Algorithmique Avancée

Michèle Soria

Compression de données

Bibliographie

M.Nelson *La Compression de données : texte, images, sons*, Dunod
D. Salomon *Data Compression: The Complete Reference*, Springer
M. Crochemore, C. Hancart, T. Lecroq *Algorithmique du texte*, Vuibert

21 octobre 2009

Généralités-1

- Compression = Codage + Modélisation (statistique ou dictionnaire)
- Abréger les redondances, réduire la taille
- Motivations : Archivage et Transmission
- *Algorithmes* pour réduire la *place* occupée, sans perdre trop de temps
- Compression conservative (sans perte d'information)
  - fonction de compression réversible (injective)

  → compression de textes, de données scientifiques, de code compilé de programmes

$$T \xrightarrow{Compression} C(T) \xrightarrow{Decompression} T$$

$$T \xrightarrow{Compression} C(T) \xRightarrow{Transmission} C(T) \xrightarrow{Decompression} T$$

Généralités-2

Compression non conservative

- transformation (inversible) des données
- quantification (perte d'information)
- compression conservative

$$I \xrightarrow{Transf.} \tilde{I} \xrightarrow{Perte} \hat{I} \xrightarrow{Compression} C(\hat{I}) \xrightarrow{Decompression} \hat{I} \xrightarrow{Transf.Inv.} I'$$

Applications : approximation acceptable

- compression de données analogiques (images, son)
- transmission(télécopie), archivage (doubleur de disque)

Entropie – Quantité d'information

ASCII 7bits (128 car) – ASCII-étendu 8bits (256 car) :
$\lfloor log_2 n \rfloor$ bits pour coder n symboles différents.

Théorie de l'information : S source produit n symboles différents $a_1, ..., a_n$ avec probabilité $p_1, ..., p_n$, avec $\sum p_i = 1$.

- *Quantité d'information QI* ou *entropie* du symbole a_i = Nombre de bits pour coder a_i. Si a_i a probabilité p_i, alors QI = $log_2(1/p_i)$ (aléatoires → QI = $\lfloor log_2 n \rfloor$)
- *entropie* de S = valeur moyenne de la quantité d'information des symboles : $H = -\sum_{i=1}^{n} p_i log_2 p_i$. Entropie max quand tous symboles équiprobables (source aléatoire). Plus entropie grande, plus grande est information transmise.

Théorèmes de Shannon : source S émet messages longueur N.

- le codage d'un message de longueur N est toujours $> HN$.
- qd $N \to \infty$, il existe un codage dont la longueur $\to HN$.

Exemple

Source émet 2 symboles a et b, avec $p_a = 0,8$ et $p_b = 0,2$.
Entropie $H = -(p_a \log p_a + p_b \log p_b) \sim 0,7219$.

Comment coder au plus court un message de taille N ?

• $a \to 0$ et $b \to 1$: le code a même taille que le message

• grouper par 2 : $aa \to 0$, $ab \to 11$, $ba \to 100$, $bb \to 101$
taille moyenne du code $0,75N$:
$(1 * 0,64 + 3 * 0,16 + 3 * 0,02 + 2 * 0,16) * N/2$

• grouper par 3 : $aaa \to 0$, $aab \to 100$, $aba \to 101$, $baa \to 110$, $abb \to 11100$, $bab \to 11101$, $bba \to 11110$, $bbb \to 11111$
taille moyenne du code $0,728N$:
$(1 * 0,64 + 3 * 3 * 0,128 + 3 * 5 * 0,032 + 1 * 5 * 0.008) * N/3$

Représentation par arbre digital, décodage de 011011101100011110

Remarque : (Source, Probabilités) $\sim$ (Texte, Fréquences)
T texte de N lettres sur un alphabet $A = a_1, ..., a_n$.
f_i, fréquence de a_i dans $T = \#a_i$ dans T/N.

Compression conservative

T texte sur un alphabet A, et codage $C : T \to C(T)$

- Encodage des répétitions
- Méthodes statistiques – $C : A \to \{0,1\}^*$
- Codage arithmétique – C : *Texte* $\to$ *Reel*
- Méthodes avec dictionnaire – $C : A^* \to \{0,1\}^*$

Souvent les progiciels utilisent plusieurs méthodes à la fois.

