La lettre de Caml numéro 4

Laurent Chéno
54, rue Saint-Maur
75011 Paris
Tél. 01 48 05 16 04
Fax 01 48 07 80 18
email : `cheno@micronet.fr`

juin 1996

Édito

Il faut que je m'y résolve : sans une contribution de votre part, je ne peux assurer en plus de mes cours (et la réforme en cours complique encore le travail, et surtout l'alourdit...) une publication de La lettre de Caml *aussi régulière que ce que je souhaiterais.*
Pour être raisonnable, je pense qu'il faut envisager un numéro à chacune des vacances scolaires, ce qui fait 5 numéros par an. Bien entendu, la longue période de l'été peut être l'occasion de la rédaction d'une brochure particulière et plus dense sur un sujet précis, de la même façon que l'été dernier m'a permis de rédiger le poly de géométrie que Pierre Weis a bien voulu installer sur le serveur de l'INRIA.
Ce numéro essaie de faire le tour des algorithmes classiques de recherche d'un motif dans une chaîne, en tentant d'insister sur la rigueur de la présentation. Je n'ai pas été exhaustif, et n'ai par exemple pas cité l'algorithme de Aho et Corasick qui aurait peut-être eu la préférence de certains. J'espère toutefois que cette compilation de certains chapitres d'ouvrages que je cite en référence pourra être utile à tous.
Ensuite, je propose une description détaillée de la bibliothèque `format` *de Caml, que des numéros précédents de* La lettre de Caml *ont utilisée, et qui me paraît d'usage plus fréquent que ce que je pouvais encore penser il y a quelques mois. J'ai ainsi fourni aux élèves des* pretty-printer *d'expressions logiques qui auront pu rendre plus lisibles leurs essais tout en restant absolument transparents à l'utilisateur. Que Pierre Weis soit ici encore remercié pour son aide, sa précision et sa supervision dans la rédaction de ce chapitre.*

Table des matières

Liste des programmes

Recherche d'un motif dans une chaîne

Position du problème

Nous nous intéressons ici à un problème très classique en algorithmique, pour lequel nous ne prétendons en aucun cas à l'originalité. Parmi les références habituelles sur ce thème, nous citerons en particulier [5, pages 213–223] (que nous aimons vraiment), [4, pages 52–62] (pour les *afficionados* de Pascal), [2, pages 213–222] qui est écrit en canadien, et enfin le très sérieux [1, chapitre 10]. Il faut bien sûr aussi citer [3] pour une analyse plus fine que celle que nous présentons ici des différents algorithmes utilisés.
De quoi parlons nous donc ?
On considère une chaîne s de longueur n, et un motif t de longueur m. On souhaite trouver, quand elle existe, la première occurrence de t dans la chaîne s, c'est-à-dire le plus petit entier $i \geq 0$ tel que

$$\forall k \in \{0, 1, \ldots, m-1\}, s_{i+k} = t_i.$$

En général on supposera que n est très grand devant m.
Une variante du même problème consiste à chercher les occurrences d'un jeu de plusieurs motifs.
La plupart des algorithmes (sauf l'algorithme naïf, bien entendu) procèdent par un pré-traitement sur le motif, qui est censé accélérer l'algorithme de recherche proprement dit. C'est dire en particulier qu'ils sont adaptés à une dernière variante de notre problème : rechercher les occurrences d'un même motif dans un jeu de plusieurs chaînes.

L'algorithme naïf

Description de l'algorithme

L'algorithme naïf observe simplement qu'avec les notations précédentes (recherche d'un motif t de taille m dans une chaîne s de taille n) l'occurrence éventuelle i de t dans s vérifie bien entendu $i + m < n$.
On va donc successivement, pour i variant de 0 à $n - m$, comparer le motif t avec la sous-chaîne $s[i..i + (m - 1)]$.

Implémentation en Caml

On obtient facilement le programme 1 page suivante.
On a choisi d'écrire un programme itératif, mais il ne serait pas bien difficile d'écrire le même algorithme dans un style parfaitement fonctionnel, par exemple comme dans le programme 2 page suivante.

Évaluation

L'analyse rigoureuse de l'algorithme naïf n'est pas si évidente qu'il y paraît.
En revanche il est assez clair qu'au pire son coût est en $O(nm)$. C'est ce qui arrive dans le cas où le motif est de la forme `aaa...aaab` et où la chaîne ne contient que la lettre `a`.

Programme 1 L'algorithme naïf de recherche d'un motif dans une chaîne

```
exception Échec and Réussite ;;

let position motif chaîne =
    let m = string_length motif
    and n = string_length chaîne
    and i = ref 0
    and k = ref 0
    in
    try while ! i <= n - m do
            k := 0 ;
            while ! k < m && motif.[! k] = chaîne.[! i + ! k] do incr k done ;
            if ! k = m then raise Réussite
            else incr i
        done ;
        raise Échec
    with Réussite -> ! i ;;
```

Programme 2 Une version fonctionnelle de l'algorithme naïf

```
exception Échec ;;

let position motif chaîne =
    let m = string_length motif
    and n = string_length chaîne
    in
    let rec coïncide i j =
        motif.[j] = chaîne.[i]
            && (j = m-1 || coïncide (i+1) (j+1))
    in
    let rec teste i =
        if i > n-m then raise Échec
        else if coïncide i 0 then i
        else teste (i+1)
    in
    teste 0 ;;
```

