Chapitre I

Codage et théorie de l'information

1 Introduction

Qu'est-ce-qu'une information pour un ordinateur ?

- une information est codée sous forme de suites de mots de 8, 16, 32, 64, 128 bits.
- une information n'a aucun sens pour lui (ne fait qu'appliquer des suites d'opérations sur des mots)

Codage : opération destinée à rendre une information manipulable par un ordinateur, ou par extension, sur un ordinateur, la transformation d'un code en un autre code répondant à d'autres contraintes (compression, chiffrement, résistance à l'erreur, ...)

Décodage : opération destinée à rendre une information sur un ordinateur interprétable par un humain, ou par extension, sur un ordinateur, le retour au code original à partir d'un code encodé (décompression, déchiffrement, correction d'erreur, ...)

Exemples de représentation de données :

- `12 F5` $\Rightarrow$ 4853
- `43 0f a0 00` $\Rightarrow$ 143.625
- `42 6f 6e 6a 6f 75 72` $\Rightarrow$ "Bonjour"
- `18 18 18 ff ff 18 18 18` $\Rightarrow$ image binaire 8×8 (croix)
  autres représentations : 64 octets en niveaux de gris, 192 octets en RGB.

Qu'est ce que la compression ?

C'est la science de la représentation de l'information sous une forme compacte.

Ce qui amène plusieurs questions :

- Qu'est-ce-que l'information ?
- Qu'est-ce-qu'une forme compacte ?

Première réponse partielle :
compact = avec peu de redondance.

Le but de ce cours va donc être d'apprendre à :

- représenter les données sous une forme peu redondante
- supprimer la redondance des données
- implémenter les algorithmes de compression/décompression

Digression philosophique :

- le langage permet d'exprimer une représentation compressée de la réalité,
- les mathématiques sont une représentation symbolique compressée d'une pensée logique,
- la physique consiste à "compresser" la réalité en un ensemble d'équation permettant de la décrire,
- ...

Sur le fond, l'intelligence humaine fonctionne :

- en structurant le réel sous forme de concept (symboles),
- en compressant le réel sous forme d'une description de celui-ci à partir des concepts (suite de symboles),

Autrement dit, nous compressons l'information dans notre cerveau en :

- faisant sens de ce que nous voulons compresser,
- ne retenant et n'organisant que les informations qui nous semblent significatives.

2 Généralités

2.1 Idée

Principe de la compression :

Il s'agit pour une suite de bits de départ :

1. de trouver une suite alternative de bits avec un algorithme de compression,
2. tel que cette suite alternative occupe une place de stockage moins importante que la suite initiale,

3. et qu'il soit possible de reconstituer la suite de bits initiale à partir de cette suite alternative avec un algorithme de décompression.

Intuitivement, un algorithme de compression dépend des données à compresser :

- dans un texte, redondance de mots ou de groupement de lettres,
- dans une image, redondance des couleurs de pixels proches,
- dans une séquence vidéo, redondance entre deux images consécutives,

2.2 Types

Il existe deux familles majeures de technique de compression :

- **compression sans perte :** si la décompression s'effectue sans perte d'information
  à savoir, le signal qui a été compressé est reconstruit exactement lors de la décompression.
- **compression avec perte :** si la décompression ne permet pas de reconstruire exactement le signal original.

Exemples

- 11111111 en unaire :
  compression : $1000_b = 0x8$
  décompression : 11111111 (exact).
- $\pi = 3,14159265358979323846264338327 95\ldots$
  compression : flottant 16 bits
  $(s1, e5, m10) = 0\ 10000\ 1001001000_b = 0x4248$
  décompression : 3.140625 (tronquée)

Le choix du type de compression à utiliser dépend du type de données :

- **compression sans perte :**
  lorsque les informations de la source de données ne doivent perdre aucun détail.
  Exemples : texte, images médicales, exécutable, ...
- **compression avec perte :**
  lorsque la perte d'information résultant de la compression a une importance relativement mineure sur la perception de l'information une fois qu'elle est décompressée (i.e. la dégradation, si elle est perceptible, est considérée comme acceptable).
  Exemples : images, vidéo, audio, ...

2.3 Applications

Les applications de la compression sont partout :

- gain de la place de stockage
- compression avant passage dans un canal de communication
- décompression à l'utilisation

Les algorithmes mis-en-oeuvre sont contraints par leur utilisation pratique :

- image/son/vidéo stockés sur un disque : la compression peut être lente, mais la décompression doit être en temps réel.
- transmission bidirectionnelle : le compression et la décompression doivent être en temps réel.

Note : compresser (trouver la redondance) est en général plus difficile que décompresser (reconstruire).

2.4 Mesure

Définitions

Taux de compression = $\frac{S_c}{S_i}$
Facteur de compression = $\frac{S_i}{S_c}$
Pourcentage de réduction = $\frac{S_i - S_c}{S_i}\%$

où S_i = taille initiale (avant compression)
S_c = taille après compression.

Exemple

une image de 262.144 octets occupe après compression 65.536 octets.

- **Taux de compression** = 0.25
- **Facteur de compression** = 4
- **Pourcentage de réduction** = 75%

Exercice 1: Mesure de compression

1. Donner une définition en français des trois mesures de compression.
2. Un fichier de 131072 octets fait 98.304 octets après compression. Calculer son taux, son facteur et son pourcentage de réduction.
3. Un fichier compressé fait 145260 octets, et a un taux de compression de 0,1. Quel est la taille du fichier décompressé ?
4. Même question si ce fichier a un pourcentage de réduction de 10%.

Exercice 2: Mesure de compression lors d'une dilatation

On regarde dans cet exercice ce qu'il se passe lorsque la taille après compression est plus grande qu'avant compression (*i.e.* $S_i < S_c$).

1. Que se passe-t-il pour le taux de compression ?
2. Que se passe-t-il pour le facteur de compression ?
3. Que se passe-t-il pour le pourcentage de réduction ?

Dans le cas de la compression avec perte, il faut également quantifier la fidélité avec laquelle le fichier a été compressé.

Définition : distorsion D

> mesure de la différence entre la source originale et la source reconstruite après compression puis décompression.

Définition : débit d

> quantité maximale d'information que l'on peut faire circuler dans un canal de communication par seconde (exemple : 1 Mbit/s).

Dans ce cas, la compression ne consiste plus seulement à atteindre le meilleur taux de compression possible.

La compression avec perte pose deux problèmes type :

- **problème distorsion/débit :**
  contrainte : débit de transmission ou place de stockage maximale.
  problème : compresser les données pour atteindre ce débit tout en conservant la qualité maximale.
  = respecter d tout en minimisant D.
- **problème débit/distorsion :**
  contrainte : fidélité.
  problème : compresser les données pour respecter cette qualité avec le plus petit débit possible.
  = respecter D tout en minimisant d.

Dans ce cours, nous n'aborderons probablement pas le problème de la compression avec perte par manque de temps.

3 Limites

Dans cette partie, on se pose des questions de bon sens sur l'intérêt et l'efficacité d'un compresseur sans perte :

- Existe-t-il un compresseur universel ?
  à savoir : un compresseur capable de tout compresser.
- Existe-t-il un compresseur qui n'empire jamais la situation ?
  est-ce que je gagne toujours de la place lorsque je compresse ?
- Est-ce que mon compresseur sera efficace souvent ?
  est-ce qu'en moyenne je gagne quelque chose lorsque je compresse ?
- Est-ce que je gagne quelque chose en compressant le résultat de ma compression ?
  ou le problème de la limite de la compressibilité.

Mais avant de répondre à cette question, nous devons tout d'abord définir ce qu'est un compresseur.

On note :

$\mathbb{B}$ l'ensemble des mots binaires.
$\mathbb{B}_n$ l'ensemble des mots binaires de longueur n exactement.
$|x|$ la longueur du mot x.

Définition 1 (compresseur)**.** Un compresseur est une fonction F injective de $\mathbb{B} \to \mathbb{B}$ qui possède au moins un mot u tel que $|F(u)| \leq |u|$.
Si $|F(u)| < |u|$, la compression est dite stricte.

Rappel : Un injection est une application telle que $f(u) = f(v) \Rightarrow u = v$.

Cette définition est vraiment peu contraignante : elle demande juste à ce que le compresseur n'augmente pas la longueur d'au moins un mot. On aimerait évidemment qu'un compresseur soit capable d'effectuer une compression stricte sur un sous-ensemble non négligeable de $\mathbb{B}$.

3.1 Compresseur universel

Proposition 1. *Il n'existe pas de compresseur F permettant de compresser strictement n'importe quel mot.*

Démonstration:

Par l'absurde : supposons qu'un tel compresseur F existe.

Soit D contenant une suite de bits de taille $|D|$.

Si on note $D_0 = D$, $D_i = F(D_{i-1})$ et $\tau_i = |D_i|/|D_{i-1}|$.

On note F^n l'application consécutive de n fois l'algorithme de compression (*i.e.* $D_n = F^n(D_0)$), alors, le taux de compression au bout de n itérations est :

$$\tau^n = \prod_{i=1}^{n} \tau_i$$

Or, d'après la définition de F, on a $\tau_i < 1$, et en conséquence :

$$\lim_{n\to\infty} \tau^n = 0$$

Donc, si un tel algorithme existe, il est possible de rendre tout mot aussi petit que l'on souhaite, c'est à dire le réduire à 1 bit, ce qui est absurde. □

Démonstration: (2)

La caractéristique d'un compresseur F sans perte est qu'il existe un decompresseur F^{-1} (son inverse) tel que pour tout mot u, $F(F^{-1}(u)) = u$.

Sans perte de généralité, considérons que le compresseur s'applique sur l'ensemble des mots binaires.

L'ensemble $\mathbb{B}_{\leq n} = \cup_{i=1}^{n} \mathbb{B}_i$ des mots de taille inférieure ou égale à n a pour cardinal $\#\mathbb{B}_{\leq n} = \sum_{i=1}^{n} \#\mathbb{B}_i$ (car les $\mathbb{B}_n$ sont disjoints).
D'où $\#\mathbb{B}_{\leq n} = 2 + 2^2 + \ldots 2^n = 2^{n+1} - 2$.

Mais si F compresse $\mathbb{B}_n$, cela signifie que F est une injection dans $\mathbb{B}_{n-1}$.

Or, $\#\mathbb{B}_n > \#\mathbb{B}_{n-1}$. Une telle injection n'existe donc pas, et il n'existe pas de compresseur qui compresse tout mot.

□

Remarquez que cet argument revient finalement au fait qu'un bit ne peut pas être compressé !

3.2 Gain ponctuel

Proposition 2. *Si un compresseur F compresse strictement un mot u (à savoir si $\exists u$ tel que $|F(u)| < |u|$), alors il existe un mot v pour lequel $|F(v)| > |v|$.*

Autrement dit, si un compresseur compresse au moins un mot, alors il existe des mots qu'il dilate.

