Soutenance : Projet OUV STL

Détection de plagiat flagrant

Farouck Cherfi & Abdelkader Boumessaoud

Plan

1. Introduction aux différentes approches
2. Approche par programmation dynamique
3. Approche par l’arbre des suffixes
4. Approche par l’arbre des suffixes compressés
5. Comparaisons
6. Conclusion

Introduction aux différentes approches

But : Résoudre un problème algorithmique d’exploration de chaînes de caractères

Comment ?

En utilisant différentes structures de données plus ou moins évoluées.

Pourquoi ?

Pour comparer l'efficacité de ces différentes structures de données

Approche par la programmation dynamique

Structure de type Matrice[n+1][m+1]

$$M(i,j) = \begin{cases} 0 & \text{si i,j} = 0,0 \text{ ou C1[i] != C2[j]} \\ S[i-1, j-1] + 1 & \text{sinon.} \end{cases}$$

Indices de la PLCC du mot C1 et C2

$$C1(i,j) = \begin{cases} j = max(M(i)) - 1 \\ i = j - max + 1 \end{cases}$$

$$C2(i,j) = \begin{cases} j = max(M(j)) - 1 \\ i = j - max + 1 \end{cases}$$

Complexité pire cas

O(n*m)
n : longueur chaîne C1
m : longueur chaîne C2

Code de la PLCC

```
let plsuffixe c1 c2 =
  let maxlen = ref 0  in
  let indicec1 = Array.make 2 0 in
  let indicec2 = Array.make 2 0 in
  let l1 = String.length c1  in
  let l2 = String.length c2  in
  let tab = Array.make_matrix (l1+1) (l2+1) 0 in

  for i = 1 to l1 do
    for j = 1 to l2 do

      if c1.[i-1] == c2.[j-1] then tab.(i).(j) <- tab.(i-1).(j-1) + 1 ;
      if tab.(i).(j) > !maxlen then (maxlen := tab.(i).(j);
                                     indicec1.(1) <- i-1;
                                     indicec1.(0) <- indicec1.(1) - !maxlen + 1
                                     indicec2.(1) <- j-1;
                                     indicec2.(0) <- indicec2.(1) - !maxlen
+ 1;);
      print_int tab.(i).(j);
    done ;
    print_string "\n";
  done;

  !maxlen, indicec1, indicec2 ;;
```

Approche par l’arbre des suffixes

Structure de type Trie

```
type 'a suffixtree =
    Node of int * (string * 'a suffixtree) list
```

Trouver la PLCC du mot C1 et C2

Stockage du nombre de fois du même caractère rencontré pour chaque nœud.
On récupère ensuite le max de tous les fils de la racine et de même pour le fils choisi jusqu'à atteindre 0 qui nous donnera un chemin d’une sous chaîne commune.

Complexité pire cas

O(n^2) pour l’insertion d’un mot
O(m) pour la recherche d’un motif

Code de création d'arbre suffixe

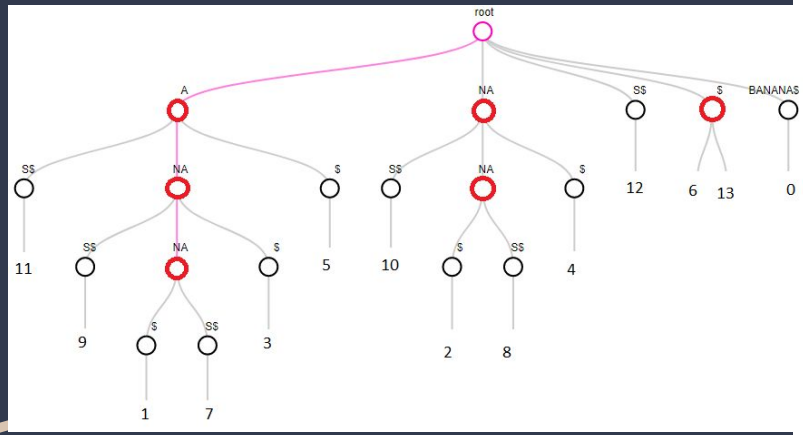
```
let rec arbresuffixe arbre mot =
  let mot_bis = (String.sub mot 1 ((String.length mot) -1)) in
  match mot.[0] with
  | '#' -> ajout arbre mot
  | x -> arbresuffixe (ajout arbre mot) mot_bis ;;
```

Code de sous-chaînes commune

```
let souschainecommunes chaine1 chaine2 =
let arbre = arbresuffixe vide chaine1 in
let arbre_2 = arbresuffixe arbre chaine2 in
match arbre_2 with
| Node(_,l) ->
    let rec parcours li max noeud=
      if max = -1 then (match noeud with
          | (x,_) -> ((x, Node(0,[]))::[])
      else
        match li with
        | (x,Node(y,ll))::q -> (if y > max
                               then parcours q y (x,Node(y,parcours ll (y-2)
                               else parcours q max noeud);
        | [] -> noeud::li

    in  let solution = (parcours l 0 ("",Node(0,[]))) in
    let rec parcours_solu solu rep=
      match solu with
      | (x,Node(_,ll))::q -> rep^parcours_solu ll x
      | [] -> rep
    in parcours_solu solution ""
```

Approche par l'arbre des suffixes compressé



Structure de type Radix

```
class suffixeTree = object
  val root = noeud
  val mutable fullText = ""
end;;
```

```
class noeud = object
  val mutable text = ""
  val mutable enfants = node []
  val mutable poisition = -1 ref
  method getEnfants = enfants
  method getPosition = position
  method getText = text
end;;
```

Trouver la PLCC du mot C1 et C2

Construction d'un arbre de suffixes généralisé pour les deux chaînes, puis on cherche les nœuds internes les plus profonds communs aux deux chaînes.

