Rapport projet

Bio-Algorithmique

Aho-Corasick

BOUZIDI Abdeldjalil

Chapitre 1 : Introduction :

La recherche de motifs s’intéresse à la recherche d’un ou plusieurs mots appelés motif dans un texte. Le motif et le texte sont écrits sur un alphabet, cet alphabet peut être l’ensemble de caractères ASCII ou l’ensemble {A, C, G, T} dans le cadre de la bio-informatique.

La problématique qui se pose est de trouver toutes les occurrences des tous les motifs dans un texte donné

Chapitre 2 : Algorithmes :

1. L’algorithme naïf :

1.1. Principe :

La méthode qui vient immédiatement à l’esprit pour l’appariement des motifs consiste simplement à vérifier, pour chaque position du texte où le motif pourrait s’apparier, s’il y a véritablement concordance ou non. L’algorithme suivant effectue de cette façon une recherche de la première occurrence d’un motif p dans un texte t.

1.2. Algorithme :



1.3. Complexité :

L’algorithme naïf est de complexité O((n-m+1)\*m) dans de pire cas. (Dans le cas de la recherche d’un ensemble de k motifs , il conduit a une complexité O((n-m+1)\*m\*k))

2. L’algorithme Knuth-Morris-Pratt :

1.1. Principe :

Pour gagner du temps, l'algorithme KMP élimine toutes les comparaisons de l'algorithme naïf qui peuvent être évitées grâce aux précédentes comparaisons. La méthode est subtile et s'appuie au préalable sur une observation attentive des comparaisons effectuées dans l'algorithme naïf.

1.2. Algorithme :



1.3. Complexité :

Cet algorithme s’effectue en temps O(m) en moyenne ce qui conduit à une complexité globale en O(m+n) (Dans le cas de la recherche d’un ensemble de k motifs, il conduit à une complexité O((m+n)\*k))

3. L’algorithme Rabin-Karp :

1.1. Principe :

Il est possible d´améliorer l´algorithme naîf en utilisant un grand volume de mémoire. Celui-ci va traiter chaque suite de m caractères du texte comme clé d´une table d´adressage dispersé standard, et effectuer des comparaisons d´entiers.

1.2. Algorithme :



1.3. Complexité :

La complexité associée à cet algorithme est en O(mn) dans le pire des cas et O(m+n) si tout va bien.

3. L’algorithme Aho-Corasick :

1.1. Principe :

Plutôt que de faire la recherche des occurrences des motifs de l’ensemble P={P0,P1,....,Pk-1} dans le texte T séquentiellement, on peut la faire en ‘‘parallèle’’. Ceci constitue l’algorithme de Aho-Corasick qui peut être vu comme une généralisation de l´algorithme de Knuth-Morris-Pratt pour faire une recherche parallèle.

1.2. Algorithme :



1.3. Complexité :

La complede l'algorithme est linéaire en longueur des motifs plus la longueur du texte recherché plus le nombre de correspondances en sortie

Chapitre 3 construction d’un alignement multiple:

**1. Structure de données utilisée :**

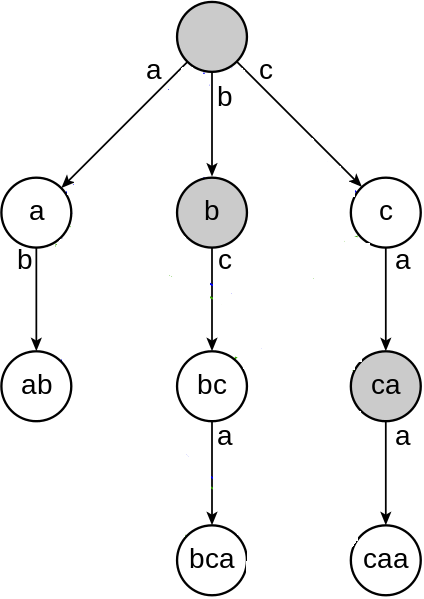
|  |  |  |
| --- | --- | --- |
| Dictionnaire  (Transition) | Sortie (Liste)  (Mot généré) | Pointeur  (Suffixe) |

**Fig.1**

**Code:**

|  |
| --- |
| class Node:  def \_\_init\_\_(self):  self.goto = {}  self.out = []  self.fail = None |

**2. L’arbre :**



**Fig.2**

**Code:**

|  |
| --- |
| def arbre(patterns):  root = Node()  for path in patterns:  node = root  for symbol in path:  node = node.goto.setdefault(symbol, Node())  node.out.append(path)    return root |

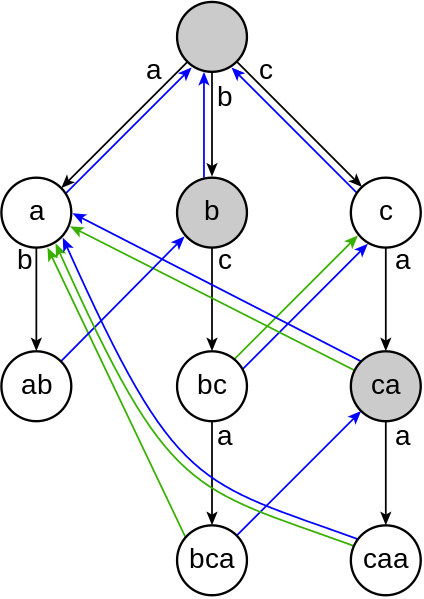
**3. L’automate :**

Le graphe ci-dessous **Fig.3** est la structure de données Aho – Corasick construite à partir du dictionnaire spécifié, chaque ligne du tableau représentant un nœud du tri, le chemin de la colonne indiquant la séquence (unique) de caractères allant de la racine au nœud.

