Chapitre III

Codage par dictionnaire

1 Introduction

Jusqu'à présent nous avons considéré que les sources produisaient des suites de symboles indépendants.

> Tirage aléatoire des symboles indépendamment les uns des autres (sans mémoire du passé, ni connaissance du futur).

Dans cette leçon, nous aborderons des techniques qui utilisent la structure des données afin d'augmenter le taux de compression.

> Recherche des motifs qui apparaissent le plus souvent.

Utile pour les sources qui génèrent de petits nombres de motifs relativement fréquemment.
Exemple : texte, image, communication par modem

Pour un texte écrit :
- Celui-ci est constitué d'une succession de mots ou syllabe.
- Chaque mot ou syllabe pourrait être considéré comme un motif.

Une approche possible serait de disposer :
- d'une liste (ou d'un dictionnaire) contenant les motifs les plus fréquents.
- lorsque qu'un des motifs de la liste apparaît, il est codé par sa référence dans le dictionnaire.
- si une suite de symboles ne fait pas partie du dictionnaire, il est codé avec une autre méthode moins efficace.

En conséquence, cela revient à séparer l'ensemble des entrées en deux classes :
- la classe des motifs fréquents.

- l'ensemble des autres motifs.

Considérons l'approche suivante :

- prenons un alphabet restreint à 32 caractères (A-Z plus quelques symboles supplémentaires) : une lettre peut être codée sur 5 bits.
- on dispose d'un dictionnaire des 256 motifs de longueur 4 les plus fréquents. Le numéro du motif dans le dictionnaire peut être codé sur 8 bits.
- lorsque l'on lit la source à coder :
  - ◇ si l'on rencontre un motif, on écrit 1 suivit du code du motif.
  - ◇ sinon on écrit 0 suivi du code de la lettre.

Quelle est la performance d'un tel code ?

Sans compression, on a besoin de 5 bits par symbole.
Soit p la proportion du texte qui contient un motif du présent dans le dictionnaire.
Avec compression, le nombre moyen de bits par symbole est :

$$p.(1+8)/4 + (1-p).(1+5) = 6 - 3,75.p$$

Pour que le codage soit efficace, il faut donc que $6 - 3,75p < 5$, ce qui conduit à $p > 1/3,75 = 26,7\%$, ce qui est considérable.

Afin de trouver un codage sensiblement meilleur, il faut faire en sorte que p soit le plus grand possible.

2 Dictionnaire statique

Par dictionnaire statique, on entend que le dictionnaire contenant les motifs est calculé à l'avance en fonction des caractéristiques de la source.
Le choix d'un tel dictionnaire est indiqué que lorsqu'il est possible de trouver une répétition importante de motifs pour la source (voir le cas précédent).
Évidemment, si la fréquence des motifs n'est pas suffisante (ou que la méthode de compression est utilisée sur une source un peu différente), les données seront dilatées plutôt que compressées.

2.1 Dictionnaire statique des digrammes

Un digramme est une combinaison de deux lettres, couramment utilisé lors de la compression ou l'analyse de texte.
L'intérêt des digrammes vient du fait que certaines associations de lettres sont beaucoup plus fréquentes que d'autres (le, es, ...).

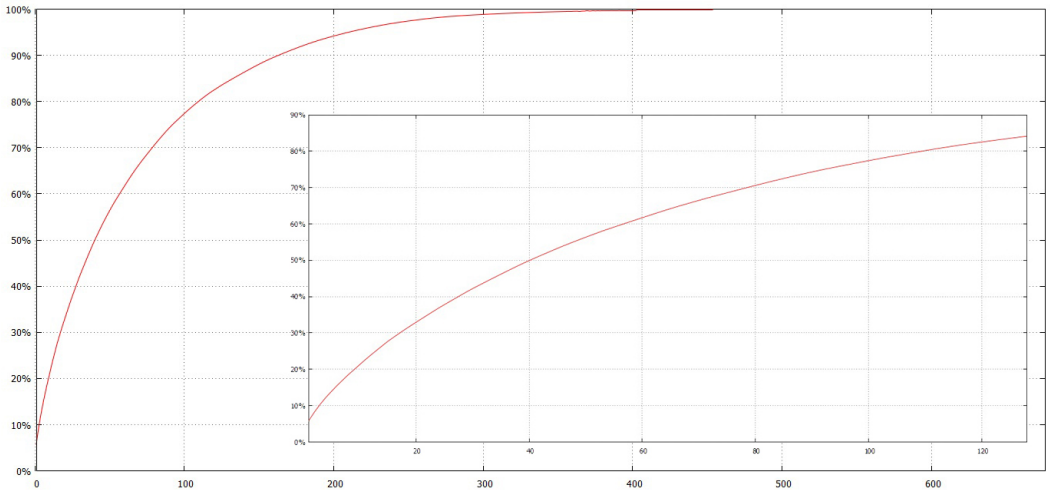


FIGURE III.1 – Fréquence cumulée des digrammes dans la langue française.

