Université de Nice-Sophia Antipolis — Licence 3 Informatique
UE Automates & Langages — 2011–2012

TP n°2

Reconnaissance de motifs

Présentation

La reconnaissance d'un motif dans un texte est une opération des plus courantes en informatique. En anglais, ce problème est connu sous le nom de *string-matching* ou *pattern-matching on strings*. Mais quels algorithmes se cachent sous les commandes `grep` et `find` d'UNIX, `C-s` d'EMACS, *Rechercher* ... de votre traitement de texte préféré ?

Les deux algorithmes les plus célèbres et les plus fréquemment implantés portent le nom de leurs inventeurs : Boyer-Moore (1974) et Knuth-Morris-Pratt (1977). Ils sont bien plus efficaces que l'algorithme naïf et sont l'objet de nombreuses améliorations.

Mise en route

Aujourd'hui, on se propose d'implanter en JAVA un algorithme de recherche naïve puis une version simple de l'algorithme de Boyer-Moore. Il faut garder à l'esprit que rechercher la première occurence d'un motif dans un texte revient à faire *glisser* de gauche à droite le motif sur le texte, jusqu'à trouver une éventuelle coïncidence.

Nos deux programmes auront la même trame JAVA notamment au niveau des entrées/sorties. Elle est disponible sur `http ://deptinfo.unice.fr/~julia/AL/02tp.java`.

Il ne reste plus qu'à écrire les fonctions de recherche et aussi peut-être les précalculs nécessaires.

Exercice 1 : recherche naïve

1. Renommez le fichier téléchargé et comprenez-en bien le contenu.
2. Concevez un algorithme naïf de recherche d'un motif dans un texte.
3. Implantez-le dans la fonction `recherche()`.
4. Testez votre programme à loisirs.
5. Après l'avoir identifié, donnez la complexité de votre algorithme dans le pire des cas. En pratique, si m est la longueur du motif et n celle du texte, la complexité moyenne est seulement en $\mathcal{O}(m+n)$.

Algorithme de Boyer-Moore

L'algorithme de Boyer-Moore est caractérisé par le fait que l'on recherche d'abord la dernière lettre du motif. Si elle coïncide avec celle du texte, on recherche la lettre précédente du motif et ainsi de suite. Quand cela ne coïncide plus, on cherche à faire glisser le motif le plus loin possible à droite sur le texte, sans bien sûr rater une seule occurence. Pour cela, l'algorithme utilise deux stratégies, calcule pour chacune le prochain décalage et fait glisser le motif selon le plus grand des deux. L'intérêt est, dans tous les cas, de faire avancer le motif très efficacement.

La première stratégie est celle du *mauvais caractère*. Si on lit dans le texte un caractère qui n'est pas dans le motif, on peut faire glisser le motif au delà de ce caractère. Si au contraire il est contenu dans le motif, on considère la dernière occurence du *mauvais caractère* dans le motif et on décale le motif d'autant vers la droite.

La deuxième stratégie est dite du *bon suffixe*, nous ne l'implanterons pas aujourd'hui. Si l'idée est simple, sa mise en œuvre l'est moins. Cette stratégie utilise l'information relative au suffixe qui a été lu (à rebours) à la fois dans le motif et dans le texte. Si ce suffixe apparaît plusieurs fois dans le motif, on peut

décaler le motif à droite de sorte à placer la prochaine occurence du suffixe en face du texte où on vient de le lire.

Exercice 2

La stratégie du *mauvais caractère* nécessite le précalcul d'une table qui ne dépend que du motif. Par exemple, avec le motif `abracadabra`, la table indiquerait la valeur -1 pour chaque lettre de l'alphabet sauf pour les lettres `a`, `b`, `c`, `d` et `r` pour lesquelles elle indiquerait :

a	10
b	8
c	4
d	6
r	9

1. Ecrivez une méthode dont l'entête est le suivant :
   ```
   static int [] construireTable (String motif)
   ```
   Son rôle est de calculer et de retourner la table voulue. Ainsi, pour chaque lettre de l'alphabet, la table stocke la position de sa dernière occurence dans le motif ou `-1` si la lettre ne figure pas. *Conseil : si jamais les* `char` *vous causent des soucis, appelez à l'aide !*
2. Avant de passer à la suite, vérifiez l'exactitude de la table calculée.

Exercice 3

Pour achever l'écriture de la version simplifiée de l'algorithme Boyer-Moore, encore faut-il écrire la méthode `recherche()`.

1. Voici le scénario à programmer : on cale le motif à gauche du texte et on considère un pointeur `i` sur la dernière lettre du motif et un pointeur `j` à la même hauteur sur le texte.

   Si la dernière lettre du motif est aussi lue dans le texte, on passe aux lettres précédentes et ainsi de suite. Si on arrive à la première lettre du motif, c'est qu'on vient de trouver la première occurence du motif recherché !

   Sinon, dès que ça ne coïncide plus, le pointeur `i` sur le motif et le pointeur `j` sur le texte sont réactualisés comme suit, dans l'ordre :
   ```
   j = j + lgMotif - Math.min(i, 1+table[code mauvais caractère]);
   i = lgMotif -1;
   ```
   C'est vite programmé, mais c'est subtil à comprendre : après le déplacement du motif, l'indice `j` se repositionne dans le texte à hauteur de la dernière lettre du motif. L'indice `i` se place aussi sur la dernière lettre du motif. Soit $t =$ `table[`*`code mauvais caractère`*`]`, le motif a pu effectuer un de ces trois déplacements :
   - $t = -1$: on fait glisser le motif après le *mauvais caractère* du texte car il est absent du motif
   - $0 \leq t < i$: on fait glisser le motif pour faire coïncider le *mauvais caractère* du texte et sa dernière occurence dans le motif
   - $t > i$: on ne fait rien de mieux que d'avancer le motif d'une seule case.
2. A vue de nez, estimez la complexité d'un tel algorithme.