Encodage des répétitions

- **Principe** : remplacer suite de n symboles $a...a$ par $a\#n$
- **Algorithmes** : Run Length Encoding (RLE) et variantes
- **formats bitmap** : n&b, BMP(format natif Windows)

Méthodes statistiques

- **Principe** : symboles fréquents codés sur petit nombre de bits
- **Algorithmes** : Shannon–Fano (48), Huffman (50), Huffmann dynamique (Gallager 78) : algorithme adaptatif
- **Programmes** : CCITT–fax (H), COMPACT Unix-(HD)

Codage arithmétique : le texte est codé par un nombre réel, L'algorithme produit un code pour un texte entier –et non pour un symbole–
Meilleur que les autres en gain de place, mais peu rapide.

Méthodes avec dictionnaire

- **Principe** : code un groupe de symboles par son index dans un dictionnaire – statique ou adaptatif–
- **Algorithmes** : Lempel-Ziv77, Lempel-Ziv78, Lempel-Ziv-Welch
- **Programmes** :
  - **compression de fichiers** COMPRESS Unix+ (LZW)
  - **formats d'image** GIF (graphic interchange format) et TIFF (Tagged image file format)
  - **archivage** LHarc (LZ77+HD)– free, PKZIP , ARC, ARJ – (MS-DOS)

Compression Statistique

1. Algorithme de Huffmann statique
   - Principes et Définitions
   - Construction de l'arbre de Huffman
   - Optimalité du code
   - Compression : algorithme et complexité
   - Déompression : algorithme et complexité
2. Huffmann Adaptatif
   - Principes et Définitions
   - Modification de l'arbre de Huffman adaptatif
   - Compresseur et décompresseur

Compression statistique – Huffman

Symbole fréquent == code court,
Symbole rare == code long
Au mieux : coder un symbole avec le nombre de bits d'information qu'il contient $log_2(1/p(i))$, (mais ici nombre de bits *entier*).
Connaître les fréquences

- Table de probabilité universelle (d'ordre 0, 1, ...)
- Table de probabilité statique pour chaque texte à compresser : meilleure compression mais surcoût (calcul des fréquences + transmission de la table) $\longrightarrow$ Algorithme de Huffmann
- Fréquences calculées au fur et à mesure (codage d'un symbole évolue) $\longrightarrow$ Algorithme de Huffmann adaptatif
  Adaptativité inversible (fonction de décompression)

Code préfixe

Huffman : code préfixe de longueur variable.

Alphabet $A = \{a_1, \ldots, a_n\}$. Codage $C : A \to \{0,1\}^+$.
P ensemble de mots de $\{0,1\}^+$, codages des caractères de A.

Définition : *P code préfixe* $\stackrel{Def}{=}$ aucun mot de P n'est préfixe propre d'un mot de P : $\forall w_1 \in P,\ w_1 w_2 \in P \to w_2 = \epsilon$.

Proposition Tout code préfixe est uniquement décodable
Aucun code d'un caractère n'est préfixe propre d'un code d'un autre caractère : $C(xy) = C(x)C(y)$.

Arbre digital

Définition : *arbre digital (trie)* $\stackrel{Def}{=}$ arbre binaire t.q. chaque feuille contient le codage de son chemin à partir de la racine (0 pour lien gauche, 1 pour droit).

Définition : P est un code préfixe ssi il est constitué des codes des feuilles d'un arbre digital.

Définition : Arbre digital *complet* $\stackrel{Def}{=}$ Code préfixe *complet*

Remarques : Pour tout $a \in A$, $C(a)$ est un mot sur $\{0,1\}^+$, de longueur $L(a)$. Et $L(a)$ est égal à la profondeur de la feuille codant a dans l'arbre digital.

Code minimal

Texte $T \in A^*$. Codage $C : T \to T(C) = T_C$.