La fréquence de ces associations dépend de la langue.

Exemple : fréquences cumulées pour la langue française

Total combinaison de 2 lettres : $26 \times 26 = 676$. Le résultat est présenté à la figure III.1.

Donc, avec 128 entrées dans le dictionnaire, on recouvre plus de 80% des digrammes.

Exemple de codage d'un texte avec les digrammes

Le codage est sur 8 bits (256 valeurs) :

- 0-94 : table ASCII restreinte aux caractères nécessaires pour coder le texte.
- 95-255 : code c du digramme ($c-95$ = numéro d'index du digramme dans le dictionnaire D), soit 161 entrées dans le dictionnaire.

Le fonction $C(v)$ retourne le code associé à l'entrée v du dictionnaire.
L'algorithme est présenté à la figure III.2. La décompression est triviale : chaque code est de longueur fixe et correspond soit au code d'un caractère ou d'un digramme.
161 digrammes couvrent $89,6\%$ des digrammes les plus fréquents.
Taille sans compression $\simeq 6$ bits/symbole.
Taille avec compression $= 0.896 \times 4 + (1-0.896) \times 8 = 4,416$ bits/symbole.
Taux de compression $= S_c/S_i = 4,416/6 = 0.736$
Pourcentage de réduction $= (S_i - S_c)/S_i = 0.264 = 26,4\%$

```
DigrammeCompress(v_1 ... v_n)
    i = 1;
    while (i < n) do
        if v_i v_{i+1} ∈ D then
            s ← C(v_i v_{i+1});
            i = i + 2;
        else
            s ← C(v_i);
            i = i + 1;
```

FIGURE III.2 – Exemple de compression avec un dictionnaire statique.

Notez que l'utilisation de trigramme ne permet en général pas de gain dans le cas du langage naturel car le nombre de trigramme est beaucoup plus important, et ils sont en conséquence aussi beaucoup plus nombreux et rares.

Exemple pratique

Soit un source avec un alphabet $\mathcal{S} = \{a, b, c, d, r\}$.

Alphabet	Code
a	000
b	001
c	010
d	011
r	100

Dictionnaire	Code
ab	101
br	110
ra	111

Source : abracadabra
Codage : ab ra c a d ab ra ⇒ 101 111 010 000 011 101 111
Décodage : 101111010000011101111 ⇒ abracadabra

2.2 Passages au dictionnaires adaptatifs

On s'intéresse maintenant au cas des dictionnaires adaptatifs.

adaptatif = le dictionnaire est construit au vol à la lecture lors de la compression de la source.

Ce type d'algorithme de compression est répandu et possède de nombreuses variantes.

Nous en exposons ici 3 familles :

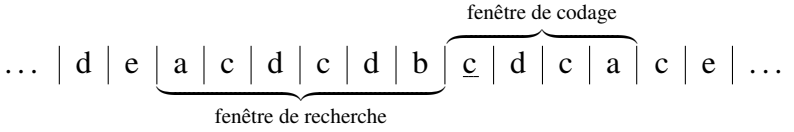
- `LZ77` de Jacob Ziv et Abraham Lempel [ZL77].
- `LZ78` toujours de Jacob Ziv et Abraham Lempel [ZL78].

- LZW de Terry Welch [Wel84].

3 Dictionnaire adaptatif LZ77

3.1 Codage

Dans cette approche, le dictionnaire est simplement une portion de la chaîne déjà codée.
L'encodeur utilise deux fenêtres glissantes :



- la fenêtre de recherche contient une portion de l'historique de codage.
  C'est la zone dans laquelle on recherche le plus long motif.
  Peut être vu comme un dictionnaire glissant.
- la fenêtre de codage contient la portion de la source que l'on cherche à coder.
  Le début de la fenêtre de codage coïncide avec le pointeur de lecture.
  C'est la zone qui contient le motif que l'on va rechercher, et que l'on va progressivement allonger s'il est trouvé dans la fenêtre de recherche.

La largeur de ces fenêtres reste fixe pendant tout le codage.

Le code renvoyé est de la forme (*Offset*, *Longueur*, *Symbole*)

- si un motif est trouvé : *Offset* est de combien il faut se décaler vers la gauche pour trouver le motif, *Longueur* est la longueur du motif et *Symbole* le symbole qui suit le motif dans la chaîne de codage.
- si aucun motif n'est trouvé, $Offset = 0$, $Longueur = 0$ et *Symbole* est le symbole sur le pointeur de lecture.

