Algorithmique du texte

On appelle texte toute suite finie de caractères, c'est-à-dire ce qu'on a appelé "mot" jusqu'à présent. L'algorithmique du texte consiste à résoudre des problèmes sur des textes qui peuvent en réalité modéliser des informations diverses : textes réels, séquences d'ADN, musique, images...

L'algorithmique du texte présente des applications dans la résolution de nombreux problèmes en informatique, par exemple :

- les recherches de similarité (plus long sous-mot commun, plus long facteur commun, distance d'édition et recherche d'alignements...)
- la recherche de motifs (penser au $\mathsf{Ctrl} + \mathsf{F}$ dans un éditeur de texte ou un navigateur)
- la compression de texte (encodage des caractères sur un nombre variable de bits, encodage par facteurs...)

Dans ce chapitre, nous traiterons et résoudrons quelques uns de ces problèmes.

Rappel (*alphabets, mots et concaténation*) :

On rappelle qu'un alphabet est un ensemble de fini et non vide de caractères. Pour Σ un alphabet, on définit l'ensemble des mots sur Σ et l'ensemble des mots *non vides* sur Σ, notés respectivement Σ^* et Σ^+, par :

$$\cdot\ \Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$$
$$\cdot\ \Sigma^+ = \bigcup_{n\in\mathbb{N}^*} \Sigma^n = \Sigma^* \backslash \{\varepsilon\}$$

où ε désigne le mot vide, unique élément de Σ^0.

On définit aussi l'opération de concaténation sur Σ^*, notée "$\cdot$" :

$$\forall \left((u_i)_{i\in[1..n]}, (v_j)_{j\in[1..m]}\right) \in (\Sigma^*)^2,\ u\cdot v = (w_k)_{k\in[1..n+m]} \text{ où } \begin{cases} \forall\, k \in [1..n],\ w_k = u_k \\ \forall\, k \in [n+1..n+m],\ w_k = v_{k-n} \end{cases}$$

Alors, $(\Sigma^*, \cdot)$ est un monoïde (*i.e.* la loi $\cdot$ est interne associative), de neutre ε.

Rappel (*sous-mots, facteurs, préfixes et suffixes*) :

Soit $(u, v) \in (\Sigma^*)^2$, on note $n = |u|$ et $m = |v|$. Alors :

- u est un sous-mot de v ssi il existe une application $\varphi \in \mathcal{F}([1..n], [1..m])$ strictement croissante telle que $u = (v_{\varphi(i)})_{i\in[1..n]}$
- u est un facteur de v ssi il existe $i \in [1..m-n]$ tel que $u = (v_{i+k})_{k\in[1..n]}$
- u est un préfixe de v ssi il existe $w \in \Sigma^*$ tel que $v = u\cdot w$
- u est un suffixe de v ssi il existe $w \in \Sigma^*$ tel que $v = w\cdot u$.

Remarque : De façon alternative, $u \in \Sigma^*$ est un facteur de $v \in \Sigma^*$ ssi il existe $(i, j) \in [1..|u|]^2$ tel que $u = (v_k)_{k\in[i..j]}$. En particulier, ε est toujours un facteur de v.

1 Plus long facteur commun

On considère le problème Plus long facteur commun, que l'on abrège en PLFC :

$$\textbf{PLFC} \left\| \begin{array}{l} \textbf{entrée :}\ u \in \Sigma^* \text{ de longueur } n \\ \qquad\quad\ v \in \Sigma^* \text{ de longueur } m \\ \textbf{sortie :}\ (\arg)\max \left\{ |f| \;\middle|\; f \in \Sigma^*, \begin{array}{l} \exists\,(i,j) \in [1..n]^2,\ f = (u_k)_{k\in[i..j]} \\ \exists\,(p,q) \in [1..m]^2,\ f = (v_k)_{k\in[p..q]} \end{array} \right\} \end{array} \right.$$

On en propose une résolution par programmation dynamique.
Soit $(u, v) \in (\Sigma^*)^2$. Notons $n = |u|$ et $m = |v|$.

