Algorithme de recherche de (K)MP

Dans ce développement, on donne les motivations de l'algorithme de motif de Morris-Pratt, le pseudo code de l'algorithme, la preuve de sa terminaison, sa correction, sa complexité, et les idées d'optimisation pour obtenir KMP.

Motivations

L'idée de base de cet algorithme de recherche est qu'on a envie d'améliorer l'algorithme naïf en utilisant la remarque suivante : lorsqu'une lettre ne matche pas, on peut (peut-être) décaller la fenêtre de test de plus qu'un caractère.

Exemple. *Dans cet exemple, la lettre a et b ne matche pas, et on peut décaler la fenêtre de deux crans et on a pas besoin de retester abab*

$t =$	c	d	a	b	a	b	a	b	...
$u =$			a	b	a	b	a		
$u =$					a	b	a	b	b

Si on note $v = abab$ le motif déjà matché, et $w = ab$ le nouveau motif matché après le décallage, on se rend compte qu'on veut prendre pour w le plus grand mot qui est préfixe de v et suffixe de va. Ainsi, $w = \mathrm{Bord}(va)$.

L'automate des occurences

Cette idée pousse naturellement à définir un automate $A = (Q, \varepsilon, u, \delta)$ avec :

- L'ensemble des états $Q = \mathrm{Pref}(u)$
- ε l'état initial (préfixe vide de u), u l'état final (préfixe plein de u)
- La fonction de transition δ définie par : $\delta(v, a) = \begin{cases} va & \text{si } va \in \mathrm{Pref}(u) \\ \mathrm{Bord}(va) & \text{sinon} \end{cases}$

Cet automate reconnait bien $\Sigma^* u$ mais il n'est pas évident qu'il peut être construit de manière efficace (ie en temps linéaire).

De la bordure

Le problème principal est de trouver comment calculer la bordure de manière efficace. Pour cela, on va montrer le lemme central suivant :

Lemme 1. $\mathrm{Bord}(va) = \begin{cases} \mathrm{Bord}(v)a & \textit{si } \mathrm{Bord}(v)a \in \mathrm{Pref}(u) \\ \mathrm{Bord}(\mathrm{Bord}(v)a) & \textit{sinon} \end{cases}$

Proof. Soit $w = \mathrm{Bord}(va)$. Puisque w est un suffixe de va, w s'écrit forcement $w = w'a$.

- w est un préfixe de $va \Rightarrow w'$ est un préfixe de v.
- w est un suffixe de $va \Rightarrow w'$ est un suffixe de v.
- Par maximalité de w, w' est le plus bord de v tel que $w'a$ est un prefixe de u.

Du coup, on a deux cas à distinguer assez naturellement :

- Si $\text{Bord}(v)a$ est un préfixe de u, alors par maximalité $w' = \text{Bord}(v)$ et donc $\text{Bord}(va) = \text{Bord}(v)a$
- Sinon, w' est un préfixe strict de $\text{Bord}(v)$, et donc un préfixe de $\text{Bord}(\text{Bord}(v))$. On peut donc se ramener au calcul de $\text{Bord}(\text{Bord}(v)a) = \text{Bord}(va)$.

□

Si on déroule ce lemme, on obtient le résultat suivant, qui se traduit en algo immédiatement :

$$\text{Bord}(va) = \begin{cases} \text{Bord}(v)a & \text{si } \text{Bord}(v)a \in \text{Pref}(u) \\ \text{Bord}(\text{Bord}(v))a & \text{sinon si } \text{Bord}(\text{Bord}(v))a \in \text{Pref}(u) \\ \text{Bord}(\text{Bord}(\text{Bord}(v)))a & \text{sinon si } \text{Bord}(\text{Bord}(\text{Bord}(v)))a \in \text{Pref}(u) \\ \dots & \\ a & \text{sinon si } a \in \text{Pref}(u) \\ \varepsilon & \text{sinon} \end{cases}$$

L'algorithme de calcul des bords

En réalité, l'algorithme a besoin seulement de la taille des bords des préfixes de u pour fonctionner, on définit donc $\pi(k) = |\text{Bord}(u[1, \dots, k])|$, qu'on peut calculer avec l'algorithme suivant :

COMPUTE_PI(u)
  $\pi(1) \leftarrow 0$
  $k \leftarrow 0$
  Pour $i = 2 \dots |u|$ **Faire**
    Tant que $k > 0$ ET $u[k+1] \neq u[i]$ **Faire**
      $k \leftarrow \pi(k)$
    Fin Tant que
    Si $u[k+1] = u[i]$ **Alors**
      $k \leftarrow k + 1$
    Fin Si
    $\pi(i) \leftarrow k$
  Fin Pour

Lemme 2. *Cet algorithme calcule la fonction π en temps linéaire.*

Proof.
- Terminaison : on a $\pi(k) < k$, ce qui donne la terminaison de la boucle while, et donc la terminaison du programme.

- Correction : découle directement du lemme 1 sur la définition récursive du bord.
- Complexité : comme k est toujours positif, à l'étape i de la boucle for, la boucle while peut désincrémenter k de au plus sa valeur. Or on incrémente k de 1 au plus une fois par passage dans la boucle for. On désincrémente donc k au plus n fois, donc on exécute la boucle while au plus n fois, et comme la boucle for est exécuté n fois, on obtient une complexité en $O(n)$.
  On peut aussi montrer cette borne avec une autre technique d'analyse amortie : on paye 2 pour chaque incrémentation de k, ce qui nous permet de désincrémenter k gratuitement, et donc la complexité est en $O(2n) = O(n)$.

□

Remarque. *On peut adapter l'algorithme précédent pour qu'il calcule l'automate des occurences en calculant la fonction* $\pi_a(k) = |\operatorname{Bord}(u[1..k]a)|$ *en temps et espace* $O(|u|\cdot|\Sigma|)$.

L'algorithme de recherche MP

L'algorithme de calcul des bords est en fait déjà l'algorithme de recherche. En effet, si on veut chercher u dans t, il suffit de calculer les bords de $v = u\sharp t$, pour $\sharp \notin \Sigma$, car on a le résultat suivant :

Lemme 3. $\forall k, \pi(k) \leq |u|$ *et* $\pi(k) = |u|$ ***ssi*** u ***apparait à la position*** $k - 2|u| - 1$ ***dans*** t.

Proof. Le premier point du lemme est assuré par le symbole $\sharp$ qui n'appartient pas à l'alphabet. Si on a $\pi(k) = |u|$, on écrit $v[1..k] = u\sharp t[1..k - |u| - 1]$ et donc $\pi(k) = |u|$ implique que u est le bord de $v[1..k]$, donc que u est suffixe de $t[1..k - |u| - 1]$ et donc que u apparait dans t à la position $k - 2|u| - 1$. □

D'après ce qu'on a vu précedemment, un tel algorithme est linéaire en $|v| = |u|+|t|+1$. On peut donc chercher u dans t en temps $O(|u| + |t|)$.

L'amélioration de KMP

L'idée assez simple de KMP est de raffiner la fonction π en prenant plutôt $\pi'(k) =$ plus long bord v de $u[1..k]$ tel que $u[|v| + 1] \neq u[k + 1]$, et -1 si un tel bord n'existe pas. En effet, la condition dans la boucle while est qu'on cherche le bord le plus long qui matche sur la lettre suivante. Mais si $u[k + 1] \neq u[|v| + 1]$ et que $u[|v| + 1] = u[\pi(|v|) + 1]$, alors $u[\pi(|v|) + 1] \neq u[k + 1]$ et ça ne sert à rien de tester ce bord.