Exemple (voir ci-dessus)

Le plus long motif en partant du début de la fenêtre de codage que l'on peut trouver dans la fenêtre de recherche est cdc, il faut se décaler de 5 vers la gauche pour le trouver ($Offset = 5$), il est de longueur 3 ($Longueur = 3$) et le symbole suivant le motif est a. Le code est donc $(5, 3, a)$. Les deux codes suivants sont $(2, 1, c)$ et $(0, 0, e)$.

Pour envoie-t-on le code du symbole suivant le motif ?

Si le pointeur de lecture se trouve sur un caractère non présent dans la fenêtre de recherche, il faut envoyer directement le symbole.

Si ce n'est pas le cas, plutôt que de perdre les bits utilisés pour ce code, on les utilise afin de stocker le symbole qui suit le motif.

Longueur du code binaire

Il faut choisir un code binaire de longueur :

$$\lceil \log_2 |\mathcal{S}| \rceil + \lceil \log_2 L_r \rceil + \lceil \log_2 L_c \rceil$$

où $|\mathcal{S}|$ est la taille de l'alphabet, L_r est la largeur de la fenêtre de recherche et L_c est la largeur de la fenêtre de codage.

Exemple

Pour du code `ASCII`, $|\mathcal{S}| = 256$.
Prenons $L_r = 32$ et $L_c = 8$.
On a besoin d'un code $8 + 5 + 3 = 16$ bits.

Exemple de codage

Entrée : "un chasseur sachant chasser doit savoir chasser sans son chien"
Taille des fenêtres : recherche $L_r = 20$, codage $L_c = 8$
Supposons que le pointeur de lecture soit au niveau du premier "chasser".

recherche	codage	sortie
...un␣**chasse**ur␣sachant␣	**chasse**r␣doit␣...	$(17, 6, r)$
...␣chas	seur␣sachant␣chasser␣	doit␣savoir␣...	$(8, 1, d)$
...hasse	ur␣sachant␣chasser␣d	oit␣savoir␣ch...	$(0, 0, o)$
...asseu	r␣sachant␣chasser␣do	it␣savoir␣cha...	$(0, 0, i)$
...sseur	␣sachant**␣**chasser␣doi	t␣savoir␣chas...	$(13, 2, s)$
...ur␣sa	chant␣ch**a**sser␣doit␣s	avoir␣chasser...	$(12, 1, v)$
...␣sach	ant␣chasser␣doit␣sav	**oi**r␣chasser␣s...	$(7, 2, r)$
...chant	**␣chasser**␣doit␣savoir	**␣chasser**␣sans...	$(20, 8, ␣)$
...sser␣	doit␣**sa**voir␣chasser␣	**sa**ns␣son␣chie...	$(15, 2, n)$
...r␣doi	t␣savoir␣chasser␣	**s**ans␣son␣chien...	$(3, 1, ␣)$

etc ...

Remarque

La fenêtre de recherche ne contient rien au début du codage.

Aussi, cela conduit à :
- de nombreuses insertions de triplets $(0, 0, c)$,
- un dilatation des symboles codés ;

ceci le temps que le fenêtre de recherche contiennent assez de symboles pour permettre à `LZ77` d'être efficace.

Afin d'éviter ce problème, les L_r premiers symboles (ou une fraction de L_r suffisamment importante) sont transmis directement en binaire (*i.e.* sans codage).

Entrées :
L_r largeur de la fenêtre de recherche
L_c largeur de la fenêtre de codage
$S[1 \ldots n]$ le suite de symboles à coder.
// transmet les L_r premiers symboles sans codage
sortie $\leftarrow S[1 \ldots L_r]$
// codage du reste
$i = L_r + 1$
while *($i<n$)* **do**
// trouve le motif le plus long de la fenêtre de recherche
// dans la fenêtre d'historique
// Longueur = longueur du motif trouvé
// $Offset$ = offset du motif trouvé dans la fenêtre d'historique
// si aucun motif n'est trouvé, renvoie $(0, 0)$
$(Offset, Longueur) =$
FindPattern($S[i, \ldots, i + L_c - 1]$,$S[i - L_c, \ldots, i - 1]$)
// sortie du codage
sortie $\leftarrow \langle Offset, Longueur, S[i + Longueur]\rangle$
$i+ = Longueur + 1$

FIGURE III.3 – Algorithme de codage LZ77.

