Chapitre 1 Automates finis

Introduction

Alan Turing a décrit à 1936 un modèle de "machine idéale" auquel il a laissé son nom.

Une **machine de Turing** est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur et sa mémoire. Alan Turing voulait donner une définition précise au concept d'algorithme ou « procédure mécanique ». Ce modèle est toujours largement utilisé en informatique théorique, en particulier pour résoudre les problèmes de complexité algorithmique et de calculabilité.

D'autres modèles furent proposés, qui sont tous équivalents à la machine de Turing. Church a démontré dans sa thèse que ces modèles sont les plus généraux possibles.

À l'origine, le concept de machine de Turing, inventé avant l'ordinateur, était censé représenter une personne virtuelle exécutant une procédure bien définie, en changeant le contenu des cases d'un tableau infini, en choisissant ce contenu parmi un ensemble fini de symboles. D'autre part, la personne doit mémoriser un état particulier parmi un ensemble fini d'états. La procédure est formulée en termes d'étapes très simples, du type : « si vous êtes dans l'état 42 et que le symbole contenu sur la case que vous regardez est '0', alors remplacer ce symbole par un '1', passer dans l'état 17, et regarder une case adjacente (droite ou gauche) ».

Ces travaux amenèrent à distinguer entre les fonctions qui sont calculables en théorie et celles qui ne le sont pas même en théorie. (On dit qu'une fonction est calculable si elle peut être calculée par la machine de Turing en temps (nombre d'opérations) fini). [1]

Un modèle plus simple et donc plus restreint mais fort utile est celui d'un **automate fini**. Un automate fini est une machine abstraite constituée d'**états** et de **transitions**. Cette machine est destinée à traiter des **mots** fournis en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée. L'automate est dit « fini » car il possède un nombre fini d'états distincts : il ne dispose donc que d'une mémoire bornée, indépendamment de la taille de la donnée sur laquelle on effectue les calculs.

Automates finis fournissent un outil de construction d'algorithmes particulièrement simple. On les retrouve dans la **modélisation de processus**, le **contrôle**, les **protocoles de communication**, la **vérification de programmes**, la **théorie de la calculabilité**, dans **l'étude**

---

[1] Il est assez aisé de simuler une machine de Turing sur un ordinateur moderne, jusqu'au moment où la mémoire de l'ordinateur devient éventuellement pleine (si la machine de Turing utilise une très grande partie du ruban) !

Il est aussi possible de construire une machine de Turing purement mécanique. Le mathématicien Karl Scherer en construisit une en 1986, en utilisant des jeux de construction en métal et en plastique, ainsi que du bois. Sa machine, haute d'un mètre et demi, utilise des ficelles pour lire, déplacer et écrire les données (représentées à l'aide de roulements à billes).

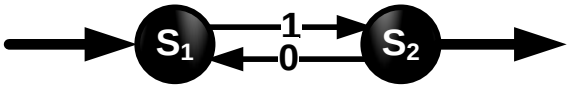
La machine est actuellement exposée dans le hall du département d'informatique de l'Université d'Heidelberg en Allemagne.

De même, en utilisant environ 300 miroirs, il est possible de créer une machine de Turing universelle optique en utilisant la méthode dite du fer à cheval, conçue par Stephen Smale

des langages formels et en **compilation**. Ils sont utilisés dans la **recherche des motifs dans un texte**.

Un automate est constitué d'états et de transitions. Son comportement est dirigé par un mot fourni en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée.

Voici un exemple très simple d'un automate fini :



Dans l'exemple ci-dessus, pour l'entrée 1010101, et si l'automate démarre en s_1, il passe successivement par les états $s_1s_2s_1s_2s_1s_2s_1$ s_2, le calcul correspondant est

$$s_1 \overset{1}{\rightarrow} s_2 \overset{0}{\rightarrow} s_1 \overset{1}{\rightarrow} s_2 \overset{0}{\rightarrow} s_1 \overset{1}{\rightarrow} s_2 \overset{0}{\rightarrow} s_1 \overset{1}{\rightarrow} s_2$$

Le mot arrivant à un état terminal (état accepteur), il est accepté (lu, reconnu) par l'automate. Le mot 101010 n'arrivera pas à un état terminal, il ne sera donc pas reconnu. La lecture du mot 01 s'arrêtera au premier caractère, car il n'y a pas de transition $s_1 \overset{0}{\rightarrow} s_2$; il ne sera pas reconnu non plus.

Un tel automate est dit « **fini** » car il possède un nombre fini d'états : il ne dispose donc que d'une mémoire bornée. On peut très bien considérer des automates sans limitation sur le nombre d'états: la théorie qui en résulte est très analogue à la théorie habituelle.

Un automate fini peut être vu comme un graphe orienté étiqueté: les états sont les sommets et les transitions sont les arêtes étiquetées. L'état initial est marqué par une flèche entrante; un état final est, selon les auteurs, soit marqué d'une flèche sortante, comme sur la figure ci-dessus, soit doublement cerclé.

Une autre façon commode de représenter un automate fini est sa *table de transition*. Elle donne, pour chaque état et chaque lettre, l'état d'arrivée de la transition. Voici la table de transition de l'automate donné en exemple :

		0	1
Entrée	S_1	--	S_2
Sortie	S_2	S_1	--

Il existe plusieurs types d'automates finis. Dans notre cours, nous n'allons étudier que les « **accepteurs** » (ou « **reconnaisseurs** »), qui produisent en sortie une réponse « oui » ou « non » selon qu'ils acceptent (oui) ou rejettent (non) le mot présenté en entrée. Dans l'exemple ci-dessus, le mot 1010 n'est pas accepté, mais le mot 10101 est accepté. D'autres automates (**systèmes de reconnaissance)** classent l'entrée par catégorie: au lieu de répondre par oui ou par non, ils répondent par une classification; de tels automates se rencontrent par

exemple en linguistique. Les **capteurs** produisent un certain résultat en fonction de l'entrée. Les **automates pondérés** associent à chaque mot une valeur numérique.

Les automates finis peuvent caractériser des **langages** (c'est-à-dire, des ensembles de mots) finis (le cas standard), des **langages de mots infinis** (automates de Rabin, automates de Büchi), ou encore divers **types d'arbres** (automates d'arbres).

Quelques définitions

Pour définir un automate fini, on a besoin des éléments suivants :

- **Un alphabet *A*** qui est un ensemble fini d'objets qu'on appelle « lettres » ou « caractères » mais qui peuvent, selon le cas, être de n'importe quel genre (instructions machine, état civil d'une personne, type d'objet géométrique etc…)

- Des **mots** sont des séquences ordonnées de caractères de l'alphabet. Dans nos applications, on aura affaire qu'à des mots de longueur finie, mais il n'est pas interdit d'utiliser des mots de longueur infinie (automates de Rabin, automates de Büchi). La longueur d'un mot est le nombre de lettres le constituant.
  Exemple : $w = abacda$ – mot de longueur 6 sur l'alphabet latin (ou bien sur l'alphabet $\{a,b,c,d\}$).

- **Un mot vide** est noté par ε ou par 1. C'est le seul mot de longueur 0.

- On note A^* **l'ensemble** de tous les mots sur l'alphabet A, y compris le mot vide.

Un automate fini sur l'alphabet *A* est donné par un quadruplet (Q, I, T, E) où

- Q est un ensemble fini d'états de l'automate,

- $I \subset Q$ est un ensemble d'états initiaux,

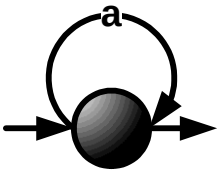
- $T \subset Q$ est un ensemble d'états terminaux,

- $E \subset Q \times (A \cup \{\varepsilon\}) \times Q$ est un ensemble de triplets $(p.a.q)$ appelés les flèches ou les transitions de l'automate.