Pour $(i, j) \in [0..n] \times [0..m]$, on pose :

$$A_{i,j} = \max\{|s| \,|\, s \text{ est un suffixe de } u_1...u_i \text{ et } v_1...v_j\} \leq \min(i, j)$$

La propriété qui suit donne alors une façon de résoudre le problème à partir de ces sous-problèmes.

Propriétés :

On a : ***i.*** $A_{0,0} = 0$
ii. $\forall\, i \in [0..n],\ A_{i,0} = 0$
iii. $\forall\, j \in [0..m],\ A_{0,j} = 0$
iv. $\forall\,(i, j) \in [1..n] \times [1..m],\ A_{i,j} = \begin{cases} A_{i-1,j-1} + 1 \text{ si } u_i = v_j \\ 0 \text{ sinon} \end{cases}$

De plus, $\mathsf{PLFC}(u, v) = \max\limits_{(i,j)\in[0..n]\times[0..m]} A_{i,j}$.

2 Recherche de motifs

On s'intéresse à présent à la recherche de motifs dans un texte. Plus précisément, on veut obtenir toutes les occurences d'un motif donné dans un texte donné, ce qui peut se formaliser comme suit.

$$\textbf{RDM} \left\| \begin{array}{l} \textbf{entrée :}\ t \in \Sigma^* \text{ un texte de longueur } n \\ \qquad\quad\ x \in \Sigma^* \text{ un motif de longueur } m \\ \textbf{sortie :}\ \left\{ i \in [1..n - m + 1] \,|\, (t_{i+k})_{k\in[0..m-1]} = x \right\} \end{array} \right.$$

Remarquons que dans cette formalisation du problème, on obtient en sortie de $\mathsf{RDM}(t, x)$ l'ensemble des indices de *début* des occurences de x dans t.

2.1 Algorithme naïf

Une première approche possible pour sa résolution est celle naïve que propose l'algorithme suivant.

Algorithme – Motif_naïf

entrée : $(t_i)_{i\in[1..n]} \in \Sigma^*$ et $(x_i)_{i\in[1..m]} \in \Sigma^*$

```
i = 1
Tant que i ≤ n − m + 1
    j = 0
    Tant que (j < m et t_{i+j} = x_{j+1})
        j ← j + 1
    Si j = m alors
        afficher i
    i ← i + 1
```

Motif_naïf est facilement en $\Theta(m(n-m))$, soit encore $\Theta(nm)$ quand $|x| \ll |t|$. Cependant, on peut faire mieux : c'est précisément l'objet de la prochaine propriété, qui introduit un principe reposant sur la représentation des entiers en base b et permettant d'écrire des algorithmes plus efficaces.

2.2 Algorithmes de Rabin-Karp

Propriété :

Soit $\Sigma = [0..b[$. Pour $w \in \Sigma^m$ et $(g,d) \in \Sigma^2$, on a :

- $\mathrm{val}_b(g \cdot w) = gb^m + \mathrm{val}_b(w)$
- $\mathrm{val}_b(w \cdot d) = b\,\mathrm{val}_b(w) + d = b(\mathrm{val}_b(gw) - gb^m) + d$

Algorithme – Rabin-Karp

entrée : $t \in \Sigma^*$ où $|\Sigma| = b$
$x \in \Sigma^*$ avec $|x| \leq |t|$

$n, m = |t|, |x|$
$c_x, c_f, b_m = 0, 0, 1$
Pour i allant de 1 à m
  $c_x \leftarrow (c_x * b) + x_i$
  $c_f \leftarrow (c_f * b) + t_i$
  $b_m \leftarrow b \times b_m$
Si $c_f = c_x$ alors *// ici $b_m = b^m$*
  afficher 1
Pour d allant de 0 à $n - m - 1$
  $c_f \leftarrow b(c_f - t_{d+1} b_m) + t_{d+m+1}$
  Si $c_x = c_f$ alors
    afficher $(d+2)$

• Si l'on compte les additions et les multiplications, la première boucle "Pour" est en $\Theta(m)$.
• De même, il y a de l'ordre de $\Theta(n-m)$ tours de la deuxième boucle "Pour", chacune faisant intervenir des opérations $=, +, -, \times$ qui sont en $\Theta(m)$.

