SCIENCES
SORBONNE
UNIVERSITÉ

Parcours STL
Rapport de projet : Detection de plagiat flagrant

Auteurs : Cherfi Farouck & Boumessaoud Abdelkader

Encadrants : Antoine Genitrini & Emmannuel Chailloux

Table des matières

1 Présentation

Le but de ce devoir est de résoudre un problème algorithmique d'exploration de chaînes en utilisant des structure de données différentes afin de comparer les performances de chaque solution proposée.

1.1 Énoncé du problème

Il s'agit d'une version simplifié de détection de plagiat : Étant donné deux chaînes de caractères, notées C1 et C2, le problème consiste à chercher les sous-chaînes communes à ces deux chaînes la plus grande. De plus il faudra indiquer leur positions dans C1 et C2. Rappelons qu'une sous-chaîne d'une chaîne de caractère C1 est constituée de caractères consécutifs dans C2.

Par exemple, étant donné les deux chaînes ANANAS et BANANE, la sous-chaîne commune la plus longue est ANAN.

2 Programmation dynamique

2.1 Sous-structure optimale

La première approche du projet est une approche par la programmation dynamique. Nous avons choisi comme sous-structure optimale une matrice[n+1][m+1] ou n et m sont les longueurs respectives de nos mots C1 et C2. On définit cette matrice et initialisent toutes ces cases à 0. On accumule ensuite la longueur du suffixe incluant le i-ème caractère de C1 et le j-ème caractère de C2. Pour calculer la longueur des sous-chaînes les plus longues entre C1 et C2 nous comparons deux caractères de C1 et C2 et si ils sont équivalents alors :

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

On peut alors retrouver les positions de la sous-chaîne dans les deux mots en utilisant :

(d,f) corresponds au caractère de début et de fin max corresponds à la valeur maximal de la matrice max(M(i)) corresponds à la ligne de l'entier le plus grand de la matrice max(M(j)) corresponds à la colonne de l'entier le plus grand

On considère 0 comme étant le premier caractère de notre chaîne.

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

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

2.2 Complexité

Nous mesurerons le nombre de comparaisons effectué sur les différentes approches. Ici dans le pire cas, nous allons parcourir tout les cases de notre matrice et les comparées une par une, nous allons donc avoir une matrice en (n+1)(m+1), au pire cas en temps et en espace notre algorithme sera en $O(n * m)$.

2.3 Code et jeu de tests

```
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;);
    done ;
  done;

  !maxlen, indicec1, indicec2 ;;

let () =
  assert(plsuffixe "ANANAS" "BANANE" = (4, [|0;3|], [|1;4|]));
  assert(plsuffixe "ABAB" "BABA" = (3, [|0; 2|], [|1; 3|]));
  assert(plsuffixe "CAMARCHE" "AHNONMARCHE" = (6, [|2; 7|], [|5; 10|]));;
```

3 Arbre des suffixes

Notre deuxième approche du projet est une approche des arbres des suffixes. En effet, nous pouvons retrouver une sous-chaîne de C1 en parcourant l'arbre des suffixes. Il existe donc un parcours dans l'arbre pour chacun de ces suffixes.

3.1 Structure

La structure de données que nous allons utiliser ici est de type Trie.

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

Dans notre structure, notre entier corresponds au numéro du mot permettant de marqué chaque noeud, par exemple si j'insère le mot "ANANAS#", tout ces noeuds seront marqués avec 1. Notre string corresponds à la chaîne de caractère du noeud, par exemple "A". Et enfin chaque noeud possède une liste de noeud correspondant a ces enfants.

On peut donc construire notre arbre vide tel que :

```
let vide = Node(0,"",[])
```

