Option Informatique

d'après Centrale 2010 - Corrigé

2021 – 2022

I - Quelques exemples

1. Cas des langages reconnaissables

(a) On peut appliquer l'algorithme suivant en supposant disposer d'un automate fini déterministe reconnaissant le langage. On suppose de plus que l'on peut accéder à la i-ème lettre de mot m par la commande $m[i]$ où i varie entre 1 et $|m|$.

Algorithme 1 : Fonction : accepte m

$q \leftarrow q_i$
pour i *de* 1 *à* $|m|$ **faire**
  $q \leftarrow \delta(q, m[i])$
fin
retourner $q \in F$

On va passer $|m|$ fois dans la boucle, de ce fait, on détermine si m appartient ou pas au langage en un temps majoré par $|m|$. Le langage L est polynomial.

(b) Commençons par remarquer que,

$$\begin{aligned} \varepsilon \text{ est reconnu par } \mathscr{A} &\iff q_0 \in F \\ &\iff i \in F' \\ &\iff \varepsilon \text{ est reconnu par } \mathscr{S} \end{aligned}$$

Soit $m = m_0 \dots m_{n-1}$ un mot non vide.

— Si m reconnu par $\mathscr{A}$ et

$$q_0 \xrightarrow{m_0} q_1 \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n$$

un calcul dans $\mathscr{A}$ sur m réussi où$q_n \in F$, alors

$$i \xrightarrow{m_0} q_1 \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n$$

est un calcul sur m dans $\mathscr{S}$ réussi.

— Réciproquement, Si m reconnu par $\mathscr{S}$ et

$$i \xrightarrow{m_0} q_1 \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n$$

un calcul dans $\mathscr{S}$ sur m réussi où$q_n \in F$, (il ne peut pas revenir à i) alors

$$q_0 \xrightarrow{m_0} q_1 \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n$$

est un calcul sur m dans $\mathscr{A}$ réussi.

On a bien montré que le langage reconnu par $\mathscr{S}$ était le même que celui reconnu par $\mathscr{A}$.

(c) Justifions que $\mathscr{A}'$ reconnait le langage L^*.

— Soit $m \in L^*$. Si $m = \varepsilon$ il est reconnu par $\mathscr{A}'$ car $q_0 \in F'$. Sinon, $m = u_1 \cdots u_n$ où $u_1, \dots u_n$ sont des mots appartenant à L. Il existe donc, pour chaque u_i un calcul dans $\mathscr{A}$

$$q_0 \xrightarrow{u_i[0]} q_{i,1} \longrightarrow \cdots \longrightarrow q_{i,|u_i|-1}$$

où $q_{i|u_i|-1} \in F$. On en déduit le calcul suivant dans $\mathscr{A}'$

$$q_0 \xrightarrow{u_1[1]} q_{1,1} \longrightarrow \cdots \longrightarrow q_{1,|u_1|-1} \xrightarrow{u_2[1]} q_{2,1} \longrightarrow \cdots \longrightarrow q_{n,|u_n|-1}$$

où, par exemple, la transition, $(q_{1,|u_1|-1}, u_2[1], q_{2,1}) \in T'$ car $q_{1,|u_1|-1} \in F$ et $(q_0, u_2[1], q_{2,1}) \in T$. On en déduit que m est reconnu par $\mathscr{A}'$.

— Réciproquement, soit m un mot reconnu par $\mathscr{A}'$.
Si m est vide alors $m \in L^*$.
Sinon, on pose $m = m_0 \dots m_{n-1}$ et on considère un calcul

$$q_0 \xrightarrow{m_0} q_1 \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n$$

sur m réussi dans $\mathscr{A}'$, c'est-à-dire $q_n \in F'$. Si ce calcul n'utilise que des transitions de T (et pas les nouvelles transitions de $T' \setminus T$, alors c'est un calcul dans $\mathscr{A}$ et $q_n \in F$ puisque que q_n ne peut pas être égal à q_0 car l'automate est standard. On a donc $m \in L$.
Dans le cas contraire, on considère la première transition de $T' \setminus T$; $q_i \xrightarrow{m_{i+1}} q_{i+1}$. Par définition de T', $q_i \in F$ et donc $m_0 \dots m_i \in L$ et il existe une transition $q_0 \xrightarrow{m_{i+1}} q_{i+1}$ dans $\mathscr{A}$. On peut donc recommencer avec le calcul

$$q_0 \xrightarrow{m_{i+1}} q_{i+1} \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n.$$

On découpe alors le mots m en des facteurs dans L et il reste à la fin, un calcul

$$q_0 \xrightarrow{m_p} q_{p+1} \longrightarrow \cdots \xrightarrow{m_{n-1}} q_n$$

avec que des transitions dans $\mathscr{A}$. On est donc ramené au cas, ci-dessus. C'est un mot de L.
Finalement $m \in L^*$.

