Automates finis déterministes

OPTION INFORMATIQUE - TP n° 3.9 - Olivier Reynet

À la fin de ce chapitre, je sais :

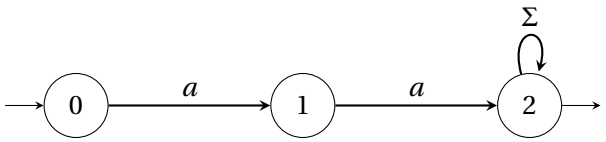
- ☞ définir un automate fini déterministe
- ☞ représenter un automate fini déterministe
- ☞ qualifier les états d'un automates (accessibilité)
- ☞ compléter un AFD
- ☞ complémenter un AFD
- ☞ faire le produit de deux AFD

A Construction d'automates simples

A1. On considère l'alphabet $\Sigma = \{a, b, c\}$. Construire les automates suivants :

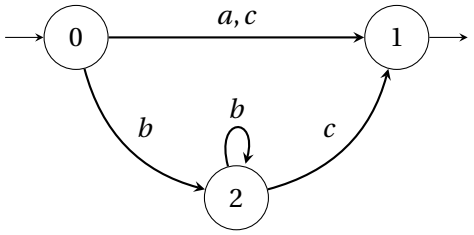
(a) $\mathcal{A}_0$ reconnaissant les mots commençant par deux occurrences de a.

Solution :

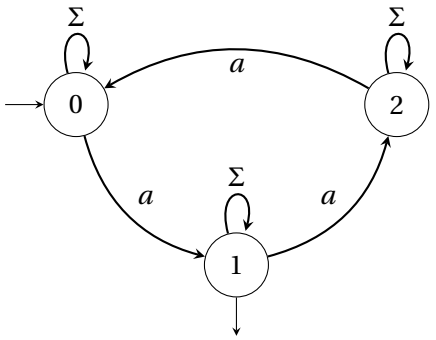


(b) $\mathcal{A}_1$ reconnaissant le langage défini par l'expression rationnelle $a|b^*c$.

Solution :

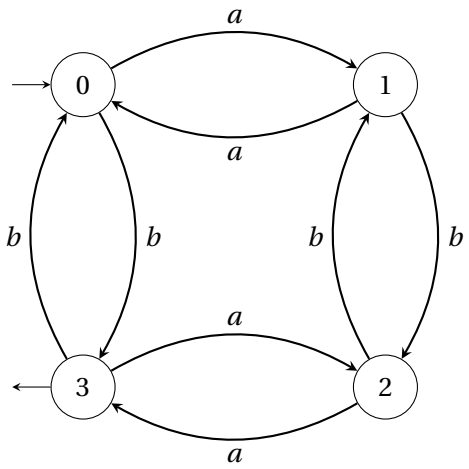


(c) $\mathcal{A}_2$ reconnaissant les mots contenant un nombre de a égal à 1 modulo 3, sans contrainte sur les autres lettres.



Solution :

(d) $\mathcal{A}_3$ reconnaissant les mots contenant un nombre pair de a et un nombre impair de b, c'est à dire $\{b, baa, aab, bbb, bababbb, aabbb, ababaab, \ldots\}$. On suppose que l'alphabet est $\Sigma = \{a, b\}$.



Solution :

A2. Donner les représentations tabulaires des automates $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$.

A3. Les automates que vous avez dessinés sont-ils complets ?

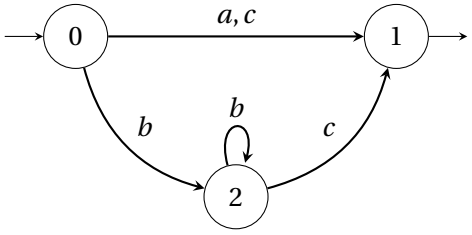
A4. Combien d'automates complets différents à n états peut-on construire ? On cherchera à exprimer la réponse en fonction de n et $|\Sigma|$

Solution : $n \times n^{n\times|\Sigma|} \times 2^n$

1. on a n états initiaux possibles
2. le nombre de fonctions de transition possibles est $|Q|^{|Q\times\Sigma|} = |Q|^{|Q|\times|\Sigma|} = n^{n\times|\Sigma|}$
3. le nombre d'ensembles possibles des états accepteurs, le cardinal des parties de Q : $|\mathcal{P}(Q)| = 2^n$

