Éléments de correction du TP n° 2

BATOG Guillaume

7 décembre 2006

1 Mots reconnus par un automate déterministe

1.1 Construction des langages $L_{n,p,q}$

On considère un automate A de fonction de transition δ. On note $A[p,q]$ l'automate dont l'unique état initial est p et l'unique état final est q. On va construire récursivement les ensembles de mots de longueur n reconnus sur ces automates : $L_{n,p,q} = \{w \mid |w| = n \text{ et } w \in \mathcal{L}(A[p,q])\}$. On obtient la relation de récurrence suivante, où $U \cdot V = \{uv \mid u \in U,\ v \in V\}$

$$\forall n \geq 0,\ \forall p,q \in Q, \quad L_{n+1,p,q} = \bigcup_{x \in \Sigma} \{x\} \cdot L_{n,\delta(p,x),q} \qquad \text{et} \qquad L_{1,p,q} = \{x \in \Sigma \mid \delta(p,x) = q\}.$$

Le langage reconnu par l'automate A est alors le suivant :

$$\mathcal{L}(A) \backslash \{\epsilon\} = \bigcup_{n \geq 1} \bigcup_{f \in F} L_{n,i,f}.$$

En Caml, on représentera un mot à l'aide du type suivant :

```
type lettre = Lettre of int;;
type mot = lettre list;;
```

La réunion dans la relation de récurrence est une réunion disjointe (les mots ne commencent pas par la même lettre) donc on peut implémenter les ensembles $L_{n,p,q}$ sous forme d'une liste de mots (`mot list`), l'opération de réunion consistant simplement en une opération de concaténation de deux listes. L'opération d'adjonction d'une lettre est la suivante :

```
let rec addLettre x = function
        | [] -> []
        | t::q -> (x::t)::(addLettre x q);;
```

À l'aide de la relation de récurrence plus haut, on construit un tableau `tab` à trois dimensions où `tab.(n).(p).(q)` contient l'implémentation de $L_{n,p,q}$ puis on retourne la liste des $L_k(A)$ pour k variant de 0 à n.

1.2 Tri lexicographique

Définition. Soit $<_i$ un ordre sur un ensemble X_i. On étend ces ordres sur $X_1 \times ... \times X_n$ par la relation $<$ notée $(<_1, ..., <_n)_{lex}$ appelée composée lexicographique des ordres $<_i$ et définie par

$$(a_1, ..., a_n) < (b_1, ..., b_n) \ \Leftrightarrow\ \exists\, j \in [\![1,n]\!],\ \forall i \in [\![1, j-1]\!],\ \ a_i =_i b_i \ \text{ et } \ a_j <_j b_j.$$

Proposition. Si $<_1, ..., <_n$ sont des ordres bien fondés, alors $(<_1, ..., <_n)_{lex}$ est bien fondé. (Un ordre $<$ sur X est dit bien fondé si toute suite de X strictement décroissante pour $<$ est finie ce qui est utile pour les preuves de terminaison de programmes.)
Pour un ordre $<$ sur X, on considère couramment l'extension $<_{lex}$ sur les mots de X définie par $x <_{lex} y$ si et seulement si x est un préfixe strict de y ou $(x_1, ..., x_n)(<, ..., <)_{lex}(y_1, ..., y_n)$

avec $n = \min(|x|, |y|)$. C'est pourquoi dans la définition Caml, une lettre est étiquetée par un entier pour obtenir un ordre pratique sur les lettres.

Pour chaque ensemble $L_k(A)$, on effectue un tri lexicographique.
Première méthode : écrire une fonction `ordreLexico` qui teste l'ordre lexicographique puis l'utiliser dans un tri fusion ; si n est le nombre de mots à trier et k la longueur de tous ces mots, on obtient une complexité en $O(kn \log n)$.
Deuxième méthode : On suppose que l'alphabet est de taille `nAlph` (variable globale, les lettres sont alors numérotées de 0 à `nAlph`-1). Cette information supplémentaire sur les mots à trier permet d'obtenir un algorithme de complexité linéaire en $O(kn)$.

```
let rec triLexico motListe = match motListe with
  | [] -> []
  | [[]] -> [[]]
  | _ ->
      let baquet = make_vect nAlph [] in
      let rec split = function
        | [] -> ()
        | mot::q -> ( match hd(mot) in
                      | Lettre x -> baquet.(x) <- tl(mot)::(baquet.(x)) ;
                                    split q )
      in
      split motListe;
      let partiel = map_vect triLexico baquet in
      let addLettreI i = addLettre (Lettre i) (partiel.(i)) in
      let partiel2 = init_vect (vect_length partiel) addLettreI in
      it_list (prefix @) [] (list_of_vect partiel2);;
```

Enfin on concatène tous les $L_k(A)$ triées pour obtenir la liste triés des mots de longueur inférieure à n reconnus par l'automate A.