  Souvent, on dénote l'automate par un quintuplet (A, Q, I, T, E), incluant l'alphabet directement dans la notation.

Remarque

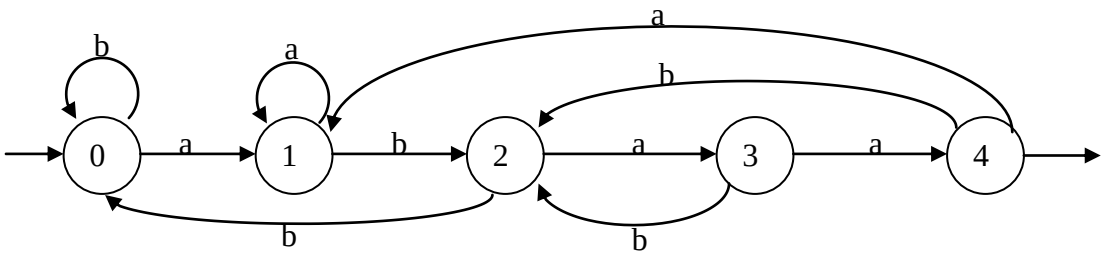
Nous allons temporairement oublier la présence du mot vide (ε) dans la définition de l'ensemble E, jusqu'à l'introduction explicite d'automates asynchrones. Dans tous les exemples que nous allons traiter jusque ce point-là, on pourra donc définir l'ensemble E comme faisant partie de $Q \times A \times Q$.
Il n'empêche que le mot vide pourra faire partie du langage reconnu par l'automate (par exemple, l'automate suivant reconnaît les mots ε, a, aa, aaa etc. :



L'interprétation intuitive :

A chaque instant l'automate se trouve dans l'un des états ***p*** de l'ensemble d'états ***Q***. Il lit alors l'une des lettres ***a*** de l'alphabet ***A*** (la lettre suivant du mot qui vient à l'entrée) et passe dans un état ***q*** tel que ***(p.a.q)*** soit une flèche.

Exemple :



$\boldsymbol{A} = \{\boldsymbol{a}, \boldsymbol{b}\}$; $\boldsymbol{Q} = \{0, 1, 2, 3, 4\}$; $\boldsymbol{I} = \{0\}$; $\boldsymbol{T} = \{4\}$

La lecture du mot ***w*** se termine dans l'état 4 ssi ***w*** se termine par ***abaa***.

Le même automate peut être représenté par une **table de transitions** :

	Etat	**état résultant**	
		en lisant a	**en lisant b**
(entrée)	0	1	0
	1	1	2
	2	3	0
	3	4	2
(sortie)	4	1	2

Remarque :

Si vous regardez cet automate de près, vous verrez que nous avons libellé les états de telle façon que l'automate se trouve dans l'état numéro ***i*** ssi le plus long **suffixe du mot déjà lu** qui est en même temps un **préfixe de *abaa,*** est de **longueur *i***.

On peut montrer que cet automate réalise de façon optimale l'algorithme de recherche de la séquence ***abaa*** dans un texte : il résout pour ce mot l'algorithme du « string matching » utilisé par exemple dans les éditeurs de textes.

La lecture :

L'automate prend un mot (constitué de symboles de son alphabet) en entrée et démarre dans son ou ses état(s) initial(-aux). Il parcourt ensuite le mot de gauche à droite: si l'automate se trouve dans un état p et le symbole à lire est a, alors s'il existe(nt) un ou des état(s) q_i tel(s) que la transition ($p.a.q_i$) existe, il passe aux états q_i, et de suite. (S'il y a plus d'un état q_i, on considère chacun d'eux séparément). La lecture se termine soit si la lettre suivante ne peut pas être lue (il n'y a pas de transition y correspondant), soit si on a atteint la fin du mot. Si, à la fin de la lecture, l'automate est dans un état final (accepteur), on dit qu'il accepte l'entrée ou qu'il la reconnaît. Autrement, on dit qu'il la rejette.

Définitions formelles et notation:

- Pour $\forall a \in A$, on note $p \xrightarrow{a} q$ ssi il existe une transition (appelée aussi une **flèche**) $(p.a.q) \in E$.

  Ensuite, pour $u \in A^*$ et $a \in A$ (rappel : donc u est un mot et a un caractère) on note $p \xrightarrow{ua} q$ ssi $\exists r \in Q$ tq $p \xrightarrow{u} r$ et $\xrightarrow{a} q$.
- De cette façon, on obtient la notion de **chemin** : $p \xrightarrow{w} q$ est un chemin de p à q d'étiquette w.

  Un chemin peut consister en une ou plusieurs transitions consécutives ; il consiste en autant de transitions qu'il y a de caractères dans le mot étiquette.

  Comme un mot de longueur supérieure à 1 peut toujours être scindé en deux mots (dont un ou tous les deux peuvent ne consister qu'en un seul caractère), un chemin de longueur supérieure à 1 peut toujours être scindé en deux chemins : pour $u,v \in A^*$ on a $p \xrightarrow{uv} q$ ssi $\exists r \in Q$ tq $p \xrightarrow{u} r$ et $r \xrightarrow{v} q$. La preuve s'obtient facilement par récurrence.
- Un mot $w \in A^*$ est donc reconnu par l'automate fini s'il existe un chemin $i \xrightarrow{w} t$ où $i \in I$ et $t \in T$ – c.à.d. qu'en partant d'un état initial et en lisant le mot w, on atteint un état terminal à la fin de la lecture.

Définition :

Le **langage L reconnu** par l'automate fini est l'ensemble de tous les mots reconnus.

* * *

Exercice :

Construire un automate fini reconnaissant les entiers écrits en base 2 divisibles par 3.

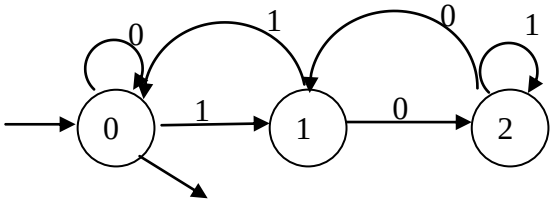
Solution :
Ajout d'un 0 à la fin d'un nombre binaire le multiplie par 2.
Ajout d'un 1 à la fin d'un nombre binaire le multiplie par 2 et lui ajoute 1.
Marquant les états par le reste de la division entière par 3 :

N	N mod 3	ajout d'un 0 à la fin	reste	ajout d'un 1 à la fin	reste
3n	0	6n	0	6n+1=3×2n + 1	1
3n + 1	1	6n + 2=3×2n + 2	2	6n+3=3×(2n + 1)	0
3n +2	2	6n + 4=3×(2n + 1) + 1	1	6n+5=3×(2n + 1) + 2	2

Notons les états par le reste de la division entière par 3. On obtient

Etat	**0**	**1**
0	0	1
1	2	0
2	1	2

L'entrée se fait en 0, et la sortie se fait également en 0 :



Mais en fait, cet automate est bon pour reconnaitre des nombres en écriture binaire avec n'importe quel reste de la division entière par 3, au prix de choisir quel état est terminal. L'état terminal 0 correspond au reste 0 (divisibilité par 3) ; l'état terminal 1, au reste 1 ; l'état terminal 2, au reste 2.
Le seul inconvénient est qu'à part les nombres divisibles par 3, cet automate reconnaît aussi, le mot vide (ε) qui n'est pas un nombre, et pour lequel la notion même de divisibilité par un nombre semble assez bizarre. Peut-on y remédier ? La réponse se trouve dans la section suivante.

