

L'ALGORITHME DE BOYER ET MOORE

- ▷ Histoire de l'informatique
- ▷ Structures de données
- ▷ Bases de données
- ▷ Architectures matérielles et systèmes d'exploitation
- ▷ Langages et programmation
- ▷ Algorithmique

1. Introduction

Rappelons tout d'abord qu'une chaîne de caractères est un objet de type `str` composé de caractères. Chaque caractère d'une chaîne est repéré par son index dans la chaîne. Les index commencent à 0.

Prenons l'exemple de la chaîne `s = 'abracadabra'`. Alors, `s[0]` a pour valeur le caractère `'a'`, `s[3]` a pour valeur le même caractère et `s[4]` le caractère `'c'`.

La fonction `len` renvoie le nombre de caractères d'une chaîne. Ici, `len(s)` a pour valeur 11.

Considérons deux chaînes de caractères, l'une appelée `texte`, l'autre appelée `motif`, on cherche s'il existe une occurrence du motif dans le texte, c'est-à-dire un index `i` tel que `texte[i:i + len(motif)] == motif`.

Plusieurs algorithmes ont été inventés pour résoudre ce problème, un des plus connus est l'algorithme de Knuth, Morris, Pratt, qui ne figure pas au programme. Un autre algorithme, très efficace, est l'algorithme de Boyer et Moore, qui a été inventé en 1977. Boyer et Moore travaillaient alors à l'université d'Austin au Texas en tant qu'informaticiens. Boyer qui était aussi un mathématicien est maintenant à la retraite.

Dans la suite on considère donc deux chaînes de caractères `texte` et `motif`, de longueurs respectives `n` et `p`.

Bien entendu, si `p > n` la recherche de `motif` dans `texte` échoue. En pratique on a $1 \leq p \leq n$, et même `p` beaucoup plus petit que `n`. Typiquement, on peut chercher un mot ou une phrase dans tout le texte d'un roman.

Le site <http://www.gutenberg.org/browse/languages/fr> propose les grands classiques de la littérature qui sont tombés dans le domaine public. On peut par exemple y trouver le texte intégral du roman *Le rouge et le noir* de Stendhal dans l'encodage UTF-8 : <http://www.gutenberg.org/ebooks/798.txt.utf-8>

On pourra alors charger en mémoire ce roman par ces quelques lignes pour chercher ensuite si le motif `'Julien trembla'` apparaît quelque part dans le roman.

```
fichier = open('LeRougeEtLeNoir.txt', 'r', encoding="utf-8")
stendhal = fichier.read()
fichier.close()
```

Ici la taille du texte est `len(stendhal)` qui vaut `n = 1020806`; la taille du motif est `len('Julien trembla')` qui vaut `p = 14`.

Une fonction intégrée de Python répond à notre problème de recherche de la première occurrence du motif `'Julien trembla'` dans ce texte : `stendhal.find('Julien trembla')` renvoie l'entier 161411 qui est l'index de la première position de ce

motif dans le roman.

D'ailleurs l'évaluation de `stendhal[161411:161445] + '...'` renvoie l'extrait suivant :

`'Julien tremblait que sa demande ne...'`

Notons que la convention choisie par la méthode `find` est de renvoyer la valeur `-1` dans le cas où le motif n'apparaît pas du tout dans le texte. Par exemple `stendhal.find('Joséphine')` renvoie `-1` : le prénom *Joséphine* n'apparaît jamais dans le roman.

Une variante de la méthode `find` a deux arguments, le deuxième précisant la position de départ de la recherche.

Avec cette convention, il est facile de calculer le nombre d'occurrences d'un motif dans un texte :

```
def nbOccurrences(texte, motif):
    compteur, i = 0, 0
    while True:
        occurrence = texte.find(motif, i)
        # occurrence est l'index du sous-texte à partir de l'index i
        if occurrence == -1:
            return compteur
        else:
            compteur += 1
            i = occurrence + 1
```

Avec cette fonction, on peut vérifier que le prénom `'Julien'` apparaît 1908 fois dans le roman, le mot `'amour'` 225 fois et le mot `'mort'` 178 fois.

2. Notations pour la suite

Dans toute la suite on cherche donc la première occurrence d'un motif de longueur p dans un texte de longueur n .

À un moment donné de la recherche, on observe une *fenêtre* de taille p du texte complet, sur laquelle on aligne le motif, et on regarde s'il y a bien correspondance.

S'il n'y a pas correspondance, on recommencera la recherche avec une fenêtre décalée vers la droite dans le texte.

