

INF 302 : LANGAGES & AUTOMATES

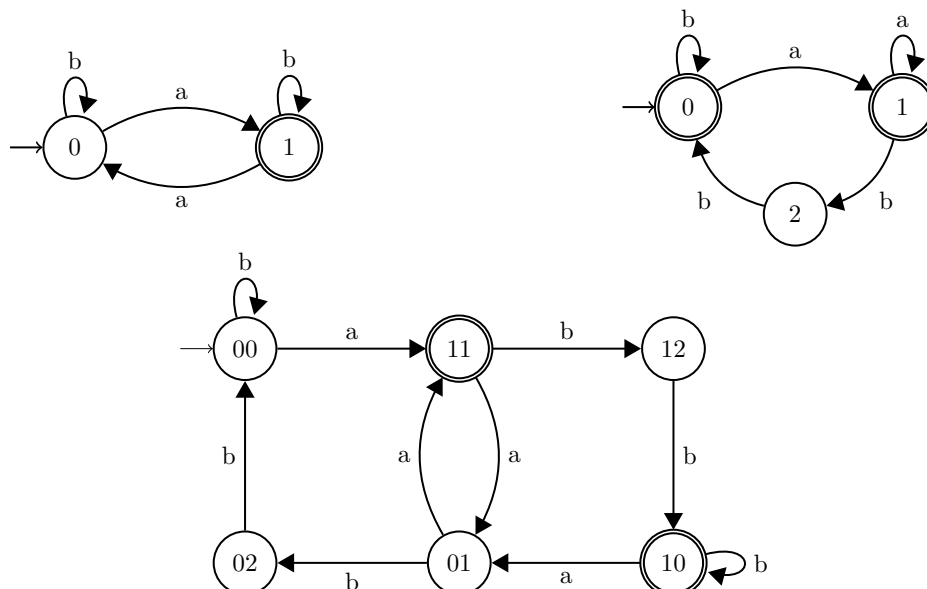
Chapitre 5 : Algorithmes et problèmes de décision sur les AD

Yliès Falcone

ylies.falcone@univ-grenoble-alpes.fr — www.ylies.fr

Univ. Grenoble-Alpes, Inria
Laboratoire d'Informatique de Grenoble - www.liglab.fr

Intuition et objectifs



- Parcours d'un automate :
 - en profondeur,
 - en largeur.
- Détection de cycles.
- Notions d'accessibilité et de co-accessibilité.
- Problèmes de décision : langage vide, langage infini.

Notations

Dans la suite, nous utilisons un AD $(Q, \Sigma, q_{\text{init}}, \delta, F)$.

Successeurs d'un état

L'ensemble des états **successeurs** d'un état $q \in Q$, selon la fonction de transition δ , est :

$$\text{Succ}(q) = \{q' \in Q \mid \exists a \in \Sigma : \delta(q, a) = q'\}$$

En itérant sur $\text{Succ}(q)$, nous obtenons les états selon la priorité donnée par les symboles.

Prédécesseurs d'un état

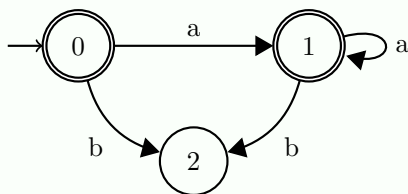
L'ensemble des états **prédécesseurs** d'un état $q \in Q$, selon la fonction de transition δ , est :

$$\text{Pré}(q) = \{q' \in Q \mid \exists a \in \Sigma : \delta(q', a) = q\}$$

En itérant sur $\text{Pré}(q)$, nous obtenons les états selon la priorité donnée par les symboles.

Remarque Un état q est successeur d'un état q' ssi q' est un prédécesseur de q □

Exemple (Successeurs et prédécesseurs)



- successeurs de 0 : 1 et 2
- prédécesseurs de 0 : aucun
- successeurs de 1 : 1 et 2
- prédécesseurs de 1 : 0 et 1
- successeurs de 2 : aucun
- prédécesseurs de 2 : 0 et 1

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

- 1 Parcours
- 2 Notions d'accessibilité et de co-accessibilité
- 3 Détection de cycles
- 4 Quelques problèmes de décision
- 5 Résumé

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

- 1 Parcours
- 2 Notions d'accessibilité et de co-accessibilité
- 3 Détection de cycles
- 4 Quelques problèmes de décision
- 5 Résumé

À propos du parcours d'un automate

Objectifs

- Visiter de manière *ordonnée* une seule fois chacun des états de l'automate (implémentation de « pour tout état q dans Q ... » selon un ordre).
- Définir une « brique de base » pour d'autres algorithmes sur les automates.

Principe

- suivi des transitions de l'automate,
- utilisation des successeurs d'un état.

Comment gérer les cycles ?

Quelques exemples d'applications

- détection de cycles,
- recherche de chemins (p. ex. : entre états, labyrinthe),
- collecte d'informations,
- tri topologique (p. ex. : ordonnancement, compilation).

À propos du parcours d'un automate

Algorithme paramétré par :

- une « opération de visite » des états de l'automate (où le traitement se fait) ;
- la « priorité » donnée entre les successeurs d'un état ;
- un « ordre » de parcours, influençant quels noeuds et quelles parties de l'automate sont visités en premières : **profondeur** ou **largeur** ;
- un ensemble d'états de départ.

Le choix largeur vs profondeur influence le choix des successeurs visités en premier :

- parcours en **largeur** : on visite *tous les successeurs* avant de visiter les successeurs des successeurs ;
- parcours en **profondeur** : on visite *les successeurs du successeur* avant de visiter les autres successeurs.

◁ Largeur vs profondeur

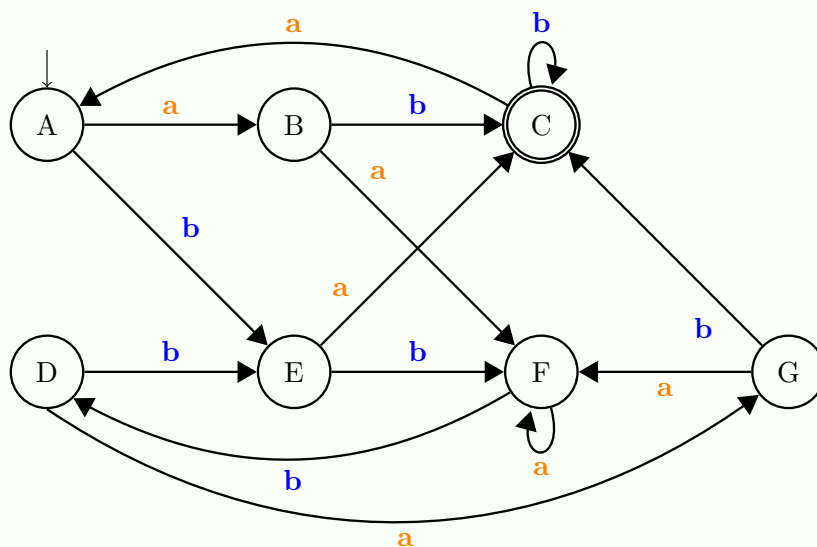
◁ Parcours largeur vs profondeur

◁ Structure de données pour largeur vs profondeur

Parcours en profondeur vs largeur depuis l'état initial

Exemple (Parcours en profondeur vs largeur)

- L'opération de **visite** consiste à afficher l'état.
- L'ensemble d'états de départ est réduit à l'état initial.