3.2 Primitives

Nous avons aussi utilisés quelques primitives utiles pour le parcours de notre structure ainsi que des getters.

```
(* Recupere les enfants du noeud actuel *)
let getenfant noeud =
  match noeud with
  | Node(i,s,tab) -> tab

(* Modifie une chaine de caracteres en liste de chaine de caractere *)
let explode wd =
  let rec exp i acc =
    if i = -1 then acc else exp (i-1) ((String.make 1 wd.[i])::acc) in
  exp (String.length wd - 1) []

(* Verifie si le noeud actuel contient l'enfant mot *)
let appartient mot noeud =
  let enfant = getenfant noeud in
  let rec parcours_enfant e =
    match e with
    | Node(i,s,l)::q -> if mot = s then (true,Node(i,s,l))
        else parcours_enfant q
    | [] -> (false,Node(0,"",[]))
  in parcours_enfant enfant ;;
```

```
(* Fonction filtre permettant de supprime un enfant deja existant *)
let filter mot l =
  List.filter (fun x ->
      match x with
      | Node(i,s,l) -> if s = mot then false else true ) l

(* Affiche les enfants (DEBUG) *)
let rec print_list_enfant noeud =
  match noeud with
  | [] -> ""
  | Node(i,s,l)::q -> s^print_list_enfant q


(* Ajoute le mot dans l'arbre avec un entier n *)
let ajout mot arbre n =
 let rec ajout_bis mot arbre =
    match (mot,arbre) with
    | [], Node(i,s,l) -> Node(i,s,l)
    | t::q, Node(i,s,l) ->
        let appartient_t = appartient t (Node(i,s,l)) in
        match appartient_t with
        | (false,_) -> (Node(i,s,ajout_bis q (Node(n,t,[])))::l))
        | (true,Node(i2,s2,l2)) -> if i2 != n then
              (Node(i,s,(ajout_bis q (Node(n+1,s2,l2 ))))::filter s2 l) )
            else (Node(i,s,(ajout_bis q (Node(i2,s2,l2 ))))::filter s2 l) )

  in ajout_bis (explode mot) arbre;;
```

3.3 Code ArbreSuffixes

On peut alors définir une fonction Arbresuffixes assez facilement avec nos anciennes primitives et notre fonction ajout.

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

3.4 Code Sous-Chaîne

Comme nous l'avons dit plus tôt pour savoir si C2 est une sous-chaîne de C1 alors, il existe un parcours dans notre arbre, il suffit alors de le parcourir jusqu'à la rencontre du caractère de fin # ou de la fin du motif que l'on recherche.

```
let souschaine mot1 mot2 =
  let arbre = arbresuffixe vide mot1 1 in
  let rec souschaine_bis mot arbre =
    match (mot, arbre) with
    | t::q, Node(i,s,l) ->
        if t = "#" then true else
          (let appartient_t = appartient t (Node(i,s,l)) in
           match appartient_t with
           | (false,_) -> false
           | (true,Node(i2,s2,l2)) -> souschaine_bis q (Node(i2,s2,l2)))
    | [], Node(_,s,l) -> true
  in souschaine_bis (explode mot2) arbre ;;
```

3.5 Jeu de tests

```
let () =
  assert(souschaine "ANANAS#" "NANAS" = true);
  assert(souschaine "ANANAS#" "NANAS#" = true);
  assert(souschaine "ANANAS#" "AS" = true);
  assert(souschaine "ANANAS#" "ANAS" = true);
  assert(souschaine "ANANAS#" "NAS" = true);
  assert(souschaine "ANANAS#" "S" = true);
  assert(souschaine "ANANAS#" "BANANE" = false);;
```

3.6 Complexités de souschaine et arbresuffixes

Pour l'algorithme arbresuffixes :
- Complexité temporelle : O(n^2) on ajoute chaque suffixe du mot soit n + (n-1) + (n-2) + ... + 1 ou n est la longueur du mot.
- Complexité en espace : O(n^2)

Pour l'algorithme souschaine :
- Complexité temporelle : O(m) ou m est la taille du motif
- Complexité en espace : O(m)

3.7 Code Souschainecommunes

Grâce a la création d'arbre des suffixes, on peut créer un arbre composé du premier mot et du second mot, nous avons ensuite juste à récupérer tout les noeuds marqués par les deux mots. Par exemple C1 sera marqué par 1 dans chaque noeud. On rajoutera C2 dans l'arbre des suffixes de C1 qui eux seront marqués par 2. Si il existe un noeud déjà marqué par 1 lorsqu'on rajoute un des suffixes de C2, alors ce noeud sera marqué par 3. Il suffira ensuite de récupérer tout les noeuds marqué par 3 et de récupérer le noeud avec la plus grande hauteur.

```
(* Recupere la chaine partant d'un noeud jusqu'a ces fils *)
let rec recup_chaine noeud =
  match noeud with
  | [] -> ""
  | Node(i,s,l)::q -> s^recup_chaine l ;;

(* Renvoie la hauteur de l'arbre *)
let rec hauteur_arbre arbre =
  match arbre with
  | Node(_,_,[]) -> 0
  | Node(i,s,l) -> 1 + hauteur_arbre (List.hd l) ;;

let souschainecommunes mot1 mot2 =
  let arbre = arbresuffixe vide mot1 1 in
  let arbre_mots = arbresuffixe arbre mot2 2 in
  let enfants = getenfant arbre_mots in
  let rec parcours_arbre_max e res=
    match e with
    | [] -> res::[]
    | Node(i,s,l)::q -> if i = 3
    then parcours_arbre_max q
    ((Node(i,s,List.hd (parcours_arbre_max l [])))::res)
        else (parcours_arbre_max q res)
  in let solu = parcours_arbre_max enfants [] in
  let rec plusgrand sol max noeud=
    match sol with
    | (Node(i,s,[]))::q -> plusgrand q max noeud
    | (Node(i,s,l))::q -> if max < (hauteur_arbre (List.hd l))
    then plusgrand q (hauteur_arbre (List.hd l)) (Node(i,s,l))
        else plusgrand q max noeud
    | [] -> noeud
  in recup_chaine (plusgrand (List.hd solu) 0 (Node(0,"",[]))::[]);;
```