Standardisation des automates

Définition :
L'automate $C=(Q, I, T, E)$ sur le langage A est standard si $I=\{i\}$ et, pour tout $a \in A$ et tout $q \in Q$, $(q.a.i) \notin E$ (cela peut se formuler en mots ainsi : l'automate fini est standard s'il possède un unique état initial et si aucune transition n'aboutit dans cet état).

Propriété importante :
Un automate standard possédant plus d'un état et dont l'état initial n'est pas terminal, ne reconnaît pas le mot vide. Cette propriété est évidente : pour arriver de l'état initial à un état terminal, il faut au moins passer par une des transitions partant de l'état initial, donc lire au moins un caractère.

Algorithme de standardisation

On peut rendre tout automate fini non standard C standard. En ce faisant, on ne modifie pas le langage qu'il reconnaît.

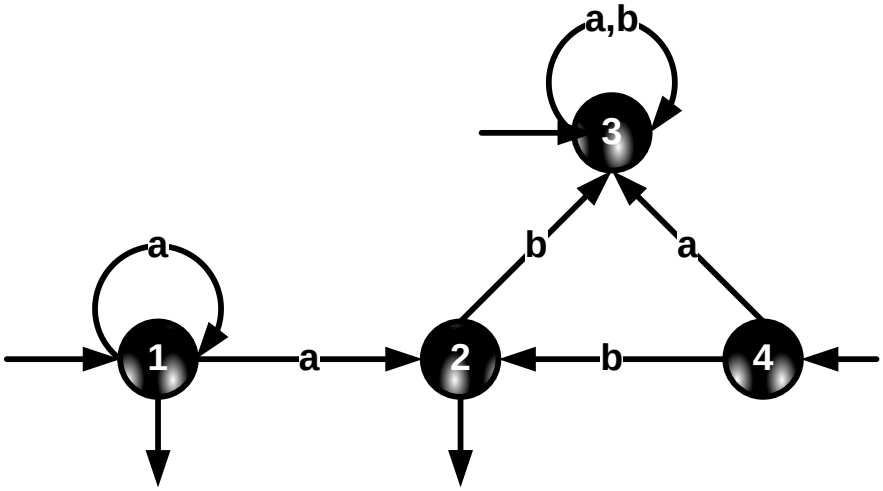
Pour standardiser, on ajoute à l'automate ***C*** un nouvel état qui devient l'unique état initial et duquel, pour chacune des transitions qui partaient des anciens états initiaux, part une transition de même étiquette vers les mêmes états.

Formellement, on construit un nouvel automate $C' = (Q \cup \{i\}, \{i\}, T', E')$ (le **standardisé** du ***C***) avec :

- $\{i\} \notin Q$;
- $T' = \begin{cases} T \text{ si } I \cap T = \emptyset \\ T \cup \{i\} \text{ si } I \cap T \neq \emptyset \end{cases}$
- $E' = E \cup \{(i.a.q) \mid \exists\, q_i \in I \text{ tel que } (q_i.a.q) \in E\}$

En mots : on ajoute un nouvel état ***i,*** et on ajoute des flèches vers tous les états (et qui portent les mêmes étiquettes) vers lesquels l'automate ***C*** possède des transitions partant de ces états initiaux. (En disant « tous les états », on y inclut éventuellement les états initiaux eux-mêmes). Si aucun état initial du ***C*** n'est terminal, l'ensemble d'états terminaux n'est pas modifié. Si au moins un état initial du ***C*** est terminal, le nouvel état ***i*** s'ajoute aux états terminaux.

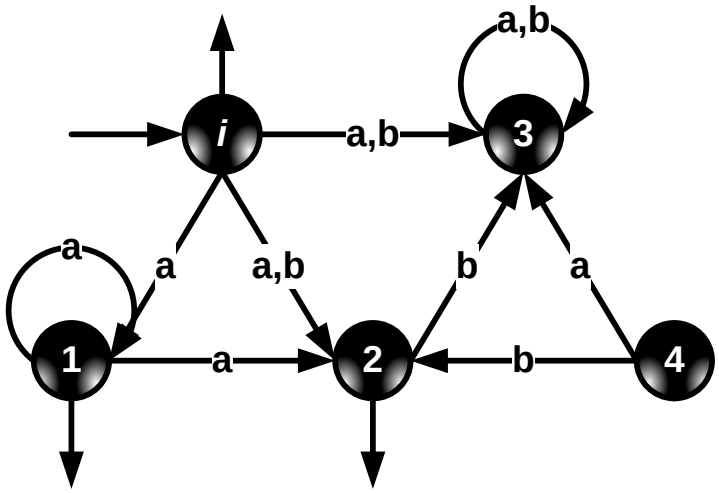
Exemple :
L'automate suivant n'est pas standard : il possède trois états initiaux et des transitions aboutissant dans chacun d'entre eux.



Voici la table de transitions de cet automate :

	état	a	b
E,S	1	1,2	--
S	2	--	3
E	3	3	3
E	4	3	2

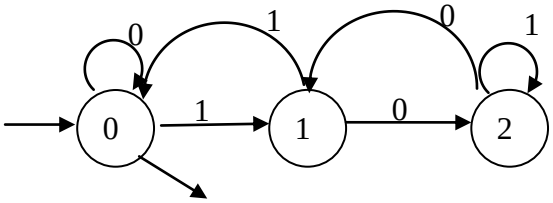
Une des entrées est une sortie ; les transitions impliquant les entrées sont (1.***a***.1), (1.***a***.2), (3.***a***.3), (3.***b***.3), (4.***a***.3) et (4.***b***.2). On ajoute donc l'état ***i*** qui est à la fois l'entrée unique et une des sorties, et on créé les transitions (***i***.***a***.1), (***i***.***a***.2), (***i***.***a***.3), (***i***.***b***.3), (***i***.***a***.3) et (***i***.***b***.2) :



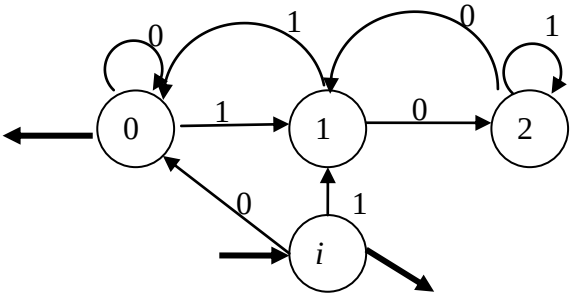
(En réalité, cet automate – appelons-le ***C'*** – est assez bizarre. Il contient un état **non accessible** 4 (où on ne peut pas arriver à partir de l'état initial) et un état **non coassessible** 3 (à partir duquel on ne peut pas arriver à une sortie). Il pourrait donc être simplifié en enlevant ces deux états avec ses transitions ; en le faisant on obtient **l'émondé** du ***C'***. Il n'empêche que cet exemple illustre bien la procédure de standardisation).

Regardons l'automate standardisé. La seule façon de laquelle il peut reconnaître le mot vide, c'est si l'état ***i*** est un état de sortie. Donc on obtient un moyen très simple de modifier un automate de telle façon qu'il cesse de reconnaître le mot vide s'il le fait : il faut le standardiser et enlever la sortie sur l'état initial.