Priorité de **a** sur **b**

- prof. : A, B, F, D, G, C, E
- largeur : A, B, E, F, C, D, G

Priorité de **b** sur **a**

- prof. : A, E, F, D, G, C, B
- largeur : A, E, B, F, C, D, G

Parcours générique (sans ordre) d'un AD : version *itérative*

Algorithme 1 *parcourir_it()* pour le parcours itératif d'un automate

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

- 1: **ens d'états** $\tilde{A}_{\text{visiter}} :=$ **ensemble d'états de départ** ;
- 2: **ens d'états** $\text{Déjà_visités} := \emptyset$; (* initialement, rien n'est visité *)
- 3: **tant que** $\tilde{A}_{\text{visiter}} \neq \emptyset$ **faire** (* tant qu'il reste des états à visiter *)
- 4: **soit** $q \in \tilde{A}_{\text{visiter}}$; (* prendre un état à visiter *)
- 5: $\tilde{A}_{\text{visiter}} := \tilde{A}_{\text{visiter}} \setminus \{q\}$; (* l'état q n'est plus à visiter *)
- 6: **Visiter l'état** q ; (* À définir en fonction de l'objectif de algorithme ; par exemple, afficher. *)
- 7: $\text{Déjà_visités} := \text{Déjà_visités} \cup \{q\}$; (* l'état q vient d'être visité *)
- 8: $\tilde{A}_{\text{visiter}} := \tilde{A}_{\text{visiter}} \cup (\text{Succ}(q) \setminus \text{Déjà_visités})$; (* les nouveaux états à visiter sont les successeurs de q non visités *)
- 9: **fin tant que**

- En utilisant un **ensemble** comme structure de données pour $\tilde{A}_{\text{visiter}}$, l'ordre de parcours n'est pas déterminé.
- L'ordre de parcours dépend de l'insertion et la sélection des états dans $\tilde{A}_{\text{visiter}}$ (lignes 4 et 8).
(Les éléments d'un ensemble ne sont pas ordonnés.)

Parcours en profondeur d'un AD : version *itérative*

L'ensemble d'états est remplacé par une **pile d'états** (dernier entré, premier sorti) :

- **empiler()** : ajoute des éléments en sommet de pile ;
 - **dépiler()** : retourne et supprime le sommet de pile ;
 - **est_vide()** : indique si la pile est vide.
- Notation :
pile.**opération()**

Algorithme 2 *parcourir_it_prof()* pour le parcours itératif **en profondeur** d'un automate

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

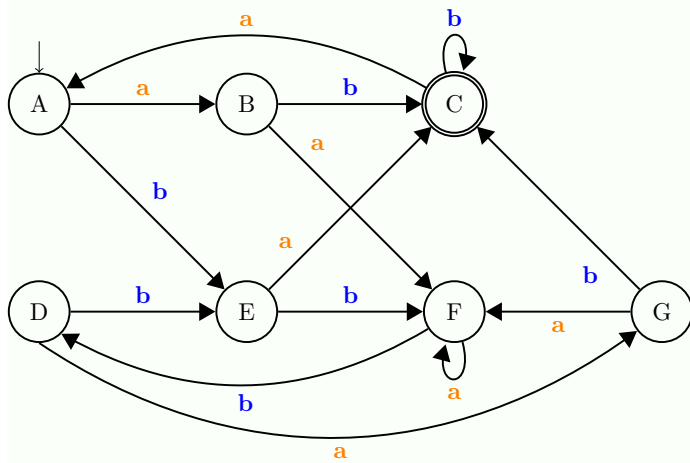
- 1: **pile d'états** $\tilde{A}_{\text{visiter}} :=$ **ensemble d'états de départ** ;
- 2: **ens d'états** $\text{Déjà_visités} := \emptyset$; (* initialement, rien n'est visité *)
- 3: **tant que** $\neg \tilde{A}_{\text{visiter}}.\text{est_vide}()$ **faire** (* tant qu'il reste des états à visiter *)
- 4: **état** $q := \tilde{A}_{\text{visiter}}.\text{dépiler}()$; (* prendre un état à visiter *)
- 5: **si** $q \notin \text{Déjà_visités}$ **alors**
- 6: **Visiter l'état** q ; (* À définir en fonction de l'objectif de algorithme ; p. ex., afficher. *)
- 7: $\text{Déjà_visités} := \text{Déjà_visités} \cup \{q\}$; (* l'état q vient d'être visité *)
- 8: $\tilde{A}_{\text{visiter}}.\text{empiler}(\text{Succ}(q) \setminus \text{Déjà_visités})$; (* les nouveaux états à visiter sont les successeurs de q non visités *)
- 9: **fin si**
- 10: **fin tant que**

L'empilement des successeurs se fait dans *l'ordre inverse* de la priorité entre successeurs.

Remarque La structure de données pour Déjà_visités n'importe pas. □

Parcours en profondeur d'un AD : version *itérative*

Exemple (Parcours itératif en profondeur d'un AD)



À_visiter (pile)	Déjà_vistés (ensemble)	Sortie (en cours de visite)
A		
B, E	A	A
F, C, E	A, B	B
D, C, E	A, B, F	F
G, E, C, E	A, B, F	D
C, E, C, E	A, B, D, F, G	G
E, C, E	A, ..., D, F, G	C
C, E	A, ..., G	E
E	A, ..., G	–
	A, ..., G	–

- À_visiter est une *pile*, représentée avec le sommet de pile à gauche.
- Priorité de *a* sur *b* dans le choix des voisins.
 ↪ on empile le successeur sur *b* puis celui sur *a*.

