907 — Algorithmique du texte

Le texte est à la base de toutes les données, et c'est pourquoi on s'intéresse à le manipuler et l'étudier (compression, recherche, ...)

I] Recherche de motifs

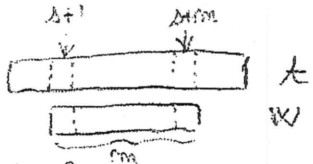
La recherche de motifs dans un texte est utilisée dans de nombreux domaines (séquences d'ADN, analyse lexicale, éditeurs de texte, langages de programmation...)

1) Algorithme naïf de recherche d'un mot [COR]

Ici le motif est un mot w, dont on cherche une occurrence dans un texte t, sur un alphabet Σ.

ENTRÉE : mot w, texte t, (tableaux)
SORTIE : oui si w apparaît dans t, non sinon

```
RECHERCHE-NAÏVE(w,t):
  n = |t|   // longueur du texte t
  m = |w|
  Pour s = 0 à n - m
  |  Si t[s+1...s+m] == w[1...m]
  |  | retourner oui
  retourner non
```



correction : w est comparé à tous les facteurs de t de longueur m.

complexité : $O((n-m+1)m)$ dans le pire des cas en nombre de comparaisons de caractères. En "moyenne" on a cependant le résultat :

PROP : Si l'alphabet Σ a au moins deux lettres, et si la distribution constituit une loi uniforme, le nombre moyen de comparaisons pour rechercher w dans t est au plus $2|t|$.

2) Algorithme de Rabin-Karp [COR]

Cet algorithme calcule une valeur W_b dans une base b du mot w de longueur m, et t_s de $t[s+1,\dots,s+m]$, le tout modulo un entier q. Hörner est utilisé pour calculer la valeur W_b à partir de w, et la relation $t_{s+1} = b(t_s - b^{m-1}t[s+1]) + t[s+m+1]$ pour $t[s+1,\dots,s+m]$.

Ensuite on utilise :

PROP : Si $W_b \neq t_s$ alors $t[s+1\dots s+m] \neq w$.

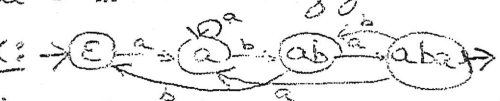
Ceci permet de rejeter des cas, et ne faire la vérification $w == t[s+1\dots s+m]$ que si le test est positif.

Complexité : dans le pire des cas c'est la même que pour l'algorithme naïf. Cependant, sous certaines hypothèses comme le fait que la réduction modulo q suive une loi uniforme et que $q \geq m$ et que le nombre de décalages valides soit borné on peut aboutir à une complexité moyenne en $O(n)$.

3) Knuth-Morris [COR + Beauquier]

a) Automate des occurrences (DEV)

Il s'agit de construire un automate minimal qui reconnaît le langage $\Sigma^* w$.

EX :


pour $w = aba$

b) Knuth-Morris-Pratt

Cet algorithme est inspiré de l'algorithme induit par le calcul de l'automate des occurrences, mais effectue un prétraitement sur le mot w qui permet de calculer la fonction de transition plus efficacement.

Il utilise la formule $\delta(p,a) = \begin{cases} pa & \text{si } pa \text{ préfixe de } w \\ Bord(pa) & \text{sinon} \end{cases}$

où $Bord(x)$ est le plus grand suffixe de x qui est aussi suffixe de x.

4) Algorithme de Boyer-Moore [Beauquier]

La particularité de cet algorithme est de comparer w aux facteurs de t de la droite vers la gauche au lieu de la gauche vers la droite.
Dans la version la plus simple (appelée algorithme de Boyer-Moor-Horspool), on cherche la première différence entre w et un facteur de t de longueur m, puis on décale w de manière à faire coïncider la lettre de w à droite de la différence avec la dernière du facteur de t.

a	a	b	b	b	a	b	a	b			t
a	a	b	a	b	a	b					w (étape 1)
		a	a	b	a	b	a	b			w (étape 2)

complexité : dans le pire des cas $O(|w||t|)$.
Cependant en pratique la complexité moyenne est excellente et c'est pourquoi cet algorithme est utilisé dans de nombreux logiciels comme GNU grep ou perl pour les cas de motifs qui se réduisent à trouver les occurrences d'un mot dans un texte.

5) Recherche d'expressions [Beauquier] 10,4 p 369

On se donne une expression rationnelle e et un texte t. On veut déterminer s'il existe un mot dans le langage $X = L(e)$ qui figure dans t, et si oui, rapporter le mot et son occurrence.