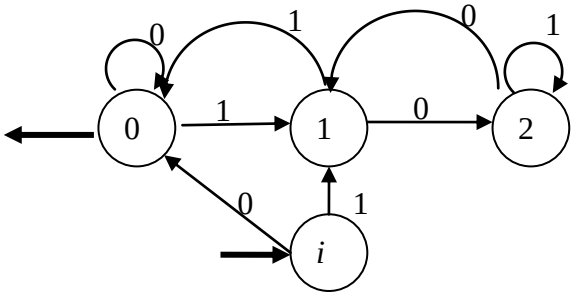
Appliquons cette recette à l'automate



qui reconnaît tous les nombres en écriture binaire divisible par 3, mais aussi le mot vide.
Il y a un seul état d'entrée, 0. Il est également le seul état de sortie. Pour standardiser cet automate, il faut donc lui ajouter un nouvel état ***i*** qui sera un état de sortie (l'état 0 le reste aussi), et créer les transitions (***i***.0.0) et (***i***.1.1). On obtient le standardisé



Cet automate est équivalent à l'automate initial ; pour lui enlever la propriété de reconnaître le mot vide sans modifier le reste, il suffit d'enlever la sortie en ***i :***



Nous avons obtenu an automate reconnaissant toutes les écritures binaires des nombres divisibles par 3, sans qu'il reconnaisse le mot vide.

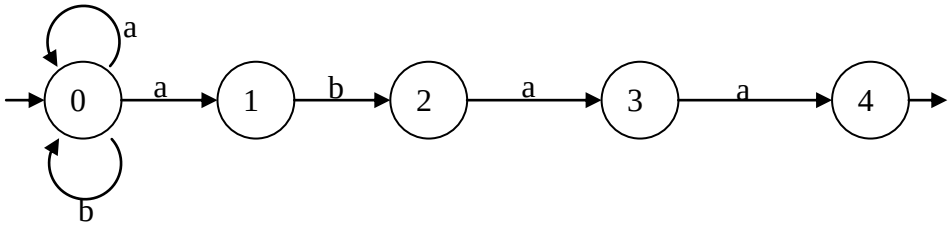
Remarques générales

1. Dans ce cours, nous allons donner ***quelques*** algorithmes concernant des opérations sur les automates. Il en existe d'autres algorithmes donnant les mêmes résultats (pour la minimisation, pour la complémentarisation, pour l'obtention du langage reconnu par l'automate donné, pour la construction d'un automate reconnaissant le langage donné, etc). Nous exigeons que vous maitrisiez parfaitement les algorithmes propres à ce cours.
2. Nous illustrons nos idées par des exemples assez simples. Cela aide à la compréhension, mais parfois cela peut donner des idées fausses. Notamment, on pourrait se demander pourquoi on doit développer des algorithmes formels tandis que très souvent la réponse « se voit » immédiatement à partir du dessin. La vérité est que dans la vie réelle, les automates faisant partie des logiciels peuvent avoir des milliers voire des millions d'états ; rien n'est « vu », et aucun raisonnement intuitif n'est possible dans de tels cas. La seule solution consiste à développer des algorithmes programmables adaptés à n'importe quelle taille d'automate.

Automates déterministes

On distingue des automates finis déterministes et non déterministes. L'automate que nous avons montré précédemment est déterministe.

Voici l'exemple d'un automate ***non déterministe*** :



(Cet automate reconnaît tous les mots qui se terminent par ***abaa***. Il est très facile à construire.)

Le fait qu'il n'est pas déterministe se manifeste en ce que, en lisant ***a*** à partir de l'état 0, on peut aller et à l'état 0, et à l'état 1, ce qui ne serait pas possible pour un automate déterministe.

On va montrer dans ce qui suit que l'on peut toujours construire, à partir d'un automate fini quelconque, un autre automate qui est déterministe et qui lui est équivalent (reconnaît le même langage).

Automate déterministe, définition

L'automate ***C = (Q, I, T, E)*** est déterministe ssi pour $\forall p \in Q$ et $\forall a \in A$ $\exists$ **au plus un** état $q \in Q$ tq ***(p.a.q)***, et qu'il y ait **un seul état initial** : ***I* = { *i* }**.
On peut donc caractériser un automate déterministe ***C*** par un quadruplet ***(Q, i, T, E)*** où ***i*** est l'état initial.

Si la transition ***(p.a.q)*** existe, on notera ***p.a=q.*** Si elle n'existe pas, on conviendra que ***p.a*** n'a pas de valeur.

On peut facilement vérifier par récurrence sur la longueur des mots que pour un automate déterministe ***C,*** pour $\forall p \in Q$ et $\forall$ mot $u \in A^*$, il existe au plus un état $q \in Q$ tq $p \xrightarrow{u} q$.

> (Cette assertion est différente de la définition ! Ici, il s'agit d'un mot et d'un chemin, tandis que la définition parle d'un caractère et d'une transition !)

On notera alors ***p.u=q*** en convenant que ***p.u*** n'est pas défini quand il n'existe pas de tel état ***q***; et on aura pour deux mots $u, v \in A^*$:

$$(p.u).v = p.(uv)$$

Déterminisation

Soit ***C = (Q, I, T, E)*** un automate fini.

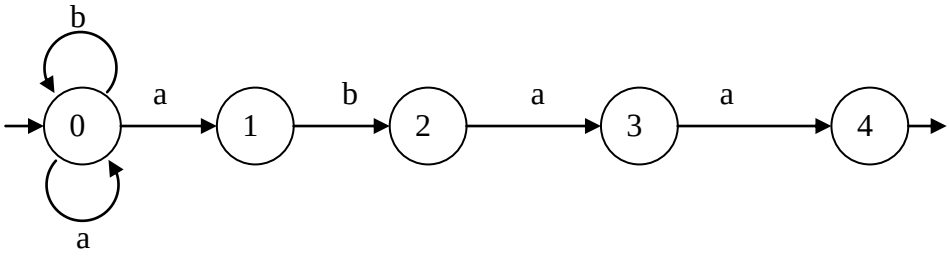
Notons Π l'ensemble des parties de ***Q***. C'est un ensemble fini puisque si ***Q*** a ***n*** éléments, Π en a 2^n. (Par exemple, pour ***Q***=$\{0,1,2\}$, Π=$\{\varnothing,\{0\},\{1\},\{2\},\{0,1\},\{0,2\},\{1,2\},\{1,2,3\}\}$: il y a 2^3=8 parties).

Un automate déterministe ***D*** correspondant à ***C*** prend

- comme ensemble d'états un sous-ensemble ***P*** de l'ensemble Π des parties de ***Q***.
- comme unique état initial l'ensemble ***I*** des états initiaux de ***C.***
- comme ensemble d'états terminaux l'ensemble $\zeta=\{u \in P | u \cap T \neq \varnothing\}$ des parties de ***Q*** qui contiennent au moins un état terminal de ***C***.
- comme flèches l'ensemble des ***(u.a.v)*** où $u \in P$ et ***v*** est l'ensemble de tous les états $q \in Q$ tels qu'il existe un état ***p*** dans ***u*** avec $(p.a.q) \in E$.

Cet algorithme formel s'explique le plus facilement sur un exemple :

Exemple de déterminisation.