Parcours en largeur d'un AD

L'ensemble d'états est remplacé par une *file d'états* (premier entré, premier sorti) :

- *enfiler()* : ajoute des éléments en début de file ;
 - *défiler()* : retourne et supprime le dernier élément de la file ;
 - *est_vide()* : indique si la file est vide.
- Notation :
file.*opération()*

Algorithme 3 *parcourir_it_larg()* pour le parcours itératif *en largeur* d'un automate

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

```

1: file d'états À_visiter := ensemble d'états de départ ;
2: ens d'états Déjà_vistés :=  $\emptyset$  ; (* initialement, rien n'est visité *)
3: tant que  $\neg$  À_visiter.est_vide() faire (* tant qu'il reste des états à visiter *)
4:   état  $q$  := À_visiter.défiler() ; (* prendre un état à visiter *)
5:   si  $q \notin$  Déjà_vistés alors
6:     Visiter l'état  $q$  ; (* À définir en fonction de l'objectif de algorithme ; p. ex., afficher. *)
7:     Déjà_vistés := Déjà_vistés  $\cup$   $\{q\}$  ; (* l'état  $q$  vient d'être visité *)
8:     À_visiter.enfiler(Succ( $q$ ) \ Déjà_vistés) ; (* les nouveaux états à visiter sont les successeurs de  $q$  non visités *)
9:   fin si
10: fin tant que

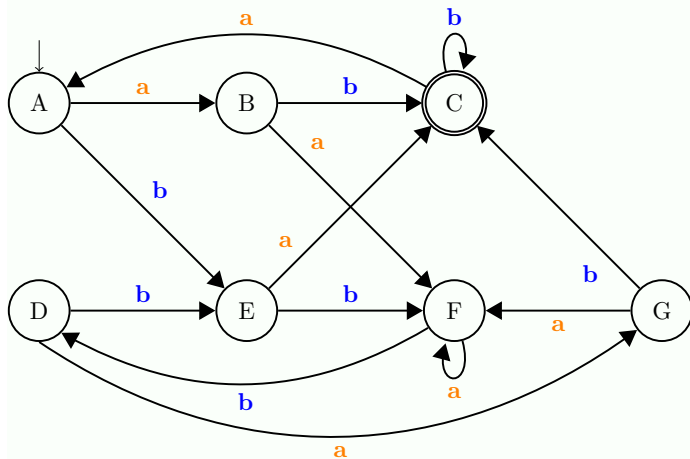
```

L'enfilage des successeurs se fait dans l'ordre de la priorité entre successeurs.

Remarque Pas de version récursive pour le parcours *en largeur*. □

Parcours en largeur d'un AD

Exemple (Parcours itératif en largeur d'un AD)



À_visiter (file)	Déjà_vistés (ensemble)	Sortie (en cours de visite)
A		
E, B	A	A
C, F, E	A, B	B
F, C, C, F	A, B, E	E
F, C, D, C	A, B, E, F	F
D, F, C, D	A, B, C, E, F	C
G, D, F, C	A, ..., F	D
G, D, F	A, ..., F	–
G, D	A, ..., F	–
G	A, ..., F	–
	A, ..., G	G

- À_visiter est une **file**, avec le début de file à gauche et la fin de file à droite.
- Priorité de **a** sur **b** dans le choix des voisins.
 ↪ on enfile le successeur sur **a** puis le successeur sur **b**.

Parcours en profondeur d'un AD : version *récursive*

Algorithme 4 *parcourir_prof()* pour le parcours en profondeur récursif d'un automate

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

- 1: **ens d'états** Déjà_vistés; (* variable globale aux deux algorithmes *)
- 2: Déjà_vistés := \emptyset ; (* initialement, rien n'est visité *)
- 3: **pour** chq état q dans l'ens. d'états de départ **faire** (* graphe possiblement déconnecté *)
- 4: **si** $q \notin$ Déjà_vistés **alors**
- 5: *parcourir_prof_rec*(q)
- 6: **fin si**
- 7: **fin pour**

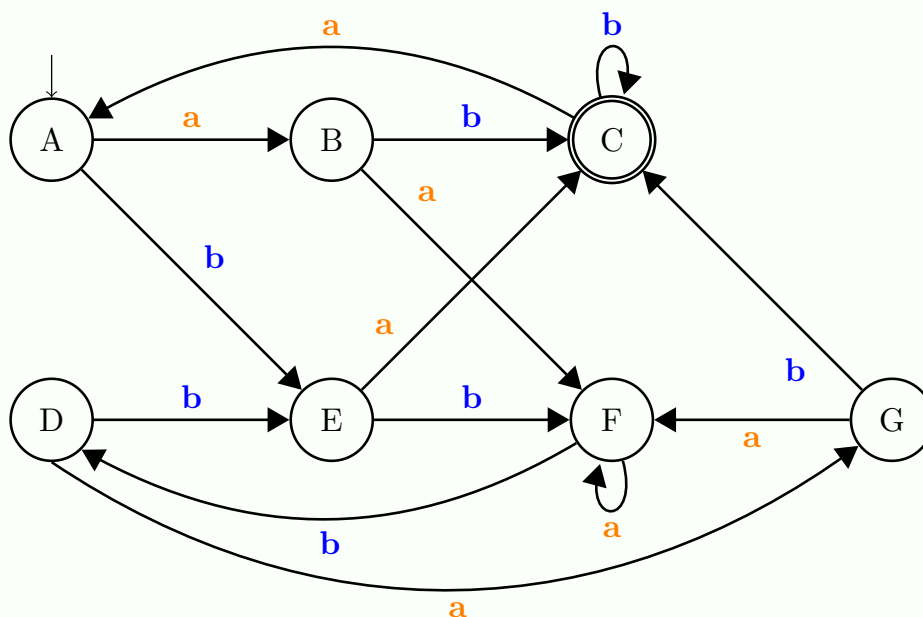
Algorithme 5 *parcourir_prof_rec()* pour le parcours en profondeur à partir d'un état

Entrée : $q \in Q$ (* un état *)

- 1: **visiter l'état** q ; (* À définir en fonction de l'objectif de algorithme ; par exemple, afficher. *)
- 2: Déjà_vistés := Déjà_vistés $\cup \{q\}$; (* l'état q n'est plus à visiter *)
- 3: **pour** chaque état $q' \in \text{Succ}(q)$ **faire** (* pour chaque successeur de q *)
- 4: **si** $q' \notin$ Déjà_vistés **alors**
- 5: *parcourir_prof_rec*(q')
- 6: **fin si**
- 7: **fin pour**

