Chapitre 1 Automates finis

1. *Introduction : machine de Turing*

L'informatique est une discipline assez récente. Il est difficile de déterminer le moment exact de sa naissance car cela dépend du point de vue ; mais tout le monde est d'accord qu'un des évènements clé c'est l'invention de la **machine de Turing** : la « machine idéale » décrite par Alan Turing en 1936.

La machine de Turing est une construction essentiellement mentale, même si on peut facilement la réaliser mécaniquement ou électroniquement. C'est une construction très simple, mais qui, selon la thèse Church-Turing, peut résoudre tout problème algorithmique, c.à.d. elle peut effectuer tout calcul basé sur une procédure algorithmique. On peut dire que tout ce que peut faire un ordinateur « normal », peut – plus lentement, il est vrai – être fait par la machine de Turing.

Alan Turing a crée sa machine en vue de 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.

Définition

Voici comment est organisée une machine de Turing (version d'origine) :

1. Il y a un « ruban » divisé en cases consécutives. Chaque case contient un symbole parmi un alphabet **fini**. L'alphabet contient un symbole spécial « blanc » (dans les exemples qui suivent, c'est '0') et d'autres symboles. Le ruban est supposé être de longueur **infinie** vers la gauche ou vers la droite, en d'autres termes la machine doit toujours avoir assez de longueur de ruban pour son exécution. Les cases non encore écrites du ruban contiennent le symbole « blanc ». ***Le ruban constitue une mémoire potentiellement infinie.***
2. Il y a une « tête de lecture/écriture » qui peut lire et écrire des symboles sur le ruban, et se déplacer d'un cran vers la gauche ou vers la droite du ruban.
3. Il y a un « registre d'état » qui mémorise **l'état courant** de la machine de Turing parmi une liste **finie** d'états. Il existe un état spécial appelé « état de départ » qui est l'état initial de la machine avant son exécution. On peut aussi introduire un état spécial d'arrêt, ce qui facilite une représentation de la machine de Turing sous forme d'un diagramme de flux. ***L'état courant forme un élément de la mémoire finie.***
4. Il y a une « table d'actions » qui, pour chaque couple (symbole lu, état courant) indique à la machine quel symbole écrire dans la case où se trouve la tête (au lieu du symbole qui vient d'être lu), puis comment déplacer la tête de lecture ('G' pour une case vers la gauche, 'D' pour une case vers la droite), et quel sera le nouvel état de la machine. Certaines combinaisons d'un symbole lu et d'un état courant n'indiquent pas une action « écriture puis déplacement », mais signifient que la machine s'arrête. ***La table d'action forme le reste de la mémoire finie.***

Exemple

Description de la tâche

La machine de Turing qui suit possède un alphabet {'0', '1'}, '0' étant le « blanc ». On suppose que le ruban contient une série de '1', et que la tête de lecture/écriture se trouve initialement au-dessus du '1' le plus à

gauche. Cette machine a pour effet de doubler le nombre de '1', en intercalant un '0' entre les deux séries. Par exemple, « 11 » devient « 11011 ».

Description de la machine qui résout ce problème :

L'ensemble d'états possibles de la machine est {e1, e2, e3, e4, e5} et l'état initial est e1.
La table d'actions est la suivante :

Exemple de table de transition

Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état
e1	0	*aucun*	*aucun*	arrêt
	1	0	Droite	e2
e2	1	1	Droite	e2
	0	0	Droite	e3
e3	1	1	Droite	e3
	0	1	Gauche	e4
e4	1	1	Gauche	e4
	0	0	Gauche	e5
e5	1	1	Gauche	e5
	0	1	Droite	e1

L'exécution de cette machine pour une série de deux '1' serait (la position de la tête de lecture/écriture sur le ruban est inscrite en caractères gras et rouges) :

Exécution (1)

Étape	État	Ruban
1	e1	**1**1
2	e2	0**1**
3	e2	01**0**
4	e3	010**0**

Exécution (2)

Étape	État	Ruban
5	e4	010**1**
6	e5	01**0**1
7	e5	**0**101
8	e1	**1**101

Exécution (3)

Étape	État	Ruban
9	e2	100**1**
10	e3	1001**0**
11	e3	10010**0**
12	e4	10011

Exécution (4)

Étape	État	Ruban
13	e4	1001**1**
14	e5	100**1**1
15	e1	**1**1011
	(Arrêt)	

Le comportement de cette machine peut être décrit comme une boucle :

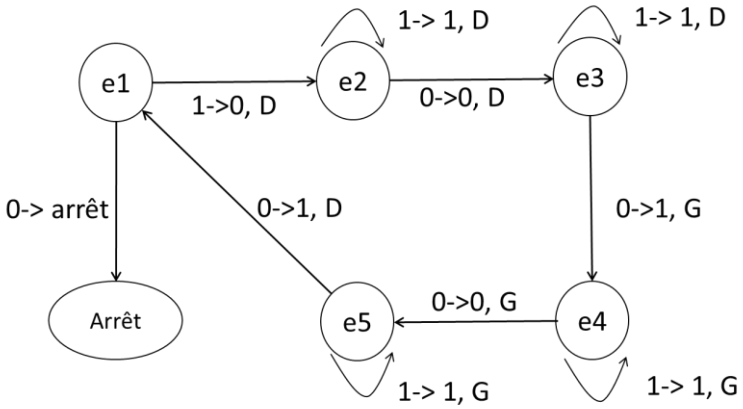
- Elle démarre son exécution dans l'état e1, remplace le premier 1 par un 0.
- Puis elle utilise l'état e2 pour se déplacer vers la droite, en sautant les 1 (un seul dans cet exemple) jusqu'à rencontrer un 0 (ou un blanc), et passer dans l'état e3.
- L'état e3 est alors utilisé pour sauter la séquence suivante de 1 (initialement aucun) et remplacer le premier 0 rencontré par un 1.
- L'état e4 permet de revenir vers la gauche jusqu'à trouver un 0, et passer dans l'état e5.
- L'état e5 permet ensuite à nouveau de se déplacer vers la gauche jusqu'à trouver un 0, écrit au départ par l'état e1.

- La machine remplace alors ce 0 par un 1, se déplace d'une case vers la droite et passe à nouveau dans l'état e1 pour une nouvelle itération de la boucle.

Ce processus se répète jusqu'à ce qu'e1 tombe sur un 0 (c'est le 0 du milieu entre les deux séquences de 1) ; à ce moment, la machine s'arrête.

Représentation alternative 1 : un graphique

Le fonctionnement de cette machine peut aussi être représenté par un diagramme :



Description alternative 2 : pseudocode

On peut aussi décrire le fonctionnement de la même machine en utilisant un pseudolangage :

```
e1.   if read 0, goto STOP  // donc cette machine particulière, pour fonctionner, DOIT avoir 1 dans la première case;
                            elle ne peut pas détecter le début d'une séquence de 1 toute seule
      if read 1, write 0, go right, goto e2    // remplacer 1 par 0
e2.   if read 0, write 0, go right, goto e3
      if read 1, write 1, go right, goto e2     //repeat
e3.   if read 0, write 1, go left, goto e4
      if read 1, write 1,  go right,  goto e3     //repeat
e4.   if read 0, write 0, go left, goto e5
      if read 1, write 1,  go left,  goto e4     //repeat
e5.   if read 0, write 1, go right, goto e1
      if read 1, write 1, go left, goto e5      //repeat
```

***

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

Ces travaux amenèrent à distinguer entre les fonctions calculables et non calculables : toute fonction calculable peut être calculée par la machine de Turing en temps fini, le temps mesuré comme nombre d'opérations.

> 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).

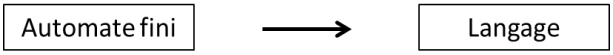
La machine est actuellement exposée dans le hall du département d'informatique de l'Université d'Heidelberg en Allemagne.

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.

2. *Automates finis*

Description informelle et exemple

Un modèle plus simple mais fort utile inspiré par la machine de Turing est celui d'un **automate fini**. Cette machine est destinée à distinguer des **mots** fournis en entrée en mots **reconnus** et **non reconnus**, donc pour chaque automate fini il existe un ensemble de mots reconnus appelé **langage :**



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 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. **L'automate est dit « fini » car il possède un nombre fini d'états distincts** et il ne possède aucune autre mémoire : il n'y a pas de ruban. Il ne dispose donc que d'une mémoire bornée, indépendamment de la taille de la donnée sur laquelle on effectue les calculs.
Pour décider si l'automate reconnait un mot donné, il passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée.

