tnsi_15_Boyer_Moore

May 23, 2024

1 Recherche de motif dans un texte

Ce TP étudie un algorithme particulièrement efficace pour rechercher une sous-chaîne (le motif) dans une chaîne (le texte). On dispose donc d'une grande chaîne de caractères, de longueur N, et d'un mot plus court de longueur n, noté $x_0x_1x_2\dots x_{n-1}$. On cherche l'indice (ou les indices) auquel apparaît le motif.
Exemple : le motif *abaa* apparaît dans le texte *acaabbabaaa* à la position 6.

1.1 Algorithme naïf

On compare le motif à une *fenêtre* glissante dans le texte, c'est-à-dire à un groupe de lettres de même longueur que le motif. Pour préparer à l'algorithme de Boyer-Moore, on effectue la comparaison de droite à gauche (et non de gauche à droite comme on le ferai habituellement). Implémenter l'algorithme suivant :

Données :
Un texte t de longueur N
Un motif x de longueur $n \geq N$
Renvoie :
Toutes les positions de x dans t
> $P \leftarrow$ liste vide # liste des positions
> **pour** i de 0 à $N - n + 1$:
» $j \leftarrow n - 1$
» **Tant que** $j \geq 0$ et $x_j = t_{i+j}$:
»> $j \leftarrow$ j - 1

si j = -1 : > Ajouter i à P

Renvoyer P

[1]:
```
def rechercheNaive(texte, motif):
    """
    Renvoie les positions d'un motif dans un texte
    @param texte, motif : chaine de caractères. La longueur N de texte est  à␣
↪celle de motif
    @return positions : liste des positions du motif dans le texte
    @return compteur : facultatif, compte les comparaisons
    """
    positions = []
    n = len(motif)
```

```
    N = len(texte)
    compteur = 0
    assert N >= n, "le texte doit être plus long que le motif"

    return positions, compteur

def testsNaifs(couples):
    for element in couples:
        print("Motif : ",element[0], "texte : ", element[1])
        print("résultats",rechercheNaive(element[1], element[0]), "\n")

couples = [("abaa", "acaabbabaaa"),
           ("aaaa", "aaaaaaaaaaa"),
           ("caaa","aaaaaaaaaaa"),
          ("pilotes", "les pecheurs rudoyes de cet ouest battu des vents font␣
 ↪des pilotes habiles")]
testsNaifs(couples)
```

```
Motif :  abaa texte :  acaabbabaaa
résultats ([], 0)

Motif :  aaaa texte :  aaaaaaaaaaa
résultats ([], 0)

Motif :  caaa texte :  aaaaaaaaaaa
résultats ([], 0)
```

On reprend l'exemple précédent (motif *abaa* , texte *acaabbabaaa*). Le tableau ci-dessous donne le déroulement de l'algorithme, le caractère en gras donnant la première lettre du motif où la différence est repérée (en partant de la droite).

i	a	c	a	a	b	b	a	b	a	a	a
0	a	**b**	a	a							
1		a	b	a	**a**						
2			a	b	a	**a**					
3				a	b	**a**	a				
4					a	b	a	**a**			
5						a	b	**a**	a		
6							a	b	a	a	
7								a	**b**	a	a

Compter le nombre de comparaisons dans les cas suivants: * motif *aaa* , texte *aaaaaaaaaaa* * motif *caaa* , texte *aaaaaaaa*

Donner la complexité de l'algorithme naïf en fonction de n et N.

1.2 Règle du mauvais caractère

Reprenons le tableau précédent. On peut constater que la deuxième lettre du texte est c, qui n'apparaît pas dans le motif. Il est donc inutile de tester le cas $i = 1$ puisque motif et texte différeront au moins sur cette lettre. Pour $i = 2$, la différence est constatée en comparant le dernier b de la fenêtre, dans le texte, avec le dernier a du motif. On peut donc chercher directement à aligner le b de la fenêtre avec un b du motif : on passe directement à $i = 4$. De la même manière, expliquer pourquoi on saute l'étape $i = 5$.

i	a	c	a	a	b	b	a	b	a	a	a
0	a	**b**	a	a							
2			a	b	a	**a**					
4					a	b	a	**a**			
6							a	b	a	a	
7								a	**b**	a	a

1.2.1 Programmation effective : l'algorithme de Boyer-Moore simplifié (Horspool)

Pour implémenter la règle du mauvais caractère, on va créer un tableau associatif (un dictionnaire), dont les clés sont les lettres du motif, et les valeurs la dernière position de la lettre dans le motif, sauf pour la dernière lettre. Ce dictionnaire permettra de calculer directement le décalage de la fenêtre dans le mot à explorer. Ecrire la fonction `lettreADroite(motif)` qui renvoie ce dictionnaire. *Exemple* : `lettreADroite('exercice')` renvoie `pos_droite = {'e':7, 'c':6, 'i':5, 'r':3, 'x':1}`.