La structure de données a un nœud pour chaque préfixe de chaque chaîne du dictionnaire. Donc, si (bca) est dans le dictionnaire, alors il y aura des nœuds pour (bca), (bc), (b) et (). Il existe un arc "enfant" dirigé en noir de chaque nœud à un nœud dont le nom est trouvé en ajoutant un caractère. Donc, il y a un arc noir de (bc) à (bca). Il existe un arc de "suffixe" bleu dirigé de chaque nœud au nœud qui en est le suffixe strict le plus long possible dans le graphique. Par exemple, pour noeud (caa), ses suffixes stricts sont (aa) et (a) et (). Le plus long de ceux-ci qui existe dans le graphique est (a). Donc, il y a un arc bleu de (caa) à (a). Il existe un arc "suffixe de dictionnaire" vert allant de chaque nœud au nœud suivant du dictionnaire, auquel vous pouvez accéder en suivant les arcs bleus. Par exemple,

À chaque étape, le nœud actuel est étendu en trouvant son enfant, et si cela n'existe pas, en recherchant l'enfant de son suffixe, et si cela ne fonctionne pas, en recherchant l'enfant de son suffixe, et ainsi de suite, en terminant par la racine. noeud si rien n'a été vu auparavant.

Lorsque l'algorithme atteint un nœud, il génère toutes les entrées du dictionnaire qui se terminent à la position actuelle du caractère dans le texte saisi. Ceci est fait en imprimant chaque noeud atteint en suivant les liens du suffixe du dictionnaire, en partant de ce noeud et en continuant jusqu'à atteindre un noeud sans lien de suffixe du dictionnaire. De plus, le nœud lui-même est imprimé, s'il s'agit d'une entrée de dictionnaire.



**Fig.3**

**Code:**

|  |
| --- |
| def automate(patterns):  root = arbre(patterns)  queue = []  for k, node in root.goto.items():  queue.append(node)  node.fail = root  while len(queue) > 0:  rnode = queue.pop(0)  for key, unode in rnode.goto.items():  queue.append(unode)  fnode = rnode.fail  while fnode != None and not key in fnode.goto.keys():  fnode = fnode.fail  unode.fail = fnode.goto[key] if fnode else root  unode.out += unode.fail.out    return root |

**3. La recherche en utilisant l’automate :**

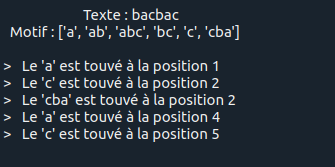
**Code :**

|  |
| --- |
| def toutes\_occurrences(s, root, callback):  node = root  for i in range(len(s)):  while node != None and not s[i] in node.goto.keys():  node = node.fail  if node == None:  node = root  continue  node = node.goto[s[i]]  for pattern in node.out:  callback(i - len(pattern) + 1, pattern)  def occurence(pos, patterns):  print ("Le %s est touvé à la position %s" % (patterns, pos)) |

**Exemple :**

Texte : “bacbac”

Motif : [‘a’, ‘ab’, ‘abc’, ‘bc’, ‘c’, ‘cba’]



**3. Comparaison en Aho-Corasick et les algorithmes de recherche :**

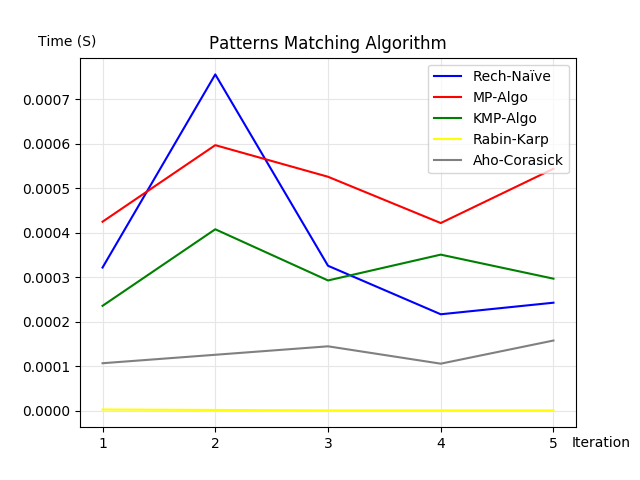
Pour obtenir une meilleure comparaison, il faut que les les quatres algorithme peuvent trouver toutes les occurrences de tous les motifs.

La fonction graphe fait appel aux algorithmes et calcule le temps d'exécution de chacun d’eux puis trace le graphe de temps d’exécution en fonction des paramètres entrés (texte et les motifs qui sont les mêmes à la ième itération)

**code :**

|  |
| --- |
| def graphe(s, patt):  global patt1, patt2, patt3,patt4, patt5  global stat, tt  lst1, lst2, lst3, lst4, lst5 = [], [], [], [], []  for i in range(len(patt)):  t1 = clock()  d1 = fct1(s,patt[i])  t1 = clock() - t1  t2 = clock()  d2 = fct2(s,patt[i])  t2 = clock() - t2  t3 = clock()  d3 = fct3(s,patt[i])  t3 = clock() - t3  t4 = clock()  #d4 = fct4(s,patt[i])  t4 = clock() - t4  t5 = clock()  ahocorasick(s,patt[i])  msg = ""  t5 = clock() - t5  lst1.append(t1)  lst2.append(t2)  lst3.append(t3)  lst4.append(t4)  lst5.append(tt)  stat = [lst1, lst2, lst3, lst4, lst5] |

**Résultat :**

****

**Conclusion :**

L’algorithme Aho-Corasick est meilleur que les autres algorithmes lorsque le texte est plus grand puisqu’il parcourt le texte une seule fois. Le pire cas est cas de petit texte dont la construction de l’automate prend beaucoup de temps.