3.8 Jeu de tests souschainecommune

```
let () =
  assert(souschainecommunes "ANANAS#" "BANANE#" = "ANAN");
  assert(souschainecommunes "CESTBONCAMARCHE#" "MAISNONCAMARCHEPAS#"
  = "ONCAMARCHE");;
```

3.9 Complexité souschainecommune

Pour l'algorithme de souschainecommune :
— Complexité temporelle : $O((n+m)^2)$ ou n et m sont les longueurs respectives des deux chaînes.
— Complexité en espace : $O((n+m)^2)$

4 Compression de l'arbre des suffixes

4.1 Code compression

Nous avons fait l'approche des arbres des suffixes, mais il existe une approche plus efficace concernant la même structure, les arbres des suffixes compressés. Nous pouvons donc implémenter une fonction compression qui utilisera la même structure d'arbre des suffixes.

```
(* Recupere la chaine de caractere de son unique enfant *)
let recup_chaine_un noeud =
  match noeud with
  | [] -> ""
  | Node(i,s,l)::q -> s


let compression arbre =
  let enfants = getenfant arbre in
  let rec parcours_arbre e =
    match e with
    | [] -> []
    | Node(i,s,l)::q ->
        if List.length l = 1
        then parcours_arbre (Node(i,s^recup_chaine_un(getenfant (Node(i,s,l))),
        parcours_arbre (getenfant (List.hd l)))::parcours_arbre q)
        else (Node(i,s,parcours_arbre l)::parcours_arbre q)
  in Node(0,"",parcours_arbre enfants);;
```

On marque un symbole # à la fin de nos chaînes de caractères pour préciser qu'il n'existe pas d'autre suffixes possible à partir de ce noeud. Quand nous arrivons à un noeud marqué de #, alors c'est une feuille.

4.2 Code SousChainesCommunesCompressé

On peut alors définir et adapter notre souschainecommunes sur des arbres compressés, il faudra alors comparé la hauteur des noeuds ainsi que la longueur de la chaîne de caractère présent dans les noeuds.

```
let souschainescommunescompress mot1 mot2 =
  let arbre = compression (arbresuffixe (arbresuffixe vide mot1 1) mot2 2)
in
  let enfants = getenfant arbre in
  let rec parcours_arbre_max e res=
    match e with
    | [] -> res::[]
    | Node(i,s,l)::q -> if i = 3
    then parcours_arbre_max q ((Node(i,s,
    List.hd (parcours_arbre_max l [])))::res)
        else (parcours_arbre_max q res)
  in let solu = parcours_arbre_max enfants [] in
  let rec plusgrand sol max noeud=
    match sol with
    | (Node(i,s,[]))::q -> plusgrand q max noeud
    | (Node(i,s,l))::q ->
        let som = ((hauteur_arbre (List.hd l)) +
        (String.length s) +
        String.length (recup_chaine l)) in
        if max < som
        then plusgrand q som (Node(i,s,l))
        else plusgrand q max noeud
    | [] -> noeud
  in recup_chaine (plusgrand (List.hd solu) 0 (Node(0,"",[]))::[]);;
```

4.3 Analyse du nombre maximum de noeuds

Un texte de N caractères, le nombre minimum de noeuds (en comptant la racine) est N+1 et son nombre maximum est de 2N-1

Chaque noeud interne introduit une branche dans l'arbre (car les noeuds internes d'un arbre de suffixe a au moins 2 enfants), chaque nouvelle branche doit éventuellement conduire à au moins une feuille supplémentaire, donc si nous avons K noeuds internes, il doit y avoir au moins K+1 feuilles mais le nombre de feuille est limités pas N donc le nombre maximum de noeuds internes est limités par N-2. Si c'est le cas, cela donne exactement N feuilles , 1 racines et au maximum de N-2 internes, soit 2n-1 noeuds.