Définitions

Fréquence $f(a_i) = \#a_i$ dans T.
Taille de $T(C)$: $||T_C|| = \sum f(a_i)|C(a_i)|$, $a_i \in A$.
C code minimal pour T ssi $||T_C|| = min\{||T_\gamma||, \gamma : A \to \{0,1\}^+\}$

Proposition

C minimal pour $T \Longrightarrow < f(a_{i_1}) \leq f(a_{i_2}) \leq \ldots \leq f(a_{i_n})$ et
$|C(a_{i_1})| \geq |C(a_{i_2})| \geq \ldots \geq |C(a_{i_n})|$

Preuve : s'il existe a et b t.q. $f(a) < f(b)$ et $C(a) < C(b)$, alors soit C' t.q. $C(x) = C'(x)$ pour tout $x \neq a, b$, et $C'(a) = C(b)$, et $C'(b) = C(a)$. $||T_C|| - ||T_{C'}|| = f(a)C(a) + f(b)C(b) - (f(a)C(b) + f(b)C(a)) = (f(a) - f(b)) * (C(a) - C(b)) > 0$
Contradiction avec C minimal pour T.

Mais la réciproque n'est pas vraie.

Construction de l'arbre de Huffman

Exemple : $T = abracadabra$,
$f(a) = 5, f(b) = 2, f(r) = 2, f(c) = 1, f(d) = 1$
Construction d'un arbre digital

- Peigne par fréquences croissantes,
  ex. 1, 1, 2, 2, 5 $||T_C|| = 23$
- Utiliser les branchements
  ex. 1, 1, 2, 2, 3 $||T_C|| = 20$ (au lieu de 21 pour peigne par fréquences croissantes : ainsi il ne suffit pas de classer par fréquences.)

Principe de construction d'un arbre de Huffman

- Unir 2 sous-arbres de "poids minimal"
- Deux degrés de liberté : choix des 2 sous-arbres + Gauche/Droite

Exemple : plusieurs arbres de Huffman pour le texte
$T = abracadabra$

Arbre de Huffman : Algorithme

ENTREE : Fréquences $(f_1, \ldots, f_n)$ des lettres (a_i) d'un texte T.
SORTIE : Arbre digital donnant un code préfixe minimal pour T.

1. Créer, pour chaque lettre a_i, un arbre (réduit à une feuille) qui porte comme poids la fréquence $f(i)$
2. Itérer le processus suivant
   - choisir 2 arbres G et D de poids minimum
   - créer un nouvel arbre R, ayant pour sous-arbre gauche (resp. droit) G (resp. D) et lui affecter comme poids la somme des poids de G et D,
3. Arrêter lorsqu'il ne reste plus qu'un seul arbre : c'est un arbre de Huffman *(pas unique)*

Structuration des données

Création arbre de Huffman contrôlée par une *file de priorité* TAS
– éléments = arbres,
– clés = poids attachés aux arbres (fréquences des lettres pour les feuilles et poids cumulés pour les arbres construits).

*Huffman : Alphabet*Fréquences→ ArbreHuffman*
renvoie *l'arbre de Huffman associé ;* $f_i \neq 0$ *fréquence de* a_i *dans T*
Fonction Huffman(A,f1,...,fn)
FP=construireTas((f1,a1),...,(fn,an))
Pour i de 1 à n-1
extraireMinTas rend l'arbre min, et le tas réorganisé
(G,FP1)=extraireMinTas(FP)
(D,FP2) =extraireMinTas(FP1)
R=ABValué(G,D,f(G)+f(D))
FP=ajouterTas(R,FP2)
Fin Pour
Retourne extraireMin(FP)
Fin Fonction

Définition récursive de la construction de l'arbre

*Huffman : Alphabet*Fréquences→ ArbreHuffman*
Fonction Huffman(A,f1,...,fn)
*construireTas : Alphabet*Fréquences→ TAS[ArbreHuffman]*
  Retourne ArbreH(construireTas((f1,a1),...,(fn,an))
Fin Fonction

ArbreH : TAS[ArbreHuffman]→ ArbreHuffman
Fonction ArbreH(FP)
  Si FP contient 1 seul arbre **Retourne** cet arbre
  Sinon
    Soient (G, FP1)=extraireMinTas(FP)
          (D, FP2) =extraireMinTas(FP1)
    Retourne
ArbreH(ajouterTas(ABValué(G,D,f(G)+f(D)),FP2))
  Fin Si
Fin Fonction