Démonstration:

Notons $\mathbb{B}_{<|u|} = \cup_{k<|u|} \mathbb{B}_k$.

Si F compresse strictement un mot u, ceci signifie que $u \in \mathbb{B}_{|u|}$ et que $F(u) \in \mathbb{B}_{<|u|}$.

Or, le nombre d'éléments de $\mathbb{B}_{<|u|}$ est fini.

Ainsi, $F(u)$ prend la place d'un élément v de $\mathbb{B}_{<|u|}$. (indépendamment de la façon dont ils se réorganisent pour ne pas être dilatés) «chaises musicales !».

Il existe donc $v \in \mathbb{B}_{<|u|}$ et $F(v) \notin \mathbb{B}_{<|u|}$. En conséquence, $|F(v)| > |v|$. □

Exercice 3: Gain moyen de compression

On suppose dans cet exercice que la taille d'un fichier est un nombre d'octets entiers. On note $\mathbb{B}_p$ l'ensemble des mots binaires de p octets, et $\mathbb{B}_{<p}$ l'ensemble des mots binaires de moins de p octets.

1. Quelle est le nombre de mots différents de p octets ? (*i.e.* calculer $\#\mathbb{B}_p$).
2. Quelle est le nombre de mots différents de moins de $p-1$ octets ? (*i.e.* calculer $\#\mathbb{B}_{\leq p-1}$).
3. Combien y-a-t-il de fois plus de mots de p octets que de mots de moins de $p-1$ octets ? (*i.e.* calculer $\#\mathbb{B}_p / \#\mathbb{B}_{\leq p-1}$)
4. Supposons que l'on définisse 256 compresseurs différents afin d'être sûr de compresser chaque mot d'au moins un octet, en découpant l'ensemble des mots de p octets en 256 ensembles E_i différents. Cette stratégie est-elle intéressante ?

3.3 Gain moyen

Proposition 3. *Le nombre de suites de bits pouvant être compressé d'au moins p octets représente une proportion de moins de $\frac{2}{256^p}$ sur l'ensemble de ces suites.*

Démonstration:

Soit D une suite de bits. Soit F une méthode de compression (donc une injection). A savoir que si $D_c = F(D)$, alors $F^{-1}(D_c) = D$.

Notons $n = |D|$ et supposons que nous voulions savoir quelle est la proportion de suites de bits (parmi toutes celles existantes) de taille n qui peuvent être compressées d'au moins un octet (= 8 bits).

Il y a 2^n suites de bits D différentes pouvant être stockées sur n bits.

Or, le nombre de suites de bits D_c compressés pour être stockés sur moins de $n-8$ bits est :

$$2^1 + 2^2 + ... + 2^{n-8} = 2^{n-7} - 2 \simeq 2^{n-7}$$

En conséquence, la proportion de suite de bits pouvant être compressé d'au moins un octet est au plus de :

$$2^{n-7}/2^n = 2^{-7} = 1/128 < 1\%$$

En reprenant le raisonnement, la proportion des D_c pouvant être compressé d'au moins p octets est $2^{-8.p+1}$. □

3.4 Compression répétée

Proposition 4. *Pour n'importe quel mot de départ sur lequel on applique de manière répétée un compresseur, on se trouve nécessairement dans l'un des deux cas suivants :*

1. *soit on est dans un cycle (= suite finie de mots qui se répète indéfiniment),*
2. *soit les mots successifs obtenus deviennent arbitrairement grands.*

Démonstration:
Soit F la méthode de compression. Notons $F^n(u)$ l'application de n compressions successives F sur le mot u et $\mathbb{F}(u) = \cup_{k>0}\left(F^k(u)\right)$ l'ensemble contenant toutes ces compressions successives de u.

Deux cas sont alors possibles :

- soit $\mathbb{F}(u)$ est fini, et dans ce cas $F^n(u)$ cycle nécessairement entre les valeurs de $\mathbb{F}(u)$ (puisqu'il y a une infinité dénombrable de $F^n(u)$).
  On notera que si $\exists p$ tel que $F^p(u) = u$, alors on entre dans un cycle de longueur p et $\#\mathbb{F}(u) = p$.
- soit $\mathbb{F}(u)$ n'est pas fini, mais comme tout $\mathbb{B}_{<p}$ est quand à lui fini, si on compresse plus de $\#\mathbb{B}_{<p}$ fois, alors on trouvera nécessairement un $F^n(u)$ tel que $|F^n(u)| > p$, et ceci pour toute valeur de p.

□

3.5 Conclusion

Conséquences :

1. Il existe une limite à la compression.
2. Il n'existe pas de compresseur capable de compresser tout mot.
3. Si un compresseur compresse strictement au moins un mot, alors il en dilate au moins un autre.
4. La proportion du nombre de mots qu'il est possible de compresser de manière appréciable est très faible.
5. L'application itérée d'un même compresseur n'est pas une façon d'améliorer la compression.

Remarque : l'ensemble de ces conclusions n'est pas directement applicable à la compression avec perte.

On peut donc sérieusement se poser la question :

La compression va-t-elle être utile à l'exception de certains cas marginaux ?
Et donc, l'effort en vaut-il la peine ?

Évidemment, oui car :

- l'humain produit naturellement de l'information redondante (langue orale, langue écrite, ...)
- le stockage d'information dans la mémoire d'un ordinateur est :
  - ◇ guidé avant tout par la rapidité d'accès à l'information (alignement).
  - ◇ exceptionnellement peu efficace (exemple : **ü** apparaît dans 4 mots du dictionnaire sur 100000, alors qu'il est codé sur le même nombre de bits que le **e** qui a une fréquence d'apparition d'environ 15%).
  - ◇ il existe des ensembles de méthodes de compression différentes adaptées aux types de données que l'on souhaite compresser.

Exercice 4: Stockage machine (rappel)

1. Qu'est-ce-que l'alignement dans la mémoire d'un ordinateur ?
2. Pourquoi l'utilisation de types comptacts (`unsigned char`, `short int`) n'est pas nécessairement une bonne idée lorsque l'on souhaite améliorer les performances d'un code ?
3. Pourquoi le `sizeof` d'une structure/classe n'est pas toujours égal à la somme des `sizeof` de ses champs ?
4. Que ce passe-t-il pour un tableau d'éléments vérifiant la condition précédente ?
5. Donner un cas où le `sizeof` de la structure est plus de deux fois plus important que la somme des `sizeof` de ses champs.
6. En `C++`, existe-t-il des cas où la classe n'a pas la taille qu'elle devrait faire, même en tenant compte de toutes les règles d'alignement indiquées ci-dessus ?

On peut donc mettre à jour la définition d'un compresseur sans perte :

Définition 2 (compresseur)**.** Un compresseur est une fonction F injective telle qu'il existe $E \subset \mathbb{B}$ tel que $\forall u \in E, |F(u)| < |u|$.

Autrement dit, s'il existe un ensemble de mots pour lesquels la fonction permet effectivement de réduire la taille du mot.

L'objectif de ce cours est de décrire les notions et les méthodes qui vous permettront de choisir et d'utiliser la méthode de compression adaptée à vos données.

Maintenant que nous savons qu'il est impossible pour un compresseur de tout compresser, prenons le problème à l'envers.

4 Codage

4.1 Formalisation

Lorsque nous créons de l'information, nous la représentons sous forme d'une suite de symboles.

Définition 3 (Alphabet)**.** Un **alphabet** $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ est l'ensemble des symboles utilisés pour représenter une information.

Définition 4 (Source)**.** On appelle **source** tout dispositif ou support capable de produire de l'information.

Exemples :

- Une phrase est représentée à l'oral, sous forme d'une suite de phonèmes.
- Un nombre est représenté sous forme d'une suite de chiffres.
- Une image discrète est représentée sous la forme d'une matrice de couleurs.

Lorsque nous transmettons de l'information, nous utilisons, à la place des symbole, un code adapté au support de stockage ou de transmission.

Définition 5 (code)**.** Un **code** C associé à un alphabet $\mathcal{A}$ est un ensemble $\{c_1, c_2, \ldots, c_n\}$ de même taille que l'alphabet qui permet d'associer un code c_i à chaque symbole s_i de l'alphabet.

Un code peut être :

- **de longueur fixe** :
  - ◇ chaque lettre de l'alphabet a un code `ASCII` codé sur 8 bits.
  - ◇ un entier est stocké sur 8, 16 ou 32 bits suivant la valeur à stocker (choix statique).

- **de longueur variable** :
  - ◇ chaque phonème d'un mot est codé à l'aide d'une à trois lettres (*/u/* = ou, */n/* = n).
  - ◇ l'alphabet Morse affecte de un à quatre points ou traits à chaque lettre (E=•, O=– – –, ...).

Pour un **alphabet** $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ et son **code** $C = \{c_1, c_2, \ldots, c_n\}$ associé,

Définition 6 (codage)**.** opération qui consiste à transformer tout symbole $s_i \in \mathcal{S}$ issu d'une source en son code $c_i \in C$.

Un exemple de fonction triviale de codage est $C : \mathcal{S} \rightarrow C$ est typiquement $C(s_i) = c_i$.

Définition 7 (décodage)**.** opération inverse du codage (= transformation du code au symbole).

A priori, rien ne garantit que la fonction de codage C soit inversible (et donc que le décodage d'une chaîne codée permettra de retrouver cette chaîne).

Dans la suite de ce cours, indépendamment du codage choisi, l'information codée sera toujours considérée comme une suite de bits (ce qui est toujours le cas sur un ordinateur).

Exemple :

Soit l'alphabet $\mathcal{S} = \{A, B, C, D, E\}$.
- code de longueur fixe : $C_1 = \{000, 001, 010, 011, 100\}$.
- code de longueur variable : $C_2 = \{0, 100, 101, 110, 111\}$.

Si l'on considère la chaine symbolique "BAAAAAAAC", son codage est :
- 001000000000000000000000010 avec le code C_1
- 1000000000101 avec le code C_2

Le code C_2 permet d'obtenir une représentation plus compacte que le code C_1.

Comment s'assurer, lorsque l'on construit un code, que celui-ci sera décodable ?

4.2 Décodabilité

Définition 8 (Code non-singulier)**.** Un code est dit non-singulier si à chaque symbole s_i a un seul code c_i unique.
Autrement dit, $s_i \neq s_j \Rightarrow c_i \neq c_j$.

Un code non singulier est donc un code pour lequel la fonction de codage C est inversible. Cette condition est néanmoins insuffisante pour garantir la décodabilité d'une chaîne codée.

Définition 9 (Extension d'un code)**.** L'extension C^* d'une fonction de codage C est l'application permettant de coder toute suite de longueur finie.
$\forall x_1 x_2 \dots x_n \in \mathcal{S}^*, C^*(x_1 x_2 \dots x_n) = C(x_1)C(x_2)\dots C(x_n)$
où $C(x_1)C(x_2)\dots C(x_n)$ est la concaténation de l'ensemble des codages individuels des symboles.

