TD #20 — Compression (avec solutions)

Exercice 1. *Stop*

On propose la technique suivante pour construire un code $\mathcal{C} : \Sigma_1 \to \Sigma_2^*$ de longueur variable, pour une fonction de coût f :

— On choisit un caractère $s \in \Sigma_2$ quelconque qui servira à indiquer la fin des mots de code.
— On trie les lettres $c \in \Sigma_1$ par coût f décroissant, et l'on définit le code de la $i$ème lettre (en partant de zéro) dans cette énumération comme $\mathbb{I} \cdot s$, où $\mathbb{I}$ est l'écriture de i en base $\#\Sigma_2 - 1$ en utilisant les lettres de $\Sigma_2 \setminus \{s\}$ comme chiffres.

1. Donnez le code $\mathcal{C}_1$ obtenu avec la technique ci-dessus pour $\Sigma_1 = \{a, b, c, d, e\}$, de coûts $f(a) = 7$, $f(b) = 1$, $f(c) = 2$, $f(d) = 3$, $f(e) = 9$, $\Sigma_2 = \{0, 1\}$, et en utilisant $s = 0$.

On obtient :

— $\mathcal{C}_1(a) = 10$
— $\mathcal{C}_1(b) = 11110$
— $\mathcal{C}_1(c) = 1110$
— $\mathcal{C}_1(d) = 110$
— $\mathcal{C}_1(e) = 0$

2. Donnez un code $\mathcal{C}_2$ obtenu pour les mêmes alphabets & coût par l'algorithme « de Huffman ».

On obtient (par exemple) :

— $\mathcal{C}_2(a) = 10$
— $\mathcal{C}_2(b) = 1110$
— $\mathcal{C}_2(c) = 1111$
— $\mathcal{C}_2(d) = 110$
— $\mathcal{C}_2(e) = 0$

C'est *presque* le même code, mais il est strictement meilleur puisque les longueurs d'encodages de a, b, d, e sont égales à celles de $\mathcal{C}_1$, mais celle de c est inférieure de un.

3. Mêmes questions pour $f(a) = 3$, $f(b) = 2$, $f(c) = 2$, $f(d) = 3$, $f(e) = 5$.

Le code $\mathcal{C}_1$ ne change pas nécessairement, tandis que $\mathcal{C}_2$ est cette fois assez différent, par exemple :

— $\mathcal{C}_2(a) = 00$
— $\mathcal{C}_2(b) = 110$
— $\mathcal{C}_2(c) = 111$
— $\mathcal{C}_2(d) = 01$
— $\mathcal{C}_2(e) = 10$

Exercice 2. *Code suffixe*

Soit un code $\mathcal{C} : \Sigma_1 \to \Sigma_2^*$ (supposé injectif), on dit qu'il est *suffixe* s'il n'existe pas de $c_1, c_2 \in \Sigma_1$ tels que $\mathcal{C}(c_1) = p \cdot \mathcal{C}(c_2)$ (autrement dit tels que l'encodage de c_2 est un suffixe de l'encodage de c_1).

1. Montrez que le code $\mathcal{C}$ étendu aux mots sur Σ_1 comme $\mathcal{C}(m) = \mathcal{C}(m_0 m_1 \cdots m_{\ell-1}) = \mathcal{C}(m_0)\mathcal{C}(m_1)\cdots\mathcal{C}(m_{\ell-1})$ est décodable de façon unique (est inversible).

Le miroir d'un code suffixe étant un code préfixe, on a que le miroir de $\mathcal{C}(m)$ est égal à $\mathcal{C}'(m_{\ell-1})\cdots\mathcal{C}'(m_1)\mathcal{C}'(m_0)$ que l'on peut donc décoder de façon unique.
Plus directement, on peut par exemple montrer l'injectivité de $\mathcal{C}$ étendu aux mots en utilisant un raisonnement par l'absurde (on suppose un plus petit mot pouvant se décoder de deux façons différentes, et déduit une contradiction du fait qu'il possède un préfixe strict qui doit lui aussi se décoder de deux façons différentes), et exhiber un algorithme de décompression qui n'échoue jamais : il suffit de lire les lettres de $\mathcal{C}(m)$ de la fin vers le début et les concaténer (à « gauche ») dans une variable a initialisée au mot vide ε, et chaque fois que celle-ci est égale

à un certain mot de code $\mathcal{C}(c_x)$ de décoder le caractère $c_x \in \Sigma_1$, réinitialiser a à ε et continuer avec la suite de $\mathcal{C}(m)$. Le code étant suffixe, si $a = \mathcal{C}(c_x)$ alors lire une éventuelle nouvelle lettre y et la concaténer à a donnerait $y \cdot a = y \cdot \mathcal{C}(c_x)$ qui n'est égal à aucun code et n'est suffixe d'aucun code non plus : l'unique façon de décoder a est donc en c_x.