Complexité de la construction

1. Construire le tas initial : ($O(n)$) (ou $O(n \log n)$ en ligne)
2. A chaque itération
   - extraire 2 fois l'arbre de poids min et réorganiser le tas : $O(\log n)$
   - fabriquer un nouvel arbre à partir des 2 précédents : $O(1)$
   - rajouter ce nouvel arbre au tas : $O(\log n)$
3. $n-1$ itérations

La construction de l'arbre de Huffman résultant est donc en $O(n \log n)$ comparaisons.

Taille du texte compressé : $||T_h|| = W(H) = \sum_1^n f(i)\,\text{prof}_h(i)$

Preuve de l'optimalité du code de Huffmann

Le code préfixe complet d'un arbre de Huffman est minimal i.e. meilleur que celui de tout autre arbre digital de $A \to \{0,1\}^+$

Preuve : Par récurrence sur le nombre n de lettres de A
– pour $n = 2$, chaque lettre codée par 1 bit donc $||T_h||$ =# caractères de T
– pour $n > 2$, l'algo choisit x et y de poids min, construit un nœud z de poids $f(z) = f(x) + f(y)$ et applique récursivement la construction à $\tilde{T}$ sur $A - \{x, y\} \cup \{z\}$ ($n-1$ lettres).
On suppose que l'arbre $\tilde{H}$ construit pour $\tilde{T}$ produit un code minimal (hypothèse de récurrence), et on montre qu'il en est de même pour l'arbre H construit pour T. Par construction : $W(H) = W(\tilde{H}) - f(z)p(z) + f(x)p(x) + f(y)p(y) = W(\tilde{H}) + f(x) + f(y)$
Démonstration par l'absurde : on suppose qu'il existe H' avec $W(H') < W(H)$, et l'on montre qu'il existe alors $\tilde{H}'$ tel que $W(\tilde{H}') < W(\tilde{H})$, contradiction avec $\tilde{H}$ minimal.
  – Si x et y ont même père ds H', alors $W(H') = W(\tilde{H}') + f(x) + f(y)$, et $W(H') < W(H) \Rightarrow W(\tilde{H}') < W(\tilde{H})$.
  – Sinon on construit un arbre H'' où x et y ont le même père, avec $W(H'') < W(H')$, et on est ramené au cas précédent.

Construction de H'' : soient b et c 2 feuilles soeurs de H', à prof. max., i.e.
$p(b) = p(c) \geq p(f), \forall f$ feuille de H'. H' est minimal donc $f(x) \leq f(b)$ et $f(y) \leq f(c)$.
On construit H'' en echangeant (x et b), et (y et c). On a donc finalement
$W(H'') = W(H') - [(f(b) - f(x)).(p(b) - p(x))] - [(f(c) - f(y)).(p(c) - p(y))] < W(H')$ •

Algorithme de compression

ENTREE : Texte T
SORTIE : Texte compressé TC et codage de l'arbre de Huffman

1. Parcourir T pour compter le nombre d'occurrences de chaque caractère
2. Construire l'arbre de Huffman à partir des fréquences des caractères
3. Calculer la table des codes à partir de l'arbre de Huffman
4. Compresser T en concaténant les codes de chaque caractère → TC
5. Coder l'arbre de Huffman de façon compacte

Analyse

Texte T : N caractères sur alphabet de n lettres,
Codage initial d'un caractère sur k bits

1. Parcours de T : $O(N)$. Stockage des fréquences : $O(n)$,
2. Construction de l'arbre H en $O(n \log n)$,
3. Construction table des codes : $2n-1 = O(n)$. Stockage : $kn + \sum L_i$ bits,
4. Parcours de T : $O(N)$. Taille de TC : $\sum f_i L_i$,
5. Codage de H : Temps $O(n)$. Taille de l'arbre codé $2n-1+kn$ bits.

