

# 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
- ☞ compléter 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  $a$ .
  - (b)  $\mathcal{A}_1$  reconnaissant le langage défini par l'expression rationnelle  $a|b^*c$ .
  - (c)  $\mathcal{A}_2$  reconnaissant les mots contenant un nombre de  $a$  égal à 1 modulo 3, sans contrainte sur les autres lettres.
  - (d)  $\mathcal{A}_3$  reconnaissant les mots contenant un nombre pair de  $a$  et un nombre impair de  $b$ , c'est à dire  $\{b, baa, aab, bbb, bababbb, aabbb, ababaab, \dots\}$ . On suppose que l'alphabet est  $\Sigma = \{a, b\}$ .
- 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  $|\Sigma|$

## B Complété et complémentaire d'automates finis déterministes



- B1. Compléter l'automate fini déterministe  $\mathcal{A}$  suivant.
- B2. Quels sont les états co-accessibles de l'automate complété de  $\mathcal{A}$ ?
- B3. Dessiner le complémentaire de l'automate  $\mathcal{A}$  précédent.
- B4. Le complémentaire de l'automate  $\mathcal{A}$  reconnaît-il le mot vide  $\epsilon$ ?

## 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 près à la définition mathématique d'un automate.

```
1 type fsm = { states : int list;  
2             alphabet : char list;  
3             initial : int;  
4             trans_f : int -> char -> int option;  
5             accepting : int list};;
```

---

- C1. Créer une variable `automata` qui représente l'automate  $\mathcal{A}$  de la section B.
- C2. Comment peut-on calculer l'état suivant lorsque l'automate `automata` est dans l'état 2 et qu'il lit la lettre 'c' ?

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

## 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`.
- E2. Créer une fonction de signature `complete : fsm -> fsm` qui crée l'automate complété d'un automate incomplet.
- E3. Créer l'automate complété de  $\mathcal{A}$  et le tester sur des mots dans et en dehors du langage.
- E4. Créer une fonction de signature `complement : fsm -> fsm` qui crée l'automate complémentaire d'un automate.

## 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.

- 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.
- 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.
- 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` qui nécessite une fonction renvoyant `unit`.
- 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.

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