Solution naïve : construire un automate déterministe qui reconnaît le langage $\Sigma^* X$; le problème est qu'il peut vite devenir trop grand.

PROP : Pour toute expression rationnelle de taille m, il existe un automate normalisé reconnaissant $L(e)$, et dont le nombre d'états est au plus $2m$.

REM : On impose dans "normalisé" que tout état soit à l'origine d'une transition étiquetée par $a \in \Sigma$ ou d'au plus deux transitions.

normalisé asynchrone

PROP : un tel automate permet une recherche en $O(|t||e|)$ avec un algorithme adapté.

Applications : éditeurs de texte, extractions de données, grep, langages de programmation, ...

II Distances entre mots [COR]

La question de déterminer si deux mots se ressemblent : par exemple en biologie on peut vouloir comparer deux fragments d'ADN, vus comme des mots sur l'alphabet $\{A, C, G, T\}$, afin de déterminer le degré de parenté de deux organismes.
On peut définir plusieurs types de distances, les plus simples s'intéressant principalement :
- aux sous-mots
- au nombre de modifications nécessaires pour obtenir un mot à partir d'un autre.

1) Distance d'édition

On se donne un mot $x[1..m]$ et un mot $y[1..n]$ et on veut construire $z = y$ à l'aide des opérations suivantes, où i et j sont deux variables d'indice :
- copier : $\{z[j] = x[i]; i{+}{+}; j{+}{+};\}$

- remplacer : $\{z[j] = c\ ;\ i{+}{+}\ ;\ j{+}{+};\}$
- supprimer : $\{i{+}{+}\}$
- insérer : $\{z[j] = c\ ;\ j{+}{+};\}$
- permuter : $\{z[j] = x[i+1];\ z[j+1] = x[i];\ i += 2;\ j += 2\}$
- équeter : $\{i = m+1\}$. (équeuter)

Coût : la distance est le nombre d'opérations nécessaires (avec des poids éventuellement).

2) Plus longue sous-séquence commune (DEV)

Étant donnés deux mots x et y on cherche le plus long mot z qui soit un sous-mot de x et y.
On parle de distance sans substitution.

III Compression de données avec le codage de Huffman. [COR]

L'idée du codage est que ce qui est fréquent doit être plus court que ce qui est rare.
On dispose d'un alphabet Σ fini, et d'un texte t sur Σ. Le codage de Huffmann utilise la fréquence d'apparition des caractères pour construire un encodage de t dans le but de le compresser.

EX : Avec $\Sigma = \{a, b, c, d, e, f\}$ et le tableau de fréquences

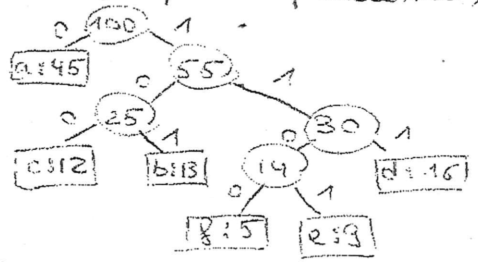
a	b	c	d	e	f
45	13	12	16	9	5

DEF : On dira qu'un arbre binaire est complet si tout noeud est soit une feuille soit le père d'exactement deux noeuds.

DEF : un codage préfixe est un codage où aucun mot de code n'est préfixe d'un autre mot de code.

DEF : un codage préfixe optimal est un codage préfixe qui minimise $C(T) = \sum_{c \in \Sigma} freq(c)\, profondeur(c)$ où T est l'arbre correspondant au codage (vu comme application de $\Sigma \to \{0,1\}^*$) dont les feuilles sont les lettres de Σ dont l'adresse est donnée par le codage (0 pour gauche, 1 pour droite).

EX : (pour les fréquences précédentes)




```
ENTRÉE : alphabet Σ
SORTIE : arbre T du codage de Huffman
HUFFMAN(Σ)
n = |Σ|
Q = Σ     // file de priorité
Pour i = 1 à n-1
    allouer un nouveau noeud z
    z.gauche = EXTRAIRE-MIN(Q)
    z.droit = EXTRAIRE-MIN(Q)
    z.freq = z.gauche.freq + z.droit.freq
    INSÉRER(Q, z)
retourner EXTRAIRE-MIN(Q)
```

où INSÉRER et EXTRAIRE-MIN sont des fonctions insérant un élément ou retirant le plus petit élément d'une file de priorité.

TH. La procédure HUFFMAN produit un codage préfixe optimal.