TP0: Compression et codage arithmétique

Objectifs du TP:

- Rappeler les principes et connaître les limites de la compression de données par l'algorithme de Huffman
- Introduire au codage arithmétique et à la notion d'entropie de Shannon
- Connaître les limites des nombres flottants et être capable d'utiliser des nombres à virgules fixes

L'algorithme de Huffman est un algorithme efficace de compression de données que vous avez vu en première année. Pour autant, il n'est en pratique jamais utilisé seul parce qu'il n'est pas très optimal. Des algorithmes comme DEFLATE (la compression utilisée pour les fichiers `.zip` et `.png`) le combine avec LZ77. D'autres formats plus récents, que ce soit pour les vidéos (H.264, AV1) ou les images (JPEG) utilisent tous à un moment de leur compression une variante du codage arithmétique.

Partie 1 *Prémisses du codage arithmétique*

L'objectif du codage arithmétique est d'être plus proche que le codage de Huffman de l'optimum théorique définie par l'entropie de Shannon. En effet, l'algorithme de Huffman est limité en tant qu'il ne peut pas coder un symbole en moins d'un bit. Cette contrainte est embêtante puisque l'entropie n'est pas quasiment jamais un entier. Voilà pourquoi le codage arithmétique essaie de faire fi de cette limite…

On sépare un intervalle $[0; 1[$ selon la probabilité qu'a un caractère d'apparaître dans le texte. Cette probabilité est le nombre d'occurrences divisé par la taille du texte, comme pour Huffman. Ainsi, pour le mot $m =$ « PARTAGER » on obtient:

Lettre	Occurences	Borne inférieure	Borne supérieure
A	2	0	2/8
E	1	2/8	3/8
G	1	3/8	4/8
P	1	4/8	5/8
R	2	5/8	7/8
T	1	7/8	1

Tableau 1. – Occurences pour « PARTAGER »

La compression se passe ainsi:

- On note l'intervalle courant $[a; b[$, initialement $a = 0$ et $b = 1$
- Tant qu'on n'a pas fini de lire le texte:
  - On lit une lettre l dans le texte
  - On restreint l'intervalle selon le tableau généré précédemment, c'est à dire que si b_l et b_h sont respectivement les bornes inférieures et supérieures liées à la lettre l, on change a et b comme suit:
    - $a \leftarrow a + b_l \times (b - a)$
    - $b \leftarrow a + b_h \times (b - a)$

Par exemple, pour compresser « PARTAGE »:

- On part de $[0; 1[$
- On lit la lettre "P". L'intervalle devient alors $[\frac{1}{2}; \frac{5}{8}[$

- On lit la lettre “A”. L’intervalle devient alors $[\frac{1}{2}; \frac{17}{32}[$
- On lit la lettre “R”. L’intervalle devient alors $[\frac{133}{256}; \frac{135}{256}[$
- et ainsi de suite…

À la fin de l’exécution de l’algorithme, on obtient un intervalle $[a; b[$. Il faut ensuite choisir un nombre compris dans cet intervalle qui se code avec le moins de bits possible. Ce code est notre message compressé.

1. On pose le type suivant:

   ```
   struct intervalle_s {
     double inf;
     double sup;
   }
   typedef struct intervalle_s intervalle_t*;
   ```

   Écrire une fonction `intervalle_t* generer_table(char* texte)` qui génère un tableau d’intervalles en fonction du texte donné. Le tableau aura une taille de 256 (pour chaque caractère pouvant se trouver dans le texte)

2. Tester le code avec les entrées « TEST » et « PARTAGE » et vérifier qu’on a bien

   ```
   TEST:
   E: 0.000000 0.250000
   S: 0.250000 0.500000
   T: 0.500000 1.000000
   ```

   ```
   PARTAGE:
   A: 0.000000 0.285714
   E: 0.285714 0.428571
   G: 0.428571 0.571429
   P: 0.571429 0.714286
   R: 0.714286 0.857143
   T: 0.857143 1.000000
   ```

3. Implémenter la fonction `intervalle_t encodage(char* texte, intervalle_t* tab)` qui renvoie l’intervalle auquel correspond le message à encoder.

4. Tester pour « TEST » et « PARTAGE » et vérifier qu’on obtient bien (à 6 chiffres significatifs) $[0,554688; 0,578125[$ et $[0,652718; 0,652806[$ respectivement

5. Écrire une fonction `int ecriture(intervalle_t interv)` qui affiche dans le terminal le nombre à virgule en binaire avec le moins de bits possibles qui soit dans l’intervalle (on écrira directement les caractères `1` et `0` !) et qui renvoie le nombre de bits au total. Par exemple, pour $[\frac{3}{32}, \frac{1}{4}[$, la fonction doit afficher `001` car

   $$0 \times \frac{1}{2} + 0 \times \frac{1}{4} + 1 \times \frac{1}{8} = \frac{1}{8}$$

   et car c’est le nombre avec le moins de bits qui soit dans cet intervalle.

