907

Algorithmique du texte

Notation : On notera l'alphabet sur lequel on travaille Σ, supposé fini. Un texte sera noté $T = t_0 t_1 \dots t_{n-1}$; où $\forall i \in [\![0;n-1]\!]$, $t_i \in \Sigma$ et n est la longueur du texte (nombre de caractère). On note de même un motif $M = m_0 \cdots m_{k-1}$ de longueur k. k et n sont supposés toujours finis.

I Algorithmes de recherche de motif dans un texte

Problème : Soit un texte T et un motif M. On cherche le plus petit indice $j \in [\![0;n-1]\!]$ où n est la longueur du texte tel que $t_j t_{j+1} \cdots t_{j+k-1} = M$, k la longueur du motif.

Algorithme 1 : Algorithme naïf

Entrée : T texte, M motif

Sortie : indice j minimal tel que $t_j t_{j+1} \cdots t_{j+k-1} = M$

```
Recherche_naive (T; M)
| n, k = longueur(T); longueur(M);
| Pour j = 0 à n-k :
|   | Si M = t_j t_{j+1} ... t_{j+k-1} :
|   |   | renvoyer j
| Renvoyer "erreur: T ne contient pas M"
```

La complexité temporelle en pire cas est en $O(nk)$

→ Exemple 2 : Si on cherche aab dans $\underbrace{aaaaa}_{\text{grand nombre de a}}\dots aab = T$

On voit apparaître la complexité en pire cas : $O(nk)$.

Remarque 3 : Dans le cas de texte en français, on peut estimer que le texte ne contient pas beaucoup de préfixes du motif. Dans le cas de la recherche de séquence dans un génome, on peut atteindre la complexité en pire cas.

Algorithme 4 : Rabin-Karp.

On suppose $\Sigma = [\![0;n-1]\!] \subset \mathbb{N}$.

Principe : On va faire un pré-traitement sur T. Si on cherche un motif de taille k, on va créer un tableau T de taille $n-k$ où la case i contient $t_i + \dots + t_{i+k-1}$. Soit $m = \sum_{i=0}^{k-1} m_i$. Il suffira alors de regarder pour les i tel que $T.i = m$ si $M = t_i t_{i+1} \cdots t_{i+k-1}$.

Complexité : Ce paradigme est en pratique plus efficace que la recherche naïve mais, en pire cas, la complexité est en $O(nk)$. De plus, il faut un tableau de taille $n-k$ en mémoire.

Exemple 5 : Soit $\Sigma = [\![0,2]\!]$. On cherche $M = 011$.

Si $T = (002) \dots (002)\ 011$,

Définition 6 : Soit T un texte ; M un motif sur un alphabet Σ.

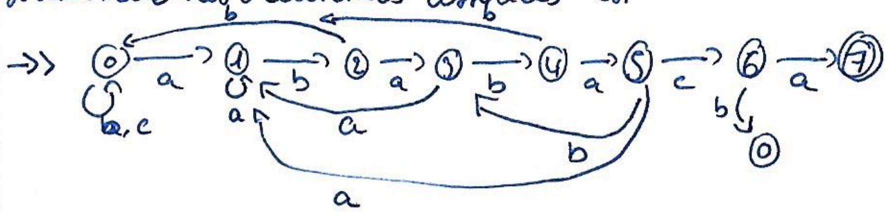
On définit l'automate des occurrences par :

- l'alphabet Σ
- $Q = [\![0, \dots, k]\!]$ les états, où k est la longueur du motif.
- $\{0\}$ l'état initial
- $\{k\}$ l'état final
- δ la fonction de transition, tel que $\forall q \in Q$, $\forall a \in \Sigma$, $\delta(q,a)$ renvoie l'indice j maximal tel que $m_0 \cdots m_{j-1}$ soit préfixe de M et suffixe de $m_0 m_1 \cdots m_{q-1}\, a$.

Remarque 7 : Un tel objet a besoin d'un espace mémoire en $O(|\Sigma| k)$

Exemple 8 : Soit $P = abababaca$; $\Sigma = \{a;b;c\}$.

L'automate des occurrences associées est



Si en 1, 2, 3, 4, 6 ; on lit un c, on est renvoyé à l'état 0.

Une fois en 7, l'automate s'arrête.

