3 – Compression de texte

Compression de texte

- optimisation du temps de transit des messages et de l'espace de stockage
- en cryptographie, utile à mélanger l'information et rendre les chiffrements plus robustes
- on distingue la compression avec perte de celle sans perte
- il s'agit de retirer autant de redondance que possible

  à l'inverse du codage pour canal bruité et donc des codes correcteurs

- on distingue respectivement les algorithmes statistiques des algorithmes dynamiques :
  - les codes de Huffman (*cf.* Cours n° 2), le codage arithmétique
  - les méthodes par dictionnaire dérivées de Lempel-Ziv
- s'y ajoutent des méthodes heuristiques plus récentes (*prédictives*)
- on n'abordera pas ici la compression d'images (Gif, Png, Jpeg, *etc*) ou du son (Mp3, *etc*).

Codage arithmétique

- il s'agit d'un codage entropique datant de 1990
- cette méthode statistique utilise un tableau des fréquences d'apparition des symboles
- elle s'avère meilleure que les codes d'Huffman dans la mesure où l'encodage n'a pas lieu en bits entiers
- on encode les caractères par intervalles
- la sortie de l'encodage est un réel dans $[0,1[$
- pour éviter les problèmes de portabilité, il y a moyen de travailler sur des entiers
- d'autres optimisations sont possibles pour manier des entiers les plus petits possibles.

Exemple

- à chaque symbole est associée sa probabilité d'occurence (par défaut ici les fréquences du mot à coder)

symbole	C	E	I	L	N	O
probabilité	0,3	0,2	0,1	0,2	0,1	0,1
intervalle	$[0;0,3[$	$[0,3;0,5[$	$[0,5;0,6[$	$[0,6;0,8[$	$[0,8;0,9[$	$[0,9;1[$

- le mot COCCINELLE sera ainsi codé par le réel r :

symbole	borne inf.	borne sup.
C	0	0,3
O	0,27	0,3
C	0,27	0,279
C	0,27	0,2727
I	0,27135	0,27162
N	0,271566	0,271593
E	0,2715741	0,2715795
L	0,27157734	0,27157842
L	0,271577988	0,271578204
E	0,2715780528	$r = 0,271578096$

Codage arithmétique en Python

- codage

```
proba = {'C':(0,0.3),'E':(0.3,0.5),'I':(0.5,0.6),'L':(0.6,0.8),'N':(0.8,0.9),'O':(0.9,1)}

def codageArithm (chaine, proba) :
    borneInf = 0.0
    borneSup = 1.0
    for c in chaine :
        x = proba[c][0]
        y = proba[c][1]
        taille = borneSup - borneInf
        borneSup = borneInf + taille * y
        borneInf = borneInf + taille * x
    return borneSup
```

- décodage *(idée)*

```
...
for c in proba :
    if proba[c][0] <= r and r < proba[c][1] :
        print(c,end=' ')
        a = proba[c][0]
        b = proba[c][1]
        taille = b - a
        r = (r - a)/taille
...
```

Codes compresseurs adaptatifs

- les codes d'Huffman et le codage arithmétique nécessitent le pré-calcul des fréquences
- il faut donc avoir une connaissance statistique du texte à compresser
- sans connaissance préalable du texte, comment faire ?
- une solution consisterait à faire une première passe sur le texte pour l'obtenir
- une meilleure solution consiste à faire du codage adaptatif

Le codage adaptatif revient à faire de la compression dynamique avec des méthodes jusqu'à présent utilisées comme statiques :

- la commande `pack` d'UNIX utilise le codage de Huffman dynamique
  - la table des fréquences est élaborée au fur et à mesure de la lecture du texte
  - l'arbre de Huffman est donc modifié à chaque caractère lu
  - lors du décodage, le destinataire est à même de recalculer à la volée cette suite d'arbres de codage
- il est également possible de faire du codage arithmétique adaptatif

Si en statique, l'implantation du codage de Huffman est plus rapide que celui arithmétique, c'est l'inverse en adaptatif.

