

Algorithmique du texte. Exemples et applications

Intérêts:

Traitement de texte, correcteur orthographique, analyse de séquences musicales, bio-informatique, détection de plagiat

I. Notions importantes

Alphabet: Ensemble non vide de symboles, fini ou infini, noté Σ

Mot: Liste de symboles issus de Σ . Σ^* : ensemble des mots

Préfixes d'un mot w : $\text{pref}(w) = \{u \in \Sigma^* / \exists v \in \Sigma^* w = uv\}$

Suffixes d'un mot w : $\text{suff}(w) = \{u \in \Sigma^* / \exists v \in \Sigma^* w = vu\}$

Bords: $\text{bord}(w) = \text{pref}(w) \cap \text{suff}(w)$

Facteurs d'un mot w : $\text{fact}(w) = \{u \in \Sigma^* / \exists v, v' \in \Sigma^* w = uvv'\}$

II Recherche de motifs dans un texte

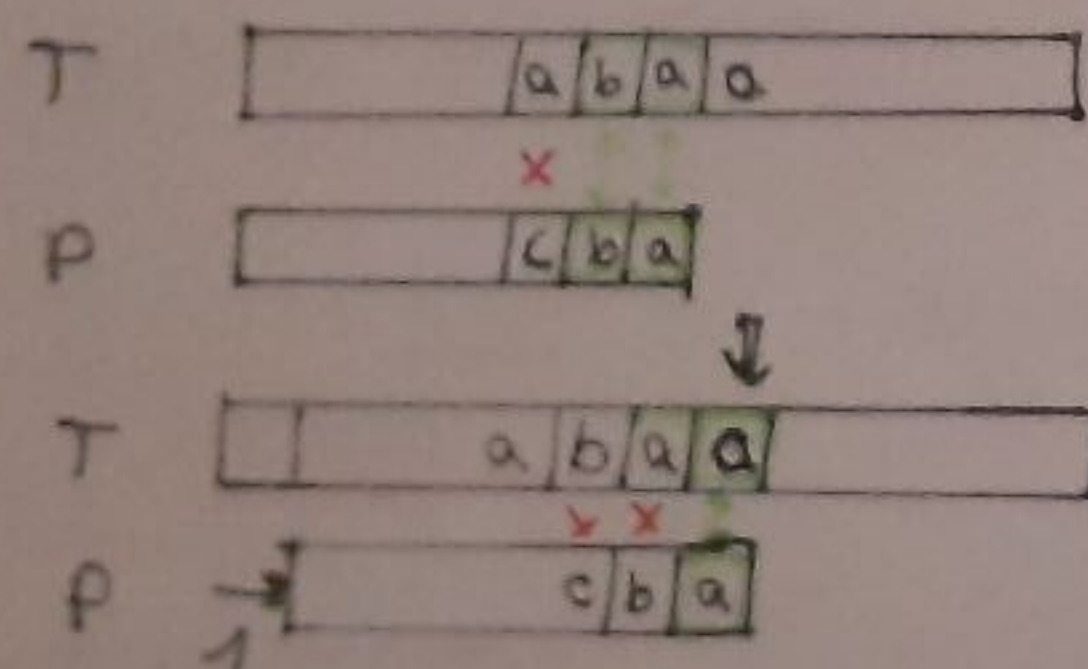
Applications: chr+F, recherche google, détection de plagiat

Motif: représentation d'un ensemble fini de mots (ex: regex)

Principe: trouver les occurrences d'un motif P de longueur m dans un texte T de longueur n

→ Résultat: $\{i / 1 \leq i \leq n-m+1 \wedge T[i, i+m-1] = P\}$

1) Algorithme naïf



Complexité:

pire-cas: $O(mn)$

meilleur-cas: $O(mn)$

Fonction Recherche(T, P):

```

L ← ∅
for i = 0 to n-m do:
    j ← m-1
    while j ≥ 0 and j < m:
        if T[i+j] = P[j]:
            j ← j-1
        else:
            j ← 0
    if j = m-1:
        L ← L ∪ {i+1}
return L
    
```