D'où une complexité totale en $\Theta(nm)$: elle n'est donc pas pour l'instant meilleure que Motif-naïf.

Afin de l'améliorer, on modifie l'algorithme en décidant de réaliser cette fois les opérations modulo un entier $q > 0$, ce qui permet de rendre leurs complexités constantes (c'est-à-dire en $\Theta(1)$).

Algorithme – Rabin-Karp_bis

entrée : $q \in \mathbb{N}^*$
$t \in \Sigma^*$ avec $|\Sigma| = b$
$x \in \Sigma^*$ avec $|x| \leq |t|$

$n, m = |t|, |x|$
$c_x, c_f, b_m = 0, 0, 1$
Pour i allant de 1 à m
  $c_x \leftarrow (c_x * b) + x_i \pmod q$
  $c_f \leftarrow (c_f * b) + t_i \pmod q$
  $b_m \leftarrow b \times b_m$
Si $c_f = c_x$ alors
  Si $t_1 ... t_m = x$ alors
    afficher 1

Pour d allant de 0 à $n-m-1$
    $c_f \leftarrow b(c_f - t_{d+1}b_m) + t_{d+m+1} \pmod q$
    Si $c_f = c_x$ alors
        Si $t_{d+2}...t_{d+m+1} = x$ alors
            afficher $(d+2)$

Cette nouvelle version de l'algorithme présente toujours des tests d'égalité en $\Theta(m)$ dans les boucles "Pour", mais ils se produisent maintenant beaucoup plus rarement, ce qui affaiblit la complexité moyenne.

2.3 Algorithmes de Boyer-Moore

On donne enfin deux exemples d'algorithmes qui exploitent la forme même du motif afin d'effectuer une recherche plus intelligente de ses éventuelles occurences dans le texte considéré.

2.3.1 Algorithme de Boyer-Moore-Horspoole

Notation :

Pour $x \in \Sigma^m$, on pose : $f_x = \begin{pmatrix} \Sigma \to \mathbb{N} \\ a \mapsto \begin{cases} m \text{ si } a \notin \{x_i \,|\, i \in [1..m-1]\} \\ m - \max\{i \in [1..m-1] \,|\, x_i = a\} \text{ sinon} \end{cases} \end{pmatrix}$

Remarque : Si $a \in \Sigma$, on peut aussi écrire $f_x(a) = m - \max\Big(\{i \in [1..m-1] \,|\, x_i = a\} \cup \{0\}\Big)$.

Algorithme – Precalcul_BMH

entrée : $x \in \Sigma^*$

$f =$ tableau indexé par Σ, initialisé à m
Pour i allant de 1 à $m-1$ $\quad$ // $\forall\, \boldsymbol{a \in \Sigma}$, $\boldsymbol{f[a]} = \begin{cases} \boldsymbol{m \text{ si } a \notin \{x_j \,|\, j \in [1..i-1]\}} \\ \boldsymbol{m - \max\{j \in [1..i-1] \,|\, x_j = a\} \text{ sinon}} \end{cases}$
    $f[x_i] \leftarrow m - i$
Retourner f

Algorithme – Boyer-Moore-Horspoole

entrée : $x, t \in \Sigma^*$

$n, m = |t|, |x|$
$f =$ Precalcul_BMH(x)
$d = 0$
Tant que $d \leq n - m$
    Si $t_{d+1}...t_{d+m} = x_1...x_m$ alors
        afficher $d+1$
    $d \leftarrow d + f[t_{d+m}]$

Exemple : *cf.* annexe.