2. Donnez un exemple simple de construction de code (injectif) qui soit à la fois préfixe et suffixe.

Un code dont tous les mots sont de même longueur est trivialement à la fois préfixe et suffixe : soit $c_1, c_2 \neq c_1$, l'on a $\mathcal{C}(c_1) \neq \mathcal{C}(c_2)$ et aucun des mots dont $\mathcal{C}(c_2)$ est un facteur non trivial n'est un mot de code puisqu'ils sont trop longs pour cela.

3. En quoi un code préfixe peut être préférable à un code suffixe pour le décodage du code étendu aux mots ?

Le décodage d'un code suffixe étendu aux mots peut en général nécessiter de lire $\mathcal{C}(m)$ « à l'envers » ou de d'abord en calculer le miroir. Lorsque $\mathcal{C}(m)$ doit être lu dans un flux, la première opération n'est pas possible et la seconde nécessite de d'abord lire **tout** le flux et le stocker avant de pouvoir seulement décoder la première lettre. Un code préfixe (qui permet le décodage « à la volée » sur un flux) est donc préférable dans ce contexte (courant).

Exercice 3. *Compression répétée*

On suppose vouloir compresser un fichier en utilisant l'algorithme « de Huffman » pour construire un code $\mathcal{C} : [\![256]\!] \to \{0,1\}^*$ (autrement dit, on souhaite encoder les octets comme des mots binaires (de taille *a priori* variable)) et ensuite encoder chaque octet x du fichier par son code $\mathcal{C}(x)$.

1. Donnez le taux de compression maximal possible (le rapport entre la taille du fichier initial et celle du fichier compressé).

Chaque octet (8 bits) apparaissant dans le fichier est encodé sur au moins 1 bit : le taux de compression maximal possible est donc de 8.

2. Donnez un exemple de contenu de fichier pour lequel celui-ci est atteint.

N'importe quel fichier dont les octets prennent au plus deux valeurs verra ceux-ci encodés sur 1 bit par l'algorithme « de Huffman », et atteindra donc le taux de compression maximal (si la description du code est stockée séparément, ou asymptotiquement si celle-ci est incluse dans le fichier compressé).

3. Montrez que la compression répétée (avec l'algorithme ci-dessus) d'un fichier contenant un unique octet peut **toujours** produire un fichier d'*un seul* octet (en supposant que la description du code est stockée séparément du fichier compressé). On pourra ignorer toute subtilité (notamment de *padding*) liée à l'encodage d'un mot binaire comme suite d'octets (on peut par exemple supposer que le fichier initial est de taille une puissance de deux et que l'encodage utilisé stocke séparément le nombre total de bits).

Dans le cas particulier où tous les octets du fichier prennent une **unique** valeur, on peut sans perte de généralité supposer que celle-ci sera encodée en 0, et le fichier compressé une fois contiendra uniquement des octets eux-mêmes constitués de huit 0 (donc identiques). On peut donc à nouveau compresser ce fichier pour obtenir un fichier compressé (deux fois) dont tous les octets sont identiques, et procéder ainsi jusqu'à obtenir un fichier d'un seul octet.

4. Pourquoi la question précédente n'implique-t'elle pas que l'on puisse atteindre un taux de compression (asymptotiquement) infini ?

Pour pouvoir retrouver le fichier original depuis le fichier compressé de multiple fois, il faut se souvenir de combien de fois il a été compressé. Cette information prend certes très peu de place par rapport au fichier initial (dans notre cas, soit n la taille de ce dernier il faut représenter le nombre $\approx \log_8 n$, ce qui peut se faire avec un $O(\log \log n)$ bits) mais elle n'est bornée par aucune constante. Le taux de compression obtenu n'est donc pas asymptotiquement infini (seulement doublement exponentiel…)

5. Pourquoi une telle compression répétée risque-t'elle d'être « peu intéressante » pour un fichier quelconque ?

Le contexte des questions précédentes exploite la présence répétée de motifs de plusieurs lettres dans le fichier original. Ceci n'a pas de raison d'être le cas en général, mais surtout l'encodage *en octets* (dans le fichier compressé) d'un même motif n'a pas de raison d'être constant : le code construit par l'algorithme « de Huffman » étant (en général)

de longueur variable, une même suite de bits peut se trouver *alignée* différemment par rapport aux « frontières » des octets. Ainsi, des motifs fréquents de plusieurs lettres n'impliquent pas forcément la présence d'octets fréquents dans le fichier compressé (qui est la seule chose qu'une compression répétée peut exploiter).

Exercice 4. *Représentation arborescente d'un code préfixe binaire*

On rappelle qu'une valeur t de :

```
type code_tree = E | L of int |
                 N of code_tree * code_tree
```

dont toutes les feuilles sont d'étiquettes distinctes représente un code préfixe binaire $\mathcal{C} : \mathcal{S} \to \{0,1\}$ (pour $\mathcal{S} \subset \mathbb{N}$) de la façon suivante : le code de x est donné par la numérotation binaire dans t de la feuille d'étiquette x.

