Information, Calcul et Communication: Cours 11

Résumé de la dernière leçon: compression de données

$$\begin{array}{ccc} \text{séquence } \mathcal{X} & \xrightarrow{\text{dictionnaire}} & \text{code binaire sans préfixe } \mathcal{C} \\ \updownarrow & & \updownarrow \\ \text{entropie } H(\mathcal{X}) & \underset{\text{théorème de Shannon}}{\leq} & \text{longueur moyenne } L(\mathcal{C}) \end{array}$$

Plan de cette leçon: correction d'erreurs

- codage par répétition
- bits de partité
- codes de Reed-Solomon

Correction d'erreurs

Une fois accepté l'idée qu'il est possible de transformer un signal physique en une suite de 0 et de 1 (et réciproquement) on serait tenté de se dire que l'essentiel est fait, puisque cette suite de 0 et de 1 pourra être enregistrée, traitée si besoin (p.ex. compressée ou cryptée) et transmise plus loin, sans qu'il soit besoin de rien faire de plus.
Or il n'en est rien! Des erreurs surviennent très fréquemment.
p.ex : . lors de l'écriture ou de la lecture de données;
. lors de la transmission de celles-ci, que ce soit par câble électrique, fibre optique ou onde électromagnétique

Or autant le bruit présent dans un signal analogique, s'il est certes une source de désagrément, n'est pas une catastrophe en soi, autant une erreur dans un signal numérique peut se révéler bien plus ennuyeuse.

Exemple :

Supposons que vous désiriez communiquer la direction à prendre (N, S, E, W) à un ami perdu au milieu d'une forêt, et que vous utilisiez pour cela le code binaire suivant (4 possibilités $\rightarrow 2 = \log_2(4)$ bits)

N $\rightarrow$ 11 , S $\rightarrow$ 10 , E $\rightarrow$ 01 , W $\rightarrow$ 00

Pour dire à votre ami d'aller au nord, vous envoyez donc 11.

1) Effacement

Si votre ami reçait le message "1?" (i.e., le 2e bit est perdu ou effacé), celui-ci ne saura pas s'il faut aller au nord ou au sud.

2) Erreur

Pire, si votre ami reçoit le message "10" (i.e., le 2e bit est faussement converti de 1 à 0), votre ami partira au sud, confiant que c'est la bonne direction.

Pour remédier à ce problème et prévenir ce genre d'erreurs, il faut ajouter de la redondance au message envoyé.

??? On vient de passer un chapitre à expliquer comment compresser de l'information en supprimant la redondance présente dans les données, et maintenant on propose d'en rajouter ???

Oui; l'idée ici est d'ajouter un minimum de redondance tout en prévenant un maximum d'erreurs ; avec des données brutes (i.e., non compressées), on n'a aucun contrôle sur le type de redondance présente dans celles-ci.

Algorithmes de correction d'erreurs dans la vie de tous les jours :

- Un père à ses enfants : (codage par répétition)
  "Mettez vos chaussures ... mettez vos chaussures, j'ai dit!"

- Vous à des amis :
  "Pour venir chez moi, c'est simple : tu tournes dans la 2e rue à droite après le feu rouge, juste après la station service ; j'habite au numéro 3, dans l'immeuble bleu avec les grands balcons." Les informations soulignées en rouge sont redondantes (donc inutiles, en théorie...)

1) Corriger des effacements (en commençant par 1 effacement)

Une manière simple de se protéger de l'effacement d'1 bit est le codage par répétition : on répète chaque bit 2 fois :

N → 1111, S → 1100, E → 0011, W → 0000

Ainsi, si votre ami reçoit maintenant le message "11?1", il saura qu'il faut aller au nord.

Cette méthode a l'avantage d'être simple, mais aussi le gros désavantage d'être coûteuse : il faut envoyer 4 bits au lieu de 2. Une solution plus économe serait :

N → 110, S → 101, E → 011, W → 000

qui corrige aussi un effacement (p.ex. si on reçoit "1?0" → N)

Généralisons l'exemple précédent et supposons qu'on désire envoyer 1 message parmi 2^n messages possibles.
Sans redondance, on a besoin de n bits pour ça.
Pour corriger 1 effacement éventuel en utilisant le codage par répétition, on aura besoin de 2n bits, car chaque bit est doublé. Très peu efficace!
On peut faire beaucoup mieux en ajoutant un simple bit de parité au message : celui-ci vaut 1 si la somme des bits du message est impaire et 0 sinon.
Ex : message = 01011010 → somme = 4 → bit de parité = 0 ($n = 8$) → message envoyé : 010110100 9 bits

Si on reçoit maintenant 010?10100, alors en regardant le bit de parité à la fin, on voit qu'il faut que le ? soit remplacé par la valeur 1 pour que la somme des huit premiers bits soit paire (et si le dernier bit de parité est effacé : 01011010?, ça n'est pas grave). Ainsi, on a utilisé n+1 bits au lieu de 2n avec le codage par répétition, pour aboutir au même résultat. Cette méthode est beaucoup plus efficace !

Comment corriger k effacements ? (avec $k \geq 2$)

Avec le codage par répétition, on a besoin de répéter $k+1$ fois chaque bit (pour gérer le pire des cas où un même bit est effacé k fois). A nouveau, c'est un système très peu efficace !