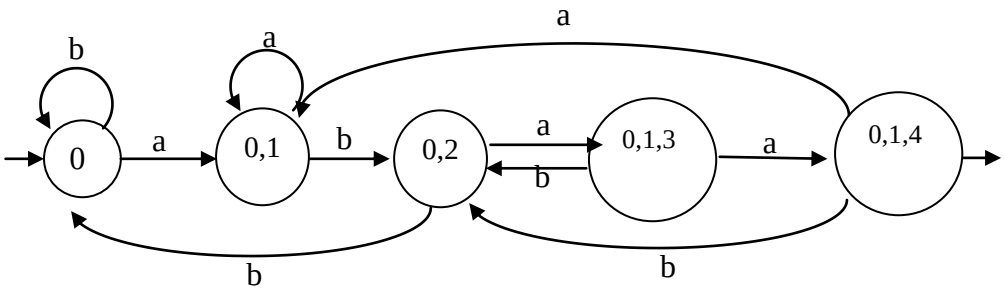
Etat	**a**	**b**
0	(0,1)	0
(0,1)	(0,1)	(0,2)
(0,2)	(0,1,3)	0
(0,1,3)	(0,1,4)	(0,2)
(0,1,4)	(0,1)	(0,2)

Explications
On commence par l'état initial qui dans ce cas particulier coïncide avec l'état initial de l'automate non déterministe 0, car notre automate non déterministe n'en a qu'un seul. (S'il en avait plusieurs, on les aurait regroupés pour former l'état initial de l'automate déterministe).

A partir de cet état 0, en ***a*** on va vers les états 0, 1 de l'automate initial. Pour l'automate déterministe, on les regroupe dans l'état qu'on appelle (0,1). En ***b***, on va vers l'état 0 de l'automate initial. Donc, 0 est lui aussi un état de l'automate déterminisé.

Chaque fois qu'on obtient un état de l'automate déterminisé, on le met dans la colonne de gauche pour agir sur cet état par les lettres de l'alphabet. Ainsi on obtient la deuxième ligne, où figure un nouvel état (0,2). On procède ainsi tant qu'il y reste des états non traités.
Les états finaux sont ceux qui contiennent un état final de l'automate initial. Dans notre cas, il n'y a qu'un seul : (0,1,4).

Voici l'automate déterminisé :



La démonstration montrant que ***D*** et ***C*** reconnaissent le même langage se fait par récurrence.

***

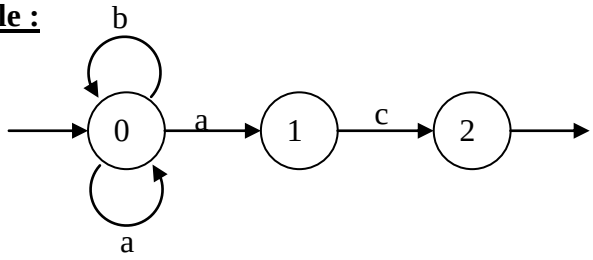
Un automate déterministe peut être ou ne pas être **complet.**

Automate déterministe complet (**définition**) :

Pour $\forall \boldsymbol{u} \in \boldsymbol{P}$ et $\forall \boldsymbol{a} \in A\ \exists \boldsymbol{v} \in \boldsymbol{P}$ tel que $\boldsymbol{u.a=v}$.
C'est-à-dire que de chaque état sortent des flèches avec toutes les étiquettes possibles.
L'automate ***D*** de l'exemple précédent est un automate complet.

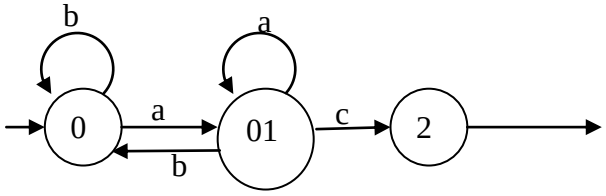
N'importe quel automate non déterministe est équivalent à un automate déterministe et complet : il suffit, dans l'automate déterminisé, d'introduire un état poubelle s'il n'est pas déjà complet.

Exemple :



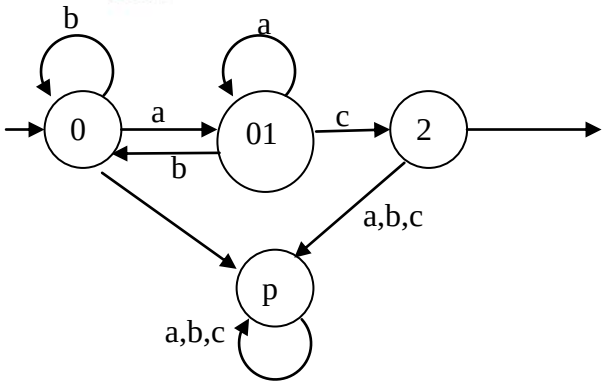
Déterminisation :

état	**a**	**b**	**c**
0	(0,1)	0	--
(0,1)	(0,1)	0	2
2	--	--	--



Cet automate déterministe n'est pas complet.
Remplaçons maintenant les traits (« pas de transition ») dans ce tableau par un nouvel état appelé état poubelle ou puits (P), qui a la propriété que toutes les transitions à partir de cet état reviennent sur lui-même :

état	a	b	c
0	(0,1)	0	P
(0,1)	(0,1)	0	2
2	P	P	P
P	P	P	P



Voici donc l'automate déterminisé complet. (Le fait que dans cet automate 0 est resté l'état initial, et le seul état terminal reste 2, est une coïncidence : cela ne se produit pas systématiquement).

Remarques

1. Il n'y a pas trop de sens de parler d'un automate non déterministe complet ou pas complet, même si cela n'est pas interdit et figure dans certains ouvrages. De même, même si introduire un « état poubelle » dans un automate non déterministe est possible et ne nuit pas à son fonctionnement, ceci n'est pas utile pour la déterminisation de cet automate. Normalement, on n'introduit un état poubelle si besoin est qu'une fois l'automate est devenu déterministe (ou directement lors de la déterminisation).
2. L'état poubelle étant un état non coassessible (il n'y a pas de chemin menant de cet état vers une sortie), la présence ou l'absence d'état poubelle ne change rien en ce qui concerne le langage reconnu par l'automate. Il n'empêche qu'on préfère travailler avec des automates déterministes complets (donc parfois contenant un état poubelle) à cause de certains opérations (minimisation, complémentarisation) qui figureront plus bas dans ce texte, et qui seraient difficiles à mener ou même donnant des résultats incorrectes si l'automate n'est pas complet.

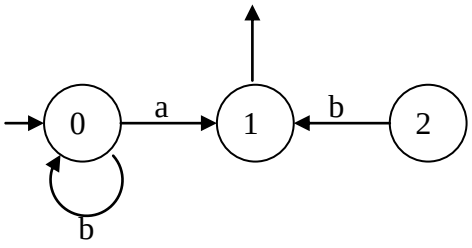
Définitions formelles d'un automate accessible, coaccessible, émondé:

1. Un automate est ***accessible*** si tous les états sont accessibles (à partir de l'état initial).
2. Un automate est ***coaccessible*** si tous les états sont coaccessibles (à partir de n'importe quel état on peut accéder à un état terminal).
3. Accessible + coaccessible = ***émondé.***
4. Un ensemble de mots ***X*** est un langage ***reconnaissable*** ssi il existe un automate fini ***D*** qui le reconnaît. **Ici, le mot « fini » est très important.**

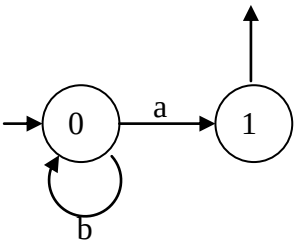
Commentaires :

1. Un automate non accessible semblerait au premier vu un objet exotique. Effectivement, on peut toujours couper l'état non accessible (et éventuellement toutes

les transitions qui y entrent) sans gêner au fonctionnement de l'automate. L'automate non accessible suivant :



est équivalent à l'automate accessible obtenu en coupant l'état 2 et la transition 0.c.2 :

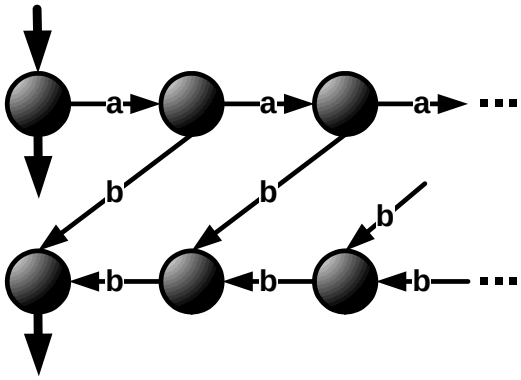


Néanmoins, la notion d'inaccessibilité n'est pas privée de sens. On peut obtenir des automates non accessibles lors des opérations dont on parlera plus bas, entre autres lors de l'obtention d'un automate reconnaissant le complément d'un langage.

