Formats de Documents

Roberto Mantaci

13 mars 2012

Chapitre 1

Introduction

Chapitre 2

Généralités

2.1 Présentation et Principes de la Compression de Données

Compression de Données = convertir un *stream* **(= source, plus approprié que** *fichier***) de données en entrée en une autre de taille plus petite.**

Raisons :
– stockage
– vitesse de transfert

La notion de compression de données est liée à la notion de codage. En fait on parle de codage de la source.

Il existent plusieurs méthodes très diverses pour cela, mais elles sont toutes basées sur un même principe : réduire la redondance**.**

Exemple 1 *Redondance alphabétique : E plus fréquent que Z donc il faut donner à E un mot de code plus court. Un code uniforme (tous les caractères sont codés avec des mots de code de la même longueur) a plus de redondance qu'un code de taille variable, même si il a son utilité.*

Exemple 2 *Redondance Contextuelle : après Q toujours U.*

Exemple 3 *Redondance Contextuelle dans images : plusieurs pixels tendent à avoir la même couleur que leurs voisins.*

> **Principe de Base de la Compression de Données : attibuer codes de petite longueur à événements plus fréquents et codes plus longs aux événements plus rares.**

La compression de données est possible car souvent les données sont représentées de manière redondante (P. ex. ASCII)

Cela explique aussi pourquoi un fichier déjà compressé ne peut pas être compressé ultérieurement : toute la redondance a été éliminée (ou presque). Sinon on pourrait continuer à l'infini. Il y a un niveau d'information "essentielle" au dessous de la quelle on ne peut pas descendre. Il faudra parler de comment mesurer l'information.

> **La plupart des fichiers de données ne peuvent pas être compressés (en fait, tous ceux qui sont random, sans structure, sans regularité et donc sans redondance).**

La plupart des fichiers de données ne peuvent pas être compressés (en fait, tous ceux qui sont random, sans structure, sans regularité et donc sans redondance ne peuvent pas l'être).

Si A, B files, A $\neq$ B, C compressé de A et D de B. Alors C $\neq$ D sinon on ne pourrait pas reconstruire.

Supposons que nous sommes satisfaits de compresser de 50% un fichier. Il y a 2^n fichiers de n bits que nous voudrions compresser en fichiers de $n/2$ bits ou moins. Il y a

$$N = 1 + 2 + 2^2 + \ldots + 2^{n/2} = 2^{n/2+1} - 1$$

de tels fichiers. Donc seulement N des 2^n fichiers peuvent être e□cacement compressés. Mais $N << 2^n$! Si $n = 100$ seulement une fraction $2^{-49} \approx 1,78 \times 10^{-35}$ des fichiers peuvent être compressés.

Si on change 50% avec 90%, cela ne change pas grand chose.

> **Il n'existe pas d'algorithme de compression universel et e□cace. Les méthodes doivent être adaptées au type spécifique de données (texte, images, son, ...)**

2.2 Vocabulaire

Compresseur, encodeur, encoder **: le programme qui compresse les données.**

Décompresseur, décodeur, decoder **: qui convertit dans le sens inverse.**

Codec **: l'ensemble des deux précédents.**

Méthodes de compression non-adaptatives **: qui ne modifient pas les opérations e□ectuées en focntion du type particulier de données à compresser (P. ex. le fax) ;**

Méthodes de compression adaptatives **: examinent les données et en suite modifient certains de leurs paramètres, opérations, etc en fonction des informations collectées sur les données. Parfois ces méthodes arrivent à faire cela avec une seule passe (une seule lecture ou écoute de la source) mais souvent elles ont besoin de deux passes. une pour avoir l'info (stats) et l'autre pour compresser. Dans ce cas elles sont dites semi-adaptatives.**

Compression avec/sans perte **: On a compression avec perte quand le fichier résultant de la décompression n'est pas identique à l'original qui avait été envoyé (il y a eu perte d'information). C'est le cas pour images, films, son. Si la perte d'info est "pétite", nous ne verrons (ou entendrons) pas la di□érence. Au contraire, les fichiers texte peuvent devenir inutiles si un seul bit est perdu ou modifié, il faut une compression sans perte (sauf exception, p. ex. $\smile$ et $\hookleftarrow$ dans un code peuvent tous être négligés).**

Compression en cascade **: Possible seulement avec compression sans perte sinon on ne pourra peut-être pas revenir en arrière.**

Compression perceptive **: Cela veut dire que le taux de compression choisi (même localement, par endroits ou par moments) dépene de ce que nous sommes prêts à perdre comme information sans que notre perception en soit a□ectée.**

Compression symétrique : **Encodeur et décodeur utilisent le même algorithme en directions opposées. Ils font donc "le même travail". Dans le cas contraires, l'un des deux doit travailler davantage. Par exemple, dans un cas où l'encodeur va stocker un fichier compressé dans une archive (opération e□ectuée une fois) et le fichier sera ensuite téléchargé pour être décompressé par plusieurs utilisateurs (décodeurs), on préférera une méthode asymétrique où la décompression est moins complexe que la compression.**

Performances de compression : **Plusieurs grandeurs et quantités :**

- le rapport (ratio) de compression : = $\frac{\text{taille stream de output (compressé)}}{\text{taille stream de input (original)}}$
  Aussi dit bpb **(bit per bit) parce qu'il exprime le nombre de bit nécéssaires en moyenne pour représenter dans le stream codé un bit du stream d'origine.**
  Pour les images on parle de bpp **(bit per pixel).**
  Pour le texte on parle de bpc **(bit par caractère).**
  $Bitrate$ **est un terme général pour bpb ou bpc.**

- le facteur (factor) de compression : = $\frac{\text{taille stream de input (original)}}{\text{taille stream de output (compressé)}}$ **(l'inverse du précédent)**

- le gain de compression : = $100 \log \frac{\text{taille stream de référence (original)}}{\text{taille stream compressé}}$
  où le strem de référence et soit le stream d'origine soit un stream obtenu par une méthode sans perte.

Modèles probabilistes : **Le modèle doit être construit avant de pouvoir comprimer. Par exemple on lit une fois, on compte les occurrences de chaque caractère, on calcule probabilités ensuite on encode à la 2e passe.**

Chapitre 3

Techniques de base (méthodes intuitives ou du passé

3.1 Brailles

Groupes de 3×2

triplets on $40^3 = 64000$ triplets. Puisque $2^{16} = 65536 > 40^3$, chaque triplet peut être codé avec 16 bits (2 octets). Sans compression, chacun des 40 caractères a besoin d'1 octet (en ASCII) donc chaque triplet fait 2 octets au lieu de 3. Rapport de compression $2/3 = 0,666$ (l'un des rares cas où il est constant et connu).

- **Code Baudot (1880, télégraphe) 5 bits (32 combinations) mais on représente plus de 32 symboles. Chaque quintuplet de bits code soit une lettre, soit une figure. Deux caractères spéciaux (meta-caractères ou caractères d'échappement) permettant de passer d'une représentation à l'autre. Au total $32 \times 2 - 2 = 62$ caractères. Pas très sûr, car il n'y a pas de bit de parité ou contrôle (parler de la question de la correction d'erreur de transmission).**

- **Codage de Dictionnaires (ou données stockées en ordre alphabétique). Deux mots adjacents partagent souvent un préfixe commun (disons de longueur n), alors un mot peut être représenté par le nombre n suivi du su□xe di□érent.**

Mot du dictionnaire	Mot codé
a	a
aardvark	1ardvark
aback	1back
abaft	3ft
abandon	3ndon
abandoning	7ing

3.4 Run Length Encoding (RLE)

Un run est une suite de valeurs toutes égales entre elles.
Principe : Si une donnée d apparaît n fois consécutives, alors remplacer $\underbrace{d \ldots d}_{n \text{ fois}}$ par $n \cdot d$.
Appliqué à texte et image.

3 1 Compression de texte par RLE

La chaîne :
`"attendre 2 secondes avant de passer"`
devient
`"a2tendre 2 secondes avant de pa2ser"` **(ambigu, pas de gain)**
donc avec un meta-symbole @ :
`"a@2tendre 2 secondes avant de pa@2ser"` **(pas ambigu, mais perte)**
Donc il convient de remplacer seulement les occurrences de 3 répétitions, ou plus.

Exercice : Ecrire l'algorithme d'un compresseur et d'un décompresseur RLE.

Problèmes :

1. **Dans les langages natures il existes beaucoup de doubles mais très très peu de triplets.**
2. **Si @ est un caractère du texte, un autre meta-caractère doit être choisi**
3. **Si le compteur de répétitions est représenté par un octet, il pourra compter au plus 255 répétitions.**

On peut calculer le facteur de compression d'un RLE pour un stream de N caractères ayant M répétitions de longueur moyenne L. Chaque répétition (run) est remplacée par un triplet (escape, compteur, donnée) :

$$\frac{N}{N-(M\times L)+(3\times M)}=\frac{N}{N-M(L-3)}$$

pour $N=1000, M=10, L=3 \Longrightarrow$ **facteur de compression** $=1.01$
pour $N=1000, M=50, L=10 \Longrightarrow$ **facteur de compression** $=1.538$

3 Variante 1

Pour alphabets de petite taille qui n'utilisent pas tous les symboles disponibles (P. ex. lettres, chi□res, caractères de ponctuation, par rapport à tous les ASCII) : codage par digramme (couples de lettres). On identifie les digrammes les plus communs et on les remplace par un caractère (les codes non utilisés) de ceux qui n'apparaissent pas dans le texte (p. ex. en anglais "E␣", "␣T", "TH", "␣A".)

3 3 Variante

Substitution de motifs. Bon pour programmes (codes source) où certains mots (P. ex les mots clé du langage de programmation) sont recourants $\Longrightarrow$ ces mots peuvent être codés par un seul caractère, ou bien par @ et un caractère. Par exemple si "a" correspond à "print" alors "print(a)" devient "@a(a)".

3 Relative Encoding

Suites de données (P. ex. numériques, comme une longue suite de températures relevées à intervalles réguliers, ou une suite de strings qui ne di□èrent pas beaucoup l'une de l'autre p. ex. les lignes consécutives d'un fax) qui ne di□èrent pas beaucoup $\Longrightarrow$ on a intérêt à coder les différences **avec la donnée précédente. Si les di□érences deviennent trop importantes, alors on peut envoyer la donnée brute $\Longrightarrow$ il faut une flag (1 bit de plus) pour distinguer donnés brutes et di□érences. En général, quand on doit envoyer des flags, on attends d'en avoir 8 avant de les envoyer pour qu'ils forment un octet (le décodeur devra attendre après 8 données (un octet chacune, par exemple) de recevoir l'octet de flags pour décoder les 8 données précédentes (codes à délai borné).**

Une autre pratique : envoyer des couples d'octets (16 bits) chaque couple est soit une valeur absolue (dans $[0, 2^{16}-1]=[0, 65535]$ ou dans $[-2^{15}, 2^{15}-1]=[-32768, 32767]$), ou bien deux di□érences sur huit bits chacune (donc dans $[-2^8, 2^8-1]=[-256, 255]$). Pour chaque couple, un flag est émis : 0 pour les valeurs absolues et 1 pour les di□érences. Après 16 couples d'octets, on envoie leurs 16 flags (un autre couple d'octets).

Exemple 4 *Par exemple :*

valeurs	51110	51115 51121	51119	8200	8202 8205	8207	8206	...
codes	51110	$(5,6)$	$(-2,-1)$	8200	$(5,6)$	$(2,-1)$	$(-1,-1)$	...
flags	0	1	1	0	1	1	1	...

3.5 RLE pour images

Une image digitale est une ensemble de petits points dits pixels. Un pixel est représenté soit par un bit (pour images en N/B), soit par plusieurs bits (pour images à niveaux de gris et en couleur). Les pixels sont stockés dans un tableau en mémoire (bitmap), d'habitude ligne par ligne.

Un pixels choisi au hasard a de grandes chances d'avoir la même valeur (ou une valeur très proche) que ses voisins. On peut donc balayer l'image ligne par ligne à la recherche de runs.

Si une ligne commence par 17B, 1N, 55B, 31N, ... et si on prend la convention qu'une ligne commence toujours avec un run de blancs (éventuellement de taille 0), seuls les 17, 1, 55, 31 doivent être écrit dans le compressé. La résolution de la bitmap (c.a.d. son nombre de colonne et éventuellement de lignes) doit aussi être inséré au début de l'output. En connaissant le nombre total de colonnes par ligne on ne doit pas envoyer le dernier run (on l'obtient par di□érence).