Notons que l'automate $\mathscr{A}'$ n'est pas déterministe mais que l'on peut le détermininiser.

2. Le cas de $L_0 = \{a^n b^n \mid n \in \mathbb{N}\}$.

(a) Supposons par l'absurde que L_0 soit reconnaissable. Soit $\mathscr{A} = (Q, q_0, \delta, F)$ un automate fini déterministe complet reconnaissant L_0. Il n'a qu'un nombre fini d'états donc il existe n et m distincts tels que $q = \delta(q_0, a^n) = \delta(q_0, a^m)$. Dès lors, $\delta(q, b^n) = \delta(q_0, a^n b^n) \in F$ car $a^n b^n \in L_0$ et $\delta(q, b^n) = \delta(q_0, a^m b^n) \notin F$ car $a^m b^n \notin L_0$. On a obtenu une absurdité donc L_0 n'est par reconnaissable.
On montre de même que L_0^* n'est pas reconnaissable car si $m \neq n$, $a^m b^n \notin L_0^*$.

(b) Commençons par un programme reconnaissant L_0
On parcours le mot m en incrémenant un compteur tant que la lettre vaut `a`. Puis, on décrémente ce même compteur quand la lettre vaut `b`. Il suffit alors de vérifier que le compteur vaut 0 et que l'on est arrivé au bout du mot

```
let recL0 m =
  let nb = ref 0 in
    let i = ref 0 in
        while !i < String.length m && (m.[!i] = `a`) do
          incr nb;
          incr i
        done;
        while !i < String.length m && m.[!i] = `b` do
          nb := !nb - 1;
          incr i
        done;
  !nb = 0 && !i = String.length m;;
```

(c) Ecrivons maintenant, sur le même principe une fonction qui reconnait les mots de L_0^*. On parcours le mot à l'aide de la référence `i`, on garde une variable booléen qui vérifie que tout ce que l'on a déjà lu est compatible. En particulier, quand on passe de la lecture d'un `b` à la lecture d'un `a` (ce que l'on sait à l'aide de la variable `danslesb` qui vaut vrai quand on lit des `b`) on vérifie que l'on a lu autant de `a` que de `b` en vérifiant que `nb` vaut 0.

```
let recL0star m =
  let possible = ref true in
  let nb = ref 0 in
  let i = ref 0 in
  let danslesb = ref false in
  while !possible && !i < String.length m do
    if m.[!i] = `a` then
      (if !danslesb then
        (danslesb := false;
        possible := !nb = 0);
      incr nb;
      incr i)
    else
      (danslesb := true;
      nb := !nb - 1;
      incr i)
    done;
  !possible && !nb = 0 && !i = String.length m;;
```

3. Un autre exemple.

(a) Le langage L_1 n'est pas reconnaissable. En effet, si $\mathscr{A} = (Q, q_0, \delta, F)$ est un automate déterministe complet reconnaissant L_1. Comme Q est fini, il existe $n < m$ tel que $q = \delta(q_0, a^{2^n}) = \delta(q_0, a^{2^m})$. Dès lors $q' = \delta(q, a^{2^n}) = \delta(q_0, a^{2^n} a^{2^n}) = \delta(q_0, a^{2^{n+1}}) \in F$ car $a^{2^{n+1}} \in L_1$. Mais d'autre part, $q' = \delta(q, a^{2^n}) = \delta(q_0, a^{2^m} a^{2^n}) = \delta(q_0, a^{2^n + 2^m})$. Maintenant, $2^n + 2^m = 2^n(1 + 2^{m-n})$ qui n'est pas une puissance de 2 car $1 + 2^{m-n}$ est impair. On en déduit que $q' \notin F$ ce qui est absurde. Le langage L_1 n'est pas reconnaissable.