Parcours en profondeur d'un AD : version *récursive*

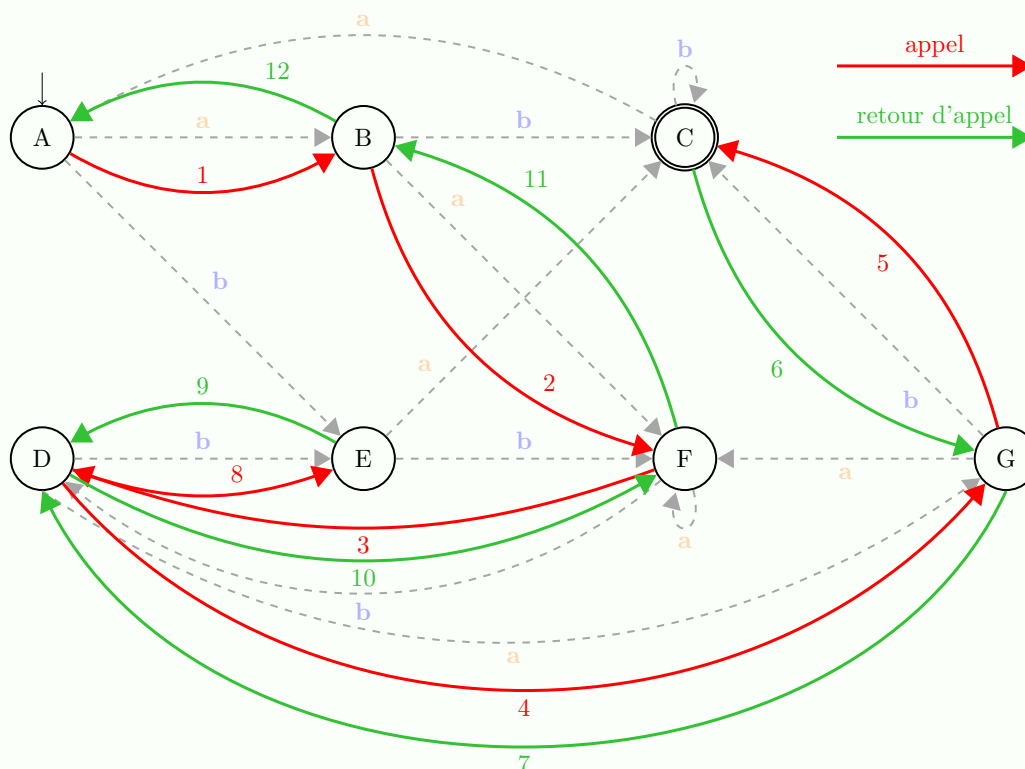
Exemple (Parcours récursif en profondeur d'un AD)



◁ Parcours récursif en profondeur

Parcours en profondeur d'un AD : version *récursive*

Exemple (Parcours récursif en profondeur d'un AD)



Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

- 1 Parcours
- 2 Notions d'accessibilité et de co-accessibilité
 - Accessibilité
 - Co-accessibilité
 - Vocabulaire et utilisation
- 3 Détection de cycles
- 4 Quelques problèmes de décision
- 5 Résumé

Accessibilité et co-accessibilité

Notions liées à la fonction de transition des ADs.

Accessibilité et co-accessibilité : définition informelle

Propriétés s'appliquant aux états d'un AD :

- état accessible : peut être atteint en suivant la fonction de transition ;
- état co-accessible : mène à un état accepteur en suivant la fonction de transition.

Nous utiliserons ces notions :

- pour répondre à des problèmes de décision sur les automates dans la section suivante ;
- caractériser certaines de leurs propriétés en TD ;
- dans les chapitres suivants.

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

1 Parcours

2 Notions d'accessibilité et de co-accessibilité

- Accessibilité
- Co-accessibilité
- Vocabulaire et utilisation

3 Détection de cycles

4 Quelques problèmes de décision

5 Résumé

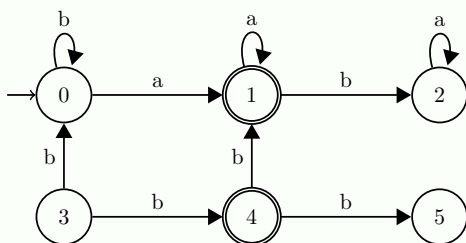
Accessibilité dans les AD : définition et décidabilité

Considérons un AD $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$.

Définition (Accessibilité d'un état dans un AD)

$q \in Q$ est accessible s'il existe un mot $u \in \Sigma^*$ tel que $\delta^*(q_{\text{init}}, u) = q$.

Exemple (États accessibles)



- accessibles : 0, 1, 2
- non accessibles : 3, 4, 5

Théorème : **accessibilité** dans les graphes finis

Le problème de l'*accessibilité* est *décidable* pour les graphes finis.

Corollaire

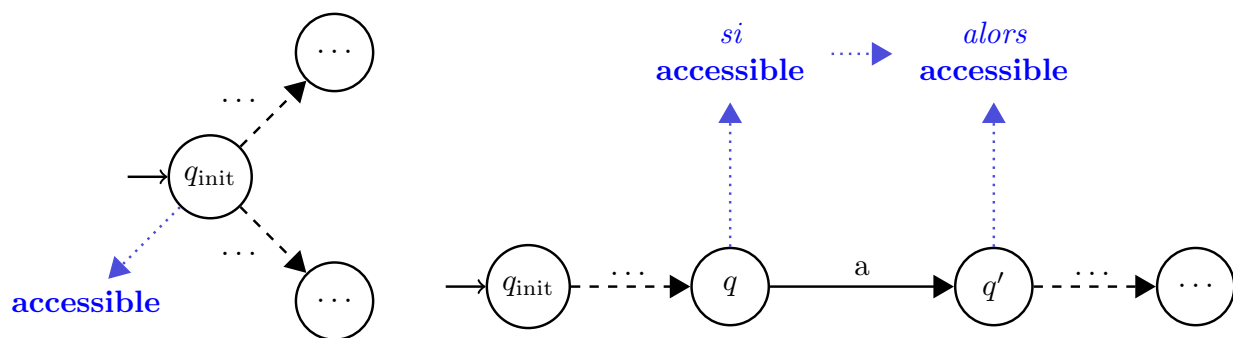
Le problème de l'*accessibilité* d'un état dans un AD est décidable.

Accessibilité dans les AD : intuition sur l'algorithme

Etant donné un AD $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$.

Idée de l'algorithme

- cas de base : q_{init} est accessible
- induction : si $q \in Q$ est accessible dans A , alors s'il existe une transition dans δ de q vers un état q' , alors q' est accessible dans A ($q' \in \text{Succ}(q)$).