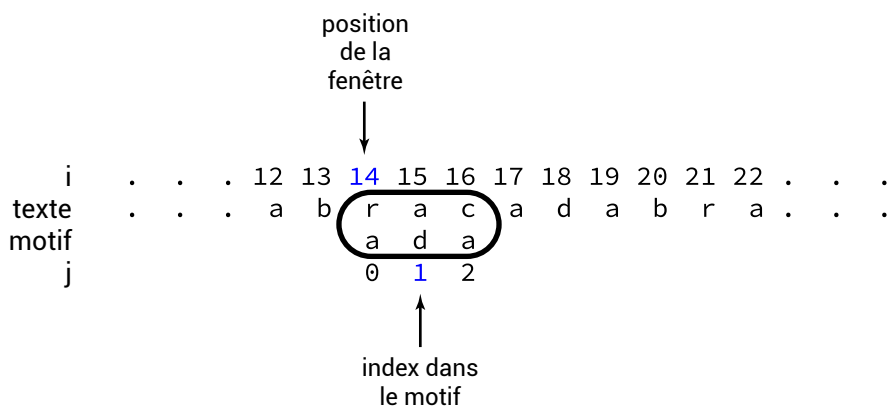
Dans tous les algorithmes présentés ici, la fenêtre se déplacera toujours de gauche à droite.

Nous noterons i la position de la fenêtre dans le texte : c'est l'index du premier caractère du texte qui apparaît dans la fenêtre.

Nous noterons j l'index dans le motif du caractère du motif que nous comparons avec son analogue du texte : on compare `motif[j]` avec `texte[i + j]`.

La recherche peut se faire à condition que $i + p \leq n$ puisque les caractères du texte qui apparaissent dans la fenêtre ont pour index $i, i + 1, \dots, i + p - 1$.

Dans la figure ci-dessous, on représente une fenêtre à la position $i = 14$ dans le texte, pour un motif de longueur $p = 3$. On compare le caractère `'d'` qui figure à l'index $j = 1$ du motif avec le caractère `'a'` qui figure à l'index $i + j = 15$ dans le texte.



Quand la fenêtre présente un défaut de correspondance entre les caractères du texte et ceux du motif, on déplace la fenêtre. Si le motif correspond parfaitement au texte dévoilé dans la fenêtre, on a trouvé une occurrence à la position i .

On écrira donc une fonction

```
def correspondance(texte, motif, p, i):
    ...
    ...
    return (ok, decalage)
```

qui renvoie, pour une fenêtre en position i , un couple formé d'un booléen `ok`, égal à `True` si on a trouvé une occurrence du motif et à `False` sinon, et un entier `decalage` qui indique de combien on souhaite augmenter la position i de la fenêtre pour la prochaine recherche.

A priori, naïvement, `decalage` vaudra 1, mais on verra qu'on peut améliorer le processus.

Dans le cas où le booléen `ok` vaut `True`, c'est-à-dire si on trouve une occurrence du motif, la valeur de `decalage` n'aura aucune importance.

Il est alors facile d'écrire une fonction de recherche :

```
def cherche(texte, motif):
    n = len(texte)
    p = len(motif)
    i = 0
    while i + p <= n:
        ok, decalage = correspondance(texte, motif, p, i)
        if ok: # on a trouvé une occurrence du motif en position i dans le texte !
            return i
        else:
            i = i + decalage
    return -1
```

3. Recherche naïve

3.1. L'algorithme naïf

L'algorithme naïf consiste simplement à comparer un à un, de gauche à droite, les caractères du texte apparaissant dans la fenêtre avec ceux du motif. En cas de non-correspondance on avance simplement la fenêtre d'une unité vers la droite.

Par exemple, dans la situation suivante,

i	.	.	.	12	13	14	15	16	17	18	19	20	21	22	.	.	.
texte	.	.	.	a	b	r	a	c	a	d	a	b	r	a	.	.	.
motif						a	d	a									
j						0	1	2									

on compare le 'a' du motif avec le 'r' du texte, obtenant immédiatement une différence : on peut avancer la fenêtre en incrémentant i , qui passe de 14 à 15.

Dans la nouvelle fenêtre, le premier caractère coïncide bien :

i	.	.	.	12	13	14	15	16	17	18	19	20	21	22	.	.	.
texte	.	.	.	a	b	r	a	c	a	d	a	b	r	a	.	.	.
motif							a	d	a								
j							0	1	2								

et on incrémente j pour tester les caractères suivants, 'd' et 'c' :

i	.	.	.	12	13	14	15	16	17	18	19	20	21	22	.	.	.
texte	.	.	.	a	b	r	a	c	a	d	a	b	r	a	.	.	.
motif							a	d	a								
j							0	1	2								

On est à nouveau en situation d'échec, et on effectue donc $i = i + 1$ et $j = 0$.
On en déduit l'écriture de la fonction correspondance.

```
def correspondance(texte, motif, p, i):
    # algorithme naïf - l'inégalité  $i + p \leq n$  est garantie
    for j in range(p):
        if texte[i + j] != motif[j]:
            return (False, 1)
    # si on arrive ici c'est qu'il y a eu correspondance
    return (True, 0)
```

