

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)
- ☞ déterminer 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?

Solution : Non, car ils possèdent tous au moins une transition multiple étiquetée par une même lettre. Par exemple, la lettre b pour les transitions $1 \rightarrow 1$ et $1 \rightarrow 2$ pour l'automate \mathcal{A}_1 .

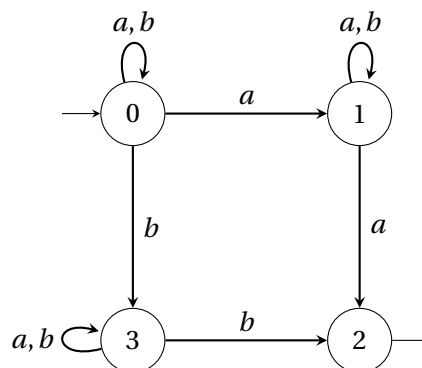
- A2. L'automate \mathcal{A}_1 reconnaît-il le mot $aabbb$? Construire l'arbre de l'exécution de cet automate pour ce mot.
- A3. Même question pour le mot $bbbaa$ et l'automate \mathcal{A}_2 .
- A4. Même question pour le mot $abab$ et l'automate \mathcal{A}_3 .
- A5. Décrire dans le langage naturel les langages reconnus par les automates \mathcal{A}_1 , \mathcal{A}_2 et \mathcal{A}_3 .

Solution :

- \mathcal{A}_1 est le langage des mots qui commencent par un a et finissent par un b .
- \mathcal{A}_2 est le langage des mots dont l'avant dernière lettre est un b .
- \mathcal{A}_3 est le langage des mots formés par le facteur ab .

B Déterminisation d'un AFND

Soit l'automate \mathcal{A} suivant :



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

Solution : Lorsqu'on se trouve en 0, 1 ou 3, il existe des transitions multiples associées à une seule lettre. Par exemple, de 0 lorsqu'on reçoit la lettre a , on peut aller en 0 ou en 1.

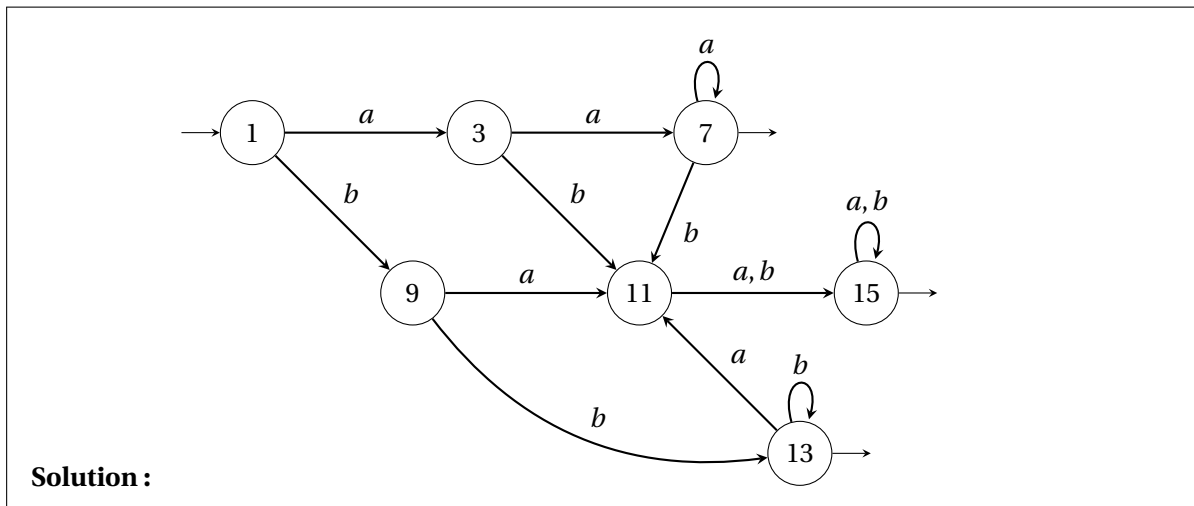
- B2. Quel est le langage reconnu par cet automate?

Solution : Le mots composés de a et de b contenant au moins deux a et finissant par a ou au moins deux b et finissant par b .

