Algorithmique du « texte » #2

Pierre Karpman

Lycée Champollion MP2I
https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Lecture/écriture de fichier

Objectif

Lire ou écrire depuis ou vers un fichier depuis un programme OCaml ou C

- en mode texte : lire / écrire des (chaînes de) caractères
- en mode binaire : lire / écrire des octets 

Processus de lecture typique

- On *ouvre* un fichier **en lecture**
  - Grâce à une fonction dédiée qui crée un **flux** (ou *canal*) (en anglais : *stream* ou *channel*) de lecture associé à un fichier dont on fournit un chemin
- On consomme des données du flux
  - on doit lire les symboles dans l'ordre (comme une liste, pas comme un tableau)
  - une fois un symbole lu, on ne peut pas le relire
- Une fois terminé, on ferme le flux

Écriture de fichier

Processus d'écriture typique

- On *ouvre* un fichier **en écriture**
  - Grâce à une fonction dédiée qui crée un **flux** (ou *canal*) (en anglais : *stream* ou *channel*) d'écriture associé à un fichier dont on fournit un chemin
  - Si un fichier existait déjà à ce chemin, son contenu est par défaut **supprimé**
- On écrit des données dans le flux
  - on doit écrire les symboles dans l'ordre
  - une fois un symbole écrit, on ne peut pas le modifier
- Une fois terminé, on ferme le flux
  - si l'on oublie de le faire, il se peut qu'*une partie ou totalité des données écrites dans le flux soient perdues*

Lecture/écriture de fichier en OCaml

Fonctions au programme

- `open_in : string -> in_channel` : crée un flux **en lecture** en « mode texte » associé au chemin donné en argument, qui doit être celui d'un fichier texte
- `open_out : string -> out_channel` : crée un flux **en écriture** en « mode texte » associé au chemin donné en argument (le fichier est créé si besoin, et écrasé si existant)
- `close_in : in_channel -> unit` : ferme un flux en lecture
- `close_out : out_channel -> unit` : ferme un flux en écriture
- `input_line : in_channel -> string` : lit la prochaine *ligne* (chaîne de caractères se terminant par un retour à la ligne) du flux
- `output_string : out_channel -> string -> unit` : écrit son second argument dans le premier

Lecture/écriture de fichier en OCaml *bis*

Exception au programme

- `End_of_file` : levée en cas de lecture dans un flux (maintenant) vide

Fonctions hors-programme 🙈

(Mais pratiques pour le TP)

- `in_channel_length : in_channel -> int` : s'évalue en la longueur (en nombre de caractères) du fichier associé au flux
- `open_in_bin : string -> in_channel` : crée un flux en lecture en « mode binaire »
- `open_out_bin : string -> out_channel` : crée un flux en écriture en « mode binaire »
- `input_byte : in_channel -> int` : lit le prochain octet (huit bits) du flux
- `output_byte : out_channel -> int -> unit` : écrit son second argument **modulo 256** dans le premier

Exemple de lecture en OCaml : un `cat` simple

```
let cat args =
    Array.iteri (fun i v -> if i > 0 then
        let istream = open_in v in
        let rec loop () =
            input_line istream |> print_string ;
            print_newline () ;
            loop ()
        in
        try loop () with End_of_file -> close_in istream)
    args
```

Exemple d'écriture en OCaml : un `echo` simple

```
let echo args =
    let ostream = Stdlib.open_out "echo.out" in
    Array.iteri (fun i v -> if i > 0 then (
        output_string ostream v ;
        output_string ostream "\n"))
    args ;
    close_out ostream
```

Lecture/écriture de fichier en C

Fonctions au programme

- `FILE *fopen(char *path, char *mode)`
  - `path` : chemin du fichier à ouvrir
  - `mode` : pour nous : `"r"` pour une ouverture en lecture, `"w"` en écriture (écrase l'éventuel fichier existant)
  - valeur de retour : un `FILE *` qui peut être utilisé dans les fonctions de lecture/écriture/fermeture
- `int fclose(FILE *stream)`
  - ferme le flux en argument
- `fprintf`, `fscanf`
  - comme `printf` et `scanf`, mais prennent un `FILE *` en premier argument vers ou depuis lequel écrire ou lire
  - `fscanf` renvoie `EOF` si la fin du fichier est atteinte

