TP Caml 8: Recherche de motif dans un texte

lionel.rieg@ens-lyon.fr

3 & 10 mars 2009

On s'intéresse ici à la recherche d'un motif (souvent un mot) dans un texte, comme par exemple lors une recherche dans un fichier PDF ou une page web avec `Ctrl-F`. On prend les conventions et notations suivantes :

- l'alphabet considéré ici est celui du codage `ASCII` (celui de Caml) contenant 256 caractères ; on peut accéder à l'indice de chaque caractère dans l'énumération par la fonction int_of_char ; la réciproque étant char_of_int
- le motif et le texte seront donnés comme deux chaînes de caractères
- le texte sera t, de longueur $|t|$; le motif sera p, de longueur $|p|$

1 Recherche sans prétraitement

1.1 Algorithme naïf

► **Question 1** Écrire un algorithme qui cherche si le motif apparaît à une position donnée dans le texte. Il sera de type

```
occur_check : string → string → int → bool
```

► **Question 2** Écrire un algorithme naïf de recherche d'un motif dans un texte, de type

```
naive_search : string → string → bool
```

► **Question 3** Modifier l'algorithme précédent pour qu'il renvoie la liste des positions des occurrences.

► **Question 4** Quelle est la complexité de cet algorithme dans le pire cas ? Un exemple du pire cas ? Et en moyenne, une idée ?

1.2 Algorithmes Shift-And et Shift-Or

On va améliorer l'algorithme précédent en effectuant simultanément lors de la lecture d'une lettre toutes les recherches où cette lettre peut intervenir dans le motif, ceci en simulant un automate non déterministe qui recherche le motif.

► **Question 5** Écrire un automate non déterministe à $|p| + 1$ états sans ε-transitions et à un seul état initial qui recherche le motif p.

Dans cet automate, on remarque d'une part qu'on autorise le motif à commencer après chaque lettre du texte et d'autre part que si la lettre lue ne permet pas d'avancer à l'état suivant dans l'automate, on « perd » l'état courant. Afin d'éviter le non déterminisme, on va plutôt simuler l'automate (déterministe) des parties qui correspond. Pour ce faire, on va représenter par des bits sur un mot machine E les différents états dans lesquels l'automate peut être en un point du texte. Le bit de poids 2^i de E représentera l'état $i + 1$ de l'automate : il vaudra 1 si l'état $i + 1$ est possible, 0 sinon. L'inconvénient de cette technique est qu'elle limite le nombre de caractères du motif à la longueur d'un mot machine. La simulation des transitions se fait alors par décalage puis filtrage des états avec un masque qui dépend des positions dans le motif de la lettre lue : vu comme un entier, E devient $2E + 1$ puis on effectue un « et » logique bit à bit (land). Voir Fig. 1 l'exécution de l'algorithme sur le mot « annale », recherché dans le texte « annale annuelle » :

► **Question 6** Implémenter le calcul des masques des lettres de l'alphabet pour un motif p donné en argument. Il sera de type :

```
sa_table : string → int vect
```

► **Question 7** Implémenter l'algorithme Shift-And. Il renverra la liste des positions où le motif apparaît.

L'algorithme utilise trois opérations lors la lecture d'une lettre : le décalage d'un bit et l'ajout du 1 à droite à l'état de l'automate puis le « et » avec le masque de bit correspondant à la lettre. On peut en économiser une en inversant le rôle des 1 et des 0 : il n'y a plus besoin dans ce cas d'ajouter un 1 à l'état de l'automate après son décalage. Cette variante est appelée algorithme Shift-Or.

► **Question 8** Implémenter l'algorithme Shift-Or. (le « ou » bit à bit se fait avec lor)

Table des caractères pour le motif « annale » :

a	n	l	e	*
001001	000110	010000	100000	000000

1	$2E+1$	000001	4	$2E+1$	001001	7	$2E+1$	000001	10	$2E+1$	000101	13	$2E+1$	000001
	‘a‘	001001		‘a‘	001001		‘ ‘	000000		‘n‘	000110		‘l‘	000000
	E'	000001		E'	001001		E'	000000		E'	000100		E'	000000
2	$2E+1$	000011	5	$2E+1$	010011	8	$2E+1$	000001	11	$2E+1$	001001	14	$2E+1$	000001
	‘n‘	000110		‘l‘	010000		‘a‘	001001		‘u‘	000000		‘l‘	000000
	E'	000010		E'	010000		E'	000001		E'	000000		E'	000000
3	$2E+1$	000101	6	$2E+1$	100001	9	$2E+1$	000011	12	$2E+1$	000001	15	$2E+1$	000001
	‘n‘	000110		‘e‘	100000		‘n‘	000110		‘e‘	100000		‘e‘	100000
	E'	000100		E'	100000		E'	000010		E'	000000		E'	000000

FIG. 1 – Recherche du motif « annale » dans le texte « annale annuelle »

2 Recherche avec prétraitement du motif

Si on autorise un traitement préalable du texte ou du motif, on peut obtenir des complexités meilleures, en pire cas et même en moyenne ! Sans aller jusqu'aux meilleurs algorithmes théoriques (on peut chercher un motif quelconque en temps linéaire et en espace constant !) assez compliqués, on va voir ici deux des algorithmes utilisés en pratique.

