

Automates finis non déterministes

OPTION INFORMATIQUE - TP n° 4.1 - Olivier Reynet

À la fin de ce chapitre, je sais :

- reconnaître un automate fini non déterministe (AFND)
- déterminiser un AFND
- expliquer comment éliminer les transitions spontanées

A Déterminisme?

Soient les automates décrits sur les figures 1, 2 et 3 sur l'alphabet $\Sigma = \{a, b\}$.



FIGURE 1 – \mathcal{A}_1



FIGURE 2 – \mathcal{A}_2



FIGURE 3 – \mathcal{A}_3

- A1. Les automates \mathcal{A}_1 , \mathcal{A}_2 et \mathcal{A}_3 sont-ils déterministes? Pourquoi?
- A2. L'automate \mathcal{A}_1 reconnaît-il le mot aabbb? Construire l'arbre de l'exécution de cet automate pour ce mot.

A3. Même question pour le mot bbbaa et l'automate \mathcal{A}_2 .

A4. Même question pour le mot abab et l'automate \mathcal{A}_3 .

A5. Décrire dans le langage naturel les langages reconnus par les automates \mathcal{A}_1 , \mathcal{A}_2 et \mathcal{A}_3 .

B Déterminisation d'un AFND

Soit l'automate \mathcal{A} suivant :



B1. Pourquoi l'automate suivant est-il non déterministe?

B2. Quel est le langage reconnu par cet automate?

B3. Déterminiser l'automate fini non déterministe \mathcal{A} suivant. On procédera en construisant le tableau des états de l'automate déterministe associé.

B4. Dessiner l'automate déterministe calculé précédemment.

C Modélisation d'un automate non déterministe en OCaml

On choisit de modéliser un automate fini non déterministe par un type algébrique de la manière suivante : les états sont représentés par des types `int`. Les lettres sont des types `char`. On représente les états par une `int list` et l'alphabet par une `char list`. On spécifie les états initiaux et accepteurs par une `int list`. Les transitions possibles forment une `(int * char * int) list`. Cette solution d'implémentation présente l'avantage de coller au plus prêt à la définition mathématique d'un automate.

```

1 type ndfsm = { states : int list;
2               alphabet : char list;
3               initial : int list;
4               transitions : (int * char * int) list;
5               accepting : int list};;
```

C1. Créer une variable `automata` qui représente l'automate non déterministes \mathcal{A} de la section B.

D Codage de l'algorithme de déterminisation

On souhaite implémenter l'algorithme de déterminisation d'un automate fini non déterministe. Pour cela, on choisit de représenter un élément de $\mathcal{P}(Q)$, l'ensemble des parties de Q , par une `int list`, c'est à dire une liste d'états. Si le nombre d'états de l'automate non déterministe est n , alors le cardinal de

$\mathcal{P}(Q)$ est 2^n . C'est pourquoi on choisit de coder chaque état par un nombre entre 0 et $2^n - 1$. Ce nombre est construit d'une manière univoque comme suit : chaque état de l'automate de départ est codé de 0 à $n - 1$; chaque état associé à un élément de $\mathcal{P}(Q)$ est obtenu en effectuant la somme des puissances de 2 associées à un état. Par exemple, pour la partition $[0; 2; 3]$ on aura $2^0 + 2^2 + 2^3 = 11 = 1011_2$: l'état correspondant de l'automate déterministe sera donc le numéro 11.

- D1. Écrire une fonction de signature `get_partition_number_from_list : int list -> int` qui renvoie le numéro associé à un élément de $\mathcal{P}(Q)$, c'est à dire l'état de l'automate déterministe associé à un ensemble d'états de l'automate non déterministe. On utilisera la fonction `lsl` qui permet de calculer rapidement une puissance de 2. Par exemple, `1 lsl 3` calcule 2^3 .
- D2. Écrire une fonction de signature `successors : ndfsm -> int -> char -> int list` qui renvoie les états suivants possibles. L'automate est dans un certain état (`int`) et il reçoit une lettre (`char`), le tout est passé en paramètres.
- D3. Écrire une fonction de signature
`successor_part : ndfsm -> int list -> char -> int list * int`
 qui renvoie l'état suivant de l'automate déterministe ainsi que le numéro associé à cet état. Les paramètres sont l'automate, l'état courant (`int list`) et la lettre reçue.
 Par exemple, l'appel `successor_part automata [0;1] 'a';;` renvoie
`- : int list * int = ([0; 1; 2], 7)` sur l'automate considéré précédemment.
- D4. Écrire une fonction de signature `build_det_fsm : ndfsm -> ndfsm` qui renvoie l'automate déterministe associé à un automate non déterministe. Il s'agit de construire :
1. les états de l'automate déterministe associé,
 2. les transitions de cet automate,
 3. d'en déduire l'état initial et les états accepteurs.

Pour la procédure, on utilisera une file d'attente (bibliothèque `Queue`) : cette file est initialisée avec l'état initiale de l'automate déterministe. À chaque itération, on défile (`pop`) un élément et on enfile (`push`) les nouveaux états découverts. La procédure s'arrête lorsque la file est vide. On a alors trouvé tous les états de l'automate et toutes les transitions de l'automate déterministe.

Pour mémoriser les partitions déjà rencontrées, on utilisera une table de hachage de la bibliothèque `Hashtbl`. Les clefs de cette table seront les numéros associés aux états de l'automate déterministe et la valeur associée à une clef sera la liste des états de l'automate non déterministe associée à cette partie de Q .

Pour savoir si un état est un état accepteur, on pourra utiliser la fonction `land` qui calcule le ET bit à bit entre deux nombres entiers.

Pour l'automate non déterministe considéré à la section précédente, on obtient :

```

1 { states = [15; 13; 11; 7; 9; 3; 1];
2   alphabet = ['a'; 'b'];
3   initial = [1];
4   transitions = [(15, 'b', 15); (15, 'a', 15);
5                 (13, 'b', 13); (13, 'a', 11);
6                 (11, 'b', 15); (11, 'a', 15);
7                 (7, 'b', 11); (7, 'a', 7);
8                 (9, 'b', 13); (9, 'a', 11);
9                 (3, 'b', 11); (3, 'a', 7);
10                (1, 'b', 9); (1, 'a', 3)];
11  accepting = [15; 13; 7]}
```

- D5. Écrire une fonction qui permet de savoir si un mot est reconnu par l'automate déterministe ainsi généré.
- D6. Proposer un algorithme permettant de savoir si un automate est déterministe.