[ ]:
```
def lettreADroite(motif):
    pos_droite = {}

    return pos_droite

def testsPosDroite(couples):
    for element in couples:
        print("Avec le motif ", element[0], "on obtient␣
 ↪",lettreADroite(element[0]))

testsPosDroite(couples)
```

On modifie l'algorithme précédent pour qu'il prenne en compte la règle du mauvais caractère. La boucle **pour** est remplacée par un **tant que**. Si le motif ne correspond pas à la fenêtre, alors le décalage est calculé en tenant compte du dictionnaire des positions. On tient aussi compte du fait qu'une lettre du texte peut ne pas être dans le motif, la clé correspondante est donc absente du dictionnaire. Compléter l'algorithme suivant, en donnant l'instruction qui permet de calculer i (remarque : la solution est en commentaire, visible en double cliquant dans cette cellule).

Données :
Un texte t de longueur N
Un motif x de longueur $n \geq N$
Le dictionnaire *pos_droite* donnant la dernière position de tous les caractères du motif (dernière

lettre exceptée)
Renvoie :
Toutes les positions de x dans t
> $P \leftarrow$ liste vide # liste des positions
> $i \leftarrow 0$
> **Tant que** $i \quad N - n$:
» $j \leftarrow n - 1$
» **Tant que** $j \geq 0$ et $x_j = t_{i+j}$:
»> $j \leftarrow$ j - 1

si j = -1 :

Ajouter i à P

$i \leftarrow$ ...

Renvoyer P

Implémenter cet algorithme. Pour tenir compte du fait que le dictionnaire des positions des lettres du motif ne contient pas forcément toutes les lettres du texte, on pourra créer une fonction supplémentaire `droite(c)` qui renvoie la valeur correspondante si la clé existe, et -1 sinon.

[ ]:
```python
def droite(c, pos_droite):
    if c in pos_droite.keys():
        return pos_droite[c]
    else:
        return -1

def horspool(texte, motif, pos_droite):
    """
    Renvoie les positions d'un motif dans un texte
    @param texte, motif : chaine de caractères. La longueur N de texte est  à␣
 ↪celle de mtof
    @return positions : liste des positions du motif dans le texte
    @return compteur : facultatif, compte les comparaisons
    """
    positions = []
    n = len(motif)
    N = len(texte)
    assert N >= n, "le texte doit être plus long que le motif"
    compteur = 0

    return positions, compteur

def tests_horspool(couples):
    for element in couples:
        pos_droite = lettreADroite(element[0])
        print("Motif : ",element[0], "texte : ", element[1], "dictionnaire :
 ↪",pos_droite)
        print("résultats", horspool(element[1], element[0], pos_droite), "\n")
```

```
couples = [("abaa", "acaabbabaaa"),
           ("aaaa", "aaaaaaaaaaa"),
           ("caaa","aaaaaaaaaaa"),
          ("pilotes", "les pecheurs rudoyes de cet ouest battu des vents font␣
 ↪des pilotes habiles")]

tests_horspool(couples)
```

On peut constater que cette amélioration ne change pas la complexité théorique. En pratique, il y a amélioration effective. Ceci est d'autant plus vrai avec la version suivante, qui est un approfondissement, assez complexe.

1.3 Algorithme de Boyer-Moore

Pour ceux qui sont en avance ou qui souhaitent approfondir

1.3.1 Règle du bon suffixe

Exemple : on cherche le motif $x = abaaaa$ dans le texte $t = abbcaacaaaabaaaa$ (il est à la fin, en position 10).

Pour `i = 0`, la fenêtre est abbcaa. Le "bon suffixe" est obtenu en partant de la droite. C'est la partie de la fenêtre commune avec une partie du motif. Ici c'est *aa* : $x = abaa$ **aa** et $t = abbc$ **aa** *caaaabaaaa*. On va décaler la fenêtre pour aligner ce suffixe avec une copie de aa entre texte et suffixe. On pourrait décaler de 1, en se basant sur x = *aba* **aa** *a* . Dans ce cas, on constate que cette copie est précédée d'un *a*. Or c'était déjà le cas pour le "bon suffixe" : on va donc reproduire la même erreur. Par contre, si on décale de 2, en se basant sur $x = ab$ **aa** *aa* , la lettre précédente est *b* , différente du *a* précédant le "bon suffixe". La décalage est donc pertinent, la comparaison de la lettre précédant le "bon suffixe" étant différente.

On se place donc en `i = 2`, la fenêtre est *bcaaca*. Le "bon suffixe est" *a*. Sa copie dans le motif qui n'est pas précédée d'un *a* est en position 2 : *ab* **a** *aaa*. On décale le motif pour aligner ce *a* avec le dernier *a* de la fenêtre (soit `i = 5`).