La taille de l'image compressée dépend de sa complexité. Mais dans le cas d'une image horizontalement connexe (1 seul morceau) le taux de compression est :

$$\frac{2 \times \textbf{demi-périmètre}}{\textbf{nombre total de pixels de la figure}} = \frac{\textbf{périmètre}}{\textbf{surface (en pixels)}}.$$

RLE marche aussi pour des images à plusieurs niveaux de gris. Chaque run de pixels est codé toujours par le couple (longueur du run, valeur commune des pixels du run). La valeurs est stockée sur plusieurs bits (4 à 8, selon le nombre de niveaux de gris)

Exemple 5 .

$$12, 12, 12, 12, 12, 12, 12, 12, 12, 35, 76, 112, 67, 87, 87, 87, 5, 5, 5, 5, 5, 5, 1$$

devient

$$\boxed{9}, 12, 35, 76, 112, 67, \boxed{3}, 87, \boxed{6}, 5, 1$$

$\boxed{\ }$ = *compteurs.*

Solutions pour distinguer compteurs et valeurs :

- **Si on n'utilise que 128 niveaux de gris (ou moins), le bit restant de l'octet peut être utilisé pour signaler s'il s'agit d'une valeur ou d'un compteur.**
- **Si on utilise 256 niveaux de gris on peut renoncer à un (P. ex le niveau 255) qui sera utilisé commé flag signalant que le prochain octet est un compteur.**

$$255, 9, 12, 35, 76, 112, 67, 255, 3, 87, 255, 6, 5, 1$$

- **On associe à chaque nombre (valeur ou compteur) un flag (0 si valeur, 1 si compteur). Ces flags sont associés par groupes de huit pour former un octet supplémentaire qui sera envoyé avant ou après les 8 données auxquelles il fait référence. On ajoute donc $1/8$ à la taille, qui augmente de $12,5\%$.**

$$10000010_2 = 130, \boxed{9}, 12, 35, 76, 112, 67, \boxed{3}, 87, 100....., \boxed{6}, 5, 1$$

- **si on a m pixels tous di□érents entre eux (aucune répétition de valeurs), on les fait précéder par $-m$. Note : pour pouvoir représenter des nombres négatifs on devra renoncer à un bit (de signe) pour les données. Si on a un octet à disposition pour les données, il faudra utiliser seulement 7 bits.**

$$9, 12, -4, \underbrace{35, 76, 112, 67}_{4 \text{ valeurs}}, 3, 87, 6, 5, ?, 1, \ldots$$

Si après le dernier 1 on a un autre 1, alors " ?" sera un compteur (longueur du run de 1), sinon il sera une valeur négative $-m$ où m est le nombre de valeur di□érentes de la précédente que l'on trouve à partir de 1. Cas le pire :

$$p_1, p_2, p_2, p_1, p_2, p_2, p_1, p_2, p_2, \ldots$$

devient

$$-1, p_1, 2, p_2, -1, p_1, 2, p_2, -1, p_1, 2, p_2, \ldots$$

Expansion de 33% !

3 1 Remarques

.

– **Puisque un run n'a pas longueur 0, on peut envoyer $l - 1$ au lieu de l**

– **Pour images couleur, un pixel est stocké sur 3 octets, le trois valeurs RGB (Rouge, Vert, Bleu) comprises entre 0 et 255. On code séparément chaque suite relative à une couleur .**

$$(171, 85, 34)(172, 85, 35)(172, 85, 30)(173, 85, 33)\ldots$$

se casse en trois :

$$(171, 172, 172, 173, \ldots); (85, 85, 85, 85, \ldots); (34, 35, 30, 33\ldots)$$

Comme pour niveaux de gris, RLE et relative encoding peuvent être utilisés.

– **Il est préférable de coder chaque ligne séparément, même si cela fait briser des runs. P. ex. si une ligne termine par 4 pixels de valeurs 87 et la suivante commence avec 9 pixels de valeur 87. IL convient de couper $(4, 87), (9, 87)$ plutôt que $(13, 87)$. Mieux : $(4, 87), eol, (9, 87)$. (code spécial pour eol).**

Raisons : Si chaque ligne est codée individuellement, le décodeur pourra décoder les lignes à intervalles réguliers (p. ex par pas de 5 : la 1^e, la 6^e, la 11^e, la $16^e, \ldots$ Ensuite la 2^e, la 7^e, la 12^e, la $17^e, \ldots$). Ainsi l'image est construite graduellement sur l'écran et on pourra plus vite comprendre sa nature et décider par exemple si terminer ou pas son téléchargement, si la garder ou pas...

Aussi, si chaque ligne est codée individuellement, il est plus facile extraire une partie de l'image (p. ex de la ligne l à la ligne h) ou de fusionner deux images sans devoir les décoder.

Dans ce cas, il faut un marquer de la fin de chaque ligne. De plus on fait précéder le fichier compressé avec un groupe de paquets de 4 octets (32 bits), un paquet de 4 octets pour chaque ligne de l'image. Le k-ème paquet de 4 octets contient l'o□set (mesuré en octets) du début du fichier au début de la ligne k. Taille plus importante mais on gagne du temps.

Désavantages de la méthode RLE pour images

- **Si l'image est modifiée, les longueurs des runs doivent être recalculées entièrement.**
- **RLE peut produir des fichiers plus gros (expansion). P. ex. une iages de lignes noires verticales sur fond blanc qui serait scannerisée horizontalement. Idéalement, un compresseur RLE devrait scanneriser une image horizontalement, verticalement et en diagonal et choisir le meilleur.**

EXO 1.2. Page 29.

Méthode de compression d'images avec perte

On améliore le taux de compression en ignorant les runs trop courts. La perte peut (ou pas) être acceptable (p.ex ; non adapté pour imagerie médicale) On demande à l'utilisateur la longueur maximale des runs à ignorer (p. ex ; 3). Le programme fusionne tous les runs de longueur 1,2, ou 3 avec leurs voisins.

Une autre application Images BMP

Si un pixel est représente par p bits on dit que l'images possède p bitplans. Le k-ème bitplan est la suite de tous les k-èmes bits.

Pour images à 1, 2, 4, 8, 16, 24 bitplans (ces dernières sont les images couleurs).

Format (simplifié) :

- **En tête du fichier : 2 octets `BM`, et la taille du fichier.**
- **En tête de l'image : largeur, hauteur, nombre de bitplans.**
- **Palette de couleurs (3 formats possibles, pas présentés ici).**
- **Les pixels (soit en format brut, ou bien compactés par RLE).**

Supposons 8 bitplans (un pixel = 8 bits = 1 octet = 1 un hexadécimal à 2 chi□res).
Le fichier compressé est organisé par couple d'octets (couples d'hexadécimaux à 2 chi□res). En général, dans chaque couple (C_{16}, P_{16}), le 1er octet est un compteur et le second une valeur de pixel P.

P. ex. : $04_{16}, 02_{16}$ **est le compressé de** $02_{16}, 02_{16}, 02_{16}, 02_{16}$**.**

Toutefois, si $C = 00_{16}$, alors la sémantique du couple $(00_{16}, P)$dépend du caractère P après le 00_{16} :

- **Si $P = 00_{16}$, le décodeur interprète le couple $(00_{16}, 00_{16})$ comme "$end\ of\ line$" (et remplit le restant de la ligne de 0).**
- **Si $P = 01_{16}$, le décodeur interprète le couple $(00_{16}, 01_{16})$ comme "$end\ of\ image$" (le reste de l'image est rempli de 0).**
- **Si $P = 02_{16}$, le décodeur interprète le couple $(00_{16}, 02_{16})$ comme l'information que ce qui suit est un run de 0's qui dépasse la fin de la ligne courante, et il faut donc e□ectuer un saut à une autre position de l'image plus en bas pour trouver le prochain pixel non nul. Dans ce cas, le couple d'octets qui suivent dit de combien de colonnes à droite et de combien de lignes vers le bas se trouve le prochain pixel non nul (par rapport au dernier pixel non nul).**
- **Si $P = k > 02_{16}$, le décodeur interprète le couple $(00_{16}, k_{16})$ comme le fait que les k octets qui suivent sont des données brutes (utilisé pour les suites de caractères tous di□érents).**
  P. ex dans le cas d'une image 4×8 avec 8 bitplans.

$$04_{16}, 02_{16}, 00_{16}, 04_{16}, a35b1247_{16}, 01_{16}, f5_{16}, 02_{16}, e7_{16}, 00_{16}, 02_{16}, 0001_{16},$$
$$01_{16}, 99_{16}, 03_{16}, c1_{16}, 00_{16}, 00_{16}, 00_{16}, 04_{16}, 08926bd7_{16}, 00_{16}, 01_{16}$$

est le code de l'image :

02	02	02	02	$a3$	$5b$	12	47
$f5$	$e7$	$e7$	00	00	00	00	00
00	00	99	$c1$	$c1$	$c1$	00	00
08	92	$6b$	$d7$	00	00	00	00

3.6 Codage Move-to-Front (MTF)

Premier exemple de code "adaptatif" (cad, qui modifie le codage basé sur des propriétés du texte à coder.

Idée de base : L'alphabet des symboles est maintenu comme une liste ordonnée. Le 1er symbole de la liste est codé par 0, le 2e par 1, le 3e par 2, etc. L'ordre de cette liste est modifié en cours de codage. À chaque fois qu'on traite (cad, qu'on code pour l'encodeur, qu'on $\mathrm{décode}$ pour le décodeur) un symbole, on le déplace en début de liste (son code devient donc 0) alors que tous les autres symboles sont décalés.

P.ex. Alphabet initial (liste) : $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}\}$**, code initiaux :**

symbole	a	b	c	d	m	n	o	p
code	0	1	2	3	4	5	6	7

Soit le stream `abcddcbamnoppnnm`**. Sans move to front (liste fixe et donc codage fixe)**

symbole à coder	**sans MTF**		**avec MTF**	
	liste (fixe)	**code**	**liste (variable)**	**code**
a	a, b, c, d, m, n, o, p	0	a, b, c, d, m, n, o, p	0
b	a, b, c, d, m, n, o, p	1	a, b, c, d, m, n, o, p	1
c	a, b, c, d, m, n, o, p	2	b, a, c, d, m, n, o, p	2
d	a, b, c, d, m, n, o, p	3	c, b, a, d, m, n, o, p	3
d	a, b, c, d, m, n, o, p	3	d, c, b, a, m, n, o, p	0
c	a, b, c, d, m, n, o, p	2	d, c, b, a, m, n, o, p	1
b	a, b, c, d, m, n, o, p	1	c, d, b, a, m, n, o, p	2
a	a, b, c, d, m, n, o, p	0	b, d, c, a, m, n, o, p	3
m	a, b, c, d, m, n, o, p	4	a, b, c, d, m, n, o, p	4
n	a, b, c, d, m, n, o, p	5	m, a, b, c, d, n, o, p	5
o	a, b, c, d, m, n, o, p	6	n, m, a, b, c, d, o, p	6
p	a, b, c, d, m, n, o, p	7	o, n, m, a, b, c, d, p	7
p	a, b, c, d, m, n, o, p	7	p, o, n, m, a, b, c, d	0
o	a, b, c, d, m, n, o, p	6	p, o, n, m, a, b, c, d	1
n	a, b, c, d, m, n, o, p	5	o, p, n, m, a, b, c, d	2
m	a, b, c, d, m, n, o, p	4	n, o, p, m, a, b, c, d	3

Note : la moyenne des codes sans MTF est 3.5 alors que avec MTF est 2.5 (on a utilisé des "plus petits nombres").

Mais cela ne marche pas toujours. Essayer avec `abcdmnopabcdmnop`**. La moyenne** $augmente$ **avec MTF.**

MTF marche bien dans l'hypothèse que si un symbole s **a été rencontré, alors il sera rencontré fréquemment au moins pour un moment. Autrement dit, le stream présente des concentrations**

de symboles identiques dans des régions du stream (on appelle cela *propriété de concentration*). **Le stream du premier exemple satisfait cette propriété, le second non.**

En général, les performances de MTF sont légèrement pire que celles du codage de Hu□man dans le pire des cas et bien meilleures dans le meilleur cas.

3.4.1 De l'intérêt de manipuler des valeurs plus petites

Une fois transformées les données numériques en valeurs plus petites (comme le fait *relative encoding*) **les nombres à valeurs plus petites apparaîtront plus fréquemment dans le stream codé, on peut donc ensuite coder ces données plus petites avec des codes à longueur variable où aux nombres plus petits sont a□ectés des codes plus courts, tandis qu'aux nombres plus grands sont a□ecté des codes plus courts (après avoir éliminé la redondance contextuelle on s'attaque à la redondance alphabétique). Il y a plusieurs manières de réaliser cela :**

- **Assigner des codes d'Hu□man (ce sera vu plus tard) aux entiers dans** $[0, n]$ **de manière que les entiers plus petit aient des codes plus courts. P ex** $0-0$**,** $1-10$**,** $2-110$**,** $3-1110$**,** $4-11110$**,** $5-111110$**,** $6-1111110$**,** $7-1111111$
- **Coder un entier** $i \geq 1$ **par son code binaire précédé de** $\lfloor \log_2 i \rfloor$ **zéros.**

i	**Code**	**Taille**
1	1	1
2	010	3
3	011	3
4	00100	5
5	00101	5
6	00110	5
7	00111	5
8	0001000	7
9	0001001	7
⋮	⋮	⋮
15	0001111	7
16	000010000	9

3.4.2 Variantes de MTF

Avance-de-k.

Le symbole lu n'est pas mis au début de lista mais avancé de k **positions (le paramètre** k **peut être choisi par l'utilisateur). En général cela réduit les performances (moyenne de codes plus élevée) pour des streams qui satisfont la propriété de concentration, mais les améliore pour les autres. Si** $k = n$ **(taille de l'alphabet), on retrouve MTF classic. Si** $k = 1$ **on échange simplement le symbole avec son prédécesseur dans la liste.**

Attend-c-et-bouge.

Un symbole et déplacé au début de la liste seulement après l'avoir rencontré c **fois (non nécessairement consécutives). Il faut un compteur par symbole. Utile si le réarrangement de** A **est couteux.**

MTF appliqué aux mots.

Normalement un symbole est un caractère (un octet). Mais si le stream contient du texte, on peut traiter chaque mot comme un symbole. On considère un mot toute chaîne de caractères comprise entre deux espaces ou entre un espace et le symbole de fin de stream (pour simplicité, on suppose que le stream contient que des lettres minuscules, l'espace et le caractère de fin de stream). Puisque on peut rencontrer n'importe quel mot, l'alphabet A n'est pas fixé au début. En fait, au début l'alphabet est vide, et on y ajoute des mots au fur et à mesure qu'il sont rencontrés et traités.

Exemple 6 *Soit le stream* the␣boy␣on␣my␣right␣is␣the␣right␣boy.

Premier mot : the. *Pas encore dans A donc on l'ajoute à A. On émet le mot* the *(brut) précédé du nombre de symboles qui le précèdent dans l'alphabet à présent, qui est* 0. *Le décodeur commence aussi avec un alphabet vide. Quand il recoit* 0 *il sait qu'il doit prendre le premier mot de A, mais puisque A est vide, il sait que* 0 *sera suivi d'un mot brut qu'il faudra ajouter à l'alphabet.*

Deuxième mot : boy. *Pas encore dans A donc on l'ajoute à A qui devient* $\{\texttt{the}, \texttt{boy}\}$. *On émet le mot* boy *(brut) précédé du nombre de symboles qui le précèdent dans l'alphabet à présent, qui est* 1. *PUIS, on avance le mot qu'on vient de coder en tête de liste :* $A = \{\texttt{boy}, \texttt{the}\}$. *Le décodeur reçoit un* 1, *il devrait donc prendre le 2e mot de l'alphabet, mais celui-ci n'a qu'un symbole (*the*) en ce moment, donc il sait qu'il va recevoir un mot brut qu'il devra ajouter (en tête !) à son alphabet.*

On continue selon la table (pour l'encodeur) :

Mot	*Alphabet (avant MAJ)*	*Code*	*Alphabet (après MAJ)*
the	{}	0the	{the}
boy	{the}	1boy	{boy, the}
on	{boy, the}	2on	{on, boy, the}
my	{on, boy, the}	3my	{my, on, boy, the}
right	{my, on, boy, the}	4right	{right, my, on, boy, the}
is	{right, my, on, boy, the}	5is	{is, right, my, on, boy, the}
the	{is, right, my, on, boy, the}	5	{the, is, right, my, on, boy}
right	{the, is, right, my, on, boy}	2	{right, the, is, my, on, boy}
boy	{right, the, is, my, on, boy}	5	{boy, right, the, is, my, on}

3.7 Quantization Numérique (Numerical Quantisation)

Chapitre 4

Méthodes Statistiques (Codes à taille variables)

4.1 Introduction

Principe : codes plus courts pour symboles plus fréquents.

Deux problèmes à résoudre :

1. **Choisir des mots qui forment un vrai code, cad qui peuvent être décodés sans ambiguité (P. ex.** $\{0, 10, 01\}$ **n'est pas un code,** 010 **a deux décodages possibles).**
2. **Choisir des codes avec taille (longueur des mots de code) moyenne la plus petite possible.**

4.2 Quelques Concepts de Théorie de l'Information

Qu'est ce que l'information ? Comment peut-on la mesurer ?

La Théorie de l'Information s'occupe précisément de répondre à la demande : "Combien d'information est contenue dans une donnée" ? Basée sur l'observation que l'info contenue dans une donnée est équivalente à la quantité de "surprise" contenue dans le message qu'elle convoie.

Exemple 1. **Le résultat de l'opération de lancer une pièce de monnaie peut être exprimée par P/F ou O/N ou 0/1. Le nombre minimal de questions (à réponse binaire) à poser pour deviner le résultat est 1. Autrement dit, 1 bit d'information su□t pour résoudre l'incertitude.**

Exemple 2. **Deviner une carte parmi 64 numérotées de 1 à 64. Le nombre minimal de question à poser (à réponse oui/non) pour deviner le résultat en tout cas est** $6 = \log_2 64$**. Autrement dit,** $6 = \log_2 64$ **bits d'information su□sent toujours pour résoudre l'incertitude.**

Exemple 3. **(Autre approche de la même question) Quelle information faut il donner pour identifier un nombre entier** $N \geq 0$ **? Réponse : les chi□res qui le composent qui sont en nombre de** $\lfloor \log_b N \rfloor$**, où** b **est la base choisie. L'info contenue dans le nombre** N **est proportionnelle au nombre de chi□res de** $N = \lfloor \log_b N \rfloor$**.**

Le logarithme est la fonction mathématique qui quantifie l'info.

1 Relation entre information et probabilité

On a déjà observé que la quantité d'information reliée à un événement est d'autant plu grande que la quantité de "surprise" apportée par l'événement est rande. Autrement dit, la quantité d'information I est d'autant plus grande que la probabilité P que l'événement se réalise est petite.

Les Exemples 1 et 2 précédents mettent en évidence cette relation entre information et probabilité dans le cas d'événements équiprobables (tels que le résultat du lancement d'une pièce ou de l'extraction d'une carte parmi 64).

Dans l'Exemple 1, le résultat du lancement d'une pièce porte une information de 1 bit alors que la probabilité qu'elle se réalise est $P = 1/2$. On a $1 = \log_2(2) = -\log_2(1/2)$.

Dans l'Exemple 2, le résultat de l'extraction d'une carte parmi 64 cartes numérotées, porte une information de 6 bit alors que la probabilité qu'elle se réalise est $P = 1/64 = 1/2^6$. On a $6 = \log_2(2^6) = -\log_2(1/2^6)$.

Dans le cas d'un événements parmi plusieurs équiprobables équiprobables on a en e□et

$$I = -\log_2(P)$$

Que se passe-t-il en cas d'une $suite$ d'évenements ayant en principe probabilités di□érentes ?

Exemple 4. **On a vu que pour gagner toujours à un jeu où il faut deviner correctement un nombre entre 1 et n il nous faut $I = \log_2(n)$ bits d'information.**

Soit un jeu qui consiste à deviner deux bons numéros en suite, d'abord l'un compris entre 1 et n et ensuite un autre, compris entre 1 et m. Ce jeu est équivalent à celui où il faudrait deviner correctement un $couple$ de bons numéros, l'un compris entre 1 et n et l'autre compris entre 1 et m. Dans ce jeu, il y a au total nm possibilités équiprobables ($P = 1/nm$). Pour deviner correctement la combinaison gagnante il faut, comme attendu

$$I = -\log_2(P) = \log_2(nm)$$

bits d'information (la même quantité d'information que pour deviner un nombre compris entre 1 et nm, puisque chaque couple peut être numérotée avec les nombres de 1 à nm et donc on revient à l'exemple 3.). Or, pour les propriétés du $\log$:

$$\log_2(nm) = \log_2(n) + \log_2(m),$$

donc, la quantité d'information nécessaire pour deviner d'abord un bon numéro compris entre 1 et n et l'autre compris entre 1 et m (la quantité d'information contenue dans une suite d'événements) est bien la somme des quantités d'information des événements pris individuellement. C'est à la fonction logarithme qu'on doit cette propriété d'additivité.

Notion d'Entropie

Considérons une source (émetteur). Dans la pratique, un vrai émetteur envoie données binaires (0 ou 1). Mais pour la généralité, supposons qu'il envoie streams composées de n symboles $\{a_1, a_2, \ldots, a_n\}$. Chaque symbole peut être vu comme un chi□re d'un système à base n, c'est à dire un entier plus petit ou égal à n, qui peut être identifiée par $\log_2 n$ bits.

Supposons que l'émetteur envoie s symboles par unité de temps (P. ex. 28800 bits par seconde, 1 bit par seconde = 1 baud). L'information envoyée est donc $H = s \log_2 n$.

$H = s \log_2 n$ est la quantité d'info (mesurée en bits) envoyée dans une unité de temps.

Soient données les probabilités P_i (pour $i = 1, 2..., n$) d'apparition des symboles a_i.

Dans le cas particulier où tous les P_i sont égaux ($P_i = P$ pour tout i) on a $1 = \sum_{i=1}^{n} P_i = nP$ et donc $n = 1/P$. On a donc

$$H = s \log_2(1/P) = -s \log_2 P$$

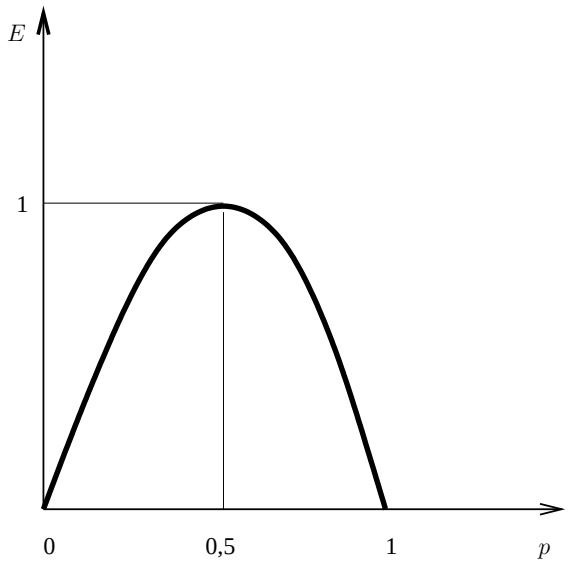
.

Par contre, dans le cas général, si une fraction P_i des s symboles émis dans une unité de temps est a_i, alors la contribution à l'entropie par ces sP_i symboles est $-sP_i \log_2 P_i$. On a

$$H = -s \sum_{i=1}^{n} P_i \log_2 P_i$$

info (mesurée en bits) envoyée dans 1 unité temps.

On appelle



Le stream émis d'une source d'entropie maximale (quand les symboles sont tous équiprobables) est complètement random et donc n'a aucune redondance. Sinon, la redondance R est définie comme la di□érence entre l'entropie maximale de la source et la vraie entropie :

$$R = \left(-\sum_{i=1}^{n} \frac{1}{n} \log \frac{1}{n} \right) - \left(\sum_{i=1}^{n} P_i \log P_i \right) = \log n - \left(\sum_{i=1}^{n} P_i \log P_i \right)$$

3 Codes taille variable

Soient quatre symboles a_1, a_2, a_3, a_4 qui apparaissent avec la même fréquence 0.25.
Entropie est : $-4 \cdot (0.25 \cdot \log 0.25) = -4 \cdot (0.25 \cdot (-2)) = 2$. Le plus petit nombre de bits nécessaires (en moyenne) pour représenter les symboles est 2. (P. ex. par le code à taille fixe : $00, 01, 10, 11$). La redondance du stream est 0, on ne peut pas compresser davantage.

Soit maintenant $p_1 = 0.49, p_2 = p_3 = 0.25, p_4 = 0.01$. Entropie 1.57. Le plus petit nombre de bits nécessaires (en moyenne) pour représenter les symboles est 1.57.

Avec le code précédent $00, 01, 10, 11$ on a une redondance de 0.43, donc un code de longueur variable peut faire peut-être mieux.

Par exemple : $a_1 = 1, a_2 = 01, a_3 = 010, a_4 = 001$.
Longueur moyenne : $1 \times 0.49 + 2 \times 0.25 + 3 \times 0.25 + 3 \times 0.01 = 1.77$ (ceci est vrai pour de longs messages, alors que pour petits messages on peut s'éloigner beaucoup de la moyenne).

Mais ce n'est pas un code ! : $01001 = 01|001 = 010|01 = a_3 a_2 = a_2 a_4$

Mieux, le suivant : $a_1 = 1, a_2 = 01, a_3 = 010, a_4 = 001$.
Cet ensemble est bien un code car il a la *propriété préfixe* **(aucun mot n'est préfixe d'un autre). Un ensemble avec la propriété préfixe est toujours un code.**
Un *code préfixe* **est un code à longueur variable avec la propriété préfixe.**

Il nous faut donc des algorithmes pour trouver des codes de taille moyenne minimale en fonction des fréquences (probabilités) des symboles de l'alphabet.

4.3 Méthode de Shannon-Fano

On applique l'algorithme suivant :
- **Ordonner les symboles par fréquences décroissantes ;**
- **Partager les symboles en deux sous-ensembles de sorte que les deux sommes des fréquences des symboles dans les deux sous-ensembles soient le plus proches possible ;**
- **Ajouter un 0 aux mots de code des symboles du premier ensemble et 1 aux mots de code des symboles du second ensemble ;**
- **Continuer récursivement à partages les sous-ensembles jusqu'à ce qu'ils contiennent un seul symbole.**

Voici un exemple (le symbole _ indique où l'on fait les partages).

Symbole	Construction								Code
0.25		1	_	1					11
0.20	_	1		0					10
0.15		0		1	_	1			011
0.15		0	_	1		0			010
0.10		0		0	_	1			001
0.10		0		0		0	_	1	0001
0.05		0		0		0		0	0000

Note. **La longueur moyenne de ce code (pondérée par les probabilités) est** 2.7 **alors que l'entropie est** 2.67**, on est donc très proche de la longueur minimale.**

Exercice. **Exécuter le premier partage entre le troisième et le quatrième symbole et montrer que le code obtenu est moins bon.**

Les partages "parfaits" (ce qui est possible si les probabilités sont des puissances négatives de 2) donnent les meilleurs codes. Par exemple :

Symbole	Construction						Code
0.25		1	_	1			11
0.25	_	1		0			10
0.125		0		1	_	1	011
0.125		0	_	1		0	010
0.125		0		0	_	1	001
0.125		0		0		0	000

Longueur moyenne 2.5**, entropie** 2.5**. Code optimal.**

4.4 Méthode de Huffman

C'est la base de beaucoup de logiciels de compressions sur les PC, soit utilisé comme seule méthode de compression, soit comme une étape d'un processus de compression en cascade.

Généralement, donne des meilleurs résultats que la méthode de Shannon-Fano et comme celui-ci donne les meilleurs résultats quand les probabilités sont des puissances (négatives) de 2. Cependant, on peut démontrer que la méthode de Hu□man produit toujours le code optimal (ce qui n'est pas vrai pour Shannon-Fano)

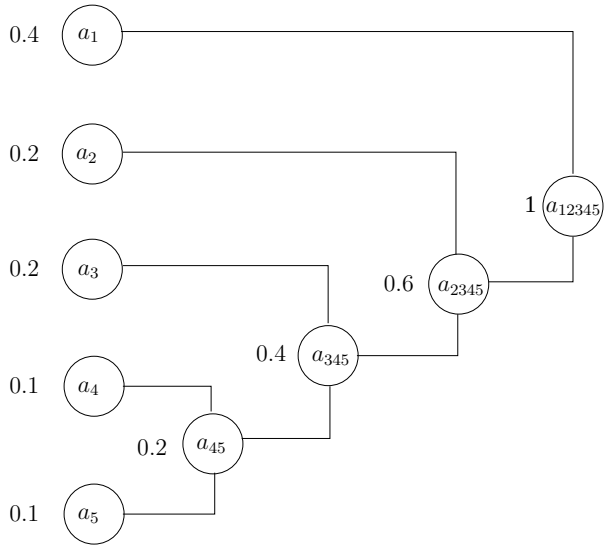
Di□érence avec Shannon-Fano, les mots de code sont produits de droite à gauche.

Voici l'algo (qui construit un arbre)

- **Ordonner les symboles de** $A = \{a_1, \ldots, a_n\}$ **par fréquences (probabilités) décroissantes ;**
- $E \longleftarrow A$ **;**
- **Tant que il y a plus d'un symbole dans** E **faire**
  - **Choisir deux symboles** $a_I, a_J \in E$ **(où** I **et** J **sont deux sous-ensembles de** $\{1, \ldots, n\}$**) ayant les deux plus petites probabilités ;**
  - **Construire l'arbre de racine** $a_{I \cup J}$ **ayant comme fils** a_I **et** a_J **;**
  - $E \longleftarrow E - \{a_I, a_J\} \cup \{a_{I \cup J}\}$ **;**
  - $Proba(a_{I \cup J}) \longleftarrow Proba(a_I) + Proba(a_J)$ **;**
- **Fin Tant que**

On obtient un arbre binaire dont les feuilles sont étiquetées par les symboles de l'alphabet. Le mot de code de chaque symbole est obtenu en parcourant le chemin de la racine à la feuille correspondante au symbole et en ajoutant un 0 à chaque fois qu'on se déplace vers un fils gauche et un 1 à chaque fois qu'on se déplace vers un fils droit.

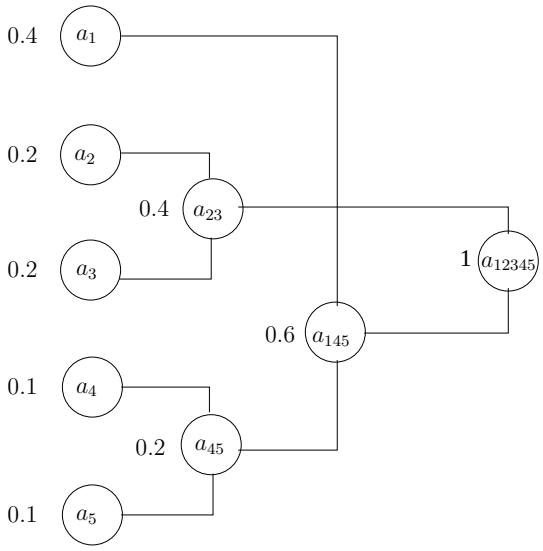
Exemple



On obtient

Symbole	Code
a_1	0
a_2	10
a_3	110
a_4	1110
a_5	1111

Toutefois, l'algorithme pourrait aussi produire un autre arbre



Correspondant au code

Symbole	Code
a_1	10
a_2	00
a_3	01
a_4	110
a_5	111

Calculons les deux longueurs moyennes. Pour le premier code :

$$M_1 = 0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2 \textbf{ bits/symbole}$$

Pour le second :

$$M_2 = 0.4 \times 2 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 3 + 0.1 \times 3 = 2.2 \textbf{ bits/symbole}$$

soit la même valeur.

Quand on choisit arbitrairement les deux symboles de probabilité minimale qu'il faut combiner, on peut obtenir des arbres di□érents mais la longueur moyenne résultera la même.

Lequel des deux codes est préférable ? Celui avec la plus petite variance (ou déviation).

Définition 1 *Soit une suite de valeurs $x_1, \ldots, x_n$ de moyenne $M = \frac{1}{n}\sum_{i=1}^{n} x_i$, la variance de la suite est :*

$$V := \frac{1}{n}\sum_{i=1}^{n}(x_i - M)^2$$

Pour une suite pondérée par les probabilités $p_1, \ldots, p_n$, où la moyenne pondérée est définie comme $M = \sum_{i=1}^{n} p_i x_i$, la variance sest définit comme suit :

$$V := \sum_{i=1}^{n} p_i(x_i - M)^2$$

Calculons les deux variances. Pour le premier code :

$$V_1 := 0.4 \times (1 - 2.2)^2 + 0.2 \times (2 - 2.2)^2 + 0.2 \times (3 - 2.2)^2 + 2 \times 0.1 \times (4 - 2.2)^2 = 1.36$$

pour le deuxième :

$$V_2 := 0.4 \times (2 - 2.2)^2 + 0.2 \times (2 - 2.2)^2 + 0.2 \times (2 - 2.2)^2 + 2 \times 0.1 \times (3 - 2.2)^2 = 0.16$$

Règle pour obtenir celui avec la plus petite variance : s'il existent plus que deux symboles ayant probabilité minimale, choisir les deux tels que la di□érence de hauteur est minimale. Ceci garantit que l'arbre obtenu est plus équilibré et revient à choisir les deux symboles les plus anciens (en ordre de création) parmi ceux ayant probabilité minimale. Ceci peut être facilement géré en implémentant l'ensemble E à l'aide d'un tas-min, où les symboles sont rangés par rapport à leur fréquence.

L'intérêt d'avoir des codes de variance minimale est que l'on peut utiliser des tampons de taille inférieure et que les temps de décodage de chaque symbole sont plus proches (important pour applications on-line).

Note. **Si les symboles ont la même proba, alors la méthode de Hu□man ne compresse pas les données (le texte est random et donc ne peut pas être compressé). Il y a bien sûr des cas spéciaux où, tout en gardant la même probas, les symboles sont disposés avec un certaine régularité, P. ex. $a_1 \dots a_1 a_2 \dots a_2 a_3 \dots a_3 \dots$, qui peut être compressé par RLE mais pas par Hu□man.**

Note. **Hu□man ne peut pas être appliqué à alphabets de taille 2 (p.ex images en N/B). On pourrait combiner paquets de bits (4, 8, ...) et les considérer un nouvel alphabet (de taille 16, 256, ...), mais on pourrait perdre certaines régularités, notamment celles dues au fait qu'un pixel est probablement de la même couleur que ses voisins. P. ex.** $00011100|00000110$ **a un run de 0 de longueur sept, qui serait cassé si on regroupe par paquets de 8 bits.**

Décodage

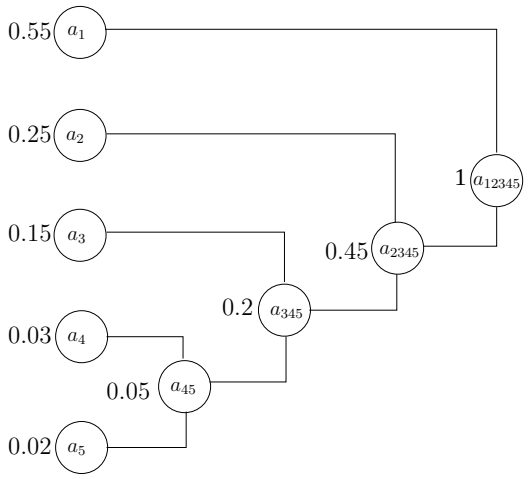
L'encodeur construit donc l'arbre sur la base des fréquences (entières) ou des probas (réelles) et ensuite il commence à coder en utilisant les mots de code ainsi obtenus.

Le décodeur doit connaître les mots de code ou avoir le même arbre qu'a construit l'encodeur pour pouvoir e□ectuer le décodage. Ou, au moins, il doit disposer de la même information qui a servi à l'encodeur pour le construire.