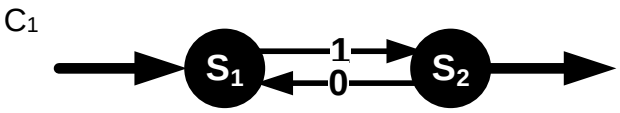
Généralisations possibles

On peut très bien considérer des automates au nombre d'états potentiellement infini : la théorie qui en résulte est très analogue à la théorie habituelle. Dans notre cours, on ne parle que des automates finis.

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. 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) consistant en 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).

Voici un exemple très simple d'un automate fini :



Les deux états sont marqués par des cercles, les transitions par des flèches libellées par des caractères, l'entrée et la sortie par des flèches non libellées. Dans l'exemple ci-dessus, pour l'entrée 1010101 (la lecture se fait à partir de la gauche), et si l'automate démarre en s_1, il passe successivement par les états $s_1s_2s_1s_2s_1s_2s_1s_2$, le calcul correspondant est

$$s_1 \xrightarrow{1} s_2 \xrightarrow{0} s_1 \xrightarrow{1} s_2 \xrightarrow{0} s_1 \xrightarrow{1} s_2 \xrightarrow{0} s_1 \xrightarrow{1} s_2$$

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 $s_1 \xrightarrow{0} s_2$; il ne sera pas reconnu non plus.

Représentations d'automates finis

1. Un automate fini peut être vu comme un graphe orienté étiqueté : les états sont les sommets et les transitions sont les arcs 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.
2. Une autre façon commode de représenter un automate fini est sa *table de transitions*. 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	--

3. On peut aussi représenter un automate fini en énumérant (i) ses états, (ii) l'état ou les états d'entrée, (iii) l'état ou les états de sortie, (iv) les transitions, (v) l'alphabet .
   Cette représentation peut servir pour donner une définition de l'automate fini de façon plus formelle.

Quelques définitions

Pour définir formellement 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 n'aura affaire qu'à des mots de longueur finie, mais comme nous avons déjà mentionné, 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 de 4 caractères $\{a,b,c,d\}$).

- **Le mot vide** est noté par ε (la seule notation utilisée dans notre cours) ou, dans certains ouvrages, par ***1***. C'est le seul mot de longueur 0. On précisera les propriétés exactes du mot vide plus tard.

  On note ***A**** **l'ensemble de tous les mots** sur l'alphabet ***A*,** plus le mot vide.
  C'est une définition très importante, à retenir par cœur.

  Attention* : *A est un alphabet, il consiste en un nombre de caractères fini. ***A**** est un **ensemble infini de mots. Ne jamais les confondre !**

- **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,

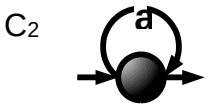
- $E \subset Q \times (A \cup \{\varepsilon\}) \times Q$ est un ensemble de triplets ***(p.a.q)*** appelés les flèches ou les transitions de l'automate. On note parfois $p \xrightarrow{a} q$, ce qui est équivalent à ***(p.a.q)*.**

  Souvent, on préfère dénoter 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 dits 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. Il reconnaît des séquences consistant en n'importe quel nombre de *a*, donc il reconnaît ***A**** pour l'alphabet ***A***={*a*} :



Comparaison avec la machine de Turing, résumé :

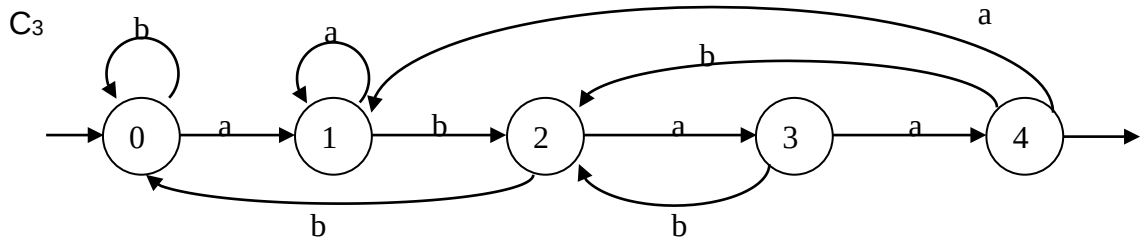
Une machine de Turing possède une mémoire potentiellement infinie : le ruban. Il y a aussi un élément fini de la mémoire : l'ensemble d'états et de transitions, ainsi que l'alphabet. À l'arrêt, le contenu de la mémoire ruban est modifié.

Un automate fini possède une mémoire finie : l'ensemble d'états et de transitions. Les mots fournis à l'automate, à la différence du texte sur le ruban, ne font pas partie de la mémoire, car ils ne peuvent pas être modifiés. À la fin du processus, le mot est reconnu ou non reconnu, mais non pas modifié.

L'interprétation intuitive « temporelle »:

À 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)*** appartienne à l'ensemble de transitions ***E*** (il existe une flèche ***(p.a.q)***).

Exemple :



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

La lecture du mot ***w*** se termine dans l'état 4 ssi ***w*** se termine par ***abaa***. (À cet instant du cours, il est facile de montrer que ***seuls*** des mots se terminant par ***abaa*** sont reconnus, mais bien plus difficile de montrer que ***tous*** des mots se terminant par ***abaa*** sont reconnus ; or, c'est vrai).

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 :

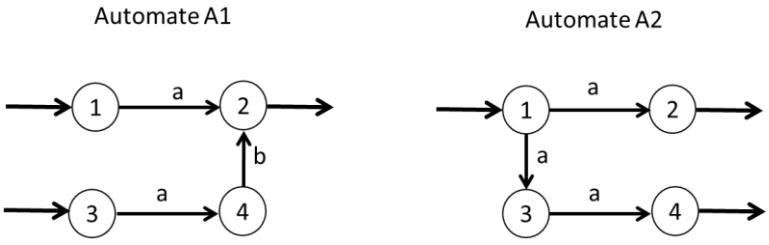
Si vous regardez cet automate de près, vous verrez que nous avons libellé les états de telle façon que l'automate se trouve dans l'état numéro ***i*** ssi le plus long **suffixe du mot déjà lu** qui est en même temps un **préfixe de *abaa*,** est de **longueur *i***.

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 (toujours dans l'approche temporelle) :

L'automate prend un mot (constitué de symboles de son alphabet) en entrée et démarre dans son état initial ou dans un de ses états initiaux. 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.

L'insuffisance de l'approche temporelle :



Prenons l'automate A1. Il y a deux entrées : les états 1 et 3. On considère un mot commençant par *a*. Où est-ce qu'on entre ? En 1 ou en 3 ? Le mot 'a' est reconnu sur la branche 12, et non reconnu sur la branche 342. Est-il reconnu ? Quelle est la séquence d'états à considérer pour un mot commençant par *a* ?

Prenons l'automate A2. Il a une seule entrée, mais de l'état 1, le mot commençant par *a* peut aller soit vers 2, soit vers 3. Comment choisit-il où aller ? Le mot 'a', reconnu sur la branche 12 et non reconnu sur la branche 134, est-il reconnu par l'automate ?

On voit qu'il y a un problème avec la représentation du fonctionnement d'un automate fini comme processus se déroulant dans le temps.

Nous verrons que la réponse à ce problème est double : (i) il s'agit ici d'un type d'automates finis (automates non déterministes) qu'on réussira à éviter en les transformant dans des automates finis ne souffrant pas de ce type d'ambigüité (automates déterministes), (ii) mais on peut définir ce que veut dire la phrase « l'automate reconnait un tel mot » de façon à ce qu'elle soit claire quel que soit le type d'automate. Seulement, le fonctionnement de l'automate ne pourra plus être considéré comme «se déroulant dans le temps ».

Chemin : définition

- Nous avons vu la définition d'une transition : pour $a \in$ A, il existe une transition $p \xrightarrow{a} q$ (une flèche libellée par a allant de l'état p à l'état q) ssi $\boldsymbol{(p.a.q)} \in \boldsymbol{E}$ (où $\boldsymbol{E}$ est l'ensemble de transitions).