Codage de Dyck

Table des codes $\Longleftrightarrow$ Tous les chemins de l'arbre de Huffman
Mieux : coder un arbre binaire complet (n feuilles et $n-1$ nœuds internes) un mot de Dyck (Lukasiewicz) de longueur $2n-1$.

Parcours préfixe

- nœud interne $\longrightarrow$ 0
- feuille $\longrightarrow$ 1
- De plus chaque 1 est suivi de k bits (code initial du caractère)

( ) - ( ) - (( ) -) -) -

Exemple : 01a01b001d1c1r
Décodage : reconstruire l'arbre : 0 = nœuds internes et 1= feuille, et après chaque 1 lire k bits = codage initial d'un caractère

Algorithme de décompression

ENTREE : Texte compressé TC et codage de l'arbre de Huffman
SORTIE : Texte décompressé T

1. Reconstruire l'arbre de Huffman ($O(n)$)
2. Parcourir TC en suivant l'arbre de Huffman ($O(||TC||)$)
   - suite de bits : de la racine à une feuille de l'arbre
   - produit un caractère de T
   - et l'on poursuit le parcours de TC en recommençant à la racine de l'arbre

Statique $\longrightarrow$ Dynamique

Inconvénients de la compression de Huffman statique

- double parcours du texte (fréquences + compression)
- mémorisation du codage (arbre de Huffman)

Version dynamique – adaptative

- l'arbre de Huffman évolue au fur et à mesure de la lecture du texte et du traitement (compression ou décompression) des caractères.
- l'algorithme de compression (*compresseur*) et l'algorithme de décompression (*décompresseur*) modifient l'arbre de la même façon
- à un instant donné du traitement les 2 algorithmes utilisent les mêmes codes – mais ces codes changent au cours du temps.

Principe Huffman dynamique

Compresseur Connaît les codes fixes (k bits) des caractères,

- Initialement, toutes fréquences nulles, et arbreH = #
- A la lecture de x dans T,
  - si prem. occ., retourne *code# dans H + code fixe*, et ajoute x à H
  - sinon retourne *code (variable) de x dans H*

  augmente de 1 la fréquence de x et modifie éventuellement H

Décompresseur Connaît les codes fixes (k bits) des caractères

- Initialement lit k bits, et ajoute caractère à arbre "vide" (H =#)
- Puis décode les bits du texte compressé sur $H \rightarrow$ feuille
  - si feuille=#, lit les k bits suivants de TC et ajoute le x à H
  - sinon retourne le caractère x correspondant à la feuille

  augmente de 1 la fréquence de x et modifie éventuellement l'arbre de Huffman de la même manière que le compresseur.

Arbre de Huffman adaptatif

Définition Numérotation hiérarchique GDBH GaucheDroiteBasHaut

Définition Un *arbre de Huffman adaptatif (AHA)* est un arbre de Huffman tel que le parcours hiérarchique GDBH $((x_1, \ldots, x_{2n-1})$ donne la suite des poids en ordre croissant $W(x_1) \leq \ldots \leq W(x_{2n-1})$

Propriété P : *Soit H un AHA et φ une feuille, de numéro hiérarchique x_{i_0}, dont le chemin à la racine est $\Gamma_\varphi = x_{i_0}, x_{i_1}, \ldots, x_{i_k}$ ($i_k = 2n-1$). Les noeuds du chemin Γ_φ sont dits incrémentables ssi $W(x_{i_j}) < W(x_{i_j+1})$, pour $0 \leq j \leq k-1$.*
N.B. le sommet x_{i_j+1} se situe après x_{i_j} dans le parcours GDBH.

Proposition : *soit H un AHA, φ une feuille de H et Γ_φ son chemin à la racine. Si Γ_φ vérifie la **propriété P**, alors l'arbre résultant de l'incrémentation de φ est encore un AHA.*

Principe de modification

Modification après lecture d'un caractère de T correspondant à la feuille φ, de chemin Γ_φ à la racine de l'arbre de Huffman

- si Γ_φ vérifie **P**, incrémenter les poids sur ce chemin, sans modifier l'arbre,
- sinon transformer l'arbre pour qu'il vérifie **P**, puis incrémenter les poids sur le (nouveau) chemin de φ à la racine.