En tout cas, l'encodeur doit faire précéder le stream compressé par un "en tête" contenant :
- **soit des couples (symboles, codes) dans un format qui devra être déterminé par protocole ;**
- **soit l'arbre, dans un format déterminé par protocole ;**
- **soit les fréquences (ou les probas, qui peuvent être rendues entières par normalisation), par lesquelles on peut construire l'arbre (certaines règles devraient être alors décidées par protocole pour assurer que encodeur et décodeur construisent le même arbre, p. ex. lequel de deux fils prend 0 et lequel le 1).**

Si le décodeur dispose de l'arbre, l'algorithme de décodage dévient extrêmement simple. En commençant à la racine de l'arbre, pour chaque bit lu du stream codé, il parcourra les branches de l'arbre pour décoder un symbole à chaque fois qu'une feuille est atteinte, avant de revenir à a racine. Il s'agit en fait d'un simple transducteur à pétales.

Propriété pour calculer la longueur moyenne d'un code d'Huffman



On calcule la longueur moyenne :

$$M = 0.55 \times 1 + 0.25 \times 2 + 0.15 \times 3 + 0.03 \times 4 + 0.02 \times 4 = 1.7$$

Mais aussi

$$1.7 = 1 + 0.45 + 0.2 + 0.05$$

Ceci est toujours vrai. La longueur moyenne d'un code peut aussi être calculée en additionnant les probabilités de tous les nœuds *internes* **de l'arbre de Hu□man.**

1 Codes d'Hu man canoniques

Soit l'exemple de la figure 4.1.

L'algo construit le code :

a	000
b	001
c	100
d	101
e	01
f	11

Toutefois, il existe d'autres codes ayant la même longueur moyenne (et donc aussi codes d'Hu□man). Notamment, tous les codes préfixes qui assignent des mots de code de longueur 3 **à** a, b, c, d **et des mots de code de longueur** 2 **à** e **et** f **sont aussi des codes d'Hu□man (et donc optimaux).**

Pour le cas de cet exemple, nous voulons calculer le nombre de tels codes. On note que les mots de code peuvent être partagés en quatre classes, selon le préfixe de longueur deux par lequel ils commencent. Le quatre classes sont :

$$\{00x \mid x \in \{0,1\}\}, \quad \{10y \mid y \in \{0,1\}\}, \quad \{01\}, \quad \{11\}$$

Or, tous les codes qu'on obtient en permutant arbitrairement ces quatre préfixes seront des codes de Hu□man aussi. Il y a $4! = 24$ **de telles permutations. De plus, pour chaque permutation**

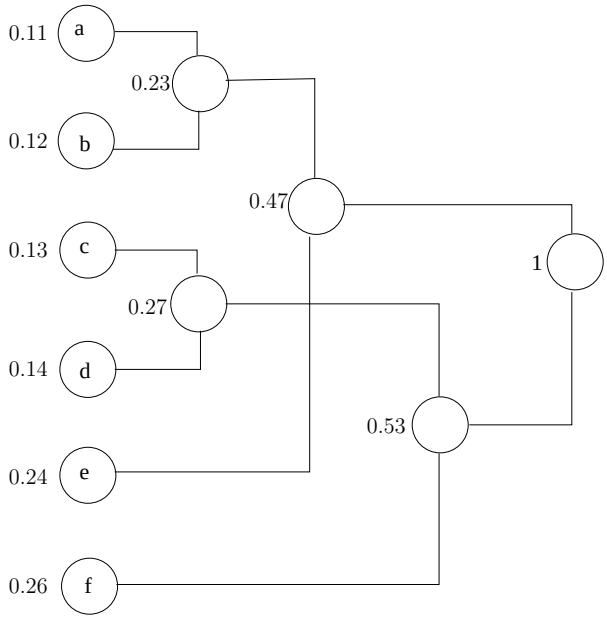


FIG. **4.1 – Un exemple pour la construction d'un code d'Hu□man canonique**

possible, on peut construire quatre codes de Hu□man di□érent en choisissant les valeur de x et y. Il y a donc au total $4 \times 24 = 96$ de tels codes d'Hu□man.

Considérons en particulier le code que l'on obtient par la permutation

$$00 \longrightarrow 00; \quad 10 \longrightarrow 01; \quad 01 \longrightarrow 10; \quad 11 \longrightarrow 10;$$

et en choisissant $x = 0$ pour a et $y = 0$ pour c.

a	000
b	001
c	010
d	011
e	10
f	11

Ce code est intéressant puisque il assigne aux symboles a, b, c, d quatre mots (sur trois bits) représentant en binaire les entiers consécutifs $0, 1, 2, 3$ alors qu'il assigne aux symboles e, f deux mots (sur deux bits) représentant les entiers consécutifs $2, 3$. Ce code est évidemment aussi un code de Hu□man, et pourtant l'algorithme de Hu□man donné ne pourra jamais le produire. En e□et, on constate que si un code a été produit par l'algo, alors e et f seront toujours combinés en dernier (ils se trouveront l'un dans le sous arbre droit de la racine et l'autre dans le sous arbre gauche) et donc leurs mots de code ne pourront jamais commencer par le même caractère (ici, 1).

On va montrer pourtant qu'un tel code de Hu□man choisissant pour toute longueur l des mots de code (sur $l$bits) représentant des entiers consécutifs est toujours possible et facile à construire. Un tel code est dit *code de Huffman canonique*. Comme on va le voir, l'intérêt d'utiliser des

codes canoniques est qu'ils peuvent être construits très facilement par le décodeur, sans besoin que celui-ci connaisse ni les probabilités, ni l'arbre. Il su□t qu'il connaisse pour toute longueur l combien de symboles ont été codés par un mot de longueur l. Voyons cela avec un exemple.

Supposons que l'encodeur execute l'algorighme pour calculer un code d'Hu□man et supposons qu'il découvre ainsi que l'algo a□ecte
- à 6 symboles, des mots de longueurs 7
- à 3 symboles, des mots de longueurs 6
- à 4 symboles, des mots de longueurs 5
- à 4 symboles, des mots de longueurs 3

Il veut alors changer le codage (pour un autre code d'Hu□man et notamment le code d'Hu□man canonique) de sorte que les n_l symboles codés par des mots de longueurs l soient codés par des mots (toujours sur l bits) représentant n_l entiers consécutifs. Autrement dit, pour un entier $premier(l)$ opportunément choisi, ces n_l symboles seront codés par les entiers

$$premier(l), premier(l)+1, premier(l)+2, \ldots, premier(l)+n_l-1.$$

Il s'agit donc simplement de déterminer pour tout l, l'entier $premier(l)$ de manière à obtenir un code préfixe.

On considère les longueurs en ordre décroissant (7, 6, 5, (4), 3). Pour la longueur maximale l_{max} (ici $l_{max}=7$), on choisit toujours $premier(l_{max})=0$. Le code canonique alors assigne aux six symboles le moins fréquents les mots sur 7 bits :

$$\begin{array}{c} 0000000 \\ 0000001 \\ 0000010 \\ 0000011 \\ 0000100 \\ 0000101 \end{array}$$

Il faut maintenant déterminer $premier(6)$, de manière à ce que les 3 mots de longueur 6 représentant $premier(6)$, $premier(6)+1$ et $premier(6)+2$ ne soient pas préfixes des mots de code déjà présents.

Pour cela, il su□t de choisir pour $premier(6)$ un entier m tel que $2\times m$ (c-a-d, m shifté d'un bit vers la gauche avec l'ajout d'un 0) soit plus grand que $0000101 = 101 = 5 = 0+6-1 = premier(7)+n_7-1$. On peut simplement choisir $premier(6) = \lceil \frac{premier(7)+n_7}{2} \rceil = 6/2 = 3$. Le code canonique alors assigne aux trois symboles suivants les mots sur 6 bits :

$$\begin{array}{c} 000011 \\ 000100 \\ 000101 \end{array}$$

Noter qu'aucun de ces mots n'est préfixe des mots de longueur 7 déjà dans le code.

De manière analogue on détermine

$$premier(5) = \left\lceil \frac{premier(6)+n_6}{2} \right\rceil = \left\lceil \frac{3+3}{2} \right\rceil = \lceil 6/2 \rceil = 3.$$

Le code canonique alors assigne aux quatre symboles suivants les mots sur 5 bits :

$$\begin{array}{c} 00011 \\ 00100 \\ 00101 \\ 00110 \end{array}$$

Noter qu'aucun de ces mots n'est préfixe des mots de longueur 6 **et** 7 **déjà dans le code.**

Même si aucun symbole n'est codé par des mots de longueur 4, on doit calculer $premier(4)$ **car sa valeur est nécessaire pour calculer** $premier(3)$**. On a**

$$premier(4) = \left\lceil \frac{premier(5) + n_5}{2} \right\rceil = \left\lceil \frac{3+4}{2} \right\rceil = \lceil 7/2 \rceil = 4$$

et

$$premier(3) = \left\lceil \frac{premier(4) + n_4}{2} \right\rceil = \left\lceil \frac{4+0}{2} \right\rceil = \lceil 4/2 \rceil = 2.$$

Le code canonique alors assigne aux quatre derniers symboles suivants les mots sur 3 bits :

$$\begin{array}{c} 010 \\ 011 \\ 100 \\ 101 \end{array}$$

L'encodeur va alors coder les symboles (triès par fréquence croissante) par les mots

0000000	00011
0000001	00100
0000010	00101
0000011	00110
0000100	010
0000101	011
000011	100
000100	101
000101	

Le décodeur utilisera le même code. Toutefois pour que le décodeur puisse le reconstruire il su□t qu'il connaissent uniquement :
- les symboles de l'alphabet triés par ordre de fréquences croissantes (dans un format accordé par protocôle, p. ex ; en ASCII)
- pour toute longueur l**, le nombre de symboles codés par des mots longueur** l**.**

Ces informations lui su□sent à reconstruire le code canonique par l'algorithme :

```
Premier[l_max]=0 ;
Pour i de l_max − 1 à 1 faire
       Premier[i] = ⌈(Premier[i+1]+Nombre[i+1])/2⌉
fPour
```

Pour l'exemple précédent on a comme donnée le tableau $\texttt{Nombre} = (0, 0, 4, 0, 5, 3, 6)$ **et l'algorithme calcule** $\texttt{Premier} = (2, 3, 2, 4, 3, 3, 0)$**.**

Le code canonique a aussi l'avantage de faciliter le décodage. Le décodeur lit le stream codé bit par bit. Pour chaque paquet de l **bits lus, il verifie si le mot** v **de** l **bit qui vient d'être lu représente un entier plus petit que** $Premier[l]$**. Si c'est le cas,** v **n'est pas un mot de code mais un préfixe d'un mot du code. Il faut donc continuer à lire bit par bit jusqu'à ce que** v **(dont la longueur** l **augmente) devienne plus grand ou égal à** $Premier[l]$**, on pourra alors le décoder.**
P. ex si le décodeur reçoit le stream $0001001\ldots$**, à chaque bit lu on aura :**

$$\begin{aligned}
&v = 0 < Premier[1] = 2\\
&v = 00 < Premier[2] = 3\\
&v = 000 < Premier[3] = 2\\
&v = 0001 = 1 < Premier[4] = 4\\
&v = 00010 = 2 < Premier[5] = 3\\
&v = 000100 = 4 \geq Premier[6] = 3
\end{aligned}$$

Il saura alors que 000100 **représente le 2e symbole parmi ceux qui sont codés sur des mots de longueur 6 et il pourra ainsi le décoder avant de poser à nouveau** $v = \varepsilon$ **et lire le prochain bit.**

Codes d'Huffman adaptatifs

Souvent, on ne connaît pas les fréquences (probabilités) de chaque symbole avant d'avoir entièrement lu le stream.

Pour éviter d'effectuer un codage à deux passes (peu efficace), la méthode développe un code d'Huffman basé sur les statistiques des symbole déjà lus.

Il faut faire cela en minimisant les calculs pour mettre à jour l'arbre lorsqu'un nouveau symbole est lu et donc les fréquences changent.

Si n **est la taille de l'alphabet, on représente l'arbre par un tableau de taille** $2n - 1$**, puisque un arbre binaire complet avec** n **feuilles (les symboles) possède exactement** $2n - 1$ **nœuds en tout. Chaque composante du tableau correspond donc à un nœud de l'arbre et est constituée d'une structure ayant des champs contenant la fréquence du symbole (le poids du noeud), le symbole lui-même (si le nœud est une feuille), ainsi que des champs permettant de faire les liens de parenté (par exemple l'index des fils gauche et droite). Chaque nœud a donc un poids (la fréquence du symbole s'il s'agit d'un feuille, la somme des fréquences de ses deux fils si le nœud est interne) et un numéro (son index dans le tableau).**

On a□ectera l'index $2n + 1$ **(soit, la dernière composante du tableau) à la racine de l'arbre. Ensuite, à chaque fois que un nouveau noeud est ajouté à l'arbre, on lui a□ectera le plus grand index pas encore attribué à un nœud. Autrement dit, on remplira le tableau** de la droite vers la gauche**.**

De plus, les nœuds sont ordonnés dans le tableau de sorte que leurs poids sont (faiblement) croissants et de sorte que le numéro d'un nœud interne est toujours supérieur à celui de ses deux fils. Ceci assure que quand on lit les poids des nœuds du bas vers le haut et , dans chaque niveau de l'arbre de la gauche vers la droite, on obtient une suite faiblement croissante.