(b) Montrons que L_1 est polynomial en exhibant un algorithme permettant de déterminer si un mot m appartient à L_1 et dont la complexité est majorée par $O(|m|)$. Il suffit en effet de savoir si la longueur du mot est d'une puissance de 2. Cela se fait en $O(\log_2 |m|)$.

Algorithme 2 : Fonction : appartientL1 m

$l \leftarrow$ longueur de m
$possible \leftarrow$ true
tant que $l > 1$ *et possible* **faire**
  si l *est pair* **alors**
    $l \leftarrow l/2$
  sinon
    $possible \leftarrow false$
  fin
fin
retourner *possible*

(c) On a alors $L_1^* = A^*$. En effet tout mot de A^* appartient à L_1^* car si $m = a^p \in A^*$, on peut décomposer p en base 2,

$$p = \sum_{i=0}^{r} \alpha_i 2^i \text{ où } \alpha_i \in \{0, 1\}.$$

On a alors

$$m = \prod_{\substack{0 \le i \le r \\ \alpha_i = 1}} a^{2^i} \in L_1^*.$$

(d) Le langage L_1^* est donc reconnaissable et polynomial.

4. Un dernier exemple

(a) Pour vérifier qu'un mot m appartient à L_2, il faut

— Vérifier qu'il est de la forme $u_1\#u_2\#u_3\#$ où u_1, u_2 et u_3 appartiennent à $\{0,1\}^*$. Cela se fait en temps linéaire par rapport à $|m|$.
— Calculer les entiers n_1, n_2 et n_3 tels que $\varphi(n_i) = u_i$. Avec un algorithme de type Horner, le calcul de n_i est linéaire par rapport à $|u_i|$ donc par rapport à $|m|$.
— On vérifie que $n_1 \times n_2 = n_3$. Le calcul de $n_1 \times n_2$ a une complexité majorée par $O(|u_1| \times |u_2|)$ donc par $O(|m|^2)$.

Le langage L_2 est bien polynomial.

Notons, que l'on peut se passer de calculer n_1, n_2 et n_3 en travaillant qu'au niveau des décompositions binaires.

(b) Le langage L_2 n'est pas reconnaissable. En effet, si $\mathscr{A} = (Q, q_0, \delta, F)$ est un automate déterministe complet reconnaissant L_2. Comme Q est fini, il existe $n < m$ tel que $q = \delta(q_0, 1^n\#) = \delta(q_0, 1^m\#)$. Dès lors $q' = \delta(q, 1\#1^n\#) = \delta(q_0, 1^n\#1\#1^n\#) \in F$ car $1^n\#1\#1^n\# \in L_2$. Mais d'autre part, $q' = \delta(q, 1\#1^n\#) = \delta(q_0, 1^m\#1\#1^n\#) \notin F$ car $1^m\#1\#1^n\# \notin L_2$ ce qui est absurde. Le langage L_2 n'est pas reconnaissable.

II - Trois algorithmes

5. Une énumération des parties de $[[0, n-1]]$

(a) On applique l'algorithme classique de décomposition en base 2. On place le bit de poids fort dans la case d'indice $n-1$.

```
let dec2 k n =
  let res = Array.make n 0 in
  let u = ref k in
  let i = ref 0 in
  while !u <> 0 do
    if !u mod 2 = 1 then res.(!i) <- 1;
    incr i;
    u := !u / 2
  done;
  res;;
```

(b) On associe à tout tableau de n cases contenant des 0 et des 1 une partie de $[[0, n-1]]$. Précisément, à un tableau `t` on associe la partie $X = \{i \mid t.(i) = 1\}$. On peut ainsi obtenir une bijection entre les entiers de 0 à $2^n - 1$ et les parties de $[[0, n-1]]$.

(c) On considère un mot m de longueur $|m|$. On numérote ses lettres de 0 à $|m| - 1$. Pour toute partie non vide de $[[0, p]]$ où $p = |m| - 2$, on peut « couper » le mot après les lettres dont les numéros sont dans la partie. Par exemple pour le mot `m = centralesupelec` on a $|m| = 15$, donc $p = 13$ et pour $X = \{3, 7\}$ on coupe après la lettre 3 qui est `t` et après la lettre 7 qui est `e`. On obtient `cent.rale.supelec`.