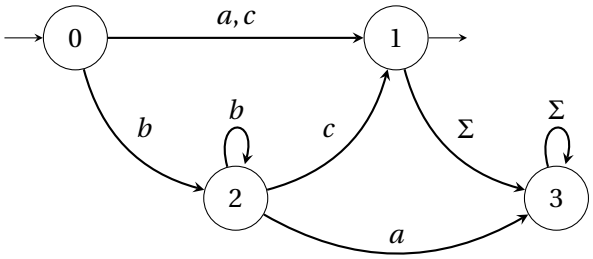
Pour un alphabet à deux lettres et deux états, on a $2 \times 2^4 \times 2^2 = 2^7 = 128$ automates possibles.

B Complété et complémentaire d'automates finis déterministes



B1. Compléter l'automate fini déterministe $\mathcal{A}$ suivant.

Solution :

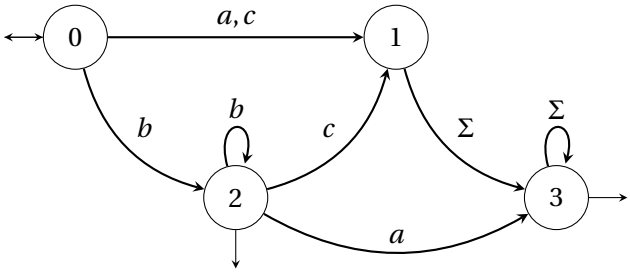


B2. Quels sont les états co-accessibles de l'automate complété de $\mathcal{A}$?

Solution : 0,1,2

B3. Dessiner le complémentaire de l'automate $\mathcal{A}$ précédent.

Solution :



B4. Le complémentaire de l'automate $\mathcal{A}$ reconnaît-il le mot vide ϵ ?

Solution : Oui, car l'état initial est également un état accepteur.

C Modélisation d'un automate en OCaml

On choisit de modéliser un automate fini 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 l'état initial (unique) et les états accepteurs forment une `int list`. La fonction de transition est une fonction. Cette solution d'implémentation présente l'avantage de coller au plus prêt à la définition mathématique d'un automate.

```
type fsm = { states : int list;
             alphabet : char list;
             initial : int;
             trans_f : int -> char -> int option;
             accepting : int list};;
```

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

Solution :

```
let sigma  = ['a'; 'b'; 'c'];;
let states = [0; 1; 2];;

let delta prev letter =
    match (prev, letter) with
        | (0,'a') -> Some 1
        | (0,'c') -> Some 1
        | (0,'b') -> Some 2
        | (2,'b') -> Some 2
        | (2,'c') -> Some 1
        | _ -> None;;

let final_states = [1];;


let automata = {states = states;
                alphabet = sigma;
                initial = 0;
                trans_f = delta;
                accepting = final_states};;
```

C2. Comment peut-on calculer l'état suivant lorsque l'automate `automata` est dans l'état `2` et qu'il lit la lettre `'c'` ?

Solution :

```
automata.trans_f 2 'c';;
```

C3. Comment peut-on savoir si un état 2 est accepteur ?

Solution :

```
List.mem 2 automata.accepting;;
```

D Calcul d'un mot par un automate

D1. Créer une fonction de signature `up_to : fsm -> string -> int` qui renvoie l'état d'arrivée dans lequel se trouve un automate à qui on a donné un mot reconnaître.

Solution :

```
let up_to a word =
    let rec aux q size =
        match size with
            | k when k = String.length word -> q  (* done *)
            | k -> let next = a.trans_f q word.[k] in
                           match next with
                               | None -> q
                               | Some(nq) -> aux nq (k+1)
    in aux a.initial 0;;

(* string to char list *)
let explode s = List.init (String.length s) (String.get s);;
let l = explode("ma jolie chaine");;

(* via char list *)
let up_to a word =
    let w = explode word in
    let rec aux state ch = match ch with
        | [] -> state
        | c::t -> let next = a.trans_f state c in
                           match next with
                               | None -> state
                               | Some(nq) -> aux nq t
    in aux 0 w;;
```

D2. Créer une fonction de signature `match_word : fsm -> string -> bool` qui renvoie vrai si un mot est reconnu par l'automate, faux sinon. Se servir de la fonction précédente et de la fonction `List.mem`.