Tentons une analyse plus fine.
Supposons que s et t soient écrits sur un alphabet de taille α. Si s est de taille n et t de taille m avec $n \geq m$, et si ces mots sont aléatoires indépendants, la probabilité que les initiales de s et t soient égales vaut $\beta = 1/\alpha$. On en déduit que la taille moyenne $\ell(m, \alpha)$ du plus grand préfixe commun à s et t vaut :

$$0 \times (1-\beta) + 1 \times \beta(1-\beta) + 2 \times \beta^2(1-\beta) + \cdots + (m-1) \times \beta^{m-1}(1-\beta) + m \times \beta^m.$$

Cette somme se calcule classiquement:

$$\ell(m, \alpha) = (1-\beta) \sum_{k=1}^{m-1} k\beta^k + m\beta^m = \frac{1-\beta^m}{\alpha - 1}.$$

Par exemple, pour l'alphabet habituel des informaticiens,

$$\ell(m, 256) = \frac{1}{255} - \frac{1}{255.256^m}.$$

C'est dire que dans le cas moyen, l'algorithme naïf n'est pas si mauvais que cela, puisqu'il tourne en $O(n)$!
Remarquons toutefois que la situation considérée est irréaliste, puisqu'elle correspond au cas de chaînes complètement aléatoires, quand en pratique on gère de l'information, ce qui est radicalement différent... Si par exemple on considère la recherche d'un mot dans un texte rédigé en français, il va de soi que notre hypothèse d'équiprobabilité des caractères, et pire encore notre hypothèse d'indépendance probabiliste, tombent en défaut.

L'algorithme de Knuth, Morris et Pratt

Description de l'algorithme de Morris et Pratt

L'algorithme que nous proposons maintenant repose sur une observation à propos de l'algorithme naïf. Supposons par exemple qu'on recherche le motif `abacabac` dans la chaîne `abacacabacababaabbab`.
On cherche d'abord si le motif est en tête de la chaîne : avec les notations précédentes, pour $i = 0$, on teste les égalités $s_{i+k} = t_k$, k variant de 0 à $m-1$. L'égalité est en défaut pour $k = 5$ car $s_5 = \texttt{c}$ alors que $t_5 = \texttt{b}$. L'observation de Morris et Pratt est la suivante : on peut affirmer au moment de l'échec des comparaisons qu'on a quand même $s_i = t_0$, ..., $s_{i+k-1} = t_{k-1}$. Quand dans l'algorithme naïf, on avance à l'indice i' dans la chaîne et on sera donc amené à comparer t_ℓ à $s_{i'+\ell}$ dont on sait qu'il vaut aussi $t_{i'-i+\ell}$, si $i' + \ell < k$. En un mot, on va comparer un préfixe du motif à un suffixe de $t[0..k-1]$. Mais on peut étudier au préalable la taille du plus grand préfixe égal à un suffixe de ces tronçons du motif. Ainsi, pour reprendre notre exemple, le tronçon commun à s et t est `abaca`, et le préfixe-suffixe est `a`, ce qui nous permet de continuer la recherche en positionnant le motif sous ce dernier `a`, comme l'indique la figure 1 page suivante, qui explicite quelques étapes de l'algorithme (on indique en italique la lettre du motif qui fait échouer la comparaison).
On aura observé dans notre exemple le décalage du motif effectué après le premier échec : on a fait coïncider avec la chaîne s proposée le préfixe `abaca`

chaîne	a	b	a	c	a	c	a	b	a	c	a	b	a	b	a	a	b	b	a	b
(1)	a	b	a	c	a	*b*	a	c												
(2)					a	*b*	a	c	a	b	a	c								
(3)						*a*	b	a	c	a	b	a	c							
(4)							a	b	a	c	a	b	a	*c*						
(5)											a	b	a	*c*	a	b	a	c		
(6)													a	b	a	*c*	a	b	a	c

Figure 1: Décalages du motif dans l'algorithme de Morris et Pratt

du motif. Pour passer de la ligne (1) à la ligne (2), on décale le motif de telle sorte qu'on fait coïncider un plus grand suffixe de `abaca` en ligne (1) avec un plus grand préfixe du même mot en ligne (2). Ici ce préfixe/suffixe vaut `a`.
Plus loin, dans le passage de la ligne (4) à la ligne (5) le tronçon du motif qui coïncide avec la chaîne est `abacaba`, qui a pour préfixe/suffixe maximal le mot `aba`, ce qui explique le décalage effectué.

Les bords d'un mot On appellera bord d'un mot non vide un préfixe strict de ce mot qui est en même temps un suffixe de ce mot. Les bords de `abacaba` sont donc le mot vide ε, et les mots `a`, `aba`. Le bord maximal est le plus long des bords.
On définit alors, pour tout mot t de taille m la fonction

$$\rho : \{0, 1, \ldots, m\} \longrightarrow \{-1, 0, 1, \ldots, m-1\}$$

définie par $\rho(0) = -1$ et la condition : $\rho(i)$ est la longueur (éventuellement nulle) du bord maximal du mot $t[0..i-1]$.
Par exemple, pour le mot $t =$ `abacabac`, on obtient la fonction suivante :

x_{i-1}		a	b	a	c	a	b	a	c
i	0	1	2	3	4	5	6	7	8
$\rho(i)$	-1	0	0	1	0	1	2	3	4

Calcul des bords maximaux Je note $\beta(x)$ le bord maximal d'un mot x. $\beta^2(x)$ dénotera le bord maximal du bord maximal de x, et on définit de façon analogue $\beta^k(x)$.
Démontrons tout d'abord le