(d) On veut écrire une fonction `dansLetoile : (string -> bool) -> string -> bool` permettant de déterminer si un mot `m` appartient à L^* en utilisant la fonction `dansL : string -> bool`.
— Si le mot `m` est de longueur 0, c'est le mot vide et il est dans L^*.
— Si le mot `m` est de longueur 1, on utilise `dansL m`.
— Sinon, on va engendrer toutes les décompositions possibles du mot `m` en utilisant ce qui précède. Pour cela on fait varier k entre 1 (et pas 0 car on ne veut pas de décomposition vide) et $2^n - 1$ où n est le nombre de lettres moins 1. Pour chaque décomposition (qui est codée par le tableau donné par `dec2 k n`) on vérifie si chaque terme est dans L à l'aide de la fonction auxiliaire `test : string -> int array -> bool`.

```
let dansLetoile dansL m =
  let nblettres = String.length m in
  if nblettres = 0 then true
  else
  if nblettres = 1 then dansL m
  else (let n = nblettres - 1 in
    let max = (exporap n) - 1 in
    let trouve = ref false in
    for k = 1 to max do
      let test m tab =
        let rec aux i j =
          if j >= n then dansL (String.sub m i (j - i))
          else if tab.(j) = 0 then aux i (j + 1)
            else dansL (String.sub m i (j - i)) && aux j j
        in aux 0 0
      in
      trouve := test m (dec2 k n)
    done;
  !trouve);;
```

(e) Le programme précédent fait passer tous les entiers k compris entre 1 et $2^n - 1$ où $n = |m| - 1$. A chaque passage dans cette boucle, on
— Réalise la décomposition en base 2 qui nécessite dans le pire des cas n divisions euclidiennes.
— Appelle la fonction dansL autant de fois que le mots a de facteurs dans la décomposition considérée. Cela peut aller de 1 à n.
La complexité globale est donc en $O(n2^n)$.

6. Un algorithme récursif

(a) Soit $n \geq 1$ et $m_0, \dots, m_{n-1}$ des lettres de A. Le mot $m = m_0 \dots m_{n-1}$ n'est pas le mot vide donc

$$\begin{aligned} m \in L^* &\iff m \in L \text{ ou } \exists p \geq 1, m \in L^{p+1} \\ &\iff m \in L \text{ ou } \exists p \geq 1, \exists k \in [\![0, k-2]\!], m_0 \dots m_k \in L \text{ et } m_{k+1} \dots m_{n-1} \in L^p \\ &\iff m \in L \text{ ou } \exists k \in [\![0, k-2]\!], m_0 \dots m_k \in L \text{ et } m_{k+1} \dots m_{n-1} \in L^* \end{aligned}$$

(b) On utilise le principe précédent.
— Si m est le mot vide, il est dans L^*
— Si m est dans L, il est dans L^*
— Sinon, on teste à l'aide de la fonction récursive aux s'il existe un entier $k \in [\![0, k-2]\!]$ tel que $m_0 \dots m_k \in L$ et $m_{k+1} \dots m_{n-1} \in L^*$. Pour cela, on commence à tester pour $k = 0$ et si ce n'est pas le cas, on recommence pour $k = 1$ et ainsi de suite. La fonction récursive s'arrêtant à $k = n - 1$

```
let rec dansLetoile2 dansL m =
  let n = String.length m in
  let rec aux k =
    if k = n - 1 then false
    else
      (
      (dansL (String.sub m 0 (k + 1))) &&
      (dansLetoile2 dansL (String.sub m (k + 1) (n - k - 1)))
      ) || aux (k + 1)
  in
  (m = "") || (dansL m) || aux 0;;
```

(c) Si on note $c(n)$ le nombre d'appels à la fonction dansL dans le pire des cas pour un mot de longueur n. On a $c(1) = 1$ et pour tout entier $n \geq 1$,

$$c(n+1) = 1 + \sum_{k=0}^{n-1} (1 + c(n+1-k-1)) = (n+1) + \sum_{k=1}^{n} c(k).$$

En effet, k peut prendre toutes les valeurs entre 0 et $(n+1)-2 = n-1$ et pour k donné, il y a un appel à `dansL` et un appel à `dansLetoile2` sur un mot de longueur $n+1-k-1$. En calculant les premiers termes, on obtient $c(1) = 1; c(2) = 2 + c(1) = 3; c(3) = 3 + c(1) + c(2) = 7$. Par une récurrence, on montrer simplement que $c(n) = 2^n - 1$.

Ce cas est atteint quand un mot n'appartient pas à L^*.

