Caml-TP3 : Recherche de sous-mots

Paul Melotti (paul.melotti@ens.fr)

17 octobre 2013 - 7 novembre 2013

L'objectif de ce TP est d'étudier plusieurs algorithmes de recherche de mots dans un texte. On dit qu'un mot `m` figure dans un texte `t` s'il existe une position $i \leq |t| - |m|$ telle que

$$m_0 = t_i,\ m_1 = t_{i+1},\ \ldots,\ m_{|m|-1} = t_{i+|m|-1}.$$

Nos algorithmes chercheront la plus petite position de `m` dans `t`, si elle existe [1]. Dans le cas où `m` ne figure pas dans `t`, ils devront renvoyer -1.

On pourra représenter les lettres de l'alphabet A par les entiers $0, 1, \ldots, |A| - 1$, et les mots `m` et `t` par des vecteurs. On déclare donc les types : `type lettre == int ;; type mot == lettre vect ;;` (si ces types ne vous plaisent pas, libre à vous d'en utiliser d'autres).

1 Algorithme naïf

L'idée la plus simple est de tester successivement toutes les positions `i` jusqu'à trouver une occurrence de `m`.

Q.1 Écrire une fonction `naif : mot -> mot -> int` qui résout le problème par l'algorithme naïf. Quelle est sa complexité ?

Q.2 Peut-on améliorer cet algorithme si on suppose que toutes les lettres de `m` sont différentes ?

2 Hachage (algorithme de Rabin-Karp)

On va utiliser une fonction de hachage des mots de longueur $|m|$, c'est-à-dire une fonction `h : mot -> int` qui transforme un mot en entier de $[0; p-1]$ où p est un entier bien choisi.

Pour chaque position `i`, on va d'abord tester si $h(t_i \ldots t_{i+|m|-1}) = h(m)$, et seulement dans ce cas on testera si $t_i \ldots t_{i+|m|-1} = m$.

Q.1 Quelle(s) propriété(s) la fonction h doit-elle posséder pour que cette méthode présente un intérêt ?

À cet égard, une bonne fonction h est donnée par

$$h(u_0 u_1 \ldots u_{|m|-1}) = \sum_{i=0}^{|m|-1} q^i u_{|m|-1-i} \mod p$$

où q est une base choisie, par exemple $|A|$. Autrement dit, on réprésente la chaîne $u_0 u_1 \ldots u_{|m|-1}$ comme un entier en base q, et on réduit cet entier modulo p. Dans notre cas, on prendra $q = |A| = 10$ et $p = 17$.

Q.2 Quelle relation lie $h(t_i \ldots t_{i+|m|-1})$ à $h(t_{i+1} \ldots t_{i+|m|})$?

Q.3 Écrire une fonction `rabinkarp : mot -> mot -> int` qui résout le problème initial par l'algorithme de Rabin-Karp.

Cet algorithme est particulièrement utile quand on recherche plusieurs chaînes de caractères dans un même texte.

1. On pourrait aussi rechercher toutes les occurrences de `m` dans `t`. La plupart des algorithmes seraient faciles à modifier pour répondre à ce problème, vous pourrez essayer de le faire si vous avez le temps.

3 Automates (algorithme de Knuth-Morris-Pratt)

L'objectif est de fabriquer un automate déterministe complet (ADC) qui reconnait le langage A^*m, puis de faire passer le texte `t` dans cet automate. À chaque fois qu'on passe par un état final de l'automate, on a une occurrence de m.

3.1 Prolégomènes

Q.1 Construire à la main un ADC reconnaissant A^*m pour $m =$ "aabaac".

Q.2 Pourquoi veut-on que l'automate soit déterministe complet ? Proposer une structure de données pour représenter un ADC. On notera `adc` ce type.

Q.3 En supposant notre automate construit, écrire une fonction `position : adc -> mot -> int` qui, appliquée à l'ADC reconnaissant A^*m et au mot `t`, résout le problème initial.

3.2 Construction de l'ADC

On définit le *bord* d'un mot u comme le plus grand mot différent de u qui soit à la fois préfixe et suffixe de u. On le note $B(u)$. Par exemple, $B(ababa) = aba$.

On construit un ADC de la façon suivante :

- Ses états sont les préfixes de m ; il y en a $|m| + 1$, en comptant le mot vide et m.
- Si u est un préfixe de m et a une lettre, l'automate contient la transition $u \xrightarrow{a} ua$ si ua est un préfixe de m, $u \xrightarrow{a} B(ua)$ sinon.
- Son état initial est le mot vide, son état final est m.

Q.1 Montrer que cet automate reconnaît A^*m.

Q.2 Montrer que

$$B(ua) = \begin{cases} B(u)a & \text{si } B(u)a \text{ est un préfixe de } u \\ B(B(u)a) & \text{sinon.} \end{cases}$$

3.3 Implémentation

On représentera l'état correspondant au préfixe de m de taille i par l'entier i. La table de transition sera une matrice de taille $(|m| + 1) \times |A|$: la matrice vaut j à l'indice i, a ssi on a la transition $m_0 \ldots m_{i-1} \xrightarrow{a} m_0 \ldots m_{j-1}$. La question précédente fournit un moyen de remplir cette matrice, et donc de fabriquer l'ADC.

Q.1 Écrire une fonction `automate : mot -> adc` qui construit l'ADC reconnaissant A^*m par cette méthode.

Q.2 En déduire une fonction `kmp : mot -> mot -> int` qui résout le problème initial par l'algorithme de Knuth-Morris-Pratt. Quelle est sa complexité ?

4 Une bonne idée : l'astuce de Boyer-Moore

Pour comparer les mots m et $t[i..i + |m| - 1]$, nos précédents algorithmes commençaient par comparer les lettres m_0 et t_i. Boyer et Moore ont eu l'idée de commencer par comparer $m_{|m|-1}$ et $t_{i+|m|-1}$. Par exemple, si la lettre $t_{i+|m|-1}$ n'apparaît pas dans m, on est sûr que m ne sera présent à aucune des positions $i, i+1, \ldots i+|m|-1$, et on peut sauter $|m|$ cases en essayant directement de trouver m à la position $i + |m|$.

Plus précisément, on va tester dans l'ordre l'égalité des lettres

$$m_{|m|-1} = t_{i+|m|-1},\ \ldots,\ m_1 = t_{i+1},\ m_0 = t_i.$$

En cas d'échec, notons $m_k \neq t_{i+k}$ la première différence constatée. Soit m_{k-p} la dernière occurrence de la lettre t_{i+k} dans m avant la position k (par convention, $k - p = -1$ si la lettre t_{i+k} ne figure pas dans m avant la position k). On remplace alors i par $i + p$ et on reprend les tests à cette nouvelle position.

Q.1 Faire tourner l'algorithme à la main pour chercher le sous-mot `"tori"` dans le mot `"torodusumotori"`.

Q.2 Justifier la validité de l'algorithme proposé.

Q.3 Écrire une fonction `bm : mot -> mot -> int` qui résout le problème initial par l'algorithme de Boyer-Moore. Il faudra notamment trouver un moyen de connaître rapidement la dernière position d'une lettre a dans le mot m avant la position k. Quelle est la complexité dans le pire des cas ?

5 Questions supplémentaires pour faire bonne mesure

Q.1 Dans l'algorithme de Knuth-Morris-Pratt, l'automate construit est-il minimal ?

Q.2 Dans l'algorithme naïf, évaluer la complextié en moyenne : si on fixe le texte t, l'indice i et la longueur $l = |m|$ du mot cherché, et que m est un mot aléatoire pouvant valoir n'importe quel mot de longueur l de manière équiprobable, on évalue la quantité

$$\frac{1}{|A|^l} \sum_{m \text{ tq } |m|=l} N(t, m, i)$$

où $N(t, m, i)$ est le nombre de comparaisons effectuées pour savoir si $m = t[i..i + l - 1]$.

On montrera que cette quantité est plus petite que $\frac{|A|}{|A|-1}$.

Remarquez que le même calcul vaut si m est fixé et t est aléatoire.

6 Problème ouvert

Après avoir choisi un type pour représenter un automate quelconque, coder en Caml l'algorithme de déterminisation vu en cours.

On pourra utiliser le module "set", disponible en Caml Light, qui permet de manipuler des ensembles. On commencera par lancer `#open "set";;`[2] qui permet de charger les fonctions de ce modules. On disposera alors du type `'a t` qui correspond à un ensemble dont les éléments sont de type `'a`. Quelques fonctions de ce module sont :

- `empty: ('a -> 'a -> int) -> 'a t` qui crée un ensemble vide ordonné par la relation d'ordre passée en argument. Cette-dernière est une fonction `f : 'a -> 'a -> int` telle que `f a b` renvoie 0 si $a = b$, est strictement négative si $a < b$ et est strictement positive si $a > b$. Par exemple, pour des `int t`, on pourra utiliser `f = fun a b -> a - b`.
- `is_empty: 'a t -> bool` teste si un ensemble est vide.
- `mem: 'a -> 'a t -> bool` est le test d'appartenance d'un élément.
- `add: 'a -> 'a t -> 'a t` ajoute un élément.
- `remove: 'a -> 'a t -> 'a t` enlève un élément, laisse l'ensemble inchangé si l'élément n'est pas présent.
- `union: 'a t -> 'a t -> 'a t` réalise l'union de deux ensembles.
- `inter: 'a t -> 'a t -> 'a t` réalise l'intersection de deux ensembles.
- `diff: 'a t -> 'a t -> 'a t` réalise la différence de deux ensembles.
- `equal: 'a t -> 'a t -> bool` dit si deux ensembles sont égaux.
- `choose: 'a t -> 'a` renvoie un élément quelconque de l'ensemble, lève l'exception `Not_found` si l'ensemble est vide.
- `elements: 'a t -> 'a list` renvoie la liste des éléments d'un ensemble.

Vous pouvez aussi coder votre propre type `ensemble` et les fonctions nécessaires...

2. Sans charger ce module, on peut aussi accéder à ses fonctions en écrivant leur nom précédé de `set__` .