Définition 10 (Code décodable de manière unique)**.** Un code est décodable de manière unique si l'extension de sa fonction de codage n'est pas singulière, *i.e.* $\forall (s, s') \in \mathcal{S}^{*2}, s \neq s' \Rightarrow C^*(s) \neq C^*(s')$.

Autrement dit, le décodage de toute chaine peut s'effectuer sans ambiguïté.

Décodabilité d'un code à taille fixe

L'extension C^* d'une fonction de codage C non singulière d'un code à taille p fixe (i.e. code tel que $|C(s_i)| = p$ pour tout $s_i \in \mathcal{S}$) est toujours décodable de manière unique.

Pour toute chaîne $x_1 x_2 \dots x_n \in \mathcal{S}^*$, $|C^*(x_1 x_2 \dots x_n)| = np$.

Donc, $C(x_i)$ = les bits de ip à $(i+1)p - 1$ de $C^*(x_1 x_2 \dots x_n)$. Comme C n'est pas singulier, il est inversible.

Cela revient à découper la chaîne codée par paquet de p bits, puis à décoder chaque paquet indépendamment l'un de l'autre.

Exemple

Soit l'alphabet $\mathcal{S} = \{A, B, C, D, E\}$ et son code de taille fixe associé $C = \{000, 001, 010, 011, 100\}$ ($p = 3$).
Codage de la chaine $BDAD$:
$C^*(BDAD) = 001011000011$.
Découpage par paquet de 3 :
$001011000011 \Rightarrow 001\ 011\ 000\ 011$.
Décodage individuel de chaque paquet :

$$\begin{aligned} C^{*-1}(001011000100) &= C^{-1}(001)C^{-1}(011)C^{-1}(000)C^{-1}(100) \\ &= BDAD \end{aligned}$$

Décodabilité d'un code à taille variable

Même pour une fonction de codage C non singuliere d'un code à taille variable, son extension C^* n'est pas nécessairement décodable de manière unique.

Exemple

Soit l'alphabet $\mathcal{S} = \{A, B, C, D\}$ et son code de taille variable associé $C = \{0, 10, 010, 101\}$.

Alors, la chaîne 0100101010 peut être décodé comme :

- 0 10 010 101 0 = $ABCDA$
- 010 0 101 010 = $CADC$
- 0 10 0 10 10 10 = $ABABBB$
- ...

Dans ce cas, il n'y a plus moyen de savoir quand on passe du code d'un symbole au code du symbole suivant.

Que faire alors ?

- Ajouter un séparateur (ou ponctuation) pour séparer les codes, mais le coût est prohibitif (= ajouter un séparateur entre chaque symbole du code).
- Trouver les conditions qui font qu'un code n'est pas ambigu.

Exercice 5: Codage

1. Soit l'alphabet $\mathcal{A} = \{A, I, P, S\}$ et le code à longueur fixe associé $C_1 = \{00, 01, 10, 11\}$. Décoder le code suivant :
   10001001100010001000100010001110001001
2. Quelle est la longueur moyenne de bits/symbole pour cette chaîne avec le code C_1 ?
3. On considère maintenant le code à longueur variable $C_2 = \{10, 110, 0, 111\}$. Coder la chaîne de symboles obtenues à la question 1 avec ce code.
4. Quelle est la longueur moyenne de bits/symbole pour cette chaîne avec le code C_2 ?
5. Interpréter le résultat.

Exercice 6: Code décodable de manière unique

Déterminer si les codes suivants sont décodables de manière unique.

1. $C_1 = \{0, 01, 11, 111\}$
2. $C_2 = \{0, 01, 110, 111\}$
3. $C_3 = \{0, 10, 110, 111\}$

4. $C_4 = \{1, 10, 110, 111\}$
5. $C_5 = \{0, 01, 011, 0111\}$

4.3 Codage préfixe

Dans l'exemple précédent, l'ambiguïté est due au fait que :

- pour deux mots de codes de longueur différente, si l'un est le préfixe de l'autre (par exemple 0 est le préfixe de 010), il n'y a aucun moyen de savoir, après avoir lu ce préfixe, lequel des deux codes choisir.
- remarquons que si deux codes de même longueur commencent par le même préfixe, il est possible d'obtenir un code de longueur variable non ambigu (par exemple $C = \{1, 01, 00\}$).
- donc, ce n'est pas le partage d'un préfixe qui pose problème, mais le fait qu'un mot de code soit préfixe d'un autre.

Nous donnons maintenant un ensemble de définition permettant de préciser ces notions.

Soit un code $C = \{c_1, c_2, \ldots, c_n\}$.

Définition 11 (Mot de code préfixe)**.** Un mot de code c_i est préfixe d'un mot de code c_j si c_j peut s'écrire comme la concaténation de c_i et d'un autre mot de taille non nulle.

Exemple

soit deux mots de code $c_1 = 011$ et $c_2 = 01101$. Alors c_1 est préfixe de c_2 car $c_2 = c_1 01$.

Définition 12 (Code préfixe (ou code instantané))**.** Un code C est préfixe si aucun de ses mots de code n'est préfixe l'un de l'autre. *i.e.* $\forall (c_1, c_2) \in C^2$ tel que $c_1 \neq c_2$, c_1 n'est pas préfixe de c_2.

Exemple

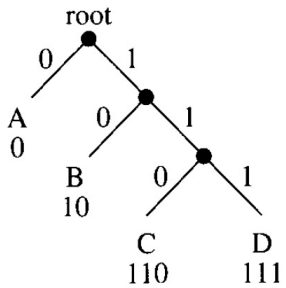
$C = \{0, 10, 010, 101\}$ n'est pas un code préfixe.
$C' = \{0, 10, 110, 111\}$ est un code préfixe.

Remarque : trivialement, si le code est préfixe, alors il peut être décodé symbole par symbole, et il est décodable de manière unique.

On peut utiliser un arbre pour vérifier si un code est préfixe.

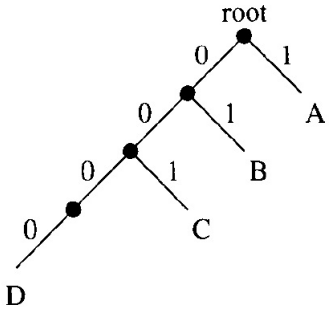
Exemples :

$C_1 = \{0, 10, 110, 111\}$

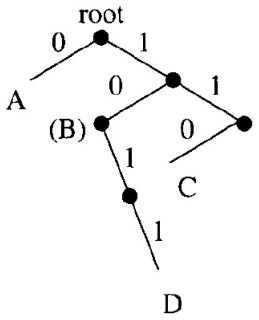


code préfixe

$C_2 = \{1, 01, 001, 0000\}$



code préfixe



$C_3 = \{0, 10, 110, 1011\}$

code non préfixe

Le code est préfixe si tous les mots du code apparaissent aux nœuds de l'arbre contenant l'ensemble des codes à ses feuilles.

Le décodage d'une code préfixe peut être effectué avec l'arbre associé (une lecture d'un mot du code = une descente de l'arbre).

Lorsque un code est décodable de manière unique, on dit qu'il a une propriété d'auto-ponctuation :

- pour un code de longueur fixe, l'auto-ponctuation est induite par la longueur fixe ($\forall(i, j), |c_i| = |c_j|$).
- pour un code prefixe de longueur variable, la ponctuation entre les codes de chaque symbole est induite de par les propriétés du codage préfixe.

Note :

Il est possible avec un code non préfixe de longueur variable de décoder des messages sans ambiguïté, pour peu que la succession des codes y conduise.

Exemple : pour $C = \{01, 11, 111\}$, la chaîne codée 11011101111 n'est pas ambiguë.

4.4 Inégalité de Kraft

Théorème 5 (Inégalité de Kraft (IK)). *Il existe un code binaire préfixe $C = \{c_1, c_2, \ldots, c_n\}$ avec n mots de code de longueurs respectives $\{l_1, l_2, \ldots, l_n\}$ ssi :*

$$K(C) = \sum_{j=1}^{n} 2^{-l_j} \leq 1$$

Idée :

Noter tout d'abord que sur un arbre binaire quelconque :

1. si on associe la valeur $v_i = 2^{-l_i}$ à chaque feuille de profondeur l_i, alors $\sum_i v_i = 1$.
2. la valeur 2^{-l} associée à tout nœud de profondeur l est égale à la somme des valeurs aux feuilles de tous ses fils.

En conséquence, si un code binaire est préfixe, il peut être représenté sous la forme d'un arbre binaire, et $\sum 2^{-l_j} \leq 1$ (le cas inférieur correspondant où certaines feuilles ne sont pas associées au code).

Inversement, si seules les longueurs de code sont connues,

- si l'IK tient, alors il est possible de construire un arbre binaire associé, et il existe donc un code préfixe.
- si elle ne tient pas, alors il est impossible de construire un arbre binaire, et il n'existe en conséquence pas de code préfixe.

Exemples :

- pour un code de longueurs $C_1 = (1, 2, 3, 3)$, alors $K(C_1) = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^3} = 1$.
  Donc il existe un code binaire préfixe (exemple : $C_1 = \{0, 10, 110, 111\}$).
- pour un code de longueurs $C_2 = (1, 2, 2, 3)$, alors $K(C_2) = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^3} = 1,125 > 1$.
  Il n'existe donc pas de code binaire préfixe associé.

Remarques :

- l'IK est une condition obligatoire pour qu'un code préfixe existe. Donc, si elle n'est pas vérifiée, le code préfixe ne peut pas exister.
- si l'IK est vérifiée par un code, celui-ci n'est pas nécessairement préfixe. Par contre, il existe un code préfixe valide avec ces longueurs.
- si l'IK < 1, alors il est probablement possible de trouver un code plus court.
- l'IK n'explique pas comment trouver ce code préfixe.

- l'IK peut être généralisée avec des codes n-aires (utiliser des arbres n-aires).

Exercice 7: Code à taille fixe et inégalité de Kraft

1. supposons que nous voulons coder n symboles avec des mots de code de la même longueur p. Comment utiliser l'inégalité de Kraft pour choisir cette longueur ?
2. Calculer la longueur p nécessaire pour $n = 16$ symboles et $n = 24$ symboles.
3. Soit le code suivant $C_1 = \{0, 000, 11, 111\}$. Vérifie-t-il l'inégalité de Kraft ? Si oui, que peut-on en conclure ?

Exercice 8: Inégalité de Kraft

1. En utilisant l'inégalité de Kraft, vérifier s'il existe un code préfixe possédant les longueurs suivantes $\{1, 2, 3, 4\}$?
2. Même question avec $\{1, 3, 4, 4\}$.
3. Même question avec $\{1, 2, 4, 4, 5, 5, 5, 5\}$.
4. Pour les codes préfixes qui existent ci-dessus, en donner un.
5. Pour les codes préfixes qui existent ci-dessus, existe-t-il un code plus court ?