Transformation de l'arbre

- soit m le premier sommet de Γ_φ qui ne vérifie pas **P**, et soit f tel que $W(x_m) = W(x_{m+1}), \ldots = W(x_f)$ et $W(x_f) < W(x_{f+1})$ (f est en *fin de bloc* de m : on dispose d'une fonction `finBloc(H,m)` qui renvoie le nœud `f`
- échanger sous-arbres $A1$ et $A2$ de racines numérotées x_m et x_l
  N.B. On n'échange jamais un nœud avec un de ses ancêtres
- recommencer le même traitement sur le nouveau chemin de la racine de $A1$ à la racine de l'arbre de Huffman

Algorithme de modification

Feuille spéciale # de fréquence 0, et dont la position varie au cours du temps

*Modification : AHA * Symbole $\rightarrow$ AHA*
Fonction `Modification` (H, s)
. **Si** s n'est pas dans H
. **Soit** Q le père de la feuille spéciale # dans H
. Remplacer # par un noeud interne de poids 1, dont
. le fils gauche est la feuille "#"
. le fils droit est une feuille "s", de poids 1
. **Sinon**
. **Soit** Q la feuille correspondant à s dans H
. **Si** Q est frère de # et `pere`(Q)=`finBloc`(H,Q),
. **Alors** Q=`pere`(Q) **Fin Si**
. **Fin Si**
. **Retourne** `Traitement`(H,Q)
Fin Fonction `Modification`

Traitement

*Traitement : AHA * noeud → AHA*
Fonction Traitement (H,Q)
Soit $Q, Q_{i1}, ..., Q_{ik} = \Gamma_Q$ le chemin de Q à la racine
Soit $x_{i0}, x_{i1}, ..., x_{ik}$ la suite des numéros des nœuds de Γ_Q
Si tous les nœuds de Γ_Q sont incrémentables
ajouter 1 à tous les poids sur le chemin Γ_Q
Retourne H
Sinon
Soit m le premier indice de Γ_Q t.q. $W(x_m) = W(x_{m+1})$
Soit $b =$ `finBloc(H,m)`
ajouter 1 à tous les poids du chemin de Q à Q_m
changer dans H les sous-arbres enracinés en Q_m et Q_b
Retourne `Traitement(H,pere(`Q_m`))`
Fin Fonction `Traitement`

Complexité $O(n)$ (hauteur maximale de H)

Algorithme de compression

ENTREE : Texte T
SORTIE : Texte compressé TC

1. H=feuille spéciale #
2. soit s le symbole suivant de T
   - Si $s \in H$, alors transmettre le code de s dans H
     Sinon transmettre le code de # dans H puis le code initial de s
   - Modifier H : `H=Modification (H, s)`
3. Recommencer l'étape 2 tant que T n'est pas vide

Complexité $O(nN)$

Implantation en TME

Algorithme de décompression

ENTREE : Texte compressé TC (et connaît codes fixes des caractères)
SORTIE : Texte décompressé T

1. H= #, décoder k bits → s, et `H=Modification(H,s)`
2. Lire les bits de TC en suivant H à partir de sa racine → feuille
   - Si c'est la feuille # alors lire les k bits suivant → s
     Sinon → caractère s de la feuille
   - `H=Modification(H,s)`
3. Recommencer l'étape 2 tant que TC n'est pas vide

Complexité $O(nN)$

Implantation en TME

Conclusions Huffmann

- Huffman Dynamique ne calcule pas les fréquences
- Huffman Dynamique "à la volée" : décompression peut commencer avant que compression terminée
- Codage Huffman Statique minimal pour $T_{(f_i)}$, mais doit transmettre l'arbre ⇒ augmente taille de l'encodage
- Résultats expérimentaux sur données homogènes : Huffman Dynamique légèrement meilleur que Huffman Statique pour petits fichiers, et bien meilleur pour gros fichiers. (A tester en TME !)