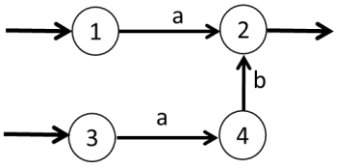
- Nous pouvons généraliser cette définition en introduisant la notion de **chemin** de façon récursive :

- Une transition $p \xrightarrow{a} q$, appartenant à l'ensemble de transitions $\boldsymbol{E}$, peut être considérée comme un chemin de longueur 1 entre les états p et q marquée par le mot $\boldsymbol{u}$ ='a' de longueur 1.
- Ensuite, pour $\boldsymbol{u} \in \boldsymbol{A^*}$ et $\boldsymbol{a} \in \boldsymbol{A}$ (rappel : donc $\boldsymbol{u}$ est un mot et $\boldsymbol{a}$ un caractère) on note $p \xrightarrow{ua} q$ ssi $\exists\, \boldsymbol{r} \in \boldsymbol{Q}$ tel que $p \xrightarrow{u} r$ et $r \xrightarrow{a} q$.
- De cette façon, on obtient la notion de **chemin** : $p \xrightarrow{w} q$ est un chemin de $\boldsymbol{p}$ à $\boldsymbol{q}$ d'étiquette $\boldsymbol{w}$. 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 $\boldsymbol{u,v} \in \boldsymbol{A^*}$ on a $p \xrightarrow{uv} q$ ssi $\exists\, \boldsymbol{r} \in \boldsymbol{Q}$ tel que $p \xrightarrow{u} r$ et $r \xrightarrow{a} q$. La preuve s'obtient facilement par récurrence.

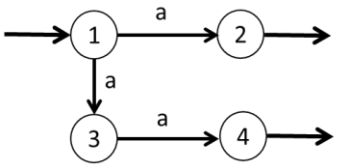
  Ayant défini la notion de chemin, on peut donner une définition de ce que veut dire la reconnaissance d'un mot par un automate fini qui ne fait pas appel à une séquence temporelle :
  Definition :
- Un mot $\boldsymbol{w} \in \boldsymbol{A^*}$ est reconnu par l'automate fini s'il existe un chemin $i \xrightarrow{w} t$ où $\boldsymbol{i} \in \boldsymbol{I}$ et $\boldsymbol{t} \in \boldsymbol{T}$, c.à.d. qu'en partant d'un état initial et en lisant le mot $\boldsymbol{w}$, on atteint un état terminal à la fin de la lecture.

Maintenant nous pouvons répondre aux questions posées tout à l'heure.

Automate A1



Automate A2



L'automate A1 reconnait deux mots : '*a*' et '*ab*', car il existent les chemins $1 \xrightarrow{a} 2$ et $3 \xrightarrow{ab} 2$, où 1 et 3 sont des états initiaux et 2, l'état terminal.

L'automate A2 reconnait deux mots : '*a*' et '*aa*', il existent les chemins $1 \xrightarrow{a} 2$ et $1 \xrightarrow{aa} 4$, où 1 est l'état initial et 2 et 4, des états terminaux.

Automates déterministes et non déterministes

Ces deux automates-ci sont des automates **non déterministes**.

- A1 l'est parce qu'il y a deux entrées.
- A2 l'est parce qu'il y a deux flèches libellées du même caractère ***a*** sortant du même état 1.

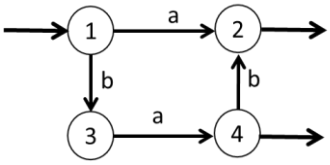
Pour ce genre d'automates, la seule façon raisonnable de juger si l'automate reconnait un mot particulier est de chercher s'il existe un chemin libellé par ce mot allant d'une entrée vers une sortie.

Un automate est **déterministe** si

- Il a une seule entrée
- Il n'y a pas d'états d'où sort plus d'une flèche libellée par le même caractère.

Par exemple, voici un automate déterministe C_4 :

C_4



Cet automate a une entrée. De l'état 1, sortes deux flèches marquées par deux caractères différents. Des états 3 et 4, sort une flèche. De l'état 2, aucune. La présence de deux sorties ne gêne pas au fait que l'automate soit déterministe.

Pour cet automate, on peut trouver les mots qu'il reconnait de la même façon, en cherchant des chemins joignant l'entrée et une sortie ($1 \xrightarrow{a} 2$, $1 \xrightarrow{ba} 4$, $1 \xrightarrow{bab} 2$), donc L={ *'a', 'ba', 'bab'*}.

Mais on peut aussi adopter une approche « temporelle » : par exemple, pour le mot 'ba', l'automate lit la première lettre ('*b*'), qui l'envoie à l'état 3, puis en lisant la seconde lettre ('*a*'), il va à l'état 4, c'est la fin du mot et c'est un état de sortie ; le mot est donc reconnu.

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 $q \in Q$ tel que $(p.a.q)$, 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 $q \in Q$ tel que $p \xrightarrow{u} q$.

(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 $u, v \in A^*$:

$(p.u).v=p.(uv)$ (concaténation des mots u, v)

Définition :

Le **langage L reconnu** par l'automate fini est l'ensemble de tous les mots reconnus.
Par exemple, le langage de l'automate A1 est L_{A1}={ *'a', 'ab'*}.
Un langage peut être fini ou infini.

* * *

Exercice :

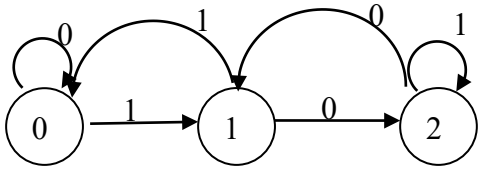
Construire un automate fini reconnaissant les entiers écrits en base 2 divisibles par 3.

Solution :
D'abord, quelques considérations vraies indépendamment de notre désir de construire un automate.

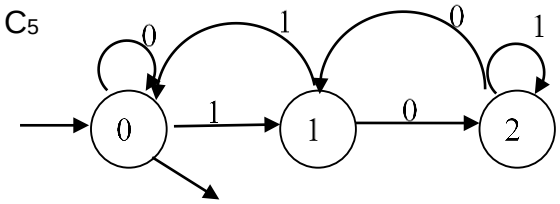
- Vis-à-vis de la division entière par 3, tous les nombres sont divisés en 3 classes qui correspondent aux trois restes possibles de cette division entière : 0 (les nombres divisibles par 3), 1 et 2. Appelons ces trois classes « 0 », « 1 » et « 2 ».
- Ajout d'un 0 à la fin d'un nombre binaire le multiplie par 2.
- Ajout d'un 1 à la fin d'un nombre binaire le multiplie par 2 et lui ajoute 1. Regardons ce qui se produit à des nombres appartenant à chacune des trois classes quand on ajoute 0 ou 1 à la fin de leur écriture binaire.

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

Nous pouvons dessiner ces transitions à l'ajout d'un chiffre 0 ou 1 à la fin :



Il suffit maintenant d'assimiler les classes à des états et assimiler l'ajout d'un chiffre à la fin de l'écriture binaire à la lecture du caractère suivant d'un mot sur l'alphabet A={0,1} pour que ce graphique devienne un graphique d'un automate à trois états. Mais pour terminer cette transformation, il faut choisir des entrées et des sorties. Commençons par la sortie : on veut que l'automate reconnaisse uniquement des nombres de classe 0, donc la sortie est sur l'état 0. Maintenant il est facile de voir où on doit entrer : le nombre binaire 0 (ainsi que 00, 000…) est de classe 0 (divisible par 3), donc on doit entrer dans l'état 0. Voici le résultat :



état	0	1
E/S		
0	0	1
1	2	0
2	1	2

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, notre 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 est donnée dans la section suivante.

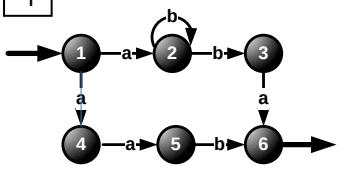
Automates finis standards

On introduit la notion d'un automate fini **standard**. C'est un automate fini, déterministe ou non déterministe,

- qui a une seule entrée
- et aucune transition n'aboutit dans cette entrée.

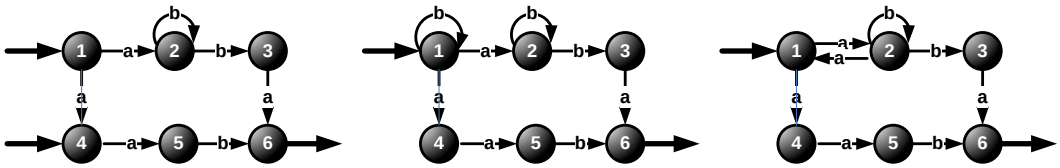
Par exemple, l'automate 1 ci-dessous est standard :

1



Les trois automates 2,3 et 4 ci-dessous ne sont pas standards :

2 3 4



L'automate 2 a deux entrées.
L'automate 3 a une transition (boucle) aboutissant à l'entrée : 1b1.
L'automate 4 a une transition aboutissant à l'entrée : 2a1.