On est donc en `i = 5`, on compare le motif *abaaaa* avec la fenêtre *acaaaa*. La "bon suffixe" est *aaaa*. Il n'apparaît pas ailleurs dans le motif, on ne peut pas appliquer la méthode précédente (pour passer de `i = 0`à `i = 2`puis `i = 5`). On va utiliser une règle différente : on va chercher le préfixe du motif x le plus long possible, qui soit aussi suffixe du "bon suffixe" (ça suit au fond ?). Ici c'est *a* car :
* le bon suffixe finit par *a* , et x commence par *a* * x commence par *ab* mais le bon suffixe ne commence pas par *ab*
On aligne ce préfixe du motif x avec le suffixe correspondant du bon suffixe. Le décalage est donc de 5 lettres, on passe à `i = 10`.

i	a	b	b	c	a	a	c	a	a	a	a	b	a	a	a	a
0	a	b	a	**a**	a	a										
2			a	b	a	a	**a**	a								
5						a	**b**	a	a	a	a					

i	a	b	b	c	a	a	c	a	a	a	a	b	a	a	a	a
10											a	b	a	a	a	a

Remarque : calculer ce décalage en temps linéaire n'est pas trivial. On ne se posera pas ce problème dans le cadre de ce TP.

1.3.2 Algorithme final

Dans l'algorithme final, on applique le meilleur des deux décalages obtenus avec la règle du mauvais caractère, et la règle du bon suffixe.

Exercice : appliquer l'algorithme final à: 1. motif = *abaaaa* , texte = *abbcaacaaaabaaaa* 2. texte = *GCATCGCAGAGAGTATACAGTACG* , motif *GCAGAGAG* 3. texte = *aabbbababacaabbaba* , motif *aababab* 4. motif = *abacab* , texte = *abacaabadcabacabaabb* 5. texte = *CBADBCACBADCB-BACACBCAABCA* , motif = *CBCAABCA*

1.3.3 Programmation de l'algorithme

On modifie l'algorithme initial.

Données :
Un texte t de longueur N.
Un motif x de longueur $n \geq N$.
Le dictionnaire *pos_droite* donnant la dernière position de tous les caractères du motif.
Deux tableaux s et p tels que : * s est le tableau des bon suffixes. $s[j] = s_x(j)$, où $s_x(j)$ est la position la plus à droite d'une copie du suffixe $x_{[j,m[}$, formé des dernières lettres de x à partir de la j-ième. Cette copie ne doit pas être précédée de la lettre d'indice *j - 1* de x. Si une telle copie n'existe pas, alors $s[j] = -1$. * p est le tableau des préfixes. $p[j] = p_x(j)$, où $p_x(j)$ est la longueur du plus long préfixe de x qui soit aussi suffixe de $x_{[j,m[}$. On impose que le préfixe soit différent de x entier, on pose donc $p[0] = p[1]$.

Renvoie :
Toutes les positions de x dans t
>$P \leftarrow$ liste vide # liste des positions
>$i \leftarrow 0$
>**Tant que** $i \quad N - n$:
»$j \leftarrow n - 1$
»**Tant que** $j \geq 0$ et $x_j = t_{i+j}$:
»>$j \leftarrow$ j - 1

si j = -1 :

Ajouter i à P
i $\leftarrow$ i + n - p[1]

Sinon :

SI $s[\ j + 1] \quad 0$: $i \leftarrow i + \max(j$ - *pos_droite* $[t_{i+j}]$, j + 1 - $s[\ j + 1])$
Sinon : $i \leftarrow i + \max($n - $p[\ j\ ]$, j - *pos_droite* $[t_{i+j}])$

Renvoyer P

Ecrire les fonctions donnant les tableaux s et p, puis implémenter l'algorithme.
On peut le tester avec les fichiers chr18 (chromosome 18). Tests rapides : chr18_3 ou chr18_4 dans chr_2, assez rapide : chr18_2 dans chr18_0. Plus long : chr18_3 dans chr_0. Pour ouvrir les fichiers et les sauver , vous pouvez reprendre la fonction ci-dessous. On peut rajouter un compteur dans le programme pour le nombre de comparaisons, et constater que l'on gagne beaucoup en efficacité par rapport à la méthode naïve.

[ ]:
```
def BoyerMoore(texte, motif):
    return

def test():
    C = []
    for i in range(5):
        with open(f'chr18/chr18_{i}') as c:
            C.append(c.read().replace('\n',''))

    for i in range(len(C)):
        print("longueur de chr18_",i,":",len(C[i]))

    print("Test de l'algorithme pour chercher chr18_3 dans chr18_0 :")
    print(BoyerMoore(C[0], C[3]))
```

<hr style="color:black; height:1px /> <div style="float:left;margin:0 10px 10px 0" markdown="1" style = "font-size ="x-small">

Un grand merci à Bruno Grenet, Université de Montpellier II, source principale de ce TD.
Voir aussi ici pour un des exercices avec le détail des décalages.

frederic.mandon@ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France