Théorème 1 (Bords d'un mot) *Soit x un mot non vide, et k le plus petit entier tel que $\beta^k(x) = \varepsilon$.*
Alors on dispose des deux résultats suivants :

— *les bords de x sont les mots $\beta(x)$, $\beta^2(x)$, …, $\beta^k(x)$;*

— *si $\mathtt{a}$ est une lettre, alors $\beta(x\mathtt{a})$ est le plus long préfixe de x qui est dans l'ensemble $\{\varepsilon\} \cup \{\beta(x)\mathtt{a}, \beta^2(x)\mathtt{a}, \ldots, \beta^k(x)\mathtt{a}\}$.*

✧ Remarquons qu'un bord étant un préfixe strict, il est clair que les tailles des mots $\beta^i(x)$ forment une suite strictement décroissante d'entiers naturels, ce qui justifie l'existence de l'entier k.

En outre, tout bord d'un bord est encore un bord, et ainsi les $\beta^i(x)$ forment donc bien des bords du mot x.

Réciproquement, si y est un bord de x, et si $y \neq \beta(x)$, alors y est un bord de $\beta(x)$. Par récurrence sur la taille des mots, on vérifie donc bien que la suite des $\beta^i(x)$ recouvre l'ensemble des bords de x.

Considérons la deuxième assertion.

Si y est un bord de xa qui n'est pas vide, alors, puisque y est aussi un suffixe de xa, y est de la forme za, où z est un bord de x, et la première assertion conclut.

Réciproquement, tout les mots proposés sont suffixes de xa, et donc un bord de xa s'ils sont également préfixes. ✦

On en déduit le théorème suivant :

Théorème 2 (Corollaire) *Soit x un mot non vide, et $\mathtt{a}$ une lettre. Alors*

$$\beta(x\mathtt{a}) = \begin{cases} \beta(x)\mathtt{a}, & \textit{si } \beta(x)\mathtt{a} \textit{ est un préfixe de } x; \\ \beta(\beta(x)\mathtt{a}), & \textit{sinon.} \end{cases}$$

✧ En effet, le premier cas est une évidence.

Dans le cas où $\beta(x)$a n'est pas un préfixe de x, on veut montrer que $\beta(x\text{a}) = \beta(y\text{a})$ où j'ai posé $y = \beta(x)$. Mais on sait depuis le théorème 1 page précédente que $\beta(x\text{a})$ est le plus long préfixe de x parmi $\beta(x)$a (ce qu'on vient d'exclure), $\beta^2(x)\text{a} = \beta(y)\text{a}$, $\beta^3(x)\text{a} = \beta^2(y)\text{a}$, etc. Réappliquant le théorème précédent, on conclut. ✦

Nous sommes maintenant en mesure de calculer la fonction ρ grâce au

Théorème 3 (La fonction ρ) *Soit x un mot non vide de longueur m. Pour $0 \leq j \leq m-1$, on a :*

$$\rho(j+1) = 1 + \rho^k(j),$$

où $k \geq 1$ est le plus petit entier vérifiant $\rho^k(j) = -1$ ou sinon $x_j = x_{\rho^k(j)}$.

On aura compris que ρ^k désigne la composée de ρ k fois par elle-même, et qu'on utilise comme Caml l'indexation à compter de l'indice 0.

Notons désormais, pour $1 \leq i \leq m$, $x|i$ le préfixe $x[0..i-1]$ constitué des i premières lettres de x. On rappelle que $\rho(i)$ est défini comme la taille de $\beta(x|i)$ si $i > 0$ et $\rho(0) = -1$.

Avant d'écrire la démonstration de ce théorème, je propose d'observer le tableau suivant, qui prolonge le précédent.

x_{i-1}		`a`	`b`	`a`	`c`	`a`	`b`	`a`	`c`
i	0	1	2	3	4	5	6	7	8
$\rho(i)$	-1	0	0	1	0	1	2	3	4
$\rho^2(i)$		-1	-1	0	-1	0	0	1	0
$\rho^3(i)$				-1		-1	-1	0	-1
$\rho^4(i)$								-1	
$\beta(x\|i)$		ε	ε	`a`	ε	`a`	`ab`	`aba`	`abac`
$\beta^2(x\|i)$				ε		ε	ε	`a`	ε
$\beta^3(x\|i)$								ε	

Cette observation conduit tout naturellement à énoncer le

Lemme 1 *Soit x un mot non vide de longueur m, i un entier vérifiant $1 \leq i \leq m$ et $j \geq 1$ un entier tel que $\rho^j(i) \geq 0$. Alors $\rho^j(i)$ est égal à la longueur de $\beta^j(x|i)$.*

✧ La démonstration se fait par récurrence sur j, le résultat étant évident quand $j = 1$.
Supposons donc que $\rho^j(i) = \text{taille}(\beta^j(x|i))$ soit non nulle, histoire que l'on ait bien $\rho^{j+1}(i) = \rho(\rho^j(i))$ différent de -1.
$\rho^{j+1}(i)$ est la taille du bord maximal du mot $y = x|\rho^j(i)$. Quel est ce mot y ? il a pour longueur $\rho^j(i)$ et c'est un préfixe du mot x, bien sûr. D'après l'hypothèse de récurrence on a donc $y = \beta^j(x|i)$. C'est terminé : le bord maximal de y est donc $\beta(y) = \beta^{j+1}(x|i)$, et $\rho^{j+1}(i)$ est bien comme on l'a déjà dit la taille de ce bord maximal de y. ✦

Passons maintenant à la démonstration du théorème 3 page précédente.

✧ Notons que le résultat à démontrer est évident pour $j = 0$, puisque $\rho(0) = -1$, donc la valeur choisie de k est $k = 1$, qui fournit bien $\rho(1) = 1 + \rho(0) = 0$: un mot réduit à 1 lettre n'a de bord que le mot vide.
Supposons donc par récurrence le résultat vérifié pour les rangs $0, 1, \ldots, j-1$, avec $1 \leq j \leq m-1$.
$\rho(j+1)$ désigne alors la taille du bord maximal $\beta(x|(j+1))$. Soit $\mathtt{a} = x_j$ la dernière lettre de $x|(j+1)$ de telle sorte qu'on a $x|(j+1) = (x|j)\mathtt{a}$. La condition $x_j = x_{\rho^k(j)}$ se traduit alors en disant que a est la lettre qui suit $\beta^k(x|j)$, d'après notre lemme. Mais c'est là exactement dire que $\beta^k(x|j)\mathtt{a}$ est un préfixe de x. Notons ici $y = x|j$. On a donc choisi le premier k tel que $\beta^k(y)\mathtt{a}$ soit un préfixe de $y\mathtt{a}$. Rappelons que $\beta^p(y)$ est plus petit que $\beta^q(y)$ si $p > q$, ce qui veut dire que choisir le premier k c'est choisir le plus long préfixe (ou le mot vide, qui correspond à $\rho^k(j) = -1$). Nous avons très exactement ici le résultat souhaité d'après la deuxième assertion du théorème 1 page 6. ✦

Implémentation en Caml

Il ne s'agit plus maintenant que de traduire en Caml ce que nous venons d'étudier, ce qui ne présente pas de difficulté particulière : on obtient le programme 3 page ci-contre.
On considérera simplement avec attention ce qui se passe quand $\rho(i)$ vaut -1.

Programme 3 L'algorithme de Morris et Pratt

```
let calcule_bords motif =
    let m = string_length motif
    in
    let r = make_vect (1 + m) (-1)
    in
    let rec calcule j k =
        if k < 0 || motif.[j-1] = motif.[k] then 1 + k
        else calcule j r.(k)
    in
    for j = 1 to m do r.(j) <- calcule j r.(j-1) done ;
    r ;;

exception Échec ;;

let position motif chaîne =
    let p = string_length motif
    and n = string_length chaîne
    and i = ref 0
    and k = ref 0
    and r = calcule_bords motif
    in
    while ! k < p && ! i < n do
        if ! k < 0 || chaîne.[! i] = motif.[! k] then (incr i ; incr k)
        else k := r.(! k)
    done ;
    if ! k >= p then ! i - p
    else raise Échec ;;
```

Évaluation

Une remarque fondamentale est la suivante : on ne revient jamais en arrière dans le balayage de la chaîne, à la différence de ce qui se passait dans l'algorithme naïf.
Une fois achevé le calcul de la fonction ρ, c'est-à-dire de la taille des bords maximaux du motif utilisé, ce qui se fait en moins de $2m$ comparaisons, on applique l'algorithme proprement dit. Il n'est pas difficile là non plus de majorer par $2n$ le nombre de comparaisons effectuées. Au total l'algorithme de Morris et Pratt tourne pour un coût inférieur à $2(n+m)$ comparaisons.

L'amélioration de Knuth

Reprenons la figure 1 page 6. Le caractère du motif qui fait échouer la comparaison en ligne (1) est le `b` qui ne correspond pas au `c` de la chaîne. C'est pourquoi Morris et Pratt évaluent le bord maximal du préfixe qui précède ce `b` dans le motif, c'est-à-dire `abaca` : le bord maximal est tout simplement `a`, et on décale en conséquence le motif, en ligne (2). La remarque de Knuth est la suivante : puisque c'est précisément encore la lettre `b` qui figure après le bord maximal, et qu'on sait bien que ce `b` a déjà mené à un test négatif, on devrait passer directement à la ligne (3).
On va donc tenir compte de cette remarque, en remplaçant la fonction ρ par une nouvelle fonction φ : on dira de deux préfixes u et v d'un mot x qu'ils sont disjoints si les lettres qui suivent u et v dans x sont distinctes ou si u ou v est égal à x tout entier. $\varphi(i)$ sera alors simplement la taille du plus grand bord de $x|i$ qui soit disjoint de $x|i$, s'il existe, et -1 sinon. Bien sûr cela n'a de sens que pour $i < m$ (m désigne toujours la taille de x), et on posera simplement $\varphi(m) = \rho(m)$. Remarquons en particulier que quand $\rho(i)$ est nul, $\varphi(i)$ vaut 0 si $x_0 \neq x_i$ et -1 sinon.
On donne dans le tableau suivant les valeurs comparées de φ et ρ sur le même motif `abacabac`.

x_{i-1}		a	b	a	c	a	b	a	c
i	0	1	2	3	4	5	6	7	8
$\rho(i)$	-1	0	0	1	0	1	2	3	4
$\varphi(i)$	-1	0	-1	1	-1	0	-1	1	4

On vérifie qu'on a toujours (évidemment !) $\varphi(i) \leq \rho(i)$: c'est le signe que Knuth améliore l'algorithme de Morris et Pratt.
Voici un autre exemple, sur un autre motif :

x_{i-1}		a	b	c	a	b	a	b	c	a	c
i	0	1	2	3	4	5	6	7	8	9	10
$\rho(i)$	-1	0	0	0	1	2	1	2	3	4	0
$\varphi(i)$	-1	0	0	-1	0	2	0	0	-1	4	0

On peut en particulier démontrer le

Théorème 4 (Calcul de φ) *Soit x un mot non vide, la fonction φ vérifie $\varphi(0) = -1$ et, pour $i \geq 1$,*

$$\varphi(i) = \begin{cases} \rho(i), & \textit{si } i = m \textit{ ou } x_i = x_{\rho(i)}; \\ \varphi(\rho(i)), & \textit{sinon.} \end{cases}$$

✧ Si $i = m$, le résultat est clair : $\varphi(m) = \rho(m)$.
Si $i < m$, posant $k = \rho(i)$, le bord maximal de $x|i$ est $\beta(x|i) = x|\rho(i) = x|k$.
Ou bien $x_i \neq x_k$ et alors il s'agit d'un bord disjoint et $\varphi(i) = k = \rho(i)$.
Ou bien $x_i = x_k$. Mais alors les bords disjoints de $x|i$ coïncident avec ceux de $x|k$, car ces deux mots partagent le même caractère sentinelle : $x_i = x_k$.
Bref, on a bien $\varphi(i) = \varphi(k)$. ✦

L'analogue du théorème 3 page 7 est le

Théorème 5 (Calcul explicite de φ) *Soit x un mot non vide de longueur m et i entier tel que $0 \leq i < m - 1$, alors*

$$\rho(1 + i) = 1 + \varphi^k(\rho(i)),$$

où k est le plus petit entier naturel tel que $\varphi^k(\rho(i)) = -1$ ou sinon $x_{\varphi^k(\rho(i))} = x_i$.

La démonstration est analogue à celle du théorème 3, et nous ne la redonnons pas.

Implémentation en Caml

Le programme 4 page suivante implémente en Caml l'algorithme de Knuth, Morris et Pratt. La seule différence avec le programme 3 page 9 est dans le calcul des bords, qui est ici fondé sur l'utilisation de φ en lieu et place de ρ.
Donnons simplement ici des assertions (invariants de boucle, si l'on veut) qui permettent de comprendre le déroulement du calcul. On vérifiera qu'avant l'exécution de la boucle `while` de la ligne 8 `!rho` vaut $\beta(j - 1)$, et qu'après l'exécution de l'incrémentation de la ligne 11 il vaut $\beta(j)$. Il suffit alors d'appliquer les théorèmes 4 et 5 pour s'assurer qu'on calcule correctement la fonction φ.

L'algorithme de Boyer, Moore et Horspool

Description de l'algorithme

L'idée de base On va là encore, bien entendu, comparer le motif t de taille m avec la chaîne s de taille n, en procédant par décalages successifs du motif le long de la chaîne. Mais ici on procède aux comparaisons de caractères entre le motif et la chaîne de *droite à gauche*.
L'air de rien, cette façon de procéder va tout changer.
Prenons un exemple, illustré par la figure 2 page 13, où, ainsi qu'on l'avait déjà fait dans la figure 1 page 6 pour l'algorithme de Morris et Pratt, on a indiqué en italiques les caractères qui font échouer la comparaison (de droite à gauche, cette fois).

Programme 4 L'algorithme de Knuth, Morris et Pratt

```
let calcule_bords motif =
    let m = string_length motif
    in
    let r = make_vect (1 + m) (-1)
    and rho = ref (-1)
    in
    for j = 1 to m do
        while ! rho >= 0 && motif.[j-1] <> motif.[! rho] do
         rho := r.(! rho)
        done ;
        incr rho ;
        r.(j) <- if  j = m || motif.[j] <> motif.[! rho]
          then ! rho
          else r.(! rho)
    done ;
    r ;;

exception Échec ;;

let position motif chaîne =
    let p = string_length motif
    and n = string_length chaîne
    and i = ref 0
    and k = ref 0
    and r = calcule_bords motif
    in
    while ! k < p && ! i < n do
        if ! k < 0 || chaîne.[! i] = motif.[! k] then (incr i ; incr k)
        else k := r.(! k)
    done ;
    if ! k >= p then ! i - p
    else raise Échec ;;
```

chaîne	a	b	a	c	a	c	a	**b**	a	**c**	a	b	a	**b**	a	**a**	**b**	b	**a**	b
(1)	a	b	a	c	a	b	a	*c*												
(2)			a	*b*	a	c	a	b	a	c										
(3)							a	b	a	c	a	b	a	*c*						
(4)									a	b	a	c	a	b	a	*c*				
(5)										a	b	a	c	a	b	a	*c*			
(6)												a	b	a	c	a	b	a	*c*	

Figure 2: Décalages du motif dans l'algorithme de Boyer, Moore et Horspool

Cette fois, nous signalons en outre en gras les caractères de la chaîne qui sont à la hauteur du dernier caractère du motif.
Le premier échec, en ligne 1, a lieu avec les caractères `c` du motif et `b` de la chaîne. C'est alors un `b` qui se trouve dans la chaîne à la hauteur du dernier caractère du motif. Or le dernier `b` à droite dans le motif est à l'index 5. On va donc, en ligne 2, décaler le motif à droite de $7 - 5 = 2$ positions.
Le second échec, en ligne 2, a lieu car $t_1 = \texttt{b} \neq \texttt{c} = s_3$. Mais peu nous chaut qu'il s'agisse de telles ou telles lettres : le seul élément déterminant est qu'il y ait eu échec. En effet, le décalage est calculé grâce à la lettre (ici $s_9 = \texttt{c}$) qui, dans la chaîne, est en face du dernier caractère du motif. On cherche ici la dernière occurrence de `c` dans le motif privé de sa dernière lettre (sans quoi on n'avancerait pas du tout) : c'est $t_3 = \texttt{c}$, et on décale donc le motif de $7 - 3 = 4$ positions. Si par exemple on avait eu $s_9 = \texttt{b}$, on aurait procédé à un décalage de $7 - 5 = 2$, et si on avait eu $s_9 = \texttt{w}$, à un décalage de 8 positions.