2.1 Algorithme de Boyer-Moore-Horspool

La complexité en $O(|t| \cdot |p|)$ de l'algorithme naïf vient du fait que le décalage dans le texte ne se fait que d'une seule lettre à cause des possibilités de chevauchement. On veut améliorer cela, au moins en moyenne, pour obtenir une complexité sous-linéaire. Pour cela, on va lire le mot depuis la fin, de façon à pouvoir sauter une partie du motif non encore lue en cas d'erreur.

En détail, on utilise une table de saut dite *table du mauvais caractère* qui va donner une heuristique de la longueur du saut que l'on peut effectuer pour placer la nouvelle fin de mot. Pour une lettre α, la table contient la longueur du saut qui permet d'aligner α avec sa dernière occurrence dans le motif : c'est la distance entre la dernière lettre du motif et la position de la dernière occurrence de α dans le motif, sauf pour la dernière lettre du motif pour laquelle c'est l'avant dernière occurrence qui est prise en compte (exercice : pourquoi ?). Lorsqu'une lettre n'apparaît pas dans le motif, on considère qu'elle apparaît avant la première lettre du motif.

Par exemple, pour le motif « annale », la table sera (« * » représente toute autre lettre) :

a	n	l	e	*
2	3	1	6	6

► **Question 9** Écrire une fonction qui prend un motif et renvoie la table du mauvais caractère associée.

► **Question 10** Écrire l'algorithme de Boyer-Moore-Horspool. Quelle est sa complexité ?

Une version plus compliquée de cet algorithme permet d'assurer une complexité linéaire dans le pire cas.

2.2 Algorithme de Knuth-Morris-Pratt

Cet algorithme construit implicitement un automate non déterministe avec ε-transitions qui reconnaît le motif. Malgré le non déterminisme, on obtient quand même au final une complexité en $O(|t| + |p|)$. L'idée est de connaître pour chaque préfixe du motif qui correspond au texte, le plus long décalage que l'on peut effectuer dans le motif sans rater d'occurrences, c'est à dire qu'on cherche le *plus long préfixe (strict) d'un préfixe w du motif qui en soit aussi un suffixe*, qu'on appelle le *bord* de w. On note φ la fonction qui a un mot associe son bord.

► **Question 11** Si α est une lettre et w un mot, quel est le lien entre $\varphi(w\alpha)$ et $\varphi(w)\alpha$, $\varphi(\varphi(w))\alpha$, ... ?

► **Question 12** Écrire un algorithme qui, pour chaque position $i \in [\![1, |p|]\!]$ dans le motif p, calcule la longueur de son bord, *i.e.* $|\varphi(p_0 p_1 \dots p_i)|$, taille du plus long préfixe strict de $p_0 p_1 \dots p_i$ qui en soit aussi un suffixe. Il sera de type :

```
failure_function : string → int vect
```

Cette fonction d'échec donne la longueur du saut que l'on effectue lorsqu'on rencontre une lettre qui ne correspond pas : on aligne la partie déjà lue du motif avec son bord et on reprend la lecture à la lettre qui fait échec.

► **Question 13** Écrire l'algorithme de Knuth-Morris-Pratt.

3 Solutions

▸ **Question 1**

```
let occur_check t p i =
  let res = ref true in
    for k = 0 to string_length p - 1 do
      res := !res && p.[k] = t.[i + k]
    done;
    !res ;;
```

▸ **Question 2**

```
let naive_search t p =
  let res = ref true in
  let i = ref 0 in
    while !res && !i < string_length t - string_length p do
      incr i;
      res := occur_check t p !i
    done;
    !res ;;
```