2. On rencontre des automates non coaccessibles bien plus fréquemment. Notamment, tout automate déterministe complet contenant une poubelle, est non coassessible, car on ne peut pas atteindre une sortie à partir de la poubelle.

3. La notion d'un langage reconnaissable est intéressante car il existent bien des langages non reconnaissables. Par exemple, un langage consistant en tous les mots contenant autant de *a* que de *b,* n'est pas reconnaissable. Il en suit entre autres qu'un langage contenant des parenthèses qui doivent être bien placées (avec une parenthèse fermante pour chaque ouvrante et une bonne imbrication des parenthèses) n'est pas reconnaissable.

4. Un langage non reconnaissable peut parfois être reconnu par un automate ***non fini***. Exemple : le langage consistant en tous les mots du type $a^n b^n$ (les mots où il y a autant de *a* que de *b* (sans fixer le nombre) et où tous les *a* précèdent à tous les *b*) n'est pas reconnaissable. Or, il est évidemment reconnu par l'automate infini suivant :



5. Si ***X*** est un langage reconnaissable sur l'alphabet ***A***, on peut le reconnaître avec un automate déterministe et complet, car l'automate ***D*** figurant dans la définition du langage reconnaissable est toujours équivalent à un automate déterministe et complet ***F***.

   Ecrivant ***F=(Q, i, T, E)***, on a alors $w \in X$ ssi $i.w \in T$ et donc $w \notin X$ ssi $i.w \notin T$.

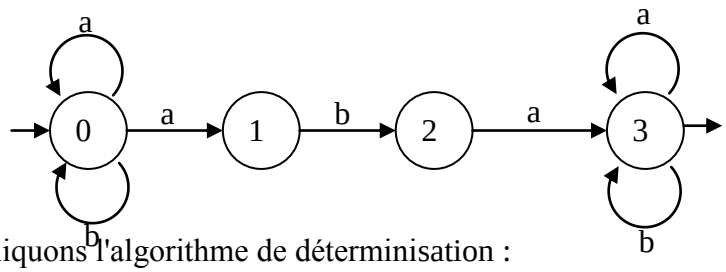
Le complément d'un langage reconnaissable.

Le complément d'un langage reconnaissable (l'ensemble de mots sur le même alphabet n'appartenant pas au langage en question) est encore reconnaissable.

Soit ***D*** un automate déterministe et complet reconnaissant le langage ***X***.
Pour construire un automate reconnaissant le complément de ***X***, il suffit de prendre comme ensemble d'états terminaux le complément de l'ensemble d'états terminaux de ***D***. Cette opération s'appelle dans certains ouvrages *complémentarisation* (à ne pas confondre avec l'opération de *complétion*, celle de rendre un automate complet !)

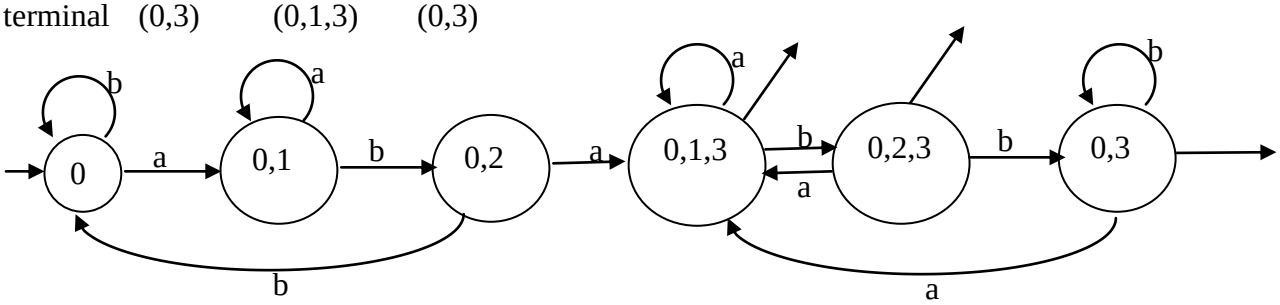
Exemple :

Construisons un automate sur $A=\{a,b\}$ qui reconnaît l'ensemble des mots n'ayant pas ***aba*** en facteur.
On commence par la construction d'un automate (non déterministe) reconnaissant tous les mots ayant ***aba*** en facteur.

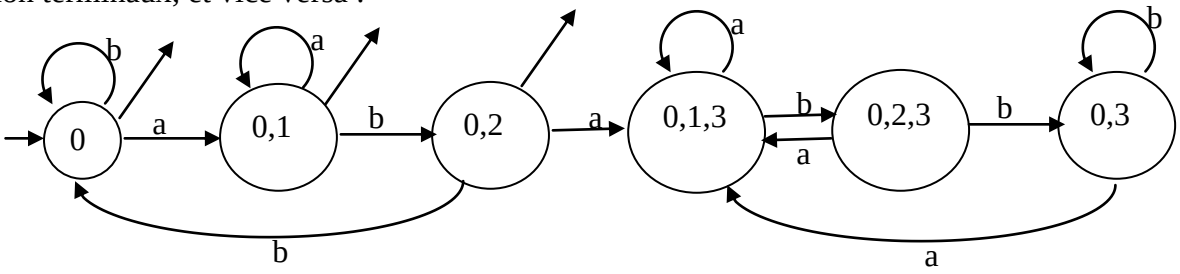


Appliquons l'algorithme de déterminisation :

	état	**a**	**b**
initial	0	(0,1)	0
	(0,1)	(0,1)	(0,2)
	(0,2)	(0,1,3)	0
terminal	(0,1,3)	(0,1,3)	(0,2,3)
terminal	(0,2,3)	(0,1,3)	(0,3)
terminal	(0,3)	(0,1,3)	(0,3)

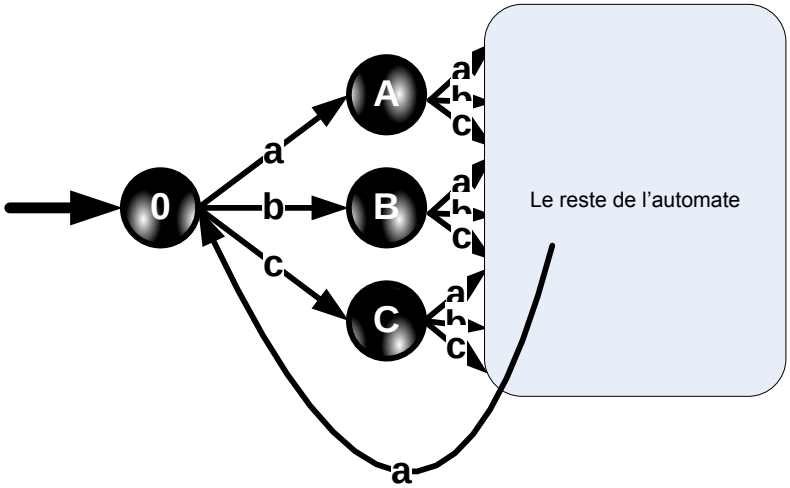


Cet automate déterministe et complet reconnaît tous les mots ayant ***aba*** en facteur. Il a trois états terminaux : (0,1,3), (0,2,3) et (0,3).
Maintenant pour obtenir un automate (lui aussi déterministe et complet) reconnaissant tous les mots qui n'ont pas ***aba*** en facteur, il suffit de transformer tous les états terminaux en des états non terminaux, et vice versa :