La fonction de décalage La fonction de décalage, appliquée au caractère de la chaîne qui est à la hauteur du dernier caractère du motif, renvoie le décalage à opérer sur le motif en cas d'échec de la comparaison.
Si le caractère n'est pas du tout dans le motif privé de sa dernière lettre, on renverra la taille du motif, puisqu'on pourra sauter toute sa largeur dans la chaîne !
Sinon, on cherche l'occurrence i la plus à droite du caractère dans le motif privé de son dernier caractère[1] et on renvoie $m - i - 1$.
Bien sûr, on calcule cette fonction de décalage une bonne fois pour toutes pour un motif donné.

Implémentation en Caml

Le plus gros problème est dans la fonction de décalage. On peut bien sûr commencer par écrire la solution du programme 5 page suivante. Mais elle demande de balayer entièrement le motif à chaque appel, ce qui est rédhibitoire.
On pense alors à l'option `remember` de Maple, ce qui donne l'idée du programme 6 page suivante. Mais là encore on est confronté à une difficulté : en effet, cette fois on doit balayer à chaque appel non plus le motif, mais la table de mémorisation de la fonction !

[1] pour éviter le sur-place...

Programme 5 Une première tentative pour la fonction de décalage

```
let decale motif c =
    let m = string_length motif
    in
    let r = ref m
    in
    for i = 0 to m - 2 do if motif.[i] = c then r := m - 1 - i done ;
   ! r ;;
```

Programme 6 Une seconde tentative pour la fonction de décalage

```
let decale motif =
    let table = ref []
    in
    function c ->
        try assoc c ! table
        with _ ->
            let m = string_length motif
            in
            let r = ref m
            in
            for i = 0 to m - 2 do if motif.[i] = c then r := m - 1 - i done ;
            table := (c,! r) :: ! table ;
           ! r ;;
```

Finalement, on utilisera la version du programme 7 page suivante qui s'inspire du hachage pour assurer un accès direct de la fonction `décale` à son résultat.

Évaluation

Il faut noter tout d'abord que l'algorithme BMH est le seul de ceux que nous avons décrits ici à ne pas avoir besoin de considérer chacun des caractères de la chaîne : tous les autres ont un coût d'au moins n comparaisons, seul BMH s'affranchit de cette contrainte.
On peut d'ailleurs démontrer plus précisément (voir [5, page 222]) le résultat suivant :

Théorème 6 (Évaluation de l'algorithme BMH) *Si l'alphabet utilisé est de taille α, la recherche d'un motif aléatoire de taille m dans une chaîne aléatoire de taille n (avec $\alpha \ll n$) par l'algorithme de Boyer-Moore-Horspool coûte en moyenne un nombre de comparaisons de l'ordre de*

$$\frac{n}{m} + \frac{n}{2\alpha}.$$

Il a été proposé différentes modifications de l'algorithme, modifiant la fonction de décalage par le calcul de nouvelles tables, un peu à la manière de l'amélioration de Morris et Pratt par Knuth, mais leur apport en termes d'efficacité n'est pas évident, la complexification du pré-traitement sur le motif n'étant pas forcément compensée par la relative accélération de la recherche proprement dite. Nous n'en parlerons pas davantage.

Programme 7 L'algorithme de Boyer, Moore et Horspool

```
exception Échec ;;

let décale motif =
    let m = string_length motif
    in
    let d = make_vect 256 m
    in
    for i = 0 to m - 2 do d.(int_of_char motif.[i]) <- m - 1 - i done ;
    d ;;

let position motif =
    let d = décale motif
    and m = string_length motif
    in
    function chaîne ->
        let n = string_length chaîne
        in
        let i = ref (m - 1)
        and j = ref (m - 1)
        in
        while ! i < n && ! j >= 0 do
            if chaîne.[! i - m + 1 + ! j] = motif.[! j] then decr j
            else
            begin
                i := ! i + d.(int_of_char chaîne.[! i]) ;
                j := m - 1
            end
        done ;
        if ! i >= n then raise Échec
        else ! i - m + 1 ;;
```

La bibliothèque `format` : pour un bel affichage

La bibliothèque `format` offre des fonctionnalités pour un bel affichage[2] et facilite ainsi l'écriture de procédures d'affichage personnalisées. La bibliothèque organise un *moteur de bel affichage* et prévoit en particulier une gestion pratique des césures de lignes (césure automatique quand nécessaire).

Boîtes

La césure des lignes se fonde sur deux concepts :

les boîtes une boîte est une unité logique de bel affichage, qui est associée à un comportement particulier du moteur de bel affichage;

les marques de coupures une marque de coupure[3] est une indication donnée au moteur d'affichage pour l'aider à choisir les endroits où il coupera les lignes. Sans ces marques, le moteur ne procèdera à aucune césure (sauf *en cas d'urgence* pour éviter des affichages vraiment trop laids). En outre, quand le moteur commence l'affichage d'une nouvelle ligne, les marques définissent des règles d'indentation de la nouvelle ligne (il s'agit des espaces initiales de la ligne).