Définition formelle :

L'automate $C=(Q, I, T, E)$ sur le langage A est standard si

$I=\{i\}$ // une seule entrée

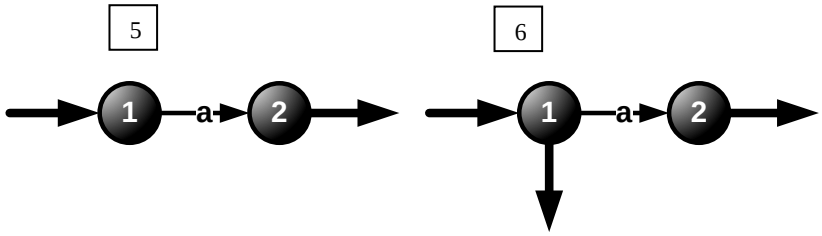
et pour tout $a \in A$ et tout $q \in Q$, $(q.a.i) \notin E$ // aucune transition ne va vers l'entrée

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.

Exemple :

5 6



Les deux automates 5 et 6 standards.
L'automate 5 ne reconnait pas le mot vide : il a plus d'un état, et l'état initial 1 n'est pas terminal.
L'automate 6 reconnait le mot vide car son état initial est terminal.

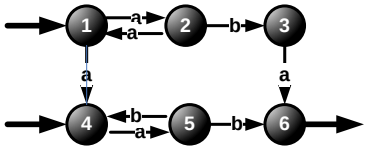
Algorithme de standardisation

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

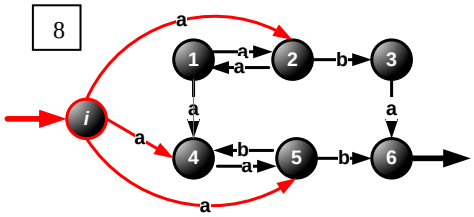
Pour *standardiser*, on ajoute à l'automate ***C*** un nouvel état qui devient l'unique état initial et duquel, pour chacune des transitions qui partaient des anciens états initiaux, part une transition de même étiquette vers les mêmes états.

Exemple :

7



8



L'automate 7 n'est pas standard (deux entrées, et les transitions 1a2, 2a1, 5b4 aboutissant dans des états initiaux). Pour le rendre standard,

- on ajoute un nouvel état initial ***i,*** qui n'est pas terminal parce que ni 1, ni 4 ne l'étaient
- en gardant toute la structure de l'automate 7 sauf les flèches d'entrée (car 1 et 4 ne seront plus initiaux),

• on liste toutes les transitions partant des états initiaux de l'automate 7	on crée la liste des transitions partant de l'état ***i*** de l'automate 8 en remplaçant l'état initial de la liste gauche par ***i.***
1.a.2	i.a.2
1.a.4	i.a.4
4.a.5	i.a.5

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

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

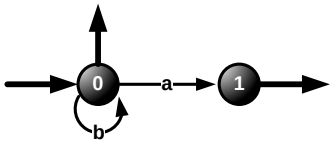
En mots : on ajoute un nouvel état ***i,*** et on ajoute des flèches vers tous les états (et qui portent les mêmes étiquettes) vers lesquels l'automate ***C*** possède des transitions partant de ces états initiaux. (En disant « tous les états », on y inclut éventuellement les états initiaux eux-mêmes). 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.

À quoi bon ?

Modification de la reconnaissance du mot vide

Un automate (n'importe quel) reconnait le mot vide uniquement si un ou plus de ses états initiaux est terminal.
Imaginons qu'on veule éliminer cette reconnaissance du mot vide, en gardant le reste du langage.
Il aurait fallu que le nouvel automate n'ait pas de flèche de sortie sur l'entrée où elle figurait ; supprimer cette sortie n'élimine pas que le mot vide du langage de l'automate, ceci élimine d'autres mots aussi.

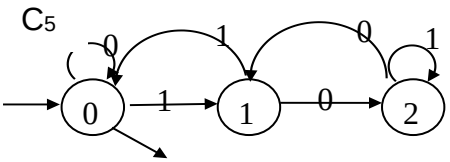
Prenons par exemple cet automate simple non standard :



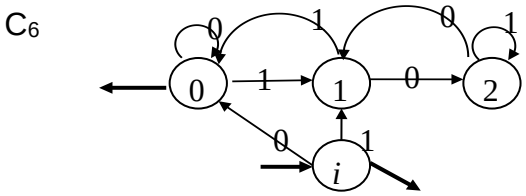
Il reconnait le mot vide, le mot '*a*', tous les mots qui consistent en n'importe quel nombre de *b* suivis d'un seul *a* et tous les mots consistant en n'importe quel nombre de *b*: $L=b^*a + b^*$. Éliminons la flèche de sortie sut l'état 0. L'automate qui en sort reconnaitra uniquement le mot '*a*' et tous les mots qui consistent en n'importe quel nombre de *b* suivis d'un seul *a* : $L'=b^*a$. À part le mot vide, nous avons éliminé la reconnaissance de tous les mots consistant en un nombre de *b*.
Ceci est dû au fait qu'il y a des mots qui se terminent à l'état 0, à cause de la présence de la boucle en *b*.
Pour qu'il n'y ait pas de mots (autres que le mot vide) se terminant à l'état d'entrée, il faut qu'il n'y ait pas de flèche aboutissant à l'état d'entrée.
C'est pourquoi avec un automate standard nous pouvons facilement jouer sur la reconnaissance du mot vide. Si son état d'entrée n'est pas terminal, il ne le reconnait pas, mais il suffit d'en faire un état terminal pour ajouter le mot vide au langage. Si son état d'entrée est terminal, il reconnait le mot vide, et il suffit d'enlever cette flèche de sortie pour soustraire juste le mot vide du langage.

Revenons vers l'exemple de l'automate qui reconnait les nombres en binaire divisibles par 3.

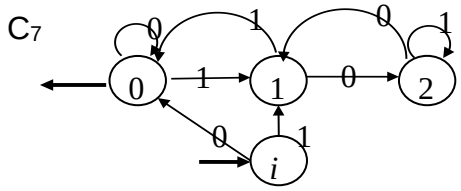
Nous avions obtenu l'automate :



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 (il reconnait exactement les mêmes mots). 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.

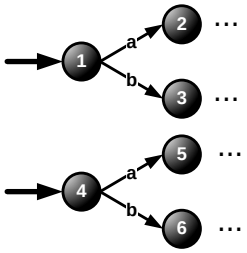
Remarques générales

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.

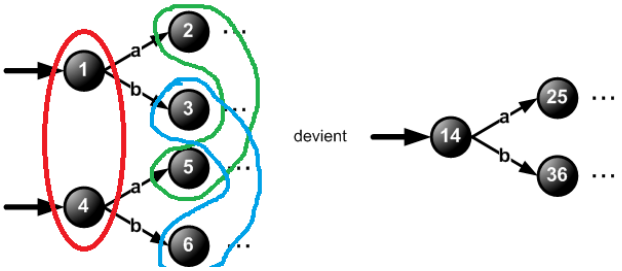
2. Nous illustrons nos idées par des exemples assez simples. Cela aide à la compréhension, mais parfois cela peut donner des idées fausses. Notamment, on pourrait se demander pourquoi on doit développer des algorithmes formels tandis que très souvent la réponse « se voit » immédiatement à partir du dessin. 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.

Déterminisation

Soit nous avons un automate fini non déterministe (AND). Il peut avoir plus d'une entrée, ou il peut y avoir des états d'où sortent deux ou plusieurs flèches portant la même étiquette, ou les deux :



Nous voulons obtenir un automate déterministe (AD) reconnaissant les mêmes mots. Ceci veut dire que chaque mot sur l'alphabet de l'automate « sait » où il entre (une seule entrée) et « sait » où il se termine (chaque chaîne de caractères aboutit dans un seul état, même si deux chaînes différentes peuvent aboutir dans un même état). Nous sommes obligés donc d'unifier les états initiaux 1 et 4 en un seul état initial que nous appellerons 14, et puis, le mot 'a' doit se terminer dans un seul état : nous unifierons donc 2 et 5 en un état composé 25 où aboutit le mot 'a', et les états 3 et 6, dans un état composé 36 où aboutit le mot 'b' :



Si on représente l'automate par une table de transitions, on obtient

	automate non déterministe					automate déterministe		
		a	b				a	b
E	1	2	3		E	14	25	36
	2	…	…			25	…	…
	3	…	…	devient		36	…	…
E	4	5	6					
	5	…	…					
	6	…	…					

Nous voyons que de façon naturelle, les états de l'automate déterministe obtenu depuis un automate non déterministe (nous appelons cette procédure « déterminisation ») sont des états composés : des ensembles constitués en états de l'AND d'origine.