Exercice 9: Code quelconque et inégalité de Kraft

On veut créer un code C pour coder tout entier $n \in \mathbb{N}$ de la manière suivante. Le code pour n est $c_n = \underbrace{0 \ldots 0}_{n \text{ zero}} 1$.

1. Le code C est-il un code préfixe ?
2. Verifie-t-il l'inégalité de Kraft ?
3. Si je tire au hasard n'importe quelle suite de 0 et 1, puis-je décoder ce code avec le code C

4.5 Compression

Performance d'un code sur une chaîne

La performance d'un code pour le codage d'une chaîne S peut se mesurer en calculant le nombre moyen de bits par symbole utilisé dans le codage :

$$\overline{\ell} = \frac{\sum_i \#_{s_i}(S).|c_i|}{\sum_i \#_{s_i}(S)} = \frac{\sum_i \#_{s_i}(S).|c_i|}{|S|}$$

où $\#_{s_i}(S)$ est le nombre de symbole s_i dans la chaîne S, $|c_i|$ est le nombre de bits utilisé pour représenter le code c_i, et $|S|$ la longueur de la chaine S.

Dans le cas d'un code longueur fixe, la longueur de chaque mot de code $|c_i|$ est une constante c pour tout i.
En conséquence, le nombre moyen de bits par symbole est toujours :

$$\overline{\ell} = c$$

A noter que ce calcul ne permet pas de connaître l'efficacité du code en moyenne, mais seulement sur la chaîne considérée.

Exemple

Soit l'alphabet $\mathcal{S} = \{A, B, C, D, E\}$.

- $C_1 = \{000, 001, 010, 011, 100\}$ est un code de longueur fixe.
- $C_2 = \{0, 100, 101, 110, 111\}$ est un code de longueur variable.

Si l'on considère la chaine symbolique "BAAAAAAAC" :

Pour le code de longueur fixe :
$\overline{\ell}_1 = 3$ bits/symbol

Pour le code de longueur variable, la fréquence de chaque symbole est $(7, 1, 1, 0, 0)$:
$\overline{\ell}_2 = \frac{7.1 + 1.3 + 1.3 + 0.3 + 0.3}{7 + 1 + 1 + 0 + 0} \simeq 1.4$ bits/symbole

Le code C_2 est donc deux fois plus efficace pour stocker la chaîne S que le code C_1.

Performance moyenne d'un code

Intuitivement, un code compresse efficacement une chaîne si les symboles les plus fréquents sont codés avec peu de bits.

Si l'on dispose des fréquences d'apparition réelle f_i de chaque symbole s_i de la source, la longueur moyenne du code est :

$$\overline{\ell} = \sum_i f_i.|c_i|$$

Sinon, on peut calculer la fréquence d'apparition empirique $\overline{f}_i$ d'un symbole s_i est à partir d'une chaîne de la source :

$$\overline{f}_i = \frac{\#_{c_i}(S)}{|S|}$$

Cette chaîne S doit avoir une longueur suffisamment importante afin que la fréquence empirique soit représentative.

Tout d'abord, remarquons que l'on peut considérer que :

- la compression est un codage.
- la décompression est un décodage.

Performance optimale d'un code

Nous recherchons la meilleure compression possible, donc un code tel que sa longueur moyenne $\overline{\ell} = \sum_i f_i.|c_i|$ est la plus faible possible.

En conséquence, un critère de choix de la longueur des codes pourrait être : trouver le codage optimal C_{optimal} tel que $\sum_i f_i.|c_i|$ soit minimum, i.e.

$$C_{\text{optimal}} = \arg\min_C \sum_i f_i.|c_i|$$

ce qui correspond à utiliser les codes de plus faible longueur possible pour les symboles les plus fréquents.

5 Théorie de l'information

Qu'est ce que la théorie de l'information ?

C'est une théorie probabiliste permettant de :

- quantifier le contenu moyen d'information présent dans un ensemble de données,
- mesurer la redondance d'information,
- coder l'information,
- compresser les données,
- mesurer la perte d'information lors d'une compression avec perte,
- mesurer le degré de sécurité d'un code

Autrement dit, un outils mathématique permettant de quantifier la qualité des méthodes utilisées pour le codage, la compression et la cryptographie.

Remarque 1 :
Cette théorie ne s'intéresse qu'aux aspects mathématiques et communicationnels de l'information, et non au contenu cognitif (sa sémantique) ou à sa forme matérielle/énergétique.

En conséquence, ses résultats dependent donc uniquement des propriétés probabilistes de l'information et du canal de communication considérés et non du sens de ces informations ou de la façon dont elles sont transportées.

5.1 Rappels

Définition 13 (espace probabilisé et variable aléatoire)**.** Un espace probabilisé est un triplet $(\Omega, \mathcal{A}, \Pr)$ formé de :

- un univers Ω (l'ensemble des évènements élémentaires).
- un ensemble d'évènements composés $\mathcal{A}$ (typiquement $\mathcal{P}(\Omega)$, σ-algèbre sur Ω *i.e.* ensemble de parties stables par complémentaire et union dénombrable).
- une mesure Pr sur $\mathcal{A}$ tel que $\Pr(\Omega) = 1$.
- une variable aléatoire X est une fonction qui associe une valeur (typiquement dans $\mathbb{R}$) à un évènement de Ω.

Exemple : pour un dé (non pipé), et X un tirage :

- $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- X est la valeur de la face. La loi de X est celle du dé.
- Pr est la mesure de probabilité $\mathcal{A}$.
- probabilité que le tirage soit 3 : $\Pr[X \in \{3\}] = \frac{1}{6}$.
- probabilité que le tirage soit impair : $\Pr[X \in \{1, 3, 5\}] = \frac{1}{2}$.
- probabilité de deux tirages indépendants impairs consécutifs :

$$\Pr\left[(X_1, X_2) \in \{1, 3, 5\}^2\right] = \Pr\left[X_1 \in \{1, 3, 5\}^2\right] . \Pr\left[X_2 \in \{1, 3, 5\}^2\right] = \frac{1}{2} . \frac{1}{2} = \frac{1}{4}$$

5.2 Self-information

Première approche :
On modélise une source par une variable aléatoire X dont tous les tirages sont :

- indépendants (i.e. on peut tirer des tirages précédents une information quelconque sur les tirages suivants).
- équiprobables (ils ont tous la même probabilité d'apparition).

On veut répondre à la question suivante dans tous les cas ci-dessous : de combien de bits ai-je besoin **au minimum** pour stocker des informations issues de la source X ?

Exemple :

X produit n symboles A_i tels que $\Pr[X = A_i] = 1/n$.
Logiquement, tous les symboles du code ont la même taille.

Que dit la théorie de l'information à ce propos ?

Définition 14 (self-information)**.** La quantité d'information apportée par la réalisation d'un évènement A de probabilité $\Pr[X = A]$ issu d'une source aléatoire X est :

$$I(A) = -\log_2 \Pr[X = A]$$

où $\log_2$ est le logarithme en base 2 (i.e. $\log_2 x = \ln x / \ln 2$).

L'unité de I est le bit.

Par la suite, on pourra noter $\Pr[A]$ pour la probabilité de l'évènement A si la source aléatoire est implicite.

Dans le cas précédent, un évènement A tels que $\Pr[A] = \frac{1}{n}$ apporte une quantité d'information :

$$I(A) = -\log_2 \Pr[A] = -\log_2(\tfrac{1}{n}) = \log_2(n)$$

Exemple :

Si on a 2^p symboles A_i équiprobables (donc $\Pr[A_i] = 2^{-p}$), alors on a besoin de $I(A_i) = -\log_2 2^{-p} = p$ bits. Donc il faut un minimum de p bits pour représenter chacun de ces 2^p symboles.

Rappel :

Si deux évènements A et B sont indépendants alors :

$$\Pr[AB] = \Pr[A].\Pr[B]$$

Conséquence :

Si on considère maintenant la self-information de deux évènements indépendants :

$$\begin{aligned} I(AB) &= -\log_2 \{\Pr[AB]\} = -\log_2 \{\Pr[A].\Pr[B]\} \\ &= -\log_2 \Pr[A] - \log_2 \Pr[B] \\ &= I(A) + I(B) \end{aligned}$$

En conséquence, la quantité d'information issue de deux évènements indépendants s'ajoutent.

Applications : nombre de bits minimum pour représenter les informations d'une source

Exemple 1 :

Si une source produit deux symboles A et B tels que $\Pr[A] = \Pr[B] = \frac{1}{2}$.

Réponse : 1 bit. $\mathcal{S} = \{A, B\}$, $C = \{0, 1\}$.

Exemple : 01111000 représente la suite des tirages $ABBBBAAA$.

Exemple 2 :

Si une source produit quatre symboles $\{A_1, \ldots, A_4\}$ tels que $\Pr[A_i] = \frac{1}{4}$.

Réponse : 2 bits. $\mathcal{S} = \{A_1, \ldots, A_4\}$, $C = \{00, 01, 10, 11\}$.

Exemple : 01111000 représente la suite des tirages A_2, A_4, A_3, A_1.

Il n'est pas possible de représenter les informations issues de ces sources avec moins de bits.

Autrement dit, il n'est pas possible de compresser plus les messages issus de telles sources car elles sont indépendantes (la connaissance des symboles suivants/précédents n'apporte aucune information sur le symbole courant).

5.3 Entropie

Deuxième approche :

Si une source produit trois symboles $\mathcal{S} = \{A, B, C\}$ tels que $\Pr[A] = \frac{1}{2}$ et $\Pr[B] = \Pr[C] = \frac{1}{4}$.

De combien de bits a-t-on besoin pour coder les informations de cette source ?

Clairement :

- le nombre de bits doit dépendre du symbole codé : plus il est fréquent, plus son code doit être court.
- le décodage doit être unique.

Une solution triviale est de choisir : $C = \{0, 10, 11\}$ (par exemple). Les symboles deux fois moins fréquents ont une longueur deux fois plus longues.

Que dit la théorie de l'information ?

Définition 15 (Entropie)**.** L'entropie d'une source est la quantité moyenne d'information émis par symbole. Autrement dit, c'est l'espérance de la quantité d'information :

$H(X) = \mathbb{E}(I(X)) = \sum_{A \in X} \Pr[A].I(A)$.

Note : $0 \log_2 0 = 0$.