Il y a quatre types de boîtes.
(On utilise le plus souvent le dernier type (`hovbox`), de sorte que vous pouvez sauter les autres descriptions en première lecture.)

boîtes horizontales on les obtient en appelant `open_hbox`. Dans une telle boîte, les marques n'impliquent pas de coupure effective des lignes.

boîtes verticales on les obtient en appelant `open_vbox`. Dans une telle boîte, chaque marque implique une coupure effective de la ligne.

boîtes verticales/horizontales on les obtient en appelant `open_hvbox`. Si c'est possible la boîte entière est affichée sur une seule ligne; sinon chaque marque implique une coupure effective de ligne.

boîtes horizontales ou verticales on les obtient en appelant `open_hovbox`. On utilise les marques pour trouver l'endroit où couper la ligne quand il n'y a plus de place.

Donnons un exemple.
Nous représentons chaque caractère par un `-`, `[` et `]` représentent l'ouverture et la fermeture d'une boîte, et enfin `m` représente une marque de coupure rencontrée par le moteur de bel affichage.
Selon le type de boîte choisie, la sortie `--m--m--` est affichée par le moteur de bel affichage de la bibliothèque `format` de la façon suivante, où le caractère `M` représente la valeur de la marque qui est explicitée ci-dessous.

[2] j'ai traduit ainsi l'anglais *pretty-printing*
[3] dans la suite on dira simplement *une marque*

— dans une `hbox`

```
--M--M--
```

— dans une `vbox` on obtient une coupure de ligne à chaque marque :

```
--M
--M
--
```

— dans une `hvbox`

```
--M--M--
```

s'il y a encore assez de place dans la ligne pour afficher toute cette boîte, et sinon

```
--M
--M
--
```

— dans une `hovbox`

```
--M--M--
```

s'il y a encore assez de place dans la ligne pour afficher toute cette boîte, et sinon, s'il y a assez de place pour cela (on ne coupe donc pas la ligne à la première marque, mais seulement à la seconde) :

```
--M--M
--
```

voire, si vraiment on dispose de peu de place avant la marge :

```
--M
--M
--
```

Espaces

Les marques de coupures sont également utilisées pour afficher des espaces (s'il n'y a pas de coupure effective de la ligne — c'est un espacement suffisant — évidemment).
Vous insérez une marque de coupure en appelant `print_break(sp,indent)`, ce qui permettra l'insertion de `sp` espaces (`sp` doit donc avoir une valeur entière). Bref, on peut interpréter `print_break(sp,...)` par : afficher `sp` espaces ou couper la ligne.
En général, une procédure d'affichage utilisant `format` ne doit pas afficher d'espaces proprement dits, mais utiliser des marques de coupures.
C'est ainsi par exemple que `print_space()` est une abréviation pratique pour `print_break(1,0)` et affiche une espace simple ou provoque une coupure de ligne. Notons à l'occasion l'existence de `print_cut()`, synonyme bien pratique de `print_break(0,0)`, qui crée une marque de coupure sans ajouter d'espace.

Indentation

L'utilisateur a deux possibilités pour fixer l'indentation des nouvelles lignes :

— à la définition de la boîte où apparaît la coupure de ligne : à l'ouverture d'une boîte, vous pouvez définir l'indentation ajoutée à chaque nouvelle ligne créée dans la boîte. Par exemple `open_hovbox 1` ouvre une boîte de type `hovbox` où chaque nouvelle ligne sera indentée avec une espace supplémentaire. C'est ainsi qu'avec la sortie `---[--m--m--m--`, on affichera :

```
---[--M--M
     --M--
```

alors qu'avec `open_hovbox 2` on aurait obtenu :

```
---[--M--M
      --M--
```

(Remarque : le caractère `[` n'est pas affiché, il est juste indiqué ici pour vous montrer où commence la nouvelle boîte.)

— à la définition d'une marque de coupure. Comme déjà dit, vous insérez une marque en appelant `print_break(sp,indent)`. L'entier `indent` est utilisé pour fixer l'indentation de la nouvelle ligne. Plus précisément, le moteur de bel affichage ajoute `indent` espaces à l'indentation de la boîte où apparaît la marque. Par exemple, dans une boîte ouverte par `open_hovbox 1`, si `m` représente `print_break(1,2)` et si la première marque ne provoque pas de coupure de ligne, mais que les deux suivantes donnent lieu à une césure, la sortie `---[--m--m--m--` sera affichée de la façon suivante :

```
---[-- --
       --
       --
```

En pratique

Quand vous écrivez une procédure de bel affichage, suivez ces quelques règles simples :

1. Les boîtes doivent être ouvertes et refermées de façon cohérente (`open_*` et `close_box` doivent être bien balancés).

2. Ne jamais hésiter à ouvrir une boîte.

3. Insérer de nombreuses marques de coupure, sans quoi le moteur de bel affichage sera dans une situation critique où il essaiera certes de faire de son mieux, mais n'oubliez pas que son meilleur résultat sera alors pire que votre plus mauvais.

4. Ne forcez pas des coupures de ligne : laissez cette responsabilité au moteur de bel affichage; c'est d'ailleurs son seul travail.

5. Terminez votre programme principal par un `print_newline()` qui vide les tables du moteur de bel affichage (et donc ses sorties).