4.4 Modification recherche sous-chaînes

Plutôt que de stocker une longueur de caractère proportionnelle à n , nous pouvons stocker en chaque noeud un couple d'entier correspondant au caractère de début et de fin. Malgré plusieurs tentatives pour l'adapter nous n'avons pas réussi à l'implémenter.

4.5 Complexité recherche sous-chaînes tuples

— Complexité temporelle : $O((n+m)^2)$ ou n et m sont les longueurs respectives des deux chaînes.
— Complexité en espace : $O((n)$ ou n est la longueur totale du mot

5 Construction efficace de l'arbre des suffixes compressé

Plutôt que de compresser un arbre de suffixe, nous pouvons créer un arbre de suffixe compréssé directement. Pour cela nous avons implémenter une nouvelle fonction d'ajout.

5.1 Code ArbreSuffixesCompresse

```
(*Permet de definir a quel endroit les deux mots sont differents *)
let compare_mot mot mot1 =
  let rec compare mot mot1 =
    match (mot,mot1) with
    | [], x -> x
    | x, [] -> x
    | t1::q1, t2::q2 -> if t1 = t2 then compare q1 q2
        else t1::q1
  in compare mot mot1;;

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

```
let ajout_mot_compress mot arbre =
  let rec ajout_bis mot arbre =
    match arbre with
    | Node(i,s,[]) -> Node(i,s,Node(0,mot,[])::[])
    | Node(i,s,l) ->
        let enfant = getenfant arbre in
        let rec parcours_enfant e =
          match e with
          | Node(i,s,l)::q -> if s.[0] == mot.[0] then
                let newmot = compare_mot (explode s) (explode mot)
                in
                let racine = (String.sub s 0
                ((String.length s) - (List.length newmot)))
                in
                let s_enfant = (String.sub s
                ((String.length racine)) ((String.length s)
                - (String.length racine)))
                in
                let s_enfant2 = (String.sub mot
                ((String.length racine)) ((String.length mot)
                - (String.length racine)))
                in
                let n_enfant = Node(0,s_enfant,l) in
                let n_enfant2 = Node(0,s_enfant2,[]) in
                Node(i,racine,n_enfant::n_enfant2::[])::q
              else Node(i,s,l)::parcours_enfant q
          | [] -> Node(0,mot,[])::[]
        in Node(i,s,parcours_enfant enfant);
  in ajout_bis mot arbre;;
```

5.2 Jeu de tests

```
let test = arbresuffixescompresse vide "ANANAS#";
=> Node (0, "",
    [Node (0, "A",
      [Node (0, "NA", [Node (0, "NAS#", []); Node (0, "S#", [])]);
       Node (0, "S#", [])]);
     Node (0, "NA", [Node (0, "NAS#", []); Node (0, "S#", [])]);
     Node (0, "S#", []); Node (0, "#", [])])
```

```
let test = arbresuffixescompresse vide "BANANE#";
=>  Node (0, "",
    [Node (0, "BANANE#", []);
     Node (0, "AN", [Node (0, "ANE#", []); Node (0, "E#", [])]);
     Node (0, "N", [Node (0, "ANE#", []); Node (0, "E#", [])]);
     Node (0, "E#", []); Node (0, "#", [])])
```

5.3 Complexité

Pour l'algorithme arbresuffixescompresse :
— Complexité temporelle : $O(n^2)$ ou n est la taille de la chaîne
— Complexité en espace : $O(n^2)$

6 Expérimentations

Nous avons pu essayer nos différentes structures de données sur une base de données de fichier textes.

6.1 Extraction des 150 premiers caractères de donnees0.txt et donnees1.txt

Nous avons utilisé notre approche par la programmation dynamique et des arbres suffixes compressés pour extraire les 150 premiers caractères de donnee0.txt et donnee1.txt La chaîne de caractère est : "D'abord confinée dans les monastères"

6.2 Temps de calcul sur [10, 20, 50, 75, 100, 125, 150, 200, 350, 400, 500, taille totale] caractères de donnees0.txt et donnees1.txt

6.3 Extraction des 150 premiers caractères sur les autres donnees.txt

— 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"

6.4 Temps de calcul sur [10, 20, 50, 75, 100, 125, 150, 200, 350, 400, 500, taille totale] caractères de donnees0.txt et donnees2 à donnees 6.txt