Dans le cas de symboles indépendants $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$, avec des probabilités d'occurrence $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, l'entropie est définie par :

$$H(\mathcal{P}) = \sum_{j=1}^{n} p_j.I(s_j) = -\sum_{j=1}^{n} \Pr[s_j] \log_2 \Pr[s_j] = -\sum_{j=1}^{n} p_j \log_2 p_j$$

Autrement dit, l'entropie est une mesure qui permet de connaître :

- le nombre minimum de symboles binaires nécessaires pour coder l'alphabet de la source.
- le nombre minimum de bits pour coder un message de longueur L (en moyenne de $L.H(\mathcal{P})$).

Exemple 1 :

Si X produit trois symboles $\{A_1, A_2, A_3\}$ tels que $\Pr[A_1] = \frac{1}{2}$ et $\Pr[A_2] = \Pr[A_3] = \frac{1}{4}$.

L'entropie de cette source est :

$$\begin{aligned} H(X) &= -\Pr[A_1] \log_2 \Pr[A_1] - \Pr[A_2] \log_2 \Pr[A_2] - \Pr[A_3] \log_2 \Pr[A_3] \\ &= -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{4} \log_2 \frac{1}{4} - \frac{1}{4} \log_2 \frac{1}{4} \\ &= 1.5 \text{ bits} \end{aligned}$$

On reprend le codage $C = \{0, 10, 11\}$. On note ℓ_i la longueur de codage de symbole A_i (*i.e.* $(\ell_1, \ell_2, \ell_3) = (1, 2, 2)$).

Pour un suite s de longueur L de symboles tirés suivant la loi X,

- Il y a en moyenne $L.\Pr[A_i]$ symboles A_i.
- La longueur moyenne de s est $|s| = \sum_i \ell_i.L \Pr[A_i] = L(1.\frac{1}{2} + 2.\frac{1}{4} + 2.\frac{1}{4}) = 1.5L$.

Le codage C est donc optimal, tout suite de symboles de loi X ne pourra pas être compressée (i.e. représenté avec moins de bits).

Exemple 2 :

Si les symboles de la source sont équiprobables.

Pour 2^n symboles A_i équiprobables (donc $Pr[A_i] = 2^{-n}$),

$H(X) = -\sum_{i=1}^{2^n} Pr[A_i] \log_2 Pr[A_i] = -2^n.(2^{-n} \log_2 2^{-n}) = n$ bits.

Exemple 3 :

Si les symboles de la source ont pour probabilité $\mathcal{P} = \{1, 0, 0, 0\}$,

$H(X) = -\sum_i Pr[A_i] \log_2 Pr[A_i] = -1 \log_2 1 - 3.(0 \log_2 0) = 0$ bits.

La source ne produit que le symbole A_1, le codage est inutile car on connait d'avance le symbole produit par la source (pas d'incertitude).

Exemple 4 :

Si les symboles ont pour probabilité $\mathcal{P} = \{0.75, 0.125, 0.125\}$,

$H(X) = -0.75 \log_2 0.75 - 2.(0.125 \log_2 0.125) = 1.06$ bits.

Pour un codage, on ne peut pas utiliser moins de $(1, 2, 2)$ bits.

En conséquence, le plus petit codage possible est de $0.75 \times 1 + 2 \times 2 \times 0.125 = 1.25$ bits.

Exercice 10: Entropie d'une source

Soit un alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$. Trouver l'entropie des sources X_i dans les cas suivants :

1. X_1/ $\Pr[a_1] = \Pr[a_2] = \Pr[a_3] = \Pr[a_4] = \frac{1}{4}$.
2. X_2/ $\Pr[a_1] = \frac{1}{2}$, $\Pr[a_2] = \frac{1}{4}$, $\Pr[a_3] = \Pr[a_4] = \frac{1}{8}$.
3. X_3/ $\Pr[a_1] = 0,505$, $\Pr[a_2] = \frac{1}{4}$, $\Pr[a_3] = \frac{1}{8}$, $\Pr[a_4] = 0,12$.
4. X_4/ $\Pr[a_1] = \frac{3}{4}$, $\Pr[a_2] = \Pr[a_3] = \frac{1}{8}$, $\Pr[a_4] = 0$.
5. X_5/ $\Pr[a_1] = 1$, $\Pr[a_2] = \Pr[a_3] = \Pr[a_4] = 0$.

Exercice 11: Comparaison de l'entropie de deux sources

Considérons deux sources P et Q utilisant un alphabet de taille m dont les probabilités d'occurence des symboles sont respectivement $\{p_1, \ldots, p_m\}$ et $\{q_1, \ldots, q_m\}$.
Par ailleurs, on a $p_i = q_i$ pour tout i à l'exception de deux (notés u et v), et pour lesquels on a : $q_u = q_v = \frac{1}{2}(p_u + p_v)$.

1. $H(P)$ est-il moins grand, égal, ou plus grand que $H(Q)$? Justifier la réponse (utilise le fait que $-x \log x$ est une fonction concave).
2. même question avec $q_u = 0$ et $q_v = p_u + p_v$.

Lemme 6 (technique)**.** *Soit X et Y deux variables aléatoires sur le même nombre n de possibilités, et dont les probabilités d'occurrence respective de chaque symbole sont* $\{p_i\}_{i=1\ldots n}$ *et* $\{q_i\}_{i=1\ldots n}$*, alors :*

$$\sum_{i=1}^{n} p_i \log_2 q_i \leq \sum_{i=1}^{n} p_i \log_2 p_i$$

avec égalité ssi $p_i = q_i$*.*

Démonstration:

On a $\log_2 x . \log 2 = \ln x$. Or, $\ln x$ est une fonction concave qui est majorée par $x - 1$ avec égalité si $x = 1$ (*i.e.* $\ln x \leq x - 1$, voir figure). Par conséquent,

$$\ln \frac{q_i}{p_i} \leq \frac{q_i}{p_i} - 1$$

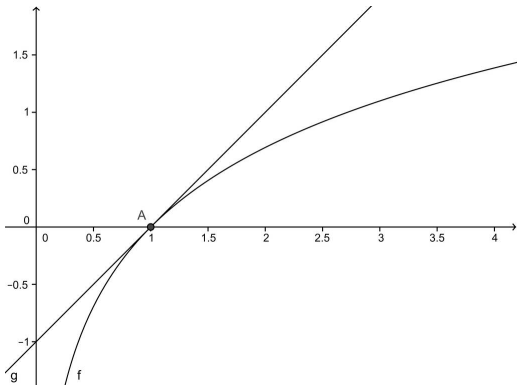


Figure : ln(x) et x+1.

D'où $\sum_{i=1}^{n} p_i \ln \frac{q_i}{p_i} \leq \sum_{i=1}^{n} p_i(\frac{q_i}{p_i} - 1) = \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i = 1 - 1 = 0.$

L'égalité est vérifiée ssi $q_i/p_i = 1$ pour tout i, c'est-à-dire si $p_i = q_i$. □

Théorème 7 (Maximum d'entropie)**.** *Pour tout variable aléatoire X, le maximum d'entropie est atteint lorsque X a une distribution uniforme (= équiprobable).*
A savoir, si X a n possibilités, la probabilité d'occurence de ses symboles est $p_i = 1/n$ *pour tout i.*

Démonstration:

On a $H(X) - \log_2 n = -\sum_{i=1}^{n} p_i \log_2 p_i + \sum_{i=1}^{n} p_i \log \frac{1}{n}$

D'après le lemme précédent, $H(X) - \log_2 n \leq 0$.

Donc le maximum est atteint pour $p_i = 1/n$ pour tout i, ce qui est une distribution uniforme. □

Remarquez qu'une distribution uniforme est une distribution pour laquelle aucun symbole n'est plus fréquent qu'un autre. L'incertitude sur le tirage d'un symbole suivant une telle distribution est donc totale.

On en déduit que :

- si l'entropie est nulle, le comportement est certain (tout symbole émis est certain).
  La certitude est totale.
- si l'entropie est maximale, les symboles sont équiprobables.
  L'incertitude est totale.

L'entropie est donc aussi un moyen de mesurer le degré d'incertitude d'une source.

Exemple

Considérons une variable aléatoire X tirant des symboles $\mathcal{S} = \{0, 1\}$ avec une distribution $(p, 1-p)$. La courbe ci-contre donne l'entropie $H(X)$ en fonction de la valeur de p.

L'entropie minimale est atteinte pour $p = 0$ ou 1.
L'entropie maximale est atteinte pour $p = 0.5$.

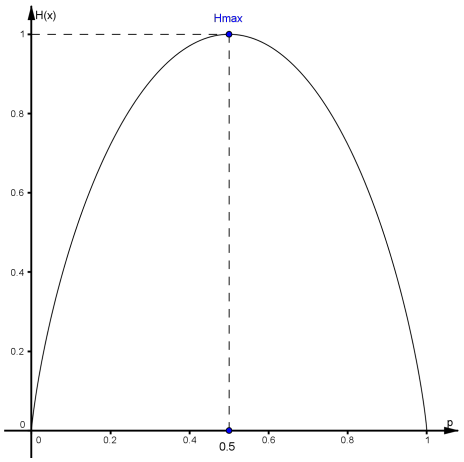


Figure : variation de l'entropie de X suivant la loi $(p, 1-p)$.

Exercice 12: Borne d'entropie d'une source aléatoire

Supposons que X est une variable aléatoire qui prend ses valeurs dans alphabet $\mathcal{A}$. Montrer que $0 \leq H(X) \leq \log_2 \#\mathcal{A}$.

Exercice 13: Inégalité de Jensen

Démontrer par récurrence que si que f est une fonction convexe sur A, alors $\forall(x_1, \ldots, x_n) \in A$, $\forall(\lambda_1, \ldots, \lambda_n) \in [0; 1]^n$ tel que $\sum_{i=1}^{n} \lambda_i = 1$, alors $f(\sum_{i=1}^{n} \lambda_i x_i) \leq \sum_{i=1}^{n} \lambda_i f(x_i)$.

5.4 Codage optimal

La théorie de l'information nous permet donc de donner une nouvelle formulation du problème du codage optimal dans le cas du codage d'une source avec un alphabet de taille n où chaque symbole s_i a une probabilité p_i.

On rappelle que la source émet des symboles indépendamment les uns des autres.

La recherche d'un codage optimal préfixe se formule de la manière suivante :

Trouver l'ensemble des longueurs $(\ell_1, \ell_2, \ldots \ell_n)$
- qui minimise la longueur moyenne $\overline{\ell} = \sum_{i=1}^{n} p_i \ell_i$,
- et tel que $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

Ce problème peut être résolu par l'analyse dans le cas d'une contrainte d'égalité, et conduit à :

$$p_i = 2^{-\ell_i^*} \Leftrightarrow \ell_i^* = -\log_2 p_i$$

Ce choix conduit à une longueur moyenne de code égale à :

$$\overline{\ell}^* = \sum p_i \ell_i^* = -\sum p_i \log_2 p_i = H(X)$$

Théorème 8 (Code optimal)**.** *Il existe un code optimal* $(\ell_1, \ell_2, \ldots, \ell_n)$ *pour un code X qui vérifie :*

$$H(X) \leq \overline{\ell}^* \leq H(x) + 1$$

La démonstration dépasse le cadre de cours (peut-être en TD).