Exemple

Tout d'abord voici une syntaxe abstraite des λ-termes, puis un analyseur lexical et un analyseur syntaxique pour ce langage :

```
type lambda =
    | Lambda of string * lambda
    | Var of string
    | Apply of lambda * lambda ;;

(* Analyseur lexical *)

#open "genlex" ;;
let lexer = make_lexer [ "." ; "\\" ; "(" ; ")" ] ;;

(* Analyseur syntaxique *)

let rec exp0 = function
    | [< 'Ident s >] -> Var s
    | [< 'Kwd "(" ; lambda lam ; 'Kwd ")" >] -> lam

and app = function
      [< exp0 e ; (other_applications e) lam >] -> lam

and other_applications f = function
    | [< exp0 arg ; stream >] -> other_applications (Apply(f,arg)) stream
    | [< >] -> f

and lambda = function
    | [< 'Kwd "\\" ; 'Ident s ; 'Kwd "." ; lambda lam >] -> Lambda(s,lam)
    | [< app e >] -> e ;;

let parse_lambda s = lambda (lexer (stream_of_string s)) ;;
```

Essayons cet analyseur :

```
#parse_lambda "(\\ x.x)" ;;
- : lambda = Lambda ("x", Var "x")
```

Maintenant, nous utilisons la bibliothèque `format` pour afficher les λ-termes : je suis la structure récursive de l'analyseur syntaxique pour écrire les procédures d'affichage, insérant ça et là les marques de coupures nécessaires et ouvrant et fermant des boîtes :

```
#open "format" ;;

let ident = print_string
```

```
and kwd = print_string ;;

let rec print_exp0 = function
    | Var s -> ident s
    | lam -> open_hovbox 1 ; kwd "(" ; print_lambda lam ; kwd ")" ;
close_box()

and print_app = function
    | e -> open_hovbox 2 ; print_other_applications e ; close_box()

and print_other_applications = function
    | Apply(f,arg) -> print_app f ; print_space() ; print_exp0 arg
    | f -> print_exp0 f

and print_lambda = function
    | Lambda(s,lam) -> open_hovbox 1 ;
                       kwd "\\" ; ident s ; kwd "." ;
                       print_space() ; print_lambda lam ;
                       close_box()
    | e -> print_app e ;;
```

Nous obtenons maintenant :

```
#print_lambda (parse_lambda "(\\ x.x)") ;;
\x. x- : unit = ()
```

Remarquer la gestion correcte des parenthèses ; l'afficheur écrit le minimum de parenthèses utile à une réutilisation correcte du résultat par le parseur :

```
#print_lambda (parse_lambda "(x y) z") ;;
x y z- : unit = ()
#print_lambda (parse_lambda "x y z") ;;
x y z- : unit = ()
```

Si vous utilisez cet afficheur pour corriger des programmes dans l'environnement interactif, déclarez le en appelant `install_printer`, de telle sorte que la boucle interactive de Caml l'utilise pour afficher ses résultats :

```
#install_printer "print_lambda" ;;
- : unit = ()
#parse_lambda "(\\ x . (\\ y . x y))" ;;
- : lambda = \x. \y. x y
#parse_lambda "((\\ x . (\\ y . x y)) (\\ z . z))" ;;
- : lambda = (\x. \y. x y) (\z. z)
```

Tout cela fonctionne très bien en conjonction avec les possibilités de trace du système interactif (d'ailleurs je trouve indispensable d'utiliser un afficheur fondé sur `format` pour obtenir une trace lisible). Nous *traçons* ici la fonction `lambda`, c'est-à-dire que nous demandons à l'environnement interactif de l'*espionner* pour nous.

```
#trace "lambda" ;;
La fonction lambda est dorénavant tracée.
- : unit = ()
#parse_lambda
"((\\ ident . (\\ autre_ident . ident autre_ident))
 (\\ truc . truc truc)) (\\ machin . (machin machin) machin) " ;;
lambda <-- <abstr>
lambda <-- <abstr>
lambda <-- <abstr>
lambda <-- <abstr>
lambda <-- <abstr>
lambda <-- <abstr>
lambda --> ident autre_ident
lambda --> \autre_ident. ident autre_ident
lambda --> \autre_ident. ident autre_ident
lambda --> \ident. \autre_ident. ident autre_ident
lambda <-- <abstr>
lambda <-- <abstr>
lambda --> truc truc
lambda --> \truc. truc truc
lambda --> (\ident. \autre_ident. ident autre_ident) (\truc. truc truc)
lambda <-- <abstr>
lambda <-- <abstr>
lambda <-- <abstr>
lambda --> machin machin
lambda --> machin machin machin
lambda --> \machin. machin machin machin
lambda --> (\ident. \autre_ident. ident autre_ident) (\truc. truc truc)
             (\machin. machin machin machin)
- : lambda =
 (\ident. \autre_ident. ident autre_ident) (\truc. truc truc)
   (\machin. machin machin machin)
```

Références

[1] Danièle Beauquier, Jean Berstel, et Philippe Chrétienne. *Éléments d'algorithmique.* Masson, Paris, 1992.

[2] Gilles Brassard and Paul Bratley. *Algorithmics, theory and practice.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[3] Philippe Flajolet and Robert Sedgewick. *An introduction to the analysis of algorithms.* Addison-Wesley, Reading, Massachussets, 1996.

[4] Niklaus Wirth. *Algorithmes et structures de données.* Eyrolles, Paris, 1986.

[5] Derick Wood. *Data structures, algorithms, and performance.* Addison-Wesley, Reading, Massachussets, 1993.