Méthodes de Compression par Dictionnaire

1. Principes de la compression par dictionnaire
2. Dictionnaire Statique
3. Dictionnaire adaptatif
   - Lempel-Ziv LZ77-LZ78
   - Lempel-Ziv-Welch LZW : phase stationnaire et phase adaptative

- **compression de fichiers** COMPRESS Unix+ (LZW)
- **formats d'image** GIF (graphic interchange format) et TIFF (Tagged image file format)
- **archivage** LHarc (LZ77+HD)– free, PKZIP , ARC, ARJ – (MS-DOS)

Compression avec dictionnaire

Principe : groupe de symboles codé par index dans dictionnaire
$C : A^* \rightarrow \{0,1\}^*$

1. Dictionnaire statique
   Dictionnaire de référence, partagé par compresseur et décompresseur

2. Dictionnaire adaptatif
   Le dictionnaire est "défini" par le fichier à compresser : apprentissage dynamique des répétitions
   Exemple : Acronymes introduits lors de la première occurrence (CCC)
   Adaptativité inversible (fonction de décompression)

   J. Ziv and A. Lempel *"A universal algorithm for data compression"*
   IEEE Transactions on Information Theory May 1977

Dictionnaire statique

1. Codes postaux (corpus particulier – méthodes ad hoc)
2. Dictionnaire de référence : Petit Robert 1993
   Pages $< 2^{12}$, 2 Colonnes, Mots par colonnes $< 2^6$
   Algorithmes et scoubidous $\rightarrow$ 56/2/13//822/2/5//2055/1/5
   25 caractères (200 bits) compressés en $3*(12+1+6) = 57$ bits
3. Dictionnaire par taille de mot : en français taille des mots entre 1 et 25
   On connaît distribution des mots selon leur taille (9000 mots de 6 lettres) 9000 à comparer aux $26^6 > 100$ millions ($\sim 2^{30}$) de possibilités
   $\rightarrow$ codage sur 14 bits par mot (+ 5 bits pour taille) au lieu de 30 bits

Limitations et inconvénients : dictionnaire figé ; rendre compte des conjugaisons diverses, exceptions . . . .

Dictionnaire adaptatif LZ77-LZSS

LZ77-LZSS : Fenêtre fixe ou coulissante sur le fichier
Ex : AIDE-TOI-LE-CIEL-T-AIDERA 25car sur 8 bits = 200 bits
1A1I1D1E1-1T1O1I1-1L0(3,2)1C1I1E1L0(4,2)1-0(0,4)1R1A
20 bits + 17car sur 8 bits + 3couples(*pos*, *long*) sur 5+2 bits = 177 bits
- fenêtre de taille $2^k \Rightarrow k$ bits pour *pos* ;
- *long* sur n bits $\Rightarrow$ on peut compresser séquences de 2^n car.

Difficultés :

- gestion et représentation de la fenêtre coulissante
- compression : rechercher, en toutes positions, la plus longue séquence de la fenêtre pour coder la suite du texte. Décompression pas de pb !
- complexité : $tailleFen * longMax$ si fenêtre structurée séquentiellement
- complexité : $\log_2(tailleFen) * longMax$ si fenêtre structurée en arbre
  On peut alors doubler taille fenêtre sans augmenter trop le temps de recherche.

LZ78- LZW

LZ78- LZW : Dictionnaire = Ensemble (Hachage-Arbre digital) de toutes les séquences déja rencontrées (potentiellement illimité).
Méthodes adaptatives : phase transitoire (gagne en compression par l'apprentissage) puis stationnaire (ne sert à rien de continuer à adapter).
Algorithme en 2 phases

1. Phase adaptative : construit un dictionnaire tout en commençant le codage
2. Phase stationnaire : le dictionnaire n'évolue plus, on l'utilise pour poursuivre le codage

Compresseur et Décompresseur construisent le même dictionnaire

Dictionnaire = table associative,
La fonction de hachage $h : A^+ \rightarrow [0, \ldots, m-1]$ *est connue par le compresseur et le décompresseur.*

Phase stationnaire

Définition : Un ensemble F de mots est dit *préfixiel* ssi lorsque $f \in F$, alors tous les préfixes de f sont dans F
Le dictionnaire de LZW est un ensemble préfixiel F de séquences du texte T à coder. Chaque $f \in F$ est codé par son index (numéro) dans la table.