Réduction d'entropie

- ce sont des heuristiques visant à la réduction d'entropie systématique en amont d'un codage statistique
- un codage statistique tire partie des occurences multiples, mais pas de leurs positions d'où l'idée du fonctionnement de RLE *(Run Length Encoding)* :

  aaaaaaaeeddddfccccccccaa

  transformé par exemple en :

  @a7ee@d4f@c9aa

  cette méthode est très utile pour les images en noir et blanc ou des pixels de même couleur, les modems utilisent la variante MNP5 de RLE, les fax aussi

- la méthode de précalcul MOVE-TO-FRONT exploite le fait que le code ASCII est lui-même compressible :

  aaaaaaaeeddddfccccccccaa

  transformé par exemple en :

  00000004040000550000000040

- en 1994, la transformation de Burrows-Wheeler BWT propose de réduire l'entropie d'une chaîne pour optimiser le MOVE-TO-FRONT et le RLE dessus

  à l'aide des permutations circulaires d'une chaîne de caractères, on conserve une version triée de moindre entropie et un index permettra ultérieurement de revenir à la chaîne intiale (cf. TD n° 1)

En pratique

Ces méthodes et leurs variantes sont combinées entre elles pour toujours plus d'efficacité :

- l'algorithme `bzip2`

  BWT → Move-to-front → RLE → Huffman

- le format JPEG

  image → Blocs 8×8 → DCT → Quantification → RLE zig-zag → Huffman ou Arithm.

  (DCT est la transformée en cosinus discrète, la *quantification* consiste à garder ce qui est le plus perceptible).

Algorithmes dynamiques

- pour l'instant, les méthodes vues reposaient à l'origine sur une analyse statistique
- avant de devenir *adaptatives*, elles ne permettaient pas le codage dynamique ou *à la volée* sans calcul de fréquences
- cela dit, les premiers algorithmes adaptatifs préfigurent des algorithmes dynamiques
- en 1977, Lempel et Ziv proposent des méthodes qui relancent la recherche en compression
- ils proposent de la compression par dictionnaire
- l'idée est de faire de la réduction d'entropie, non plus par caractères mais par mots entiers.

LZ77

- LZ77 est le premier algorithme de compression par dictionnaire
- il consiste à remplacer des facteurs d'un texte par des codes courts qui sont leurs indices dans le dictionnaire construit dynamiquement
- une seule lecture du fichier à compresser est donc requise
- une des implantations les plus connues est LZMA

  le Lempel-Ziv-Markov chain-Algorithm combine LZ77 avec un dictionnaire adaptatif suivi d'un codage arithmétique entier

- utilisé dans `gzip` d'Unix, `pkzip`, `compress`
- il apparaît aussi dans les formats Gif, Png, Mpeg *etc*

De nombreuses variantes existent du fait de la recherche du compromis entre rapidité et qualité de compression :

- LZ78
- LZW (Lempel-Ziv-Welch)
- LZH

Principe de LZ77

- l'algorithme fait coulisser sur le texte 2 fenêtres consécutives : la fenêtre de recherche et la fenêtre de lecture

  à gauche : la fenêtre de recherche tient lieu de dictionnaire courant des facteurs lus et dernièrement compressés
  à droite : la fenêtre de lecture contient les prochains symboles à compresser

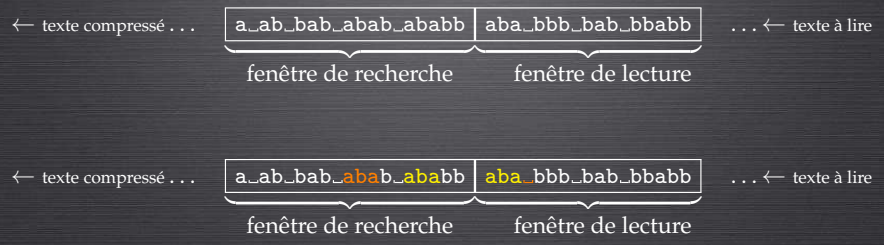
- il parcourt l'entrée à compresser de la gauche vers la droite en recherchant le motif à la fois le plus long préfixe de la fenêtre de lecture et un facteur de la fenêtre de recherche