On peut donc trouver un code optimal à au plus 1 bit de l'optimal donné par l'entropie.

Codage optimal = il n'est pas possible de trouver avec une représentation qui utilise moins de bits.

Donc, il s'agit de la meilleure compression possible.

Donc, le cours de compression est fini.

Ce n'est évidemment pas le cas : **pourquoi ?**

5.5 Limites

Réponse 1 :

les sources que nous avons considérées, et pour lesquels nous avons le résultat d'optimalité, produisent des chaines dont les caractères consécutifs sont indépendants.

A savoir le modèle suppose que chaque symbole émis par la source :
- ne doit son apparition seulement qu'en raison de sa fréquence propre,
- n'est influencé par aucun des caractères précédents émis par la source,
- n'a d'influence sur aucun des caractères suivants émis par la source

Ceci n'est évidemment pas le cas pour les sources usuelles d'information.

Ce modèle est purement théorique et ne reflète pas le comportement de la très grande majorité des sources réelles.

Il ne s'applique pas à la compression avec perte.

Réponse 2 :

Toute information présente généralement une part de redondance qui implique une dépendance "spatiale" ou "temporelle" (à savoir que les symboles suivants peuvent dépendre des symboles précédents).

Par exemple dans un texte :
- En terme de fréquence, un E est le plus souvent suivi d'un S, un N ou d'un L.
- En terme de sémantique, dans "la banane est bonne", l'article "la" et l'accord de "bon" sont des redondances au genre de "banane".
- En terme de codage, le codage du type `char` ne tient pas compte de la fréquence des caractères.

Réponse 3 :

Nous ne savons rien sur la façon de choisir l'alphabet afin garantir un codage efficace.

Par exemple dans un texte, pour le choix de l'alphabet faut-il utiliser :
- tous les différents caractères de l'alphabet Français,
- les digrammes (couples de lettres consécutives),
- les phonèmes,
- autre chose ?

La compréhension de la nature des données aide à les représenter sous une forme où la redondance d'information est clairement identifiable, et permettre ainsi la compression.

Exercice 14: Entropie d'une source

Soit une source aléatoire utilisant un alphabet $\mathcal{A} = \{a, b, c, d, e, f, g, h\}$ constitué de l'ensemble des nombres binaires sur 3 bits $\{000, 001, 010, 011, 100, 101, 110, 111\}$

1. On lit 1000 symboles issus de cette source. Le comptage des différents symboles nous donne les résultats suivants : $\{250, 62, 125, 124, 32, 250, 32, 125\}$. Calculer la fréquence f_i de chaque symbole s_i.
2. Y-a-t-il un lien entre f_i est la probabilité d'apparition du symbole s_i ?
3. Quel est la longueur moyenne du code pour ce comptage ?
4. Quelle est l'entropie de cette source ?
5. On décide de stocker dans des mots de taille 2 les mots les plus fréquents, et dans des mots de 4 les mots les moins fréquents. Comment l'inégalité de Kraft peut-elle m'aider à trouver le nombre maximum de mots de taille 2 que je peux choisir ?
6. Donner un exemple de code vérifiant les contraintes indiqués.
7. A partir de l'affectation des codes obtenus à la question précédente, calculer la longueur moyenne du code.
8. Proposer un autre codage tel que la longueur moyenne soit strictement inférieure à 3.
9. Comparer cette longueur moyenne et l'entropie du code.

6 Redondance

6.1 Exemples

Nous avons vu, avec la notion d'entropie, qu'une information ne peut pas être compressée plus que le nombre de bits nécessaires pour représenter l'information.

En conséquence, la compression a pour but d'éliminer :

- les bits qui ne servent pas à stocker de l'information.
  Exemple : n nombres de 0 à 255 sur 32 bits (24 bits perdus, 8 bits efficaces par nombre).

- les bits qui contiennent une information redondante.
  Exemple : n nombres de 1.234.567.000 à 1.234.567.255 (1.234.567.000 est redondant, donc stocker un entier 32 bits + n entiers 8 bits).

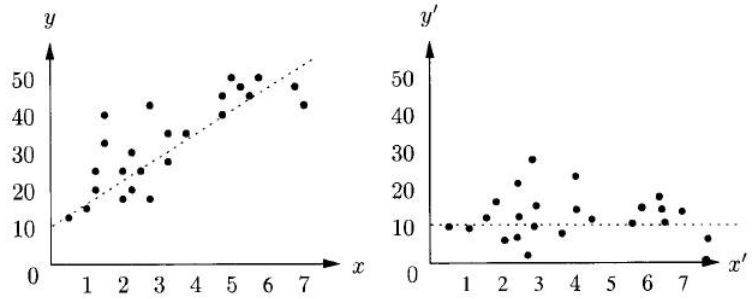
La nature de la redondance dans les données est l'un des facteurs essentiels permettant de concevoir un algorithme de compression efficace.

Exemples de redondance : (avec exemple de compression)

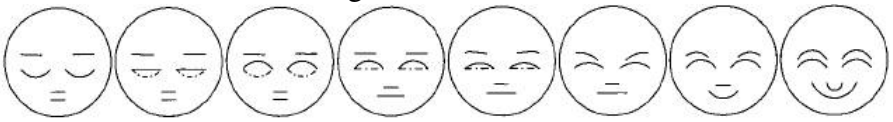
- `BAAAAAAAAAC` : redondance d'un caractère consécutif.
  compression : remplacer les 9 `A`s par `r9A`.
- `ABACAA` : redondance de caractères non consécutif.
  compression : utiliser pour `A` un code plus court.
- `ABCDABCEDABCEABC` : redondance du motif `ABC`.
  compression : utiliser un code pour `ABC`.
- vecteur = $(6, 428, 32, 67, 125)$
  compression : les valeurs sont dans l'intervalle $[6, 427]$ et peuvent être stocké sur 9 bits.

Exemples de redondance :

- **Cas d'un nuage de points :** dépendance 2D des données



  compression : équation droite (a, b) + 1 distance d à la droite par point.
- **Cas d'une vidéo :** les changements d'une frame à l'autre sont minimes.



  compression : redondance importante entre chaque frame, stockage d'une frame de référence, puis stockage de la différence par rapport à la frame précédente.

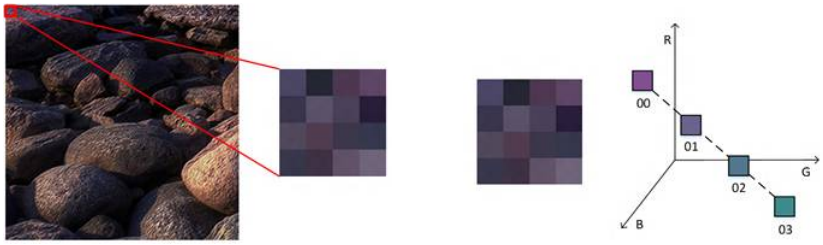
Pour la compression avec perte, on s'autorise en plus à avoir une représentation partielle des données (approximation raisonnable).

Exemples de redondance :

- vecteur = $(6, 428, 32, 67, 125)$
  approximation : en acceptant une approximation à ± 16, ce vecteur peut être représenté comme $(0, 13, 1, 2, 4) \in [0, 15]$ avec un facteur de mise à

l'échelle de 32, et peuvent être stockés sur 4 bits.

- image



approximation : utilise le fait que localement, les couleurs sont similaires. BC1 = 2 couleurs extrêmes (2 interpolés) + 16×2 bits (indice de la couleur parmi les 4 couleurs).

6.2 RLE (RunLength Encoding)

6.2.1 Principe

Le RLE (**R**un**L**ength **E**ncoding) est une méthode de compression simple permettant de donner un exemple d'élimination de la redondance.

"run" = répétition consécutive d'un seul et même caractère.
"runlength" = la longueur d'un "run"

Exemples de "run"s :

- `ABCKKKKKKKABC` : 1 run de `K` de longueur 7.
- `ABCDEFGFEDCBA` : aucun run.
- `ABCABCABCABC` : aucun run (la répétition du motif `ABC` n'est pas considéré comme un run).
- `ABBBCDDDDE` : 2 runs, un run de `B` de longueurs 3, et un run de `D` de longueur 4.

Idée du RLE : remplacer les "run"s par des chaines plus compactes.

Principe :

1. Un caractère spécial (par exemple=@) indique le début d'un run.
2. Un run de i caractères `X` est codé sous la forme @i`X`.

Application :

- `ABCKKKKKKKABC` ⇒ `ABC@7KABC`
- `ABCABCABCABC` ⇒ `ABCABCABCABC`
- `ABBBCDDDDE` ⇒ `A@3BC@4DE`

Remarques :

- Un run de 3 caractères n'est pas compressé : `XXX` ⇒ `@3X`.

- Un run de 2 caractères provoque une dilatation : `XX` ⇒ `@2X`.
- Il est inutile d'encoder un run de 3 caractères ou moins. Donc :
  - ni `XX`, ni `XXX` ne sont encodés.
  - le codage `@`i`X` pourra représenter un run de $(i + 4)$ `X`.
- dans BinHex 4.0, le caractère special c est utilisé de la manière suivante : $c0$ = le caractère c, ciX = un run de X de longueur i.

6.2.2 Codage

Il suffit de compter les caractères consécutifs identiques, et effectuer les sorties au fur et à mesure.

Algorithme de compression

```
RLEcompress(In, Out)
    n = 0
    p = ReadSymbol()
    while ( !EOF ) do
        q = ReadSymbol()
        if (q != p) then
            RLEoutput(n,p)
            n=0
        else
            n++
        p = q
    RLEoutput(n,p)
```

```
RLEoutput(n, p)
    if (n≤3) then
        for j=0 à n do
            output ← p
    else
        while n>0 do
            k = min(n,255)
            output ← @kp
            n-=255;
```

6.2.3 Décodage

Le décompression est triviale : les caractères lus sont répliqués sur la sortie, et les runs sont décompressés.

Algorithme de décompression

```
while ( !EOF ) do
    p = ReadCode();
    if (p == '@') then
        n = ReadCode();
        p = ReadCode();
        for j=1 à n do
            output ← p
    else
        output ← p
```

EXERCICE 15: RLE

On veut utiliser la méthode RLE pour stocker des grands nombres en utilisant des chiffres. Le symbole 00 représente le chiffre 0 et 056 représente un run de 6 de longueur 5. On ne compressera pas les runs de longueur inférieure ou égale à 3.