7. Une programmation dynamique

   (a) Soit $i \in [[0, n-1]]$ la valeur de $T_{i,i}$ dépend du fait que m_i appartiennent à L ou non. C'est donc `dansL m.[i]`.

   (b) Comme à la question, 6.a, le mot $m_i \dots m_j$ appartient à L^* si et seulement, il appartient à L ou il existe k tel que $m_i \dots m_k$ appartiennent à L (ou à L^*) et $m_{k+1} \dots m_j$ aussi. On a donc

$$T_{i,j} = (m_i \dots m_j \in L) \bigvee \left(\bigvee_{k=i}^{j-1} T_{i,k} \wedge T_{k+1,j} \right).$$

   (c) On va construire un tableau `t` tel que `t.(i).(j)` va contenir la valeur de $T_{i,j}$.
   — On commence en initialisant toutes les valeurs à `false`
   — Pour i allant de 0 à $n-1$, on place `dansL m.[i]` dans la case `t.(i).(i)`
   — On remplit alors le tableau en utilisant la formule de la question précédente, on remplissant d'abord les case `t.(i).(i+1)` puis `t.(i).(i+2)` et ainsi de suite.

   (d)

```
let dansLetoile3 dansL m =
  let n = String.length m in
  let t = Array.make_matrix n n false in
  for i = 0 to (n - 1) do
    t.(i).(i) <- dansL (string_of_char m.[i])
  done;
  for p = 1 to (n - 1) do
    for i = 0 to (n - 1 - p) do
      t.(i).(i + p) <- dansL (String.sub m i (p + 1));
      for k = i to (i + p - 1) do
        t.(i).(i + p) <- t.(i).(i + p) || (t.(i).(k) && t.(k + 1).(i + p))
      done
    done
  done;
  t.(0).(n - 1);;
```

   (e) En dénombrant les passages dans les boucles, on obtient $O(n^2)$ appels à `dansL` et $O(n^3)$ opérations logiques.

III - Utilisation d'un graphe

8. Structure de file

   (a)

```
let creerFile n = {contenu = Array.make n 0 ; debut = 0 ; fin = -1};;

let estVide f = f.fin < f.debut;;

let put x f = f.fin <- f.fin + 1 ; f.contenu.(f.fin) <- x;;

let get f = f.debut <- f.debut + 1 ; f.contenu.(f.debut -1);;
```

   (b) On peut utiliser le tableau comme un cylindre. Quand on arrive à la fin du tableau on recommence à placer des éléments au début du tableau. De fait, si `findebut` les éléments de la files sont contenus entre `début` et la fin du tableau puis entre le début du tableau et `fin`.

9. Introduction d'un graphe orienté

(a) Voici $G_{L_0}(aabbaba)$ où L_0 est le langage de la question 2.

$$
\begin{tikzcd}[column sep=small]
-1 \arrow[rrrr, bend right, "aabb"'] & 0 \arrow[rr, bend left, "ab"] & 1 & 2 & 3 \arrow[rr, bend left, "ab"] & 4 & 5 & 6
\end{tikzcd}
$$

(b) On construit un tableau ne contenant que des listes vides. Ensuite pour $i < j$ si $m_{i+1} \dots m_j$ est dans L on ajoute j à la i-ème liste d'adjacence.

```
let construitGraphe m dansL =
  let n = String.length m in
  let t = Array.make n [] in
  for i = 0 to (n - 2) do
    for j = i + 1 to n - 1 do
      if dansL (String.sub m (i + 1) (j - i)) then t.(i) <- j :: t.(i)
    done;
  done;
  t;;
```

(c) On réalise un parcours en largeur en partant su sommet 0. A la fin du parcours, on vérifie si le sommet $n - 1$ a été atteint.

```
let dansLetoile4 dansL m =
  let g = construitGraphe m dansL in
  let n = Array.length g in
  let visit = Array.make n false in
  let f = creerFile (4 * n) in
  put 0 f;
  let rec enfile f liste =
    match liste with
      | [] -> ()
      | s :: q -> put s f; enfile f q
    in
  while not (estVide f) do
    let s = get f in
    if not visit.(s) then
      (visit.(s) <- true;
      enfile f g.(s))
  done;
  visit.(n - 1);;
```

(d) La création du graphe se fait avec $O(n^2)$ appels à `dansL`. Le parcours en largeur du graphe qui a n sommets et au plus n^2 arêtes nécessite $O(n^2)$ opérations.