- pour tout motif non vide trouvé, il encode un triplet

  1. un pointeur permettant de retrouver dynamiquement le motif dans la fenêtre de recherche
  2. la longueur dudit motif
  3. le caractère suivant le motif dans la fenêtre de lecture

(*cf.* TP n° 3)

LZ77 : les 2 fenêtres



LZ77 : EXEMPLE

CODAGE :	fenêtre RECHERCHE	f. LECTURE	TRIPLET
		place_d	(0,0,p)
p	p	lace_de	(0,0,l)
pl	pl	ace_des	(0,0,a)
pla	pla	ce_des_	(0,0,c)
plac	plac	e_des_m	(0,0,e)
place	place	_des_mi	(0,0,_)
place_	place_	des_mil	(0,0,d)
place_d	place_d	es_mill	(3,1,s)
place_des	place_des	_mille_	(4,1,m)
place_des_m	place_des_m	ille_ve	(0,0,i)
place_des_mi	place_des_mi	lle_ven	(11,1,l)
place_des_mill	place_des_mill	e_vents	(10,2,v)
place_des_mille_v	place_des_mille_v	ents_le	(13,1,n)
place_des_mille_ven	place_des_mille_ven	ts_les_	(0,0,t)
place_des_mille_vent	place_des_mille_vent	s_les_j	(12,2,l)
place_des_mille_vents_l	place_des_mille_vents_l	es_joue	(16,3,j)
place_des_mille_vents_les_j	place_des_mille_vents_les_j	oueurs_	(0,0,o)
place_des_mille_vents_les_jo	place_des_mille_vents_les_jo	ueurs_c	(0,0,u)
place_des_mille_vents_les_jou	place_des_mille_vents_les_jou	eurs_co	(25,1,u)
place_des_mille_vents_les_joueu	place_des_mille_vents_les_joueu	rs_couv	(0,0,r)
lace_des_mille_vents_les_joueur	lace_des_mille_vents_les_joueur	s_couve	(24,2,c)
e_des_mille_vents_les_joueurs_c	e_des_mille_vents_les_joueurs_c	ouverts	(8,2,v)
es_mille_vents_les_joueurs_couv	es_mille_vents_les_joueurs_couv	erts_de	(31,1,r)
_mille_vents_les_joueurs_couver	_mille_vents_les_joueurs_couver	ts_de_g	(21,3,d)
le_vents_les_joueurs_couverts_d	le_vents_les_joueurs_couverts_d	e_givre	(30,2,g)
vents_les_joueurs_couverts_de_g	vents_les_joueurs_couverts_de_g	ivre_so	(0,0,i)
ents_les_joueurs_couverts_de_gi	ents_les_joueurs_couverts_de_gi	vre_son	(11,1,r)
ts_les_joueurs_couverts_de_givr	ts_les_joueurs_couverts_de_givr	e_sont_	(6,2,s)
les_joueurs_couverts_de_givre_s	les_joueurs_couverts_de_givre_s	ont_par	(26,1,n)
⋮	⋮	⋮	⋮

[in : La joueuse de go, Shan Sa]