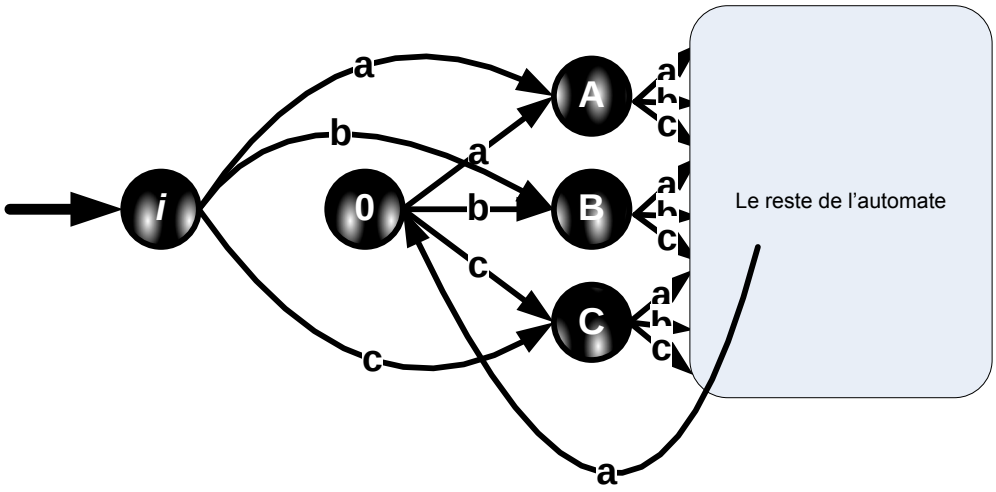


Remarque très importante :
Ici, on a obtenu un automate complet sans qu'il y ait besoin de le compléter en introduisant l'état poubelle. Evidemment, ceci n'est pas toujours le cas. **Si on a affaire à un automate non complet, on n'a pas le droit de remplacer directement les états terminaux par des états non terminaux et vice versa : il faut d'abord le compléter.** Alors l'état poubelle (qui ne peut pas être terminal) devient un état terminal de l'automate reconnaissant le complément du langage.

Remarque : déterminisation et standardisation
Le standardisé d'un automate déterministe non standard (comme un AD n'a qu'une seule entrée, cela veut dire qu'il a des transitions aboutissant dans l'état initial), est déterministe. Cela découle du fait que les transitions partant du nouvel état initial portent les mêmes étiquettes que celles qui partent de l'état initial de l'automate d'origine, or celles-ci sont toutes différentes. Ceci est illustré sur le schéma suivant, où on a choisi d'avoir affaire à un AD ne reconnaissant pas le mot vide (cela ne change rien dans le raisonnement) et où on a explicité les états directement liés à l'état initial (A,B,C). La flèche retournant vers 0 (avec l'étiquette *a*) est choisie juste comme un exemple.



En standardisant cet automate, on obtient :



qui reste un AD.

Minimisation

Pour tout langage reconnaissable il existe ***le plus petit*** (c.à.d. contenant le plus petit nombre d'états) automate déterministe qui le reconnaît, et il est ***unique*** : c'est l'automate minimal du langage.

Algorithme de minimisation par la méthode des équivalences est basé sur la notion de partitionnement de l'ensemble d'états de l'automate.

Principe

Tout d'abord, si l'automate à minimiser n'est pas complet, il faut lui ajouter l'état poubelle pour qu'il le devienne. Sinon on risque de faire une erreur grave, qui se manifeste facilement au cas d'un automate déterministe non complet dont tous les états sont des états terminaux (essayez le voir vous-mêmes après avoir compris l'algorithme de minimisation).

On sépare tous les états de l'automate déterministe initial en deux groupes : terminaux et non-terminaux. Puis, on analyse la table des transitions en marquant vers quel groupe va chaque transition. On repartitionne les groupes selon les patterns des transitions en terme de groupes. On répète ce processus en utilisant cette nouvelle partition, et on réitère jusqu'à ce qu'on arrive à ne plus pouvoir partitionner. Les groupes restants forment l'automate minimal. On appelle souvent les itérations les étapes.

Description du processus de partitionnement itératif

Soit
un automate fini déterministe complet $D=(Q, i, T, E)$
Résultat à obtenir: Un automate fini déterministe complet D' qui reconnaît le même langage que D et qui a aussi peu d'états que possible.

Méthode

1. Construire une partition initiale Θ_0 de l'ensemble des états avec deux groupes : les états terminaux T et les états non-terminaux $Q-T$.

2. Procédure applicable à la partition courante Θ_i, à commencer par Θ_0 :

POUR chaque groupe G de Θ_i **FAIRE**
début
partitionner G en sous-groupes de manière que deux états e et t de G soient dans le même sous-groupe ssi pour tout symbole $a \in A$, les états e et t ont des transitions sur a vers des états du même groupe de Θ_i ;

/* au pire, un état formera un sous-groupe par lui-même */

remplacer G dans par tous les sous-groupes ainsi formés ; on obtient la partition de l'étape suivant, Θ_{i+1} ;
fin

3. Si $\Theta_{i+1} = \Theta_i$ alors $\Theta_{final} = \Theta_i$ et continuer à l'étape 4. Sinon, répéter l'étape (2) avec Θ_{i+1} comme partition courante.

4. Choisir un état dans chaque groupe de la partition Θ_{final} en tant que représentant de ce groupe. Les représentants seront les états de l'automate fini déterministe réduit ***D'***.

   Soit ***e*** un état représentatif, et supposons que dans ***D*** il y a une transition $e \xrightarrow{a} t$.

   Soit ***r*** le représentant du groupe de ***t*** (***r*** peut être égal à ***t***).

   Alors ***D'*** a une transition de ***e*** vers ***r*** sur ***a*** ($e \xrightarrow{a} r$).

L'état initial de ***D'*** est le représentant du groupe qui contient l'état initial ***i*** de ***D*** ; les états terminaux de ***D'*** sont des représentants des états de ***T***.

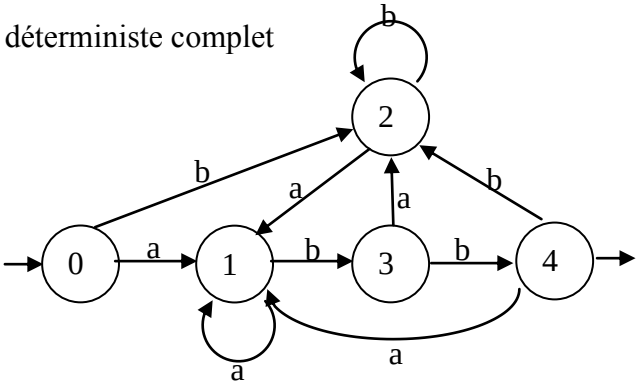
Pour tout groupe ***G*** de Θ_{final} , soit ***G*** est entièrement composé d'états de ***T***, soit ***G*** n'en contient aucun.