Algorithme de codage

L'algorithme de codage est présenté à la figure III.3 Les symboles codés sont envoyés au fur et à mesure sur la sortie.

3.2 Décodage

Algorithme de décodage

L'algorithme de décodage est présenté à la figure III.4

Entrées :
L_r largeur de la fenêtre de recherche
L_c largeur de la fenêtre de codage
$C[1 \ldots n]$ la suite de codes à décoder.
Sortie :
$S[1 \ldots]$ suite des symboles décodés.

```
// les L_r premiers symboles ne sont pas codés
S[1...L_r] ← C[1...L_r]
// décodage du reste
i = L_r + 1
while ( ! EOF) do
    ReadTriplet(⟨Offset, Longueur, c⟩)
    if (Longueur=0) then
        // un seul caractère
        S[i++] = c
    else
        // suite de caractères (copie depuis les caractères déjà décodés)
        S[i,...,i+Longueur-1] = S[i-Offset,...,i-Offset+Longueur-1]
        S[i+Longueur] = c
        i += Longueur+1
```

FIGURE III.4 – Algorithme de codage LZ77.

EXERCICE 27: Compression LZ77

1. Utiliser LZ77 avec un fenêtre de recherche de 8 et un longueur de codage de 8 pour compresser la phrase "si ta tata tasse ta tata, ta tata tassee sera".
2. Donner la longueur du code binaire obtenu, et la comparer au texte non compressé.
3. Utiliser LZ77 avec une fenêtre de recherche de 16 et un longueur de codage de 8 pour décompresser :
   - Fenêtre de recherche non compressée : 'trois petites tr'
   - Offset/Longueur/Suivant : (0/0/u) (8/5/c) (7/5/,) (16/3/o) (9/1/s) (6/1/p) (11/1/t) (15/4/_) (14/2/u) (8/5/c) (8/2/e) (6/1/_)

3.3 Remarques

Remarque

LZ77 est une méthode adaptative simple qui n'a besoin d'aucune connaissance

a priori sur la source.

De nombreux compresseurs utilisent des variations de LZ77 suivis d'un codage à taille variable. Citons pkzip, zip, LHarc, PNG, gzip, arj.

Dans l'exemple précédent, on voit que pour être efficace, l'historique doit être suffisamment grand pour permettre de trouver des motifs de recherche assez long avec une fréquence suffisamment grande.

Exemple : zip et gzip utilisent une fenêtre d'historique de 32Ko et une fenêtre de recherche de 258 octets.

Performance

La performance de cet algorithme approche asymptotiquement ce que l'on peut obtenir au mieux lorsque l'on a une connaissance des caractéristiques (et statistiques) de la source [ZL77].
Asymptotiquement = quand on fait tendre la taille des données vers ∞. La vitesse de convergence peut être lente. La méthode peut donc être améliorée.

Améliorations

- ajout d'un bit de flag pour indiquer le codage d'un caractère unique, et si le caractère n'est pas unique, transmettre seulement l'offset et la longueur (aussi nommée LZSS, voir [SS82])
- utiliser des codes de taille variable pour stocker le triplet.
- semi-adaptatif : algorithme en deux passes.
- largeurs des fenêtres variables.
- ...

Elles sont utilisées dans les différentes variations de LZ77.

4 Dictionnaire adaptatif LZ78

4.1 Codage

LZ77 suppose que les motifs similaires apparaissent à proximité les uns des autres. La fenêtre de recherche est toujours à proximité directe de la fenêtre de codage. Donc, tout motif se répétant à une fréquence plus grande que la largeur de la fenêtre de recherche est manqué. LZ78 résout ce problème en créant au vol un dictionnaire adapté aux données en cours de compression.

Problème

Le dictionnaire doit pouvoir être construit de manière identique, à la fois par l'encodeur et par le décodeur.

Que contient le dictionnaire ?

Le dictionnaire $D = \left\{m_j\right\}_{j=1\ldots p}$ représente un tableau de motifs, où m_j est le motif dont le numéro d'ordre est j (noter que j commence à 1).

Les insertions de nouveaux motifs s'effectuent toujours à la fin du dictionnaire.

Pour le codage

On note $s_1, \ldots, s_n$ la suite de symboles à coder.

On part du premier symbole $i = 1$ avec un dictionnaire vide $D = \emptyset$.

Si s_i n'est pas dans le dictionnaire, alors le code de s_i est $(0, s_i)$, on insère s_i à la fin du dictionnaire, et on passe au symbole suivant.

Sinon on recherche le plus long motif $m_j = s_i \ldots s_p$ présent dans le dictionnaire, on le code (j, s_{p+1}), on insère le motif $s_i \ldots s_{p+1}$ dans le dictionnaire, et on passe au symbole s_{p+2}.