On voudra maintenir cette propriété comme un invariant. Par la suite on appellera *bloc* **une sous-suite de nœuds (un intervalle du tableau) constituée de tous les nœuds ayant le même poids.**

La procédure de mise à jour de l'arbre

Au début, l'encodeur et le décodeur ne connaissent rien sur les statistiques du stream. Les deux commencent alors avec un arbre vide. En fait cet arbre contient un seul nœud racine : "PEC" (pour $Pas\ Encore\ Codé$**) (de numéro** $2n-1$ **et de poids 0) correspondant à tous les symboles de l'alphabet qui n'ont pas encore été codés. Pendant le traitement du stream, l'encodeur et le décodeur maintiennent des arbres identiques (synchrones), méthode symétrique.**

Quand un caractère c **est rencontré pour la première fois, l'encodeur l'envoie sous forme non compressée (brute). Bien entendu, il faut que le décodeur puisse reconnaitre les suites de bits qui correspondent à des caractères non compressés, il faut donc un meta-caractère pour annoncer l'arrivée d'une donnée brute. On comprend que le code de ce meta-caractère ne peut pas être fixe dans le temps, parce que, quel que soit le code qu'on choisit pour lui, il se peut que plus tard ce code doive être attribué à un autre symbole. On utilise le nœud PEC (et donc son code, qui est variable) avec précisément cette fonction.**

Ensuite, l'encodeur ET le décodeur ajoutent à l'arbre un nouveau nœud (de poids 1) correspondant à c**. Ceci est fait en transformant le nœud PEC (qui était une feuille) en un nœud interne (de poids 1) ayant deux fils : le nouveau nœud PEC (de poids toujours 0 et qui reste une feuille) et le nœud correspondant à** c **de poids 1. La position de PEC dans l'arbre donc change, ainsi que son code. Ce sera son code (courant) qui sera envoyé comme caractère d'échappement avant d'envoyer un caractère sous forme brute. Puisque le décodeur maintient un arbre synchrone, il reconnaitra le code du caractère d'échappement PEC et donc le flag.**

Quand un caractère c **est déjà dans l'arbre, l'encodeur envoie son code que le décodeur pourra décoder (les deux ont toujours le même arbre). Ensuite les deux procèdent à la (même) mise à jour de l'arbre. Pour cela, il faudra incrémenter d'une unité le poids du nœud de l'arbre correspondant à** c **et rétablir l'invariant s'il a été cassé. En e□et, l'invariant est cassé si le nœud contenant** c **n'est pas le dernier de son bloc. Pour rétablir l'invariant il su□t d'échanger le nœud** c **avec le dernier nœud de son bloc (il y a une exception, quand le dernier nœud de son bloc est précisément son père ; dans ce cas aucun échange n'est nécessaire). La procédure de mise à jour ne peut pas s'arrêter là, puisque il faudra incrémenter d'une unité aussi le poids du nœud père de** c **et répéter récursivement la procédure de mise à jour à son niveau pour rétablir l'invariant.**

Chapitre 5

Méthodes à Dictionnaire

Les méthodes statistiques utilisent un modèle statistique des données, les résultat de compression dépendent de la qualitè du modèle.

Les méthodes à dictionnaire sélectionnent chaînes de caractères à coder comme des "token" utilisant un dictionnaire.

Le dictionnaire peut être
– statique (fixe, permanent) bien que des ajouts soient parfois permis, ou
– dynamique (adaptatif), ajouts et suppressions du dictionnaire sont alors permis en fonction de ce qu'on trouve dans le texte.

En principe, une méthode à dictionnaire peut compresser une chaîne de n symboles en nE bits où E est l'entropie de la chaîne (si le fichier est su□samment large). Elles fournissent donc des codages optimaux du point de vue du taux de compression.

Elles sont appliquées à texte mais aussi à image et audio.

Si une chaîne est déjà dans le dictionnaire (on dit dans ce cas qu'on a un $matching$), un token correspondant à son "index" est envoyé. Sinon, on envoie la chaîne brute. Le stream compressé contient donc indices et mots bruts, il faut donc pouvoir les distinguer.

Une possibilité est de réserver un bit supplémentaire (flag) pour chaque item. Généralement 19 bit sont su□sants pour représenter indices dans le dictionnaire ($2^{19} = 524.288$ mots dans le dictionnaire). Avec un 20^e bit on indique si le mot a été trouvé (0) ou non trouvé (1) dans le dictionnaire . Si le mot est brut (1), il faut faire suivre ce 1 par la longueur du mot.

Donc, si un mot est dans le dictionnaire , son codage tient sur 20 bits. S'il ne l'est pas, un octet pour le flag et la longueur (sut 7 bits), plus un octet par lettre. Si les mots ont une longueur moyenne de 5 lettres, cela fait 6 octets, soit 48 bits.

Si on peut compresser 48 bits en 20 bits, c'est très bien, à condition que cela arrive souvent (qu'un mot est dans le dictionnaire). Si la probabilité de trouver un mot dans le dico est P, alors après lecture/codage de N mots on a un stream compressé de taille :

$$N[20P + 48(1 - P)] = N[48 - 28P]bits$$

alors que la taille du stream non compressé (dans l'hypothèse que les mots ont une longueur moyenne de 5 lettres) est $5N$ octets = $40N$ bits.

On a donc compression si $N[48 - 28P] < 40N$, ce qui donne $P > 0.29$.

Dès qu'on a des matchings avec une probabilité de 29%, on a compression.

Pour un texte en langue naturelle, un dictionnaire statique de 500000 fait très bien l'a□aire (puisque tous les mots y seront).

Pour fichiers binaires, quand ils sont lus en ASCII cela peut donner un nombre très grand de mots (sans sens). Dans ce cas, c'est plus di□cile de trouver un mot dans le dictionnaire , donc on risque d'avoir expansion plutôt que compression, un dictionnaire statique ne marche pas bien.

Cependant, un dictionnaire statique marche bien dans des cas particuliers. P. ex. quand on utilise souvent des mots techniques, qui font partie d'un petit dictionnaire ad hoc. Dans ce cas un petit dictionnaire statique donne de très bonnes performances.

En général, on a de meilleurs résultats avec des méthodes adaptatives. Le dictionnaire est initialement vide (ou avec quelques mots par défaut). Au fur et à mesure que l'on rencontre des nouveaux mots, on les ajoute. Si le dictionnaire devient trop grand, on peut e□acer certains mots (p. ex ceux que l'on ne rencontre plus depuis le plus longtemps). Cela permet de réduire le temps de recherche.

Comme pour toutes le méthodes adaptatives, quand on trouve un mot dans le dictionnaire, on envoie son token. Sinon, on envoie le mot non compressé et et ensuite on l'ajoute au dictionnaire. De plus, avant de passer au mot suivant, on regarde si un vieux mot peut être e□acé du dictionnaire .

Avantages des méthodes à dictionnaire :
- **le calculs e□ectuées par les algorithmes sont des algos de "string search" or "string matching" et non des opérations numériques (qui peuvent causer des problèmes d'arrondi).**
- **les décodeur est très simple (méthodes asymétriques, alors que pour les méthodes statistiques, le plus souvent c'est symétrique). Le décodeur qui reçoit un token (index), doit seulement chercher le mot correspondant dans le dictionnaire .**

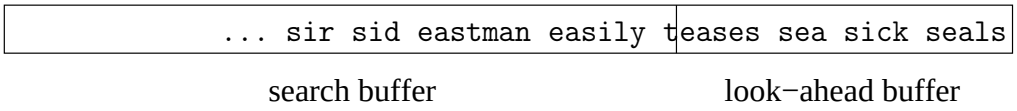
5.1 LZ77 (fenêtre glissante)

1 1 L'encodage

L'encodeur maintient une fenêtre (tampon) qui est virtuellement glissée sur le texte à coder en la déplaçant vers la droite. Le tampon est composé de deux parties :
- **la partie gauche est le tampon de recherche (search bu□er, ou s.b.), c'est le dictionnaire proprement dit, où se trouvent les symboles récemment codés. Généralement il a une taille de l'ordre de milliers d'octets.**
- **la partie droite est le tampon d'anticipation (look ahead bu□er ou l.a.b.) qui contient du texte encore à coder. Généralement il a une taille de l'ordre de quelques dizaines d'octets.**

Exemple 7 *Soit le texte :*

Supposons que `eases␣sea␣sick␣seals` **doit encore être compressé. Le premier symbole du l.a.b. (premier symbole à coder) est un** `e`**. On regarde le s.b. de gauche vers la droite à la recherche d'un** `e`**. On en trouve un à distance (o□set) 8 de la fin du tampon. Ensuite on regarde**



combien de symboles suivant ce e **coïncident avec des symboles suivant le** e **à coder. Il y en a deux : le** a **et le** s**, donc au total on un "matching" de longueur 3. On continue de balayer le s.b. vers la gauche à la recherche du matching le plus long. En cas d'égalité, on retient le plus à gauche. Il y a celui à o□set 16 ayant aussi longueur 3. C'est celui qui sera retenu. Le token de codage est alors le triplet formé par l'o□set o, la longueur l du matching et le premier caractère suivant les l caractères du matching. Dans ce cas le token envoyé sera** $(16, 3,$ e$)$**.**

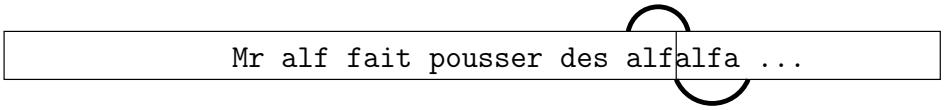
Le token est écrit dans l'output et la fenêtre est "glissée" vers la droite de 4 positions.

Si aucun matching n'est trouvé dans le s.b., alors le token $(0, 0, symbole)$ **est envoyé. Ces tokens, qui n'e□ectuent pas une bonne compression, sont communs au début :**

s ⟶ **(0, 0,** s**)**
i ⟶ **(0, 0,** i**)**
r ⟶ **(0, 0,** r**)**
␣ ⟶ **(0, 0, ␣)**
sid ⟶ **(4, 2,** d**)**

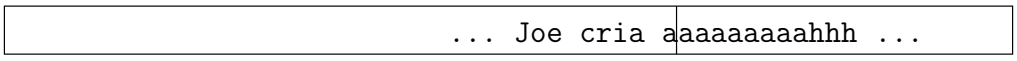
Quelle est la taille en bits d'un token ? Si S est la taille du s.b., L la taille du l.a.b. et A la taille de l'alphabet, un token a une taille $\lceil \log_2 S \rceil + \lceil \log_2 (L-1) \rceil + \lceil \log_2 A \rceil$. Si S est de l'ordre de quelques milliers de bits, $\lceil \log_2 S \rceil =$ 10 - 12 bits. Si L est de l'ordre de quelques dizaine de bits, $\lceil \log_2 (L-1) \rceil =$ 4 - 6 bits. Les symboles de l'alphabet sont typiquement codées sur 8 bits. Pour un total de 24 bits environ, y compris pour les tokens de type $(0, 0, symbole)$**, ce qui est trois fois plus long que le symbole brut.**

Attention au cas :



Le plus long matching est celui a o□set 3 qui est de longueur 4 (parce qu'il comprend le a, même si celui-ce est le permier caractère du l.a.b.) et non celui de o□set 13 et longueur 3, donc le token (3, 4, ␣) sera envoyé et non (13, 3, a**).**

Penser aussi au cas :



1 Le décodage

Le décodeur est plus simple (LZ77 est asymétrique, donc adapté si la compression s'e□ectue une seule fois et la décompression plusieurs fois). Il doit maintenir une tampon de taille égale à

la fenêtre de l'encodeur. Quand il reçoit un token, il cherche dans son tampon le string correspondant au matching et ajoute le symbole en troisième position du token. Ensuite il e□ectue le shift nécessaire (de longueur du matching plus 1).