La théorie du codage étudie comment ajouter des bits de parité de façon optimale à un message pour corriger le plus d'effacements possible en ajoutant le moins de bits de parité possible au message. C'est tout un art ! Nous allons y revenir ...

2) Corriger des erreurs (en commençant par 1 erreur)

Reprenons notre exemple: N → 11, S → 10, E → 01, W → 00

Doubler chaque bit pour corriger une erreur ne fonctionne pas ici. P.ex., si on reçoit "1101", on ne sait pas alors faut partir au nord (1111) ou au sud (1100).

Une possibilité qui permet de corriger une erreur est de répéter 3 fois chaque bit:

N → 111111, S → 111000, E → 000111, W → 000000

Si on reçoit "111101", alors on comprend qu'il faut aller au nord en appliquant la règle de la majorité pour chaque série de 3 bits: 111 → 1, 101 → 1

Mais à nouveau, cette solution est (très) coûteuse : chaque bit est triplé ; on aura donc besoin de $3n$ bits pour envoyer un message de longueur n à l'origine.

Une meilleure méthode pour corriger 1 erreur ?

Lorsque $n = 4$: on ajoute 2 bits de parité

- le premier indique la parité de la somme des bits 1 et 2
- le deuxième indique la parité de la somme des bits 1 et 3

(cela vous rappelle-t-il quelque chose ?)

Exemple : Pour envoyer le message 1101,

on envoie 110101.

Supposons qu'on reçoive le message "100101". Qu'en déduit-on ?

- bits 1 et 2 : $1+0=1 \neq 0$ ✗
- bits 1 et 3 : $1+0=1$ ✓

} c'est donc le bit 2 qui est faux : le message d'origine est 1101

Oui, mais ce système de codage a un problème. En effet, si aucun des 2 bits de parité n'indique une erreur, on n'a aucun moyen de savoir si le 4e bit du message est juste ou non.

Pour bien faire les choses, on a besoin ici de 3 bits de parité : c'est le code de Hamming que vous verrez aux exercices.

Ajouter 3 bits de parité pour corriger une seule erreur sur un message de 4 bits peut sembler peu efficace, mais en généralisant le principe, on peut corriger 1 erreur sur un message de n bits en ajoutant au plus $\log_2 n + 1$ bits ; bien plus efficace que de tripler la longueur du message !

Et que faire pour corriger plusieurs erreurs ?

A nouveau tout l'art de la théorie du codage est nécessaire. (On y vient !)

Les codes de Reed-Solomon

Deux personnes (disons A et B) cherchent à communiquer, mais une proportion non-négligeable des informations transmises par A sont effacées/bruitées avant d'être reçues par B. Les codes de Reed-Solomon sont un bon moyen de gérer une telle situation. Pour simplifier leur description, nous allons supposer que A et B travaillent avec des moyens de communication capables de manipuler directement des nombres réels (et non des bits).

Avant toute chose, il importe que A et B se mettent d'accord sur un ensemble de nombres $t_1, \dots, t_n \in \mathbb{R}$ pour communiquer (cf. dictionnaire pour la compression de données). On suppose de plus que tous ces nombres sont différents.

A désire maintenant envoyer un message qui est une suite de nombres $x_1, \dots, x_k \in \mathbb{R}$, avec $k \leq n$. Pour ce faire, A définit le polynôme suivant:

$$P(t) = \sum_{i=1}^{k} x_i \, t^{i-1}$$

Notez que P est un polynôme de degré $k-1$.

A envoie ensuite le message $y_1, \dots, y_n$ construit ainsi :

$$y_1 = P(t_1), \dots, y_n = P(t_n)$$

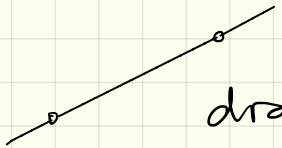
Supposons maintenant que $n-k$ de ces nombres soient effacés lors de la transmission. B ne reçoit donc que k d'entre eux (peu importe lesquels, mais disons les k premiers pour simplifier les notations).

Le but de B est de retrouver les nombres $x_1, \dots, x_k$ à partir des nombres $y_1, \dots, y_k$ reçus. On sait que

$$y_j = P(t_j) = \sum_{i=1}^{k} x_i t_j^{i-1} \quad \text{pour } 1 \leq j \leq k$$

Il s'agit ici de résoudre un système linéaire, mais on sait qu'il admet une solution unique, car il existe un unique polynôme P de degré $k-1$ passant par k points différents :

$k=2$:



droite (deg 1)

$k=3$:

parabole (deg 2)

Et donc B sera capable à coup sûr de retrouver le message envoyé par A. Avec cette technique de codage, on a donc réussi à corriger beaucoup d'effacements ! On peut de la même façon corriger beaucoup d'erreurs (mais un peu moins que d'effacements).

En pratique:

- On ne travaille pas avec des nombres réels, mais avec des nombres entiers modulo p (où p est un nombre premier) : $\{0, 1, 2, \ldots, p-1\}$, ou plus généralement, des nombres appartenant à un groupe fini. Les mêmes principes concernant les polynômes s'appliquent dans ce cadre.
- Vu leur efficacité, les codes de Reed-Solomon ont été implémentés dans pratiquement tous les supports numériques existants pour protéger ceux-ci d'erreurs de lecture et d'écriture.