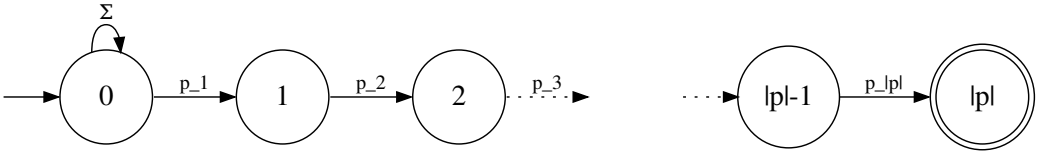
▸ **Question 3**

```
let naive_find p t =
  let res = ref [] in
  let i = ref 0 in
    while !i <= string_length t - string_length p do
      if occur_check t p !i then res := !i :: !res;
      incr i
    done;
    rev !res ;;
```

▸ **Question 4** La complexité en pire cas est $O(|t| \cdot |p|)$. En effet, si on note i la position de début de motif qu'on est en train d'inspecter et j la lettre du motif qu'on lit, la quantité $i \cdot |p| + j$ augmente strictement à chaque nouvelle lettre lue. Comme elle est majorée par $|t| \cdot |p| + |p|$, l'algorithme termine en au plus autant d'étapes. Le pire cas se produit lorsque le motif doit être presque totalement lu avant de déceler une erreur, et ce pour toutes les positions du texte, par exemple pour $p = a^5b$ et $t = a^{100}b$, qui se généralise bien évidemment à d'autres valeurs que 5 et 100.

La complexité en moyenne est plus difficile à établir. Elle nécessite de se donner une distribution de probabilité sur les textes. Si on prend une distribution uniforme pour chaque lettre du texte, la complexité est linéaire : cela se comprend en remarquant qu'obtenir k lettres correctes ne se produit qu'avec probabilité $(\frac{1}{\Sigma})^k$ et que donc la plupart des erreurs seront effectuées au tout début de la lecture.

▸ **Question 5** Il contient une transition par lettre pour avancer dans le motif plus une transition de l'état initial vers lui-même par toute lettre. On autorise ainsi avec le non déterminisme le motif à commencer à chaque nouvelle lettre lue.



Intuitivement, son fonctionnement est d'attendre le début du motif en restant dans l'état 0 puis de le lire complètement et d'arriver dans l'état $n = |p|$. Le non déterminisme permet « d'attendre le bon moment ».

▸ **Question 6**

```
let sa_table p =
  let table = make_vect 256 0 in
    for i = 0 to string_length p - 1 do
      let index = int_of_char p.[i] in
        table.(index) <- table.(index) + 1 lsl i
    done;
    table ;;
```

▸ **Question 7** Il y a deux méthodes pour détecter la présence d'une occurrence du motif : la première (présentée ici) consiste à tester une inégalité pour voir si on est allé assez loin dans le motif mais nécessite de remettre à zéro le bit correspondant ; la seconde (utilisée dans Shift-Or) consiste à tester spécifiquement la valeur du bon bit en utilisant un land et évite la remise à zéro du bit.

```
let shift_and p t =
  let shift_p = string_length p - 1 in
  let table = sa_table p in
  let is_occur = 1 lsl shift_p in
  let occur = ref [] in
  let state = ref 0 in
    for i = 0 to string_length t - 1 do
      state := ((! state lsl 1) lor 1) land table.(int_of_char t.[i]);
      if ! state >= is_occur
      then
        begin
          occur := (i - shift_p) :: !occur;
          state := ! state - is_occur
        end
    done;
    rev !occur ;;
```

▸ **Question 8** Il faut aussi refaire sa_table. Mais comme l'algorithme consiste à inverser « 0 » et « 1 », il suffit de faire de même pour la table :

```
let so_table p = map_vect lnot (sa_table p);;
```

On adapte ensuite l'algorithme Shift-And :

```
let shift_or p t =
  let shift_p = string_length p - 1 in
  let table = so_table p in
  let is_occur = 1 lsl shift_p in
  let occur = ref [] in
  let state = ref (lnot 0) in
    for i = 0 to string_length t - 1 do
      state := (! state lsl 1) lor table.(int_of_char t.[i]);
      if ! state land is_occur = 0 then occur := (i - shift_p) :: !occur
    done;
    rev !occur ;;
```

▸ **Question 9**

```
let firsts p =
  let m = string_length p in
  let chars = make_vect 256 m in
    for i = 0 to m - 2 do
      chars.(int_of_char p.[i]) <- m - i - 1
    done;
    chars ;;
```

▸ **Question 10**

```
let boyer_moore_horspool p t =
  let p_size = string_length p - 1 in
  let shifts = firsts p in
  let rec read acc i =
    let k = ref p_size in
      if i + !k >= string_length t then acc
      else
        begin
          while !k >= 0 && p.[!k] = t.[i + !k] do decr k done;
          if !k < 0 then read (i :: acc) (i + 1) (* occurrence found *)
          else
            let table_shift = shifts.(int_of_char t.[i + !k]) + !k - p_size in
              read acc (i + max 1 table_shift)
        end
  in
    rev (read [] 0);;
```