Arguments d'exécutables

Objectif

Être capable de traiter les arguments fournis à un programme exécutable (dans le cas d'OCaml, par exemple compilé avec `ocamlopt`)

Principe général

Les arguments sont fournis comme un tableau de chaînes de caractères, dont la première est le nom avec lequel le programme a été invoqué

- Il faut éventuellement convertir ces chaînes, par exemple vers un type numérique

En C

Il faut utiliser une fonction `main` de signature :

```
int main(int argc, char **argv)
```

ou compatible

- Le nombre d'arguments fournis plus un est contenu dans `argc`
- Le `i`ème argument (en comptant à partir de 1) est dans `argv[i]`

En OCaml

Les arguments sont contenus dans `Sys.argv`

Exemple en OCaml

```
let bin_name_from_arg0 s =
    String.split_on_char '/' s |> (* h.p. *)
    List.rev |> List.hd

let main () =
    let args = Sys.argv in
    match bin_name_from_arg0 args.(0) with
    | "cat" -> cat args
    | "echo" -> echo args
    | _ -> failwith "can only echo or cat"

;;

main ()
```

Exemple en OCaml *bis*

busybox trick : le programme exécuté dépend du nom avec lequel l'exécutable est invoqué

Compilation / usage

- Utilisation de *liens symboliques* pour pouvoir invoquer le même exécutable sous plusieurs noms

```
> ocamlopt -o cat_echo cat_echo.ml
> ln -s cat_echo cat
> ln -s cat_echo echo
```

Démo

Table des matières

Un problème de compression

Énoncé informel

Étant donnée une suite de symboles, quelle est la façon la plus économe d'encoder celle-ci en binaire *en encodant chaque symbole séparément* ?

Une (presque) formalisation

Soit un alphabet Σ, **et** $m \in \Sigma^*$, on veut calculer un *encodage* (ou *code*) $C : \Sigma \to \{0,1\}^*$ des lettres de Σ vers les mots binaires tel que :

- On puisse retrouver m à partir de $\mathcal{C}(m) := C(m_0)C(m_1)\cdots$ et la connaissance de C
- $|\mathcal{C}(m)|$ soit « le plus petit possible »

Avant de compresser

- Proposez une solution simple à ce problème qui ignore la seconde contrainte
- Estimez $|\mathcal{C}(m)|$ pour cette solution (en fonction de $|m|$ et $\#\Sigma$)

Encodage de taille fixe

Une solution simple mais généralement pas optimale

- Soit $n = \lceil \log \#\Sigma \rceil$, on encode chaque $x \in \Sigma$ par son rang écrit en base deux **sur** n **bits** dans une énumération de Σ pour un ordre quelconque
- $|C(x)| = n$ pour tout x : décodage facile
- $|\mathcal{C}(m)| = n \times |m|$: généralement pas minimal

Petite optimisation

- L'encodage étant **adaptatif** (dépend de m), on encode seulement les symboles de Σ qui apparaissent dans m
- (Mais s'il y en a plus de la moitié, on ne gagne rien)

Encodage de taille variable

Variabilité

- Les symboles de Σ **ne sont pas tous encodés sur le même nombre de bits**
- On peut encoder les symboles fréquents sur **moins de bits** que les symboles rares

Problèmes à résoudre :

- Comment garantir la possibilité du décodage ?
- Comment trouver de « bons » encodages ?

Code préfixe

Un encodage $C : \Sigma_1 \to \Sigma_2^*$ est dit *préfixe* si il n'existe aucun $c_1, c_2 \in \Sigma$ tels que $C(c_1)$ est un préfixe de $C(c_2)$

Proposition

L'encodage $\mathcal{C} : \Sigma_1^* \to \Sigma_2^*$ associé à un encodage préfixe C est inversible (connaissant C)

- On lit les symboles de $\mathcal{C}(m)$ en séquence et renvoie $x \in \Sigma_1$ dès que ceux-ci correspondent à $C(x)$

