 et 0.22208 est le format compressé du mot "ESIPE".
- De la même manière, nous allons présenter la décompression par l'exemple en décompressant notre format compressé.

Prenons le nombre 0.2208 qui code le mot "ESIPE". Le principe de la décompression est très simple. Celle-ci suit les deux étapes suivantes qui se répète jusqu'à l'obtention du mot :

- La prochaine lettre du mot est celle dont l'intervalle contient le nombre du mot actuel (Ex : 0.2208 est dans l'intervalle de E donc la première lettre est E).
- On modifie le nombre représentant le mot à l'aide du calcul « (nombre du mot – borne inférieure de la lettre) / probabilité de la lettre (Ex : nombre du mot = (0.2208 – 0.0) / 0.4 = 0.552)

Le tableau suivant montre les différentes étapes de la décompression :

Mot	Lettre	Nouveau code
	E	0.552
E	S	0,76
ES	I	0,8
ESI	P	0,0
ESIP	E	

Ainsi, nous avons récupéré notre premier mot : ESIPE

Exemple 2

Compression

On appliquera ici l'algorithme de compression arithmétique sur le texte « WIKI ». On obtient dès lors le tableau suivant :

Lettre	Probabilité	Intervalle
W	1/4	[0;0,25[
I	2/4	[0,25;0,75[
K	1/4	[0,75;1[

Par convention on initialise l'algorithme avec une borne inférieure valant 0 et une borne supérieure valant 1. Il ne reste plus qu'à appliquer la suite d'opérations vue précédemment à chaque ajout d'un caractère.

Caractère ajouté	Borne inférieure	Borne supérieure
	0	1
W	0	0,25

I	0,0625	0,1875
K	0,15625	0,1875
I	0,1640625	0,1796875

Donc, tout nombre compris dans intervalle [0,1640625; 0,1796875 [sera une version compressée de la chaîne de caractère « WIKI ». Le nombre 0,17 étant compris dans cet intervalle, il peut convenir pour représenter « WIKI » compressé. À l'inverse, 0,16 ou 0,1796875 n'étant pas dans l'intervalle, ils ne conviendront pas et entraineront des erreurs lors du décodage.

Décompression

Supposons que l'on reçoive le message compressé 0,17, voici comment il serait décodé :
(On utilise évidemment le même tableau que précédemment pour connaitre les intervalles de chaque lettre et leurs probabilités d'apparition).

Code compressé	Intervalle	Lettre correspondante	Texte récupéré
0,17	[0;0,25[	W	W
0,68	[0,25;0,75[	I	WI
0,86	[0,75;1[	K	WIK
0,44	[0,25;0,75[	I	WIKI

On retrouve donc la bonne chaine de caractères auparavant compressée.

1.5 Définitions et résultats fondamentaux liés au codage de source

-Régularité

Un code est dit régulier si tous les mots de code mi sont distincts. Tous les codes doivent au moins être réguliers pour permettre un décodage univoque.

-Déchiffrabilité

Un code régulier est déchiffrable si pour toute suite m1 , m2 , ... mn de mots de code il est possible de distinguer les mi sans ambiguïté et reconstruire ainsi les symboles si correspondants. Pour les codes de longueur variable, cette propriété est satisfaite pour les codes dits sans préfixes, obtenus en évitant qu'un mot du code ne soit identique au début d'un autre.

-Codes instantanés

On dit d'un code qu'il est à décodage instantané s'il est possible de décoder les mots de code dès lors que tous les symboles qui en font partie ont été reçus.

-Extension d'un code

L'extension d'ordre n d'un code est le code formé par les séquences de n mots du code initial.

-Condition nécessaire de déchiffrabilité

Pour qu'un code soit déchiffrable il faut et il suffit que toutes ses extensions soient régulières.

-Inégalité de Kraft

Constitue un résultat fondamental en théorie des codes. Elle fournit une condition nécessaire et suffisante d'existence de codes déchiffrables et instantanés, exprimée en fonction de la longueur des mots de code.

Soient l 1 , l2 , ... , l N des longueurs de mots candidates pour coder une source N-aire dans un alphabet binaire. Alors l'inégalité de Kraft :

$$\sum_{i=1}^{N} \frac{1}{2^{li}} \leq 1$$

est une condition nécessaire et suffisante d'existence de codes déchiffrables et instantanés respectant ces longueurs de mots.

-Condition du préfixe :

D'autre part le code n'est utile que si on sait le décoder sans ambiguïté c'est à dire si pour une suite binaire codée de longueur finie il n'y a qu'une séquence de symboles d'entrée possible. Pour satisfaire cette condition sans introduire de séparateurs entre les mots de code , le code doit satisfaire la condition dite du préfixe qui stipule qu'aucun mot de code ne doit être le préfixe d'un autre mot de code.

Exemples : {0,10,101,0101} n'est pas un code à préfixe

{0,10,110,111} est un code à préfixe

La condition du préfixe assure l'unicité du décodage du code