Algorithme 9 : Morris-Pratt

```
Morris-Pratt (T; M, Σ)
| δ = créer-delta(M, Σ)
| n = longueur(T)
| q = 0
| pour i = 0 à n-1 :
|   | q = δ(q, T.i)
|   | si q = m
|   |   | renvoyer i-k
| renvoyer "T ne contient pas M"
```

La complexité hors pré-traitement est linéaire en la taille du texte. Le prétraitement (contenu dans créer_delta, qui renvoie la fonction δ de transition de l'automate des occurrences associé à M) nécessite une place mémoire en $O(|\Sigma||M|)$ et s'effectue en $O(|\Sigma| k)$ opérations.

```
creer_delta(M; Σ):
  k = longueur(M)
  Pour q = 0 à m
      pour tout a ∈ Σ
        | k = min(m+1, q+2)
        | répeter
        |    | k = k-1
        | jusqu'à M_k ⊐ M_q a
        | δ(q, a) = k
  retourner δ
```

On note "$M_k \sqsupset M_q a$" le fait que $m_0 \cdots m_{k-1}$ soit suffixe de $m_0 \cdots m_{q-1} a$.
Les 2 premières boucles comptent pour $O(m|\Sigma|)$ opération. La boucle répeter peut se faire au plus $q \leq k$ fois, la vérification $M_k \sqsupset M_q a$ se fait en au plus k opérations
→ Complexité en $O(k^3|\Sigma|)$.

Remarque 10 (admis) : Il est possible de faire tomber la complexité de creer_delta à $O(k|\Sigma|)$ opérations.

Dev 1 : Algorithme de Knuth-Morris-Pratt, validité. L'algorithme a une complexité spatiale en $O(k)$, temporelle en $O(n)$.

Remarque 11 : Les algorithmes Morris-Pratt et Knuth-Morris-Pratt peuvent se généraliser au cas d'ensemble de mots fini.

II Distance d'édition et recherche approché de motif

Exemple 12 : Si la recherche n'est pas sensible à la casse, on peut retrouver "Abc" au lieu de "abc". On va généraliser la recherche approchée de motif.

Definition 13 : On pose les fonctions suivantes :
- Sub : $\Sigma \times \Sigma \to \mathbb{R}^+$; où $sub(a;b)$ est le cout de substitution de la lettre a par b. $su(a;a) = 0$, $(a;b) \in \Sigma^2$
- Del : $\Sigma \to \mathbb{R}^+$; où $del(a)$ est le cout de la suppresion de a, $a \in \Sigma$
- Ins : $\Sigma \to \mathbb{R}^+$, où $Ins(a)$ est le cout d'insertion de la lettre a.

Definition 14 : On note $\forall x; y \in \Sigma^*$, mots sur Σ, Σ_{xy} l'ensemble des suites d'opérations menant du mot x au mot y. C'est non vide, car on peut supprimer toutes les lettres de x puis ajouter toutes celles de y.

Définition 15 : $Lev : \Sigma^* \times \Sigma^* \to \mathbb{R}^+$

$$(x; y) \mapsto \min(\text{cout de } \sigma, \sigma \in \Sigma_{xy})$$

est appelé distance d'édition.

Théorème 16 : Lev est une distance sur Σ^* si et seulement si
- sub est une distance sur Σ
et
- $\forall a \in \Sigma$, $del(a) = Ins(a) > 0$

Definition 17 : Un alignement entre x et $y \in \Sigma^*$ est un mot z sur l'alphabet $(\Sigma \cup \{\varepsilon\})^2 / \{(\varepsilon;\varepsilon)\}$ dont la projection sur la première composante est x ; y sur la dernière composante. On note $z = \begin{pmatrix} \overline{x_0} & \cdots & \overline{x_n} \\ \overline{y_0} & \cdots & \overline{y_n} \end{pmatrix}$

Exemple 18 : Si $x = ACGA$; $y = ATGCTA$, un alignement est $z = \begin{pmatrix} A & C & G & \varepsilon & \varepsilon & A \\ A & T & G & C & T & A \end{pmatrix}$. $\binom{A}{\varepsilon}$ représente une suppression.
représente la substitution. représente l'ajout

Définition 19 : Le cout d'un alignement est la somme des couts de chaque opération représentée.

Définition 20 : Un alignement z entre $x; y \in \Sigma^*$ est dit optimal si son cout est $Lev(x,y)$.