Accessibilité dans les AD : algorithme itératif

Algorithme 6 *etats_accessibles()* pour le calcul des états accessibles

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

Sortie : $\text{Accessibles} \subseteq Q$ (* ensemble des états accessibles dans A par δ *)

```

1: ens d'états Accessibles,  $\tilde{A}_{\text{visiter}}$ , Déjà_vistés ;
2: Accessibles :=  $\{q_{\text{init}}\}$ ; (* cas de base *)
3:  $\tilde{A}_{\text{visiter}}$  :=  $\{q_{\text{init}}\}$ ; (* parcours depuis l'état initial *)
4: Déjà_vistés :=  $\emptyset$ ; (* initialement, rien n'est visité *)
5: tant que  $\tilde{A}_{\text{visiter}} \neq \emptyset$  faire
6:   soit  $q \in \tilde{A}_{\text{visiter}}$ ;
7:    $\tilde{A}_{\text{visiter}}$  :=  $\tilde{A}_{\text{visiter}} \setminus \{q\}$ ;
8:   Déjà_vistés := Déjà_vistés  $\cup \{q\}$ ;
9:   Accessibles := Accessibles  $\cup \text{Succ}(q)$ ;
      (* induction ; on ajoute les successeurs de  $q$  à l'ensemble des états accessibles *)
10:   $\tilde{A}_{\text{visiter}}$  :=  $\tilde{A}_{\text{visiter}} \cup (\text{Succ}(q) \setminus \text{Déjà_vistés})$ ;
      (* les nouveaux états à visiter sont ceux « découverts », cad les successeurs non visités *)
11: fin tant que
12: retourner Accessibles ;

```

Remarque Cet algorithme s'adapte pour calculer les états accessibles à partir d'un état ou d'un ensemble d'états donnés de l'automate (en modifiant les lignes 2 et 3). \square

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

1 Parcours

2 Notions d'accessibilité et de co-accessibilité

- Accessibilité
- Co-accessibilité
- Vocabulaire et utilisation

3 Détection de cycles

4 Quelques problèmes de décision

5 Résumé

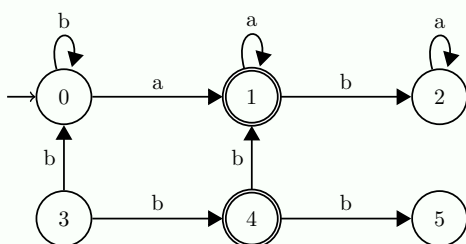
Co-accessibilité dans les AD : définition et décidabilité

Considérons un AD $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$.

Définition (Co-accessibilité d'un état dans un AD)

$q \in Q$ est co-accessible s'il existe un mot $u \in \Sigma^*$ tel que $\delta^*(q, u) \in F$.

Exemple (États co-accessibles)



- co-accessibles : 0, 1, 3, 4
- non co-accessibles : 2, 5

Théorème : co-accessibilité dans les graphes finis

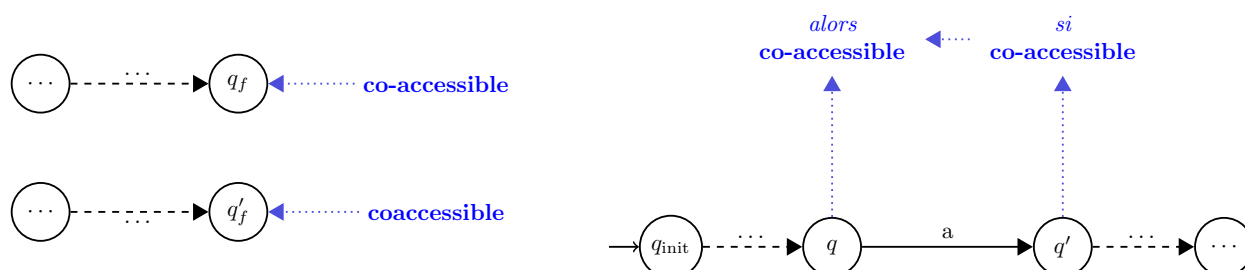
Le problème de la co-accessibilité est décidable pour les graphes finis.

Co-accessibilité dans les AD : intuition sur l'algorithme

Etant donné un AD $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$.

Définition (Idée de l'algorithme)

- cas de base : les états $q \in F$ sont co-accessibles,
- induction : si $q \in Q$ est co-accessible dans A , alors s'il existe une transition dans δ depuis un état q' vers q , alors q' est co-accessible dans A .



Remarque La co-accessibilité d'un état q peut aussi se calculer :

- avec l'accessibilité depuis q vers q_f , avec $q_f \in F$;
- avec l'accessibilité de q dans l'automate « miroir ».

□

Co-accessibilité dans les AD : algorithme

Algorithme 7 *etats_coaccessibles()* pour le calcul des états accessibles

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

Sortie : $\text{Coaccessibles} \subseteq Q$ (* ensemble des états accessibles dans A par δ *)

```

1: ens d'états Coaccessibles, A_visiter, Déjà_visés ;
2: Coaccessibles := F ; (* cas de base *)
3: A_visiter := F ; (* on veut les états co-accessibles à tous les états accepteurs *)
4: Déjà_visés := ∅ ;
5: tant que A_visiter ≠ ∅ faire
6:   soit  $q \in A\_visiter$  ;
7:    $A\_visiter := A\_visiter \setminus \{q\}$  ;
8:    $Déjà\_visés := Déjà\_visés \cup \{q\}$  ;
9:    $Coaccessibles := Coaccessibles \cup \text{Pré}(q)$ 
      (* induction ; on ajoute les prédécesseurs de  $q$  à l'ensemble des états co-accessibles *) ;
10:   $A\_visiter := A\_visiter \cup (\text{Pré}(q) \setminus Déjà\_visés)$  ;
      (* les nouveaux états à visiter sont ceux « découverts » *)
11: fin tant que
12: retourner Coaccessibles ;

```

Cet algorithme peut facilement être modifié pour calculer les états co-accessibles à ensemble d'états donnés de l'automate (en modifiant les lignes 2 et 3).

Co-accessibilité en réutilisant l'accessibilité

La co-accessibilité peut se résoudre en considérant un automate « miroir »

- en « inversant » la fonction de transition,
- en « partant des états accepteurs ».

puis en ré-utilisant l'algorithme d'accessibilité.

Algorithme 8 *etats_coaccessibles()* pour le calcul des états accessibles

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ un AD

Sortie : ensemble des états co-accessibles dans A par δ

- 1: $\delta' = \{(q', a, q) \mid (q, a, q') \in \delta\} \cup \{(q_{\text{new}}, a, q_f) \mid q_f \in F \wedge a \in \Sigma\}$;
 - 2: **automate** $E := (Q \cup \{q_{\text{new}}\}, \Sigma, \delta', q_{\text{new}}, q_{\text{init}})$;
 - 3: **retourner** $\text{etats_accessibles}(E) \setminus \{q_{\text{new}}\}$;
-

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

1 Parcours

2 Notions d'accessibilité et de co-accessibilité

- Accessibilité
- Co-accessibilité
- Vocabulaire et utilisation

3 Détection de cycles

4 Quelques problèmes de décision

5 Résumé

Vocabulaire

Un automate est dit *accessible* si tous ses états sont accessibles.