Nous devons donc étudier ces ensembles.

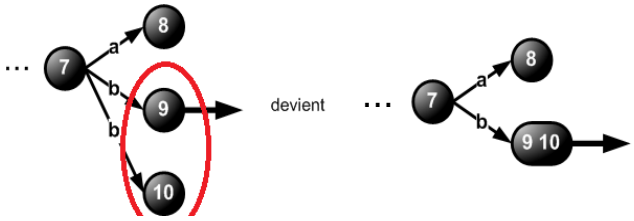
Soit l'AND possède n états, l'ensemble $Q=\{q_1, q_2, q_3, \ldots, q_n\}$. Les combinaisons de ces états (de 0 éléments, 1 élément, 2 éléments, …, n éléments) constituent **l'ensemble des parties de Q** noté Π. Par exemple, si $Q=\{1,2,3\}$, alors $\Pi=\{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$. Ici, il a été facile d'énumérer à la main toutes les combinaisons des éléments de Q. Pour un ensemble Q à trois éléments, nous avons obtenu l'ensemble Π de 8 éléments. Or, $8=2^3$. Est-ce que c'est une coïncidence ? La réponse est non : pout tout ensemble de *n* éléments, l'ensemble de parties est constitué de 2^n éléments.

Pourquoi ? Parce que le nombre d'éléments de Π, c'est le nombre de toutes les combinaisons de quelques éléments parmi n : $N = \sum_{k=0}^{n} C_n^k$.

Or, $\sum_{k=0}^{n} C_n^k = 2^n$. La preuve est très simple en utilisant le binôme de Newton:

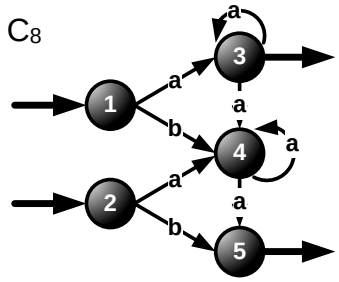
$$2^n = (1+1)^n = \sum_{k=0}^{n} C_n^k 1^{n-k} 1^k = \sum_{k=0}^{n} C_n^k .$$

Donc, en déterminisant un AND à n états, on peut obtenir un AD équivalent à au plus 2^n états. Chaque état de l'AD sera une combinaison d'états de l'AND, mais parmi eux peuvent figurer une « combinaison de 0 états parmi n » (notée par ∅ dans notre énumération) et des combinaisons de 1 état parmi n que nous avons notées {1},{2},{3},… et que, dans le futur, on notera simplement 1,2,3 etc.
Il nous reste à identifier le ou les états terminal (-aux). Un état terminal c'est l'état tel que tous les mots y aboutissant sont reconnus. Si, par exemple, l'état 7 de l'AND est terminal, cela veut dire que les mots y aboutissant sont reconnus. Tous ces mots aboutiront dans tous les états composés de l'AD qui contiennent 7 comme composante. Donc, il suffit que l'état composé de l'AD contiennent une sortie de l'AND pour qu'il soit terminal :



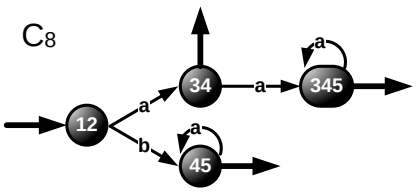
9 est un état terminal de l'AND, l'AD contient un état composé (9 10), donc cet état (9 10) est terminal.

Avant de donner une définition formelle, prenons un exemple.
Déterminisons l'AND



		a	b
E	1	3	4
E	2	4	5
S	3	3, 4	--
	4	4, 5	--
S	5	--	--

En suivant les règles évoquées tout à l'heure, nous obtenons l'état initial {1,2} que nous allons noter simplement 12, et de là nous construirons l'AD :



		a	b
E	12	34	45
S	34	345	--
S	45	45	--
S	345	345	--

la notation « 12 » veut dire {1,2} etc.

Explications : l'état composé cible de transition en, par exemple, a à partir de l'état 12, est composé des cibles de transitions en a de ses composantes. Dans l'AND, $1 \xrightarrow{a} 3$, $2 \xrightarrow{a} 4$, donc dans l'AD, $12 \xrightarrow{a} 34$. Chaque fois qu'on obtient un état de l'automate déterminisé dont on ne connait pas encore les

transitions, on le met dans la colonne de gauche et on suit la même recette. On procède ainsi tant qu'il y reste des états de l'AD non traités.
Les états terminaux sont ceux qui contiennent au moins un état final de l'automate initial (3 ou 5). Dans notre cas, ce sont les états 34, 45 et 345.

Définition formelle :

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.

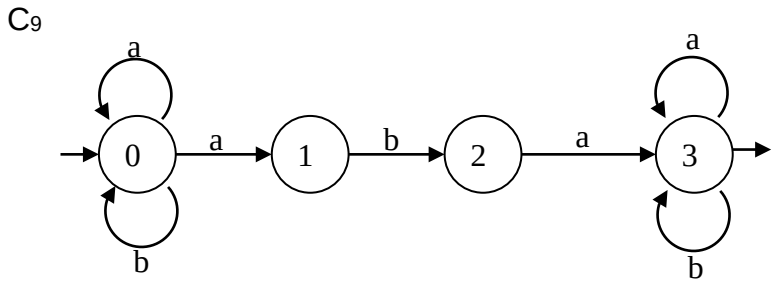
Un automate déterministe D correspondant à C prend

- comme ensemble d'états un sous-ensemble P de l'ensemble Π des parties de Q. (Donc au maximum 2^n états).
- comme unique état initial l'ensemble I des états initiaux de C.
- comme ensemble d'états terminaux l'ensemble $\zeta=\{u \in P \mid u \cap T \neq \varnothing\}$ 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 $(p.a.q) \in E$.

Un autre exemple :

Construisons un automate sur $A=\{a,b\}$ qui reconnaît l'ensemble des mots aba en facteur.

On commence par la construction d'un automate le plus simple (et qui sera non déterministe) reconnaissant tous les mots ayant aba en facteur :



Déterminisons-le :

automate non déterministe

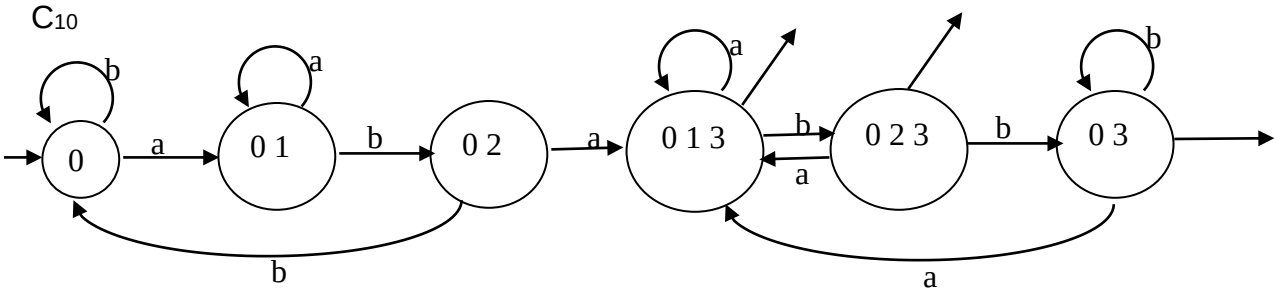
		a	b
E	0	0, 1	0
	1	--	2
	2	3	--
S	3	3	3

automate déterministe

		a	b
E	0	01	0
	01	01	02
	02	013	0
S	013	013	023
S	023	013	03
S	03	013	03

{0,1} etc. ; « 0 » veut dire {0}.

L'automate déterminisé est déjà complet, donc on n'y ajoute pas l'état poubelle. Le voilà :



Cet automate déterministe et complet reconnaît le langage $L_{C_{17}} = L_{C_{18}}$: l'ensemble de tous les mots ayant ***aba*** en facteur. Il a trois états terminaux : (013), (023) et (03).