13 Petites améliorations possibles

La méthode LZ77 attribue les codes (tokens) en fonction de ce qui a été lu récemment (ce qui se trouve toujours dans le s.b.) donc les textes qui seront mieux compressés sont ceux où des chaines similaires sont distribuées de manière proche les unes des autres.

Quelques améliorations des performances de compression peuvent être obtenues par les techniques suivantes

- Utiliser des champs de taille variable pour l'o□set et les longueurs
- Augmenter la taille du s.b. Un s.b. plus grand permettera de trouver des matching plus fréquemment et plus longs, mais le prix à payer est que les algos de recherche d'un mot dans le dictionnaire (recherche de motif) sera plus longue et élaborée.
- Changer la technique pour faire coulisser la fenêtre. En général un shift coûte $O($taille(fenêtre)$)$. Mais si on omplémente la fenêtre comme une file circulaire, alors à chaque nouvel input il su□ra d'écraser la partie la plus ancienne du s.b. et déplacer les deux pointeurs de début et de fin de file.
- Organiser le dictionnaire (le s.b.) sous forme de ABR, mieux si équilibré, dans lequel on stocke toutes les sous-chaînes du s.b. le longueur $\mid l.a.b. \mid$ avec l'o□set qui leur est associé. On envoie donc un token à 2 champs (o□set, longueur). Il faut mettre à jour l'arbre après chaque shift (éliminer des nœuds et en ajouter des nouveaux).

Défauts de LZ77

- Assume que les motifs égaux ou similaires sont proches. si un chaîne apparaît souvent mais plus espacée que $\mid l.a.b. \mid$ on a mauvaise compression.
- Taille limitée des bu□ers. Elle doit être limitée parce que les algorithmes de string matching doivent vérifier caractère par caractère. L.a.b. plus grand donne meilleur compression mais encodeur ralenti (doit chercher des concordances plus longues). de même pour le s.b. Augmenter la taille des tampons produit aussi des tokens plus gros et donc réduit la compression.

1 LZ 8

Pas de s.b., ni de l.a.b. ni de fenêtre. Mais un dictionnaire de chaînes déjà rencontrées. La taille du dictionnaire est limitée seulement par la mémoire disponible.

L'encodeur émet des tokens à deux champs : un pointeur à une entrée du dictionnaire et le code d'un symbole (la longueur n'est plus nécessaire).

Chaque token correspond à une chaîne et est ajouté au dictionnaire juste après que le token a été émis. On n'e□ace jamais rien du dictionnaire.

Le dictionnaire est initialisé avec la chaîne vide ε en position 0.

Quand on lit un symbole x :

- si on le trouve pas, on émet $(0, x)$ et on ajoute la chaine x à la première place disponible du dictionnaire.
- si on le trouve (par exemple en position 37), on lit le caractère suivant y

- si xy n'est pas dans le dictionnaire, on émet le token $(37, y)$
- si xy est dans le dictionnaire, on lit le caractère suivant z et on cherche dans le dictionnaire xyz
- ...

En général, on lit des caractères jusqu'à ce que la châine lue w n'est pas dans le dictionnaire, alors que w' (avec $w = w'y$ pour un symbole y) est dans le dictionnaire. Alors on émet le token $(index(w'), y)$.

La taille du dictionnaire peut être limité (on verra quoi faire s'il se remplit), puisque un dictionnaire plus grand signifie plus de chance d'y trouver la chaîne, mais pointeurs plus longs et recherche dans le dictionnaire plus lente.

Structure de données pour le dictionnaire : un arbre lexicographique. Pour ajouter la chaîne $w'y$ on ajoute un nœud $(nouvel.index, y)$ comme fils du nœud correspondant à w'.

Une recherche dans le dictionnaire parcourt l'arbre de la racine vers une feuille (si trouvé) ou un nœud interne (si non trouvé).

L'arbre doit être dynamique c-à-d implementé de manière à ajouter un nombre arbitraire de fil à chaque nœud.

Quand l'arbre est plein, plusieurs solutions :
- Le fixer et continuer à l'utiliser comme un dictionnaire statique pour coder le reste.
- L'e□acer entièrement et recommencer à nouveau. Conseillé si on change de "section" dans le texte, susceptibles d'avoir distributions de motifs très di□érentes.
- E□acer des nœuds (entrées du dictionnaire), idéalement celles moins utilisées récemment (problème : pas d'algo e□cace pour décider cela).

LZ78 est symétrique (le décodeur est plus complexe que pour LZ77)

1 LZ

Peut-être la variante la plus populaire de LZ78. On élimine aussi le deuxième champ du token. Un token est simplement un pointeur à une entrée du au dictionnaire.

Supposons pour l'instant pour simplicité que le dictionnaire soit un tableau et pas un arbre .

On initialise les premiers items du dictionnaire avec les symboles de l'alphabet (pour ASCII, les 256 premières position).
Donc un symbole est toujours présent dans le dictionnaire.

Pour le codage, on lit les symboles et on stocke la chaîne lue dans une chaîne I. Quand on rencontre un symbole x tel que I est dans le dictionnaire mais Ix ne l'est pas, on émet l'index de I, on ajoute Ix au dictionnaire et on change l'ancien I en x avant de continuer à lire.

Structure du dictionnaire de LZW et codage

La meilleurs structure est un arbre lexicographique, où pour tout mot w et tout caractère x, le mot wx se trouve dans un nœud fils du nœud qui contient w.

Cela permet de rechercher facilement un mot dans le dictionnaire (on parcourt une branche à partir de la racine) et lorsque w est dans le dictionnaire mais wx n'y est pas on sait où l'insérer

(comme fils de w étiqueté par x).

Il faut quand même utiliser une structure "dynamique" qui permet à un nœud d'avoir un nombre quelconque de fils sans leur réserver de place à l'avance.

On implémente cet arbre comme un tableau de nœuds.

Chaque nœud contient 2 champs : un pointeur vers le père (l'index du nœud père) et un symbole.

Pour descendre dans l'arbre on utilise une fonction de hachage h qui associe un nouvel index à une paire (index, symbole).

Exemple. **Supposons de lire** `abcd` **et qu'après lecture de** `abc` **on sache que** `abc` **est dans le dictionnaire avec** `a` **en position 97,** `ab` **en position 266 et** `abc` **en position 284. Les positions correspondantes du tableaux contiendront :**

•••	0	•••	97	•••	266	•••
•••	a	•••	b	•••	c	•••
	97		266		284	

Le prochain symbole est un `d`**. On doit donc chercher si** `abcd` **est dans le dictionnaire, plus précisément s'il y a un nœud avec symbole =** `d` **et père = 284.**

On calcule alors $h(284, 100)$ (puisque 100 est le code ASCII de `d`**) qui retourne un index (p. ex 299). Il y a alors 3 possibilités :**

1. **Le nœud 299 est inutilisé. Cela veut dire que** `abcd` **n'appartient pas encore au dictionnaire et doit être ajouté à cette place. On crée alors le nouveau nœud 299 et le dictionnaire devient :**

•••	0	•••	97	•••	266	•••	284	•••
•••	a	•••	b	•••	c	•••	d	•••
	97		266		284		299	

2. **Le nœud 299 a bien** `d` **dans le champ symbole et 284 (d'où on vient) dans le champ père. Cela veut dire que** `abcd` **appartient ou dico. On passe à lire le symbole qui suit le** `d`**.**
3. **Le nœud 299 contient autre chose. Cela veut dire qu'il y a une collision de la fonction de hachage. La manière plus simple de traiter les collisions est celle de consulter les nœuds suivants (300, 301,...) jusqu'à ce qu'on tombe sur un nœud inutilisé (on est alors dans le premier cas) ou bien sur un nœud qui contient le couple (288,** `d`**) (on est alors dans le deuxième cas)**

En fait on utilise aussi un troisième champ de type index qui servira au décodeur. Ce champs contient la valeur retourné par la fonction de hachage sur le couple formé par le deux autres

champs. Donc la valeur contenue dans ce champs coïncide avec la position du nœud seulement s'il n'y a pas eu de collision.

Décodage

Le décodeur n'a pas besoin de fonction de hachage. Il initialise le dictionnaire comme l'encodeur, puis il lit les tokens et les utilise pour identifier un nœud et donc une entrée du dictionnaire.

Chapitre 6

Compression d'images

Une image digitale est un tableau rectangulaire de points (dits $picture\ elements$ **ou** $pixels$**) disposés sur** m **lignes et** n **colonnes. L'expression** $m \times n$ **est dite** $résolution\ de\ l'image$**.**

6.1 Introduction à la compression d'images.

Puisque la plupart des application aujourd'hui présentent des interfaces graphiques, on doit tout le temps manipuler des images.

Mais les images peuvent être très volumineuses ! ! Généralement, pour images couleurs, chaque pixel est représenté par un nombre sur 24 bits (3 fois 8, un octet pour chaque compostante R, G, B), ce qui permet de représenter 2^{24}, soit environ $16,78$ millions de couleurs. Donc une image (non compressée) de 512×512 pixels fait 768.432 octets, alors qu'une image de 1024×1024 pixels fait $3.145.728$ octets.

En conséquence la compression d'images est extrêmement importante.

Une observation importante est que la compression d'image peut être avec perte, en particulier perte de ces caractéristiques de l'images qui ne sont pas visible à l'œuil.

Nous savons que l'information peut être compressée seulement si elle présente de la redondance. Dans le cas de compression avec perte (d'images) on pourra compresser l'image même si elle ne présente aucune redondance mais elle présente de l'information non essentielle (en anglais : $irrelevant$), c'est à dire des caractéristiques qui peuvent être éliminées sans que l'œuil s'en aperçoive. En conséquence, une image peut être compressée (avec perte) même si il n'y a pas de redondance.

Il existe une méthode simple pour quantifier l'information perdue dans une compression avec perte. Étant donnée une image A, la compresser pour obtenir B. Ensuite décompresser B pour obtenir C. Soit $D = C - A$. Si A a été compréssée sans perte, alors $C = A$ et D sera uniformément blanche. Sinon, plus il y a eu la perte dans la compression et plus D sera loin d'être une image blanche.

Les méthodes vues jusqu'ici (RLE, quantisation scalaire, méthodes statistiques et méthodes à dictionnaire...) peuvent marcher, mais utilisées toutes seules ne donnent pas de bons résultats.

- RLE. **On a déja vu dans le premier chapitre comme RLE peut être utilisé pour compresser des images. Toute seule elle ne marche pas bien pour images, sauf celles en N/B. Et même pour les images en N/B du fax, elle es utilisée en combinaison avec la méthode de Hu□man. On verra que le format JPEG pa exemple utilise RLE combinée avec d'autres méthodes.**
- $\mathrm{Les\ méthodes\ statistiques}$. **Ces méthodes compressent bien quand les symboles ont des probabilités très di□érentes entre elles. Or, souvent ce n'est pas le cas pour les di□érentes couleurs ou niveaux de gris dans les images à tons continus. Les méthodes statistiques compressent mieux les images qui présentent des discontinuités de couleurs (pixels adjacents peuvent avoir couleur très di□érentes). Cependant, il n'est pas facile de juger à la vue si une image présente assez de discontinuité de couleur pour que la compression soit bonne.**
- $\mathrm{Les\ méthodes\ à\ dictionnaire}$. **Ces méthodes ne compressent pas bien non plus les images à tons continus. En fait, ces images présentent des pixels adjacents ayant couleurs proches, mais pas de patterns qui se répètent. Même des images qui paraissent présenter des patterns qui se répètent (P. ex. une barre verticale noire sur fond blanc) peuvent les perdre si l'image à traiter est le résultat d'une scannerisation où la feuille n'a pas été placée parfaitement verticale. Un autre problème des méthodes à dictionnaire est qu'elles scannerisent l'image horizontalement, ainsi elles négligent des régularités verticales (et donc des corrélations entre les pixels placés sur une même colonne), qui donnent lieu aussi à de la redondance qui ne sera pas éliminée. P. ex., GIF compresse beaucoup mieux une image avec des barres noires horizontales que la même image tournée de 90 degrés.**

11 Compression d'images et Correlation

La compression d'images est lié au concept de *correlation* **(ou** *redondance spatiale***) entre pixels. Des éléments d'information qui sont co-reliés peuvent être compressés de manière importante. Voyons cela avec deux exemples.**

Exemple 1. **On considère la suite suivante de données :**

$$12, 17, 14, 19, 21, 26, 23, 29, 41, 38, 31, 44, 46, 57, 53, 50, 60, 58, 55, 54, 52, 51, 56, 60$$

On note que seulement deux valeurs sont identiques et que la moyenne de ces valeurs est $40,3$**. On voit cependant qu'il y a une relation (correlation) entre chaque valeur et la suivante, ce qui est mis en évidence par la suite de leurs di□érences :**

$$12, 5, -3, 5, 2, 4, -3, 6, 11, -3, -7, 13, 4, 11, -4, -3, 10, -2, -3, 1, -2, -1, 5, 4$$

Cette suite est susceptible d'être plus fortement compressée que la précédente pour plusieurs raisons :

- **Ces di□érences ont des valeurs absolues plus petites que celles de la suite d'origine, elles pourront donc être en principe codées sur un nombre plus petit de bits.**
- **Il y a des répétitions, ce qui réduit le nombre de valeurs di□érents et donc aussi le nombre de bits pour les représenter. Ici par exemple il n'y a plus que 15 valeurs distinctes et donc 4 bits pourraient su□re pour les représenter.**
- **De plus, valeurs répétées déterminent plus facilement la création de runs.**

On peut donc utiliser la correlation pour compresser, et quand il n'y a pas de correlation apparente, on peut la créer pour l'exploiter comme illustrer par l'exemple suivant.

