LYCÉE LECONTE DE LISLE

Algorithme de McNaughton et Yamada

Vincent Picard

1
L'algorithme

Des automates vers les expressions régulières

- L'algorithme que nous allons présenter est (encore) une autre manière de transformer un automate fini en expression régulière.

- Son fonctionnement repose sur la même idée que celui de **Floyd-Warshall** qui détermine les distances entre toutes les paires de sommets d'un graphe pondéré.

- Soit $A = (Q, I, F, T)$ un automate fini non déterministe. L'ensemble de ses transitions est modélisé par une partie $T \subset Q \times \Sigma \times Q$.

- Supposons que les états sont numérotés de 0 à $n-1$: $Q = \{q_0, q_1, \dots, q_{n-1}\}$.

Équations de récurrence

- Soit $i, j \in [\![0, n-1]\!]$ des numéros d'états. Soit un entier $k \in [\![0, n]\!]$.
- On note $L_{i,j,k}$ le langage des mots sur Σ qui mènent de l'état q_i à q_j **en ne passant que par des états intermédiaires** dans $\{q_0, \dots, q_{k-1}\}$. C'est-à-dire qu'on ne s'autorise à transiter qu'entre les k premiers états de l'automate.
- On se propose de trouver par récurrence une expression régulière dénotant $L_{i,j,k}$.
- Si $k = 0$, alors les seuls chemins possibles de q_i à q_j sont des boucles :

$$L_{i,j,k} = \left\{x \in \Sigma,\ (q_i, x, q_j) \in T\right\}$$

- Si $k > 0$, alors un chemin de q_i à q_j
  - soit ne passe pas par q_{k-1}, alors il est étiqueté dans $L_{i,j,k-1}$
  - soit passe un $p > 0$ fois par q_{k-1} et est étiqueté par un mot $u = sv_1v_2 \dots v_{p-1}t$ où s est le mot qui mène à q_{k-1} pour la première fois, $v_1, \dots, v_{p-1}$ les mots qui ramènent à q_{k-1} et enfin t le mot qui mène de q_{k-1} à q_j.
  - Au final :

$$L_{i,j,k} = L_{i,j,k-1} \cup L_{i,k-1,k-1} \cdot L^*_{k-1,k-1,k-1} \cdot L_{k-1,j,k-1}$$

Algorithme de McNaughton et Yamada

- L'algorithme consiste donc à construire des matrices $(M_{i,j}^{(k)})$ qui contiennent des expressions régulières pour dénoter $L_{i,j,k}$.
- **Initialisation** : $M_{i,j}^{(0)} = \sum_{x/(q_i,x,q_j)\in T} x$
- **pour k allant de 1 à n** :

$$M_{i,j}^{(k)} = M_{i,j}^{(k-1)} + M_{i,k-1}^{(k-1)} {M_{k-1,k-1}^{(k-1)}}^{*} M_{k-1,j}^{(k-1)}$$

- **Considérations finales :**
  - Lors de l'état initial l'algorithme ignore la possibilité de chemin vide ε, il faut donc ajouter à la fin un ε sur la diagonale de la matrice.
  - On reconstruit l'expression régulière cherchée comme la somme de toutes les expressions régulières menant d'un état initial à un état final...

2
Programmation en OCaml

Les types

```
type etat = int;;

type auto = {
    taille: int;
    init: etat list;
    final: etat list;
    trans: (etat * char * etat) list;
};;

type regexp =
    | Vide
    | Epsilon
    | Lettre of char
    | Concat of regexp * regexp
    | Union of regexp * regexp
    | Etoile of regexp
;;
```

Définition d'opérateurs

En OCaml, on peut définir des nouveaux opérateurs, par exemple l'**union** d'expressions régulières :

```
let ($+) e1 e2 = match e1,e2 with
    | Vide, e2 -> e2
    | e1, Vide -> e1
    | _ -> Union(e1, e2)
;;
```

Définir vous-même un opérateur `$.` similaire pour la **concaténation**.

Initialisation de l'algorithme

Écrire une fonction `matrice_initiale : auto -> regexp array array` qui construit la matrice $M^{(0)}$ associée à un automate.

Initialisation de l'algorithme : solution

Écrire une fonction `matrice_initiale : auto -> regexp array array` qui construit la matrice $M^{(0)}$ associée à un automate.

```
let matrice_initiale a =
    let n = a.taille in
    let m = Array.make_matrix n n Vide in
    let ajoute_trans (i, x, j) =
        m.(i).(j) <- m.(i).(j) $+ (Lettre x)
    in
    List.iter ajoute_trans a.trans;
    m
;;
```

Calcul des $M^{(k)}$

```
let mcnaughton a =
    let n = a.taille in
    let m = matrice_initiale a in
    for k = 1 to n do
        let mnew = Array.make_matrix n n Vide in
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                mnew.(i).(j) <-
                    m.(i).(j) $+
                    (m.(i).(k-1) $. (Etoile m.(k-1).(k-1)) $. m.(k-1).(j))
            done
        done;
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                m.(i).(j) <- mnew.(i).(j)
            done
        done
    done;
    m
;;
```