Un automate est dit *co-accessible* si tous ses états sont co-accessibles.

Un automate est dit **émondé** s'il est accessible et co-accessible.

Intuitivement, dans un automate émondé, tous les états sont « utiles » à la reconnaissance des mots.

Utilisations des parcours et de la detection de cycles

- Problèmes de décision (voir section suivante).
- Garder uniquement les états « utiles » d'un automate.
- Connaître et générer les états d'un automate dans les situations suivantes :
 - Lorsque les états ne sont pas donnés de manière explicite, mais par exemple sous forme de règles.
Exemple : automate de l'étrange planète vu dans le premier cours d'introduction.
 - Lorsqu'on ne veut pas les générer a priori.
Exemple : calcul du produit de deux automates "à la volée".

En TD / Exercices

- Garder uniquement les états accessibles d'un automate : étant donné un automate, écrire un algorithme qui retourne un automate équivalent avec tous ses états accessibles.
- Même question avec la co-accessibilité.
- Produire la version émondée d'un automate. Langage reconnu ?
- Produit de deux automates « à la volée ».

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

- 1 Parcours
- 2 Notions d'accessibilité et de co-accessibilité
- 3 Détection de cycles
- 4 Quelques problèmes de décision
- 5 Résumé

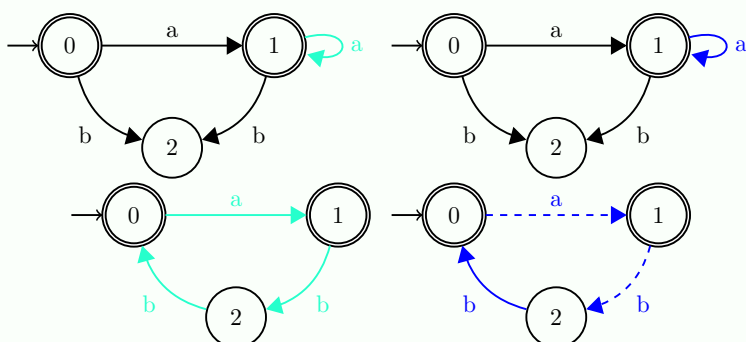
Détection de cycles

Définition (Cycle - deux définitions équivalentes)

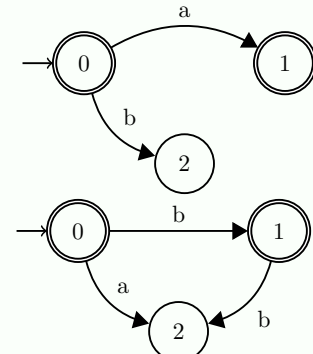
- **séquence (non-vide) de transitions consécutives** (l'état d'arrivée d'une transition est l'état de départ de la prochaine transition) t. q. le premier et le dernier états soient identiques.
- automate avec une **transition arrière** :
 - soit une transition d'un état sur lui même,
 - soit une transition d'un état vers l'un de ces ancêtres dans l'arbre produit par le parcours en profondeur.

Exemple et contre-exemple (Cycle)

Automates avec cycle



Automates sans cycle

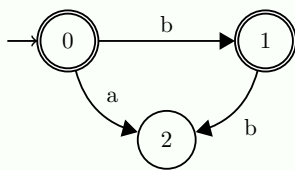


Détection de cycles - Une remarque / observation

Question

Lors d'un parcours en profondeur, est-ce que la découverte d'un état déjà visité implique l'existence d'un cycle ?

Exemple (Découvrir un état déjà visité n'implique pas l'existence de cycle)



Lors du parcours depuis l'état initial (où les voisins sont explorés dans l'ordre alphabétique) : 2 est visité, puis il est rencontré à nouveau.

Remarque La notion d'état visité n'est pas assez fine. Elle sert uniquement pour la terminaison de l'algorithme de parcours (et ne pas traiter des états déjà traités). □

Détection de cycles - Intuition

Intuition pour trouver un cycle avec le parcours en profondeur récursif

- Se fait suivant la sous-structure d'arbre produit par le parcours.
- Lors du parcours *qui a pour origine un état q* , si on rencontre l'état q , alors il y a un cycle.
- **Se souvenir des noeuds par lesquels on est passé depuis et provenant de l'appel initial.**

◀ Détection de cycle avec parcours en profondeur.

Nous allons utiliser un ensemble d'états, `Pile_d_appel`, pour « se souvenir » des états parents par lesquels le parcours est passé.

Nous définissons, comme pour le parcours récursif, une procédure `detection_cycle()` appelée sur l'automate et une procédure `detection_cycle_état()` appelée sur les états.

Lors de tout appel à `detection_cycle_état(q)`, pour un état q , `Pile_d_appel` contient l'ensemble des états (« parents ») avec lesquels `detection_cycle_état` a été appelée et qui ont donné lieu à l'appel `detection_cycle_état(q)`.

Détection des cycles dans un AD

Basé sur l'algorithme récursif de parcours en profondeur - algorithme 1

Algorithme 9 *detection_cycle()* pour détecter l'existence d'un cycle dans un automate

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

Sortie : **vrai** s'il existe un cycle dans A , **faux** sinon

- 1: **ens d'états** $\text{Déjà_visités}, \text{Pile_d_appel} := \emptyset$;
- 2: **pour** chaque état $q \in Q$ **faire**
- 3: **si** *detection_cycle_état*(q) **alors retourner vrai fin si**
- 4: **fin pour ; retourner faux**

Algorithme 10 *detection_cycle_état()* pour détecter les cycles à partir d'un état

Entrée : $q \in Q$ (* un état *)