Représentation arborescente d'un code préfixe

On peut représenter un code préfixe $\Sigma \to \{0,1\}^*$ par des chemins dans un arbre binaire dont les feuilles sont étiquetées par des symboles de Σ et les nœuds internes sont non étiquetés

- Le code d'un symbole est donné par la numérotation binaire **suffixe** de la feuille (unique) dont il est l'étiquette
  - Rappel : la racine est de numéro ε ; l'enfant gauche (resp. droit) d'un nœud de numéro ν est de numéro $\nu \cdot 0$ (resp. $\nu \cdot 1$)
- Une feuille n'a pas d'enfant : soit ν le code associé, il n'existe aucun code de la forme $\nu \cdot s$: ν n'est préfixe d'aucun code
- Réciproquement, tout code préfixe peut être représenté par un arbre, éventuellement non strict (admis)

Code préfixe optimal

Soit $f : \Sigma \to \mathbb{Q}^+$ une fonction de pondération pour les symboles de Σ, un code préfixe optimal pour f est un code C minimisant :

$$\sum_{c \in \Sigma} |C(c)| \times f(c)$$

- Si f dénote les fréquences d'apparition des symboles de Σ dans un message m et soit σ la somme ci-dessus, alors $|\mathcal{C}(m)| = |m| \times \sigma$
- Minimiser σ pour les fréquences d'apparition correspond exactement à minimiser $|\mathcal{C}(m)|$

Vers un code préfixe optimal

- Intuitivement : plus un symbole est fréquent, plus son code doit être court
- Si trois symboles c_1, c_2, c_3 sont de fréquence f_1, f_2 et f_3 avec $f_1 + f_2 < f_3$, il est rentable d'augmenter la longueur des codes de c_1 et c_2 de 1 bit si cela permet de diminuer celle du code de c_3 de 1 bit

Algorithme « de Huffman »

L'algorithme « de Huffman » construit une représentation arborescente d'un code préfixe optimal de façon gloutonne

Algorithme de construction

- On initialise une file de priorité min avec $\{f(c), \ell_c\}$
  - ℓ_c : feuille d'étiquette c
- Puis tant qu'elle contient au moins deux éléments, on extrait les deux éléments (f_1, t_1), (f_2, t_2) de priorité minimum et l'on réinsert $(f_1 + f_2, \langle t_1, t_2 \rangle)$
  - On « fusionne » les symboles et récurse sur un alphabet plus petit
- Le résultat est donné par l'arbre contenu dans la file quand elle n'a plus qu'un seul élément

Exemple

Preuve d'optimalité

On montre par récurrence forte sur $\#\Sigma$ que le code construit par l'algorithme « H » (« de Huffman ») est optimal. On pose $\mathcal{P}(n)$: les codes construits pour $\#\Sigma \leq n$ sont optimaux

- Cas de base ($\#\Sigma \in \{1, 2\}$) : okay
- Conservation ($\mathcal{P}(n-1) \Rightarrow \mathcal{P}(n)$) : on suppose par l'absurde :(l'existence d'un code pour n symboles strictement meilleur que celui d'arbre t_H de coût σ_H construit par H. Soit σ_M son coût, t_M son arbre, c_1, c_2 les symboles de coûts f_1, f_2 minimaux
  - On peut supposer que c_1 et c_2 sont des adelphes de profondeur maximale dans t_M
  - Si l'on fusionne c_1 et c_2 en un unique caractère c_{12} de coût $f_1 + f_2$ et remplace le sous-arbre $\langle \ell_1, \ell_2 \rangle$ par ℓ_{12} dans t_M et t_H, **on obtient des codes sur Σ' de $n-1$ symboles de coût $\sigma_M - (f_1 + f_2)$ et $\sigma_H - (f_1 + f_2)$**
  - L'arbre obtenu à partir de t_M est de coût strictement inférieur à celui obtenu à partir de t_H
  - Mais ce dernier est exactement celui construit par H pour Σ', ce qui par hypothèse de récurrence donne une contradiction

Table des matières

Un autre problème de compression

Énoncé informel

Étant donnée une suite de symboles, comment peut-on prendre en compte les motifs courants de répétition pour compresser la suite ?