6. Écrire une fonction `double ratio_compression(char* texte)` qui compresse le texte donné à l’aide de ce qui a été implémenté précédemment, et qui renvoie le nombre de bits dans le message compressé divisé par le nombre de bits dans `texte`.

7. Montrer que la taille du message compressé dans le pire cas du codage arithmétique et dans le pire cas du codage de Huffman sont identiques.

   L’entropie de Shannon est définie par (où p_l est la probabilité d’une lettre l):

   $$H(m) = -\sum_{\substack{l \in \Sigma \\ p_l \neq 0}} p_l \log_2(p_l)$$

Complément C'est la moyenne pondérée de l'information que nous donne chaque événement (l'apparition d'une lettre dans un texte).

Supposons qu'on ne connaisse pas la forme de $H(X)$. Pour la réinventer, on sait qu'on veut que:

- L'information que nous donne un événement lorsqu'il arrive tende vers $+\infty$ plus l'événement est rare ($p \to 0$), et donc que $I(p) \underset{p\to 0}{\to} +\infty$
- Un événement de probabilité 1 ne donne aucune information lorsqu'il arrive donc $I(1) = 0$
- $I(p_1 p_2) = I(p_1) + I(p_2)$ car deux événements indépendants donne comme information la somme des deux

On retrouve bien $-\log_2$ comme fonction adaptée pour exprimer $I(x)$, d'où

$$\begin{aligned} H(X) &= \sum_{x\in X} p(x) I(x) \\ &= -\sum_{x\in X} p(x) \log(p(x)) \end{aligned}$$

8. Implémenter une fonction `double entropie(char* texte)` qui calcule l'entropie d'un texte donné.
9. La quantité d'information contenue dans un texte t est défini comme étant $|t|\ H(X)$. Calculer cette quantité d'information pour plusieurs exemples et comparer au nombre de bits après compression par la fonction `encodage`. Que se passe-t-il ? Proposer des explications.
10. Comparer pour divers messages (d'au plus 16 lettres), le nombre de bits du message compressé par le codage arithmétique avec l'entropie de ce message. Conclure.

Bonus

11. Montrer que, dans le cas où les probabilités sont de la forme $\frac{1}{2^k}$, la compression arithmétique et le codage de Huffman produisent exactement la même chose.
12. Montrer que, si on compresse un mot m de longueur n et qu'on note $c(m)$ le mot compressé, alors

$$\lim_{n\to+\infty} \frac{|c(m)|}{|m|} \geq H(m)$$

Partie 2 *Le problème de la précision des flottants*

1. Avec des flottants de 64 bits (1 bit de signe, 53 bits de mantisse, 11 bits d'exposant), par combien doit on majorer n tel qu'on puisse calculer l'intervalle d'un message de taille 2^n sans pertes de précision liées aux flottants avec le codage arithmétique ?

   On se concentre maintenant sur des messages de taille 2^n avec $n \in \mathbb{N}$.

   Contrairement aux nombres à virgule flottante, les nombres à virgule fixe on, comme leur nom l'indique, un nombre de chiffre après la virgule qui est fixe. Ils sont composés d'une partie entière de $k \in \mathbb{N}$ bits et d'une partie fractionnaire de $p \in \mathbb{N}$ bits.

   Par exemple, avec $k = 2, p = 2$, on peut représenter 3.25 de la manière suivante:

2^1	2^0	2^{-1}	2^{-2}
1	1	0	1

2. Proposer une manière d'adapter la partie précédente à l'utilisation de nombre à virgules fixes, en donnant des valeurs intéressantes pour k et p (en sachant qu'on veut utiliser des `int` comme types pour stocker les nombres à virgule fixe)
3. Combien de bits suffisent pour représenter toutes les probabilités pour un message de taille 2^n ?

   L'addition et la soustraction de nombres à virgule fixe se fait de l'exacte même manière que pour les entiers (les opérateurs `+` et `-` du C fonctionnent comme attendu). En revanche, si on s'aventure à la même chose pour la multiplication, on risque d'être surpris…

   En reprenant $k = 2, p = 2$, prenons le cas de 0.5 et 1:

2^1	2^0	2^{-1}	2^{-2}
0	0	1	0

Tableau 3. – 0.5

2^1	2^0	2^{-1}	2^{-2}
0	1	0	0

Tableau 4. – 1

   En appliquant la même multiplication que pour les entiers, on obtiendrait alors que $0.5 \times 1 = 2$. Il faut en fait décaler le résultat de k bits vers la droite. On obtient bien

$$\begin{aligned} 0.5 \times 1 &= 2 \gg 2 \\ &= 0.5 \end{aligned}$$

4. Reprenant les paramètres k et p trouvés en question 2. Que se passe-t-il si on multiplie 0.5 par 0.5 avec des `int` ? Que faire pour éviter ce problème ? Implémenter une fonction `int fp_multiply(int a, int b)` qui multiplie deux nombres.
5. Réimplémenter `generer_table` et `encodage` avec les nombres à virgule fixe décrits dans cette partie.
6. A-t-on gagné par rapport aux flottants ?