2.3.2 Algorithme simplifié de Boyer-Moore

Notation :

Pour $x \in \Sigma^m$, on note : $g_x = \begin{pmatrix} \Sigma \to \mathbb{N} \\ a \mapsto \begin{cases} m \text{ si } a \notin \{x_i \,|\, i \in [1..m]\} \\ m - \max\{i \in [1..m] \,|\, x_i = a\} \text{ sinon} \end{cases} \end{pmatrix}$

Algorithme – Precalcul_BMS

entrée : $x \in \Sigma^*$

$g =$ tableau indexé par Σ, initialisé à m
Pour i allant de 1 à m
    $f[x_i] \leftarrow m - i$
Retourner g

Algorithme – Boyer-Moore_simplifié

entrée : $x, t \in \Sigma^*$

$n, m = |t|, |x|$
$g =$ Precalcul_BMS(x)
$d = 0$
Tant que $d \leq n - m$
    $i = 0$
    Tant que ($t_{d+m-i} = x_{m-i}$ et $i < m$)
        $i \leftarrow i + 1$
    Si $i = m$ alors
        afficher $d + 1$
        $d \leftarrow d + 1$
    Sinon
        $d \leftarrow d + \max(1, g[t_{d+m-i}] - i)$

Exemple : *cf.* annexe.

3 Compression

Étant donné un texte $t = t_1 t_2 ... t_k ... t_n$ dont les caractères sont dans un alphabet Σ, on cherche à l'encoder par des caractères d'un "sur-alphabet" ou "super-alphabet" $\widehat{\Sigma}$ dont certains caractères encoderont des facteurs de t, ceci dans le but de réduire la taille de stockage du texte.

On aura donc un application $\varphi \in \mathcal{F}_p(\Sigma^+, \widehat{\Sigma})$ telle que $\Sigma \subseteq \mathcal{D}(\varphi)$, injective (voire même bijective, quitte à restreindre $\widehat{\Sigma}$).

Illustration : Concrètement, pour un texte $t = \overbrace{t_1...t_{r_1}|t_{r_1+1}...t_{r_2}|\dots|t_{r_{K-1}+1}...t_n}^{K}$, on aura un encodage de la forme $c = \varphi(t_1...t_{r_1})\,\varphi(t_{r_1+1}...t_{r_2})\dots\varphi(t_{r_{K-1}+1}...t_n) \in \widehat{\Sigma}^K$.

3.1 Algorithmes de Lempel-Ziv-Welch

Voyons dans un premier temps comment on peut intuitivement réaliser la compression et la décompression d'un texte, à partir d'un exemple bien choisi.

Exemple : Prenons $t =$ AUTOAUTOTAU sur Σ_ℓ, l'ensemble des lettres de l'alphabet latin majuscule.

On donne maintenant les pseudo-codes correspondant aux deux algorithmes que nous venons d'appliquer, appelés algorithmes de Lempel-Ziv-Welch.

Algorithme – Comp_LZW

entrée : $t = (t_i)_{i \in [0..n]}$ un texte
d un dictionnaire
hypothèses : $t \in \Sigma^+$, les clés de d sont exactement Σ et ses valeurs sont dans $[0..|\Sigma| - 1]$

$n, k = |t|, |\Sigma|$
res $= \varepsilon$
$m = t[\,0\,]$
Pour i allant de 1 à $n - 1$
  Si $m \cdot t[\,i\,] \in d.$clés alors
    $m \leftarrow m \cdot t[\,i\,]$
  Sinon
    res $\leftarrow$ res $\cdot\, d[m]$
    $d.$ajouter $(\overbrace{m \cdot t[\,i\,]}^{\text{clé}}, \overbrace{k}^{\text{valeur}})$
    $k \leftarrow k + 1$
    $m \leftarrow t[\,i\,]$
  Retourner res

Algorithme – Décomp_LZW

entrée : $c = (c_i)_{i \in [0..n-1]}$ un texte non vide
d un dictionnaire
hypothèses : les clés de d sont $[0..|d| - 1]$

$n, k = |c|, |d|$
res $= d[\,c[0]\,]$
$m = d[\,c[0]\,]$
Pour i allant de 1 à $n - 1$
  Si $c[\,i\,] \in d.$clés alors
    $r = d[\,c[i]\,]$
  Sinon
    $r = m \cdot m[\,0\,]$
  res $\leftarrow$ res $\cdot\, r$
  $d.$ajouter $(\overbrace{k}^{\text{clé}}, \overbrace{m \cdot r[\,0\,]}^{\text{val. associée}})$
  $k \leftarrow k + 1$
  $m \leftarrow r$
  Retourner res