Remarque

En réalité, il est parfois plus facile, au lieu de choisir un représentant, marquer les groupes finaux comme tels, ou bien les renommer par, par exemple, A,B, C … , et remplacer dans les transitions tout état par le nom de son groupe :

Si $e \xrightarrow{a} r$, $e \in$A, $r \in$B où les groupes A,B$\in \Theta_{final}$, alors dans l'automate minimisé on a $A \xrightarrow{a} B$ où maintenant on considère A et B comme états de l'automate minimisé. Cette remarque deviendra tout à fait claire à la fin de l'exemple suivant.

Exemple 1. Minimisons l'automate déterministe complet



décrit par le tableau de transitions suivant :

état	**a**	**b**
0	1	2
1	1	3
2	1	2
3	2	**4**
4	1	2

L'unique état terminal : 4.

Donc, la partition initiale : $\Theta_0 = \{(0, 1, 2, 3), (4)\}$

Notons les groupes non terminal et terminal : I = {(0, 1, 2, 3)} et II = {(4)}

On ne peut pas, évidemment, essayer de séparer le groupe II qui consiste déjà en un seul état. On regarde donc dans quel groupe tombent les transitions à partir des états du groupe I :

état	**a**	**b**
0	I	I
1	I	I
2	I	I
3	I	II

Donc, le groupe I se sépare en deux, et on a Θ_1= {(0, 1, 2}, (3), (4)}.

Notons les groupes (en recyclant la notation) : I = {(0, 1, 2)}, II = {(3)}, III = {(4)}.

Maintenant essayons de séparer le groupe I en regardant où tombent les transitions à partir de ses états dans les termes de la séparation {(0, 1, 2}, (3), (4)}.

état	**a**	**b**
0	I	I
1	I	II
2	I	I

Donc, le groupe I se sépare en deux, et on a Θ_2= {(0, 2}, (1), (3), (4)}.

Notons les groupes (en recyclant la notation) : I = {(0, 2)}, II = {(1)}, III = {(3)}, III = {(4)}.

Essayons de séparer le groupe I en regardant où tombent les transitions à partir de ses états dans les termes de la séparation {(0, 2}, (1), (3), (4)}.

état	**a**	**b**
0	II	I
2	II	I

Le groupe I ne se sépare pas, Θ_3 reste identique à la séparation de l'étape précédent : $\Theta_3 = \Theta_2$ ={(0, 2}, (1), (3), (4)}.

Fin de la procédure itérative de séparation.

Voici les itérations qu'on a effectuées :

Etape 1 : $\Theta_{courant}$= {(0, 1, 2, 3), (4)}
Θ_{new}= {(0, 1, 2}, (3), (4)}

Etape 2 : $\Theta_{courant}$ = {(0, 1, 2}, (3), (4)}
Θ_{new}= {(0, 2}, (1), (3), (4)}

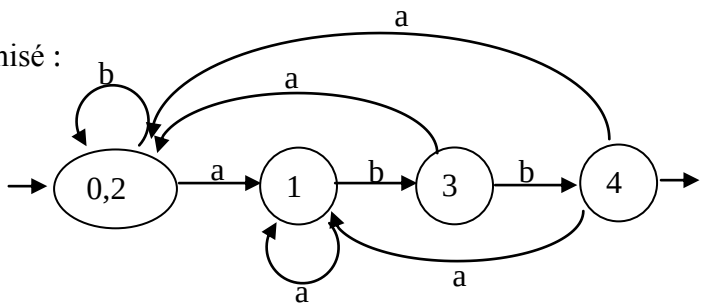
Etape 3 : $\Theta_{courant}$ = {(0, 2}, (1), (3), (4)}
Θ_{new}= {(0, 2}, (1), (3), (4)}= $\mathbf{\Theta_{final}}$

Passons aux représentants :

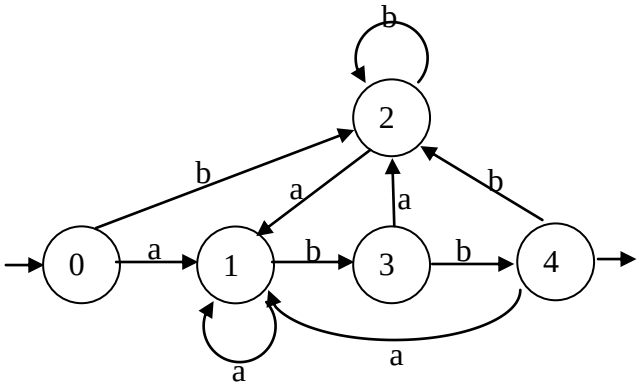
groupe	**représentant**
I	0 ou 2
II	1
III	3
IV	4

Il devient clair maintenant qu'on peut marquer les états de l'automate minimisé soit par un représentant, soit par les chiffres I, II, III, IV suivant le groupe du dernier étape, soit par le contenu du groupe (dans ce cas, l'état initial sera marqué (0,2) ; cette dernière solution est pratique tant que le groupe consiste en peu d'états).

L'automate minimisé :



On peut maintenant poser la question suivante : pourquoi les états 0 et 2 ont resté ensemble après la minimisation ? Qu'y a-t-il de spécial concernant ces deux états par rapport aux autres ? Regardons à nouveau l'automate initial :



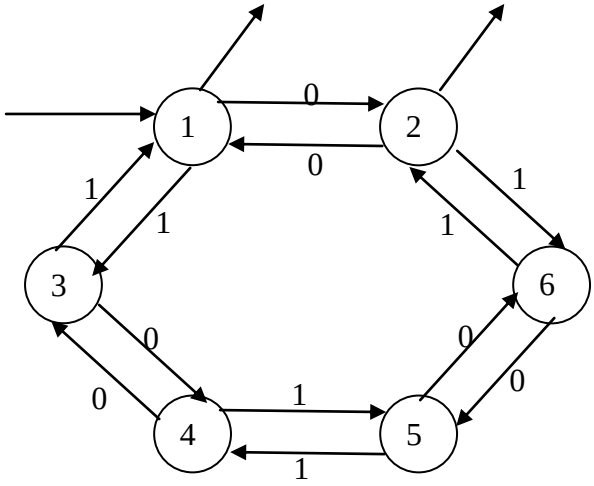
et introduisons la terminologie suivante :

on dit que la chaîne ***w*** **distingue** l'état ***s*** de l'état ***t*** si quand on commence dans l'état ***s*** et lit ***w*** on arrive dans un état terminal, et si on commence dans ***t*** et lit ***w*** on arrive dans un état non terminal ou vice-versa.

Ici, tous les états peuvent être distingués par une chaîne ou une autre, sauf les états 0 et 2. C'est facile à voir : et à partir de 0, et à partir de 2 on arrive en 1 en lisant un nombre quelconque (y compris zéro) de ***b*** et un ***a*** ; puis, comme on est dans le même état (1), il nous restent que les mêmes chaînes pour arriver à l'état final (ici, il y en a un seul). Donc, les états non distinguables se fondent dans un même état (en l'occurrence (0,2)) de l'automate minimisé.

En fait, on peut montrer que l'algorithme de minimisation qu'on a expliqué, est basé sur la recherche des tous les groupes qui peuvent être distingués par une chaîne ou une autre.

Exemple 2 Voici un automate déterministe complet avec deux états terminaux, avec l'alphabet A={0,1} :



La même procédure de minimisation donne :
I = {1} T = {1, 2}
Θ_0={(1,2), (3, 4, 5, 6)}={I, II}
Θ_1= {(1, 2), (3, 6), (4, 5)}={I, II, III} (en recyclant les chiffres romains)
$\Theta_2=\Theta_1$

Voici l'automate minimisé :