Dev 2 (Bastien) Calcul de l'alignement optimal entre 2 mots sur Σ^*

Problème : Soit un texte T, un motif M, de longueur respective n et k. On souhaite trouver les séquences M', de longueur $k' \geq k$, avec un nombre maximal de différences, ie où $Lev(M, M') \leq d$, d est fixé, $d \leq k$. On suppose que toute substitution, suppression et ajout coute 1

Dev 3 (Rémi) Recherche des facteurs de T à une distance au plus k de M par la programmation dynamique. Amélioration avec les cases l-spéciales. Complexité.

Exemple 21 : Ce problème convient dans le cas de recherche de séquences ADN dans un génome en tenant compte d'éventuelles mutations.

III Compression

Remarque 22 : On ne peut pas compresser indéfiniment sans perte d'information. On ne peut pas forcément stocker l'information de n bits en n-1 bits.

Définition 23 : L'algorithme de Lempel Ziv (LZ) est un algorithme de compression par substitution de facteurs. Il remplace des séquences par l'indice des séquences dans un dictionnaire. Il fonctionne en une seule lecture du texte.

Exemple 24 : "Un bon gars est un gars bon." donnera alors "Un bon gars est [1] [3] [2]".

Remarque 25 : Cette base d'algorithme est aujourd'hui majoritairement utilisée.

Algorithme 26 : LZ 77

Basée sur une fenêtre qui glisse de gauche à droite sur le texte, divisée en 2 parties : une qui contient le dictionnaire, l'autre qui rencontre le texte en premier (tampon de lecture).
Initialement, la première partie est vide, la dernière contient T.
A chaque itération, on cherche la plus longue séquence préfixe de T contenue dans le dictionnaire, codé par (i, j, c) où
i est la distance entre le début du tampon et la position de la répétition dans le dictionnaire
-> j est la longueur de la répétition
-> c est le premier caractère du tampon différent du caractère correspondant dans le dictionnaire.

La fenêtre coulisse de j+1 caractères.

Exemple 27 : On applique LZ 77 à abcdefabcdef abcdefabcdef
On obtient :

$$\underbrace{(0,0,a)\,(0,0,b)\,(0,0,c)\,(0,0,d)\,(0,0,e)\,(0,0,f)}_{abcdef}\ (6,6,a)\ (12,10,f)$$

abcdef a | (abcdef)x (abcde) f

6

(bcdefabcde)

10

Algorithme 28 Lempel-Ziv-Welch (LZW)

Il consiste à ne coder que l'indice n dans le dictionnaire.
Il est nécessaire d'avoir un dictionnaire initial (par exemple la table ASCII). On remplace chaque groupe de caractères déjà connu par son code et on ajoute au dictionnaire un nouveau groupe formé par ce groupe suivi du prochain caractère.

```
LZW_Comp (T, dictionnaire)
  chaine ← Ø
  Tant qu'on n'est pas à la fin de T
    lire c caractère de T
    Si chaine|c ∈ dictionnaire
      | chaine ← chaine|c
    Sinon
      | afficher code(chaine)
      | ajout chaine|c au dictionnaire
      | chaine ← c
```

```
LZW_decompression (code, dichi.)
prec ← lire_octet(code)
Tant qu'on n'est pas à la fin de code
  cour ← lire_octet(code)
  Si cour ∈ dictionnaire
    | chaine ← trad(cour)
  Sinon
    | chaine ← trad(prec)
    | c ← premier_caractère(chaine)
    | chaine ← chaine|c
  afficher chaine
  ajout chaine au dictionnaire
  prec ← cour
```

Pour la compression, le si_sinon gère le seul cas problématique : la répétition de chaine (cas où cour ∉ dictionnaire). On part avec les mêmes dictionnaires initiaux.

IV Codes correcteurs

Problème : On veut vérifier algorithmiquement si un message reçu par un ordinateur est correct.

Exemple 29 : Code de parité

On envoie un message de n bits. On va envoyer un n+1 ième bit de sorte à ce que la somme des éléments soient pair.

0 1 0 -> 0 1 0 1
0 1 1 -> 0 1 1 0 || Permet de détecter une erreur mais ne permet pas de corriger.

Exemple 30 : Code de répétition.

Soit m le message envoyé. On va répéter 3 fois m.

1001 -> 1001 1001 1001

Si on envoie 1000 1001 1001, on peut détecter et corriger l'erreur.