3.2. Coût d'exécution

Comme le programme suggère de s'y limiter, on n'étudie que la complexité dans le cas le pire. Mais quel est-il, au fait ?
Le pire cas est quand on est obligé de faire passer la fenêtre par tous les indices i de l'intervalle $\llbracket 0, n - p \rrbracket$ et si en plus, pour chaque position i de la fenêtre, on doit comparer tous les caractères du motif au texte, c'est-à-dire si j varie dans tout l'intervalle $\llbracket 0, p \rrbracket$.
On vérifie que c'est le cas pour un texte ne contenant que des a et un motif ne contenant que des a sauf sa dernière lettre : on cherche aa...ab dans aa...aa.
Mais alors il y a $n - p + 1$ appels à correspondance, chacun de ces appels nécessitant p comparaisons de caractères : la complexité dans le cas le pire est donc, si on la mesure par le nombre de comparaisons, égale à $p(n - p)$.

4. Algorithme de Boyer-Moore, version simplifiée de Horspool

Nigel Horspool est né en Grande-Bretagne mais citoyen canadien. Il est professeur émérite d'informatique de l'université de Victoria, retraité depuis 2016. Il a conçu l'algorithme que nous décrivons maintenant.

4.1. Deux idées

La première idée consiste à comparer le motif avec la portion du texte qui apparaît dans la fenêtre de droite à gauche, et non pas de gauche à droite. Ainsi, on fait décroître j à partir de $p - 1$ jusqu'à trouver que le caractère qui lui fait face dans le texte, c'est-à-dire $x = \text{texte}[i + j]$, est différent du caractère $y = \text{motif}[j]$ du motif.
La deuxième idée consiste à opérer un décalage de la fenêtre qui varie en fonction de la paire de caractères qui ont révélé la non-correspondance, c'est-à-dire en fonction de (x, y) .

4.2. Le déroulement de l'algorithme

Nous considérons ici la recherche du motif 'dab' dans le texte 'abracadabra'.
Avec nos notations, $p = 3$, $n = 11$ et la première occurrence du motif dans le texte apparaît en position $i = 6$.
On commence avec la fenêtre tout à gauche, c'est-à-dire avec $i = 0$.

```
abracadabra
  dab
```

Comme on commence à comparer de droite à gauche, c'est pour $j = 2$ qu'il y a non-correspondance :
`motif[2] == 'b' != 'r' == texte[0 + 2]`.

On note $x = 'r'$ le caractère du texte qui ne correspond pas à $y = 'b'$ le caractère du motif qui lui fait face.
De combien peut-on décaler la fenêtre ? Comme x n'apparaît nulle part dans le motif, on peut carrément décaler le motif de $p = 3$ unités vers la droite !

```
abracadabra
      dab
```

Ainsi on se retrouve avec $i = 3$ et le premier échec intervient avec $j = 2$, où le caractère $x = \text{texte}[3 + 2] == 'a'$ du texte est distinct du caractère face à lui dans le motif, c'est-à-dire $y = \text{motif}[2] == 'b'$.
Mais à la différence du cas précédent, le caractère x apparaît bien dans le motif. On déplace donc la fenêtre d'une unité vers la droite.

Voici la prochaine étape :

```
abracadabra
  dab
```

$i = 4$, $x = 'd'$, $y = 'b'$, décalage de 2

Finalement avec $i = 6$, on trouve la première occurrence du motif.

```
abracadabra
  dab
```

4.3. Le calcul du décalage

Dans le cas où x n'apparaît pas du tout dans le motif, il convient de déplacer la fenêtre pour qu'elle débute juste à droite du couple (x, y) qui a provoqué l'échec de la recherche. Autrement dit, dans ce cas, le décalage est $\delta = j + 1$.

Dans le cas où x apparaît dans le motif, il convient de déplacer la fenêtre pour que x apparaisse juste au-dessus de la lettre du motif qui lui est égale. Si on note r la position de x la plus à droite dans le motif, il s'agit donc d'utiliser un décalage de $\delta = j - r$ si cette quantité est strictement positive. À défaut, (c'est-à-dire si $\delta \leq 0$) on se contentera d'un décalage d'une unité, comme dans l'algorithme naïf.

4.4. On passe à la programmation effective

Il faut donc commencer par calculer un dictionnaire dont les clés sont les caractères du motif et les valeurs la position la plus à droite du caractère.

C'est ce que réalise la fonction `calculeADroite`.