Automates déterministes complets, complétion

Un automate déterministe peut être ou ne pas être **complet.**
Il est très facile de voir si l'AD est complet en examinant sa table de transitions. L'automate que nous venons de construire :

		a	b
E	12	34	45
S	34	345	--
S	45	45	--
S	345	345	--

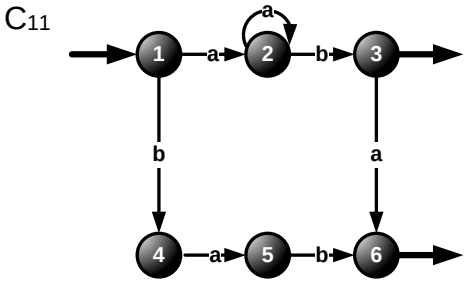
n'est pas complet : il lui manque des transitions. Il n'y a pas de transitions en b de l'état 34, de l'état 45 et de l'état 345. Un automate déterministe est complet si de chaque état sortent toutes les flèches avec toutes les étiquettes possibles.

Automate déterministe complet (définition) :

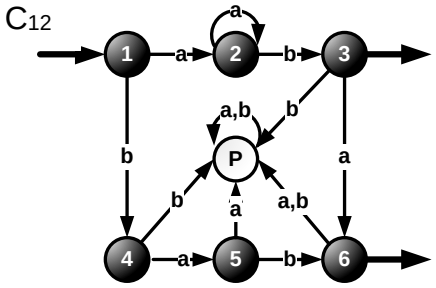
Pour $\forall \boldsymbol{u} \in \boldsymbol{P}$ et $\forall \boldsymbol{a} \in A$ $\exists \boldsymbol{v} \in \boldsymbol{P}$ tel que $\boldsymbol{u.a=v}$.
N'importe quel automate fini est équivalent à un automate déterministe complet, car n'importe quel automate fini est équivalent à un automate déterministe, et n'importe quel automate déterministe est équivalent à un automate déterministe complet : il suffit, dans l'automate déterminisé, d'introduire un « état poubelle » si l'automate tel quel n'est pas déjà complet. Voici comment on le fait :

Exemple de complétion:

AD non complet



AD complet (ADC)



Sous forme de table de transitions:

		a	b
E	1	2	4
	2	2	3
S	3	6	--
	4	5	--
	5	--	6
S	6	--	--

		a	b
E	1	2	4
	2	2	3
S	3	6	P
	4	5	P
	5	P	6
S	6	P	P
	P	P	P

Tous les traits (transitions absentes) sont remplacées par l'état P (état poubelle ou état puits), et une ligne est ajoutée à la table de transitions correspondant à une boucle sur P marquée par tous les caractères de l'alphabet. L'automate complété est équivalent à l'automate d'origine.

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. 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.

États et automates accessibles, coaccessibles :

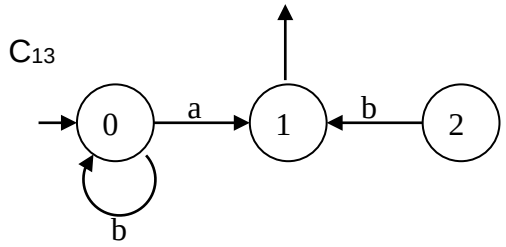
1. Un état est ***accessible*** s'il y a un chemin menant vers cet état depuis une entrée.
2. Un état est ***coaccessible*** s'il y a un chemin menant depuis cet état vers une sortie.
3. Un automate est ***accessible*** si tous les états sont accessibles.
4. Un ***automate*** est ***coaccessible*** si tous les états sont coaccessibles.
5. Accessible + coaccessible = ***émondé***.

Langages reconnaissables :

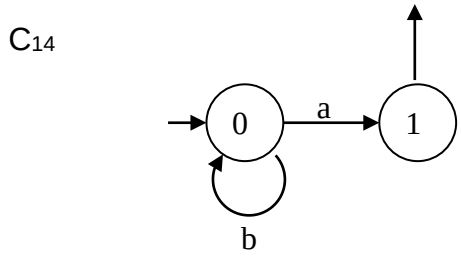
Un ensemble de mots ***X*** est un langage ***reconnaissable*** ssi il existe un automate fini ***D*** qui le reconnaît. **Ici, le mot « fini » est très important.**

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 :



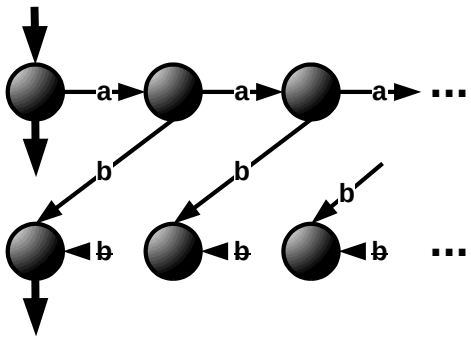
Néanmoins, la notion d'inaccessibilité n'est pas privée de sens. On peut obtenir des automates non accessibles lors des opérations dont on parlera plus bas, entre autres lors de l'obtention d'un automate reconnaissant le complément d'un langage. On peut aussi, des fois, obtenir un automate non accessible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

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 existe 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 :

C15



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***.

Le complément d'un langage reconnaissable

Le complément $\overline{L}$ d'un langage reconnaissable L ($\overline{L}$ est également appelé le langage complémentaire) est défini ainsi :

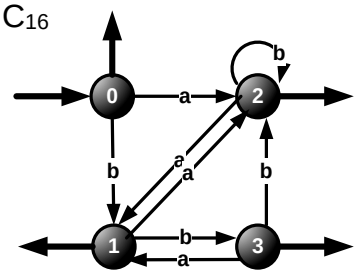
$\overline{L} = A^* \setminus L$ (ceci est équivalent à $\overline{L} \cap L = \varnothing$ **ET** $\overline{L} \cup L = A^*$)

C'est donc l'ensemble de mots de A* non appartenant à L. On peut dire « tous les mots sur l'alphabet A non appartenant à L », mais si on veut utiliser cette formulation, il faut alors préciser que si le mot vide appartient à L, il n'appartient pas à $\overline{L}$, et vice versa.

*Construction d'un automate fini reconnaissant le **complément d'un langage reconnaissable** donné*

On commence par considérer un ADC dont tous les états sont terminaux, par exemple :

C16



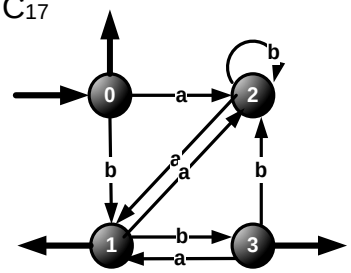
Quel langage reconnait-il ? La question semble compliquée. Elle ne l'est pas. Cet automate reconnait L=A*. Pourquoi ?

Cet automate est déterministe complet. Donc, aucun mot ne sera « coupé en vol », tout mot aboutit dans un état. Mais cet état est terminal, car tous les états sont terminaux, donc le mot est reconnu. Tous les mots sont reconnus, le langage de l'automate est A*.

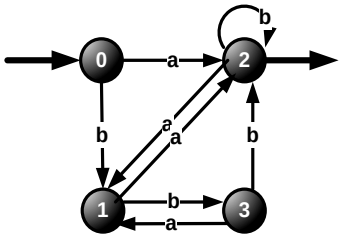
De là, on peut tirer des conséquences très intéressantes.

Prenons deux automates suivants :

C17



C18



Ils ont été obtenus à partir de l'automate C_{16}, en supprimant, dans l'automate C_{17}, la sortie sur l'état 2, en gardant toutes les autres, et dans l'automate C_{18}, les sorties sur les états 0,1 et 3, en gardant la sortie sur l'état 2.

L'automate C_{16} reconnait A*.

L'automate C_{17} reconnait tous les mots de C_{16} sauf ceux qui se terminent dans l'état 2.

L'automate C_{18} reconnait tous les mots de C_{16} sauf ceux qui se terminent dans les états 0, 1 et 3.

Donc $L_{C_{17}} \cup L_{C_{18}} = A^*$, et $L_{C_{17}} \cap L_8 = \varnothing$, ce qui veut dire que $L_{C_{17}} = \overline{L_{C_{18}}}$: les deux langages sont mutuellement complémentaires.

Nous voyons donc que si nous sommes en possession d'un automate déterministe complet reconnaissant un certain langage *L*, nous pouvons construire un autre automate déterministe complet reconnaissant le langage complémentaire à *L*. Pour le faire, il suffit de transformer tous les états terminaux en non terminaux et vice versa.