1. Peut-on trouver un manière non ambiguë de coder un run de 0 ?
2. Indiquer comment coder un run de 4 de longueur 24.
3. Coder la chaîne suivante : 1111123333333330201000000000344 ?
4. Quel est le taux de compression ?
5. Décoder le chaîne suivante codée avec RLE : 0414061003330922240514 44.
6. Existe-t-il des chaînes qui ne représentent pas une chaîne codée par RLE ? Si non, justifier, si oui donner un exemple.
7. On veut maintenant optimiser cette représentation RLE en utilisant le fait que les runs de longueur inférieure ou égale à 3 caractères ne sont pas compressés. Proposer cette représentation.
8. Donner un cas où l'on constate qu'elle est plus efficace.

6.3 RRR (recursive range reduction)

La méthode RRR (recursive range reduction) est une méthode de compression de l'échelle d'entiers proposée par Yann Guidon [Gui03].

Cette méthode est facile à implémenter, et sa performance ne dépend pas de la quantité de données à compresser.

Nous allons aborder cette méthode en deux étapes :

- tout d'abord avec la méthode RR (range réduction) qui permet de décrire le codage d'une liste d'entiers triés.
- puis cette méthode est étendue à tout ensemble d'entiers (méthode RRR).

On définit :

- MSB (Most Significant Bit) : bit de poids le plus fort.
- LSB (Least Significant Bit) : bit de poids le moins fort.

Avant de commencer, remarquons que pour tout ensemble d'entiers, il est possible de se débarrasser de leur signe :

- soit en faisant passer ce signe du MSB au LSB (shift circulaire à gauche).
  exemple : 10011 ⇒ 00111.
- soit en ajoutant à l'ensemble des entiers $\{x_i\}$ le plus petit de ces entiers ($x_{\min} = \min_i x_i$), ainsi la dynamique de cet ensemble passe de $[x_{\min}; x_{\max}]$ à $[0; x_{\max} - x_{\min}]$.
  exemple : $\{-4, -3, 0, 2, 6\} \Rightarrow \{0, 1, 4, 6, 10\}$

Par la suite, on supposera que les entiers des listes considérées sont toujours positifs.

6.3.1 Cas trié

Soit une suite d'entiers $\{x_i\}$, codés sur 2^n bits, triée par ordre décroissant.

Soit MSB(x_i) le numéro du bit non nul de poids le plus fort.
Exemple : MSB(00011101) = 4
MSB(0001) = 0

Soit Bits(x_i, q) les $q + 1$ derniers bits de x_i.
Exemple : Bits(00011101, 4) = 11101

La méthode RR consiste à coder une suite d'entiers de la façon suivante :

1. émettre MSB(x_1) sur 4 bits (= entiers jusqu'à 16 bits).
2. émettre Bits(x_1, MSB(x_1) − 1) (*i.e.* x_1 sans son MSB).
3. pour tous les x_i suivant
   si MSB(x_i) ≠ MSB(x_{i-1})
   alors émettre MSB(x_i) − MSB(x_{i-1}) fois 0.
   émettre Bits(x_i, MSB(x_i))

Exemple

Entiers	Code RR	
	0110	nombre de bits du MSB
1011101	011101	1er entier sans son MSB
1001011	1001011	MSBs
0110001	0110001	0 (perte 1 MSB) + MSBs
0101100	101100	MSBs
0001110	001110	0 (perte 2 MSB) + MSBs
0001101	1101	MSBs
0001100	1100	MSBs
0001001	1001	MSBs
0000010	0010	0 (perte 2 MSB) + MSBs
0000001	01	MSBs
70 bits	53 bits	

Remarques

- Les résultats les pires sont obtenus avec les listes qui décroissent :
  - ◇ soit très peu (= pas ou peu de perte de MSB)
    exemple : $\{900, 899, 898, 897\}$
  - ◇ soit rapidement (= peu de profit de la perte de MSB)
    exemple : $\{100000, 1000, 10, 1\}$.
- Comme l'entête n'est que de quelques bits, le codage RR ne dilate quasiment jamais les données.

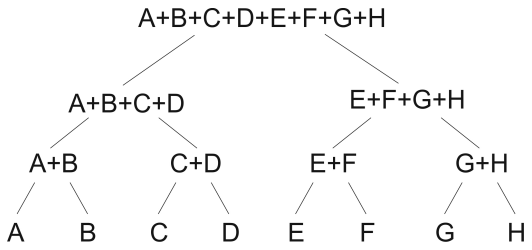
Comment faire maintenant avec des listes non triées ?

6.3.2 Principe

Dans le cas où la liste n'est pas triée, on construit un arbre avec :

- à ses feuilles, la liste non triée.
- à ses nœuds, la somme de toute la branche.

Exemple



Remarquons maintenant que sur un tel arbre, la descente vers une feuille constitue une liste triée.

Exemple

$A + B + C + D + E + F + G + H \geq A + B + C + D \geq A + B \geq A$

Remarquons également, que si on a n entiers $\{x_i\}$, il est possible de créer n suites partielles (dont l'une est $\sum_i x_i$) qui permettent de retrouver tous les entiers.

Exemple

Si je connais $A + B + C + D$, $A + B$, A et C, je peux reconstituer :

- $B = (A + B) - (A)$
- $D = (A + B + C + D) - (A + B) - (C)$

Pour reconstituer la totalité des valeurs de l'arbre précédent, je peux donc utiliser les sommes partielles suivantes :

$A + B + C + D + E + F + G + H$, $A + B + C + D$, $A + B$, A, C
$E + F$, E

G

En effet, en plus de B et D donné précédemment, on a :

- $E + F + G + H = (A + B + C + D + E + F + G + H) - (A + B + C + D)$
- $G + H = (E + F + G + H) - (E + F)$
- $F = (E + F) - E$
- $H = (G + H) - G$

Il est donc équivalent de stocker les 8 entiers ou les 8 sommes partielles indiquées.

Chacune des sommes partielles stockées peuvent donc être maintenant considérées comme faisant toutes parties de listes triées dont les éléments :

- font soit directement partie des sommes partielles stockées,
- soit peuvent être reconstitués à partir des autres sommes (en rouge dans l'exemple ci-dessous).

Exemple

Les listes triées associées à l'arbre précédent sont :

- $A + B + C + D + E + F + G + H$, $A + B + C + D$, $A + B$, A
- $A + B + C + D + E + F + G + H$, $A + B + C + D$, $\mathbf{C + D}$, C
- $A + B + C + D + E + F + G + H$, $\mathbf{E + F + G + H}$, $E + F$, E
- $A + B + C + D + E + F + G + H$, $\mathbf{E + F + G + H}$, $\mathbf{G + H}$, G

Ces listes triées sont maintenant codées en utilisant la méthode RR. Notons que ceci ne permet d'utiliser que le strict nombre de bits nécessaires.

Seuls sont conservés les codes des éléments nécessaires (on ne stocke donc pas plus d'entiers que le nombre initial d'entiers).

Au décodage, les élèments manquants des listes sont reconstitués (et codés) afin d'en déduire comment les élèments présents ont été codés.

L'un des avantages dans l'utilisation des sommes partielles, et que la plupart du temps, un phénomène de moyennage apparaît sur les sommes partielles aux nœuds, qui conduit à des valeurs souvent assez proches.

Ce codage n'apporte qu'assez peu de compression, mais il nécessite très peu de mémoire pour être construit, ce qui lui permet d'être utilisé par exemple sur des systèmes embarqués.

Yann Guidon fournit sur son site (voir [Gui03]), une démonstration interactive du codage, les codes sources de l'encodeur 3R, et des exemples d'applications.

EXERCICE 16: RRR

On souhaiter coder avec la méthode RRR la liste d'entiers suivant que l'on représentera sur 8 bits : $P = \{23, 19, 2, 12, 1, 13, 29, 9\}$

1. On veut tout d'abord stocker cette liste triée avec la méthode RR.
   a) Donner le codage RR de cette liste.
   b) Combien de bits fait le résultat de ce codage ?
   c) Calculer le taux de compression.
   d) Si on suppose que les chiffres 8 bits sont répartis uniformément sur tout l'intervalle. Quel est le gain espéré pour RR ?
2. On effectue maintenant la compression RRR. On veut coder la première moitié des chiffres.
   a) Construire l'arbre :
   b) Donner les liste triées associées à l'arbre associée à toutes les feuilles les plus à gauche du dernier noeud.
   c) Donner la liste des valeurs à stocker nécessaires pour la reconstruction.
   d) Pour les valeurs non présentes dans cette liste, indiquer comment reconstruire une valeur par niveau.
   e) Effectuer le codage RR.

6.4 Transformation

Les méthodes de codage présentées dans cette section n'ont pas pour but d'obtenir des codes plus courts.

En revanche, elles réorganisent les données (d'une façon inversible) afin d'essayer de faire apparaître la redondance de caractères ou de motifs.

Ces méthodes sont parfois employées comme première étape d'une phase de compression.

Si elles peuvent aider à la compression, ces transformations peuvent aussi perturber les dépendances spatiales entre symboles que le compresseur utilisé aurait peut-être pu exploiter.

Elles sont donc à utiliser avec beaucoup de précautions.

6.4.1 Burrows-Wheeler

La transformée de Burrows-Wheeler (BWT, voir [SMB10] page 1089) n'est pas une méthode de compression à proprement parler mais une méthode de **codage par bloc** effectuant des permutations cycliques dans le but de réorganiser les données d'une manière plus propice à leurs compressions.

Elle nécessite l'accès à la totalité des données pour pouvoir être utilisée (*i.e.* elle n'est pas applicable sur un flux de données continu).

L'idée de la BWT est la suivante : soit un suite $c_1 c_2 \dots c_n$ de longueur n

- construire $n - 1$ autres suites un chacun de ces suites est un décalage cyclique de la suite initiale.
  la $k^{\text{ème}}$ permutation est $c_k \dots c_n c_1 \dots c_{k-1}$.
- trier les n suites par ordre lexicographique.
- retenir la position p de la suite initiale dans l'ensemble des suites triées.
- utiliser pour le codage la dernière colonne de chaque suite dans l'ordre lexicographique.
  le tri permet de regrouper les caractères identiques, la dernière colonne contient les caractères qui les précède.

Exemple

Pour la chaine "papou à pou", on construit la totalité des décalages cycliques possibles avec cette chaine.