Solution :

```
let match_word a word =
    let final_state = up_to a word in
        List.mem final_state a.accepting;;
```

D3. Tester l'automate sur plusieurs mots dans ou en dehors du langage. Que constatez-vous ?

Solution : Certains sont reconnus alors qu'ils ne devraient pas : l'automate est resté bloqué dans le dernier état.

```
match_word automata "a";;
match_word automata "c";;
match_word automata "bc";;
match_word automata "bbbc";;
match_word automata "bbbcc";; (* should not !*)
match_word automata "bac";;
match_word automata "bca";;  (* should not !*)
match_word automata "cb";;  (* should not !*)
match_word automata "bca";; (* should not !*)
```

```
match_word automata "";;
```

E Algorithmes de transformation simple d'un automate

E1. Créer une fonction de signature `is_complete : fsm -> bool` qui teste la complétude d'un automate. Dès qu'un état est détecté comme incomplet, c'est à dire il manque une transition pour une certaine lettre de l'alphabet, cette fonction renvoie `false`.

Solution :

```
let is_complete a =
    let check_letter q letter =
        match  a.trans_f q letter with
            | None -> false
            | Some(_) -> true in
    let rec check_state q alpha =
        match alpha with
            | [] -> true
            | h::t -> if check_letter q h then check_state q t else false in
    let rec check s =
        match s with
            | [] -> true
            | q::t -> if check_state q a.alphabet then check t else false
    in check a.states;;

is_complete automata ;;
```

E2. Créer une fonction de signature `complete : fsm -> fsm` qui crée l'automate complété d'un automate incomplet.

Solution :

```
let complete a =
    if not (is_complete a)
    then
        begin
            let well = List.length states in (* le puits ! *)
            let new_trans_f prev letter =  (* fonction de transition *)
                match a.trans_f prev letter with
                            | None -> Some(well) (* dans le puits *)
                            | Some(next) -> Some(next) in
            {states = (well::a.states);
             alphabet = a.alphabet;
             initial = a.initial;
             trans_f = new_trans_f;
             accepting = a.accepting}
        end
    else a;;
```

```
is_complete (complete automata);;
```

E3. Créer l'automate complété de $\mathcal{A}$ et le tester sur des mots dans et en dehors du langage.

Solution :

```
let cautomata = (complete automata);;
match_word cautomata "a";;
match_word cautomata "c";;
match_word cautomata "bc";;
match_word cautomata "bbbc";;
match_word cautomata "bbbcc";; (* yes, better !*)
match_word cautomata "bac";;
match_word cautomata "bca";;  (* yes !*)
match_word cautomata "cb";;  (* yes !*)
match_word cautomata "bca";; (* yes !*)
match_word cautomata "";;
```

E4. Créer une fonction de signature `complement : fsm -> fsm` qui crée l'automate complémentaire d'un automate.

Solution :

```
let complement a =
    {states = a.states;
    alphabet = a.alphabet;
    initial = a.initial;
    trans_f = a.trans_f;
    accepting = List.filter (fun q -> not (List.mem q a.accepting)) a.states
        };;

automata;;
complement(automata);;
complement(complete automata);;
```

F États accessibles et co-accessibles d'un automate

Pour déterminer les états accessibles ou co-accessibles d'un automate, on utilise les algorithmes des graphes sur le graphe orienté que constitue l'automate.

F1. Créer une fonction de signature `succ : fsm -> int -> int list` qui renvoie la liste des successeurs d'un état d'un automate passé en paramètre.

Solution :

```
let succ a q =
    let rec aux successors alpha =
```

```
            match alpha with
                | [] -> successors
                | letter::t -> match a.trans_f q letter with
                | None -> aux successors t
                | Some(next) -> if List.mem next successors
                                            then aux successors t
                                            else aux (next::successors) t
        in aux [] a.alphabet;;
```

F2. Créer une fonction de signature `fsm_to_graph : fsm -> int list array` qui crée le graphe d'un automate sous la forme d'une liste d'adjacence. On utilisera la fonction précédente et la fonction `Array.of_list` pour transformer la liste résultat `int list list` en `int list array`. On pourra ainsi pouvoir réutiliser les codes sur les graphes de nos TP précédents.

Solution :

```
let fsm_to_graph a =
    let rec aux g s =
        match s with
            | [] -> g
            | q::t -> aux ((succ a q)::g) t
    in Array.of_list(aux [] a.states);;

fsm_to_graph automata;;
```

F3. En effectuant un parcours en largeur d'un graphe, créer une fonction de signature `accessible_states : fsm -> int list` qui renvoie la liste des états accessibles.

Solution :

```
let accessible_states a =
    let g = fsm_to_graph a in
    let visited = Array.make (Array.length  g) false in
    let rec explore queue =
        match queue with
            | [] -> []
            | v::t when visited.(v) -> explore t
            | v::t -> visited.(v) <- true; v::(explore (t @ g.(v)))
    in explore [a.initial] ;;

accessible_states (complement (complete automata));;
```

F4. Créer une fonction de signature `switch_direction : int list array -> int list array` qui transforme un graphe en un autre graphe en inversant le sens de tous ses arcs. On utilisera une boucle `for` et la fonction `List.iter` qu nécessite une fonction renvoyant `unit`.

Solution :

```
let switch_direction g =
    let n = Array.length g in
        let sg = Array.make n [] in
        for i = 0 to (n - 1) do
            List.iter (fun v -> sg.(v) <- i::sg.(v)) g.(i)
        done;
    sg;;

fsm_to_graph automata;;
switch_direction (fsm_to_graph automata);;
```

F5. Créer une fonction de signature `coaccessible_states : fsm -> int list` qui renvoie la liste des états co-accessibles. On s'appuiera sur un parcours en largeur en partant des états accepteurs et la fonction précédente.

Solution :

```
let coaccessible_states a =
    let g = switch_direction (fsm_to_graph a) in
        let visited = Array.make (Array.length  g) false in
            let rec explore queue =
                match queue with
                    | [] -> []
                    | v::t when visited.(v) -> explore t
                    | v::t -> visited.(v) <- true; v::(explore (t @ g.(v)))
    in explore a.accepting;;

coaccessible_states (complete automata);;
```

G Problème de la finitude d'un langage reconnaissable

Le problème de la finitude d'un langage est décidable pour les AFD. L'idée fondamentale est la suivante : le langage accepté sera infini si et seulement s'il existe un circuit accessible et co-accessible, c'est à dire dans lequel on peut arriver et duquel on peut sortir vers un état accepteur.

G1. Proposer un algorithme pour détecter les cycles dans un graphe orienté. La fonction créée renverra faux si un cycle est détecté.

Solution : Un parcours en profondeur qui renvoie faux dès qu'il repasse sur un sommet.

```
let dfs g v0 =
   let visited = Array.make (Array.length g) false in
       let rec explore stack =
           match stack with
               | [] -> true
               | v::_ when visited.(v) -> false
               | v::t -> visited.(v) <- true; explore (g.(v) @ t)
```

```
    in explore [v0] ;;
```

G2. Pour un automate fini déterministe donné, créer une fonction de signature `is_finite_language : fsm -> bool` qui permet de savoir si le langage reconnu par un automate est fini.

Solution :

```
let is_finite_language a =
    dfs (fsm_to_graph a) a.initial;;

is_finite_language automata;;
```

G3. Modifier l'automate $\mathcal{A}$ pour que le langage reconnu soit fini.

Solution :

```
let fdelta prev letter =
    match (prev, letter) with
        | (0,'a') -> Some 1
        | (0,'c') -> Some 1
        | (0,'b') -> Some 2
        | (2,'c') -> Some 1
        | _ -> None;;

let fautomata = {states = states;
                 alphabet = sigma;
                 initial = 0;
                 trans_f = fdelta;
                 accepting = final_states};;

is_finite_language fautomata;;
```