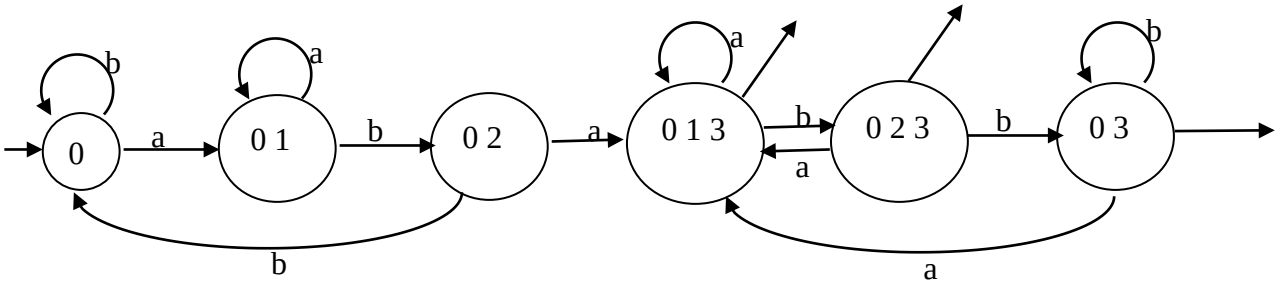
Attention ! Cette opération, qu'on appelle parfois « complémentarisation » (ne pas confondre avec complétion !), **ne peut être effectuée qu'à partir d'un ADC**. Si on a un automate qui n'est pas déterministe complet, il faut le déterminiser si besoin est et puis compléter si besoin est.

Un autre exemple :

Construisons un automate sur ***A***={***a,b***} qui reconnaît l'ensemble des mots n'ayant pas ***aba*** en facteur.

On a déjà construit un automate déterministe complet C_{10} reconnaissant tous les mots ayant ***aba*** en facteur :

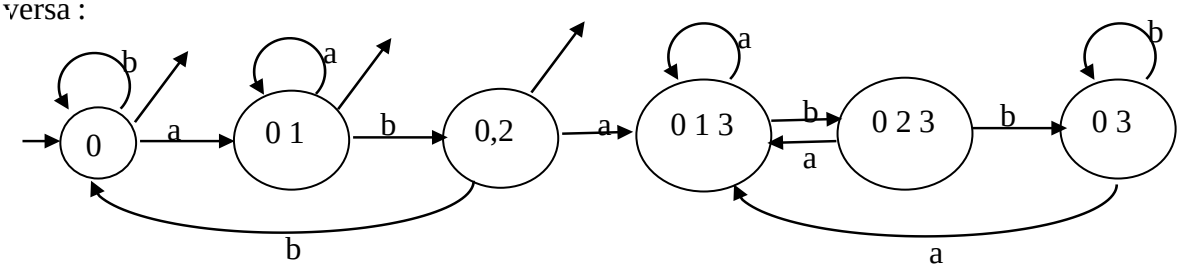
C_{10}



Maintenant on se rappelle que le but est de construire un automate qui reconnait $\overline{L_{C_{10}}}$, le langage complémentaire. Nous sommes prêts à le faire, car nous disposons d'un ADC reconnaissant $L_{C_{10}}$. 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 :

C_{19}



Rappelons encore une fois :
Ici, on a obtenu un automate complet sans qu'il y ait besoin de le compléter en introduisant l'état poubelle. Évidemment, 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
Dans l'automate C_{19}, par hasard, les états 013, 023, 03 sont inutiles. Ils peuvent être supprimés (alors l'automate ne sera plus complet) ou, mieux vaut, remplacés par un seul état poubelle.

Minimisation

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

(Evidemment, si cet automate contient un état poubelle, il lui correspond un automate déterministe non complet plus petit d'un état obtenu par l'élimination de la poubelle. Dans ce cours, nous aurons affaire systématiquement à des automates minimaux **complets**.)

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

Principe On doit d'abord s'assurer de deux choses :

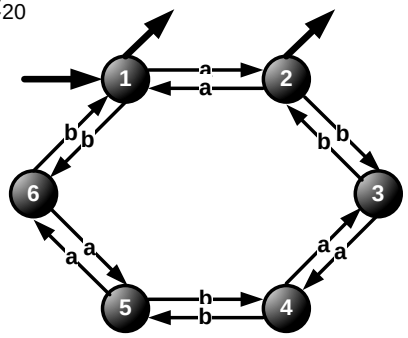
1) qu'on a un ADC. Donc si l'automate à minimiser n'est pas complet, il faut lui ajouter l'état poubelle pour qu'il le devienne.
2) Qu'il n'y a pas d'états non accessibles (s'il y en a, la méthode proposée ne fournira pas l'automate minimal, il y aura toujours des états non accessibles à la fin). Donc s'il y en a, on s'en débarrasse.

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 (motifs) des transitions en termes de groupes. On répète ce processus en utilisant la nouvelle partition obtenue, et on réitère jusqu'à ce qu'on arrive à ne plus pouvoir partitionner. Les groupes restants sont les états de l'automate minimal. On appelle souvent les itérations les étapes.

Nous commencerons par deux exemples, puis nous donnerons une description formelle utile pour pouvoir programmer une minimisation.

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

C20



E/S	a	b
1	2	6
2	1	3
3	4	2
4	3	5
5	6	4
6	5	1

1. Initialisation : nous séparons tous les états en états terminaux et non terminaux : Θ_0={T, NT} où T={1,2} et NT={3,4,5,6}. Θ_0 s'appelle *partition initiale.* T et NT *s'appelle le groupe terminal* et le *groupe non terminal*.

2. Nous réécrivons la table de transitions en séparant les états appartenant à T et à NT : et nous indiquons à quel groupe de Θ_0 appartient chaque état cible :

		a	b		sous Θ_0 a	b
T	1	2	6		T	NT
	2	1	3		T	NT
NT	3	4	2		NT	T
	4	3	5		NT	NT
	5	6	4		NT	NT
	6	5	1		NT	T

3 Itération 1 : On regarde, à l'intérieur de chaque groupe de Θ_0, quels *motifs* de transitions sont présents. Ici, on aurait pu avoir quatre motifs : « T T », « T NT », « NT T », « NT NT ». En réalité, à l'intérieur du groupe T, seul le motif « T NT » est présent, et à l'intérieur du groupe NT, il y a deux motifs : « NT T » et « NT NT ». J'ai noté les deux motifs différents dans le groupe NT par des couleurs.

Chaque motif, séparément dans chaque groupe, correspond à un sous-groupe, ce qui est la même chose que groupe de la partition suivante Θ_1. Ceci veut dire qu'à la fin de l'itération 1 nous obtenons la partition Θ_1 consistant en trois groupes : Θ_1= {(1,2), (3, 6), (4,5)}. Pour continuer, nous devons donner des noms à ces groupes, par exemple I, II, III ; mais en l'occurrence, il est plus facile de les appeler « 12 », « 36 », « 45 ».

4 Itération 2 : Nous réécrivons la table de transitions séparée maintenant en trois groupes, et nous faisons la même chose, mais en termes de la partition Θ_1.

				sous Θ_1		
		a	b	a	b	
Gr.12	1	2	6	12	36	aucune
	2	1	3	12	36	séparation
Gr.36	3	4	2	45	12	aucune
	6	5	1	45	12	séparation
Gr.45	4	3	5	36	45	aucune
	5	6	4	36	45	séparation

À l'intérieur de chaque groupe de Θ_1 un seul motif est présent : « 45 12 » pour le groupe 36, et « 36 45 » pour le groupe 45. Il ne se produit aucune séparation en sous-groupes. La partition suivante Θ_2 est identique à Θ_1 . C'est la **condition d'arrêt** : $\Theta_2=\Theta_1=\Theta_{fin}$.

5 Les groupes de la partition finale sont les états de l'automate minimal : il consiste donc en trois états. On identifie l'état initial et le ou les états terminal (-aux).

- L'état initial est le groupe comportant l'entrée de l'automate C_{19} ; c'est donc (12).
- Est terminal tout état consistant en états terminaux de l'automate C_{19} (on peut le formuler comme « est terminal tout état descendant du groupe T de la partition initiale »). (Il est claire par construction qu'aucun groupe de Θ_{fin} ne peut comporter et des états terminaux et des états non terminaux de l'automate C_{19}). Ici, donc, il y a un seul état terminal (12).