Sortie : **vrai** s'il existe un cycle à partir de q , **faux** sinon

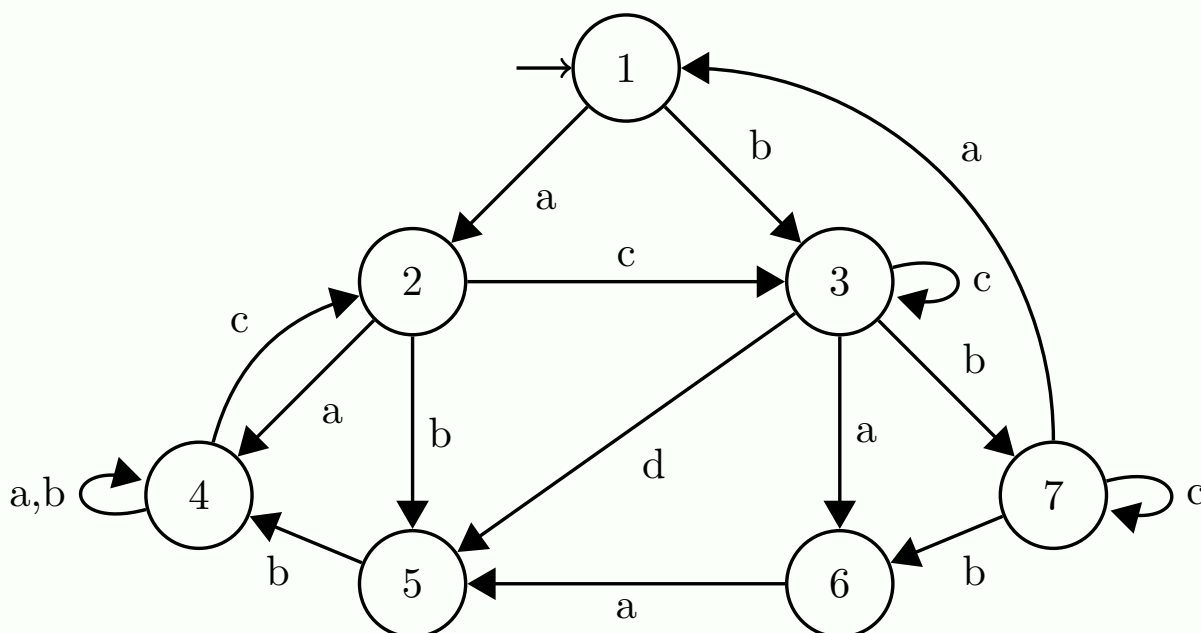
- 1: **si** $q \in \text{Pile_d_appel}$ **alors retourner vrai fin si**
- 2: **si** $q \in \text{Déjà_visités}$ **alors retourner faux fin si**
- 3: $\text{Pile_d_appel} := \text{Pile_d_appel} \cup \{q\}$; (* on ajoute q à la pile d'appel *)
- 4: $\text{Déjà_visités} := \text{Déjà_visités} \cup \{q\}$; (* l'état q devient déjà visité *)
- 5: **pour** chaque état $q' \in \text{Succ}(q)$ **faire**
- 6: **si** *detection_cycle_état*(q') **alors retourner vrai fin si**
- 7: **fin pour**
- 8: $\text{Pile_d_appel} := \text{Pile_d_appel} \setminus \{q\}$; (* l'état q est supprimé de la pile d'appel *)
- 9: **retourner faux**

Remarque Cet algorithme peut être adapté pour retourner tous les cycles. □

Détection des cycles dans un AD

Basé sur l'algorithme récursif de parcours en profondeur - algorithme 1

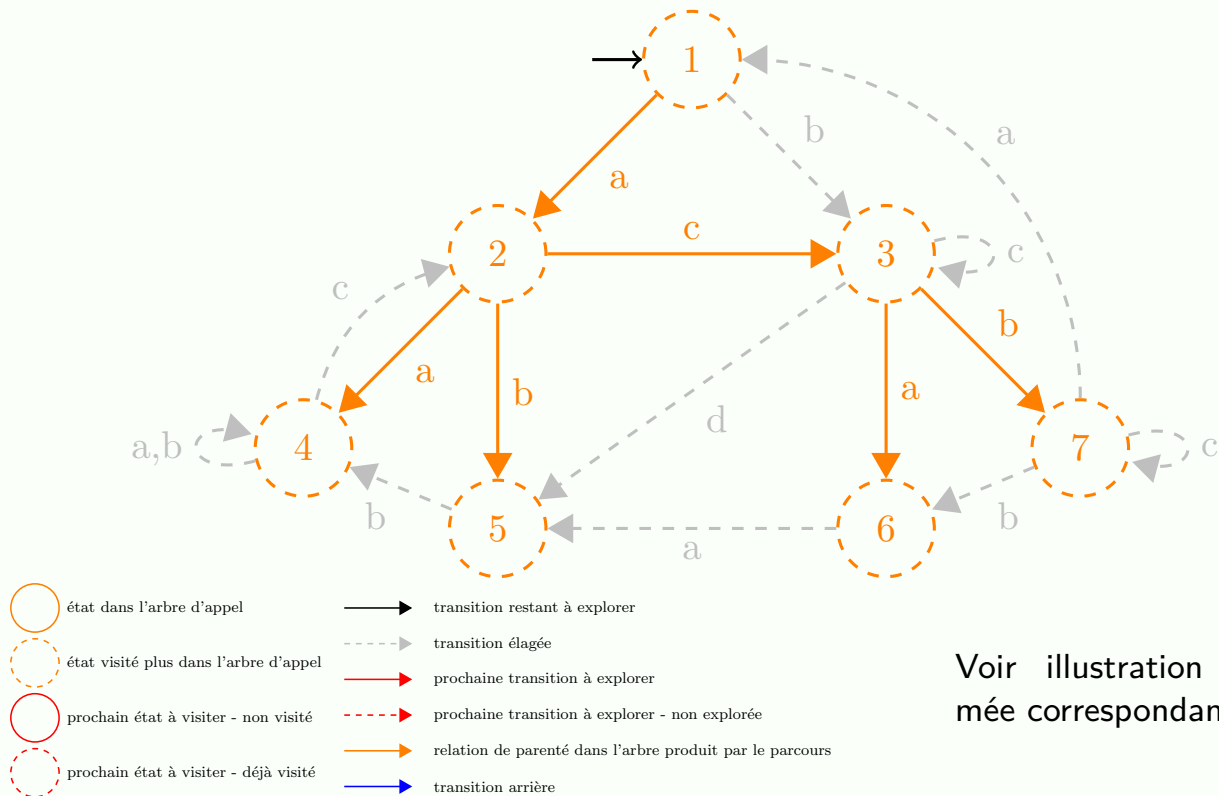
Exemple (Détection des cycles dans un AD avec l'algorithme 1)



Détection des cycles dans un AD

Basé sur l'algorithme récursif de parcours en profondeur - algorithme 1

Exemple (Détection des cycles dans un AD avec l'algorithme 1)



Voir illustration animée correspondante

Détection des cycles dans un AD

Limites de l'algorithme 1 - Vers un second algorithme

Limites de l'algorithme 1

- Les tests $q \in \text{Pile_d_appel}$ et $q \in \text{Déjà_visités}$ sont en $\mathcal{O}(|Q|)$.
- La pile d'appel existe déjà grâce aux appels récursifs dans le parcours en profondeur.

Modifications à l'algorithme 1

- Utiliser un coloriage des états dans $\{\text{BLANC}, \text{GRIS}, \text{NOIR}\}$:
 - **BLANC** : état non traité ;
 - **GRIS** : état en cours de traitement, successeurs pas tous traités ;
 \hookrightarrow l'état est dans la pile
 - **NOIR** : état et tous ses successeurs traités ;
 \hookrightarrow l'état n'est pas dans la pile
- Tester la couleur d'un état est en $\mathcal{O}(1)$.