Complexité pire cas

Pour la création d'un arbre suffixe compressé = O(n*m)
n : taille du mot
m : taille de l'alphabet du mot

Code de compression

```
let compression arbre =
  match arbre with
  | Node(x,[]) -> Node(x,[])
  | Node(_,l) ->
      let rec parcours_arbre arbre =

        match arbre with
        | (x,Node(p,ll))::q ->
            if List.length ll != 1 then
              (x,Node(0, ((parcours_arbre ll))))::parcours_arbre q
            else
              ((x^(getval_noeud_enfant ll),Node(1,(parcours_arbre (getnoeud ll)

        | [] -> []

      in Node(0,parcours_arbre l);;
```

Approche mené pour les arbres suffixes compressé

- **1ere solution envisagée :** Algorithme d'Ukkonen
  - lecture du mot de gauche à droite
  - pointers pour la position dans le texte originale
  - complexité en O(n)

-> solution abandonné vu la complexité d'implémentation

- **Cas problématique pour la compression de l'arbre :** Parfois il y a des noeuds qui ne sont pas compressé, par exemple si un noeud a deux fils, la compression s'opère que sur un seul des deux s' ils possèdent un enfant

- **Structure choisis non optimal :** structure de base choisis pour implémenter l'arbre de suffixe mal adapté pour une compression

Comparaisons

```
val l : int array = [|10; 20; 50; 75; 100; 125; 150; 200; 350; 400; 500|]
# somme_time_plsuffixe String_donnes0 String_donnes1 l ;;
Temps d'excution en 0.000000 secondes
Temps d'excution en 0.000000 secondes
Temps d'excution en 0.000000 secondes
Temps d'excution en 0.000000 secondes
Temps d'excution en 0.000000 secondes
Temps d'excution en 0.001000 secondes
Temps d'excution en 0.002000 secondes
Temps d'excution en 0.003000 secondes
Temps d'excution en 0.010000 secondes
Temps d'excution en 0.008000 secondes
Temps d'excution en 0.015000 secondes
Temps d'excution en 0.033000 secondes
- : unit = ()
# somme_time_souschainecommunes (String_donnes0^"#") (String_donnes1^"#") l ;;
Temps d'excution en 0.001000 secondes
Temps d'excution en 0.003000 secondes
Temps d'excution en 0.020000 secondes
Temps d'excution en 0.013000 secondes
Temps d'excution en 0.019000 secondes
Temps d'excution en 0.033000 secondes
Temps d'excution en 0.043000 secondes
Temps d'excution en 0.053000 secondes
Temps d'excution en 0.166000 secondes
Temps d'excution en 0.284000 secondes
Temps d'excution en 0.356000 secondes
Temps d'excution en 0.987000 secondes
- : unit = ()
```

Remarques sur le temps d'exécution :

- exécution de *plsuffixe* (programmation dynamique) plus rapide que celle de *souschainecommunes* (Arbre des suffixes) sur la majorité des échantillons testés
- Cas spéciaux ou l'arbre des suffixes est plus rapide

Conclusions sur le temps d'exécution :

- Pour un temps constant d'alphabets, l'arbre suffixe peut être plus rapide en temps O(n * log(n))
- L'utilisation d'un arbre des suffixes est uniquement utile dans le cas ou il est compressée,
- Les cas ou la programmation dynamique est plus rapide que les arbres suffixes compressés dépendant surtout de la fréquence des lettres

Chaîne de caractères communes entre les textes

Exemples de séquences communes trouve entres les échantillon :

- Donnes0 et Donnes1 : "des premières universités"
- Donnes0 et Donnes2 : "es premiers grands poêtes - Chrétien de Troyes, — Marie de France, Rutebeuf, Jean de Meung - sont "
- Donnes0 et Donnes3 : "des premières universités"
- Donnes0 et Donnes4 : "des premières universités"
- Donnes0 et Donnes5 : "s’étend "
- Donnes0 et Donnes6 : "dans l"

Conclusion

- L'efficacité des arbres de suffixes en temps d'exécution dépendant surtout des chaînes de caractères en entrée (diversité de l’alphabet, fréquence des lettres .. etc)

- La solution d’optimisation avec les tuples stockant l’information (départ/fin) à la place des chaînes dans l’arbre de suffixe diminue drastiquement la complexité en espace

- **Erreurs d'implémentation a eviter :** créer des structure non adaptée à la compression ce qui a posé problème lors des appel récursives de compression des enfants