LZ77 : EXEMPLE

CODAGE :	fenêtre RECHERCHE	f. LECTURE	TRIPLET	DECODAGE
		place_d	(0,0,p)	p
p	p	lace_de	(0,0,l)	l
pl	pl	ace_des	(0,0,a)	a
pla	pla	ce_des_	(0,0,c)	c
plac	plac	e_des_m	(0,0,e)	e
place	place	_des_mi	(0,0,_)	_
place_	place_	des_mil	(0,0,d)	d
place_d	place_d	es_mill	(3,1,s)	es
place_des	place_des	_mille_	(4,1,m)	_m
place_des_m	place_des_m	ille_ve	(0,0,i)	i
place_des_mi	place_des_mi	lle_ven	(11,1,l)	ll
place_des_mill	place_des_mill	e_vents	(10,2,v)	e_v
place_des_mille_v	place_des_mille_v	ents_le	(13,1,n)	en
place_des_mille_ven	place_des_mille_ven	ts_les_	(0,0,t)	t
place_des_mille_vent	place_des_mille_vent	s_les_j	(12,2,l)	s_l
place_des_mille_vents_l	place_des_mille_vents_l	es_joue	(16,3,j)	es_j
place_des_mille_vents_les_j	place_des_mille_vents_les_j	oueurs_	(0,0,o)	o
place_des_mille_vents_les_jo	place_des_mille_vents_les_jo	ueurs_c	(0,0,u)	u
place_des_mille_vents_les_jou	place_des_mille_vents_les_jou	eurs_co	(25,1,u)	eu
place_des_mille_vents_les_joueu	place_des_mille_vents_les_joueu	rs_couv	(0,0,r)	r
lace_des_mille_vents_les_joueur	lace_des_mille_vents_les_joueur	s_couve	(24,2,c)	s_c
e_des_mille_vents_les_joueurs_c	e_des_mille_vents_les_joueurs_c	ouverts	(8,2,v)	ouv
es_mille_vents_les_joueurs_couv	es_mille_vents_les_joueurs_couv	erts_de	(31,1,r)	er
_mille_vents_les_joueurs_couver	_mille_vents_les_joueurs_couver	ts_de_g	(21,3,d)	ts_d
le_vents_les_joueurs_couverts_d	le_vents_les_joueurs_couverts_d	e_givre	(30,2,g)	e_g
vents_les_joueurs_couverts_de_g	vents_les_joueurs_couverts_de_g	ivre_so	(0,0,i)	i
ents_les_joueurs_couverts_de_gi	ents_les_joueurs_couverts_de_gi	vre_son	(11,1,r)	vr
ts_les_joueurs_couverts_de_givr	ts_les_joueurs_couverts_de_givr	e_sont_	(6,2,s)	e_s
les_joueurs_couverts_de_givre_s	les_joueurs_couverts_de_givre_s	ont_par	(26,1,n)	on
⋮	⋮	⋮	⋮	⋮

[in : La joueuse de go, Shan Sa]

VARIANTE LZ78

- LZ78 fonctionne un peu comme LZ77 mais le dictionnaire n'est plus composé d'une fenêtre coulissante mais constitué de l'intégralité du texte déjà traité (taille des entrées bornée en pratique)
- au départ, on ne connaît aucun facteur : le but est de mémoriser dans le dictionnaire la totalité des facteurs rencontrés en les indexant
- on recherche le motif m qui est le plus long préfixe du texte à compresser qui est déjà dans le dictionnaire à l'index i
- on encode le couple (i, c), c étant le caractère suivant
- on ajoute ensuite l'entrée mc au dictionnaire
- en cas de caractère inconnu c, on encode le couple $(0, c)$.

LZ78 : EXEMPLE

Soit la chaîne *aabbababababbbabbbabb* à compresser :

dictionnaire		couple
0	`null`	
1	a	$(0, a)$
2	ab	$(1, b)$
3	b	$(0, b)$
4	aba	$(2, a)$
5	ba	$(3, a)$
6	bb	$(3, b)$
7	bba	$(6, a)$
8	bbb	$(6, b)$
9	abb	$(2, b)$

LZ78 : DÉCOMPRESSION

Soit la liste de couples à décompresser :

couple	dictionnaire		décompression
	0	`null`	
$(0,a)$	1	a	a
$(1,b)$	2	ab	ab
$(0,b)$	3	b	b
$(2,a)$	4	aba	aba
$(3,a)$	5	ba	ba
$(3,b)$	6	bb	bb
$(6,a)$	7	bba	bba
$(6,b)$	8	bbb	bbb
$(2,b)$	9	abb	abb

On retrouve bien la chaîne initiale : *aabbababababbbbabbbabb*.

VARIANTE LZW

- en 1984, amélioration de LZ78 par T. Welch
- utilisé dans GIF et TIFF

```
for i in range(0,128) :
    dico[chr(i)] = hex(i)

def LZW(texte,dico):
     w = ''
     lt = len(texte)
     i = 0
     while i < lt :
          c = texte[i]
          p = w + c
          if p in dico :
               w = p
          else :
               ajouter(p,dico)
               print(dico[w])
               w = c
          i++
```

A suivre ...