Dans le cas du mot `maman`, par exemple, on exécute tour à tour des affectations

```
aDroite['m'] = 0
aDroite['a'] = 1
aDroite['m'] = 2
aDroite['a'] = 3
aDroite['n'] = 4
```

de sorte qu'à la fin de l'exécution, `aDroite['m']` est bien égal à 2, position la plus à droite de la lettre 'm' dans le mot 'maman'. Cela dit, on voudrait calculer le décalage de la fenêtre même quand le caractère qui provoque l'échec n'apparaît pas dans le motif. Mais `aDroite['Z']` par exemple n'existe pas et demander sa valeur déclenche une erreur d'exécution. C'est pourquoi on a écrit la fonction `droite` qui renvoie `-1` si le caractère n'est pas dans le dictionnaire `aDroite`.

```
def calculeADroite(motif, p):
    # remplit (partiellement) un dictionnaire pour donner les positions les plus à
    # droite de chaque caractère
    global aDroite
    aDroite = {}
    for j in range(p):
        aDroite[motif[j]] = j

def droite(c):
    global aDroite
    # renvoie -1 si c n'est pas dans le motif ou sinon aDroite[c]
    if c in aDroite.keys():
        return aDroite[c]
    else:
        return -1
```

On a utilisé une variable globale pour le dictionnaire `aDroite`. Ce n'est pas toujours une bonne pratique, mais elle semble ici raisonnable.

Les deux seules différences dans l'écriture de la fonction `correspondance` résident dans le parcours du motif de droite à gauche (ligne 2) et dans le calcul du décalage, qui n'est pas égal à 1 mais se déduit du dictionnaire `aDroite` (ligne 5). L'utilisation de la fonction `max` n'est pas indispensable, on aurait pu remplacer la ligne 5 par

```
1     decalage = j - droite(x)
2     if decalage < 1:
3         decalage = 1
```

C'est vraiment une affaire de goût.

```
1 def correspondance(texte, motif, p, i):
2     for j in range(p - 1, -1, -1): # j varie de p-1 à 0 inclus en décroissant
3         x = texte[i + j]
4         if x != motif[j]:
5             decalage = max(1, j - droite(x))
6             return (False, decalage)
7     return (True, 0)
```

On modifie très légèrement l'écriture de la fonction `cherche` en ajoutant, en ligne 4, l'instruction de calcul du dictionnaire utile aux décalages.

```
1 def cherche(texte, motif):
2     n = len(texte)
3     p = len(motif)
4     calculeADroite(motif, p)
5     i = 0
6     while i + p <= n:
7         ok, decalage = correspondance(texte, motif, p, i)
8         if ok:
9             return i
10        else:
11            i = i + decalage
12    return -1
```

4.5. Remarques finales

Complexité

La complexité d'un algorithme de recherche textuelle se mesure essentiellement par le nombre d'opérations de comparaison de caractères qu'il effectue. L'analyse précise de la complexité de l'algorithme que nous avons programmé est difficile, et dépasse le niveau attendu en enseignement NSI.

Simplement, signalons qu'il est considéré comme un algorithme sous-linéaire : dans des cas favorables, les décalages de la fenêtre sont de l'ordre de la taille p du motif. En fait, l'algorithme n'est même pas tenu de lire l'intégralité du texte : c'est la sous-linéarité. Dans les cas favorables, on peut estimer un coût de l'ordre de n/p , ce qui est évidemment très intéressant.

Prétraitement

Le calcul du dictionnaire `aDroite` est effectué une seule fois pour un motif donné.

Bien sûr, un tel pré-traitement a un coût, mais celui-ci ne doit être compté qu'une fois même si on effectue plusieurs recherches avec le même motif. Par exemple, la recherche des 1908 occurrences de '`Julien`' dans le roman *Le rouge et le noir* peut se contenter d'un seul pré-traitement : il suffit de déplacer la ligne 4 de la fonction `cherche` et de l'insérer au début de la fonction `nbOccurrences`.

Cette stratégie d'un pré-traitement des données est un paradigme de programmation qu'on peut garder en tête pour d'autres applications.

L'algorithme complet de Boyer-Moore

Nous n'avons présenté qu'une version simplifiée de l'algorithme complet.

L'algorithme complet de Boyer-Moore utilise une deuxième table de décalage, beaucoup plus difficile à calculer, qui permet de tenir compte des caractéristiques du motif dans le cas où celui-ci présente des similarités internes, ce qui permet d'effectuer des décalages plus importants, donc d'augmenter l'efficacité de la recherche.

L'algorithme complet de Boyer-Moore présente des difficultés en termes de justification et de programmation effective qui dépassent le niveau attendu en NSI. C'est pourquoi nous ne l'évoquons pas ici.

Le lecteur curieux pourra lire les pages 360–366 de l'ouvrage de Berstel, Beauquier et Chrétienne, disponible en ligne : <http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf>