Université Côte d'Azur
Automates & Langages

L3 Info, L3 MI, L3 ST
2023–24

TP n°3

Reconnaissance de motifs (suite)

Mise en route

Aujourd'hui, on se propose d'implanter en PYTHON deux algorithmes classiques. Le premier est l'algorithme de BOYER-MOORE (1974) que nous implémenterons sous sa version simplifiée aussi connue sous le nom de BOYER-MOORE-HORSPOOL. Comme KMP, il repose sur l'idée que, pour rechercher une occurrence d'un motif, il faut faire *glisser* le plus efficacement possible le motif sur le texte jusqu'à obtention d'une éventuelle coïncidence.

Le second est celui de RABIN-KARP (1987) qui utilise une idée radicalement différente : le calcul d'une empreinte du motif par une fonction de hachage. L'algorithme consiste à trouver un facteur du texte qui a la même empreinte, à lancer une comparaison classique entre motif et facteur retenu puis à recommencer en cas d'échec.

Récupérez sous MOODLE le fichier du jour dans lequel effectuer le TP.

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 occurrence.

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 présent dans le motif, on considère la dernière occurrence 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 occurrence du suffixe en face du texte où on vient de le lire.

Exercice 1 : dictionnaire des dernières occurrences

La stratégie du *mauvais caractère* nécessite le précalcul d'une table qui ne dépend que du motif. Par exemple, voici ce qu'indiquerait la table correspondant au motif `abracadabra` :

a	10
b	8
c	4
d	6
r	9

1. Travaillez dans le fichier `03tp.py` du jour.
2. Ecrivez une fonction `dicoDerOcc()` prenant un motif en entrée et retournant le dictionnaire des dernières occurrences. Ainsi, pour chaque lettre du motif, le dictionnaire stocke la position de sa dernière occurrence dans le motif.
3. Avant de passer à la suite, vérifiez l'exactitude du dictionnaire calculé.

Exercice 2 : fonction de recherche

Pour achever l'écriture de la version simplifiée de l'algorithme Boyer-Moore, il faut écrire la fonction `rechercheBM()`. 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 occurrence 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 ... il faut trouver comment.

Après le déplacement du motif, l'indice `j` doit être repositionné dans le texte à hauteur de la dernière lettre du motif, l'indice `i` se place aussi sur la dernière lettre du motif.

Soit $d =$ `dicoDerOcc[`*`code mauvais caractère`*`]`, d prendra la valeur -1 s'il n'est pas défini dans la table. Le motif a pu effectuer un de ces trois déplacements :

- $d = -1$: on fait glisser le motif après le *mauvais caractère* du texte car il est absent du motif ;
- $0 \leq d < i$: on fait glisser le motif pour faire coïncider le *mauvais caractère* du texte et sa dernière occurrence dans le motif ;
- $d > i$: on ne fait rien de mieux que d'avancer le motif d'une seule position sur le texte.

Algorithme de Rabin-Karp

De façon radicalement différente, l'algorithme RABIN-KARP se fonde sur les *fonctions de hachage*. L'idée est de calculer une *empreinte* par une fonction de hachage donnée du motif à rechercher. Ensuite, on considère séquentiellement de gauche à droite dans le texte les facteurs dont la longueur est identique à celle du motif et on calcule à chaque fois leur empreinte par la fonction de hachage.

Dès qu'un facteur a la même empreinte que le motif, on procède à la comparaison lettre-à-lettre entre facteur et motif. Le nombre de comparaisons en est fortement diminué.

Exercice 3 : fonction de recherche

1. Comprenez ce que fait réellement la fonction `str2int()` fournie. Son calcul est à la base de celui de la fonction de hachage. *Facultatif :* Réécrivez cette fonction sans aucun calcul !
2. Ecrivez la fonction `rechercheRK()` qui, selon le scénario présenté en introduction, recherche la première occurrence du motif dans le texte, retourne sa position ou -1 sinon.

L'appel à la fonction de hachage reste cependant trop coûteux. Mais son calcul est susceptible d'être *amorti* du fait qu'on l'applique successivement à des facteurs très proches qui ne diffèrent que d'une lettre. En effet, en supprimant la première lettre d'un facteur et en ajoutant la lettre suivante du texte, on obtient le facteur suivant.

Exercice 4 : amortissement du calcul de l'empreinte

Si vous avez compris en profondeur le calcul de la fonction de hachage, vous pouvez imaginer une mise à jour en temps constant $\mathcal{O}(1)$ de cette fonction qui tire parti de la grande similitude entre deux facteurs successifs.