6 On doit maintenant construire l'automate minimal. Une façon simple de le faire est de prendre la table des transitions de l'automate C_{19} et remplacer chaque état par son groupe de Θ_{fin} :

	a	b
1	2	6
2	1	3
3	4	2
4	3	5
5	6	4
6	5	1

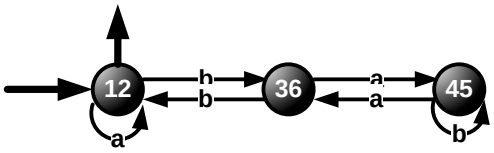
engendre

	a	b
12	12	36
12	12	36
36	45	12
45	36	45
45	36	45
36	45	12

Vu qu'il y a moins de groupes qu'il y avait d'état de C_{19}, il y a des lignes identiques. On les barre et on obtient l'automate minimal C_{20} :

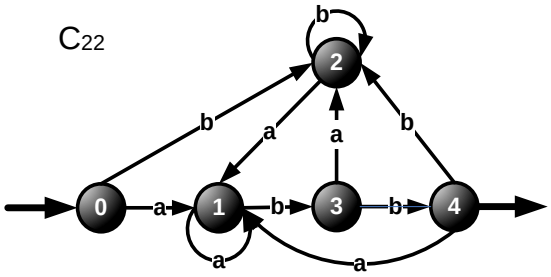
	a	b
E/S 12	12	36
36	45	12
45	36	45

C_{21}



Exemple 2 L'ADC à mininiser :

		a	b
E	0	1	2
	1	1	3
	2	1	2
	3	2	4
S	4	1	2



Ici, il y a un unique état terminal. Donc la partition initiale Θ_0={T, NT} où T={4} et NT={0,1,2,3}.
On ne peut pas, évidemment, essayer de séparer le groupe T qui consiste déjà en un seul état. Il n'y a aucun travail à faire concernant le groupe T. On travaille uniquement avec le groupe NT.

Itération 1 :

état	a	b	sous Θ_0	
0	1	2	NT	NT
1	1	3	NT	NT
2	1	2	NT	NT
3	2	4	NT	T

Le groupe NT s'est séparé en deux : (0,1,2) et (3). Θ_1={T,(0,1,2), (3)} ;
je peux appeler (0,1,2)=A et (3)=3. Pour plus de simplicité, je vais appeler le groupe T=4 : Θ_1={4,A,3}.

De nouveau, le groupe 3 de Θ_1 consiste en un seul état, et ne peut plus être séparé en sous-groupes. Nous travaillons avec le groupe A.

Itération 2 :

état	a	b	sous Θ_1	
0	1	2	A	A
1	1	3	A	3
2	1	2	A	A

Le groupe A s'est séparé en deux : (0,2) et (1). Θ_2={4,(0,2), (1),3} ;
je peux appeler (0,2)=02 et (1)=1.

De nouveau, le groupe 1 de Θ_2 consiste en un seul état, et ne peut plus être séparé en sous-groupes. Nous travaillons avec le groupe B.

Itération 2 :

état	a	b	sous Θ_2	
0	1	2	A	02
2	1	2	A	02

Aucune séparation ne s'est produite ; $\Theta_3=\Theta_2=\Theta_{fin}$. Fin d'itérations.

L'automate minimal consiste en états 02, 1, 3, 4. 02 est l'entrée car il contient 0. 4 est la sortie car c'est le seul descendant du groupe T.

En remplaçant dans la table de transitions d'origine les états par leurs groupes de Θ_{fin}, on obtient

	a	b
0	1	2
1	1	3
2	1	2
3	2	4
4	1	2

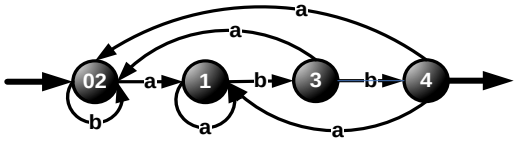
engendre

	a	b
02	1	02
1	1	3
02	1	02
3	2	4
4	1	2

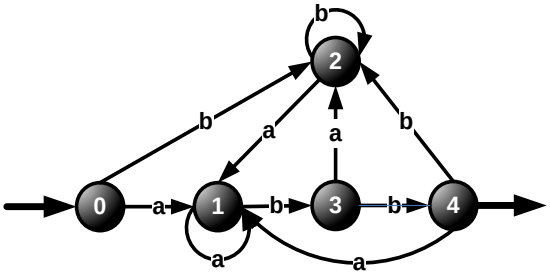
où la troisième ligne est de trop, car elle coïncide avec la première.

Voici donc l'automate minimal :

C_{23}



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 C_{22} :



et introduisons la terminologie suivante :
on dit que la chaîne ***w*** **<u>distingue</u>** 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 seul ***a*** ; puis, comme on est dans le même état (1), il nous restent que des 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.

C'est sur cette distinction, que nous n'allons pas détailler plus que cela, qu'est basé l'algorithme de minimisation : on recherche tous les groupes qui peuvent être distingués par une chaîne ou une autre.

Remarques :

1) on a vu que la position de l'entrée ne joue aucun rôle dans les itérations, ce n'est qu'une fois qu'on a obtenu le contenu de l'automate minimal en états, qu'on identifie l'entrée de celui-ci comme l'état contenant l'entrée de l'automate d'origine.
2) La présence éventuelle de l'état poubelle ne joue aucun rôle spécial non plus. C'est un état (non terminal) comme un autre. Dans la grande majorité des cas, suite à la minimisation l'état poubelle reste seul (formant un groupe de $=\Theta_{fin}$ en soi), mais ce n'est pas le cas s'il y a d'autres états non coaccessibles dans l'ADC d'origine.
3) L'automate qu'on minimise peut être déjà minimal. Il est difficile de le voir sans le minimiser. Si c'est le cas, tous les états de l'ADC d'origine vont séparer, il y aura autant de groupes de Θ_{fin} qu'il y avait d'états dans l'ADC d'origine. Dans ce cas, évidemment, il est inutile de « construire » l'automate minimal : si le nombre d'états n'a pas diminué, on dit que l'automate d'origine est déjà minimal.

Nous pouvons maintenant formaliser l'algorithme de minimisation.

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 consistant en deux groupes : les états terminaux ***T*** et les états non-terminaux ***NT=Q\T***.

2. Procédure applicable à la partition courante Θ_i, à commencer par i=0 :

début
pour chaque groupe ***G*** de la partition courante Θ_i **faire**
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 */

obtenir la nouvelle partition Θ_{i+1} en remplaçant chaque groupe ***G*** par des sous-groupes ainsi formés, qui seront les groupes de 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. **Si** le nombre de groupe de Θ_{final} est égal au nombre d'états de ***D*** **alors** ***D'*** = ***D***
sinon
Transformer la table de transitions d'origine en remplaçant chaque état de ***D*** par le groupe de Θ_{final} auquel il appartient : si dans ***D*** il y a une transition $e \xrightarrow{a} t$, $e \in G_1$ et $t \in G_2$ où $G_1 \subset \Theta_{final}$ et $G_2 \subset \Theta_{final}$, alors $G_1 \xrightarrow{a} G_2$ est une transition dans ***D'***.

5. Dans la liste de transitions ainsi obtenues il y aura des transitions identiques. Les identifier et supprimer des transitions qui ont déjà été obtenues.

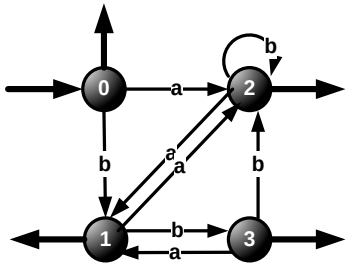
Pour tout groupe ***G*** de Θ_{final} , soit ***G*** est entièrement composé d'états de ***T***, soit ***G*** n'en contient aucun.

Remarque Il existe aussi une variante des étapes 4,5 dans laquelle on choisit dans ***G*** un état **représentant** de chaque groupe de Θ_{final} et en remplace tout occurrence d'autres états appartenant au même groupe ***G*** par ce représentant. En fait, cela remonte à la même chose.

Remarque importante

Pour minimiser, il faut commencer par séparer tous les états en terminaux et non terminaux. Et s'il n'y a d'états non terminaux ? Comme, par exemple, pour l'automate C_{16} que nous avons vu plus tôt ?

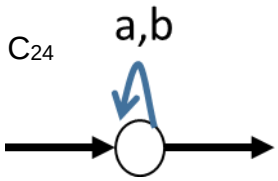
C_{16}



On peut répondre à cette question de deux façons, en obtenant la même réponse.

1) Si Q=T et NT=∅, alors $\Theta_0=\{T\}$ il n'y a aucun moyen pour séparer le groupe T en sous-groupes. T sera le seul état de l'automate minimise :

C_{24}



2) Ou bien on peut raisonner en se souvenant que le langage reconnu par un ADC dont tous les états sont terminaux, est A* (où, ici, A={a,b}). Or, il est évident que l'automate C_{24} reconnait A*. On ne peut pas mieux faire, cet automate est certainement minimal pour le langage L=A* !