En phase stationnaire (Le dictionnaire est connu des 2 cotés)

- La compression consiste à déterminer le plus long préfixe de T qui est dans le dictionnaire F et transmettre son index dans F, et recommencer sur la suite de T.
- Pour la décompression il suffit de remplacer chaque index reçu par son contenu dans la table.

EXEMPLE : T = ...ababcbababaaaaaaabbabaabca

index	*f*	*(père, lettre)*
1	a	
2	b	
3	c	
4	ab	1b
5	ba	2a
6	abc	4c
7	cb	3b
8	bab	5b
9	baba	8a
10	aa	1a
11	aaa	10a

T= ...
ababcbababaaaaaaabbabaabca

Structures de données

$T = ...ababcbababaaaaaaabbabaabca$
$C(T)$= 4 6 9 5 12 10 2 9 6 1
Fonction de hachage (collisions résolues) :
$h(a) = 1, h(b) = 2, h(c) = 3, h(ab) = 4, h(ba) = 5, h(abc) = 6, h(cb) = 7,$
$h(bab) = 8, h(baba) = 9, h(aa) = 10, h(aaa) = 11$

1. A la compression,
   - pour reconnaître le plus long préfixe de T dans F : **Arbre digital**,
   - stocker dans la table la valeur du père (index) et la dernière lettre (gain de place)
2. A la décompression
   - la table est suffisante (pas besoin d'arbre digital)
   - décoder le contenu (père, lettre)

Phase adaptative

Au départ la table F contient les lettres de A.

1. Le compresseur
   - détermine le plus long préfixe f de T dans F
   - transmet l'index de f dans F
   - ajoute dans F la séquence fx (où x est le caractère suivant dans T)
   - recommence à lire T à partir de x
2. Le décompresseur, lit le premier numéro et renvoie la lettre, puis
   - Lit un numéro i et consulte la table à cet index
   - s'il y a un contenu, le décode en w_i, puis ajoute à la table (par hachage) une entrée formée de w_{i-1} (ou son index), suivi de la première lettre de w_i
   - sinon ajoute à cet endroit w_{i-1} (ou son index), suivi de **sa** première lettre. Cas de **chevauchement** : $w_{i-1} = bw$ et $w_i = bwb$.

Exemple

Initialement la table contient les lettres :
$h(a) = 1, h(b) = 2, h(c) = 3$
$T = ababcbababaaaaaaa$
$C(T)$= 1 2 4 3 5 8 1 10 11
Fonction de hachage (collisions résolues) :
$h(ab) = 4, h(ba) = 5, h(abc) = 6, h(cb) = 7,$
$h(bab) = 8, h(baba) = 9, h(aa) = 10, h(aaa) = 11$

- Compression
- Décompression

Conclusion : Statistique vs. Dictionnaire

- Dictionnaire code les redondances de séquences
- apprentissage des répétitions $\neq$ méthodes statistiques d'ordre supérieur
  ordre infini, en ne stockant que ce qui est nécessaire.
- Exemple : fichier uniforme contenant 1000 fois le même caractère C
  - Huffman : C codé sur 1 bit $\Rightarrow$ 1000bits,
  - LZW : A chaque étape ajoute un préfixe contenant un C de plus : $1 + \ldots + k = 1000 \Rightarrow k \sim$ 45adresses $\Rightarrow$ 45 * 7bits + 8bits $<$ 350 bits

Conclusion : Statistique vs. Dictionnaire

A l'inverse il existe des cas où la compression statistique est mieux adaptée que LZW :
Fichier d'octets $\rightarrow$ 8 fichiers binaires (le premier formé de tous les premiers bits, le deuxième formé de tous les deuxièmes bits,. . . .).
Compression des 8 fichiers par Huffman OK, mais pas par LZW, car en découpant en 8 "plans" on a pu casser des régularités.

Succession de deux compressions :

- Huffman suivi de LZW : améliore souvent la compression
- LZW suivi de Huffman : NON car le hachage de LZW "cache" les fréquences.