1. Écrivez une fonction OCaml :
   `code2tree : (int * int list) list -> code_tree`
   qui construit un arbre représentant le code (préfixe) donné en argument sous la forme d'une liste de couples `(x, cx)` où `x` représente une lettre $\in \mathcal{S}$ de façon naturelle et `cx` son code sous la forme d'une liste de `0` et `1`.

   On propose :

```
let rec add_code_to_tree t c
    = match c, t with
    | (x, [0]), N (E, t') -> N (L x, t')
    | (x, [0]), E -> N (L x, E)
    | (x, [1]), N (t', E) -> N (t', L x)
    | (x, [1]), E -> N (E, L x)
    | (x, 0::c'), N (t1, t2) -> N (add_code_to_tree t1 (x, c'), t2)
    | (x, 0::c'), E -> N (add_code_to_tree E (x, c'), E)
    | (x, 1::c'), N (t1, t2) -> N (t1, add_code_to_tree t2 (x, c'))
    | (x, 1::c'), E -> N (E, add_code_to_tree E (x, c'))
    | _ -> failwith "not a prefix code"

let code2tree code_list = List.fold_left add_code_to_tree E code_list
```

Exercice 5. *LZW sans tableau associatif*

On rappelle que l'algorithme de compression LZW utilise une fonction partielle $\mathcal{T} : [\![n]\!] \to \Sigma^*$ pour certains n & Σ, et que cette fonction doit pouvoir être efficacement inversible (en particulier, on doit pouvoir efficacement déterminer si $m \in \Sigma^*$ possède un antécédent pour $\mathcal{T}$).

1. Expliquez comment $\mathcal{T}$ et sont inverse $\mathcal{T}^{-1}$peuvent être représentés en utilisant une structure de donnée de tableau (éventuellement associatif).

   En utilisant des types standard OCaml pour illustration, où un mot $\in \Sigma^*$ est représenté par une `'a list`, on peut représenter $\mathcal{T}$ par un « bête » tableau `(t : 'a list array)` (où `t.(i)` contient l'image de $\mathcal{T}$ en l'entier i), et $\mathcal{T}^{-1}$ par une table de hachage `(tinv : ('a list, int) Hashtbl.t)` (dont les clefs sont les mots de Σ^* sur lesquels $\mathcal{T}^{-1}$ est définie et les valeurs les images associées).

2. Quelle « structure » possèdent les images de $\mathcal{T}$? Notamment, soit $m \in \Sigma^*$ de longueur > 1, que peut on déduire du fait que c'est une image de $\mathcal{T}$?

   Lors de la compression d'un flux par LZW, un motif m ne peut être « ajouté » (comme image) à $\mathcal{T}$ que si son préfixe de longueur $|m|-1$ en est déjà une. On a donc notamment que si m est image de $\mathcal{T}$, alors tous ses préfixes le sont.

On suppose ci-dessous que les lettres de Σ sont représentés par des `int`.

3. Expliquez comment la structure mise en évidence à la question précédente permet de représenter un motif $m \in \Sigma^*$ par un type (par exemple) `type pat = P of pat option * int`.

Les motifs correspondant aux lettres de Σ (les cas de base) peuvent être représentés par des valeurs `P (None, x)` avec `x` les représentations de ces lettres, et un motif $m \cdot x$ par `P (Some m, x)` avec `m` (récursivement) une représentation de m.

4. Déduire de ce qui précède comment un tableau à deux dimensions $n \times \#\Sigma$ peut être utilisé pour calculer l'antécédent d'un motif par $\mathcal{T}$ en temps linéaire en sa longueur.

Soit `tinv` un tel tableau, et `p` un motif de code (d'image par $\mathcal{T}$) `c`, il suffit pour tout motif `p'` de la forme `P (Some p, x)` de code `c'` de définir `tinv.(c).(x)` comme `c'`. Les cas de bases sont donnés pour les motifs de la forme `P (None, x)` dont les codes sont (typiquement) définis comme `x` (ou si ce n'est pas le cas, stockés séparément), et l'on peut compléter `tinv` par des valeurs quelconques (par exemple `-1`) pour les motifs non définis.
Le code associé à un motif représenté par le type `pat` peut alors se calculer simplement comme :

```
let rec pat2code tinv
    = function
    | P (None,  x) -> x
    | P (Some p, x) -> try tinv.(pat2code tinv p).(x) with _ -> -1
```

On peut remarquer que si elle est efficace et se passe d'une structure de donnée avancée comme un tableau associatif, l'approche utilisée ici n'est pas forcément économe en mémoire, nécessitant un espace $\approx n \times \#\Sigma$, tandis qu'une table de hachage utiliserait un $O(n \times \ell)$ avec ℓ la longueur moyenne des motifs (qui peut être un $o(\#\Sigma)$).