Contrainte

On souhaite compresser & décompresser un **flux**

- On ne peut pas se permettre de lire le flux une première fois afin d'apprendre les motifs courants

(Ceci permet d'adapter ce que l'on considère être un motif courant au fil de la compression)

Algorithme « de Lempel-Ziv-Welch »

Principe

- On encode des mots de Σ^* par des codes de **taille fixe** $\in \{0,1\}^d$
  - Par exemple pour $\#\Sigma = 256$, par des codes de 12 bits
  - Pas de difficulté pour lire un code
- On stocke l'association entre code et mot dans un (bête) tableau t : le mot associé à s se trouve à l'indice de t dont s est l'écriture en base deux
- On initialise t avec Σ, et l'on rajoute au fur et à mesure les mots que l'on rencontre dans le flux qui ne sont pas encore dedans
- On fait cela d'une façon telle que t n'a pas besoin d'être fourni pour la décompression : il est reconstruit à la volée

LZW : compression

Algorithme de compression

- On initialise $t[c] = c$ pour $c \in \Sigma$
- On initialise une variable m représentant le motif en train d'être lu à ε
- Tant que le flux n'est pas vide, on lit un symbole x et :
  - Si $m \cdot x \in t$ (en tant qu'image), on pose $m := m \cdot x$ et l'on continue
  - Sinon on émet le code de m (l'indice c de t t.q. $t[c] = m$) ; si t n'est pas plein on ajoute $m \cdot x$ dans t ; on pose $m := x$
- On émet le code de m si $m \neq \varepsilon$

On peut par exemple utiliser un tableau associatif pour tester efficacement si un motif est présent dans t

Exemple

LZW : décompression

Problématique

- On doit reconstruire t à la volée
- Tous les codes ajoutés dans t lors de la compression **ne sont pas forcément émis** (tout de suite)
- Il ne faut rien oublier

Intuition

- On sait que lors de la compression (hors fin de transmission), quand un code pour m est émis, on vient de lire un symbole x (qui sera le premier du prochain motif encodé) **et l'on a ajouté le code** $m \cdot x$ **dans** t
- Quand on décompresse, on mémorise chaque motif m qui vient d'être décompressé et quand on décompresse le motif $x \cdot m'$ suivant on ajoute $m \cdot x$ dans t

LZW : décompression *bis*

Une subtilité

- L'intuition ci-dessus fonctionne mais elle a « un temps de retard » : pour ajouter le code de $m \cdot x$ (déterminé à l'étape i de la compression), il faut avoir décodé l'étape $i+1$!
- Il se peut que l'étape $i+1$ utilise précisément le code ajouté à l'étape i !
- Mais dans ce cas cela veut dire que l'on a encodé un motif de la forme $m \cdot (x \cdot m')$ avec $x \cdot m' = m \cdot x$, donc x est la première lettre de m et le décodage reste possible

Exemple

Compression & décompression de *ABABABA*

LZW : décompression *ter*

Algorithme de décompression

- On initialise $t[c] = c$ pour $c \in \Sigma$
- On initialise une variable m représentant le dernier motif décodé à ε
- On initialise une variable n à $\#\Sigma$
- Tant que le flux n'est pas vide, on lit un code c et :
  - Si $c = n$: soit x la première lettre de $m \neq \varepsilon$ on définit $t[n] := m \cdot x$
  - Sinon soit x la première lettre de $t[c]$ on définit $t[n] := m \cdot x$
  - On incrémente n, affecte $m := t[c]$ et émet la même valeur

LZW : remarques

- Contrairement à l'algorithme « de Huffman », LZW ne minimise aucun critère précis
  - Les garanties théoriques sont dures à exprimer
- Une implémentation naïve ne compressera pas forcément mieux les textes en langue naturelle que l'algorithme « de Huffman »
- Beaucoup de stratégies sont possibles à l'implémentation, en particulier quand le tableau est plein
  - On peut repartir de zéro
  - On peut garder les codes les plus utilisés et oublier les autres
  - On peut augmenter la taille du tableau pour émettre des codes plus gros
- La reconstruction à la volée reste possible tant que les règles sont déterministes