Voici également une version impérative où l'on voit plus facilement quelles sont les variables modifiées suivant le résultat du test et la valeur de la table de saut.

```
let boyer_moore_horspool p t =
  let p_size = string_length p - 1 in
  let shifts = firsts p in
  let occur = ref [] in
  let i = ref 0 in
    while !i <= string_length t - string_length p do
      let k = ref p_size in
        while !k >= 0 && p.[!k] = t.[!i + !k] do decr k done;
        if !k < 0 (* occurrence found *)
        then
          begin
            occur := !i :: !occur;
            incr i
          end
        else
          let table_shift = shifts.(int_of_char t.[!i + !k]) + !k - p_size in
            i := !i + max 1 table_shift
    done;
    rev !occur;;
```

La complexité en $O(|t| \cdot |p|)$ du pire cas se réalise comme pour l'algorithme naïf avec $p = a^n b$ et $t = a^m b$. En revanche, pour la complexité en moyenne, si on reprend l'argument utilisé pour l'algorithme naïf, l'erreur intervient au début de la lecture du motif, *i.e.* à la fin du motif et il y a peu de chance pour que la lettre qui fait conflit réapparaisse dans le motif, de sorte qu'on peut souvent sauter $|p|$ lettres du texte sans les lire. Et en effet, la complexité en moyenne de l'algorithme de Boyer-Moore-Horspool est en $O(|t|/|p|)$!

▸ **Question 11** Considérons le bord $\varphi(w\alpha)$ de $w\alpha$. C'est un préfixe et un suffixe de $w\alpha$ donc en retirant la dernière lettre (α), on obtient un préfixe et suffixe de w, qui est donc plus petit que son bord $\varphi(w)$. Si $\varphi(w)\alpha$ est préfixe de $w\alpha$, c'est gagné : $\varphi(w\alpha) = \varphi(w)\alpha$. Sinon, on doit regarder les préfixes et suffixes de w plus court que $\varphi(w)$. Ce seront donc des préfixes et suffixes de $\varphi(w)$. Le même raisonnement conduit alors à regarder $\varphi((\varphi(w))\alpha$, puis si cela ne convient pas, $\varphi((\varphi(w))\alpha$, *etc*.

▸ **Question 12**

```
let rec filling_step p phi i k =
  if k = 0 && p.[i] <> p.[k] then 0
  else
    if p.[i] = p.[k] then k + 1
    else filling_step p phi i phi.(k);;
```

```
let failure_function p =
  let phi = make_vect (string_length p) 0 in
    for i = 1 to string_length p - 1 do
      phi.(i) <- filling_step p phi i phi.(i - 1)
    done;
    phi;;
```

▸ **Question 13** Il ne faut pas oublier que lorsque $\varphi(w)$ est le mot vide le traitement c'est pas le même : on avance dans le texte plutôt que de reculer dans le motif.

```
let rec kmp_rec acc p phi t i k =
  if i = string_length t then rev acc
  else
    if p.[k] = t.[i]
    then
      if k = string_length p - 1 (* occurrence found ? *)
      then kmp_rec ((i + 1 - string_length p) :: acc) p phi t i phi.(k)
      else kmp_rec acc p phi t (i + 1) (k + 1)
    else
      if k = 0 (* phi is undefined if k = 0 *)
      then kmp_rec acc p phi t (i + 1) k
      else kmp_rec acc p phi t i phi.(k);;

let kmp p t = kmp_rec [] p (failure_function p) t 0 0;;
```

Voici à nouveau une version impérative où l'on peut voir quelles variables sont modifiées dans chaque cas.

```
let kmp p t =
  let phi = failure_function p in
  let k = ref 0 in
  let i = ref 0 in
  let occur = ref [] in
    while !i < string_length t do
      if p.[!k] = t.[!i]
      then
        if !k = string_length p - 1 (* occurrence found ? *)
        then
          begin
            k := phi.(!k);
            occur := (!i - string_length p + 1) :: !occur
          end
        else
          begin
            incr i;
            incr k
          end
      else
        if !k = 0 (* phi is undefined if k = 0 *)
        then incr i
        else k := phi.(!k)
    done;
    rev !occur;;
```

Sa complexité s'établit en remarquant que si on note i la position de début de motif qu'on est en train d'inspecter et j la lettre du motif qu'on lit, la quantité $2i - j$ augmente strictement à chaque nouvelle lettre lue. Elle est majorée par $2|t|$ donc l'algorithme est linéaire (le prétraitement du motif est linéaire par le même argument).