2) Optimiser le déplacement

a. Sans automate: algorithme de Boyer-Moore

Calcul de deux décalages optimaux

mauvais caractère

$$d_1(a) = m - \max\{i / i < m \wedge P[i] = a\}$$

T: a b a a

P: a c b a

$d_1(a)$

T: a

P: a c b a

$d_1(a)$

bon suffixe

T: a

P: a c b a

$d_2(P)$

$d_1(P)$

$d_2(P)$

$d_2(P) = \min(d_1(P), d_2(P))$

→ On décale du maximum de $d_1(a), d_2(P)$

Complexité

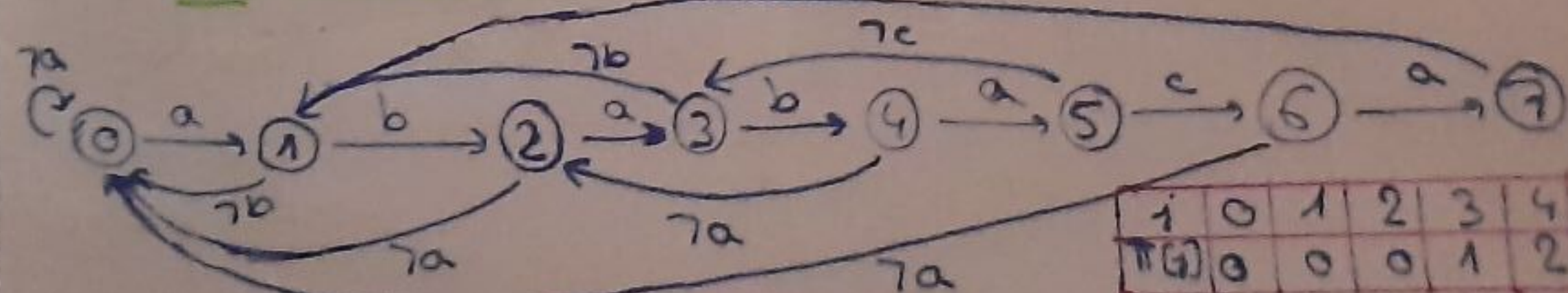
pire-cas: $O(m \cdot m)$

meilleur-cas: $O(\frac{m}{m})$

b. Avec automate

Automate de Knuth-Morris-Pratt:

ex: ababaca



Etude de l'algorithme: DEV 1

	1	0	1	2	3	4	5	6	7
$\pi(i)$	0	0	0	1	2	0	0	1	
	a	b	a	b	a	c	a		

3) Utilisation de la table de hachage

Algorithme de Rabin-Karp

- Fonction de hachage avec paramètres d (base) et q (taille)
- Chaque lettre de l'alphabet est associée à un nombre (ex: ASCII)
- $R_p = (\sum_{i=0}^{m-1} d^{m-i} P[i]) [q]$ - $R_t(s) = (\sum_{i=0}^{m-1} d^{m-i} T[i]) [q]$
- On cherche $E = \{s \mid s \text{ de } n-m+1 \mid R_t(s) = R_p\}$
- On compare pour tout s de E $T[s..s+m-1]$ à P

Complexité moyenne: $O(m+n)$

Complexité pire cas: $O(mn)$

III - Mesures de similarité

Applications: alignement de génomes, correction d'erreur


1) Distances

distance préfixe: distance du plus long préfixe commun

distance suffixe: longueur du plus long suffixe commun

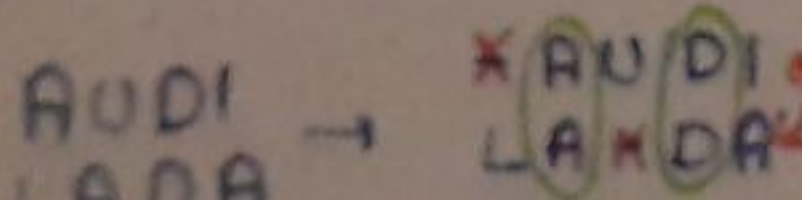
distance facteur: longueur du plus long facteur commun

distance de Hamming: nombre de positions où les caractères diffèrent

ex:  → distance Hamming (u, v) = 3

distance d'édition: nombre d'opérations à effectuer pour obtenir le même mot

→ opération: insertion, suppression, modification

ex:  → $\text{dist}(\text{AUDI}, \text{LADA}) = 3$

Calculer la distance d'édition: algorithme de Levenshtein

→ programmation dynamique

$D[i, j]$: distance entre $u[1..i]$ et $v[1..j]$

$\forall i, D[i, 0] = i, \forall j, D[0, j] = j$

$\forall i, j > 0, D[i, j] = \min \begin{pmatrix} D[i-1, j-1] + \mathbb{1}_{u[i] \neq v[j]} \\ D[i-1, j] + 1 \\ D[i, j-1] + 1 \end{pmatrix}$

ex:

	A	U	D	I
L	0	1	2	3
A	1	1	2	3
D	2	2	2	3
A	3	3	3	3

$\text{dist}(\text{AU}, \text{LAD}) = 2$

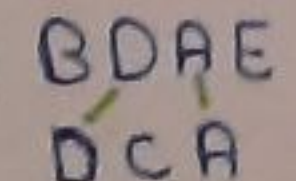
→ On peut utiliser cet algorithme pour une recherche de motif approchée (trouver l'occurrence la plus proche du motif dans un texte)

→ on change l'initialisation:

$D[i, j]$: distance de la plus proche occurrence de $p[1..i]$ dans $T[1..j]$

→ changer $\forall i, D[i, 0] = 0, \forall j, D[0, j] = j$

2) Plus longue sous-séquence commune (PLSC)

ex:  → $\text{PLSC}(\text{BDABE}, \text{DCA}) = \text{DA}$

Algorithme: programmation dynamique

$D[i, j]$: longueur de la PLSC entre $u[1..i]$ et $v[1..j]$

$\forall i, D[i, 0] = 0$ et $\forall j, D[0, j] = 0$

$\forall i, j > 0, D[i, j] = \begin{cases} D[i-1, j-1] + 1 & \text{si } u[i] = v[j] \\ \max(D[i-1, j], D[i, j-1]) & \text{sinon} \end{cases}$

ex:

	B	D	A	E
	0	0	0	0
D	0	0	1	1
C	0	0	1	1
A	0	0	1	2

DEV2 : Algorithme de la plus longue sous-séquence commune + optimisation mémoire

V. Compression de texte

Applications: zip, reformattage de fichiers

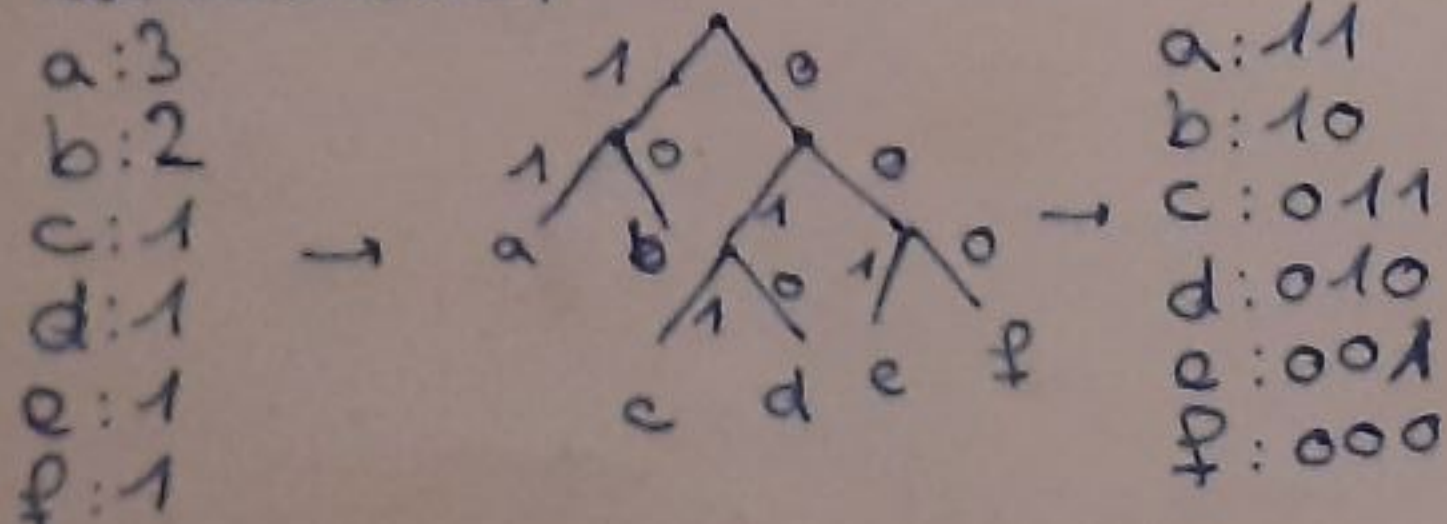
Principe: transformer un texte pour réduire sa taille

1) Algorithme de Huffman

Idee: Les symboles les plus fréquents utiliseront moins de places que les symboles plus rares.

- on compte le nombre d'occurrences de chaque lettre
- on construit un arbre binaire tel que les éléments les plus fréquents soient les moins profonds, et chaque branche correspond à un bit 0 ou 1

ex: abaabcdef



2) Algorithme de Lempel-Ziv-Welch

Idee: On ne code pas seulement des lettres mais des chaînes de caractères

ex: babaaba
D = {a:1, b:2}

n°	T	D	résultat
1	<u>b</u> aaba	ba:3	2
2	b <u>a</u> aaba	ab:4	1
3	ba <u>b</u> aaba	baa:5	3
4	ba ba aba	aba:6	4

→ résultat: 2-134

Fonction LZW(T, D):

3/3

```

résultat ← []
w ← ""
while T ≠ ∅:
    c ← Line(T)
    p ← w + c
    if p in D:
        w ← p
    else:
        D[p] = length(D) + 1
        résultat.ajouter(D[w])
        w ← c
return résultat

```

Conclusion

Domaine vaste et nombreuses applications