Exemple 2. **Soit une image** A **de résolution** 32×32 **dont les pixels ont une valeur entre 0 et 255 choisie au hasard. L'image** A **sera donc une distribution totalement random de pixels de di□érents niveaux de gris. Il n'y a donc pas de correlation entre les pixels et l'image sera di□cilement compressible. D'autre part, considérons la matrice** B **obtenue en inversant la matrice** A**. On rappelle que le calcul d'une valeur** $B[i,j]$ **de la matrice inverse dépend de toutes les valeurs de la matrice** A**, puisque cette valeur est :**

$$B[i,j] = (-1)^{i+j} \frac{det(A_{i,j})}{det(A)}$$

où $A_{i,j}$ **est la matrice (de taille** 31×31**) obtnue en e□açant la ligne** i **et la colonne** j **de la matrice** A**.**

On voit donc que deux pixels adjacent de B **auront des valeurs proches (les expressions pour** $B[i,j]$ **et** $B[i+1,j]$ **par exemple seront très similaires, puisque les matrices** $A_{i,j}$ **et** $A_{i+1,j}$ **ont beaucoup d'éléments en commun), ils sont donc co-reliés. La matrice** B **se compressera donc beaucoup mieux que la matrice** A**. On aura donc intérêt à transformer d'abord** A **en** B **et compresser** B **au lieu de** A**. Le décodeur décompressera la version compressée de** B **et appliquera la transformation dans le sens opposé pour retrouver** A**.**

Ceci est le principe de l'approche des *transformées***. On transforme une image en une autre qui présente plus de correlation et on compresse cette dernière. Dans cet exemple, la transformation consiste simplement à prendre la matrice inverse.**

1 Compression d'images et Luminance

Par expérience, dans une image, la luminosité des pixels est aussi co-reliée. Si un pixel est clair (lumineux) alors très souvent ses voisins seront aussi clairs (lumineux) même s'ils sont d'une autre couleur. De plus, l'œuil humain est beaucoup plus sensible aux variations de luminosité qu'aux variations de couleurs. On peut voir l'ensemble de toutes le couleurs comme un espace vectoriel tridimensionnel généré par la base constitué des trois vecteurs $\mathbf{R} = (1, 0, 0)$, $\mathbf{G} = (0, 1, 0)$ et $\mathbf{B} = (0, 0, 1)$ (p. ex, la couleur $(78, 202, 113)$ est obtenue comme la combinaison linéaire : $78 \cdot \mathbf{R} + 202 \cdot \mathbf{G} + 113 \cdot \mathbf{B}$).
Il convient en fait d'exprimer les couleurs en fonction d'une autre base que RGB. En e□et, on peut définir un vecteur Y, dit $luminance$, qui est une combinaison linéaire de RGB et ayant la propriété que la composante d'une couleur en direction mesure le degré de luminosité du pixel. Ce vecteur Y a une forte composante verte et une faible composante rouge et bleu, parce que c'est surtout la couleur verte qui donne la sensation de luminosité. De plus, on peut bien entendu trouver deux vecteurs C_R et C_B (à une forte composante rouge et bleue, respectivement) qui forment avec Y une nouvelle base.

Les composantes Y de pixels adjacents auront valeurs proches, elles seront donc co-réliées et on pourra exploiter cette propriété de correlation pour une meilleur compression.

6.2 Présentation des approches de la compression d'images.

1. **Pour images binaires (en N/B). Digitaliser l'image dans l'ordre du rateau (ligne par ligne, de droite vers la gauche) et utiliser RLE. Les longueurs des runs peuvent ensuite être codées par un code préfixe à taille variable (méthode statistique à la Hu□man comme pour le fax). Variantes : digitaliser l'images par d'autres ordres qui pourraient s'avérer plus avantageux : par colonnes, en zig-zag...**
2. **Pour images binaires (en N/B). Utiliser les $quadtrees$. Il sont définit récursivement. Supposons d'avoir une image carrée de taille $2^k \times 2^k$ (si ce n'est pas le cas, on peut toujours "border" l'image avec un nombre opportun de colonnes ou de lignes de zéros pour la rendre carrée et ayant comme coté une puissance de 2). Si une image est entièrement blanche (resp. noire) alors son quadtree est un arbre constituée d'un seul nœud (sa racine) ayant couleur blanc (resp. noir). Sinon, le quadtree associé à m'image est un arbre constituée d'une racine (n'ayant pas de couleur) ayant quatre sous-arbres, qui sont les quadtrees des images obtenues en divisant l'image en quatre (correspondants aux quadrants Nord-Ouest, Nord-Est, Sud-Ouest et Sud-Est).**
3. **Pour images à plusieurs niveaux de gris. Chaque bitplan d'une telle image est en fait équivalent à une image en N/B. On donc peut appliquer à chaque bitplan les approches vues pour les images N/B.**
   Toutefois il est important de faire une considération. Le fait que pixels adjacents aient tonalité de gris proche, n'implique pas forcément qu'ils ont composantes identiques sur la plupart des bitplans. P. ex., si dans une image à 4 bitplans deux pixels adjacents ont couleur 7 (0111) et 8 (1000), alors sur chacun des quatre bitplans la valeur va changer (de 0 a 1 sur le 1er bitpan, de 1 à 0 pour les autres. Ceci est dû au fait que deux nombres consécutifs n'ont pas forcément des représentations similaire en binaire. Si on veut que pour deux pixels adjacents de couleur proche, un maximum de bitplans présentent valeurs égales (ce qui résulte à priori en runs de plus grande longueur), il faut utiliser pour les entiers un codage ayant la propriété que le codes de deux entiers consécutif di□èrent pour au plus un bit (les codes ordinaires de 7 et 8 di□èrent sur 4 bits). Un tel code existe, il s'appelle $code$

de Gray. **Le code de Gray de rang n, noté $G(n)$, utilise des mots de longueur n (n bits) pour représenter les entiers entre 0 et $2^n - 1$. Il est défini récursivement comme suit :**

$$G(1) = \{0, 1\}$$

(on code l'entier 0 par le mot `0` et l'entier 1 par le mot `1`)

$$G(n) = 0 \cdot G(n-1) \cup 1 \cdot \widetilde{G(n-1)}$$

où $G(n-1)$ dénote bien entendu le code de Gary de rang $n-1$, l'opérateur $\cdot$ dénote la concaténation (ainsi $0 \cdot G(n-1)$ représente l'ensemble de tous le mots obtenus en concaténant un 0 avec tous les mots de $G(n-1)$, alors que l'opérateur $\widetilde{\ }$ dénote le renversé (ainsi $\widetilde{G(n-1)}$ est l'ensemble des mots $G(n-1)$ parcouru de la fin au début)
P. ex :

$$G(2) = 0 \cdot G(1) \cup 1 \cdot \widetilde{G(1)} = 0 \cdot \{0, 1\} \cup 1 \cdot \{1, 0\} = \{00, 01, 11, 10\}$$

ce qui donne : 0 codé par `00`, 1 codé par `01`, 2 codé par `11`, 3 codé par `10`

$$\begin{aligned} G(3) &= 0 \cdot G(2) \cup 1 \cdot \widetilde{G(2)} && \textbf{(6.1)} \\ &= 0 \cdot \{00, 01, 11, 10\} \cup 1 \cdot \{10, 11, 01, 00\} && \textbf{(6.2)} \\ &= \{000, 001, 011, 010, 110, 111, 101, 100\} && \textbf{(6.3)} \end{aligned}$$

donc : 0 codé par `000`, 1 codé par `001`, 2 codé par `011`, 3 codé par `010`, 4 codé par `110`, 5 codé par `111`, 6 codé par `101`, 7 codé par `100`. Quand les entiers sont codés par le code de Gray, on voit tout de suite apparaître une plus grande régularité sur les bitplans, notamment, moins de distribution random et plus de runs (voir figures 4.9, 4.10 et 4.11 dans le livre de David Salomon), ce qui permet de compresser davantage.

4. **Utiliser le *contexte* d'un pixel pour *prédire* sa valeur.**
   Soit une image P. On choisit un contexte pour chaque pixel $P[i, j]$ (p. exemple les 4 pixels ayant un coté en commun avec lui, ou les 8 pixels ayant un coté ou on sommet en commun avec lui) et on calcule la moyenne des valeurs pixels du contexte. Soit la matrice A telle que$A[i, j]$ est la moyenne des valeurs des pixels dans le contexte de $P[i, j]$
   Le principe de la compression d'image nous dit qu'en général la matrice A est presque partout égale à P (ou ses valeurs sont presque partout très proches des valeurs de P).
   Une fois calculé la matrice A, on peut calculer la matrice $\Delta = P - A$. Cette matrice est composée de beaucoup de zéro et de beaucoup de valeur très petites. En fait, la moyenne représente la valeur qu'on peut *prédire* pour P, elle est donc une information redondante qui peut être éliminée. Ma matrice Δ se compresse donc bien par des méthodes standard (RLE, quantisation, ou méthodes statistique qui assignent des codes plus courtes aux petites valeurs qui seront plus fréquentes). L'encodeur peut donc envoyer Δ au lieu de P.
   Le contexte peut être défini de plusieurs manières, il peut même être constitué de un ou deux pixels seulement. On peut aussi attribuer des poids plus forts aux pixel du contexte qui sont plus proches. On obtient ainsi une meilleure prédiction et une meilleure compression. Le décodeur reçoit l'encodée de Δ, la décode, et commence à calculer A. Pour chaque $A[i, j]$, il pourra calculer $P[i, j] = A[i, j] + \Delta[i, j]$ et reconstruire l'image d'origine.
   Note. Pour que le calcul de A soit possible il faut que le premier pixel soit envoyé sous forme brute et que le contexte utilise uniquement des pixels qui ont déjà été décodé. Par exemple, le contexte du deuxième pixel codé ne pourra que contenir le premier.

5. **Transformer l'image et encoder la transformée. La méthode des transformées sera présentée en détail dans la prochaine section. En général il s'agit d'éliminer la correlation et obtenir une version de l'image où les pixels sont non correliés. En éliminant la redondance on crée aussi des valeurs beaucoup plus communes alors que d'autres sont rare (petite entropie). La quantisation marche aussi bien sur les images transformées.**

   Regardons maintenant les images en couleurs.

6. **Séparer l'image en trois images à niveaux de gris et appliquer l'une des trois méthodes 3, 4, ou 5. Préférablement en utilisant une décomposition, luminance, chrominance, couleur plutôt que la représentation RGB.**
7. **Les images à tons discrets peuvent profiter d'un traitement particulier. En général elle sont composées d'un assemblage d'éléments de base (rectangles, lignes, courbes) qui sont répétés à di□érentes positions, à di□érentes échelles, tournées d'angles di□érents. On code chacun des éléments de base ainsi que les transformation qu'il a subi aux di□érents endroit de l'image (translation, rotation...).**

6.3 Transformées.

À VENIR

Annexe A

Titre

Annexe B

Titre

Conclusion et discussion