0	p	a	p	o	u		à		p	o	u
1	a	p	o	u		à		p	o	u	p
2	p	o	u		à		p	o	u	p	a
3	o	u		à		p	o	u	p	a	p
4	u		à		p	o	u	p	a	p	o
5		à		p	o	u	p	a	p	o	u
6	à		p	o	u	p	a	p	o	u	
7		p	o	u	p	a	p	o	u		à
8	p	o	u	p	a	p	o	u		à	
9	o	u	p	a	p	o	u		à		p
10	u	p	a	p	o	u		à		p	o

La chaîne étant de longueur 11, il y a 10 décalages cycliques possibles en plus de la chaîne originale. Puis on trie la table des décalages cycliques par ordre

lexicographique (= alphabétique, colonne par colonne).

position	indice											
0	5		à		p	o	u	p	a	p	o	u
1	7		p	o	u	p	a	p	o	u		à
2	1	a	p	o	u		à		p	o	u	p
3	6	à		p	o	u	p	a	p	o	u	
4	3	o	u		à		p	o	u	p	a	p
5	9	o	u	p	a	p	o	u		à		p
6	0	p	a	p	o	u		à		p	o	u
7	2	p	o	u		à		p	o	u	p	a
8	8	p	o	u	p	a	p	o	u		à	
9	4	u		à		p	o	u	p	a	p	o
10	10	u	p	a	p	o	u		à		p	o

La première colonne est le numéro d'ordre, la seconde est l'indice du numéro d'ordre du décalage cyclique avant le tri.

La chaîne passée au décodeur est "uàp_ppua_oo" et le numéro d'ordre de la chaine originale dans le tableau trié est 6.

On remarque que :

- la dernière colonne contient toujours le caractère qui précède la première colonne.
- cette transformation va donc regrouper ensemble les symboles dont les caractères qui suivent sont identiques.

Les résultats sont plus intéressants lorsque la chaîne que l'on transforme est plus longue.

Comment décoder ?

On reçoit comme information de la table des décalages cycliques triées :

- la dernière colonne L.
- le numéro p de la ligne contenant la chaîne originale.

Remarquons qu'avant le tri, la table des décalages cycliques est symétrique ($n^{\text{ème}}$ ligne = $n^{\text{ème}}$ colonne). En conséquence, chaque colonne contient exactement le même ensemble de caractères que chaque ligne. Cette propriété n'est pas modifiée par le tri.

En conséquence, la seule différence entre la première et dernière colonne est que la première est triée. Donc, on peut obtenir la première colonne F en triant la dernière colonne.

Comment décoder ? (suite)

On construit la table des couples t_i comme :
$t_i = (i, L_i)$
où L_i est la dernière $i^{\text{ème}}$ lettre de la dernière colonne.

Puis, on trie la suite $\{t_i\}_{i=1...n}$ d'abord suivant L_i, puis suivant i

L'algorithme de reconstruction de la chaine originale est le suivant :

Algorithme de décodage

Entrées :
$\{([i], L_{[i]})\}$ = le tableau t_i trié (suivant l'indice $L_{[i]}$ puis $[i]$).
Id = le numéro d'ordre de la chaîne originale dans le tableau trié.
Sortie :
S la chaîne décodée.

$p = Id$
for $i = 0$ *à* $n - 1$ **do**
  $S_i = L_{[p]}$
  p = [p]

Exemple de décodage

Chaine à décoder L ="uàp_ppua_oo" et numéro d'ordre de la chaine originale $Id = 6$.

Construction de la table t_i des couples :

i	0	1	2	3	4	5	6	7	8	9	10
L_i	u	à	p	_	p	p	u	a	_	o	o

Tri de la table (critère L_i puis i)

i	0	1	2	3	4	5	6	7	8	9	10
$[i]$	3	8	7	1	9	10	2	4	5	0	6
$L_{[i]}$	_	_	a	à	o	o	p	p	p	u	u

Algorithme

Itération		0	1	2	3	4	5	6	7	8	9	10
p	6	6	2	7	4	9	0	3	1	8	5	10
$S_i = L_{[p]}$		p	a	p	o	u	_	à	_	p	o	u
$[p]$		2	7	4	9	0	3	1	8	5	10	6

La chaîne est bien décodée comme "papou_à_pou".

Pourquoi ce décodage marche-t-il ?

On dispose de (i, L_i) et du numéro Id de la ligne contenant la chaîne originale dans la table des décalages cycliques.

On veut reconstruire la chaîne S originale.

- On peut construire $([i], L_{[i]})$ en triant la liste de couple.
- $\{L_{[i]}\}$ est la première colonne de la table des décalages cycliques.
- donc $L_{[i]}$ et L_i représentent respectivement la première et la dernière colonne de la $i^{\text{ème}}$ ligne de la table des décalages cycliques.
- on sait que la ligne contenant la chaîne originale a pour numéro Id, donc le premier caractère de S est $L_{[Id]}$.
- or, cette première lettre $L_{[Id]}$ était avant sur la ligne $[Id]$,
- or, sur la ligne $k = [Id]$, L_k est suivi par le caractère $L_{[k]}$,
- or, le caractère $L_{[k]}$ était avant sur la ligne $[k]$,
- etc ...

Les performances de ce codage s'améliorent avec des blocs plus longs, et lorsque des suites de 3 lettres consécutives se répètent relativement souvent.

Autre exemple de codage (avec un bloc de 265 caractères)

```
"Donc chez les papous, il y a des papous papas a poux papas,
des papous papas a poux pas papas, des papous pas papas a
poux papas et des papous pas papas a poux pas papas."
```

```
"yssssca,,ts,zssssxxsssssssssxxaaaalssss.     pppppppppppppp
pppppppppppn     ddddl hc i oDpppppppppp            aaaaa
  aaaaaaaa    aaaaaauuaaaeeeeeuuaauaeooooooooouuuu e"
```

et $Id = 58$.

En pratique

Il faut utiliser des blocs de taille importante (plusieurs milliers de caractères).

Lors de l'implémentation, la table de permutation n'est jamais construite, car chaque décalage cyclique peut facilement être engendré au vol (le $i^{\text{ème}}$ décalage cyclique est construit comme les i derniers caractères suivis des $n - i$ premiers, où n est la longueur de la chaîne).

Exercice 17: Transformée du Burrow-Wheeler

1. Effectuer la transformée de Burrow-Wheeler de 'patate'.
2. Décoder le mot transformé.

6.4.2 MTF (Move To Front)

Cette méthode de transformation (proposée par Bentley, voir [BSTW86]) consiste à essayer de maintenir une liste des symboles récemment utilisée afin de coder les symboles comme une position dans cette liste.

L'objectif est de faire en sorte que ces derniers codes aient un numéro d'ordre le plus petit possible. On espère ainsi en déduire la dynamique des numéros utilisés pour le codage.

Soit l'alphabet $\mathcal{A}$ des symboles, on définit les opérateurs suivants :

- $\mathcal{A}\langle x\rangle$ = indice de x dans l'ensemble $\mathcal{A}$.
- $\mathcal{A}[i]$ = symbole à la $i^{\text{ème}}$ place de l'ensemble $\mathcal{A}$.
- MTF$(\mathcal{A}, i)$ = retourne l'ensemble $\mathcal{A}$ en déplaçant l'élément de la $i^{\text{ème}}$ place à la première place, à savoir :
  $$\text{MTF}(\{x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n\}, i) = \{x_i, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\}$$

L'utilisation de codage est très simple : un caractère est codé comme étant sa position dans l'alphabet. Une fois le caractère codé, on le positionne en début d'alphabet et l'on recommence. Si le caractère réapparaît peu de temps après, il devrait avoir un code proche de 0.

Exemple : codage de `abcddcbamnopponm`

$\mathcal{A} = (a, b, c, d, m, n, o, p)$

in	out	alphabet	in	out	alphabet
a	0	abcdmnop	m	4	mabcdnop
b	1	bacdmnop	n	5	nmabcdop
c	2	cbadmnop	o	6	onmabcdp
d	3	dcbamnop	p	7	ponmabcd
d	0	dcbamnop	p	0	ponmabcd
c	1	cdbamnop	o	1	opnmabcd
b	2	bcdamnop	n	2	nopmabcd
a	3	abcdmnop	m	3	mnopabcd

code = $\{0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3\}$
La moyenne du code est 2,5.

Par simple codage direct (index dans $\mathcal{A}$), la moyenne du code est 3,5.

L'utilisation de codage est très simple. L'alphabet $\mathcal{A}$ doit être connu de l'encodeur et du decodeur, et dans le même ordre au début de l'algorithme (au besoin, le trier).

Codage

while *(!EOF)* **do**
  c =GetSymbol()
  $i = \mathcal{A}\langle c \rangle$
  $\mathcal{A}$ = MTF($\mathcal{A}, i$)
  output ← i

Pour le décodage, on décode le symbole et on modifie l'alphabet en miroir de l'encodeur.

Décodage

while *(!EOF)* **do**
  i = GetCode()
  output ← $\mathcal{A}[i]$
  $\mathcal{A}$ = MTF($\mathcal{A}, i$)

Ce codage doit être utilisé avec beaucoup de précaution car il est loin d'être toujours efficace.

Exemple

La chaine "abcdmnopabcdmnop" est codée comme :
code = $\{0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7\}$

La moyenne de code est 5.25, donc moins bon qu'avec un codage direct (moyenne 3.5).

Par simple analyse de l'algorithme, on voit que :

- ce code n'est efficace que lorsque un ou quelques symboles (et eux seulement) sont répétés sur des suites consécutives de caractères,
- une application préalable de BWT est souvent une bonne idée.
- les "performances" de ce codage deviennent très médiocres dès que l'on cycle sur plus que quelques symboles.
- il n'a pour but que de préparer les données en vue d'une compression ultérieure.

Variations

- **Move-ahead-k** : l'élément courant est remonté de k places dans l'alphabet. Réduit les performances, mais limite l'impact des caractères isolés.
- **Wait-c-and-move** : un élément est remonté devant seulement s'il a été rencontré c fois (pas nécessairement consécutive).
  Permet de fixer un seuil pour limiter le déplacement des caractères isolés. A noter que le comptage s'effectue y compris sur des caractères non consécutifs. La fenêtre de comptage peut être glissante.

Remarque

Dans l'exemple donné pour lequel MTF est moins performant qu'un codage direct, aucune des deux variations ci-dessus ne réussit à battre le codage direct.

Exercice 18: Move-to-front

On considère le mot "how much wood would a woodchuck chuck".

1. Donner l'alphabet de cette chaine.
2. Donner le code utilisant les numéros d'index dans l'alphabet.
3. Donner le code en utilisant la méthode "Move-to-Front".
4. Quel est l'intérêt de la chaîne obtenue par rapport à l'index simple ?
5. Décoder le code obtenu avec la méthode "Move-to-Front".

7 Conclusion

Fin du cours d'introduction sur le cadre général de la compression.

Pour les notions que nous verrons dans cette première partie du cours, les références sont :

- pour les méthodes de compression, le livre de Salomon [SMB10].
- pour la théorie de l'information, le livre de Cover et Thomas [CT06].
- pour les probabilités de base, le livre de Papoulis [Pap91].

Les prochains cours seront consacrés à la découverte des différentes familles des méthodes de compression sans perte.

- le codage entropique (basé sur la probabilité d'apparition des symboles),
- le codage basé sur dictionnaire (basé sur l'indexation et la répétition de motif),
- le codage prédictif (basé sur les probabilités contextuelles des symboles)