Exemple

Entrée : "si␣ton␣tonton␣tond␣ton␣tonton␣
ton␣tonton␣sera␣tondu␣par␣ton␣tonton"

Codage	i	Dict.
$\langle 0, s\rangle$	1	s
$\langle 0, i\rangle$	2	i
$\langle 0, \text{␣}\rangle$	3	␣
$\langle 0, t\rangle$	4	t
$\langle 0, o\rangle$	5	o
$\langle 0, n\rangle$	6	n
$\langle 3, t\rangle$	7	␣t
$\langle 5, n\rangle$	8	on
$\langle 4, o\rangle$	9	to
$\langle 6, \text{␣}\rangle$	10	n␣
$\langle 9, n\rangle$	11	ton
$\langle 0, d\rangle$	12	d
$\langle 7, o\rangle$	13	␣to
$\langle 10, t\rangle$	14	n␣t
$\langle 8, t\rangle$	15	ont

Codage	i	Dict.
$\langle 8, \text{␣}\rangle$	16	on␣
$\langle 11, \text{␣}\rangle$	17	ton␣
$\langle 11, t\rangle$	18	tont
$\langle 16, s\rangle$	19	on␣s
$\langle 0, e\rangle$	20	e
$\langle 0, r\rangle$	21	r
$\langle 0, a\rangle$	22	a
$\langle 13, n\rangle$	23	␣ton
$\langle 12, u\rangle$	24	du
$\langle 3, p\rangle$	25	␣p
$\langle 22, r\rangle$	26	ar
$\langle 23, \text{␣}\rangle$	27	␣ton␣
$\langle 18, o\rangle$	28	tonto
$\langle 8, \texttt{EOF}\rangle$	-	- (n)

Codage LZ78

Entrée : suite $s_1, \ldots, s_n$ de symboles à coder
// Dictionnaire initialement vide
$D = \{\emptyset\}$
// Motif actuel
$m = ''$
// boucle sur le reste des symboles
while *(!EOF)* **do**
  e = ReadSymbol()
  if *(ExistInDictionnary(me))* **then**
    // extension du motif
    $m = me$
  else
    // indice du motif dans le dictionanire (0 si absent ou vide)
    $i = IndexInDictionnary(m)$
    // code pour ce motif
    sortie $\leftarrow (i, e)$
    // mise-à-jour du dictionnaire
    $AppendToDictionnary(me)$
    // on repart de rien
    $m = ''$

4.2 Décodage

Analyse du codage

Pour chaque couple (i, c) utilisé comme code :

- i fait toujours référence à un motif m_j déjà existant dans le dictionnaire.
- c est le code du caractère qui, ajouté au motif m_j permet de créer une nouvelle entrée dans le dictionnaire.

En conséquence, à chaque fois que le décodeur reçoit un code (i, c), il sait qu'il doit :

- décoder (i, c) comme la suite de symboles $m_i c$
- insèrer le nouveau motif $m_i c$ comme nouvelle entrée dans le dictionnaire.

L'encodeur et le décodeur ainsi construisent des dictionnaires symétriques.

Décodage LZ78

```
// Dictionnaire initialement vide
D = {∅}
// boucle sur le reste des codes
while (!EOF) do
    // lecture du code
    (i, e) = ReadCode()
    // recherche du motif codé par i
    m = FindInDictionnary(i)
    // décodage en me
    sortie ← me
    // ajout de me au dictionnaire
    AppendToDictionnary(me)
```

EXERCICE 28: Compression LZ78

1. Utiliser LZ78 pour compresser la phrase "si ta tata tasse ta tata, ta tata tassee sera".
2. Donner la longueur du code binaire obtenu, et la comparer au texte non compressé.
3. Décoder le code LZ78 suivant : `(0,s) (0,i) (0,x) (0,_) (0,c) (0,e) (0,n) (0,t) (4,s) (2,x) (9,u) (2,s) (1,e) (1,_) (1,u) (5,e) (7,t) (9,i) (3,_) (16,n) (8,_) (1,i) (19,s) (0,a) (0,u) (5,i) (1,s) (6,s)`

4.3 Remarques

Remarques

- Par rapport à LZ77, on insère des couples au lieu de triplets.
- Les mots dans le dictionnaire sont de plus en plus long au fur et à mesure de la répétition des mots (chaque répétition fait grandir le motif de 1).
- Le dictionnaire grossit sans cesse au fur et à mesure de l'arrivée de nouveaux symboles.
- Tous les motifs créés sont conservés en mémoire.