2 Opérations sur les automates

2.1 Produit d'automates

Le produit de deux automates est une construction qui permet d'obtenir un automate reconnaissant les langages $A \cup B$, $A \cap B$, $A \backslash B$, $A \Delta B$ où A et B sont reconnaissables. Si $\mathcal{A}'$ et $\mathcal{A}''$ sont deux automates définis sur un même langage Σ, on définit formellement l'automate produit par

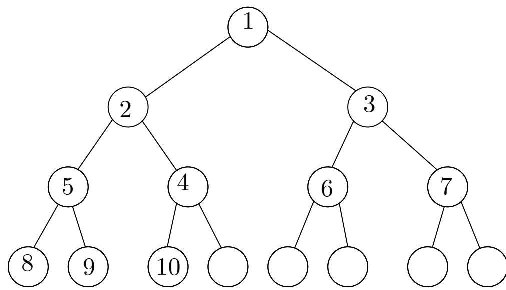
$$\mathcal{A}' \times \mathcal{A}'' = (Q' \times Q'', \Sigma, \tilde{\delta}, I' \times I'', \tilde{F})$$

$$\text{où} \quad ((p', p''), a, (q', q'')) \in \tilde{\delta} \quad \Longleftrightarrow \quad (p', a, q') \in \delta' \ \text{ et } \ (p'', a, q'') \in \delta''$$

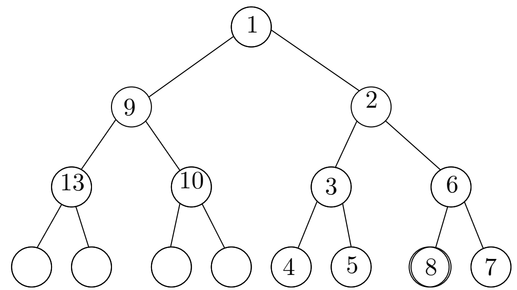
Un chemin dans l'automate produit simule en parallèle deux chemins de même étiquette dans les automates correspondants. On choisit $\tilde{F}$ égal à $F' \times F''$ pour reconnaître l'intersection des langages des deux automates initiaux, $F' \times Q'' \cup Q' \times F''$ pour la réunion, $F' \times (Q'' \backslash F'')$ pour la différence ($\mathcal{A}''$ doit être déterministe complet), $F' \times (Q'' \backslash F'') \cup (Q' \backslash F') \times F''$ pour la différence symétrique (les deux automates doivent être déterministes complets). De plus, le produit d'automates déterministes est déterministe : cette méthode est donc très pratique d'un point de vue algorithmique, la seule difficulté étant de coder les produits d'états en la structure adéquate. Par exemple, si $Q' = \{0, 1, ..., N-1\}$ et $Q'' = \{0, 1, ..., M-1\}$, on code le couple (p, q) par $pN + q$: on obtient une bijection entre $Q' \times Q''$ et $\{0, ..., N + M - 1\}$ (la réciproque étant facilement calculable...).

2.2 Partie accessible d'un automate

La partie accessible d'un automate est l'ensemble de ses états qui peuvent être atteints par un chemin à partir d'un état initial. Dans ce problème, on ne se soucie plus de l'étiquette des



Exemple de parcours en largeur



Exemple de parcours en profondeur

transitions/chemins, il suffit de considérer le graphe sous-jacent à l'automate. Les solutions algorithmiques de ce problème repose tout simplement sur un *parcours* de l'automate / du graphe. Il en existe deux types : en largeur et en profondeur. Des algorithmes plus efficaces seront étudiés au cours du TP à venir sur les graphes.

2.2.1 Une solution de parcours en largeur

Si on note $\text{Succ}(P) = \{\delta(p, a) \,|\, a \in \Sigma,\ p \in P\}$ les états accessibles à partir des états de P à l'aide d'une seule transition, l'ensemble R des états accessibles de l'automate est le plus petit point fixe (au sens de l'inclusion) de la fonctionnelle ϕ contenant $\{i\}$ (cf définition de l'exercice) où $\phi(P) = P \cup \text{Succ}(P)$. On définit par récurrence les ensembles A_n où $A_0 = \{i\}$ et $A_{n+1} = \phi(A_n)$. La suite $(A_n)_{n\geq 0}$ est strictement croissante tant que R n'est pas atteint et R est de cardinal au plus N (nombre d'états de l'automate) donc $R = A_j$ avec $j \geq N$. Sans parler de plus petit point fixe, A_n représente l'ensemble des états accessibles à partir de i par un chemin de longueur au plus n. Or tout état accessible l'est par un chemin de longueur au plus $N - 1$: supposons que tous les chemins permettant d'accéder à un état q sont de longueur au moins N, soit un tel chemin C de longueur minimale $\geq N$. Il comporte au moins $N + 1$ états dont au moins deux égaux à q' : C est de la forme $i \to q' \to q' \to q$. Le nouveau chemin $i \to q' \to q$ est de longueur strictement inférieure ce qui contredit la minimalité de C.

Dans cette algorithme, on calcule la suite des $(A_n)_{n\geq 0}$ tant qu'elle ne stationne pas. Cet algorithme s'arrête d'après ce qui vient d'être dit. On implémente les ensembles par des listes, on prendra garde à ce que les éléments de la liste soient deux à deux distincts : on construira alors une fonction `insere` qui insère un élément en tête de liste s'il n'appartient pas à cette liste. Ainsi on peut tester facilement si $A_{n+1} = \phi(A_n)$ en comparant les têtes des listes représentant A_{n+1} et A_n, ce qui est implémenté dans `itere`. La fonction `succ` implémente la fonctionnelle ϕ. La fonction générale de calcul de R est notée `etatsAcc`. Pour le cas particulier du TD, on obtient :

```
let etatsAcc auto =
 let d = auto.delta in
 let rec succ stockAcc = function
   | [] -> stockAcc
   | t::q ->  let newStock = insere (d.(t).(1)) ( insere (d.(t).(0)) stockAcc ) in
              succ newStock q
 in
 let rec itere acc_n =
   let acc_n2 = succ acc_n acc_n in
   if hd(acc_n2) = hd(acc_n)
      then acc_n
      else itere acc_n2
 in
 itere [auto.initial];;
```

2.2.2 Une solution de parcours en profondeur

L'idée d'un parcours en profondeur est de parcourir tant qu'on peut parcourir en stockant les noœuds non traîtés dans une pile. Le parcours sera terminé lorsque la pile sera épuisée. Dans l'exemple de la figure, lorsqu'on se trouve dans l'état numéroté 4, la pile est formée des états numérotés \$-9-6-5. Dans l'algorithme ci-dessus, la liste `stock` tient lieu de pile des états à traîter et la liste `acc` contient tous les états accessibles qui ont déjà été rencontrés. La fonction `cloture` termine : en effet si on note N le nombre d'états de l'automate, la mesure $(N - |acc|, |stock|)$ décroît strictement pour l'ordre lexicographique après chaque exécution de la fonction `cloture`. Le programme ci-dessus est récursif terminal.

```
let etatsAcc auto =
  let d = auto.delta in
  let misajour a b q =
      if not (mem q a)
         then (q::a,q::b)
         else (a,b)
  in
  let rec cloture acc stock = match stock with
      | [] -> acc
      | t::q -> let (a2,s2) = misajour acc q (d.(t).(0)) in
                let (a2,s2) = misajour acc q (d.(t).(0)) in
                cloture a3 s3
  in cloture [auto.initial] [auto.initial];;
```

3 Problèmes de décidabilité

Ceci n'a pas été donné en TP, mais il est possible de déterminiser un automate, à l'aide de la méthode sous-ensemble : pour tout automate fini $A = (Q, \Sigma, \delta, I, F)$, on construit l'automate fini déterministe et complet $B = (\mathscr{P}(Q), \Sigma, \delta', \{I\}, F')$ tel que $\mathcal{L}(A) = \mathcal{L}(B)$ avec $F' = \{S \subset Q \mid S \cap F \neq \varnothing\}$ et l'application de transition δ' définie par

$$\forall S \in \mathscr{P}(Q), \forall a \in \Sigma, \quad S \cdot a = \{q \in Q \mid \exists q' \in Q, ((q', a), q) \in \delta\}.$$

La partie accessible de l'automate B est appelée déterminisé de A. Il existe des automates à n états pour lesquels le déterminisé est de taille 2^n, d'où une complexité exponentielle dans le pire des cas.

Les problèmes suivants sont décidables :

- Un mot appartient-il au langage d'un automate ? Se fait en temps linéaire.
- Soit A un automate, $\mathcal{A}$ est-il vide ? fini ? infini ? Se fait en temps polynômial. On utilise ici la technique du parcours du graphe sous-jacent. Le langage est vide si et seulement si les états finaux ne sont pas accessibles. Le langage est fini si et seulement si l'automate ne possède pas de cycle : dans nos algorithmes, on peut détecter un cycle lors du test d'appartenance aux sommets déjà rencontrés dans les fonctions `insere` et `misajour`.
- Soient A et B deux automates, $\mathcal{A} = \Sigma^*$? $\mathcal{A} \subseteq \mathcal{B}$? Se fait en temps exponentiel (à cause du passage au complémentaire qui nécessite une déterminisation de l'automate).

Bilan : le but du TP était de montrer qu'on pouvait décider si un système d'équations diophantiennes linéaires possédait une solution à l'aide d'automates finis. Dans le cas d'équations diophantiennes générales, ce problème est indécidable (il n'est pas possible de trouver un algorithme qui répond OUI si l'équation possède une solution et qui répond NON si elle n'en possède pas).