Détection de cycles dans un AD

Basé sur l'algorithme récursif de parcours en profondeur - algorithme 2

Algorithme 11 *detection_cycle()* pour l'existence de cycles dans un AD

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ (* un AD *)

Entrée : $q \in Q$ (* un état *)

Sortie : **vrai** s'il existe un cycle à partir de q , **faux** sinon

- 1: couleur : $Q \rightarrow \{\text{BLANC}, \text{GRIS}, \text{NOIR}\}$ (* initialement : $\forall q \in Q : \text{couleur}(q) = \text{BLANC}$ *)
- 2: **pour** chaque état $q \in Q$ **faire**
- 3: **si** couleur(q) == **BLANC** et *detection_cycle_état*(q) **alors retourner vrai fin si**
- 4: **fin pour retourner faux**

Algorithme 12 *detection_cycle_état()* pour l'existence de cycles à partir d'un état

Entrée : $A = (Q, \Sigma, q_{\text{init}}, \delta, F)$ un AD, $q \in Q$ un état

Sortie : **vrai** s'il existe un cycle à partir de q , **faux** sinon

- 1: couleur(q) := **GRIS**
- 2: **pour** chaque état $q' \in \text{Succ}(q)$ **faire**
- 3: **si** couleur(q') == **GRIS** **alors retourner vrai fin si**
- 4: **si** couleur(q') == **BLANC** et *detection_cycle_état*(q') **alors retourner vrai**
 (* q' n'a pas été traité et il y a une transition arrière dans le sous-arbre de racine q' *)
- 5: **fin si**
- 6: **fin pour**
- 7: couleur(q) := **NOIR**
- 8: **retourner faux**

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

- 1 Parcours
- 2 Notions d'accessibilité et de co-accessibilité
- 3 Détection de cycles
- 4 Quelques problèmes de décision
- 5 Résumé

Problèmes de décision

Définition

Définition (Problème de décision)

- Question que l'on peut *exprimer mathématiquement* (formellement).
- Question à un certain nombre de paramètres que l'on souhaite pouvoir passer à un programme informatique.
- La réponse à la question est *oui ou non*.

Deux sortes de problèmes de décision existent :

- **problèmes décidables** : on peut trouver un algorithme ou un programme qui sait répondre à la question (avec une mémoire et un temps non-borné).
- **problèmes indécidables** : on ne peut pas trouver un algorithme ou un programme qui sait répondre à la question (de manière générale).

Exemple (Problème de décision)

- décidables : évaluation d'un circuit, voyageur de commerce, SAT(isfiabilité), ...
- indécidables : arrêt d'une machine de Turing ou de Minsky (automate avec deux compteurs) ou d'un programme, solutions entières d'une équation diophantienne, ...

Deux problèmes de décision sur les AD

Considérons un AD A .

Problème du langage vide

Le langage reconnu par A est-il vide ?

Problème de la finitude du langage

Le langage reconnu par A est-il (de cardinalité) fini(e) ?

Pour répondre à ces questions, nous allons utiliser les notions d'*accessibilité*, de *co-accessibilité* et de *cycle*.

Décision du problème du langage vide

Théorème : décision du problème du langage vide

Le problème du *langage vide* est *décidable* pour les automates à états finis (déterministes).

Algorithme 13 Procédure de décision pour le problème du langage vide

Entrée : $A = (Q, \Sigma, \delta, q_{\text{init}}, F)$ un AD

Sortie : **vrai** si le langage reconnu par A est vide, **faux** sinon

- 1: **ens d'états** Accessibles $:= \text{etats_accessibles}(A)$;
- 2: **retourner** $(\text{Accessibles} \cap F) == \emptyset$;

Décision du problème de la finitude du langage

Théorème : décision du problème de la finitude du langage

Le problème de *la finitude du langage* est *décidable* pour les AD.

Idée intuitive de l'algorithme

- Le langage accepté ne sera pas fini, si l'automate peut accepter "autant de mots qu'on veut".
- Il y a un nombre fini d'états.
- Il faut pouvoir donc arriver dans un état accepteur par un nombre infini de chemins différents.
- Il faut l'existence d'un **cycle**.

Le langage reconnu par un automate $(Q, \Sigma, \delta, q_{\text{init}}, F)$ est infini
ssi
il existe un cycle non trivial accessible et co-accessible.

Décision du problème de la finitude du langage

Algorithme 14 Procédure de décision pour le problème du langage infini

Entrée : $A = (Q, \Sigma, \delta, q_{\text{init}}, F)$ un AD

Sortie : **vrai** si le langage reconnu par A est infini, **faux** sinon

- 1: **ens d'états** $\text{EtatsEmonde} := \text{etats_accessibles}(A) \cap \text{etats_coaccessibles}(A)$;
 - 2: **automate** $E := (Q \cap \text{EtatsEmonde}, \Sigma, \delta \cap \text{EtatsEmonde} \times \Sigma \times \text{EtatsEmonde}, q_{\text{init}}, F \cap \text{EtatsEmonde})$;
 - 3: **retourner** $\{q \in \text{EtatsEmonde} \mid \text{detection_cycle_etat}()(E, q)\} \neq \emptyset$;
-

Plan Chap. 5 - Algorithmes et problèmes de décision sur les AD

- 1 Parcours
- 2 Notions d'accessibilité et de co-accessibilité
- 3 Détection de cycles
- 4 Quelques problèmes de décision
- 5 **Résumé**

Résumé du Chap. 5 - Algorithmes et problèmes de décision sur les AD

Algorithmes sur les automates déterministes

- *Parcours* :
 - profondeur : en récursif et itératif,
 - largeur (itératif).
- Détection de cycle dans un automate :
 - basée sur un parcours en profondeur,
 - détection de transition arrière,
- *Accessibilité et de co-accessibilité* :
 - état accessible : existence d'un chemin depuis l'état initial vers cet état,
 - état co-accessible : existence d'un chemin depuis cet état jusqu'à un état accepteur.
- *Décidabilité* de certains *problèmes de décision* : langage vide et finitude du langage.

Remarque Les notions des prochains chapitres et les exercices de TD nous permettront de répondre à d'autres problèmes de décision. ☐

Remarque Les algorithmes de parcours, détection de cycle, de calcul des états accessibles et co-accessibles restent valables pour les automates non-déterministes que nous aborderons dans un prochain chapitre. ☐

Chap. 5 - Bonus

Bonus

Définir des algorithmes permettant d'obtenir des automates :

- reconnaissant l'intersection des langages d'automates passés en paramètres ;
- reconnaissant l'union et l'union exclusive des langages d'automates passés en paramètres.