En pratique, comment gérer un dictionnaire qui peut grossir indéfiniment ?

Différentes stratégies existent :

- ajouter un bit à l'index à chaque fois que l'on atteint le maximum de stockage du dictionnaire.
  Exemple : si on démarre avec un index 8 bits, lorsque l'on reçoit le motif associé à l'entrée 255 du dictionnaire, on sait que les prochains seront codés sur 9 bits, etc ...
- une fois que le dictionnaire à atteint une taille maximale (index sur $b_{\max}$ bits, 16 par exemple), on utilise le dictionnaire construit comme un dictionnaire statique.
- si le taux de compression est au-dessus d'un certain seuil, on détruit le dictionnaire courant et on commence la construction d'un nouveau dictionnaire.
  On peut également purger du dictionnaire tous les motifs peu utilisés.

Attention : ce codage fait l'objet d'un brevet logiciel aux États-Unis.

5 Dictionnaire adaptatif LZW

5.1 Codage

L'algorithme LZW [Wel84] est une modification célèbre de LZ78.
Afin d'éviter d'avoir à envoyer le second élément de la paire $\langle i, c \rangle$,

- Le dictionnaire D est initialisé avec son alphabet (par exemple, pour coder un texte, D est initialisé avec les codes ASCII).
- La construction du plus grand motif est permanente :
  - ◇ tant que le motif est dans le dictionnaire, on étend le motif.
  - ◇ on insère tout nouveau motif dans le dictionnaire.
- Le dictionnaire est donc mis à jour au fur et à mesure.
- Un motif inséré est toujours l'extension d'un caractère d'un motif déjà existant dans le dictionnaire.
- Le codage consiste en l'indice du motif dans le dictionnaire contenant le code à utiliser.

Nous donnons immédiatement un exemple afin de comprendre la construction.

Codage LZH

// lecture du premier symbole
e_1 = ReadSymbol()
$w_1 = e_1$
sortie ← DictionnaryCode(e_1)
// boucle sur le reste des symboles

w_{n-1}	e_n	$w_{n-1}e_n$	$\in D$	w_n	sortie	Dictionnaire
	A		$\in$	A		$A(1), B(2), C(3)$
A	C	AC	$\notin$	C	1	$+AC(4)$
C	B	CB	$\notin$	B	3	$+CB(5)$
B	B	BB	$\notin$	B	2	$+BB(6)$
B	A	BA	$\notin$	A	2	$+BA(7)$
A	A	AA	$\notin$	A	1	$+AA(8)$
A	C	AC	$\in$	AC		
AC	EOF				4	

TABLE III.1 – Codage LZH de ACBBAAC.

```
while ( !EOF) do
    e_n = ReadSymbol()
    // teste si le motif étendu est dans le dictionnaire
    if (ExistInDictionnary(w_{n-1}e_n)) then
        // le motif est dans le dictionnaire : construction d'un motif plus grand
        w_n = w_{n-1}e_n
    else
        // on redémarre un nouveau motif avec un seul symbole
        w_n = e_n
        // le motif n'est pas dans le dictionnaire : on l'ajoute
        DictionnaryAdd( w_{n-1}e_n )
        // on envoie sur la sortie le motif construit le plus grand
        // qui appartient au dictionnaire
        sortie ← DictionnaryCode(w_{n-1})
```

Exemple 1 (codage)

Entrée : ACBBAAC
Alphabet : S = $\{A, B, C\}$
Voir le résultat du codage à la table III.1.

Le code est donc : 1 3 2 2 1 4.

Dictionnaire à la fin de l'encodage :
$D = \{A(1), B(2), C(3), AC(4), CB(5), BB(6), BA(7), AA(8)\}$

w_{n-1}	e_n	$w_{n-1}e_n$	$\in D$	w_n	sortie	Dictionnaire
	A		$\in$	A		$A(1), B(2)$
A	A	AA	$\notin$	A	1	$+AA(3)$
A	A	AA	$\in$	AA		
AA	B	AAB	$\notin$	B	3	$+AAB(4)$
B	A	BA	$\notin$	A	2	$+BA(5)$
A	A	AA	$\in$	AA		
AA	B	AAB	$\in$	AAB		
AAB	B	AABB	$\notin$	B	4	$+AABB(6)$
B	B	BB	$\notin$	B	2	$+BB(7)$
B	B	BB	$\in$	BB		
BB	EOF				7	

TABLE III.2 – Codage LZH de AAABAABBBB.

Exemple 2 (codage)

Entrée : AAABAABBBB
Alphabet : S = $\{A, B\}$
Voir le résultat du codage à la table III.2.
Le code est donc : 1 3 2 4 2 7.
Dictionnaire à la fin de l'encodage :
$D = \{A(1), B(2), AA(3), AAB(4), BA(5), AABB(6), BB(7)\}$

Analyse du codage :

- on n'envoie (évidemment) sur la sortie que les motifs présents dans le dictionnaire.
  Le passage au motif suivant n'a lieu que lorsque le motif étendu n'est plus présent dans le dictionnaire.
- Un motif ms inséré dans le dictionnaire est la concaténation d'un motif m déjà inséré dans le dictionnaire, et d'un nouveau symbole s.
- Au moment où le motif ms est inséré dans le dictionnaire, le code du motif m est utilisé.
- Mieux, le symbole ou le motif qui suit m est s.

Donc, en reconstituant les portions de motifs utilisés pour construire le dictionnaire, on peut construire un dictionnaire symétrique dans le décodeur.

5.2 Décodage

Principe du décodage

- Le dictionnaire est initialisé, de manière symétrique, avec l'ensemble des codes de l'alphabet.
- On part de e_1 = code dans le dictionnaire du premier caractère, puis pour chacun des codes c suivant reçu, on applique la méthode suivante.
- si un symbole c reçu est déjà dans le dictionnaire,
  alors on obtient immédiatement la nouvelle chaîne courante $e_n = D(c)$.
- on sait que dans un tel cas, il n'y a pas de chaîne plus longue actuellement dans le dictionnaire. En conséquence, pour la prochaine chaîne lue e_n, il faudra insérer le mot $w = e_{n-1}e_n[0]$ dans le dictionnaire.
- si le symbole n'est pas dans le dictionnaire, c'est que la chaîne e_n est une répétition de la chaîne précédente e_{n-1} avec un symbole supplémentaire, à savoir $w = e_n = e_{n-1}e_{n-1}[0]$.
- puis, on insère w dans le dictionnaire, et on envoie e_n sur la sortie.

Décodage `LZH`

```
// lecture du premier code
e_1 = ReadCode()
sortie ← DictionnaryCode(e_1)
// boucle sur le reste des codes
while (!EOF) do
    c = ReadCode()
    if (c ∉ D) then
        // cas particulier où le code lu n'est pas encore inséré (voir exemple 2)
        e_n = w = e_{n-1}e_{n-1}[0]
    else
        // lecture du code dans le dictionnaire
        e_n = D(c)
        // construction du code à insérer dans le dictionnaire
        w = e_{n-1}e_n[0]
    // insertion du nouveau code dans le dictionnaire
    D = D ∪ {w}
    // sortie du motif associé au code
    sortie ← e_n
```

c	c∈D	e_n	w	sortie	Dictionnaire
					$A(1), B(2), C(3)$
1	∈	A		A	
3	∈	C	AC	C	$+AC(4)$
2	∈	B	CB	B	$+CB(5)$
2	∈	B	BB	B	$+BB(6)$
1	∈	A	BA	A	$+BA(7)$
4	∈	AC	AA	AC	$+AA(8)$
EOF					

TABLE III.3 – Décodage LZH de 1 3 2 2 1 4 avec $\mathbf{S} = \{A, B, C\}$.

Exemple 1 (décodage)

Entrée : 1 3 2 2 1 4
Alphabet : S = $\{A, B, C\}$
Voir le résultat du décodage à la table III.3.
Le décodage est donc : ACBBAAC.
Dictionnaire à la fin du décodage :
$D = \{A(1), B(2), C(3), AC(4), CB(5), BB(6), BA(7), AA(8)\}$
Il est donc identique au dictionnaire d'encodage.

Exemple 2 (décodage)

Entrée : 1 3 2 4 2 7
Alphabet : S = $\{A, B\}$
Voir le résultat du décodage à la table III.4.
Le décodage est donc AAABAABBBB.
Dictionnaire à la fin du décodage :
$D = \{A(1), B(2), AA(3), AAB(4), BA(5), AABB(6), BB(7)\}$
Il est donc identique au dictionnaire d'encodage.

c	c∈D	e_n	w	sortie	Dictionnaire
					$A(1), B(2)$
1	$\in$	A		A	
3	$\notin$	AA	AA	AA	$+AA(3)$
2	$\in$	B	AAB	B	$+AAB(4)$
4	$\in$	AAB	BA	AAB	$+BA(5)$
2	$\in$	B	AABB	B	$+AABB(6)$
7	$\notin$	BB	BB	BB	$+BB(7)$
EOF					

TABLE III.4 – Décodage LZH de 1 3 2 4 2 7 avec $\mathbf{S} = \{A, B\}$.

EXERCICE 29: Compression `LZW`

1. Utiliser `LZW` pour compresser la phrase "si ta tata tate ta tata, ta tata tatee sera".
2. Donner la longueur du code binaire obtenu, et la comparer au texte non compressé.
3. Décoder le code `LZW` suivant : les caractères du dictionnaire sont { , a, d, e, g, n, o, r, u, é } et le code est { `9, 6, 1, 3, 8, 2, 5, 7, 12, 5, 15, 3, 10, 13, 10, 20, 2, 3, 4, 1, 11, 1, 26, 22, 13, 15, 17, 6` }

5.3 Remarques

De nombreux algorithmes de compression utilisent `LZW` comme base. Citons en particulier le `compress` sous Unix, ainsi que le format `GIF`.

Les stratégies d'optimisation de `LZW` sont très similaires à celle de `LZ78` :

- Ajout progressifs de bits pour s'adapter à la taille croissante du dictionnaire,
- Dictionnaire adaptatif jusqu'à $b_{\max}$ bits, puis statique après,
- ...

`LZH` a tendance à fonctionner moins bien que `LZ77`.

- certains motifs du dictionnaire `LZH` ne sont jamais utilisés.
- Le dictionnaire de `LZ77` est implicite. Les t-uples sont codés sur quelques bits.

w_{n-1}	e_n	$w_{n-1}e_n$	$\in D$	w_n	sortie	Dictionnaire
	A		$\in$	A		$A(1), B(2)$
A	B	AB	$\notin$	B	1	$+AB(3)$
B	A	BA	$\notin$	A	2	$+BA(4)$
A	B	AB	$\in$	AB		
AB	A	ABA	$\notin$	A	3	$+ABA(5)$
A	B	AB	$\in$	AB		
AB	A	ABA	$\in$	ABA		
ABA	B	ABAB	$\notin$	B	5	$+ABAB(6)$
B	A	BA	$\in$	BA		
BA	B	BAB	$\notin$	B	4	$+BAB(7)$
B	A	BA	$\in$	BA		
BA	B	BAB	$\in$	BAB		
BAB	A	BABA	$\notin$	A	7	$+BABA(8)$

TABLE III.5 – Codage LZH de ABABABABABABABABABA.

6 Conclusion

Les algorithmes de codage par dictionnaire sont utiles pour éliminer la redondance liée à la présence de motifs dans des données.

En général, cette phase de compression est suivi d'un deuxième codage basé sur code à longueur variable (de type entropique) afin d'améliorer l'efficacité du stockage des codes.

7 Annexe : cycle LZW

Exemple 3 (codage)

Entrée : ABABABABABABABABABA
Alphabet : S = $\{A, B\}$
Voir le résultat du codage à la table III.5.

Exemple 3 (décodage)

Entrée : 1 2 3 5 4 7
Alphabet : S = $\{A, B\}$
Voir le résultat du décodage à la table III.6 ou III.7 pour un décodage utilisant la méthode dit des motifs partiels.

w_{n-1}	e_n	$w_{n-1}e_n$	$\in D$	w_n	sortie	Dictionnaire
	A		$\in$	A		$A(1), B(2)$
A	B	AB	$\notin$	B	1	$+AB(3)$
B	A	BA	$\notin$	A	2	$+BA(4)$
A	B	AB	$\in$	AB		
AB	A	ABA	$\notin$	A	3	$+ABA(5)$
A	B	AB	$\in$	AB		
AB	A	ABA	$\in$	ABA		
ABA	B	ABAB	$\notin$	B	5	$+ABAB(6)$
B	A	BA	$\in$	BA		
BA	B	BAB	$\notin$	B	4	$+BAB(7)$
B	A	BA	$\in$	BA		
BA	B	BAB	$\in$	BAB		
BAB	A	BABA	$\notin$	A	7	$+BABA(8)$

TABLE III.6 – Décodage LZH de 1 2 3 5 4 7 avec $\mathbf{S} = \{A, B\}$.

c	w_n	D upt ?=w_n[0]	D add +(w_n?)	sortie
		A(1),B(2)		
1	A		A ?(3)	A
2	B	AB(3)	B ?(4)	B
3	AB	BA(4)	AB ?(5)	AB
5	AB ? $\xrightarrow{upd}$ ABA	ABA(5)	ABA ?(6)	ABA
4	BA	ABAB(6)	BA ?(7)	BA
7	BA ? $\xrightarrow{upd}$ BAB	BAB(7)	BAB ?(8)	BAB
EOF				

TABLE III.7 – Décodage LZH avec motif partiel de 1 2 3 5 4 7 avec $\mathbf{S} = \{A, B\}$.