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

| Solution : | binaire | $\downarrow\{0\}$ | $\{0,1\}$ | $\uparrow\{0,1,2\}$ | $\{0,1,3\}$ | $\uparrow\{0,1,2,3\}$ | $\{0,3\}$ | $\{0,2,3\}$ |
|------------|---------|-------------------|-------------|---------------------|---------------|-----------------------|-------------|---------------|
| | | $\downarrow 1$ | 3 | $\uparrow 7$ | 11 | $\uparrow 15$ | 9 | $\uparrow 13$ |
| <i>a</i> | | $\{0,1\}$ | $\{0,1,2\}$ | $\{0,1,2\}$ | $\{0,1,2,3\}$ | $\{0,1,2,3\}$ | $\{0,1,3\}$ | $\{0,1,3\}$ |
| <i>b</i> | | $\{0,3\}$ | $\{0,1,3\}$ | $\{0,1,3\}$ | $\{0,1,2,3\}$ | $\{0,1,2,3\}$ | $\{0,2,3\}$ | $\{0,2,3\}$ |

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.

```
type ndfsm = { states : int list;
               alphabet : char list;
               initial : int list;
               transitions : (int * char * int) list;
               accepting : int list};;
```

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

Solution :

```
let sigma = ['a'; 'b'];;
let states = [0; 1; 2; 3];;
let init = [0];;
let final = [2];;
let trans = [ (0,'a',0); (0,'b',0); (0,'a',1); (0,'b',3);
              (1,'a',1); (1,'b',1); (1,'a',2);
              (3,'a',3); (3,'b',3); (3,'b',2)];;
```

```

let automata = {
  states = states;
  alphabet = sigma;
  initial = init;
  transitions = trans;
  accepting = final};;

```

D Codage de l'algorithme de détermination

On souhaite implémenter l'algorithme de détermination 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 .

Solution : Deux versions, avec ou sans folding.

```

let get_partition_number_from_list qset =
  List.fold_left (fun acc q -> acc + (1 lsl q)) 0 qset;;

let rec get_partition_number_from_list qset =
  match qset with
  | [] -> 0
  | q::t -> (1 lsl q) + (get_partition_number_from_list t);;

```

- 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.

Solution : Deux version, avec ou sans folding.

```

let successors a state letter =
  List.fold_left (fun acc (p,l,n) -> if p = state && l = letter then (n::acc) else acc ) [] a.transitions;;

let successors a state letter =
  let rec aux trans succ =
    match trans with

```

```

        | [] -> succ
        | (s,l,q)::t when s = state && l = letter -> aux t (q::succ)
        | _::t -> aux t succ in
aux a.transitions [];;

```

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.

Solution :

```

let rm_dup s = List.fold_left (fun acc x -> if List.mem x acc then acc else
  x :: acc) [] s;;

let rm_dup s =
  let rec aux sleft acc =
    match sleft with
    | [] -> acc
    | x::t -> if List.mem x acc then aux t acc else aux t (x :: acc)
  in aux s [];;

let successor_part a qset letter =
  let part = rm_dup (List.flatten (List.fold_left (fun acc state -> (
    successors a state letter)::acc) [] qset)) in
  (part, get_partition_number_from_list part);;

(* Alternativement on pourrait utiliser une table de hachage... puis
   convertir celle-ci en liste*)

```

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 :

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

Solution :

```
let build_det_fsm a =
  let n = List.length a.states in
  let partitions = Hashtbl.create (1 lsl n) in
  let states = ref [] in
  let transitions = ref [] in
  let init_part_number = get_partition_number_from_list a.initial in
  Hashtbl.add partitions init_part_number a.initial;
  states := (init_part_number)::!states;
  let queue = Queue.create () in
  Queue.push init_part_number queue;
  let update q letter =
    let (part, part_number) = successor_part a (Hashtbl.find partitions
      q) letter in
    if not (Hashtbl.mem partitions part_number)
    then
      begin
        Hashtbl.add partitions part_number part;
        states := (part_number)::!states;
        Queue.push part_number queue;
        transitions := ((q,letter,part_number)::!transitions);
      end
    else transitions := ((q,letter,part_number)::!transitions)
  in while (Queue.length queue) > 0 do
    let q = Queue.pop queue in
    List.iter (update q) a.alphabet;
  done;
  let is_final_part a part_number =
    let f_numbers = get_partition_number_from_list a.accepting in
    (f_numbers land part_number) != 0 in
  { states = !states;
    alphabet = a.alphabet;
    initial = [init_part_number];
    transitions = !transitions;
    accepting = (List.filter (is_final_part a) !states)};;

build_det_fsm automata;;
```

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

Solution :

```
let up_to a word =
  let rec aux q size =
    match size with
    | k when k = String.length word -> q (* done *)
    | k -> let t = List.find_opt (fun (p,l,n) -> q = p && word.[k] =
      l) a.transitions in
      match t with
      | None -> failwith "Undefined transition !"
      (* cas où l'automate n'est pas complet *)
      | Some( (_,_,nq)) -> aux nq (k+1)
  in aux (List.hd a.initial) 0;;

let match_word a word =
  try let final_state = up_to a word in
    List.mem final_state a.accepting with
  | Failure "Undefined transition !" -> false;;
```

D6. Proposer un algorithme permettant de savoir si un automate est déterministe.

Solution : Il suffit de balayer les transitions sortantes pour une même lettre : si, pour un état donné, il existe une lettre pour laquelle il y a plusieurs transitions différentes conduisant à différents états, alors l'automate n'est pas déterministe. Par ailleurs, s'il contient une transition spontanée, il n'est plus déterministe non plus.